CoSolve: A Novel System for Engaging Users in Collaborative Problem-Solving

Sandra B. Fan

A dissertation
submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy

University of Washington

2013

Reading Committee:

Steven L. Tanimoto, Chair

Alan Borning

Linda Shapiro

Program Authorized to Offer Degree:
Computer Science and Engineering

University of Washington

Abstract

CoSolve: A Novel System for Engaging Users in Collaborative Problem-Solving

Sandra B. Fan

Chair of the Supervisory Committee:
Professor Steven L. Tanimoto
Computer Science and Engineering

In an increasingly connected world, there are ever more opportunities for online collaboration. As the potential for more collaborators grows, so does the complexity of such communications. Problem solving, already a difficult task when performed by just one person, can grow more and more complicated. However, there is great potential for generation of more ideas and parallelization of work. How can we harness this potential while minimizing user frustration inherent in a large-scale project?

I present CoSolve, an online collaborative problem-solving environment that helps users engage in problem-posing and problem-solving. CoSolve uses the state-space model of problem-solving to present a tree-based visualization of a solution space to users. Users can then collaborate to generate, explore and annotate the state-space tree as they engage in the problem-solving process. CoSolve also allows users to formulate their own problems through a unique, scaffolded problem-posing interface.

I evaluated CoSolve by conducting user studies to analyze how users engage in online collaborative problem-solving. I also examined case studies of users posing their own problems on CoSolve to study patterns of problem-posing behavior. To test how a system such as Cosolve can encourage users' collaborative behavior, I built a collaborative user roles system and evaluated its performance in a user study. Finally, I conclude this dissertation with observations and recommendations for the design of tree-based, collaborative problem-solving systems.

TABLE OF CONTENTS

Page

List of Figures iv

List of Tables vii

Chapter 1: Introduction 1
1.1 CoSolve Overview 4
1.2 Research Contributions 10
1.2.1 Problem-Posing Framework 10
1.2.2 Problem-Posing Analysis 10
1.2.3 Collaborative Problem-Solving Framework 11
1.2.4 User Problem-Solving Behavior in State-Space-Based Systems 11
1.2.5 Collaborative User Roles System 12
1.2.6 Recommendations for Collaborative Problem-Solving Environments . 13
1.3 Dissertation Outline 13

Chapter 2: Background and Related Work 14
2.1 Automated Solvers 14
2.2 Problem-Solving Environments 15
2.3 Problem Solving through Crowdsourcing 16
2.3.1 T-Star 17
2.4 Computer-Supported Collaborative Work Systems 18
2.5 Online Collaborative Problem-Solving and Design Projects 20
2.5.1 Polymath Project: Collaborative Mathematics 21
2.5.2 Collaborative Artifact Production: Wikipedia, Open Source Software Projects, Flash "collabs" 25

Chapter 3: Problem Posing in CoSolve 33
3.1 Description of the Problem-Posing Process in CoSolve 34
3.1.1 Creating Problem Templates 35
3.1.2 Example Problem Template: Towers of Hanoi 41

3.2 Case Studies and Analysis of Problem-Posing 54
3.2.1 Student A: "Knight's Tour" 59
3.2.2 Student B: "UW Classrooms" 62
3.2.3 Student C: "Opportunity Ladder" 65
3.2.4 Summary 69
3.3 Discussion of the Scope of Problem Posing in CoSolve 73
3.3.1 Transforming real-world problems into CoSolve problem templates . . 77

Chapter 4: Problem Solving in CoSolve 80
4.1 Description of the Problem-Solving Process in CoSolve 81
4.1.1 State-Space Search and CoSolve's Solving Process 81
4.1.2 CoSolve's Solving Session User Interface 88
4.2 Evaluation: CitySim User Study 99
4.2.1 CitySim Problem Description 100
4.2.2 The CoSolve Consultant 101
4.2.3 Study Design and Procedures 104
4.2.4 Study Results 108
4.2.5 User Interface Evaluation 124
4.2.6 Discussion 129
4.3 Using CoSolve for Design Problems 131
4.3.1 Go Atom 133
4.3.2 Eco-avelli 136
4.3.3 State Space Design 138

Chapter 5: Collaboration Roles in CoSolve 141
5.1 Roles User Study 144
5.1.1 Roles User Interface Design and Implementation 146
5.1.2 Study Procedure 154
5.1.3 Overall Results 156
5.1.4 Discussion 175

Chapter 6: Design and Implementation of CoSolve 184
6.1 Overview of CoSolve's System Architecture 184
6.2 TStar-C Implementation Details 187
6.3 CoSolve Web Services API 192
6.4 Design Rationale and Trade-offs 194

Chapter 7: Conclusions and Future Work . . . 199
7.1 Problem Posing . . . 201
7.1.1 Summary . . . 201
7.1.2 Future Work . . . 202
7.2 Problem Solving . . . 204
7.2.1 Summary . . . 204
7.2.2 Future Work . . . 205
7.3 Collaboration and Communication . . . 206
7.3.1 Summary . . . 206
7.3.2 Future Work . . . 206
7.4 Conclusion . . . 208

Bibliography . . . 210

Appendix A: Code Listing: "Towers of Hanoi" CoSolve Problem Template . . . 216
A.1 Problem Description . . . 216
A.2 Problem Template Code . . . 216
A.3 Operator Code . . . 218

Appendix B: CitySim User Study: Background Questionnaire . . . 221

Appendix C: CitySim User Study: Pretest/Post-test . . . 223

Appendix D: CitySim User Study: Tutorial Script . . . 225

Appendix E: CitySim User Study: Wrap-up Questionnaire . . . 235

Appendix F: Roles User Study: Wrap-up Questionnaire . . . 239

Appendix G: CitySim User Study: Interview Questions . . . 243

LIST OF FIGURES

Figure Number — Page

1.1 CoSolve full user interface. . . . 3
1.2 Example of a partial CoSolve state-space tree for the Towers of Hanoi problem. 5
1.3 Applying an Operator 6
1.4 Close-up of a CitySim solving session node, with the "Add Annotation" menu displayed. . . . 7
3.1 Solving session with root node created from code in Listing 3.1 43
3.2 Partial screenshot of template web form after adding the code from Listing 3.1, and turning on the "Using Python Code for Visualization" toggle. Note that the Visualization code area is still in its default state. . . . 45
3.3 Root node of a Towers of Hanoi solving session after saving the code from Listing 3.2 45
3.4 Solving session with visualization of root node created from code in Listing 3.3 47
3.5 Completed visualization of Towers of Hanoi 49
3.6 Parameter Specifications for operator "Move disk" 53
3.7 Student A's "Knight's Tour" - Partial screenshot of a solving session 59
3.8 Student B's "UW Classrooms" - Screenshot of a node in a solving session . . 63
3.9 Student C's "Opportunity Ladder" - Screenshot of an answer status state node in a solving session 71
3.10 Student C's "Opportunity Ladder" - Screenshot of a question state node in a solving session. . . . 72
4.1 Three example possible states of the 15-Puzzle. . . . 82
4.2 Example solving session tree for Minesweeper problem 83
4.3 A CoSolve solving session tree for Towers of Hanoi. Nodes are labeled. Number indicates level, letter indicates position from left to right. The initial state is at the root, node 1a. Two goal states were reached, at nodes 9a and 8b. Nodes 3a, 4a and 3d are non-goal leaf nodes, indicating where solvers abandoned a solution path as unpromising, and decided to explore elsewhere first. . . . 87

4.4 Example of a simple version of the Solving Session creation web form, with some limited options shown. Depending on user permissions, some session creators have more options for solving sessions. 89
4.5 New Towers of Hanoi Solving Session in the Flash client UI. 89
4.6 Solving Session Flash Client - State Views menu. 91
4.7 Solving Session Flash Client - Applying a filter. 91
4.8 Hovering the cursor over any node in the solving session tree will bring up the node "visual options" menu buttons above the node, and the node operations menu or "node menu" below the node. 92
4.9 Options to highlight a node in a solving session. 93
4.10 Applying an operator to the root node in a Towers of Hanoi Solving Session . 95
4.11 CitySim closeup 95
4.12 The Annotation Form submenu of a node. 96
4.13 Annotations in a solving session. 98
4.14 CitySim Solving Session User Interface, with Full CoSolve Consultant (FC) . 102
4.15 CoSolve Consultant information windows 105
4.16 Minimal Consultant interface. 106
4.17 Team 1 - Solving-Session Tree 114
4.18 Team 2 - Solving-Session Tree 114
4.19 Team 3 - Solving-Session Tree 115
4.20 Team 4 - Solving-Session Tree 115
4.21 Team 5 - Solving-Session Tree 116
4.22 Team 6 - Solving-Session Tree 116
4.23 Example of a Go Atom game in progress. 135

5.1 Roles UI within a Solving Session 147
5.2 Roles UI - Current Role tab 151
5.3 Overall In-Role tab in Roles UI 152
5.4 Role change alert. 155
5.5 Team 7 - Solving-Session Tree. 160
5.6 Team 8 - Solving-Session Tree. 161
5.7 Team 9 - Solving-Session Tree. 162
5.8 Participation metrics: solving actions by user, by team. These plots show nodes created, annotations created, and annotation types created. 166

5.9 Equitable participation metrics. These plots show how many nodes each user created on the solution path, and the post-activity percentage (proportion) of nodes or annotations each user estimated that they created out of the total their team created. . . . 168
5.10 Percentage contributions: subjects' guesses versus actual percentage in the Roles and Minimal Consultant conditions. . . . 168
5.11 Roles study CitySim high scores, by user. . . . 169
5.12 Change in attitude variables before and after activity. One subject from Team 8 and one from Team 9 were dropped, as they replied "n/a" to some of the questions. . . . 171
5.13 Change in attitude variables before and after activity, cont. . . . 172
5.14 Self-reported change in attitude after activity. . . . 173
5.15 Composite attitude score before and after activity, for roles and control team subjects. . . . 174
5.16 Roles subjects rated the Roles UI, control subjects rated the Minimal Consultant. Answers are on a Likert scale; 1 is Strongly Disagree, 5 is Strongly Agree. . . . 176
6.1 CoSolve System Architecture . . . 186

LIST OF TABLES

Table Number Page

3.1 Description of CSE 415 student CoSolve projects. 56
3.2 Student A's "Knight's Tour" Problem Template Data 61
3.3 Student B's "UW Classroom" Problem Template Data 64
3.4 Student C's "Opportunity Ladder" Problem Template Data 68

4.1 Overall CitySim user study team results. 109
4.2 Subjects' Likert-scale responses regarding the CoSolve Consultant. 125
4.3 Subjects' post-activity questionnaire responses, as percentages of total number of responses (n=18) 127

5.1 Role phase schedule. 149
5.2 Completed role phases per user. 157
5.3 In-role actions summary for roles condition teams. 158
5.4 Average time, in minutes, that users took to perform goal number of in-role actions per phase, for all phases where goal was met. (Phases during which a user did not meet the goal are not included here.) 159
5.5 Overall performance data for roles and control (MC) conditions. The "guess" row data are users' guesses at the percentage of node or annotations out of the team's total that each user thought he created after the activity finished. "Solving action" count is the sum of nodes created and annotations created. . 164
5.6 This table shows the number of times each user viewed the Current Role bar dialog box, the Overall Role bar dialog box, clicked on each of the tabs, and clicked to acknowledge role changes. 175

6.1 List of fields that comprise a TStar node. 189

ACKNOWLEDGMENTS

First and foremost, my deepest gratitude to my advisor, Steven Tanimoto, for patiently guiding me through the process of becoming a PhD.

Many thanks to the National Science Foundation and the Washington NASA Space Grant Consortium for their financial support.

I would like to acknowledge members of the Online Learning Environments group: Robert Thompson, Laura Dong, Chris Brenan, Yizhou Wang, and Christopher Clark who all contributed to implementing CoSolve. Thank you to Michael Duong, Yifan Zhang, Richard Rice, Galen Knapp, and Katherine Hulsman for their work on problem-posing in CoSolve. Thanks most especially to Tyler Robison, who conducted user studies with me, implemented the CoSolve Consultant, and provided much moral support throughout our years in graduate school.

I am grateful to my many user study participants, who came and sat through hours in the lab, patiently enduring absent teammates and user interface bugs. Thank you also to the CSE 415 students who chose to implement CoSolve projects. I would especially like to acknowledge Jordan Atwood, Charliz Burks, and Cezanne Camacho for letting me use their excellent problem template projects in this dissertation.

A huge thank you to the morning meeting group: Katherine Velas, Fay Shaw, Nicole Nichols, Julie Medero, and Karen Studarus. You kept me focused and kept me going, and gave me a reason to get to campus before lunchtime. I would never have finished this without you.

Finally, thank you to Craig Prince, for everything.

DEDICATION

To my parents, Jordan Fan and Denise Fan.

Chapter 1

INTRODUCTION

Humans have engaged in problem solving for thousands of years, for everything from taming fire to constructing skyscrapers. With today's ever-increasing computational power and connectivity, we have the opportunity to enhance this venerable activity by using computers to solve problems faster and better than ever before. The Internet now allows people in different parts of the world to collaborate to solve problems, presenting more viewpoints and generating a greater diversity of ideas. However, such proliferation of both computer-generated and human-generated information could be difficult for humans to process without some structure. Additionally, as collaborative activity increasingly moves online, individuals with different backgrounds must adjust to communication without the help of physical social cues.

As an examination of these issues, we present the design and evaluation of CoSolve, an online system for collaborative problem-solving. CoSolve brings the power and logic of computing together with the aesthetics and creativity of human intuition into an environment for users to engage with each other in exploring solutions. CoSolve provides users with a unique, interactive, tree-based visualization of the problem space, which allows them to see the history of their problem-solving process, a focal point for communication, and also a different way of thinking about the problem-solving process itself. Additionally, while we explore problem solving as one specific area that can be enhanced by such a problem-space visualization system, CoSolve can also be used for design activity, in which coming up with a good design could be seen as an instance of problem solving, an idea from Herbert Simon's *Sciences of the Artificial* [1].

In CoSolve's user interface (Figure 1.1), a problem's space of possible solutions, or possible designs, is represented as a state-space search tree, a model taken from the field of classical artificial intelligence. For a given problem, each node in a CoSolve search tree

represents some state of the problem in the solving process. The root of this tree represents the initial state of the unsolved problem. The user then searches through the tree by applying transformations, or *operators*, to each state to generate new possible solution paths. A problem is solved when a state is generated such that it satisfies some goal criteria. In CoSolve, instead of merely internally using the tree as a data structure within the computer, we explicitly show these states to the users, and the users actively manipulate and interact with the tree, and can use the tree as a reference point in communicating with their teammates.

We have successfully used CoSolve in a number of different contexts and problem types. To name a few, CoSolve has been used for diverse problem types from simulations–for example, modeling optimal energy usage in a residential neighborhood–to game design–e.g. creating educational computer games. CoSolve can be used to present mathematics problems to students, or to model constraint-satisfaction problems such as work scheduling. CoSolve itself can also be used as a platform for a different type of gaming, one in which game moves can be taken back and explored in a non-linear fashion. CoSolve has also been used as a component in undergraduate artificial intelligence classes, in which students learn to pose and solve problems of their choosing in CoSolve, as a way to teach both state-space search and the problem-posing process.

This dissertation will provide a description of CoSolve, its development and implementation, an analysis and evaluation of CoSolve's problem-posing and problem-solving processes through case studies and formal user studies, a summary of lessons learned about the nature of online collaborative problem-solving environments, and recommendations for the design of such systems. We will begin with an overview of the CoSolve system.

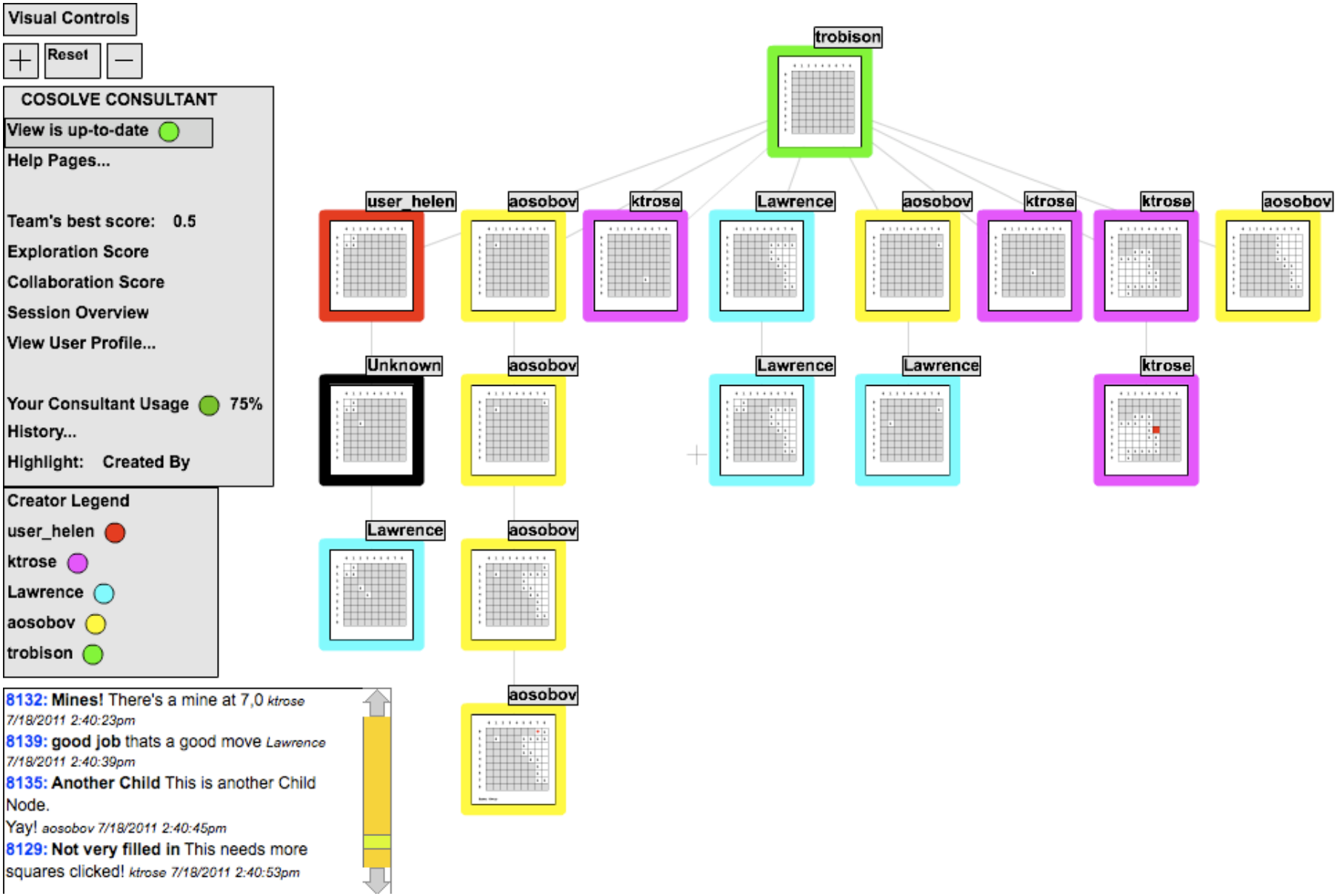


Figure 1.1: A screenshot of the CoSolve user interface. The solving session tree is on the right, labeled with the username of the user who created each node. On the left are user controls and information. On the bottom left is the annotation list, similar to a chat box. This solving session is an instantiation of the Minesweeper problem template.

1.1 CoSolve Overview

In the following subsections[1], we will briefly introduce CoSolve by providing a summary of its functionality. Chapters 3 and 4 will cover CoSolve problem-posing and CoSolve problem-solving processes in more detail.

Problem Solving in CoSolve

In CoSolve, a user can perform the role of a *problem poser*, a *problem solver*, or both. A problem poser specifies problems by creating problem templates, and a problem solver initiates and participates in CoSolve solving sessions for a particular problem template.

In a solving session, a team of solvers generates a state-space search tree that explores the solution space for a problem, typically with the goal of finding a state in the tree that represents an acceptable solution. Figure 1.2 shows an example of a partial solving session tree for the Towers of Hanoi problem. Towers of Hanoi is a simple puzzle with three wooden pegs, or posts, and three differently-sized disks with holes in the middle. The puzzle starts with all three disks stacked, in order of ascending size from top to bottom, on the far left post. The goal is to move all the disks on this left post onto the far right post, moving only one disk onto a post at a time, and never placing a larger disk on top of a smaller disk.

To solve this problem in CoSolve, first, a member of a team of solvers uses the web interface to initialize a new solving session from the Towers of Hanoi problem template in CoSolve. The team is then presented with a single node, the root node, at the top of the tree. This node represents the initial state of the puzzle–all the disks on the first peg. Then any solver on the team clicks on the node to select and apply an operator to transform this state into another. For example, the solver might select the operator "Move the top disk on the first post to the third post" (Figure 1.3). When the solver does so, CoSolve generates the next state, showing the disk moved to the right, and displays it to the user as a child of the root node, with a line connecting the two. From there, solvers can either continue to create children of the most recently created nodes, or they can backtrack to create new

[1]We gratefully acknowledge the IEEE Computer Society for allowing us to reprint the contents of this section, originally published in [2].

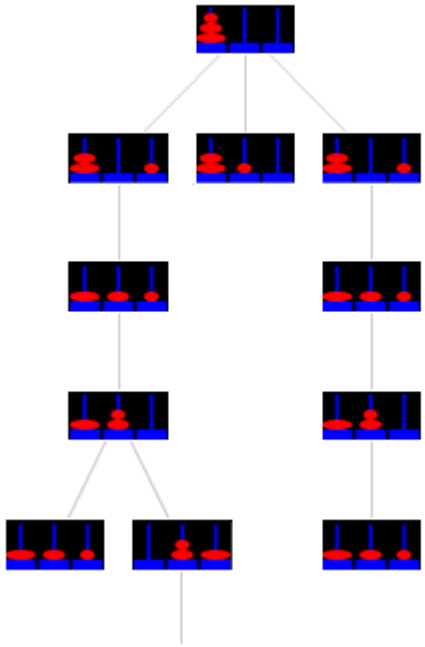


Figure 1.2: Example of a partial CoSolve state-space tree for the Towers of Hanoi problem.

branches. Solvers can view all branches of nodes and can apply operators to nodes they created or nodes other solvers have created. The goal is to eventually find a state or path to a state that satisfies the criteria to solve the problem.

Figure 1.1 depicts a solving session for the Minesweeper problem. The tree is displayed in the main area of the viewport, and a variety of user controls are available on the left. In the upper left corner are controls for zooming, switching to different tree layouts, and switching between different views of the tree and its nodes.

Figure 1.4 shows a close-up of a node in a solving session tree for the CitySim problem (to be described in Section 4.2.1). Underneath the node representation is a menu of node operations. Solvers select an operator to apply to the state, highlight the node for reference, or textually annotate the node as a form of communication with other solvers. There are three types of annotations, as seen in the figure: "positive" (thumbs-up), "neutral" (thumbs-sideways), and "negative" (thumbs-down). Solvers can use these to indicate whether they

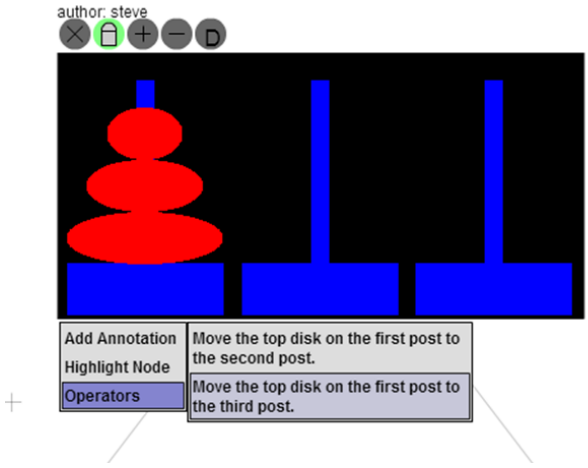


Figure 1.3: Here we see a single node, the root node, in a Towers of Hanoi solving session. When the solver hovers the cursor over the node, the node operations menu appears under the node, seen on the bottom left, with several options for actions to be performed on a node. Clicking on "Operators" gives the operator menu, with a list of two possible operators to apply. Clicking the "Move the top disk on the first post to the third post" operator will apply that operator immediately, and create a new state underneath that with the smallest disk moved to the third post.

believe the node to be on the path to a good solution or not. All annotations created anywhere in the tree are immediately displayed in the Annotations List, a box in the lower-left corner of the user interface (Figure 1.1). Clicking on an annotation in the Annotations List will pan and zoom the users tree view to display the associated node.

Problem Posing in CoSolve

We have just seen examples of how the solving process works in the CoSolve's tree-based solving interface. But perhaps even more challenging than solving a problem is posing the problem in the first place. Problem formulation is a crucial initial step, but can be challenging to support in software. Not only is it difficult for users to know a priori what tools they will need to solve a problem, it is also difficult for many users to formally represent

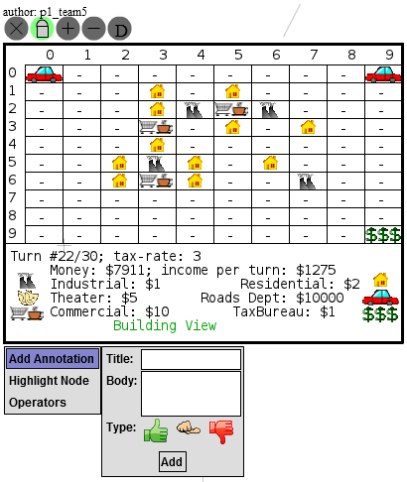


Figure 1.4: Close-up of a CitySim solving session node, with the "Add Annotation" menu displayed. Solvers can attach textual comments, or "annotations" to nodes.

the problem in a way that a computer can understand.

In CoSolve, we address this problem with our scaffolded problem-posing framework. Problem posers (*posers*) create a CoSolve problem template containing a programmatic representation of the problem to be solved. CoSolve provides a web interface for problem posers to create custom problem templates for problems of their choosing, which can then be used by solvers in the process described above. The problem-posing interface consists of a web form where posers enter scaffolded Python code fragments called *poser snippets*. By "scaffolded," we mean that we present posers with bits of "starter code" and "fill-in-the-blank" areas to assist with the posing process. CoSolve combines these poser snippets and applies them as needed during the subsequent solving activities.

A poser must specify two parts of a problem: (1) the state representation, or structure, of a problem, and (2) a set of operators for transforming each state into a resulting new child state. To specify the state structure to be used in a problem, posers enter key-value pairs into a special Python dictionary state variable that CoSolve provides. The keys represent a state's fields. The keys for Towers of Hanoi might be peg1, peg2, and peg3, and the value for each would be a list representing which disks are on that peg. If our

disks are numbered 1, 2, and 3, then the initial state for the Towers of Hanoi problem might be the following set of three key-value pairs, with the three disks all on the first peg: `{"peg1":[1,2,3],"peg2":[],"peg3":[]}`.

Optionally, when specifying the state variable, the poser may also provide a visualization method for the problems states using scaffolded Python code; otherwise, a default representation outputting the key-value pairs of a state will be displayed to the user.

Next, to create operators, posers specify two things: the precondition code and the state transformation code. The state transformation code is Python code that alters the values in the current state. For instance, if the solver applies the "move smallest disk to far right peg" operator on the initial state, the Python code that the poser wrote for this operator would change the values of the first-peg and third-peg keys appropriately, and the resulting state would be represented as: `{"peg1":[2,3], "peg2":[], "peg3":[1]}`. The precondition code checks whether the operator can validly be applied to a state; for instance, if an operator causes the solver to place a larger disk on top of a smaller disk, this would cause a precondition to fail and so, CoSolve will not display the operator to the solver in the first place, and no branch can be created from that operator. So, CoSolve operator preconditions, in some sense, provide a way to prune the state-space search tree.

In part because problem posing does involve some minimal coding, we decided to separate it from problem solving, so that it would be easier for non-programmers to solve problems with CoSolve. At the same time, posers with different levels of technical background have been able to create problem templates. For example, the CoSolve posing interface has been successfully used in undergraduate classes, both for computer science majors and non-majors, to create a range of problem template projects. These projects are discussed in more detail in Section 3.2.

Problem Templates in CoSolve

As previously mentioned, CoSolve can be used to solve a variety of different classes of problems. It can be used for any well-defined problems with explicit, discrete moves that are applied in sequence, and that have clearly specified winning criteria, for example, Towers of

Hanoi or chess. When we used CoSolve in class projects for undergraduate computer science courses, students have created problem templates for this class of problems, e.g., problem templates for sudoku, checkers, poker, stratego and other games, or puzzles like pentominoes or the 15-puzzle. Students have also created tools in CoSolve such as a class scheduler or transistor schematics diagramming tool. Other such templates created by undergraduate researchers in our lab have included minesweeper (shown in Figure 1.1), a simple fantasy role-playing game, and two ecological/environmental simulation problems. These types of problem templates will be discussed in more detail in Chapter 3.

We have also explored CoSolve problem templates in a mathematics education context. A plane geometry construction template[1] allows students to apply operators to draw a circle, draw a line, etc. to solve geometry problems. Another template allows students to visually explore functional composition.[1] When used for education, CoSolve allows a teacher to see all the paths that students have explored and inspect their work at any step, allowing discovery of misconceptions and assessment of each students contribution.

CoSolve can also be used in a very different manner: for creative problems that require subjective, human evaluation, such as design problems. For example, if a team is designing a playground, CoSolve states can be used to represent alternative layouts of the playground, and examples of problem operators might be "place a jungle gym in the northwest corner of the playground" and "use sand under the swing set." Another example is the Color Swatch[2] problem template. A website designer and his client could use CoSolve to explore color palette possibilities. Each state is a palette of colors, and solvers can use operators to change RGB and HSV values of individual colors, or the whole set (e.g. "select complementary colors"). Users can explore, and the entire history of their exploration is saved. Here, there are no clearly specified goal criteria. Instead, CoSolve becomes a tool to aid discussion, brainstorming and consensus. We have also used CoSolve to aid in designing educational card games, which will be described in Chapter 4. Each state is a game description, and designers apply operators to change game rules, for example, to change the number of cards

---

[1] Both of these templates were developed by Richard Rice.

[2] `http://cosolve.cs.washington.edu/problem/color-swatch`

allowed in a starting hand. Used in this manner, CoSolve becomes a way for human solvers to explore the space of possibilities in a visual way, and it provides a record that facilitates revisiting the many different ideas they may have developed.

1.2 Research Contributions

Now that we have introduced how CoSolve works, we will list the research contributions resulting from our work on CoSolve.

1.2.1 Design and implementation of a novel framework for problem-posing

How can we design a problem-posing system that will be powerful enough to model a variety of different problems, but understandable enough for a diverse range of users? What level of abstraction versus detail is necessary?

In CoSolve, we have developed a scaffolded problem-posing framework for users to provide a formal specification for a problem. This framework allows novice programmers to pose problems in CoSolve by modifying existing *poser snippets* into our system, while giving advanced programmers the full power of the Python programming language. Our web-based posing system abstracts away the details of creating a state-space tree, but allows flexibility for problem posers to create complex problem representations and transformations. The CoSolve system is then able to take these snippets and compose them, during the solving process, to generate parts of the state-space search tree on demand. Users are able to continually refine problem specification even after the solving process has begun.

1.2.2 Analysis of user problem-posing in actual practice

Over a hundred problem templates have been created in CoSolve so far. Twenty-one of these were created as class projects in an undergraduate computer science course. We analyze these and discuss different, emergent ways in which users have made use of CoSolve affordances to model problems of their own choosing. We identified different working styles in CoSolve's posing interface, and found several different methods of structuring problem posing code and creating problem state representations.

Additionally, we examined the use of CoSolve in an educational game design context, where designers iterated between problem posing and problem solving in a consensus-based design project, and identified the issues for creating flexible problem templates with the ability to handle this type of collaborative problem-solving.

1.2.3 Design and implementation of a novel framework for collaborative problem-solving

The problem-solving process is the most substantial part of CoSolve, intended for all levels of users to engage in, and hence, needs to be easy to use. At the same time, CoSolve's solving process needed to be sophisticated enough to represent complex problems. CoSolve's solving interfaces have been designed with this in mind.

Anyone with access to the web can solve problems in CoSolve, without any need for programming knowledge. During solving, CoSolve provides users with a visualization of the state-space tree, which gives users a tangible way to explore the solution space of a problem. Users can then, through a point-and-click menu interface, generate possible solutions by applying operators and evaluate them by annotating states. They can also collaborate with other solvers by communicating via the annotations.

1.2.4 Evaluation of state-space search as a model for collaborative problem-solving

Although problem-solving in CoSolve is meant to be easily accessible to users of any level of computer expertise, using state-space search trees as a model for problem-solving might still be too complex for non-technical users to understand. Or perhaps users can understand it, but find it difficult or confusing to use in practice. Secondly, even if users are able to understand and solve problems with a CoSolve-style system, there is still the question of how such a system affects their problem-solving and collaborative behavior. How do users actually behave when problem-solving is presented to them as an exploration of branching options?

We examined these issues by conducting a user study with teams of solvers working together to solve a problem in CoSolve. We found that users were able to successfully use CoSolve for problem-solving, and overall they felt that CoSolve's tree visualization was

easy to understand and helpful in solving the problem. We also analyzed user behavior, conducted user interviews, observed several strategies employed by solvers in using tree-specific structures for collaborative problem-solving, and identified different user styles of tree-based solving. Finally, we collected feedback on further user interface features that would be helpful in a CoSolve-style system.

1.2.5 Design and exploration of a user role system for encouraging equitable participation in collaborative problem-solving

CoSolve was designed for collaboration, but in actual usage, we have seen large disparities in collaborative activity, depending on users' personal solving styles. Not all users participated or interacted equally in the problem-solving process. Some users may feel unsure of themselves, or feel that others on the team are better at the task than they are, and so they maintain their distance from their teammates so as not to hinder the team. Others feel overly sure of themselves and ignore their teammates, remarking that looking at their teammates ideas would simply be a waste of time. However, in most of the solving session outcomes we saw, successful final solution paths were usually a combination of different user's branches and ideas.

To encourage collaboration, we developed a set of user participation roles to guide solvers' behavior and help them feel more comfortable with participating in the solving process. The CoSolve Roles System assigns one of these roles–Brainstormer, Supporter, or Critic–to the participants, and periodically changes a participant's roles, to give all team members an equal chance to participate in and understand each of the different processes involved in problem-solving. The Roles System interface also keeps track of user progress in their respective roles and presents this information to the team as a way of promoting accountability within the team. We then tested these roles in a user study, and studied how users reacted to it. While we found that the differences in users' task performance with and without roles was not statistically significant due to small sample size, in this dissertation we provide a qualitative analysis of how users are able to problem-solve within a rigid role structure, and we identify a set of collaboration behaviors that can inform future work on role-based

collaborative problem-solving.

1.2.6 Recommendations for the design of online collaborative problem-solving environments

Based on our work on CoSolve and our study of its actual usage by problem-solvers and problem-posers, we have compiled a set of recommendations for others designing online collaborative problem-solving environments. These recommendations are grouped into three categories. The first set of recommendations are for enhancing problem-posing functionality. The second set of recommendations are user-interface recommendations for problem-solving environments, including user interface features for tree-based, problem-solving navigation. Finally, we suggest a set of design recommendations for facilitating collaboration in online problem-solving environments.

1.3 Dissertation Outline

This dissertation is organized as follows. In the next chapter, Chapter 2, we provide a discussion of related work in the fields of problem-solving environments and online, computer-supported collaborative work, to provide context and ground our work on CoSolve and on the CoSolve Roles System. We will also talk about the T-Star project, which is a direct predecessor to CoSolve. Chapter 3 describes the problem-posing process in more detail, from the user's point of view, and we present case studies of problems posed in CoSolve. Problem solving is discussed in Chapter 4; this chapter also presents findings from a CoSolve problem-solving user study, and from a study on using CoSolve for educational-game design. In Chapter 5, we present the CoSolve Roles System, and our observations from the results of our roles user study. In Chapter 6, we describe the design of CoSolve, and relevant design decisions and implementation details. Finally, Chapter 7 contains direction for future work and recommendations for the design of collaborative problem-solving environments.

Chapter 2

BACKGROUND AND RELATED WORK

To ground our work in online collaborative problem-solving environments, we will now review the relevant literature on state-space search for problem-solving, other problem-solving systems, groupwork and collaborative systems. We will also look at other online collaborative projects to see how people currently collaborate on the web.

2.1 Automated Solvers

CoSolve's main premise is derived from Herbert Simon's *Sciences of the Artificial*, which argues that complex human problem-solving can be approached as an instance of state-space search [1]. A human problem-solver is searching through a space of alternative states to find one that "satisfices" some criteria, and this is the state that represents the solution. In line with the idea of state-space search for problem-solving, Newell, Shaw and Simon implemented the General Problem Solver (GPS) in 1959, with the goal of creating a universal problem-solving computer program that could solve any formalized, symbolic problem [3]. However, GPS had difficulty with problems which required new or outside knowledge, or which were complex enough that the state-space was too large for the search algorithm to process. GPS eventually evolved into Newell and Laird's Soar (State, Operator And Result) project, a more sophisticated architecture for modeling cognitive problem-solving [4] that uses state-space search in conjunction with machine learning techniques such as reinforcement learning. Newell's work inspired Anderson's ACT-R (Adaptive Character of Thought–Rational) system, a cognitive theory implemented as a production system that is meant to model human thought processes based on a set of declarative and procedural "memory modules" [5].

While CoSolve draws on the same concept of state-space search for human problem-solving, we chose a different approach to automating this process. Instead of creating a

system that attempts to model human problem-solving in its entirety, we created a system meant to *support* humans in their problem-solving efforts. In existing solving systems, the solving process is represented internally, the human solver creates a model or set of data to feed into the system, and then lets the solving system search the state-space on its own to produce a solution. With CoSolve however, the state-space search tree is implemented literally as an interactive visualization, allowing users to collaboratively generate and explore solutions. We do not claim that state-space search is necessarily an accurate, natural model of human problem-solving; instead we wish to explore the potential benefits a highly structured model of problem-solving may have for human solvers. Since it is meant to be used by human solvers, CoSolve is more similar to *problem-solving environments* than automated solvers. We will examine these next.

2.2 Problem-Solving Environments

Problem-solving environments, or PSEs, combine automation with human solving activity through a user interface. Gallopoulos [6] defines PSEs as follows:

> A PSE is a computer system that provides all the computational facilities necessary to solve a target class of problems. These features include advanced solution methods, automatic or semiautomatic selection of solution methods, and ways to easily incorporate novel solution methods. Moreover, PSEs use the language of the target class of problems, so users can run them without specialized knowledge of the underlying computer hardware or software

PSEs are usually built by software developers to be used by non-programmer domain experts. Most PSEs are created for scientific computing in areas such as JigCell for computational cell biology [7], GISolve for geographic information analysis [8], and the ELLPACK system for solving partial differential equations [9]. However, PSEs are almost always geared toward a very specific set of problems, whereas CoSolve's goal is to be generalizable to any problem. Attempts have been made to create a general theory for PSE development (e.g. Gallopoulos [6]), but as of yet, PSEs themselves are usually domain-specific. One example of a more general PSE is Kobashi et al.'s PSE Park [10], a "meta-PSE" for creating PSEs,

however the focus of PSE Park is still to create PSEs for scientific simulations. Additionally, the audience for PSEs is usually professional scientists or domain experts; CoSolve is intended to be usable by the general public. Also, PSE user interfaces are often meant for a single user. CoSolve, on the other hand, aims to provide a collaborative interface for multiple remote solvers to work together on a solution.

2.3 Problem Solving through Crowdsourcing

In the past decade, a new form of problem-solving environment has gained prominence in the scientific research community. Citizen science, or crowd-sourced science, is a method of solving difficult scientific problems that cannot easily be tackled by a small group of professional scientists, and so the scientific community engages the public to assist with gathering large amounts of data, or to perform data analysis that is either too computing-intensive, or that requires human intervention to accurately analyze. Many of these include a gamification element to keep participants engaged. Some of these systems simply make use of the human brain's pattern-recognition ability to visually classify data; for example GalaxyZoo participants classify galaxy shapes [11], and SETILive participants search radio frequency signals for signs of extra-terrestrial communication [12]. Others are more interactive, and rely on human spatial-reasoning and puzzle-solving ability to find solutions. One example of this is FoldIt, which is implemented as a game in which players fold protein structures; "high scoring" solutions are then used by scientists to determine the protein's native structural configuration [13].

There are also examples of crowdsourced human problem-solving systems that are better thought of as crowdsourcing platforms, rather than as solving environments themselves. Amazon Mechanical Turk [14] is a general crowdsourcing platform that allows task "requesters" to specify small "human intelligence tasks" (HITs) and post them to the Mechanical Turk site. Then, "workers" can select and work on these HITs for small amounts of money. In this way, requesters with problems that require human intelligence, and can be broken down into smaller tasks, can have their tasks completed in a short amount of time. However, this platform is not ideal for solving sophisticated problems that might require solvers to understand all the bigger picture beyond their individual tasks. A more holistic

view of crowdsourcing can be found in InnoCentive's platform [15]. InnoCentive allows companies ("seekers") to present R&D problems to InnoCentive's community of solvers. Solvers then propose solutions, and the solver who invents the best solution, as judged by the seeker, wins a cash award for the idea. This structure allows seekers to present more complex problems whose solutions might require more expertise and creativity than in the Mechanical Turk method, where HITs are often small, rote tasks.

In most examples of citizen science applications, and in Mechanical Turk, solvers only work on small parts of the overall problem. InnoCentive is somewhat closer to CoSolve in the sense that solvers work on a solution to the entire problem, rather than only a chunk of it, but solvers are typically competing against each other for the cash award, and not working together to solve a problem. This contrasts with CoSolve's goal of being a *collaborative* solving environment.

2.3.1 T-Star

In order to explore using a transparent-interfaces methodology with interactive state-space search for problem solving, Tanimoto created a Python toolkit for state-space search called T-Star (Transparent STate-space search ARchitecture) [16]. T-Star's interface did not provide any facilities for collaboration, but based on T-Star, Tanimoto then created CoStar, in which individual states were composed of component states, and these component states could be created by different users. He also created PRIME Designer, a desktop program which uses T-Star to enable users to collaboratively design a game [17]. PRIME Designer can be thought of as a CoStar "Problem Template," the way Towers of Hanoi is a Problem Template in CoSolve. In CoStar, however, there was no scaffolded posing facility such as the one currently in CoSolve. Creating PRIME Designer required implementing an entire standalone Python program.

PRIME Designer uses T-Star to create the tree visualization and generation of the problem space, which was the space of possible fully-formed versions of the game. In PRIME Designer itself, users are assigned to roles, such as an "architect" role, responsible for the layout of the rooms in the game, or the "music puzzle designer" role, responsible for

the music puzzles in the game. In this way, the work could be divided up and different users could bring their different strengths to the group. However, in a preliminary user study [18], we found that TStar was not easily usable for synchronous groupwork. As a result, CoSolve was built from scratch, but based on the initial ideas in T-Star. Unlike T-Star, CoSolve was developed from the start as a web application, and problems can be both posed and solved through its web interface.

T-Star is CoSolve's immediate predecessor. When designing CoSolve, we tried to incorporate features to make it more capable as a synchronously collaborative system. In the next section, we will discuss background literature on user interfaces for computer-supported, collaborative work systems.

2.4 Computer-Supported Collaborative Work Systems

CoSolve's design has been informed by much previous work in the field of user interfaces for collaboration. Dourish and Bellotti's work on CSCW systems identified the need for group awareness [19] which informed our evolution from TStar to CoSolve. CoSolve users engage in mixed-focus collaboration, as discussed in [20] and [21], when they move between working on their own branches of the tree to evaluating their group's work on the entire tree. Park et al. [22] and Heldal et al. [23] stress the importance of supporting an individual's ability to work independently when engaged in collaboration. Constantly being notified of one's teammates' activities can be distracting. To keep users' tree views synchronized without disturbing each individual's work, we queued up their teammates' changes, similar to work done by Suthers et al. [24], but we went a step further to empower users by displaying a Refresh indicator (discussed in Section 4.1.2) that showed how many updates from teammates were in the queue. The user was then allowed to click the indicator to update his or her view to include new changes when desired.

Gutwin and Greenberg [25] also identified a two-dimensional design space for applying workspace awareness to UI design: *literal* versus *symbolic* presentation of the awareness information, and *situated* and *separate* placement (whether the information is presented in the same environment as it took place in, or separate from it, for example, in a window with a log of actions that have been taken in the workspace). They suggested that literal-situated

is best because it most accurately reflects the current status of the workspace. CoSolve makes use of this, with *situated* placement of states in the tree, which solvers can use as a spatial point of reference. CoSolve also makes use of *separate* placement, for example, when nodes are annotated, the solvers are made aware of their teammates' new annotations when they appear in an annotation list on the bottom-left of the screen, away from the node that was actually annotated.

Even after the information is filtered with the addition of the Refresh indicator, how can it be presented in a way that doesn't distract the user? Gutwin and Greenberg [25] discussed ways in which workspace awareness is maintained in the physical world, and consider ways of providing similar affordances virtually. For example interface designers must make sure that actions performed by one user on an artifact can be visible to and noticed by other users, without distracting either the performer or the viewer. In the physical world, this happens through consequential communication (when an action is performed, for instance, a pilot lowers the landing gear, the action itself lets the co-pilot know that the action has been already been performed), and feedthrough (when the object has been manipulated, i.e. the landing gear lowered, one of its properties is that it makes a certain noise, and this noise caused by the object lets the co-pilot know the action has occurred). Possible functions to facilitiate awareness include slowing down the visual representation of a change–for instance, if a user has deleted an object, rather than have it immediately disappear, have it slowly fade away so that other users can see what has happened. Another recommendation was to provide feedthrough–for instance, show the pop-up menu that the other user is navigating through when they are accessing the "delete object" menu item. We felt direct feedthrough would be too distracting in CoSolve, but we do make use of Gutwin and Greenberg's suggestion to slow down the visual representation of changes. For example, when a solver uses the Refresh indicator to refresh his or her view, the new nodes all appear in the order they were created, one after another, rather than all at once, so that the solver has a chance to catch up on what happened.

Finally, there are currently many commercial, online collaborative work web applications that incorporate synchronous group awareness. Basecamp is a popular collaborative project management application that allow users to see projects updates on a "Daily Progress" time-

line in real-time [26]. Google Docs allows one to see teammates' edits to shared documents in real-time [27]. These tools are great for easily sharing information. However, they are meant as a repository for information, or for versions of a shared artifact, rather than for problem solving. Additionally, they don't have much support for meaningful computation, such as the transformation of potentially complex data objects required in CoSolve when an operator is required.

As we can see, much work has been done on collaborative software, and many such systems exist. However, no system to date has explored user interface issues inherent to using a state-space-search model for collaborative problem-solving visualization. For this reason, let us change our perspective from looking at systems that support collaborative work, to examining online collaborative problem-solving and collaborative design projects themselves.

2.5 Online Collaborative Problem-Solving and Design Projects

Why do people collaborate online? Butler et al. studied mailing list communities, and found that users participate in online communities for four different kinds of benefits [28]. Participants are looking for informational benefits of acquiring knowledge, social benefits of a community, visibility in a community, and altruistic benefits of helping the community. Newer or less active participants are generally looking for the first. Owners of a mailing list or more active participants are generally more interested in the altruistic benefit derived from a sense that one is helping the community itself.

In order to determine how to support users in an online collaborative problem solving environment, let's analyze examples of how users behave in a wide variety of other online collaborative projects. In particular, we will examine the Polymath Project, Wikipedia, open-source software (OSS), and collaborative Flash animation projects known as collabs. These represent a diverse set of needs, but nevertheless, all are attempting to make use of collaboration as a way to solve a problem. The Polymath Project is an example of collaboration to literally solve a problem: proving a mathematical theorem. Open-source software, Wikipedia and collabs are all very different examples of artifact generation, with OSS posing engineering challenges, Wikipedia focusing on collection of factual information,

and collabs favoring creativity and aesthetics as goals. Artifact generation itself is a form of problem solving–figuring out how to design an artifacts that fits a list of specifications. By examining how online collaborative problem-solving projects currently work, we can better inform the design of CoSolve.

2.5.1 Polymath Project: Collaborative Mathematics

On January 27, 2009, University of Cambridge mathematician Timothy Gowers asked the world through his blog: "Is massively collaborative mathematics possible?" [29]. Can mathematics research be crowdsourced? To explore this issue, he posted a mathematical research problem to his blog [30], and invited anybody and everybody to provide suggestions for a proof by posting comments in response to the blog entry. The idea was that, given enough minds working on trying to prove a mathematical theorem, eventually someone would be likely to come up with the right idea.

The problem Gowers selected was to find a combinatorial proof of the density Hales-Jewett theorem. As explained on his blog [31], he chose this problem because it was a "genuine research problem in [his] own area of mathematics" rather than an "elementary" or "recreational" problem, to show that this was a serious endeavor. At the same time, he refrained from selecting famously unsolved problems, wanting instead a problem for which he felt there was some realistic chance of solving. He also opted for a problem that was not obviously parallelizable, to better test the "question of whether it is possible for lots of people to solve one single problem rather than lots of people to solve one problem each" [31]. Gowers was not completely confident that this "massively collaborative" technique would find a solution to the problem, but remarked that he would consider the project a success if they could at least make "genuine progress towards an understanding of the problem" [30].

So what happened? Only six weeks later, the group not only proved the initially proposed theorem, but managed to prove an even stronger result [32], [33]. Twenty-three unique participants had contributed to the conversation now known as "Polymath1"[34]. The Polymath Project's first attempt at massively collaborative mathematics was a great success and proved that it is possible to throw a problem out into the wild, and through the

participation of the crowd, come up with a solution.

The Polymath Project is a clear example of how technology today can help humans solve problems. In the examples of citizen science mentioned earlier, we saw that the web can enable scientists to gather or process data quickly, but these cases were more examples of the "parallelization" of problem-solving that Gowers referred to, rather than actual collaboration. Here we see something different: solvers using the web to work together and communicate with each other to solve the problem. Before the advent of the web, mathematicians could perhaps meet at conferences to discuss proofs, or conduct a conference call, but realistically, most mathematical proofs have at most three or four authors [29]. More than that, and the rest of the participants become simply audience members. In the Polymath project, the web allowed twenty-three different people to actively participate in the proof of a mathematical theorem. Collaborating on the web made it possible for anyone to contribute, including people who may not have the ability, financial, logistical, or otherwise, to attend a mathematics research conference. It also enabled participants to join in as their schedules allowed, without limiting themselves to pre-determined meeting times, and without the hassle of scheduling such meetings.

Without these advantages afforded by the web, such a swiftly successful collaboration would have been more difficult. The past several decades have seen an increase in the ubiquity of computing, growth of computational power and data storage, and maturity in the field of computer networks. With such improvements in technology in place, now is the time to focus on how it can be used to human activities. Ben Shneiderman [35] characterizes this shift as follows: "the old computing is about what computers could do; the new computing is about what people can do." In particular, with the Internet and online social networking, we now have the power to harness the minds of many to solve the problems of today by allowing each individual to contribute according to his or her strengths. This is precisely what the Polymath Project proves.

Gowers had hypothesized that Polymath would enable people to provide expertise in their own areas of knowledge [29]. Different people know different things, so many minds working in isolation won't have the large volume of knowledge that many minds working together will. Additionally, different people have different ways of doing research. Some

prefer to look at the big picture of a math problem, some prefer to work on the details of a proof, some prefer to criticize ideas, some prefer to reformulate them, and so on. If everyone collaborates, then people can do what they're best at: one person can throw out a wild idea, someone else could refine it, someone could point out its flaws, someone could work out its details. Everyone can apply only a small amount of effort, but in their area of specialty, and so the end result will be better than if each person alone had tried to play many different roles. In the end, this indeed prove to be true. Gowers recalls [33]:

> To give one example, Randall McCutcheon made some very useful comments, but they were in the language of ergodic theory, which I understand only in a very limited way. But Terence Tao is a master at translating concepts back and forth between combinatorics and ergodic theory, so I was able to benefit from Randall's contributions indirectly.

However, in his analysis of the Polymath project, Gowers identified many issues and ways in which the technology hindered the group's participation [36]. Although Polymath was a success in that the group successfully solved the problem, it had not become as "genuinely massive" a collaboration as Gowers hoped. There were over a thousand comments, but only 23 participants–certainly a lot by conventional mathematics research standards, but not as many as expected. He also noted that most of the participants were mathematicians he knew, rather than the public at large.

Gowers asked those who did not participate, why they didn't do so. The response from many was that the format of the unthreaded comment posts that they were using as a communication medium hindered newcomers' ability to follow the conversation [32], [36]:

> A signicant barrier to entry was the linear narrative style of the blog. This made it difcult for late entrants to identify problems to which their talents could be applied. There was also a natural fear that they might have missed an earlier discussion and that any contribution they made would be redundant. [32]

Gowers had posted his problem as a blog entry, and the discussion was expected to take place in a list of unthreaded comments attached to the bottom of the list. This structure

made it difficult to keep track of what was going on in the discussion. If a participant came late to the game, he or she had no hope of catching up on hundreds of comments. It also meant that the latest comment was likely to be a response to some subsection of the problem, and it was difficult to keep different threads of the conversation straight. The group ended up improvising a variety of techniques, such as referring to different numbered comment posts in the discussion, and trying to separate and summarize sub-discussions onto different webpages or wiki pages, but none were satisfactory.

Additionally some felt that the conversation was so fast-paced that there wasn't enough time to both absorb what had happened and also contribute something back. For instance, a participant might post a comment, only to find someone else already said the same thing fifty comments earlier. Others didn't want to disrupt the flow of conversation by asking for clarification. However, the linear narrative style was also noted by some readers as being a fascinating and educational look at how mathematical research is done; commenters said they were able to learn a lot about how expert mathematicians solve problems. From this example, we can see how filtering the information could have been very useful to some users, at the same time, providing it all in one place was useful to others. Additional structure such as CoSolve's tree visualization might have proven beneficial to keeping users both aware of the ongoing conversation, while at the same time, allowing users to work on the sub-threads of interest to them without having to wade through all of the comment posts.

Gowers's examination also suggested some social issues in the group that may have prevented full participation. Some readers of the blog were apparently intimidated by the status of the other participants, which included a Fields medalist (Terence Tao) and several other renowned mathematicians. Some prospective participants did not want to look foolish by making "obviously wrong" comments. Here, we see that a possible, unintentional hierarchy in the status of the collaborators may have prevented some from participating.

This leads us to the question of whether or not such full participation may have actually mattered. It could possibly have been the case that important contributions to the proof were only made by the top participants, and that the stray comment from a reticent participant now and again never actually contained content significant to the final solution. Cranshaw and Kittur examined this very question in their research into the Poly-

math Project's process[37]. After the problem was solved, a Polymath wiki was created, containing a timeline that listed the 215 comments that had been important milestones in solving the theorem. Cranshaw and Kittur found that of the participants who commented five times or less (out of over 1200 total comments in the project), at least 23% of these participants had made comments that were listed on this timeline, meaning that "it was not uncommon for users who commented very infrequently to nevertheless have a large impact on the proof" [37]. So we see that less active participants still contributed meaningfully to the final solution. If more participants were encouraged to contribute, or to contribute more, perhaps the solution would have been found even more quickly.

From looking at Polymath, we find several lessons we can apply to the design of CoSolve. First, we saw that participants can be lost as to where they can best contribute. Second, outside factors such as social status or embarrassment can also prevent participants from contributing as much as they would like. Finally, we saw that even infrequent contributors played an important part in finding the final solution, and hence there could potentially be benefit in encouraging reluctant participants to contribute. Based on these observations, we designed the CoSolve Roles System, described in Chapter 5, which gives participants structured roles to help them deal with these issues. In the next three examples, we will look in particular at how roles are used in collaborative projects, to inform our development of CoSolve's roles system.

2.5.2 Collaborative Artifact Production: Wikipedia, Open Source Software Projects, Flash "collabs"

Luther and Bruckman [38] found that collaborative artifact production projects can be described along a continuum of completion, characterized by the release types, from single release to frequent release to continuous release. Some projects, for example, a movie–is only released in one version (usually). Other projects, for instance, a piece of software, will often have a version update every few months or years. Wikipedia is one of the few examples of a continuous release project–there is always a new release available of Wikipedia, and it is constantly changing. This completion dimension is useful for thinking about roles, because

the more frequent the release, the more freedom is in the hierarchy of roles. We will examine how this property effects each of these particular examples of collaboration.

Wikipedia One well-known example of successful collaboration is Wikipedia, a collaboratively edited, online encyclopedia. It has a hierarchy of contributors that play functional roles. Wikipedia itself says that "[t]heoretically all editors and users are treated equally with no 'power structure'. There is, however a hierarchy of permissions" [39]. This hierarchy is a progression:

1. **Anyone.** Any user is able to edit any page at any time, anonymously or with an account.
2. **Autoconfirmed editors.** This status is granted to any account registered 4 days or longer and with 10 or more edits. This permission level grants extra permissions such as voting rights and ability to edit certain protected articles.
3. **Editors in "good standing."** This status is granted as needed by higher status editors. This status grants users extra permissions to use certain tools for editing, on a per-tool basis.
4. **Administrators.** Users at this level, who are also known an "admins," are granted this status after applying and being approved via a consensus "election." Admins have extra permissions such as deleting articles and blocking accounts.
5. **Arbitration Committee.** This committee of editors is usually made up elected members and has the power to settle disputes.
6. **Bureaucrats.** Users at this status level are administrators with extra abilities, such as granting administrator status.
7. **Stewards.** These are users entrusted with complete power over the Wikipedia interface, ability to implement technical changes and handle emergencies.
8. **Founder.** The Founder is Jimmy Wales, the founder of Wikipedia. A special designation that basically has the same rights as a steward.

While Wikipedia aims to treat all its editors equally, there is definitely a hierarchy of power. This hierarchy is practical and necessary one for the prevention of vandalism and enforcement of Wikipedia policies. Its roles provide functional access to tasks–such as ability

to grant admin status–and to objects–such as ability to edit protected articles. These roles are explicitly defined through the granting of permissions.

There are also implicit, content-specific roles. Wesler et al. [40] identified roles among Wikipedia users; examples of such roles include substantive experts (have extensive knowledge of a topic, and contribute substantially to the content of articles), technical editors (perform fact checking, grammar correction, and other incremental contribution), counter-vandalism editors (find and fix vandalized articles, usually have many edits but those edits have little to do with the article topic), and social networkers (make excessive use of Wikipedia's social networking features, active participants of Wikipedia culture, such as the "Welcoming Committee" or "Birthday Committee").

In Wikipedia, awareness is maintained through facilities such as watchlists, which allows a user to specify which pages they are interested in, and whenever those pages are updated, the changes will be summarized in an interested user's watchlist. Users check their watchlists for activity so that they know when to review those pages to ensure that it is still accurate and follows Wikipedia policies. In some sense, a user's watchlist can be seen as another representation of their content-specific role in the system.

There is also a hierarchy of expertise, as in learning environments. Bryant, Forte and Bruckman [41] noted the evolution of a user from novice to expert Wikipedian: novices participate in tasks such as fixing typos in articles, and eventually as they become acclimated to the community, move on to tasks of a "meta user"–instead of editing content on articles, they perform duties central to the community and its preservation, such as mediating disputes and advising other community members. This is usually a natural process–novices entering the Wikipedia community do not do so with the intent of becoming an admin. They usually start searching for information on Wikipedia, and then notice a small correction to be made or error to be fixed. As their interest in Wikipedia continues, they perform larger and larger tasks, such as content creation, until they become full members of the community. This progression is also very similar to what has been found in other online collaborative communities, such as OSS projects.

Unlike OSS projects, where only certain users can have their changes incorporated into the core version of the product, in Wikipedia, absolutely anyone can contribute to the end

product, and hence the attitude of the contributors is slightly different. The reason for the difference perhaps can be found in Luther and Bruckman's [38] continuum of completion. Wikipedia is an example of continuous release. In Wikipedia, every single last edit is really a new release, a new version of the product. Hence, there is less of a need for strict hierarchical control over its participants–any bad edit can be immediately changed, and that fix is reflected in the product immediately. This is likely why Wikipedia's policy for the lowest role in its hierarchy is fairly permissive: anyone is allowed to make edits. This structure is most similar to CoSolve's, in that every state is a possible correct "solution," and if it is found to be incorrect, nothing needs to be done to remove it, other participants can simply continue creating their own nodes. While it may seem that Wikipedia's hierarchy is still fairly strict, we will see how it is even more strict in other forms of collaboration.

Open Source Software One area where we can clearly see the success of online collaboration is in the proliferation of open source software (OSS) projects. In Raymond's classic essay on the topic [42], he described the model of commercial software development as the "Cathedral" model, where a massive piece of software is carefully and reverently designed from ground up and assembled by a carefully managed, dedicated group of experts, in contrast to the OSS model "Bazaar" model of free-for-all software development, where contributions from participants varying wildly in style and agenda are thrown together in a hodge-podge of code, in hopes that something better will emerge from the sum of its parts. In reality, most OSS projects have more structure and more of a "Cathedral" style to them than it may seem. Study after study ([43],[44],[45],[46]) has found that these projects often do have social and administrative organization. They tend to have a hierarchy of functional roles as follows, with each role's membership being smaller than the previous [46]:

1. **Passive / End Users.** These are the users of the final product, who have little or no desire to read the code or improve the project, mostly interested in their own use of the product.
2. **Active Users.** These users report bugs, provide support, help new users, may read the code and/or occassionally provide minor patches or bug fixes.

3. **Developers.** These users make up a dedicated team of people who write most of the functionality of the code, and filter out contributions by the active users (around 40 for large projects like Python [43]).
4. **Core Team / Maintainers.** These users coordinate and review the activities of the developers, act as the buffer between the project leader to take on the tasks and make decisions the leader may not have the time or inclination for (only around 5-10 people for Python [43]).
5. **Project Leader.** This role is that of the "supreme leader," who is usually the initiator of the project, and has the final say on major decisions about the direction of the project. There is usually only one person in this role.

It may seem surprising that a "bazaar" model would have a leader, but this is often the case, e.g. Linus Torvalds for Linux, Guido van Rossum for Python, also known as "Benevolent Dictator for Life" [47]. This is true of many other kinds of collaboration as well, such as Wikipedia's Jimmy Wales, and it is also true for collabs, as we will see in the next section.

Nakakoji et al. [44] further divided OSS project roles down into: passive user, reader (reads the source code but does not write any), bug reporter, bug fixer, peripheral developer (occasionally contributes to a project), active developer (regular contributor, implements major features), core member, and project leader. These roles are neither mutually exclusive nor rigid, a user may simultaneously play more than one role, or may switch between roles through the course of their involvement in the community. They also note that some projects, for example PostgreSQL or Apache, are run by a small group that functions in the role of the project leader. This style is known as council style control, as opposed to the cathedral and bazaar styles.

Also, one's role, and hence influence in an OSS, is earned through one's contributions [44]. Ducheneaut [43] found that in order to gain influence in an OSS community, novices had to build an identity for themselves; in his study of the Python community, he found established members of the community had to know who a participant were before he or she was given access to the source control repository. This is in contrast to a system like

Wikipedia, where anyone's edits are accepted. He also found that a participant is more successful in having his proposals or changes accepted by the community if he understand the political structure of the community. Again, we see that even in a seemingly free-for-all environment like OSS development, a hierarchy of roles exist.

As far as continuum of completion, in a frequent release product like an OSS project, the idea is to "release early and often" [42]. Hence, new versions of the product are released every few days (or weeks, or months, etc.) As a result, although creators don't want to release a version until its reasonably stable, at the same time, they know that they can release a version later if bugs are found. This is the property that makes OSS thrive: while group structure is not completely free-for-all, for the sake of putting out working versions of the product, bugs can be fixed relatively quickly because of there is still some amount of freedom.

In this example, we see a similarity to Polymath in the hierarchy of roles, albeit an unintentional one. In Polymath, Gowers and Terence Tao very much played the roles of "supreme leaders" of the project, even though the project was meant to be in the "bazaar" model. They guided the direction of the project; Cranshaw showed that comments posted by either of them spurred more subsequent commenting activity than comments by other participants [37]. This suggests that, no matter the domain, perhaps humans naturally tend to fall into roles. We shall see this is strongly so in our last example, Flash collab projects.

Flash Collabs An example of a online, collaborative, single release project is that of a Flash collab [38], [48]. A collab is an animated movie, produced in Adobe Flash, created by a cooperative group via mostly or entirely online communication. These movies are usually published to online Flash communities such as Newgrounds, where participants generate Flash content such as videos and games, and comment on and rate each others' work. This is an example of a very different type of collaborative project, one that involves producing an aesthetically pleasing, creative artifact.

In a collab, generally one person in the community comes up with the idea for the theme of the movie and writes up the specifications for the movie. This is the project leader, a functional role. They then recruit other members of the community to work with them on

the collab, either by posting their collab idea and accepting whoever decides to join, holding a tryout competition for those interested, or actively ask members who they know or wish to work with to join their collab. In most collabs, each recruited artist is usually responsible for completely animating an entire, separate scene, or "module" of the final film, as opposed to in a professional animation studio, where some team members may be in charge of story, others in charge of sound effects, and so on. This is often necessary because in these Flash communities, there are high feelings of ownership and identity in one's work. As a result, collaboration is not so much between members in a collab, but between each member and the project leader, making for a very flat, two-level hierarchy, but also a very rigid hierarchy.

Luther and Bruckman speculated that this is because in single release online collaborative projects, there is more pressure for the final product to be perfect since there is only one chance for release, and so the project leader ends up shouldering a greater burden. This is also different from professional animation because, although they also produce single release projects, participants are not volunteers, and most are more interested in "mak[ing] their own thing" [38]. So we see that, in voluntary, online creative collaboration, the fewer releases a product can have, the stricter the control over the collaborators.

All communication on the collab occurs within a single discussion thread on the larger Flash community website's discussion forum. Here we see the same problem as with Polymath: users have a hard time keeping track of information. This suggests that CoSolve's tree branching structure could help organize information. Additionally, CoSolve's positive/negative annotation facility (described in more detail in Section 4.1.2) provides a way for solvers to mark and draw attention to the most important nodes, helping their teammates figure out what parts of the tree are currently the most fruitful to work on.

Summary of online collaborative projects

In these four projects, Polymath, Wikipedia, OSS, and Flash collabs, we have seen differing strategies for coordinating groupwork. Wikipedia and OSS have highly-organized tiers of hierarchy and access. Collabs had a strict hierarchy of only two tiers, but a strong sense of ownership. On the other hand, Polymath had less initial organizational or content-related

structure than the other projects, but content-related structure emerged as needed, and Gowers and Tao were inadvertently viewed as "leaders" of the project. As noted earlier, this unintentional hierarchy may have prevented some from participating as fully as they would have liked.

We address this problem with the CoSolve Roles System (Chapter 5), in which solvers' roles are assigned roles that are then automatically switched around on a schedule. Each participant has a chance to act within each role; we do this to see if it will increase participation rates, by turning low-contributing solvers into higher-contributing solvers. Additionally, many of the projects we have reviewed struggle with the issue of how to best structure large amounts of information. CoSolve's tree structure provides one solution to this issue.

Conclusion

In this chapter, we have discussed the idea, first pioneered by Simon, of human problem-solving and design activity, as instances of processes that can be modeled by state-space search. We reviewed automated solvers that use state-space search for problem solving, and then looked at user-oriented problem-solving environments, and other collaborative work systems. Finally we analyzed examples of online collaboration projects, and their strengths and weaknesses. These projects inform the design of our own collaborative problem-solving system. However, none of them provide a general problem-solving system that allows users to work together to visualize, generate, and manipulate a state-space search tree, nor do they afford a general problem-posing system for novice solvers to easily formulate their own problems. In the next chapter, we will discuss how the problem-posing framework in CoSolve provides such a system.

Chapter 3

PROBLEM POSING IN COSOLVE

CoSolve is designed for problem-solving, but a crucial part of the problem-solving process is specifying the problem that needs to be solved. Before a problem can be solved collaboratively, there must be agreement among collaborators as to what the problem is, what factors are important and unimportant in a solution, what the variables are, what the constraints are. These issues can vary from one formulation of the problem to another.

Take the example of designing a community playground. One neighborhood wants to build a playground on a small, empty lot between two buildings–for that neighborhood, the constraint may be the size of the plot, and so the residents must select playground equipment that will fit on a small land area. Another neighborhood might have a budgetary constraint, but not yet have chosen a suitable site–in which case, their problem specification needs to have an extra variable to account for different kinds of build sites, which may vary in cost as well. Or, in one busy neighborhood, the residents may be more concerned about their children's safety in a high car-traffic area, whereas in a residential cul-de-sac, this requirement need not be in the problem specification at all. Without understanding what the parameters of a problem are, it is difficult to know how to come up with a suitable solution.

This is often an iterative process. First, the person with a problem to solve might attempt to define the problem, then try to brainstorm solutions, and finally select and evaluate a candidate solution. However, in the course of evaluation, the solver might realize that the initial problem specification was insufficient–perhaps there were additional parameters he or she did not consider, for example, perhaps residents did not realize, until they started trying out different layouts for playground equipment, that they would need to factor in enough space to walk between them, and the cost of gardening and materials for those paths. So in the next iteration of the solving process, the solver re-frames the problem specification

based on this new knowledge, and the "define, brainstorm, select, and evaluate" cycle begins again. Often the very act of attempting to solve a problem changes one's assumptions about what the problem itself is. Problem posing and problem solving are connected and closely intertwined activities.

CoSolve provides support for both the problem solving process, and the problem posing process. The solving process is detailed in Chapter 4. This chapter discusses problem posing, and, as defined in Chapter 1, we will refer to problem posing as the process of designing and then implementing a CoSolve problem template, rather than the entire problem formulation process, though the boundaries of this can be fuzzy. Also, although these posing and solving activities are physically divided into two parts of the CoSolve interface, they can be cycled iteratively, with observations from a solving session informing changes to the problem specification, which can then be reflected in subsequent solving sessions.

We have designed CoSolve's posing interface as a scaffolded programming system with starter code so that problem-posers with minimal programming skill are still able to pose problems in CoSolve. This chapter will begin with a technical section (Section 3.1) on our posing system design, and how it scaffolds the posing process for novices. Then in Section 3.2, I will examine a set of problems that have been posed as class projects by undergraduates in an artificial intelligence class, to illustrate the different ways problems can be posed in CoSolve. Finally, I will end with a discussion of the scope of problems that can be be posed in CoSolve in Section 3.3.

3.1 Description of the Problem-Posing Process in CoSolve

Our design goal when developing the problem posing interface in CoSolve was to find the appropriate balance between two different ideals: on the one hand, we wanted to give users the power and flexibility to create templates for as a wide range of problems as possible. At the same time, we wanted CoSolve to be accessible to people of all backgrounds and minimal to no programming knowledge.

We settled on a medium between the two, by setting up CoSolve's posing interface as a web interface, with scaffolded code snippets that the user can either modify, or replace entirely. The term *scaffolding* has meaning in two relevant contexts: computer program-

ming and education. In the field of computer programming, the term *scaffolding* is most commonly used in reference to Ruby on Rails and other similar frameworks, to describe a technique for a programmer to provide a model specification from which working database code is then automatically generated [49]. In the field of education, *scaffolding* refers to learning artifacts–particularly educational software–where some expert knowledge has been included in the design in order to support the learner in some learning task[50]. In terms of CoSolve posing, we use *scaffolding* in a sense closer to the educational context–teaching our posers how to write problem templates–though also a little bit similar to the programming context of automatically generating runnable code. To make posing easier for novices, when a problem template is initially created, CoSolve's posing interface provides skeleton code that is already a valid, runnable problem template; this way, new users can begin with relatively little start-up cost. They can pose problems by making small modifications to the template, and they can learn about the posing process by looking at valid code already provided for them.

We chose Python as the language for these poser code snippets as it is relatively easy to learn, and is currently popular and well-supported, with plentiful tools and resources, and hence there is a higher chance posers may be familiar with it or willing to learn it. Other than certain CoSolve-reserved constructs described in the following section, any code that is valid Python is valid CoSolve Python, giving more advanced users the power and flexibility to create templates as complex as any Python script they can write.

In the next section (Section 3.1.1), I will describe CoSolve's problem-posing constructs. Then in Section 3.1.2, I will provide a concrete example using the Towers of Hanoi problem to illustrate how posing works in practice.

3.1.1 Creating Problem Templates

To pose a problem in CoSolve, a user creates a *problem template*. Users who create problem templates are called problem-posers, or *posers*. A problem template specifies what the initial state in the problem space should look like, and provides other details about the problem. Each problem template has a set of associated *operators* which posers create. Later, as part

of the solving process, these operators are applied to states to create child nodes.

Problem templates and operators contain both a set of human-readable text descriptions for the solvers, and a set of scaffolded Python code snippets that specify what the structure of a state in this problem should look like. The poser specifies all of these. The code snippets, also called *poser snippets*, are run when solvers instantiate a solving session from a problem template and when they apply operators.

The most important thing for a poser to specify, the structure of each *state* of a problem, is done by initializing a Python *dictionary* variable, where each key-element pair in the dictionary is some property of the state. The poser specifies properties to be stored in a special CoSolve-reserved state variable name, `S`. So the poser might write the following Python code snippet for, say, an office redesign project:

```
S = { "allocated budget": 5000,
      "amount spent": 130,
      "furniture": ["wastebasket","office plant","couch"],
      "approved": false,
      "project name": "Office Redesign Project" }
```

The poser can set some initial values for the variables, which will be changed during the solving process as the solvers apply operators. Also, applying operators can also enable new key-element pairs can be added to the `S` state dictionary.

There are several such CoSolve-reserved Python variables:

`S` - Described above. Each new state (tree node) shown to the solver is stored internally as a Python dictionary `S`, but the poser needs only to explicitly initialize the initial state representation of `S`.

`D` - Also a Python dictionary variable, stores any commonly-used constant data that may need to be accessed by operators, but should not necessarily part of the `S` state representation, e.g. value of π.

`IMG_HIGH` - a variable that holds a Python Imaging Library (PIL) Image object instance. When writing code for visualizing a given state, the poser creates and writes to this

Image object. `IMG_HIGH` stores a high-resolution version of the image, useful for a zoomed-in view of the state.

`IMG_LOW` - Same as `IMG_HIGH`, except `IMG_LOW` is used as a thumbnail in zoomed-out views of the tree. `IMG_LOW` view can potentially show a summarized version of the data, or only some of the relevant properties of a state, whereas `IMG_HIGH` could be used to display all details of a state.

`IMGS` - a Python dictionary variable to store alternative visualizations of each state. Instead of just having two options, high-resolution and low-resolution, for displaying a state, the poser can specify alternate visualizations of a state that the user can select from, for instance, a top-down view of a map versus a street-view versus a topological map, etc. In each item of the `IMGS` dictionary, the key is a name label for the view, and the element is a PIL Image object.

To begin posing, the poser first creates a problem template on CoSolve by logging on to the CoSolve website, and navigating to the "Create new template" page.[1] This page is a web form with a set of elements to fill in which specify what the structure of the problem is. The main elements the poser provides are:

Problem title - A human-readable string of text that identifies this problem template, e.g. "Happy Valley Playground Design Problem."

Problem description - A text description, e.g. "We are trying to solve the problem of designing our neighborhood playground."

State Initialization - Code snippet that describes the unsolved, initial root state of the problem, e.g. code for a two-dimensional array that represents an initial plot of land with no playground equipment on it, and an integer variable that represents dollar amount spent. Initializes the `S` variable. Required.

Visualization Code - (a.k.a. *Visualization*) Code snippet that tells CoSolve how to generate the image to be displayed to solvers for each state. Initializes the `IMG_HIGH`, `IMG_LOW` and/or the `IMGS` variables. Optional; if not provided, CoSolve will generate default state images, by printing and formatting the members of the `S` dictionary.

[1] http://cosolve.cs.washington.edu/node/add/problem

Common Data - Code snippet for any datasets that may be needed by the problem, e.g. a list of playground equipment and their prices and dimensions. Optional. Initializes the `D` variable.

Common Code - Code that user provides as a "library" for the rest of the problem template and operators. Contains commonly used functions, e.g. a Python function that recalculates the total amount spent on the playground at each given state and returns whether or not it has exceeded the budget. Optional.

After the poser has filled out the problem template form, he or she can then create the operators for this problem. Posers can create as many or as few operators as they wish for a given problem template. Again, an operator is used by solvers during the solving process by selecting a state, and then selecting an operator to apply to that state. The operator applies some transformation to that state, and creates a child state that represents the transformation, keeping the former "parent" state intact. Operators may optionally have parameters. For example, if I decided to apply the operator "Place monkey bars" to my initial state playground, I can specify the location of the monkey bars as a parameter.

To create an operator, the poser navigates to the "Create Operator" web form in CoSolve.[2] This page contains the following elements:

Operator name - A human-readable string of text shown to solvers that identifies this operator, e.g. "Add swing set to playground"

Operator description - A text description of the operator, e.g. "This operator will put a swing set onto the playground layout in a specified location."

Parameter type - The poser selects from one of the following:

- **None, no parameters** - No parameters will be specified for this operator
- **String/text** - There is a single parameter for this operator, which will be parsed as a text string
- **Named parameters** - There may be more than one parameter, or the parameters may not be strings, so the poser will specify details in the "Parameter

[2] http://cosolve.cs.washington.edu/node/add/operator

Specifications" section

- **Parameter prompt** - A text prompt displayed to the users to explain what the parameter is, e.g. "Please provide coordinates for the location to place the swing set"

Parameter specifications - This section of the web form allows the poser to add an arbitrary number of parameters to the operator.[3] If the poser selected "Parameter prompt" as the "Parameter type" above, then the poser must fill in this section. Each parameter has the following values, which the poser provides:

- **Label** - Text label displayed to solvers that identifies this parameter, e.g. for the operator "Add swing set to playground", there might be a parameter to specify location, and the Label could be "Swing set location"
- **Name** - Programmatic name of this parameter to be used by poser to identify this parameter in code snippets, e.g. "`swing-set-location`"
- **Description** - Additional text instructions to display to the user, e.g. "Enter horizontal and vertical coordinates of the location you would like to place the swing set in, relative to the northwest corner of the lot."
- **Type** - The expected programmatic data type of the input parameter. CoSolve will attempt to validate and parse the user's input accordingly.

  There are five possible types: string (text), integer, decimal (float), Boolean (true/false), 2D Point (x,y string list). Type defaults to string if none selected.

  2D Point is a special data type that is used to capture the solver's mouse clicks. For example, instead of typing in the location to place the swing set, during a solving session the solver can click on a location within the state image to specify where he or she would like to place the swing set, and this information is passed to CoSolve as a 2D Point parameter to the "Add swing set to playground" operator.

---

[3]Laura Dong, an undergraduate researcher in our lab, implemented this feature.

- **Default value** - Specify a default value for this parameter that is used if the solver does not provide one.
- **Required** - Checkbox to specify whether or not the solver is required to provide this parameter when applying this operator.

State transformation code - Code snippet that runs when this operator is applied, transforming the selected state into the appropriate child state. E.g. if this is the "Add swing set to playground" operator, this code will add a swing set object to the representation of the plot of land, at the location specified in the parameters, and look up the cost in the Common Data problem template field and add it to the dollar amount spent.

Precondition code - Code snippet that must evaluate to true or false, based on the currently selected state. This determines whether or not this operator can be applied to the state a solver has chosen; if the snippet evaluates to false, this operator will not be shown as part of the available operators, and the solver does not have the option to apply it. For example, if a swing set costs $5000, but the budget of the selected state that the solver wants to apply this operator to only has $100 left, this operator cannot be applied, and will not be available to the solver.

Problem - Select from a list of available problem templates to specify which problem template or templates will use this operator, e.g. the "Add swing set to playground" operator is applicable to the "Happy Valley Playground Design Problem" and the "High-Traffic Neighborhood Playground Problem" problem templates.

The poser does not always have to be a single user. Multiple users can work together to edit problem templates and operators; if a conflict arises, the last poser to save his or her changes is notified of the conflict and given the opportunity to make corrections.

Also, problem templates and operators can be continually refined and updated. If solvers using the problem template to solve a problem find that it lacks something–say the initial posers did not have an operator to add a sandbox, the solver could go and create a new "Add sandbox to playground" operator for the problem template, to use in the solving session. In this case the solver would be doing a "poser" action, but these two roles are not

necessarily mutually exclusive.

3.1.2 Example Problem Template: Towers of Hanoi

Now that we have given an overview of the available posing facilities, we will go through a specific example of problem posing, for the Towers of Hanoi puzzle. The rules of the puzzle are described in Chapter 1. The complete code listing for this example is in Appendix A.

There are two steps to posing in CoSolve, as mentioned before. The first step is filling in the Problem Template form, and the next step is creating operators by filling in one or more Operator forms. We must start with the Problem Template form.

Problem Template form

We start by accessing the "Create Problem Template" page on CoSolve[4]. We are then presented with a web form with a several text areas. The first is "Problem Title", so we will fill in "Towers of Hanoi".

The next is "Problem Description", in which we will type some text to describe the game and its rules, for solvers to see. (actual Problem Description used is in Appendix A

After that we see a text area labeled "Common Data". It currently contains the following:

```
D = {} # add any common data as a dictionary list to the D variable
```

The Towers of Hanoi puzzle can have variations based on how many disks are available. We'd like our problem template to handle three disks, but to be modifiable in the future should we choose to edit or copy this template. So we'd like to set a constant value for the current problem template to store the number of disks for this particular template. So we do the following:

```
D = { 'numberOfDisks' : 3 }
```

[4]http://cosolve.cs.washington.edu/node/add/problem

We can set any number of variables into the `D` dictionary as we like, but this is all we need for now.

The next section of the form we will look at is the State Initialization, which currently looks like this:

```
S = { }
```

This code represents an initial empty state for this problem, the root of the solving session tree. If we were to save the template now and create a solving session for it, the session would be created successfully, and CoSolve would simply generate an empty state as a root. However, for our problem, we want our root state to be the initial state of a Towers of Hanoi problem, which contains three pegs, with `D['numberOfDisks']` disks on the first peg.

There are many different ways to represent this in CoSolve. For example, we could create three state variables, `S['first-peg']`, `S['second-peg']`, `S['third-peg']`. We could even define our own `Peg` Python class if we wanted to, and our own `Disk` class, etc, but that's probably overkill for such a simple problem. Here we will create a single variable of the built-in Python type `list`, of length 3, to store the pegs. Each of the items in the list represents one peg.

So, first, we create a state variable called `peg` as follows:

```
S = { 'peg' : []}
```

We want three items representing the three pegs inside the list, so let's instead write:

```
S = { 'peg' : [ [], [], [] ]}
```

So now we have three lists (pegs), that can be accessed by executing `S['peg'][0]` to get the first peg, `S['peg'][1]` to get the second peg, and `S['peg'][2]` to get the third peg. Let's actually put some disks on the peg. Again, we can represent these however we want, as strings, as numbers, or even as class instances. Here, let's use ints to represent the disks, with 1 representing the smallest disk, 2 representing the second largest, and so on. We will use our `D['numberOfDisks']` variable, and place these disks on the first peg, `S['peg'][0]`. So let's modify the code again, producing our final code, shown in Listing 3.1.

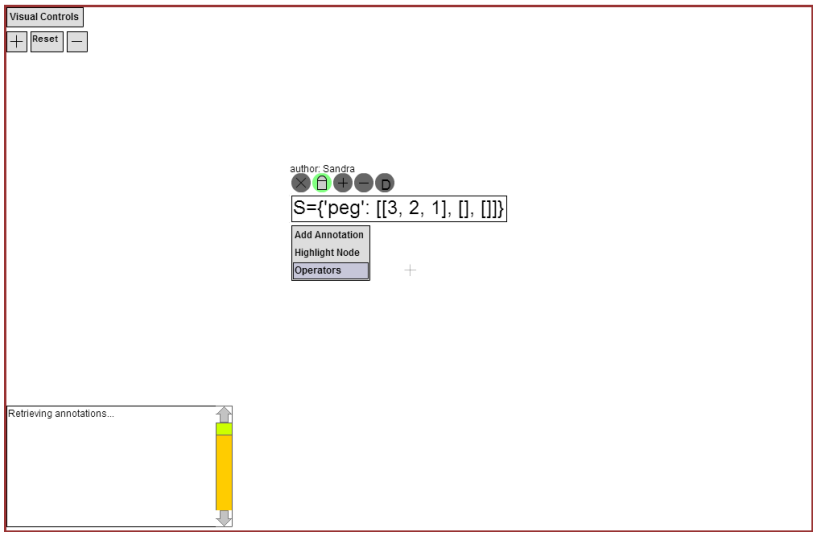


Figure 3.1: Solving session with root node created from code in Listing 3.1

Listing 3.1: Finished State Initialization code for Towers of Hanoi.

```
1 S = { 'peg' : [ list(range(D['numberOfDisks'],0,-1)), [], [] ] }
```

The call to the Python function `range` will create a list of integers, from 3 (the value of `D['numberOfDisks']`) inclusive to 0 exclusive. So, when this code is run, the first peg, at `S['peg'][0]`, will be set to a list `[3, 2, 1]`. The end of the list can now represent the top of the peg, so we can see that the smallest disk, `1`, is at the top of the peg, and as long as the disks are in descending order, we know we have not violated the disk stacking rule (no larger disks on top of smaller disks). The other two pegs are left empty, and we are now done defining the state. If we save the template now and instantiate a solving session, the root node will display a text printout of our state variables, as seen in Figure 3.1.

At this point, there are two different things we can do. We can work on the Visualization of this state, or we can work on the Operator forms. Let's work on the Visualization first.

The Visualization code looks like this by default:

```
import PIL, Image
# Use IMG_HIGH to store an Image object that represent your high resolution
    image
# Use IMG_LOW to store an Image object to represent the thumbnail
```

The first thing to notice is that PIL, Python Imaging Library, and the PIL Image class is imported for you. CoSolve will look for any variables in your Visualization code named `IMG_HIGH` or `IMG_LOW`. We need to store objects of PIL type Image in those variables, and then CoSolve will generate this image and save it as a file on CoSolve's web server. When viewing the solving session, this image file is displayed as a node in the tree.

The Visualization code is actually not run by default, instead, CoSolve simply generates the `S` dictionary variable as text without running any of the Visualization. The poser can elect to use the Visualization code by selecting the appropriate toggle on the web form. At this point, our template web form will look like Figure 3.2.

Now, let's create our Visualization code by adding the following:

Listing 3.2: Visualization code to create a new image with nothing in it

```
import PIL, Image, ImageDraw
# Use IMG_HIGH to store an Image object that represent your high resolution
    image
# USI IMG_LOW to store an Image object to represent the thumbnail
width = 400;
height = 200;
im = Image.new("RGB", (width,height))
IMG_HIGH = im
```

If we now save our template form and create a new solving session, it will generate a tree with just a root, and at the root will be a 400 by 200 pixel solid black rectangle that is the blank image we just created in Listing 3.2, as seen in Figure 3.3, instead of the `S` variable text as before.

So now let's make our solid rectangle actually contain information from `S`. To do so, we simply access the `S` variable to get information, and use the PIL class ImageDraw to draw.

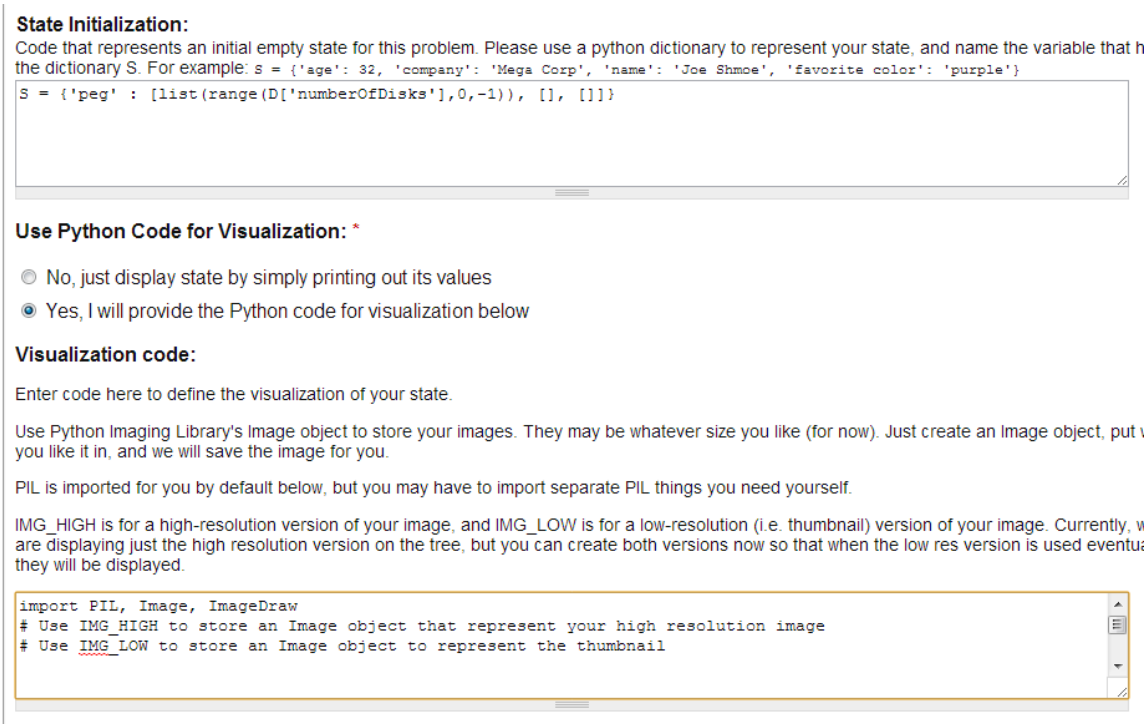


Figure 3.2: Partial screenshot of template web form after adding the code from Listing 3.1, and turning on the "Using Python Code for Visualization" toggle. Note that the Visualization code area is still in its default state.

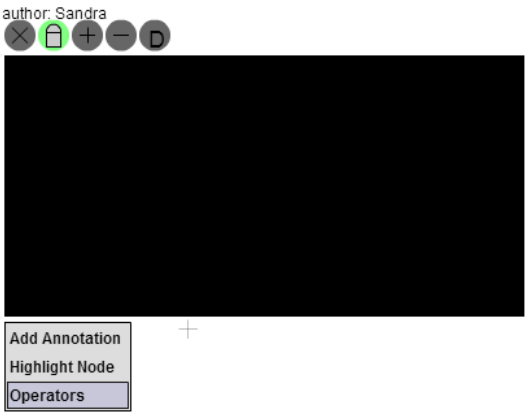


Figure 3.3: Root node of a Towers of Hanoi solving session after saving the code from Listing 3.2

Let's just start with drawing three vertical rectangles to represent the pegs (Listing 3.3).

Listing 3.3: Partial Visualization code for Towers of Hanoi

```
1  import PIL, Image, ImageDraw
2  # Use IMG_HIGH to store an Image object that represent your high resolution
       image
3  # USI IMG_LOW to store an Image object to represent the thumbnail
4  width = 400;
5  height = 200;
6  im = Image.new("RGB", (width,height))
7  draw = ImageDraw.Draw(im)
8  peg_area_width = width / len(S['peg']) # notice we can access S directly
       here. The len function will allow us to get the number of pegs in our
       state variable S['peg']
9  peg_color = (0,0,255) # this is RBG, so pegs will be blue
10 for peg in range(len(S['peg'])): # and also accessing S here
11     location_x = peg * peg_area_width
12     topLeftY = height / 5
13     bottomRightY = height * 4 / 5
14     topLeftX = location_x + (peg_area_width * 0.45)
15     bottomRightX = topLeftX + (peg_area_width * 0.1)
16     draw.rectangle((topLeftX,topLeftY,bottomRightX,bottomRightY),
17                    fill=peg_color)
18 IMG_HIGH = im
19 IMG_LOW = im
```

If we now save our template with this Visualization code and create a solving session, the solving session would look as shown in Figure 3.4.

Notice that we are directly accessing the `S` variable in lines 8 and 10 in Listing 3.3 above. We set `IMG_LOW` to be the same Image object in this example, but normally, one could set it to be a lower resolution version for the thumbnail when zoomed out.

Let's finish the rest of the code to draw the disks, in Listing 3.4. The details of this are not important, but do notice the use of `D` and `S`. We can access the `D` Common Data variable, in lines 10, 27 and 29, to get the number of disks. We also access the `S` variable again in line 26, to access the disks on each peg.

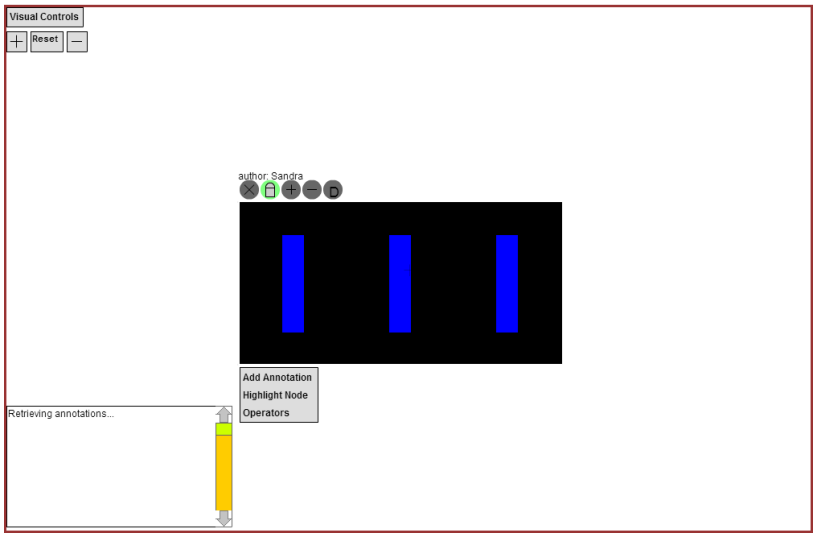


Figure 3.4: Solving session with visualization of root node created from code in Listing 3.3

Listing 3.4: Complete Visualization Code

```
1  import PIL, Image, ImageDraw
2  width = 400;
3  height = 200;
4  im = Image.new("RGB", (width,height))
5  draw = ImageDraw.Draw(im)
6  peg_area_width = width / len(S['peg']) # notice we can access S directly
       here. The len function will allow us to get the number of pegs in our
       state variable S['peg']
7  post_width = peg_area_width * 0.1
8  post_top_y = height / 5
9  post_bottom_y = height * 4 / 5
10 disk_height = ((height*2/5) / D['numberOfDisks']) - 5; # Using Common
       Data's 'numberOfDisks' to calculate a variable height for the disks
       based on how many disks there are
11 peg_color = (0,0,255) # this is RBG, so pegs will be blue
12 disk_color = (255,0,0) # disks are red
13 for peg in range(len(S['peg'])): # and also accessing S here
14     location_x = peg * peg_area_width
15     topLeftX = location_x + (peg_area_width * 0.45)
```

```
    bottomRightX = topLeftX + post_width;
    draw.rectangle((topLeftX,post_top_y,bottomRightX,post_bottom_y),
                   fill=peg_color) # post
    draw.rectangle((location_x+5,
                    post_bottom_y,
                    location_x+peg_area_width-5,
                    post_bottom_y+post_width),
                   fill=peg_color) # base
    # Now draw disks, from bottom up
    bottom_y = post_bottom_y
    for disk in S['peg'][peg]: # Here we access S again, this time to get
        the disks from each peg
        disk_width = (disk * (peg_area_width-10) / D['numberOfDisks'])
        x1 = (location_x + (peg_area_width - disk_width) / 2)
        y1 = bottom_y-(disk_height*(D['numberOfDisks']-disk)) - 2
        x2 = x1 + disk_width
        y2 = y1-disk_height + 2
        draw.rectangle(( x1,y1,x2,y2 ),
                       fill=disk_color)

IMG_HIGH = im
IMG_LOW = im
```

Now we can save our template and create a new solving session to get the result in Figure 3.5. However, note that even though the "Operators" menu item is selected, no operators appear in the menu. So our next step will be to create operators for this problem template.

Operator form

After creating a problem template, the Problem Template webpage will have a link to "Create new operator for this problem". Clicking on it will bring us to the Operator form. We can also directly go to the Operator form page[5] without going through a Problem Template. This will allow us to create an Operator that we could associate with potentially many different problem template (presumably with similar `S` state variables).

[5]URL: http://cosolve.cs.washington.edu/node/add/operator

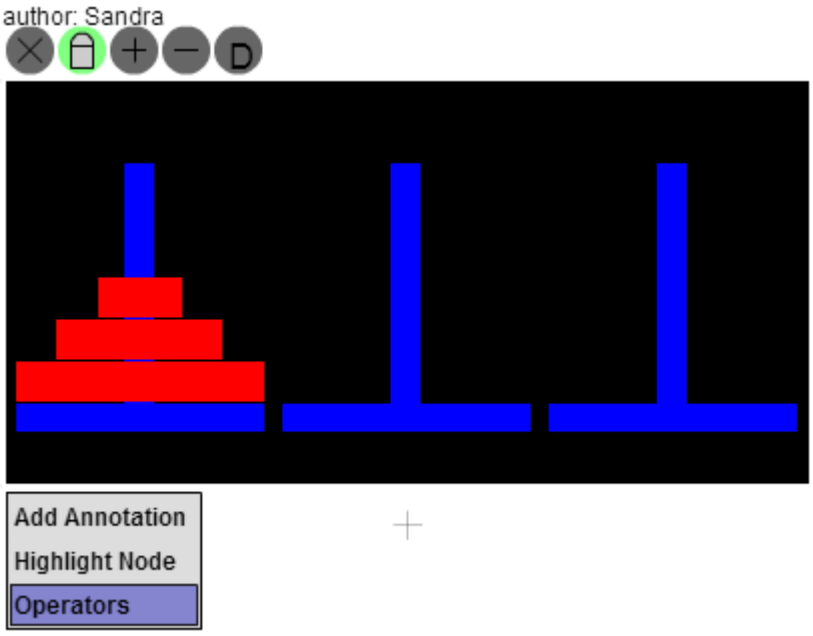


Figure 3.5: Completed visualization of Towers of Hanoi from code in Listing 3.4.

As before, we first fill out the "Operator name" and "Operator Description." Here, we will implement the operator to move the first disk from the first peg to the second peg (going from left to right). So we will name this operator "Move the top disk on the first post to the third post." and give it a description of "Move disk from peg one to peg three." by filling out the appropriate text areas in the web form.

When this operator is run, we want to take the disk at the top of `S['peg'][0]`, which is currently disk `1`, and move it to the top of `S['peg'][2]`. So, in the State Transformation text box, we will put:

Listing 3.5: State Transformation code for "Move the top disk on the first post to the third post" operator.

```
disk = S['peg'][0].pop() # removes last item from the list
S['peg'][2].append(disk) # adds it the the end of the list
```

Now, we can click "Save" on the Operator form. Now when we start a new solving session, the "Move the top disk on the first post to the third post" operator will be displayed

in the menu. However, it will always be displayed, even after there are no more disks on the first post to move, which means the solver might be in an illegal state. To tackle this, we could either put in a check in our State Transformation code, then add a variable such as `S['status']="Illegal move."` and edit our Visualization code to display it, or we can just add a precondition for this operator. Let's do the latter. We will now go back to the Operator form and edit our code to include a Precondition.

The Precondition area of the code looks like this initially:

```
PRECONDITION = True # replace this, the value of PRECONDITION will be used
    to determine whether this operator is run
```

The `PRECONDITION` variable is read by CoSolve to determine whether or not to display this operator. We can again use the `S` variable to set this value as true or false.

We want this variable to be true if there exists a disk on the first peg, *and* this disk is smaller than any disk already on the third peg. So we write:[6]

Listing 3.6: Precondition code for "Move the top disk on the first post to the third post"

```
if S['peg'][0]: # if list is NOT empty, i.e. there is a disk on the first
    peg
    if S['peg'][2]: # if peg 3 has disks on it
        PRECONDITION = S['peg'][0][-1] < S['peg'][2][-1] # check last disks
            on pegs
    else:
        PRECONDITION = True
else:
    PRECONDITION = False
```

Now if we save this operator and go back to our solving session, we should see that the "Move the top disk on the first post to the third post" will only appear in the menu when the `PRECONDITION` for this operator is true.

The last area we have not discussed is the Common Code area, which is actually in

[6]The precondition code in Listing 3.6 can be more briefly written as follows:
`PRECONDITION = bool( S['peg'][0] and (not S['peg'][2] or (S['peg'][0][-1] < S['peg'][2][-1])) )`
but we are using a longer version in the code listing above for the sake of clarity.

the Problem Template form. Since we will be reusing the code from this operator State Transformation and Precondition areas for all the other operators, e.g. "Move the top disk on the second post to the third post", "Move the top disk on the third post to the second post" etc., we could instead just call a single function from each of these to do the work. So in our Common Code, which is currently blank, we could put:

Listing 3.7: Common Code for Towers of Hanoi

```
# move top disk between pegs
def move(from_peg,to_peg):
  disk = S['peg'][from_peg].pop() # removes last item from the list
  S['peg'][to_peg].append(disk) # adds it the the end of the list

# check whether move between pegs is valid
def isValidMove(from_peg,to_peg):
  result = False
  if S['peg'][from_peg]: # if list is has items, i.e. there is a disk on
      the from_peg
       if S['peg'][to_peg]: # if the to_peg has disks on it
           result = S['peg'][from_peg][-1] < S['peg'][to_peg][-1] # check
               top disks on pegs
       else:
           result = True
  return result
```

Then, we can change our code in the Operator form for State Transformation and Precondition as follows:

Listing 3.8: Revised State Transformation code for "Move the top disk from the first post to the third post"

```
move(0,2) # this function is defined in Common Code
```

Listing 3.9: Revised Precondition code for "Move the top disk from the first post to the third post"

```
PRECONDITION = isValidMove(0,2) # this function is defined in Common Code
```

Notice that we did not need to explicitly use the `S` variable, because it is used in definition of these functions in the Common Code area.

Then we simply repeat this method for all the other operators, for example, for "Move the top disk from the second post to the first post", the State Transformation code would be:

```
move(1,2)
```

and the Precondition code would be

```
PRECONDITION = isValidMove(1,2)
```

and so on for all the operators.

There is one more Operator topic we have not covered yet, and that is the Parameters feature. We did not make use of parameters in this template, but many templates do, so let's discuss them now.

Parameters in the Operator Form

Instead of creating six operators for each of the possible moves, i.e. from first post to second post, from first post to third post, etc., it is possible to create just one "move" operator, and then have the solver directly specify which posts to move the disks to and from, as parameters to these operators, for instance, by entering "1" and "3" when applying an operator from a solving session to move from disk 1 to disk 3.

Let's see an example of what this would look like. We can go back to our Operator and change the name to "Move disk" now, as this operator will move disks between any posts. Then let's look at the Parameter Specifications section of the Operator form. In Figure 3.6, we have added two parameters to this operator. The first is labeled `from_peg` and will show up as "From which post?" to the solver. The second is labeled `to_peg` and will show up as "To which post?" for the user. We can optionally set a default value, here arbitrarily set to 1 and 2 as an example, but they can be left blank, as this operator is meant to work for moving a disk between *any* posts. Then we can select what type of value the parameter should be, here, we select "Integer" (CoSolve will validate solvers' parameter input type).

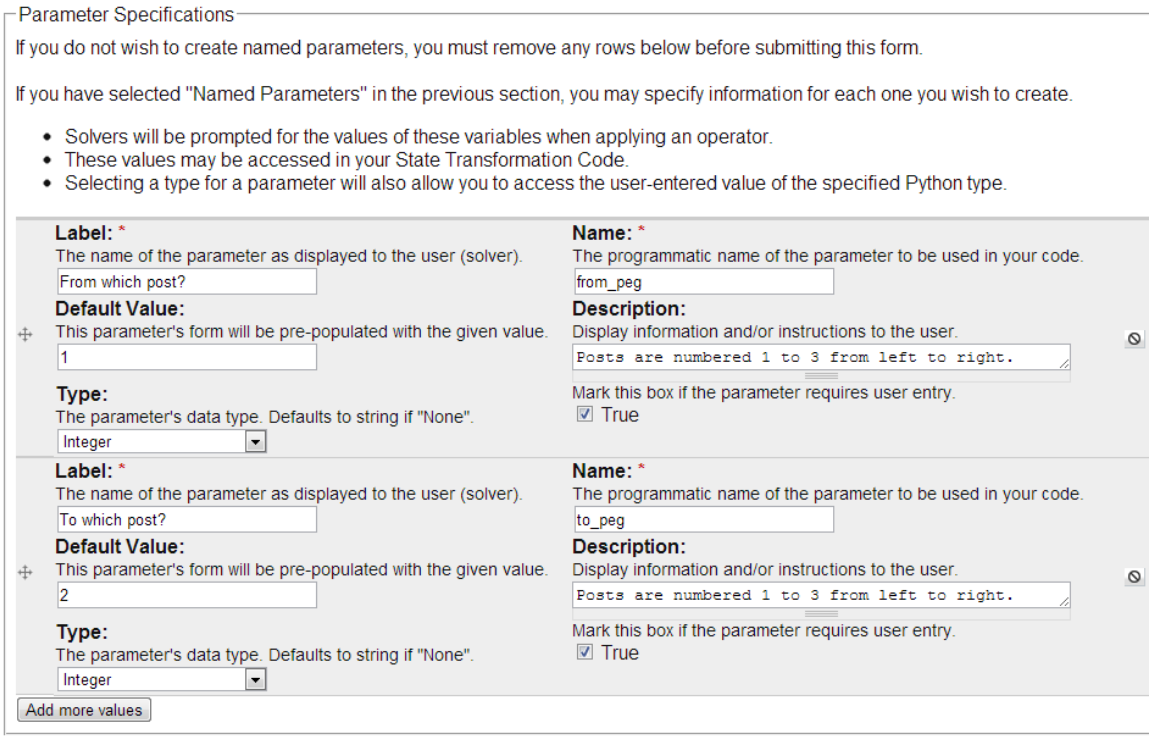


Figure 3.6: Parameter Specifications for operator “Move disk”

After creating filling out this form, we can then go back to the State Transformation area to change our code to accept the parameters. Parameters will be automatically stored in the `P` variable, and we can access them using the names we just created in the Parameter Specifications form.[7]

Listing 3.10: State Transformation code for “Move disk” operator, with named parameters

```
1 move(P['from_peg'],P['to_peg']) # this function is defined in Common Code
```

Then we also will make a similar change in the Precondition section (not shown here). Now if we save the Operator form and go back to our solving session, we will be prompted to enter two parameters when applying the “Move disk” operator, and they will be used to generate the next state.

[7]This web form for this functionality was created by Laura Dong.

It is worth noting that another way to create parameters for this problem would have been to allow solvers to click directly on the posts themselves, within the image visualization. This method is used in problem templates such as CitySim (discussed in Chapter 4). To implement this, the poser selects the parameter type to be "2D point" instead of "Integer", and the corresponding `P` variable will be set to hold an (x,y) pair of the location of a solver's click, in relation to the visualization image's upper-left corner.[8]

Now that we have thoroughly discussed the technical details of how a problem template is created, let's step back and analyze how posing is actually done in practice by looking at examples of templates created by users other than the CoSolve developers.

3.2 Case Studies and Analysis of Problem-Posing

CoSolve contains a variety of different problem templates created by posers. In this section, we will more closely examine problem templates created by actual users on the website. First I will give an overview of the set of templates I looked at, and then I will delve deeper into three problem templates in particular: "The Opportunity Ladder" (Section 3.2.3), "Knight's Tour" (Section 3.2.1) and "Trip(s) to UW Classrooms" (Section 3.2.2).

To analyze CoSolve posing practices, I reviewed problem templates created as student projects in CSE 415, an introductory artificial intelligence class for undergraduate non-computer-science majors at the University of Washington. Students were given an option of several different project types, one of which was to create a problem template in CoSolve to solve a problem of their choice. Students generally had two to three weeks to complete their projects, and could work alone or with a partner. The CoSolve project assignment was to either implement a "standard puzzle and/or game" or a "wicked problem".

In total, there have been 21 such projects over four years (the course is offered once per year in the autumn): three projects in 2012, six projects in 2011, six projects in 2010, and six projects in 2009.

Our data includes all project reports written by the students, and we have the code for all the projects from 2012, 2011 and 2010, but only the full code for one of the projects

[8]This functionality was implemented by Rob Thompson and Laura Dong.

from 2009, for a total of 16 full-code projects. For these 16 projects, the average mean lines of code per project is 210, maximum is 410 lines, minimum is 59 lines, and median is 200 lines of code. Lines of code are calculated by counting all the lines contained in the *poser snippet* boxes of the final Problem Template and Operators. As for the other five projects from 2009, we have all code except operator code for 3 of these projects, for a total of 19 projects from which we have at least partial code, and no CoSolve for code for the last two.

Table 3.1 lists the projects and a short description of each, to give the reader a sense of the scope of these projects. As can be seen from the descriptions, most students chose puzzle-type problems, for example, one-player games, like Sudoku, or two-player games like Checkers. "The Opportunity Ladder" project in 2012 was the only one that attempted to examine a wicked problem. Also, we can see that most projects could be completed with fairly few lines of code, around 200 lines, because CoSolve takes care of a lot of the functionality of running a solving session.

In general, most of projects, 16 out of 19 projects for which we have code data, relied on the Common Code area as a place to put the majority of the code, with most putting reusable Python functions in the field. Of these 16 projects that used Common Code, each project had a median of 7 functions, and mean of 8.45, in the Common Code, and on average, around half (53%) of the total lines of code were Common Code. A couple of students defined new Python classes in the Common Code area to be used by the rest of a template, a behavior that we had not anticipated when designing CoSolve, but which worked well.

The next largest chunk of code was in the Visualization section, with an average of 28.3% of the lines of code being devoted to Visualization. Common Data and State Initialization usually only had one or two lines of code, and the rest went to defining the operators. State Initialization was interesting because we saw a number of different ways of defining a state. Some chose to put nothing at all in the initial state, relying on operators to add state variables as needed. Others defined a lot of constant, static information in State Initialization, rather than putting it in Common Data. Most students defined values directly into the `S` variable, but some called their Common Code functions to initialize values that they would then put into `S`.

There are two basic styles of implementing operators in CoSolve. One is to create many operators to represent all the possibilities for child nodes at each parent node, and the other is to implement just a few operators and then rely on parameters to achieve the breadth of possible children. Most students did the latter; the problem templates had a median of 3 operators, mean of 3.8. The most we saw for a problem was ten operators.

Table 3.1: Description of CSE 415 student CoSolve projects, sorted by term and year. Students could work on their own or in pairs. The number after the project title indicates number of students in that project team. The number of total lines of code used in the problem template is listed at the end of each description; however the ones listed as partial code do not contain the code for the problem's operators.

Project (# Students)	**Description**
Autumn 2009	
8 Puzzle (1)	Sliding tile puzzle; solver slides numbered tiles around on a 3x3 board until the numbers are in order. (partial code only, at least 141 lines of code)
Maze Explorer (2)	Solver must navigate a maze while racing against an AI opponent (412 lines of code)
Rubik's Cube (1)	Solver must complete a Rubik's Cube by aligning each side's colors by turning the cube's faces. (no code)
Smart Scheduler (2)	Helps solvers schedule classes by priority, such that there are no time conflicts, and accounting for prerequisites. Solvers add the classes they want to take, and classes they have taken. (partial code only, at least 369 lines of code)
Sudoku Puzzle Solver (1)	Implementation of 3x3 Sudoku. Student did not finish debugging in time for turn in, and thus was unable to complete this project.
Video Poker (1)	Solvers play hands of poker in each state. Includes AI for calculating best move to make given each hand. (partial code only, at least 641 lines of code)

Continued on next page

Table 3.1 – continued from previous page

Project (# Students)	**Description**
Autumn 2010	
15 Puzzle (1)	4x4 version of the sliding-tile puzzle. (266 lines of code)
Arimaa (1)	Implementation of a recently-invented board game using chess board and pieces but with different rules: pieces are "pushed" or "pulled" rather than captured. (341 lines of code)
Bridges (1)	Solvers collaboratively solve a logic puzzle consisting of building bridges between islands such that the bridges do not overlap and each island contains the specified number of bridges. (322 lines of code)
Magic Squares puzzle (1)	Solvers must place unique numbers on an NxN grid such that each row and column adds up to a sum M. (164 lines of code)
Stratego (1)	Solvers play the Stratego board game in CoSolve by strategically placing hidden pieces on a board. (134 lines of code)
Transistor Schematics (2)	Allows solvers to design circuits. This is an example of a problem template where there is no explicit goal represented within the problem template itself; the solvers decide for themselves what their goal is and whether a goal state has been reached. (139 lines of code)
Autumn 2011	
Checkers (2)	Implementation of Checkers, including an approximate "score" displayed to indicate how many pieces have been captured for each player. (271 lines of code)
Pentomino (1)	Implementation of Pentominoes puzzle, where solvers must fit 5-block "pentomino" pieces such that they cover the board but do not overlap. (206 lines of code)
Schedule Planning (2)	Tool for assisting with scheduling employees for work shifts. Solvers can add employees and constraints to try to find an optimal work schedule. (389 lines of code)
Sudoku (1)	Complete implementation of 9x9 sudoku. (193 lines of code)

Continued on next page

Table 3.1 – continued from previous page

Project (# Students)	**Description**
Traveling Man (1)	Implementation of the Traveling Salesman Problem. Each operator allows the solver to add new cities to their tour. (59 lines of code)
Traveling Salesman (1)	Implementation of the Traveling Salesman Problem. Each operator allows the solver to add new cities to their tour. (60 lines of code)
Autumn 2012	
Knight's Tour (1)	Implementation of the chess problem known as the Knight's Tour, with the addition of "forbidden squares" that the knight should not visit. (96 lines of code)
Trip(s) to UW Classrooms (1)	Solver interactively explores possible shortest paths to different buildings on the University of Washington campus. (98 lines of code)
The Opportunity Ladder (1)	Problem template meant to educate solvers about issues in engineering diversity at the college level. Solvers play a virtual engineer who climbs up a "ladder" based on their answers to multiple choice questions. (217 lines of code)

Now that we've looked at the general scope of problem templates created by these students, let's examine how the process of problem-posing in CoSolve works, by investigating the three projects from 2012 in more detail. In addition to their problem template code and their written project reports, I also have data from briefly interviewing these three students during their problem-posing process.

All three were single student projects, and they were interesting in that their styles represented points along the spectrum of different styles of CoSolve posing. Student A's project is "Knight's Tour (with forbidden squares)" which I will call *"Knight's tour"*. Student B's project is "Trip(s) to UW Classrooms" (*"UW Classrooms"*). Student C's project is "The Opportunity Ladder" (*"Opportunity Ladder"*).

3.2.1 Student A: "Knight's Tour"

Student A's problem was the Knight's Tour[9], which is a chess puzzle in which the solver must find a way to move the knight piece such that it visits every space on the chessboard without ever revisiting a square. (A knight can only move in an L shape, 2 spaces in one direction, and 1 space in the other.) At the instructor's suggestion, to make the problem more interesting, the student added a "forbidden squares" component to the problem–the solver can specify certain spaces that the knight must never visit.

The problem template has only two operators. The "Move Knight" operator takes a string location (such as "e,3" to represent chessboard space *e3*) to specify where to move the knight. There are three possible resulting states, shown in 3.7: the move is not a legal knight move, the space has already been visited, or the space is valid. The other operator, "Forbid Square" allows the solver to specify the forbidden space.

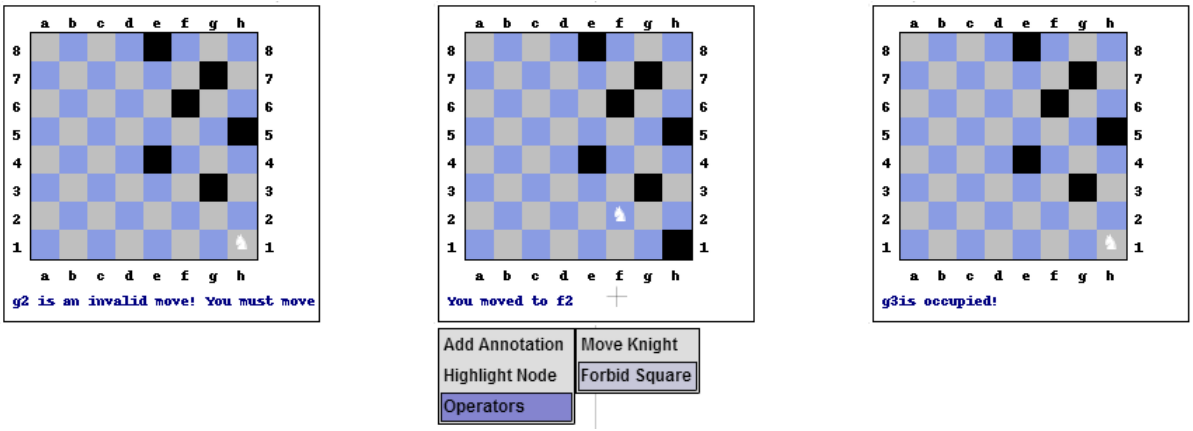


Figure 3.7: Student A's "Knight's Tour." Screenshot of one level of a solving session tree created by Student A. This shows three child nodes of a single parent (not shown) to which the solver applied the "Move Knight" operator three times, from left to right, with a parameter value of `g2`, then `f2`, then `g3`, with three different results. The knight, in its current position, is shown in white, black squares are previously visited squares.

When I asked Student A about his posing process, he said that first of all, he looked

[9]Jordan Atwood is *Student A*, the creator of the Knight's Tour template.

around on the CoSolve site at other problem templates that were similar, and ended up basing his problem template on another problem template, the 8 Queens template (another chess problem). He felt that it would be best to get a visualization working first, so that he could see what he was doing as he made changes to the template, so his first step was to copy of the 8 Queens chess board visualization from that template, and put it into the Visualization of his problem template. We later discussed how he would represent the forbidden squares in his template; at first I assumed he meant how to represent it in the `S` state dictionary, but soon it became obvious that he meant in the visualization. This brought me to an interesting realization that hadn't occurred to me when designing the posing interface: for some posers, the visualization is the way they think about the problem. To them, the visualization *is* the state representation itself and the code simply supports that. This is supported by the fact that he only has two state variables in his state representation: a 2D array representing the board, and a string variable to hold any status messages (e.g. "Illegal move" or "You won" etc.). We will see that this contrasts greatly with the other two templates.

Another interesting thing that came up was Student A's workflow. He primarily edited via the web interface, putting his code directly into the web forms, making changes in the web form, then testing those changes in a solving session, and then going back and making changes in the web form. This, as we will see, contrasts with the other students' workflows. Student A said he preferred his method of working because it was really quick and easy to get the project up and running, getting feedback and seeing the visualization, rather than having to worry about details of setting up a user interface or an executable program. As we had hoped when designing CoSolve, it seems being able to construct something that doesn't cause an error from the start is important for some novice CoSolve users.

Only one function was defined in the Common Code. Student A placed all of the work of his operators in the operators themselves, rather than defining Common Code functions and calling them in the operators' State Transformation area. As seen in Table 3.2, most of the problem template's code was placed directly in the operators' code areas.

Student A did not use preconditions in his operators, and noted in his report that he had wanted to, but felt it would be "unwieldy" do to so because of "how the problem is currently

Knight's Tour	
Number of Operators	2
Number of Common Code functions	1
Number of State Variables at Initialization	2
Lines of Code: Common Data	0
Lines of Code: Common Code	9
Lines of Code: State Initialization	1
Lines of Code: Visualization	47
Lines of Code: Operator: Move Knight	28
Lines of Code: Operator: Forbid Square	11
Total Lines of Code	96

Table 3.2: Student A's "Knight's Tour" Problem Template Data

set up." Indeed, because he had only one operator to move the knight, and the destination space is indicated by parameters, rather than a different operator for each possible move, it would have been impossible to use preconditions, since the "Move" operator always needs to be displayed.

The alternative approach would have been to create operators for every space on the board, e.g. "Move to a1", "Move to a2", etc. and then use preconditions to check whether the operator is legal and hence should be displayed or not. This approach would have required him to create 64 operators, and have the preconditions hide the ones that are currently not valid. Although it is certainly a more unwieldy approach from the posing point of view, it would also be easier for the solver, in that it would have immediately made obvious to the solver which moves were available and hence valid, and which were not. This way a solver could not even attempt to enter an illegal square. These two different approaches demonstrate two fundamental ways of creating CoSolve operators: using parameters to allow solvers a wide range of–possibly invalid–inputs, versus limiting the range with a set

of pre-constructed operators and using preconditions to display the currently relevant ones.

Student A may also have meant that he would have liked to use preconditions to only allow the solver to apply the "Forbid Squares" operator at certain points, such as at the beginning of the solving session. However, to do so would have meant that the template needed extra variables to keep track of when the beginning was. Student B's "UW Classrooms" template, as we we shall see, addresses this issue, and also provides many other illustrative points of contrast when compared to Student A's "Knight's Tour".

3.2.2 Student B: "UW Classrooms"

Student B's project[10] was a shortest-path problem, involving University of Washington buildings. The solver specifies a starting location and a destination location, both on the University of Washington campus, as well as a "maximum distance" the solver wants to travel. Then the solver can select different paths to take from one UW building to another, and the state keeps track of the distance traveled along the path. Student B measured distances between buildings on a campus map and used actual distances in her data, storing all of this in her `S` dictionary.

The UW Classrooms template has three operators. "Start" takes three parameters: a starting building, destination building, and max distance to travel. "Move" takes one parameter: the next building to add to a path. "Backtrace" allows the solver to backtrack along their path by one step–i.e. remove the most recently added building from the path. I asked her why she would implement such an operator, when in fact, simply backtracking up the CoSolve branch would have done the same thing, and she said it was just to add something so her project would be more interesting and have more than just two operators.

Like Student A, Student B put all the work of her operators in the operators themselves, rather than in the Common Code (Code breakdown is listed in Table 3.3). Unlike Student A however, she had nothing at all in her Common Code. As mentioned before, she also had nothing at all in her Common Data; absolutely everything was stored in the `S` dictionary variable.

---

[10]Charliz Burks is *Student B*, the creator of the UW Classrooms template.

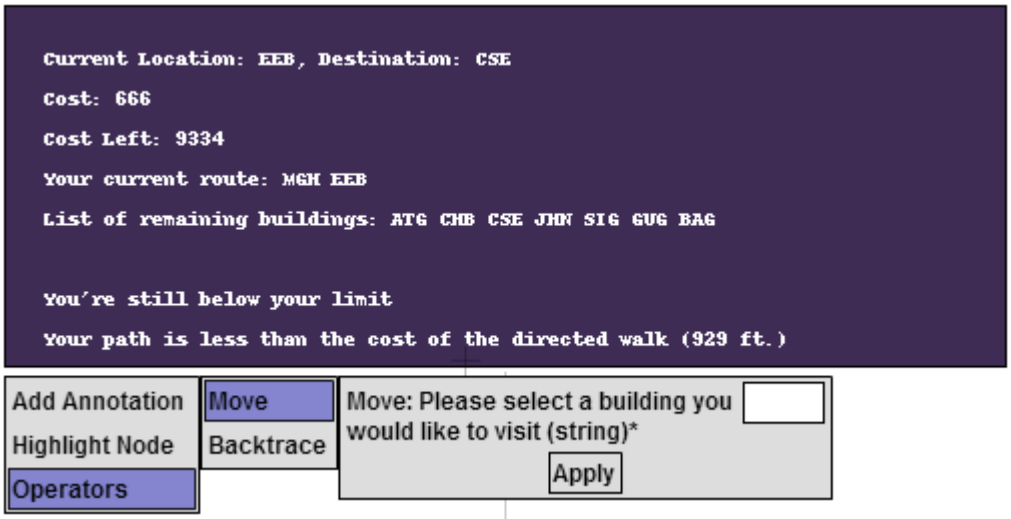


Figure 3.8: Student B's "UW Classrooms". Screenshot of a node in a solving session created by Student B.

With regards to her workflow, she said she looked through different templates first to find examples to draw from, as did Student A. However, when she was actually developing her code, she took a different approach from Student A's. She edited her code in a separate code editor or IDE, such as Eclipse or Notepad++, and then pasted the code into the appropriate places in CoSolve's web forms, and tested her code online. Using a separate editor in this case has the advantage of having a separate saved backup of code files, as well as syntax highlighting and syntax checking without having to run a solving session. When she found bugs, she then went back to the code editor to make changes, and copied over the changes into CoSolve again, going back and forth between her editor and CoSolve's web interface. While Student B was demonstrating her template for a small group of her classmates, one student asked if Student B if she found her project hard to implement. Student B said that it was very easy, and the hardest part was actually in the planning, measuring and inputting all the real building distances into her problem template.

I asked her about how she came up with the `S` state representation. She said she started by adding a lot of variables to her state based on what she thought she'd need, thinking a priori about the buildings in her problem and looking at other problem templates

UW Classrooms	
Number of Operators	3
Number of Common Code functions	0
Number of State Variables at Initialization	5
Lines of Code: Common Data	0
Lines of Code: Common Code	0
Lines of Code: State Initialization	1
Lines of Code: Visualization	65
Lines of Code: Operator: Move	13
Lines of Code: Operator: Start	7
Lines of Code: Operator: Backtrace	12
Total Lines of Code	98

Table 3.3: Student B's "UW Classroom" Problem Template Data

for examples. However, as she continued developing her template, she found she had far more state variables than she needed, and then later cut down on the size of her state representation, removing variables she never ended up using.

This was an interesting contrast to the way Student A developed his state representation, where his focus was on just getting an adequate visualization first. In fact, when I interviewed her, Student B said she had not done anything with her visualization at all (as seen in Figure 3.8, her state nodes are basically text), and that she wanted to get her template working before worrying about the visualization at some point in the future, a completely opposite approach from that of Student A. Student B's focus was completely on the `S` dictionary state representation.

Another contrasting aspect of Student B's problem template is the need to specify or "start" the problem within the solving session itself. In Knight's Tour, the goal is clear from the beginning and at any state, including the root, you can always move the knight or forbid a square. However, there is no goal and nothing to do in "UW Classrooms" until

you have applied the "Start" operator to the root to specify a starting building, destination building, and max distance. This means that at the beginning of every "UW Classrooms" solving session, the solver must start by specifying the problem at the root node: setting the start, destination, and distance. Until that is done, the problem hasn't actually been fully specified enough to begin solving yet.

This is a common problem template design for many CoSolve templates, and is a way to generalize problem-posing, creating a template that is also also a *meta-problem template* for a class of problems. An interesting side effect is that a single solving session tree might actually contain multiple problems. The solver might apply "Start" with the CSE Building as a destination, and then also apply "Start" again with the EE Building as a destination to create a sibling node. This then means that the branches of these two children of the root node are exploring two different, though related, problems.

Another case in which problem specification can change, this time from solving session to solving session, is seen in the "Maze Explorer" problem template (Table 3.1, Autumn 2009). Unlike "Knight's Tour" and "UW Classrooms" and most of the problem templates in this data set, it has 65 lines of code in the State Initalization; other templates have 1 or 2 lines. This is because "Maze Explorer" generates a new random maze every time a new solving session is initialized. In this case, the problem template generates a different problem specification, or maze, for each solving session, but each solving session can only have maze. All these different styles point to how problem-posing, and even what problem specification is, can vary from template to template.

In the next section, we will see how Student C's template demonstrates even more variation in posing style.

3.2.3 Student C: "Opportunity Ladder"

Student C's problem template "Opportunity Ladder" attempts to address diversity in engineering students at a college level[11]. This is the only student problem template to try to tackle a "wicked" problem, but does so by implementing a game to educate users about

[11] Cezanne Camacho is *Student C*, the creator of the Opportunity Ladder template.

issues that can create lack of diversity, rather than directly generating ways to solve the problem. Solvers start by naming and creating a "virtual engineer" who must try to climb the "opportunity ladder" of engineering education. Each point moves the engineer up the ladder. Points are given for both privileges (inherent advantages such as "family history of engineers" or "wealth", generated when the virtual engineer is initially created) and for opportunities that the engineer correctly takes on, in the form of questions the solver correctly answers, such as:

> [Engineer's name] does not want to pursue engineering because all they have heard about it is that the classes are hard and boring, do you:
>
> 1. Encourage [Engineer's name] to look up different engineering programs online to see what activities they entail
> 2. Try to involve [Engineer's name] in research, mentoring, or teaching within an engineering program
> 3. Tell [Engineer's name] that once you get through the introductory classes, the classes start to get really interesting

If the solver answers correctly[12], a colored dot representing the engineer moves up on the ladder, as seen in Figure 3.9. The solver "wins" the game if they can move all the way up the ladder, though this may not be possible if their virtual engineer was not assigned enough privileges at the beginning.

Student C's workflow was completely different from the other two students. When I interviewed Student C, she had mostly completed the entire project–as a separate, runnable, interactive Python script. This version of the project was linear only; the user played straight through from the beginning to end, unlike in a CoSolve solving session where paths can branch. She had even implemented her own text-only, ASCII-art-style "ladder" graphic to be displayed to the user. Student A demoed a fully-working project, but she had not created even an initial skeleton problem template on CoSolve.

[12]The answer according to the problem template is choice "2", with the following explanation: "Academic engagement is crucial for a student to sustain interest in engineering, but not all students get this opportunity."

However, when I asked to look at her code file, I noticed that she had divided up all her code into labeled CoSolve posing sections: "Common Data", "Common Code", "State Initialization", "Visualization", etc. She told me she had looked on CoSolve to see how problem templates were structured, and then mimicked that structure in her own Python script, and that her final step would be to actually put everything into CoSolve. In her written report, she says that having to create operators helped her "simplify [her] game construction and break it down into steps" and that using CoSolve helped her learn how to "divide the information and code of my project." So even though she did not directly use CoSolve at the beginning, CoSolve still had an effect on her code structure.

Student C's CoSolve posing workflow was to have everything running *before* before putting anything into CoSolve; this is a completely different style from both Student A's and Student B's. We can see this reflected in her code breakdown as well (Table 3.4). She has the most state variables (14, vs. Student A's 2 and Student B's 5) and the most lines of Common Code (65% of her code was Common Code, vs. Student A's 0.09% and Student B's 0%)–as it was probably easiest to put most of her Python script directly into that section–and the fewest lines of visualization code. It also may have affected the way she structured her states, as we shall discuss next.

This problem template is notable for its use of modal states. With the other two templates, all states were relatively equal: for Knight's Tour, at any state in the solving session tree, the solver's main option was to move the knight somewhere; for UW Classrooms, at any state other than the root, the solver could only add or remove buildings to the path.

With Opportunity Ladder, however, two different kinds of states can be presented as the solver plays the game, *question* states and *answer status* states (these names are given by me, not Student C). The answer status state nodes, as seen before in Figure 3.9, show whether the answer was correct, and displays a graphic with the engineer's position on the ladder. However, in order to display the next question, the solver must then explicitly apply the "Move" operator, to try to "move" the engineer up the ladder by requesting another question to answer. Applying this "Move" operator generates nodes of *question* state mode, shown in Figure 3.10, which do not display the ladder visualization, but instead display the text of the question, and the only operator available is "Answer." Student C uses boolean

Opportunity Ladder	
Number of Operators	3
Number of Common Code functions	8
Number of State Variables at Initialization	14
Lines of Code: Common Data	2
Lines of Code: Common Code	142
Lines of Code: State Initialization	1
Lines of Code: Visualization	55
Lines of Code: Operator: Name	5
Lines of Code: Operator: Move	5
Lines of Code: Operator: Answer	7
Total Lines of Code	217

Table 3.4: Student C's "Opportunity Ladder" Problem Template Data

state variables `S['isAnswer']` and `S['isQuestioning']` to identify which mode the states are in, and checks these variables in operator precondition code and in the visualization code.

There are two interesting things about this method. First of all, Student C did not have to structure her states this way, and the fact that she did may have to do with her posing workflow. Student C could simply have implemented each state to display both the result of the previous answer and the next question. The only available operator would then always be "Answer." However, because she first developed the project almost like an interactive "text adventure" first, without making use of the tree structure, she saw the project as a linear sequence of events, and hence the different modes make more sense. It is conceivable, though not certain, that if she had developed in CoSolve from the start, she would have visually seen each node in the solving session tree and hence thought of them as more equal from the start. However her method is neither better nor worse.

Secondly, note that this method of using modal states contrasts with our initial state-space search idea of every state representing a possible solution or a valid state for a solution. Here, if a node happens to be in the *question* state mode, it is in a kind of intermediate condition, where you might say our virtual engineer has reached for the next rung but hasn't gotten a hold of it yet, so we can't leave him hanging. Student B's "UW Classrooms" meta-problem template could also be said to have a mode, to a lesser extent, in its "start" root state. "Opportunity Ladder" also has a start state to name the engineer and assign inherent privileges, but takes it a step further with its two alternating modes.

This style is actually not uncommon in other problem templates on CoSolve. It is most commonly seen in multi-player game problem templates, for example, "Defeat the Giant"[13], which alternates between different players' turns (Healer, Wizard, Archer, etc.), affecting the types of operators available, e.g. the "Heal" operators can only be applied by the Healer, on the Healer's turn, and is not available on anyone else's turn.

3.2.4 Summary

From this set of problem templates, we can see that there is great variation in posing behavior, both in the posing style and workflow style of the poser and in the structure of a problem template. Some posers, like Student A, choose to develop the visualization first, and the rest of their code supports the visualization of their states. Others, like Student B and Student C, choose to work on the state representation first, and create visualizations as a final step.

In terms of workflow, there is a dichotomy between working directly within CoSolve versus working in separate development environment or text editor. Student A stayed almost completely within CoSolve's web posing interface, and Student C created a complete working program outside of CoSolve, and then imported it into CoSolve at the end of the posing process, while Student B did a hybrid of the two methods, to take advantage of both immediate feedback from CoSolve, and syntax highlighting and a conventional text editing environment within Eclipse and Notepad++.

---

[13]This template is not part of the CSE 415 projects, but was created by OLE undergraduate researcher Yifan Zhang. Template can be found at http://cosolve.cs.washington.edu/problem/defeat-giant

As CoSolve developers, our knowledge of these disparate workflow processes can inform the development of new, easier-to-use posing interfaces, that better incorporate the benefits of an Integrated Development Environment (IDE) like Eclipse. One idea might be a CoSolve posing plug-in for Eclipse, that allows posers to run solving sessions and debug directly within Eclipse. This would be an alternative to the regular web interface, which would still be maintained for novice posers, so that they don't have to setup and learn to use Eclipse. At the same time, posers who are already programmers can use a development environment they may be more familiar with. Currently, problem templates can be exported from CoSolve as a single CoSolve problem template file, which is a valid Python script that can then be loaded into Eclipse or any text editor, although there is not yet a way to automatically load the file back into CoSolve.

We also saw differences in how posers chose to structure their problem templates. We saw differences in how state variables were used, and how much focus was placed on visualization. There were differences in the way different CoSolve code areas were used: student B put all her operator code within the operator State Transformation area itself, whereas student C placed the work of the operators as functions defined in Common Code, and simply calls the relevant function in the operator's State Transformation area.

We identified two opposing ways of validating states; either by limiting operators with preconditions and hence never allowing an invalid child to be created, or allowing a wide range of possible children through parameters and then determining a child state's validity when the child is created. We also found some common patterns, such as meta-problem templates like "UW Classrooms", and templates whose problem specification is unique to each solving session, like the "Maze Explorer", and finally, templates with modal states. The existence of all these different methods demonstrate the diverse range of posing styles and templates that CoSolve can support.

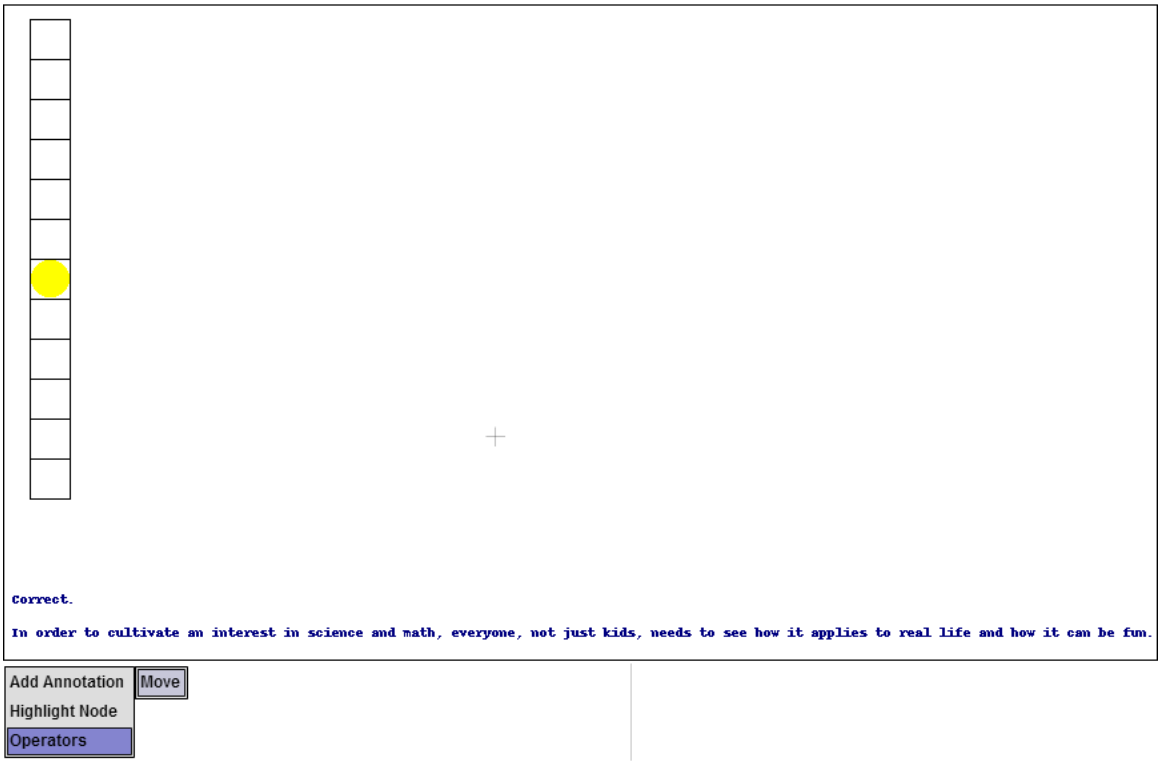


Figure 3.9: A node from a solving session for Student C's "Opportunity Ladder," showing an example of an answer status state. This visualization's text on the bottom shows that the solver's answer was correct, and the yellow dot on the ladder has moved up a rung. The node visualization also gives an explanation of the answer at the bottom. Notice that the only possible next operator is "Move" operator, which gives the solver an opportunity to move up the ladder by presenting them with another question as the next node.

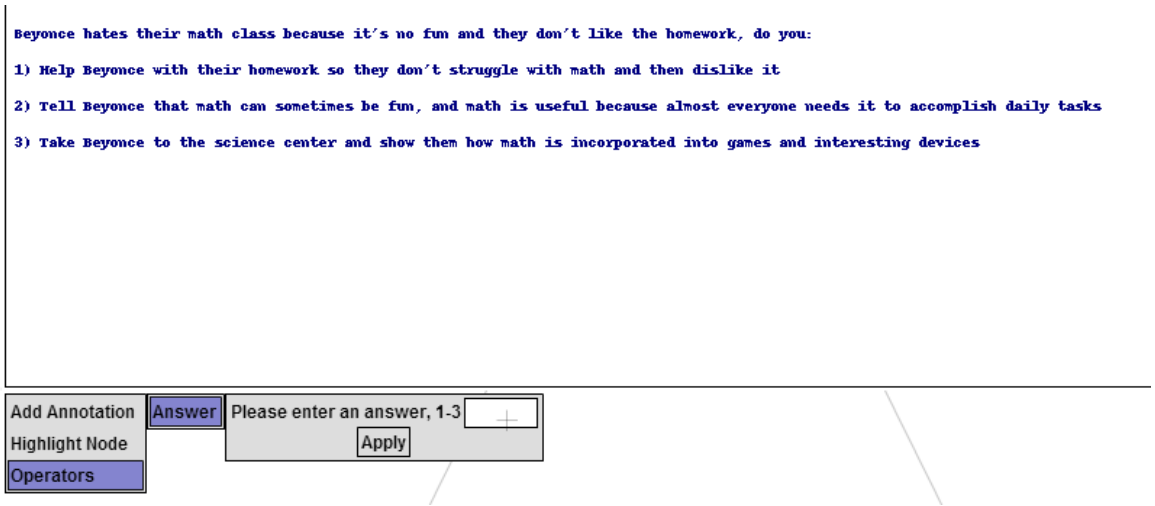


Figure 3.10: A solving session node from Student C's "Opportunity Ladder," showing an example of a question state. Solver must select the correct answer and enter it as a parameter to the "Answer" operator. (The font is slightly hard to read because this is a direct screenshot and the student's visualization code used a small font.)

3.3 Discussion of the Scope of Problem Posing in CoSolve

Now that we have reviewed several different CoSolve problem templates, let us consider the nature of problem-posing in CoSolve. What kinds of problems are best suited for CoSolve? Are there problems that cannot be posed in CoSolve? How do we go from a problem idea to a fully-implemented CoSolve problem template?

The first thing to consider is what our goals are in posing a problem. The obvious answer is to solve the problem: find a goal state whose state variables satisfy some criteria. This is easy to do when the problem is a logic puzzle or straightforward constraint-satisfaction problem, easily divided into tidy, little variables. However, not all problems have an obvious solution, e.g. design or engineering problems, social or public policy problems, the problem of deciding what to have for dinner–does that mean we should not pose the problem to begin with? What other advantages might posing have?

For one, even if the problem is a "wicked" problem [51], and CoSolve solvers may never find the one solution "goal state" they seek, CoSolve can still provide a medium for communication between interested parties. A CoSolve problem template can generate solving sessions to serve as a brainstorming or discussion tool for examining different solutions. Solvers may want to use the template to generate possible solution "candidates" for comparison, to engage others in conversation. Student C's "Opportunity Ladder" used a problem template as a way to spread awareness of diversity issues in engineering education. Problem templates can be used for more than just solving; problem templates can be a channel for dialogue, either between poser and solver, or between solvers.

Another goal for problem-posing is education: to teach students how to organize and structure a problem before solving it, the "devising a plan" step of Polya's four-step problem solving process [52]. Many of the CSE 415 students mentioned in their reports that the project helped them learn how to break down a problem.

Hence, the "solvability" of a problem is not necessarily a factor in whether or not a problem can or should be posed in CoSolve. There may be merit in the very act of problem posing itself. So, instead of deciding whether or not it is possible for a problem to be posed in CoSolve, let's examine how readily a problem can be posed in CoSolve, i.e. the

CoSolve *poseability* of a problem. We will define the *poseability* of a problem as how *directly translatable* into CoSolve that problem is, in other words, how easy it is to create a problem template that faithfully represents the problem, without overly distorting or simplifying the dimensions of that problem.

The main factor that affects the poseability of a problem is the ease with which the problem can be represented by a set of state variables, and how well-defined those variables are. How well-defined the variables are affects not only whether they can be stored computationally, but more importantly, whether operators can be easily written to transform them. On one end of the spectrum are the *very poseable* problems, those with a well-defined, limited set of variables. These are the kinds of problems from classical AI theory that can be represented by state-space search, like chess. The state variables are the pieces and their locations on the board. The operators are the allowed movements of each piece. To use Jonassen's typology of problems [53], *very poseable* problems of the "logical problem", "algorithm" and "rule-using problem" types, which are basically all problem types that either have a procedural set of rules or algorithmic set of steps for solving, or involve constraint-satisfaction over a known, limited number of variables. The problem template's state variables are part of the very definition of the actual problem, and the problem does not exist outside these variables.

On the other end are the *least poseable* problems. These are the wicked problems, the ones for which it is not at all clear how to generate all the possible state variables, and even if we could list them, they might be so ill-defined it may be meaningless to try and apply operators to them. Social, political, environmental and economic problems are the prime examples of these. There is no way we can model all the forces in the world that cause societal wicked problems such as poverty, war, disease or famine, and so we cannot claim to create state variables that capture all such forces. But even for more specific problems on a smaller scale, this can be difficult to do. For example, there is also no way to know all the factors that make a business marketing campaign successful. We might think a Super Bowl commercial that is funny and eye-catching will do the trick, and we may attempt to create state variables to represent that, but we may not have been able to anticipate the fact that a rival company happened to come up with an even better, more memorable commercial

that aired right after ours. These are all examples of problems for which it is not possible to define state variables that would satisfy all stakeholders based on our current knowledge of the domain, but there are still ways to pose these problems, albeit incompletely. We could still create state variables that contain whatever variables we can think of, such as "chance that rival brand will also air a commercial and it will be more popular than ours" and define them in some artificial or simulated way, say whether their ad agency is more expensive than ours. But we could never account for all the possibilities. Problems of the *Least poseable* type generally require significant modification or simplification of the actual problem to pose it as a CoSolve template. The last section of this chapter further discusses some more practical ways to turn a wicked problem into a CoSolve template.

Then there are the problems in the middle of the spectrum. These are templates for which the problem is not defined in terms of variables and operators, but, given some thought, can be represented as such without too much difficulty. For example, consider the problem of writing a poem expressing your feelings about the beautiful weather today. At first glance, it may not seem like a problem of the cold-hard-logic variety–the chess puzzles and the resource schedulers. But a poetry-writing problem template could be easily posed by having either a single state variable that holds the poem's text, either as a single string, or a list of strings, each representing a single line of poetry. Then the "add line" or "edit line" operators would allow the solver to enter or edit any given line of text. Using this representation, in the resulting solving session tree, each state would represent a poem, or some revision of a poem, for the solver to explore. There could even be other operators; perhaps there is an operator to add a random line or word to the poem, which the solver can then keep or edit. There are many possibilities for other kinds of creative, poem-writing operators that could be added to the problem as well, providing for more variation in how a poser might create such a problem template. This is in contrast to the *very poseable* class of problems, where the operators are more limited or more clearly defined for a given problem.

Or let's say a company is redesigning its logo, and wants a problem template that will help them pick a color scheme. The Color Swatch template[14] in CoSolve, displays a set

[14]http://cosolve.cs.washington.edu/problem/color-swatch

of colors in each state, and the state variables are a list of color values. Operators like "complementary colors" or "analogous colors" can be applied to the set of colors to change them all in a coherent, matching fashion. Solvers can also apply operators to directly change the HSV or RGB values. The solving session tree will contain nodes showing sets of many different colors, and allow the logo team to explore and talk about different options, and remember options they had come up with previously, to find candidate color schemes they like. This is a fairly poseable problem, we can clearly define all needed state variables–they are just the list of colors–and although there are no clearly defined rules about what the operators should be, we can come up with them easily, and think of more to add as needed.

Notice also that, although there is not necessarily one correct "goal state" in either of these templates, the existence of a goal state does not make a problem more or less poseable. The fact that the problem is not initially presented as a set of variables and operators, and hence the poser had to decide how to do so, is what makes it less poseable. If we used a chess problem template and, instead of a winning game determining the goal state, decided our goal was to find the most visually pleasing, but still valid, board state, our goal is now subjective, but the problem itself is still *very poseable*, as the operators and state are fully determined by the problem.

Another thing to consider is, while CoSolve *could* potentially be used to pose many different types of problems, which *problems* are best suited for CoSolve? When should you pose a problem in CoSolve? A problem may be very poseable, but it may not take advantage of CoSolve's strengths. Solving checkers in CoSolve is all well and good, but where CoSolve really shines is when the solution requires a human element. We could implement a traditional state-space search tree to solve many of the *very posable* problems, but for other problems, human intuition or creativity is needed.

Usually, there are two ways in which the human factor is most useful in CoSolve solving. The first is for tree pruning. In classical state-space search, computable heuristics are used for pruning. However, some problems are difficult to write heuristics for, while a simple glance from a human can allow us to say, no, we definitely don't want to apply the "place neon flamingo floor lamp" operator near the polka-dot green wall in our interior design problem. Without human intervention, an AI computer agent might not know to avoid

going down that branch of the solving session tree.

The second is for identifying whether or not a goal has been reached. Not all problems may have computationally-determinable goals. For instance, deciding which painting is more aesthetically appealing is subjective, human decision, and so if we had a template for generating pleasing images, only a human can determine if the goal has been reached. Note again that a problem template in which the goal state is not easily identifiable does not make a template less poseable. In fact, it makes the template better suited for CoSolve than for a traditional state-space search program.

CoSolve is also useful for problems that may require a "social" solution. As the *Co-* in CoSolve reminds us, CoSolve is a collaborative solving tool. Many solvers can work on a single problem at one time, making posing in CoSolve more worthwhile if the solution can benefit from solver collaboration. This is especially beneficial for problems on the wicked *least poseable* part of the spectrum–the more possibilities there are, the more people might needed to explore them.

3.3.1 Transforming real-world problems into CoSolve problem templates

How, then, can we turn the wicked-style problems on the *least poseable* end into CoSolve problem templates?

First of all, understanding our goals for posing a wicked problem in CoSolve will inform how it should be transformed into a problem template. Is it to generate feedback from our stakeholders, in the form of candidate solutions? Or is it to raise awareness, like "Opportunity Ladder"? Or are we trying to use solving session nodes as reference points for dialogue? Different goals may be best served by different styles of problem templates.

For example, with the community playground design problem from the beginning of the chapter, perhaps the goal is to allow the neighbors to express what they would like in a playground design, to "generate feedback." Then the template could focus on visualizing the different design layouts, and allowing solvers to add new elements and colors to the design, and allowing solvers to add notes on what they value in the design ("I like having open grass space" or "I think safety is most important"). On the other hand, if the goal of

the template is to raise awareness–spread the word about the new playground, or help the neighbors understand the conflict between budget, physical space, aesthetics, etc., then the template could be implemented more as a simulation, or perhaps a game, stressing budget constraints and limited space. The playground example is simple, but for more complex templates, the simulation would require more computation and calculation, making it more different from the "generate feedback" template. Finally, perhaps the solving session is meant to be used as an aid during neighborhood meetings, in which case we may want to make it easy for our solvers to see the pros and cons of each state, and perhaps visualize budget numbers and predicted traffic flow effects in the neighborhood, etc.

Finally, there are the several, simple "tactics" we can use to make posing a wicked problem easier, though they all diminish the problem somewhat, in that they don't take away the wickedness of the problem, but merely tries to work around it. The first is to tackle a subdomain of the problem that might be more solvable. For example, in tackling issues with diminishing populations of salmon, one of our undergraduate researchers created a problem template that simulated only a smaller problem–balancing water across a dam for salmon to swim upstream–rather than modeling all the possible environmental factors involved[15].

Another tactic is to simplify the problem enough to simulate the entire thing. This is the case with ClimateSim[16], which simulates energy generation and usage in a residential area, and CitySim[17], which simulates city planning. Both of these breakdown their problems into a fixed, manageable set of state variables, and the operators implemented only deal with those variables.

One last tactic to deal with a limited set of state variables in a CoSolve problem template is to create and use an operator that adds arbitrary state variables to the `S` state dictionary. Some existing templates already start with no entries in the initial `S` state dictionary; solvers

---

[15]Created by Katherine Hulsman.
URL: http://cosolve.cs.washington.edu/problem/salmon-game-jr

[16]Created by Galen Knapp.
URL: http://cosolve.cs.washington.edu/problem/climatesim

[17]Created by Tyler Robison and Sandra B. Fan.
URL: http://cosolve.cs.washington.edu/problem/citysim-study-version

apply operators to add variables to the `S` dictionary during the solving session. The variables of a wicked problem are difficult to impossible to nail down, meaning solvers inevitably have to add more state variables, so an operator for adding arbitrary variables may be useful. The operator can perhaps add a special prefix to these variables' names to distinguish them from any pre-existing variables in the template code. Other operators in the template could then be written in such a way so as to iterate through these prefixed state variables and transform their values as needed. Alternatively, solvers could act as posers and create new operators to deal with these new state variables, as operators can always be created and added to a template as long as the template's original poser set the template permissions to allow edits by others. This also means that new operators can be added to transform the old state variables in new ways as well, which would also be helpful in tackling the dynamic nature of wicked problems. By definition, wicked problems have no enumerable or exhaustively describable set of potential solutions [51], but we can keep adding new state variables and operators to generate evermore new solutions.

Conclusion

In this chapter, I have presented CoSolve's problem-posing process. We began with a detailed explanation of the technical aspects of creating a CoSolve problem template. Then we studied several existing problem templates, identifying different workflows, posing styles, and template structures utilized by actual CoSolve posers. Finally, we ended with a theoretical discussion of the nature of problem posing in CoSolve, and ideas for creating problem templates for wicked problems.

Now that we have dissected how a problem is posed in CoSolve, let's see how the problem templates can be turned into solutions in the CoSolve problem-solving process.

Chapter 4

PROBLEM SOLVING IN COSOLVE

We have just discussed CoSolve *problem posing* in great depth, so now let's turn our attention to CoSolve *problem solving*. Problem solving commonly refers to the entire process of solving a problem, from problem formulation to implementing a final solution. However, in terms of CoSolve, this process is represented by two different phases. The first phase is *problem posing*, covered in the previous chapter, which can be thought of as the first two steps of Polya's Four Steps of problem solving [52], "understanding the problem" and "devising a plan." The third and fourth steps, "carrying out the plan" and "looking back" are done in the *problem-solving* phase of CoSolve. Solvers instantiate *solving sessions* from existing problem templates created by posers, and in these solving sessions, solvers construct, explore and evaluate potential solutions.

This type of solving is at the center of CoSolve, and CoSolve was designed first and foremost as a platform for solvers. CoSolve is unique in its implementation of a user-manipulable, interactive, collaborative state-space-search structure for problem solving. In this chapter, we will study how solvers use CoSolve to explore solutions.

In the following section, we will discuss issues surrounding problem solving in a state-space environment, and describe the process of creating and using a solving session in CoSolve (Section 4.1). Then in Section 4.2, we will present results of a user study conducted with CoSolve to examine how solvers actually use CoSolve. In that section, we also describe the use of the CoSolve Consultant, a tool created by Robison [54] which provides extra meta-cognitive information to the solvers to help them understand the dynamics of their problem-solving process.

We would also like to explore the difference between CoSolve posing and CoSolve solving. The distinction between whether a user is acting as a "solver" versus a "poser" is a blurry one because of the fluidity of posing and solving–as mentioned in Chapter 3 on problem

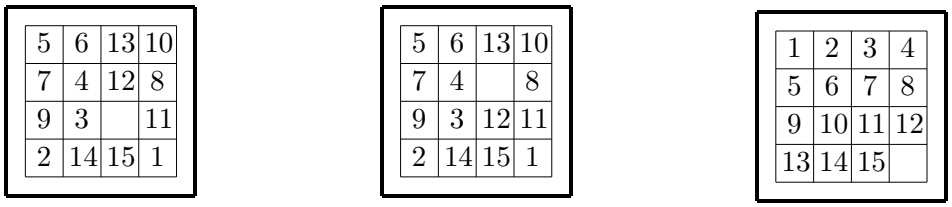
posing, the line between the two is not set in stone, and there are ways to create "meta" problem templates that allow solvers to create their own problem specifications. We shall see an example in Section 4.3 of how these two activities can be thought of as part of the same problem-solving activity; in that section, we investigate the use of CoSolve for an open-ended problem: educational game design.

4.1 Description of the Problem-Solving Process in CoSolve

4.1.1 State-Space Search and CoSolve's Solving Process

As described in Section 1.1, CoSolve uses the theory of state-space search from classical AI as a model for its solving paradigm [55]. In state-space search, the process of finding the solution to a problem is represented as the execution of a tree search algorithm that searches through possible states of a problem to find the desired goal state. The root of the search tree represents the initial state of the problem. At each state, one or more operators can be applied, and these represent all possible "steps" that can be taken toward a solution at that point. Each operator that is applied generates a new child state, to which more operators can be applied, and more children generated. Eventually, an entire tree of possible states is created. To solve the problem, the search algorithm must traverse down the tree to find a leaf node that represents the "goal" state–the solution to the problem.

Let's take the example of the 15-Puzzle problem, seen in Figure 4.1. The goal of this classic sliding puzzle is to slide 15 numbered tiles on a 4x4 frame to place the numbers in order. The state at the root node would represent some starting version of the puzzle, with the numbers out of order, as in Figure 4.1a. An example of applying an operator to this root state would be sliding a single tile, say the 12 tile, into the empty slot. Then the new state generated would be the one in Figure 4.1b, with the "12" tile moved down; in the tree, this state would be a child node of the root node. Each additional operator application generates a new state, and at any point the solver can backtrack to a parent and create a sibling child, and thus a new branch to the tree. In this way, an entire tree of possible states can be created. When a state is achieved where all the numbered tiles are in order (Figure 4.1c), then a *goal state* of our problem has been reached, and the branch from the root to



(a) Example of an unsolved, starting state

(b) Child state after sliding the 12 tile down

(c) Solved state

Figure 4.1: Three example possible states of the 15-Puzzle.

the goal state represents the *solution path*.

Solving sessions in CoSolve are represented by these state-space trees. CoSolve's major feature is that, instead of using an artificial intelligence algorithm or heuristic to search this tree, human intelligence and intuition is used. Rather than the tree being simply an internal data structure for representing the state space, users can view and manipulate the tree directly to search for solutions. Instead of relying on AI heuristics to figure out where to prune large search trees, human users themselves decide which branches to explore, i.e. which operators to apply.

Figure 4.2 shows an example of a tree from an actual CoSolve solving session for the Minesweeper problem[1], with callouts to explain where different parts of the tree are. The initial root node, indicated by the top callout box, shows the initial state of the problem, which is with all the mines still hidden. Each of the child states branching off of the root shows a possible next state, where each of the solvers who created those child states uncovered a cell in a different location on the board. If the tree were fully explored, there would be 81 children from the root, with every possible mine location explored, because it is a 9x9 minesweeper board.

[1]Minesweeper problem template created by Yifan Zhang. This is an implementation of the built-in Minesweeper computer game that those who have used the Windows operating system may be familiar with. The game consists of an NxN grid with some number of mines randomly hidden behind each cell. Players click on cells one at a time to uncover what is hidden behind the cell. If a mine is revealed, the player loses the game. If it is not a mine, the cell reveals information about how many mines are adjacent to that cell. The goal is to uncover all non-mine cells, without ever clicking on a mine.

Notice however that there aren't 81 child nodes of the root, because the solvers did not choose to create all those possible states. Whereas a computer algorithm or automated solving agent[2] might try all possibilities, only states that human solvers explicitly choose to explore are generated.

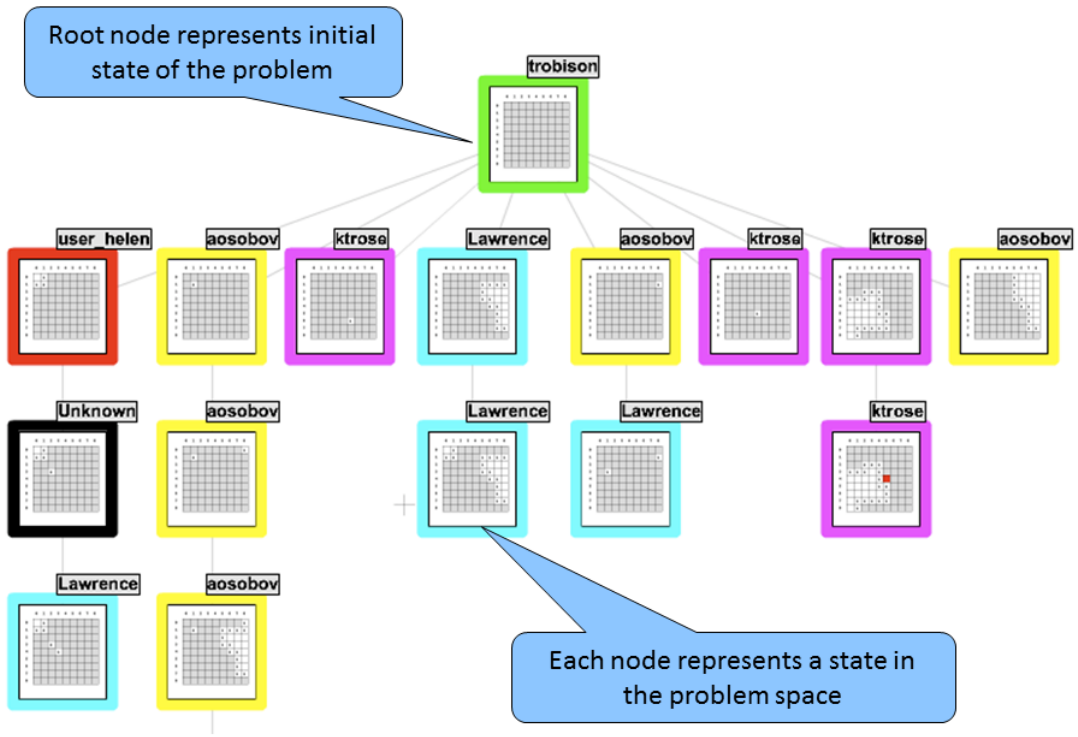


Figure 4.2: Example solving session tree for Minesweeper (the computer game) problem, with callouts to show different parts of the tree. The small gray box above each node is the username of the solver who created that node. Notice that the space is not fully explored, i.e. the root does not have 81 children, because only cells that solvers chose to explore are generated.

[2]An automated solving agent for CoSolve has been created by Robert Thompson, which allows humans to make use of AI agents for solving parts of the tree, combining the best of both human and artificial intelligence.

Also, out of these 81 child nodes, if any of these contained states where the player revealed a mine, and hence ended and lost the game, these nodes would be examples of terminal nodes that are not goal states. As for the rest of the child nodes, 80 more operators could be applied to each of them–theoretically. Of course, if the solvers were paying attention to the other children that were generated, they would know which cells revealed a mine, and not repeat those states. However, in the CoSolve model, the path to a solution matters, and two states that look identical may have been generated by two very different paths–both in terms of which operators were applied, and the order in which they were applied–and so they are displayed as separate branches in CoSolve. To take this a step further, CoSolve also does not collapse two branches even if the order of operations was identical, because they were created at different times, perhaps by different people, and represent a record of the exploration process, rather than strictly representing the problem space of possible states as a traditional state-space tree would.

Let's examine these issues more fully in an example of a CoSolve solving session for Towers of Hanoi. Figure 4.3 shows a CoSolve solving session tree for the Towers of Hanoi problem, the rules for which were described in Section 1.1. The nodes have been numbered by level, and lettered from left to right, for the purposes of this example. The root, node 1, represents the initial state of the problem, which is with all three disks on the first post.

At this point, a solver, Ann, looks at the root and realizes there are only two options for next moves. The first is to move the smallest disk to the second peg, the other is to move the smallest disk to the third peg. She is unsure which would be the better option, so she first applies the "Move smallest disk to second peg" operator. This generates a new child, node 2a. Then she goes back to the root and applies another operator, and this generates a new child, node 2b. Now she has branched from the root, creating two different *branches* beneath it. At this point, she can choose the node from one of these two branches to continue her exploration.

Let's say she thinks node 2a is more promising, so Ann chooses that first. Here, she has three options: she can move the smallest disk back onto the first peg. This would generate a new state exactly like the root's state, except it would be a child of node 2a. Or she could move either the smallest or the medium disk to the third peg. Let's say that she does both,

generating states at nodes 3a and 3b. Now, node 3b contains a state the same as node 2b. However, these two states, while equivalent, were created through different exploration paths, so CoSolve considers them different, and shows them as two different nodes in the tree.

This was a conscious decision we made in our design of CoSolve. Since CoSolve's focus is on the problem-solving process, we felt that we should not automatically merge or collapse parts of the tree that may look equivalent, because solvers may have had different reasons for creating them. One possibility for future work, however, could include functionality to alert solvers that a similar-looking state has already been created elsewhere in the tree. This concept extends to goal states as well. Some problems may have different goal states–e.g. the final state for a winning chess board looks different from game to game. Other problems have the same goal state, but the paths there will differ.

Let's refer again to Figure 4.3. Let's say a teammate, Bob, now joins Ann in the solving session, and upon inspecting the tree at the state at which we left Ann after creating nodes 3a and 3b, decides that he would like to continue working starting at node 2b. He applies operators twice, creating nodes 3c and 3d, and then continues working on 3c. They both work together in parallel, each on the branches that they have created, and both arrive at a goal state: all three disks on the third peg. These are in the leaf nodes at the bottom of the tree, nodes 9a and 8b. However, they took different paths to arrive at the same state, and so these could really be thought of as separate solutions to the same problem. Knowing just the starting state and the goal state of Towers of Hanoi does not tell an observer anything about how to solve a problem. It's the path to the solution that counts.

There are several benefits to using a tree-based visualization such as CoSolve's for collaborative problem-solving. The first is that there is increased transparency in the design and problem-solving process. Since CoSolve provides a record of every state created, no matter how or when, solvers can visualize their own thought process, as well as others' thought processes, whereas in many problem solving or design tools, only the final draft or solution, or a few intermediate versions of a solution or design, are shown. Other ideas or implicit steps along the way are lost. CoSolve makes these steps visible to all users, allowing users to learn from each other they may otherwise not have been able to see, and to understand

more about their own solving. For example, one solver might discover that she searches deeply down a branch, but doesn't try a lot of different options. Or, another user may find that his solving style involves spending time examining many options (branching, creating child states) but less time going all the way through to a solution. By providing a structured space for solving, CoSolve helps solvers better understand the process of solving itself.

CoSolve also makes it easier for solvers to collaborate. If users can casually see what others are working on, there are more opportunities for serendipitous cooperation. They can easily share information on their progress without having to explicitly and actively show others everything they have done. The branching tree structure helps solvers keep track of threads of ideas, and empty areas in the tree visually suggest areas for further work. It also gives solvers a shared context and reference points for communication–they can easily refer to different nodes to illustrate ideas they may want to share with others. Of course, there are many collaboration tools that allow users to share a workspace, including commercial products such as Google Documents or Basecamp, but most are not specific to problem solving.

Additionally, many of these other tools do not include meaningful computation[3]. Having a structured framework for collaboration makes it easier to incorporate human-computer collaboration. As mentioned earlier, Robert Thompson has developed an AI computer agent for CoSolve that human solvers can use. These solvers specify which parts of the tree they wish to explore, and the AI agent can then to automatically generate and evaluate those parts of the tree. Then the human solvers can review the results and select the ones they want to include in the remainder of the solving session.

Now that we've looked at the theory and motivation behind using state-space search in CoSolve, let's examine CoSolve's solving session user interface.

---

[3]By "meaningful computation," we mean possibly intensive computing of object transformations and the like, as opposed to systems that primarily support storing data and files for collaboration, or systems whose main purpose is to provide a means for communication.

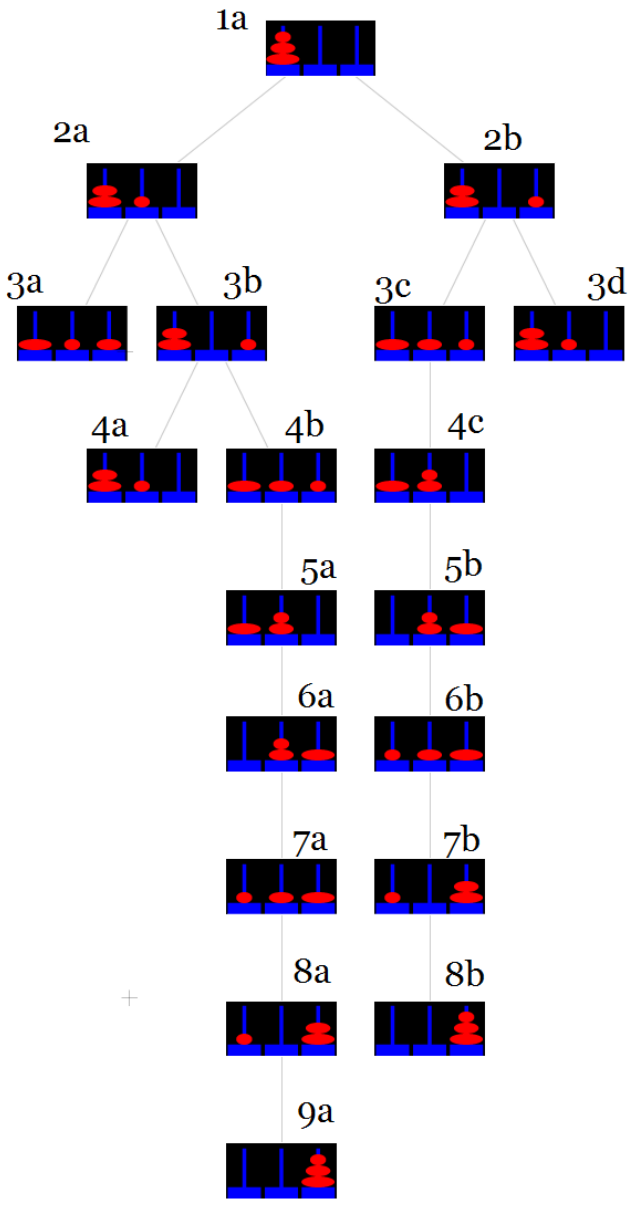


Figure 4.3: A CoSolve solving session tree for Towers of Hanoi. Nodes are labeled. Number indicates level, letter indicates position from left to right. The initial state is at the root, node 1a. Two goal states were reached, at nodes 9a and 8b. Nodes 3a, 4a and 3d are non-goal leaf nodes, indicating where solvers abandoned a solution path as unpromising, and decided to explore elsewhere first.

4.1.2 CoSolve's Solving Session User Interface

CoSolve has multiple solving user interfaces that all connect to the same solving engine and database. The very first, original interface that we developed was a static HTML webpage solving interface. Later, other interfaces were created by our research group, including an AJAX client, a Java client and a Flash client. All of these clients connect to CoSolve via an API that we implemented as part of CoSolve. More on this API and on these other user interfaces will be discussed in the chapter on implementation, Chapter 6. For the remainder of this current chapter, we will refer only to the Flash client[4], which is the most commonly used CoSolve solving interface, and is at present the default interface.

Creating a new Solving Session

To start a new solving session, a solver accesses the CoSolve website and looks for a problem template for which the solver would like to instantiate a solving session. A solver can instantiate a solving session for any problem template that she has created as a poser, or for any templates that other posers have made available to others. On the problem template's webpage, there is a button labeled "Create new solving session for this problem." Clicking this button will bring the solver to the "Create Problem Solving Session" web form. This form can be seen in Figure 4.4.

The session creator (the solver who creates this solving session) can then name the solving session and enter an optional text description of the session. There are also a handful of other settings the session creator can select, including whether or not to show the CoSolve Consultant, whether the solving session is public, private, or viewable by some group of users she selects, etc. These options can be edited at any point later by the session creator, or by others if she so chooses.

Upon creating and saving a new session, the session creator is brought to a web page that displays the Flash client view of the session (Figure 4.5).

At this point, other solvers can view and participate in the solving session by visiting

[4]While I advised on the design and features of the client, the code was implemented primarily by Chris Brenan and Robert Thompson.

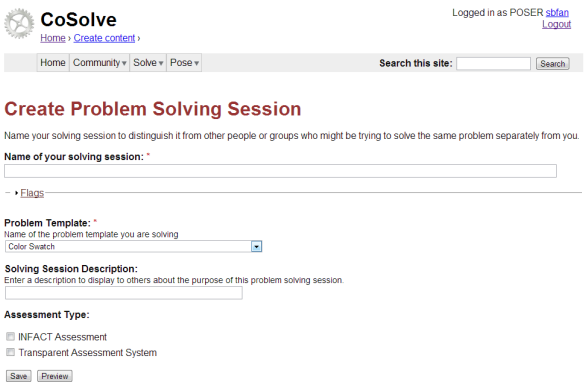


Figure 4.4: Example of a simple version of the Solving Session creation web form, with some limited options shown. Depending on user permissions, some session creators have more options for solving sessions.

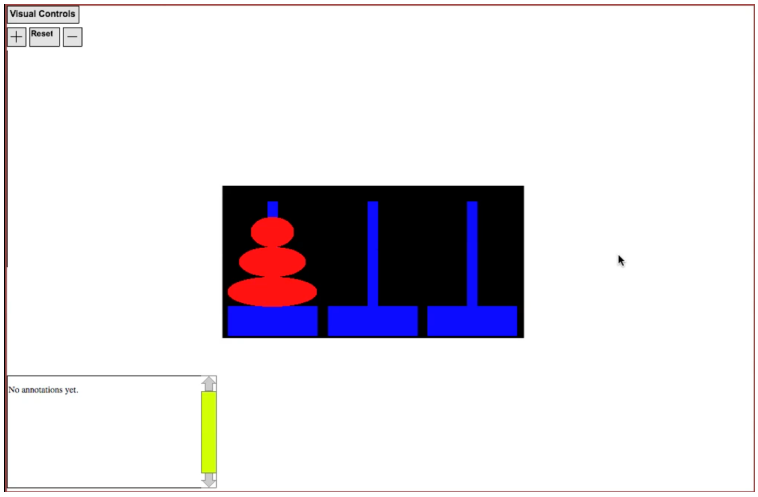


Figure 4.5: A newly created Towers of Hanoi solving session in the Flash client UI. Currently this contains only one node, the root, which is displayed in the center of the screen. This root node represents the starting state of the Towers of Hanoi problem. User interface controls are in the upper left. The annotations list, currently empty, is in the bottom left. To navigate the tree, users can zoom in and out, and can pan across the tree as it grows by clicking and dragging the viewport.

this webpage. Unless the session creator explicitly restricts access to a specific group of users, there is really no sense of "joining" a solving session in CoSolve, there isn't a list of people who have "joined" a session. Solving sessions are open to anyone who can access them. If a session is public, anyone can view it, apply operators or create annotations on nodes.

When first created, the solving session has only the root node, displayed in the center of the screen (Figure 4.5). On the left of the screen are the solving session controls. The first of these is the Visual Controls button, at the top left, which we will explain momentarily. Below that, there are the zoom controls: "+" and "-" buttons zoom in and out, respectively, and the "Reset" buttons resets the to default position and zoom level.

Clicking on the "Visual Controls" button at the top left corner opens up the Visual Controls menu, with the following options:

State Views. (Available only if Problem Template defines State Views) Opens a submenu listing all available state views (alternate visualizations of each state), which are defined by the `IMGS` variable described in Section 3.1.1. An example of the State Views for the CitySim problem template (described in Section 4.2.1) is shown in Figure 4.6.

Refresh. Refreshes the nodes in the tree, including immediate update of any new nodes that have been created.

Layouts. Opens a submenu with different tree layout display options.

Spacing. Opens a submenu to select different spacing options for displaying the tree (e.g. adding or removing default spacing between nodes, etc.)

Image Resolution. Loads high or low resolution images, as specified by `IMG_HIGH` or `IMG_LOW` in the Problem Template.

Apply Filter. Displays options for applying filters to the tree layout. Solvers fill out a form by selecting options from drop menus to specify a search condition (e.g. "all nodes created by username Alice" or "all nodes of depth 3") and a visualization condition (e.g. "use red to highlight the nodes" or "minimize the nodes"). (Figure 4.7)

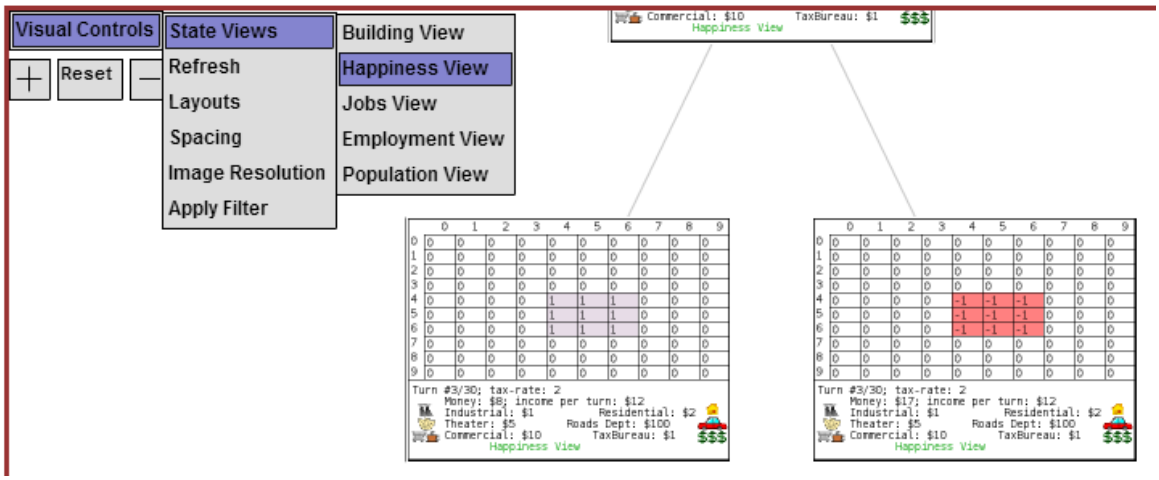


Figure 4.6: Problem templates can specify different views of a state, which the solver can then select during the solving session. Selecting a view will change the visualization of each state in each node. Here we see, under the Visual Controls menu, the options for State Views for the CitySim problem template, described in Section 4.2.1.

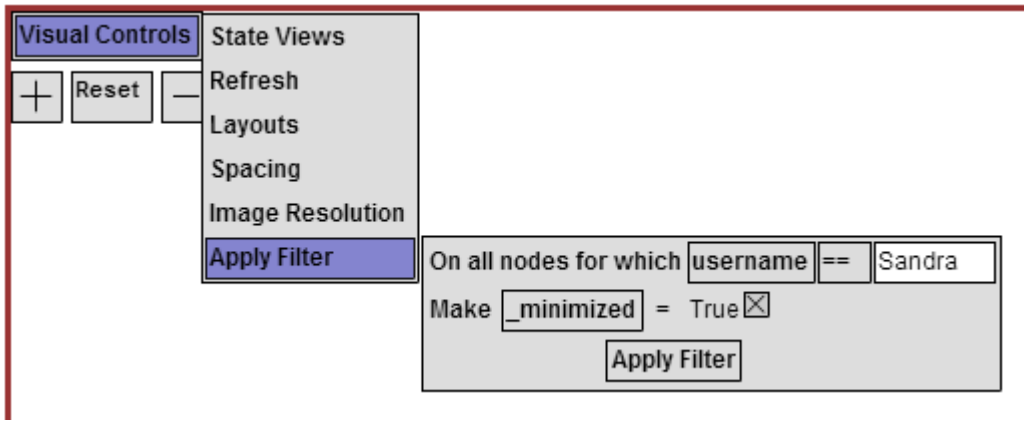


Figure 4.7: Solvers in a solving session can use the "Apply Filter" menu option to hide or highlight certain nodes. In this example, the solver is creating a filter to hide, or "minimizing" all nodes created by user "Sandra."

Node Menu and Applying Operators

Let's return to the tree root node in the middle of our solving session interface (Figure 4.5). Hovering the cursor over this node, or any other nodes that are created in the future, will bring up the node operations menu, or "node menu," which is shown beneath the node itself, and the node's visual options menu, which is shown as small icons above the node. This can be seen in Figure 4.8. The username of the solver who created this node, or in the case of the root, who created the solving session, is visible above the node.

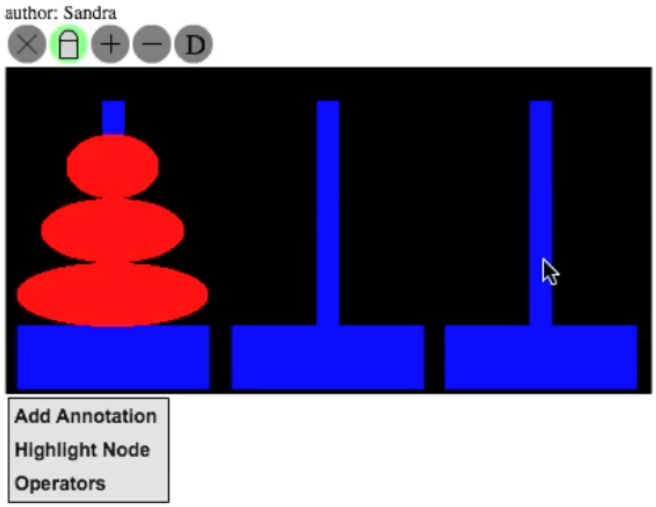


Figure 4.8: Hovering the cursor over any node in the solving session tree will bring up the node "visual options" menu buttons above the node, and the node operations menu or "node menu" below the node.

The visual options menu of a node allows the user to magnify or shrink the size of that node in relation to other nodes by clicking the "+" or "-" buttons, or to hide all the node's descendants by clicking the "x" button. The "lock" button locks the node as always visible, so that even if one of its ancestor nodes minimizes all its descendants, the locked node remains visible. These options could be used if, for instance, there are a large number of nodes, but the solver is only interested in viewing a few of them. To minimize visual clutter, the solver can lock only the nodes that are interesting to her, and then minimize the rest

of the tree. Finally, there is the "D" button, which displays node additional details (e.g. node creation timestamp, node ID, etc.) All of these visual options affect only the current solver's view; other solvers' views are not affected, and each solver can change their visual options independently of other solvers.

Below the node, in the node menu, there are three options: "Add Annotation," "Highlight Node" and "Operators." Annotations will be discussed in the next section. Selecting "Highlight node," as seen in Figure 4.9, will allow a solver to select a color, and then CoSolve will draw a colored halo around the node. This allows solvers to mark nodes that they wish to make note of.

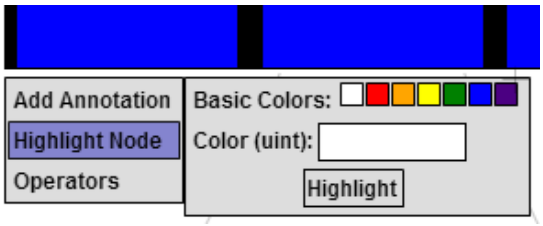


Figure 4.9: Options to highlight a node in a solving session.

To apply an operator to this node, we select "Operators" from the node menu. CoSolve will then display a submenu, with a list of available operators to be applied to this node. At this point, there are three possibilities. For problem templates with operators that do not take parameters, there will simply be a list of operators, and the solver selects one to apply, and it is applied immediately. An example of this is seen in the Towers of Hanoi node in Figure 4.10.

For problem templates with operators that do accept parameter inputs, the solver must then enter the parameters after selecting an operator. If the poser specified the operator's parameter as a 2D Point (as described in Section 3.1.1), the solver can click anywhere within the state visualization, and CoSolve will accept the click location's coordinates as the parameter input to the operator. The CitySim problem template, described later in this chapter, demonstrates this type of parameter usage. Figure 4.11 shows a node from a CitySim solving session. The solver applies operators to place "buildings" on a "city" map

state visualization. The solver selects the operator representing the building he wishes to place, and then clicks on the location in the map at which he wishes to place the building. CoSolve then applies this operator and creates a new node with the building in the desired location.

The third possibility is that the problem template has operators that require textual or numerical values as parameters. An example of this was given in the modified Towers of Hanoi template, in Section 3.1.2, in which users explicitly type numbers from 1 to 3 indicating which pegs they wish to move disks to and from. In this case, after selecting an operator, the solver is presented with another submenu, similar to the input forms seen in Figure 4.6 or Figure 4.9, where the solver can input the necessary parameters.

After applying an operator, a new child node appears in the user interface, below the current node, now a parent node. A gray line is displayed between the nodes to indicate that they are parent and child. If further operators are applied to the parent, the new child nodes will always appear to the right of any existing nodes. This serves the purpose of maintaining consistency in a solving session's visualization of the tree, which means that solvers can make use of their spatial memory to locate nodes they have previously explored. Additionally, solvers can generally find newer nodes towards the right of the tree, and can always tell in what order a node's children were created.

As more nodes are created and the tree grows larger, solvers can navigate the tree by using the visual controls and zooming buttons. They can also pan across the tree by clicking and dragging, to view different parts of the tree.

To avoid disturbing each solver's individual view, we display only a solver's own, newly-created nodes in the tree. New nodes created by others are queued up and displayed only when the solver requests them by clicking the Refresh button in the Visual Controls menu. When the solver clicks this button, the new nodes appear in the order that they were created, so that the solver has time to observe how they were created.

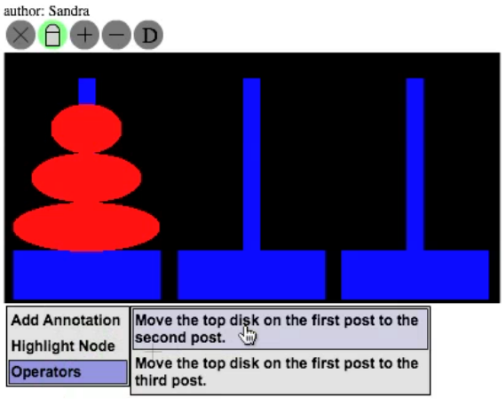


Figure 4.10: Applying an operator to the root node in a Towers of Hanoi Solving Session. The solver clicks to select the operator she wishes to apply, and a new node will be created.

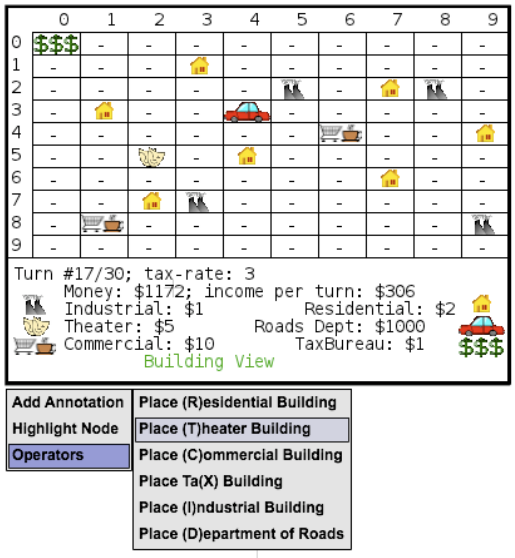


Figure 4.11: Applying an operator to a node in the CitySim problem template. Here, the solver selects one of the six different building types that can be placed into the grid above. Upon selecting an operator, the solver is then instructed by CoSolve to click on the desired grid cell in which to place the building. Then, CoSolve will create a new child node underneath this one, with a visualization image showing the new state containing the placed building in the desired cell.

Annotations

The final option in the node menu that we have not discussed is the "Add Annotation" menu item. When clicked, this menu item brings up the Annotation Form submenu (first seen in Figure 1.4). The Annotation Form submenu looks like this (Figure 4.12):

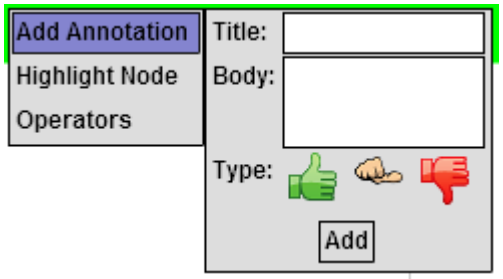


Figure 4.12: The Annotation Form submenu of a node.

Annotations are a way for solvers to provide comments regarding specific nodes. To annotate a node, solvers provide some text in the "Title" field. Solvers can optionally include extra text information in the "Body" field, then optionally select an annotation type (types will be explained shortly), and finally, click "add" to add the annotation to the node.

When a new annotation is added to a node, it is displayed in two places, as seen in Figure 4.13. First, it is displayed immediately beneath the node, in the Node Annotations area. The annotation title and the username of the annotation creator are displayed in bold. To the left of each annotation is an icon indicating the annotation type if one was selected, e.g. a "thumbs up" icon to indicate a positive annotation. To minimize visual clutter, the annotation body is hidden initially; a small button next to each annotation expands the annotation to display the annotation body text beneath it. The Node Annotations area can also be hidden entirely. When an annotations has nodes, the "Node Annotations" button appears above the node. Toggling this button allows solvers to show or hide the Node Annotations area as needed.

The second place in which annotations are displayed is the Annotations List, shown in the bottom left of Figure 4.13. Whenever an annotation is added anywhere in the tree, in

addition to appearing underneath the node, the annotation is also added to the Annotations List. This scrolling text area becomes a chronological list of all annotations created during the life of the entire solving session, and was intended to make it easy for solvers to see when new annotations had been made. Solvers often end up using the Annotations List as a sort of "chat history" box, as will be discussed later in this chapter. In the Annotations List, each annotation is displayed with the following information: its annotation type icon, the ID of the node that the annotation is associated with, the annotation title, annotation body, annotation creator's username, and the annotation creation timestamp (e.g. "8/30/2012 11:25am"). Clicking on the annotation will cause the solver's view to pan the tree and zoom in on the node associated with that annotation, making the annotations list a quick way to bookmark nodes. Additionally, unlike new nodes, these annotations are updated automatically without the need for a user to explicitly click refresh.

As mentioned earlier, each annotation can optionally have an annotation type. There are three annotation types: positive (represented by a "thumbs up" icon), neutral ("thumbs sideways" icon) and negative ("thumbs down" icon). These three are meant as ways for solvers to quickly mark their evaluations of these nodes. For example, if a solver thinks a node represents a potentially good solution to the problem, they can mark it with a "thumbs up" by creating a annotation with type set to "positive." Then, the "thumbs up" icon is displayed next to the annotation in the Annotations List. This way, the solver can both remember to come back to this node, and also alert other users that this is a node they may want to examine further. Other solvers can then jump to that node by clicking on that annotation in the Annotations List. Nodes marked as negative are nodes can be reminders not to continue down a path. Annotations types could also potentially be used for voting; solvers could "vote" for or against a node as a possible solution to the problem by creating positive and negative annotations.

In this section, we have described the problem-solving user interface in our CoSolve Flash client. We discussed the features we have designed to aid solvers in their problem-solving process. Let's now examine how solvers actually behave when using our interface, and how they collaborate, by analyzing the results of a user study we conducted on teams of solvers.

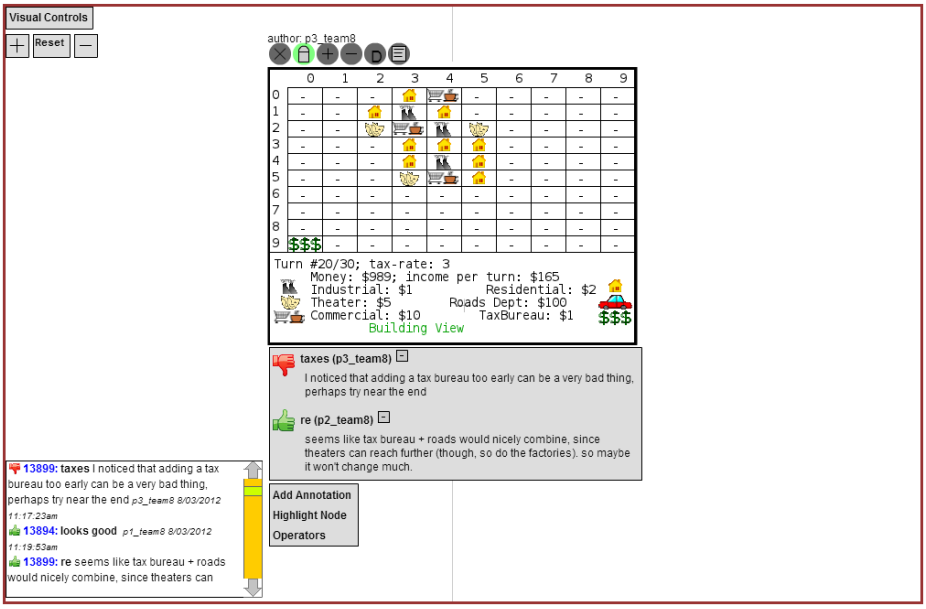


Figure 4.13: Annotations in a solving session. The right-most button above the node, the button with a list image inside, is the Node Annotations button. It shows and hides the Node Annotations area beneath the node. This area is currently shown. Two annotations have been added to this node, by users *p3_team8* and *p2_team8*. Both annotations here have been expanded. Additionally, we can also see these two annotations listed in the Annotations List for this solving session, as well as another annotation ("looks good" by user *p1_team8*) added to another node, not displayed here. The node ID of the node associated with the annotation is displayed in blue text, in front of the text of each annotation.

4.2 Evaluation: CitySim User Study

CoSolve's key premise is that the state-space-search problem-solving methodology can serve as the framework for human interaction within a collaborative problem-solving system. The main questions, then, are:

1. Are human users able to *understand* and successfully use state-space search for problem solving? Or, on the contrary, will they find such a representation too confusing or cumbersome to use?
2. *How* do these solvers use Cosolve-style systems for collaborative problem solving? What strategies do they employ?
3. How could such a framework aid solvers' understanding of their own collaborative problem-solving processes, and as a result, improve their collaborative efforts?

To answer these research questions, we conducted a user study with small teams of users collaborating to solve a city-simulation problem we formulated called CitySim, described in Section 4.2.1. To answer the first two questions, we examined our users' resulting solving-session trees, administered attitude surveys, and conducted individual interviews with the users. To address the issue of users' understanding of their collaborative process, we also implemented and user-tested the CoSolve Consultant,[5] a specific set of tools built within CoSolve for enhancing group awareness and metacognition during the collaborative problem-solving process, which will be described in detail in Section 4.2.2. The results of our study are also published in [2].[6]

In the following sections, we will describe the CitySim problem template and the CoSolve Consultant, and then we will talk about our user study design, and finally, present our results and findings.

---

[5]This user study was jointly and equally conducted with Tyler Robison. I will include some of my own findings as it relates to the Consultant, but see his dissertation [54] for a full analysis of the CoSolve Consultant itself.

[6]We gratefully acknowledge IEEE Computer Society for permission to reprint Figures 4.11, 4.14, 4.15 and Tables 4.1, 4.3, 4.2 in this section, which were originally published in [2].

4.2.1 CitySim Problem Description

For the purposes of this user study, we created a problem template entitled CitySim[7], a turn-based, stylized urban-planning simulation, for our subjects to solve. It was inspired by Maxis's SimCity game [56],[57].

In designing CitySim, we wanted to design a problem template that would be interesting to our subjects, easy to understand, and easy to train our subjects to use given the time constraints of a single user study session. Additionally, we wanted a problem template that was not so open-ended as to be difficult to compare across teams; we wanted concrete ways to measure subjects' performance on task. To attempt to meet these criteria, we implemented a template that was game-like, and had a well-defined set of operators such that solving-session tree nodes had a manageable branching factor.

At the same time, we wanted to come up with a problem template that had sufficient complexity for a team of solvers to feel challenged during the time allotted for the task. We feel we achieved this goal; in our interviews with the subjects, many of them told us that while they were not unhappy with their solutions, they felt like they could, and wanted to, continue trying to solve CitySim. They also discovered a variety of interesting, different techniques for maximizing CitySim score that we had not anticipated.

A CitySim state consists of a 10x10 grid of a "city," upon which different types of buildings can be placed. Each building costs the city *money* to place, and each placed building has a different type of effect on its cell and the cells around it. The city earns *money* through taxes paid by the employed residents of the city. The goal of CitySim is to find a way to place 30 buildings in the grid such that the city has the most *money* after the placement of the last, 30th building. The *money* value at a state is the "CitySim score" for that state, and the final, overall *CitySim score* for a solving session is the highest score of any node over the entire session.

A close-up of a CitySim state can be seen in Figure 4.11. Here we see a close-up of the solving session, with single node in a CitySim solving-session tree. This state shows an in-progress city, with different buildings, placed in various locations on the city grid.

[7]Tyler Robison was the primary designer of CitySim.

The operator menu is open, showing the different building types that can be placed. Upon selecting an operator, the solver then clicks on the desired grid cell, and CoSolve will create a new state with the selected building in that space. The city grid is in the top half of the state visualization. In the bottom half, we have information about the city's attributes, and then a legend showing the available buildings, their icons, and how much each building costs.

Each cell in the grid has several attributes: *population, happiness, jobs* and *employment*. In a nutshell, you want your city to have a high *population* and enough *jobs* for all the residents, so that they can pay the city money through their income taxes. However, if the *happiness* of the citizens is too low, they will leave their jobs for greener pastures. Placing different kinds of buildings affects the values of these attributes, for example, adding a *theater building* increases the *happiness* of citizens living near the *theater*. Or, for example, adding a *department of roads* increases the area of effect of existing buildings. Some buildings have multiple effects. The CitySim problem template uses the CoSolve's State Views feature to show an overlay of the spread of each of the attributes–*population*, *jobs*, etc.–over the city grid, so that solvers don't have to manually calculate what effect each placed building has, as shown in Figure 4.6.

4.2.2 The CoSolve Consultant

To help users understand their own collaborative problem-solving process, and thereby possibly collaborate better, the CoSolve Consultant was developed for solvers to use within CoSolve while solving. This tool was created by Tyler Robison after my initial development of CoSolve. Below, we describe the relevant features of the Consultant, but a more in-depth description can be found in Robison's thesis [54].

The CoSolve Consultant is a collection of tools that display extra information about the solving session, the solvers, and the solving-session tree, to the user. The goal is to see whether the Consultant will improve or affect the solving behavior of the users. The Consultant interface can be seen on the left of the image in Figure 4.14, between the Visual Controls at the top and the Annotations List at the bottom of the screen. Some of the more

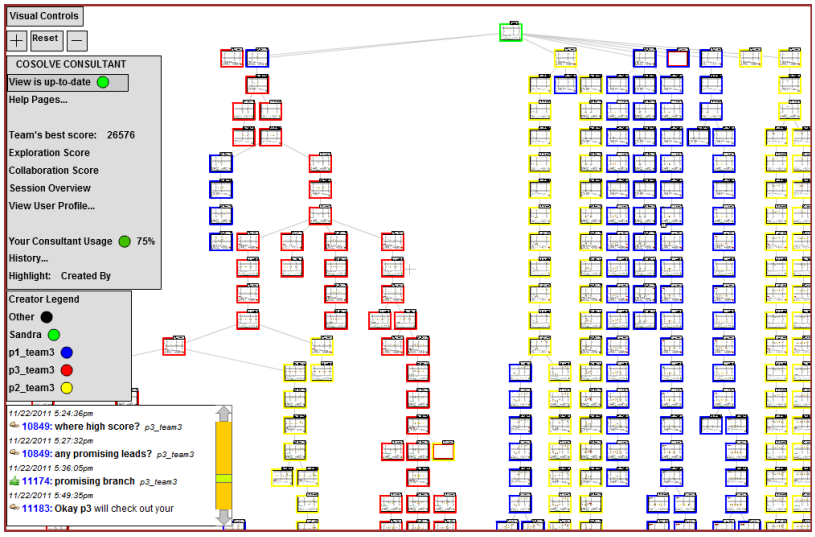


Figure 4.14: This is an in-progress solving session for CitySim, with the Full CoSolve Consultant along the left of the screen. Pan and zoom controls are in the upper left, the Annotations List is in the bottom left. The rest of the area displays the solving-session tree.

important features include the following:

Highlighting by solving and collaboration metrics. By default, solvers can add a colored halo around each node in the tree, which is referred to as "node highlighting". Solvers can do this to mark or draw attention to different nodes. The CoSolve Consultant allows solvers to select options for automatically highlighting all nodes in ways that might enhance their solving and collaboration process. The node highlight options are:

Creator. Each solver is assigned a color, and nodes are highlighted with the color of the solver who created the node. This is the CoSolve default, and is available even without the Consultant tool.

Solution Path. A CitySim-specific highlight, that highlights the path from the root node to the node that currently has the highest CitySim score. This is useful for solvers to quickly see where the best solution currently is, and is increasingly important as the tree grows larger and larger.

Number of Annotations. The more annotations a node has, the greener the highlight color will be. Nodes highlighted in red have no annotations. Unlike the first two highlight options, which assign a discrete set of highlight colors to the nodes, the highlight colors here range on a gradated scale of color from red to green (intermediate colors between red and green are brown-ish shades, see the left side of the screen under "Recently Created" in Figure 4.15). Drawing attention to nodes with lots of annotations helps the solver know which areas of the tree are being discussed by his or her teammates, and hence may potentially signal the most interesting or useful nodes in the solving session.

Recency / Creation Time. The more green, the more recently the node was created. This metric is meant to show solvers which parts of the tree have the most current activity.

Node Score. Another CitySim-specific highlight, that highlights each node based on its score, on a scale from the lowest scoring node to the highest. Note that this is different from Solution Path highlighting, which highlights an entire path in one color. Node Score highlights each node individually, with a darker green indicating a higher score. The intention is for solvers to be able to easily compare nodes by score when zoomed out.

Annotation Type Balance. Annotations can be positive, negative or neutral. This highlight option colors the nodes again on a gradated scale, based on whether the sum of the annotations on a node are more negative, more positive, or about equal. This helps solvers see at a glance which nodes have been mostly positively or mostly negatively annotated.

Our intention is that these types of node highlighting options, by making it easier to compare nodes across the whole tree, encourage users to take a high-level view of their problem-solving process, rather than focus on the solving process on a per-node basis.

Session and solver statistics. To further encourage solvers to take a higher-level view, the Consultant offers metrics to help them keep track of their team's progress, and the

contributions of their teammates, in the form of statistics about each solver and about the overall session. These include simple counts such as the total number of nodes and annotations, as well as richer measures that describe the shape of the tree, and the contributions of individual solvers.

High-level guidance. For novice solvers, CoSolve might present a totally new approach to problem solving. As such, solvers may be unsure of how to proceed, and are unaware of high-level strategies that may make their task easier. The Consultant tries to assist the solver by displaying solver profile metrics to help solvers to consider the possibilities and implications of their actions, and it places judgments on some of these actions in order to hint at "good" solving behavior. For instance, CitySim allows solvers to place six different types of buildings. The CoSolve Consultant's computed profile for a solver includes a metric for number of buildings that solver has tried (*User-operators-used*), and seeing that, say, you've tried only two buildings of the six available buildings, may encourage more experimentation by the solver.

Solvers access the last two features, statistics and high-level guidance, by clicking on the corresponding menu items in the Consultant interface. This brings up information windows as seen in Figure 4.15, and these windows have links to additional information explaining each metric.

4.2.3 Study Design and Procedures

Eighteen subjects (11 male, 7 female, ranging from 18 to 31 years of age) were divided into six teams of three subjects each. These teams will be called *Team 1*, *Team 2*, *Team 3*, etc. The subjects were recruited via the Internet and with flyers on the University of Washington campus, and by word of mouth, and were compensated with Amazon $20 gift cards.

To test whether the CoSolve Consultant was helpful, we divided teams into two groups. The first group, consisting of three teams, saw and used the CoSolve Consultant, as presented in Section 4.2.2 and seen in Figure 4.14. These teams were *Team 2*, *Team 4*, *Team 6*. As a control condition, the other three teams (*Team 1*, *Team 3*, *Team 6*) also saw a version of the *Consultant* in the same UI area of the screen. However, it was actually a

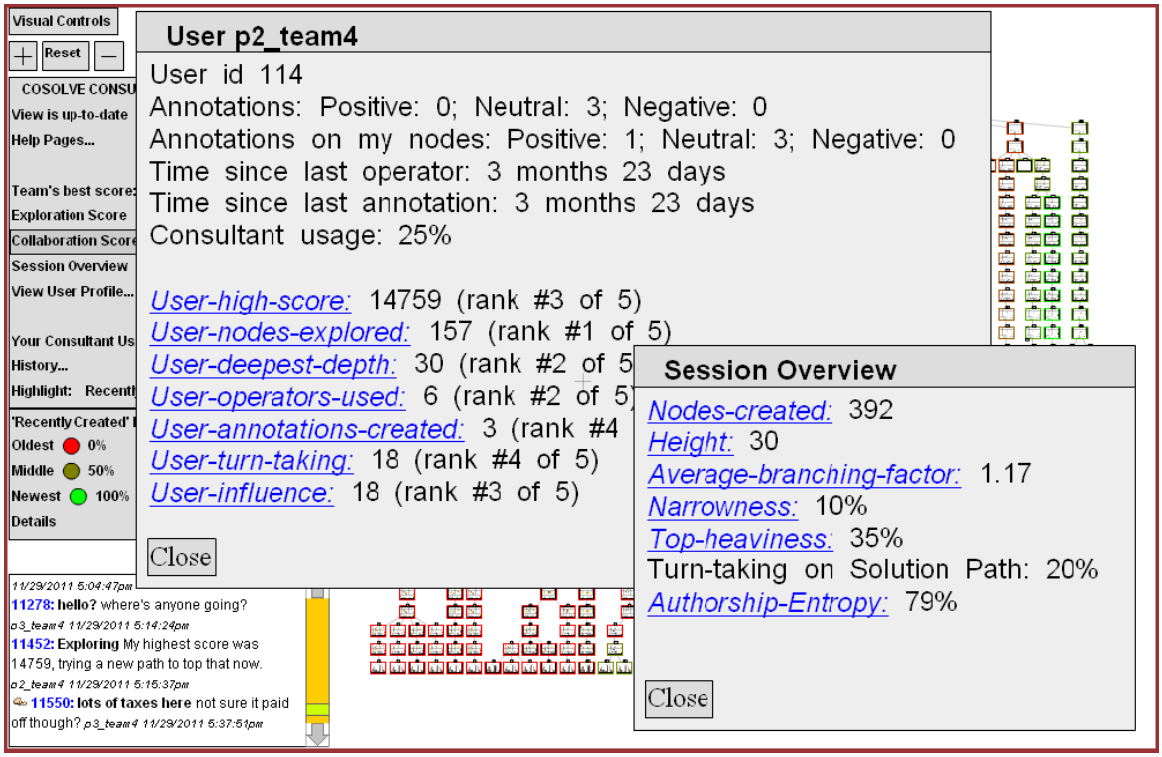


Figure 4.15: Two CoSolve Consultant information windows for subject *p2-team4*. The window in the back shows per-user profile statistics; the window in the foreground shows overall session statistics, including tree shape metrics such as *Average-branching-factor*, etc.

stripped-down adaptation of the interface. In our discussion, we will call this the *Minimal Consultant (MC)*. The *Minimal Consultant* did not have any of the exploration score, collaboration score, session overview, or user profile menu items. Only the default highlighting option (highlight tree by creator) was available in the MC interface. This interface can be seen in (Figure 4.16). For clarity, we will call the regular Consultant interface seen by *Teams 2, 4* and *6* the *Full Consultant (FC)*.

Per-Team User Study Session

One team at a time was scheduled to come into the lab for the user study session together, with each subject seated at a separate computer. Two experimenters were present at each

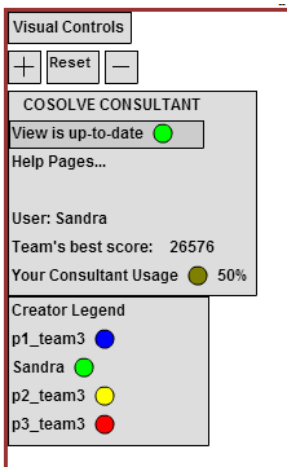


Figure 4.16: The Minimal Consultant (MC) interface does not contain the links to all the extra windows that the Full Consultant (FC) provides. It also includes only the default highlighting option–by creator–and so the creator color legend is always shown beneath the MC interface, whereas in the FC, this legend changes based on which highlighting option is selected.

user study session.

Upon entering the lab, subjects were asked to individually fill out the following forms:

1. **Human Subject Consent Form.** Subject provides their consent to be in the study.

2. **Background Questionnaire.** Demographic data, as well as questions about the subject's familiarity with computers, online collaboration, etc. (Appendix B)

3. **Pretest.** This form tested each subject's prior knowledge of graphs and trees, so that we could assess how familiar each subject was with these concepts before beginning to use CoSolve. (Appendix C)

After all scheduled subjects in a team arrived and completed these three forms, the user study began. The experimenters spent approximately 25 minutes giving a tutorial on how to use CoSolve, and what the CitySim rules are, as well as giving subjects hands-on sample

CitySim tasks in the CoSolve. Each subject had a chance to try out applying operators, creating annotations, viewing their teammates' nodes, and navigating the user interface. The script used during the tutorial is included in Appendix D.

Then the participants began their task. They were told that they needed to achieve the highest overall score among the teams in the study, and that the winning team would receive an additional $20 gift card per person. On each subject's user interface, in the Consultant area, there was also a Consultant Usage meter. Subjects were also told that clicking on the Consultant's features (whichever version theirs happened to be, MC or FC) would keep their Consultant Usage meter full, and earn their team additional points. This was so that we could compare Consultant usage between groups, and ensure that if the FC group created fewer nodes than the MC group, it was not because they instead spent their time clicking on the Consultant.

The solving activity was divided into three phases. Each 30-minute phase consisted of 25-minutes of solving activity, followed by a five minute break, for a total of 90 minutes, of which 75 minutes were on the task and 15 were on breaks. During all three phases, a team worked using the same solving session, i.e. the same tree. Due to a scheduling issue, Team 1 only worked for two of the three phases, but we have included them in our data.

While working on the task, subjects were asked to communicate with each other only through CoSolve, not verbally out loud, despite being co-located. During breaks, they were allowed to talk to each other, but we asked that they not discuss the solving session task or CitySim in any way.

To recap, a summary of the overall timeline for a user study session is as follows:

1. Fill out consent form, background questionnaire and pretest (10 min)
2. Introduction and System Tutorial (25 min)
    - CoSolve, CitySim, CoSolve Consultant interface tutorial
3. User Study Task: CitySim solving session (90 min)
    - Phase 1: Work on solving session (25 min), break (5 min)
    - Phase 2: Work on solving session (25 min), break (5 min)

- Phase 3: Work on solving session (25 min), break (5 min)

Post-Activity Wrap-Ups

We also scheduled each subject for an *individual* post-activity wrap-up. Each wrap-up was conducted after the subject's user study session, and lasted about 45 minutes (though some subjects were done in half an hour, and others stayed for nearly 1.5 hours).

Each wrap-up was scheduled at the subject's convenience, but not more than one or two days after the user study session. Most subjects chose to do the wrap-up immediately after the user study session.

The schedule for a wrap-up session was as follows:

1. **Post-test.** Subject completes a post-test (Appendix C), similar to the pre-test, to assess whether there was any learning gain in terms of state-space tree understanding and terminology. (5 min)
2. **Interview.** Experimenter conducts a guided interview (script is in Appendix G) with the subject, looking over his or her solving-session tree and asking the subject to describe his or her problem-solving processes. All but one of the interviews were audio-recorded. (20-60 min, depending on the subject)
3. **Wrap-up questionnaire**. Subject completes a questionnaire (Appendix E) on his or her impressions of CoSolve and on his or her attitudes toward collaboration after having done this activity. (5-20 min, depending on the subject)

Upon completion of a subject's wrap-up session, the subject was given his or her compensation, the $20 Amazon gift card. After all teams had been run, each of the subjects in the team with the highest score, was emailed an additional $20 Amazon virtual gift card.

Now that we have thoroughly outlined the study procedure, let's discuss the results of this CitySim study.

4.2.4 Study Results

In our data, each subject is referred to by their "participant number" and *team number*, so the three subjects in Team 1 are *p1-team1*, *p2-team1* and *p3-team1*, the subjects in Team 2

Table 4.1: Overall CitySim user study team results. The CitySim score for a team is the CitySim state that has the highest score (*money* value) within that team's solving session. Annotations each have a "type" that the subjects could select: positive, neutral, or negative. If the subject did not select a type, that annotation is counted as a "no type" annotation.

	Minimal Consultant (MC)			Full Consultant (FC)			Overall	
	Team 1	Team 3	Team 5	Team 2	Team 4	Team 6	Mean	Median
CitySim score	33087	26576	26865	31314	14848	20352	25507.0	26720.5
Nodes created	177	398	319	263	391	100	274.7	291.0
Total Number of Annotations	38	25	5	21	22	55	27.7	23.5
- "Positive" annotations	17	18	4	7	7	5	9.7	7.0
- "Neutral" annotations	1	7	0	3	2	5	3.0	2.5
- "Negative" annotations	1	0	1	2	0	2	1.0	1.0
- No type annotations	19	0	0	9	13	43	14.0	11.0

are *p1-team2*, *p2-team2* and *p3-team2*, and so on. The *participant numbers* were randomly assigned to the subjects.

We will first present overall performance data for all the teams, and then provide details of the per-team and per-subject results. Finally, we will compare the FC teams and the MC teams. Except where otherwise stated, the results presented are not statistically significant due to a small sample size of only three teams per condition. However, we can use this data to gain insight on how subjects use tree-based problem-solving tools, and to identify areas for further investigation.

Overall Activity

Table 4.1 shows each team's CitySim scores, number of nodes created, and number of annotations created, by type. Overall, the data shows quite a large variation between teams.

As mentioned earlier, a team's CitySim score is defined as the *money* value of the CitySim state with the highest *money* value out of all the states in a team's solving session. Overall, teams did well in terms of score; when we were developing the CitySim template, we did not achieve more than about 30,000 points in a solving session during testing, but two teams

were able to exceed that score. There was a fairly large range in scores: the lowest scoring team, Team 4, scored only 14848 points, and the highest, Team 1, scored 33087 points.

The number of nodes created in a solving session did not affect the final CitySim score of a team. One might think that creating a lot of nodes might mean that a team would score more highly, since they explored more options. Or it might mean the team was unfocused and didn't know what they were doing and so, teams with lots of nodes would have scored poorly. In actuality, neither was the case. Team 3 created the most nodes, 398 nodes all together, but their CitySim score was close to the median. Team 6 created the fewest nodes, only 100, and their score was 20352, which was second-lowest score. However, the team that created the second-fewest nodes, Team 1 with 177 nodes, was actually the highest-scoring team.

There was also a wide range in the number of annotations each team created: the minimum was 5 annotations (Team 5), and maximum was 55 annotations (Team 6), though the rest of the teams created between 21-38 annotations. Again, number of annotations created did not correlate with either CitySim score or number of nodes created, although the two teams that created the most annotations (Team 6 with 55 annotations and Team 1 with 38 annotations) also created the fewest nodes (Team 6 with 100 nodes, Team 1 with 177 nodes).

As for type of annotation, we found that subjects mostly created *positive* annotations, or else did not select a type for the annotation when submitting it. Groups had very few *neutral* annotations, and almost no *negative* annotations, creating one or two of these at most. Between the *positive* and the *no type* annotations, we saw different kinds of behavior. Team 6, created mostly *no type* annotations (43 out of 55 total annotations), whereas some teams created mostly *positive* annotations (Team 3, 18 out of 25). Others, like Team 1, split their annotations between *positive* and *no type* (17 *positive*, 19 *no type*).

Overall, our data shows us that there is high variation in solving outcomes even for what initially appears to be a fairly simplified problem template. Our sample size of six teams was not very large, so it's possible that we don't have enough data to notice any trends. So, instead, let's examine each team's solving session trees directly.

Team Solving-Session Trees

Figures 4.17 through 4.22 show the final solving session trees of each team in the study[8]. Nodes are highlighted in red, blue or yellow depending on which of the three subjects in a team created it; the root is always created by the experimenter and is always in green. As can be seen in the images, each tree looks very different, and from this we can deduce different solving behaviors of each team. We will now point out two properties of these trees that may give us some insight into our teams' behavior: branching, and turn-taking.

In CitySim, all trees have a maximum depth of 31–one node for each of the 30 buildings place, plus the root node (always created by the experimenters, username is either *trobison* or *Sandra*) with no buildings in it. If a solver went straight through and placed one building after another, then there would be one straight branch of the tree directly from the root to the leaf node at depth 31. However, not all paths are *straight*, i.e. not all nodes have just one child. Some of them branch, meaning that different options were explored at that point, either by placing a different building than the one that was placed in the first child, or perhaps placing the same building in a different location, etc. These two children might be created by the same solver, or different solvers.

Team 5's tree (Figure 4.21) shows an example of this. Notice that there is a lot of *branching* in *p3-team5*'s paths (red nodes), indicating that this subject experimented with a lot of different options and did a lot of exploration. Now, notice that the third branch from the right (the second blue branch from the right) is almost perfectly straight going from the root to the 31st level. It is also entirely blue. This shows us an example of a single solver going straight down to try to create a solution, without trying many other options along the way.

Both of those branches were entirely of one color–the first was all red, the second was entirely blue, meaning no one else attempted to branch off of these nodes, and that the subjects were working on them in isolation. This is in contrast to the middle section of this tree, where there are red (*p3-team5*), blue (*p2-team5*) and yellow (*p1-team5*) branches coming off of a red branch in the middle. Here, we see that all three subjects branched off

[8]These screenshots were taken by Tyler Robison.

of a particular node. Usually, this indicated that it was a particularly high-scoring node, and so the rest of the team wanted to build off of it. Robison calls this property of applying operators to nodes created by solvers other than oneself, *turn-taking* [54]. For this particular tree, however, this node is the *only* node where solvers created nodes off a node other than their own, so there is not much turn-taking in this tree.

This is in contrast to, for example, Team 4's tree (Figure 4.20). Notice that on the left half of the tree, on the leftmost branches, there is a lot of interleaving of different creator colors, i.e. a lot of turn-taking. This indicates that our subjects were watching each others work and building off of it. Lack of turn-taking does not necessarily mean that subjects were unaware of others' activity, they could be keeping track of others' nodes but simply not adding on to them. However, presence of turn-taking definitely indicates that solvers were aware of others' work, because they had to refresh the tree to view newly-created nodes, and they had to look at and find the node they wished to build from. Team 4 was the most extreme example of this; the upper, leftmost section shows the three solvers each taking their turn, one after the other. Near the bottom of the branch, however, *p1-team4* (yellow) has dropped out, and the interleaving is mostly between *p2-team4* and *p3-team4*.

This brings us to another interesting attribute that can be gleaned from the placement of nodes in a tree. Since children are placed from left to right, and nodes necessarily must go from top to bottom, a rough estimate of the time of node creation can be ascertained from just looking at a finished solving session tree. For example, in Team 4's tree (Figure 4.20), since the branching and color-interleaving is mostly on the left, we can see that this team started out branching and turn-taking, but later branches, on the right, are single-color, meaning the subjects started to work on their own more. Also, there is a lot of interleaving of color at the bottom of the tree, indicating that at some point later, after the initial left branch, subjects went back to turn-taking.

This behavior was generally confirmed based on our interviews with the subjects afterwards. Subjects in FC teams told us that, encouraged by their training with the Consultant, they would start off trying to do turn-taking. (Notice that there is less turn-taking in the upper part of the leftmost branches of the MC teams than in the FC teams.) Later on, they said they felt it was not worth it, as the lag and need to refresh the screen became a greater

burden, or they would simply want to explore on their own more, usually by creating a brand-new branch from the top. Then in the final stages of the solving session, when time was running out, subjects would go to the bottom of the tree to try incremental changes to improve on whichever nodes had the highest scores, rather than start a whole new branch from the root to a completed city leaf node, since that would take longer.

Team 3's tree (Figure 4.19) is an example of both kinds of behavior, but in a slightly different way. In their tree, on the right there are the single-creator non-branching paths. On the left, there looks to be a high incidence of color interleaving, but there is actually less turn-taking going on than it appears: most connected areas are of a single color, meaning each was created by a single subject, and then at certain nodes in those areas, another subject would branch off from those nodes, but then proceed to work entirely alone.

Annotation Usage

Overall, subjects neither used annotations as much as we had anticipated, nor did they use them in ways that we had anticipated. Some subjects said they didn't annotate much because of the time pressure–they felt they could get more done by actually creating nodes rather than discussing them. While subjects often did use annotations to mark states as "good" or "bad" in terms of whether or not the state was potentially on the path to a good solution, e.g. *p3-team4* annotated a node with "has lots of money here. 1218 and 256 in taxes!", subjects also often used the annotations as a chat box, and their annotation text sometimes had nothing to do with the actual node they were annotating. These types of annotations usually dealt with overall strategy ("Don't go for theater. it seems better to go for indust/res first" (*p3-team3*)) or meta-strategy ("let's coordinate a little before starting" (p3-team1) or "Guys lets not forget to refresh our consultant percentage" *(p1-team3*). Teams would sometimes have extended conversations on strategy that did not deal with any particular node.

We also saw social niceties ("Good job p2" (*p1-team3*), then *p2-team3* responded "thanks :D") and even annotations about the interface itself ("testing rootchat. body of annotation!" (*p1-team2*), "sorry. i can't add annotation.s" (*p1-team1*), "click this message. it

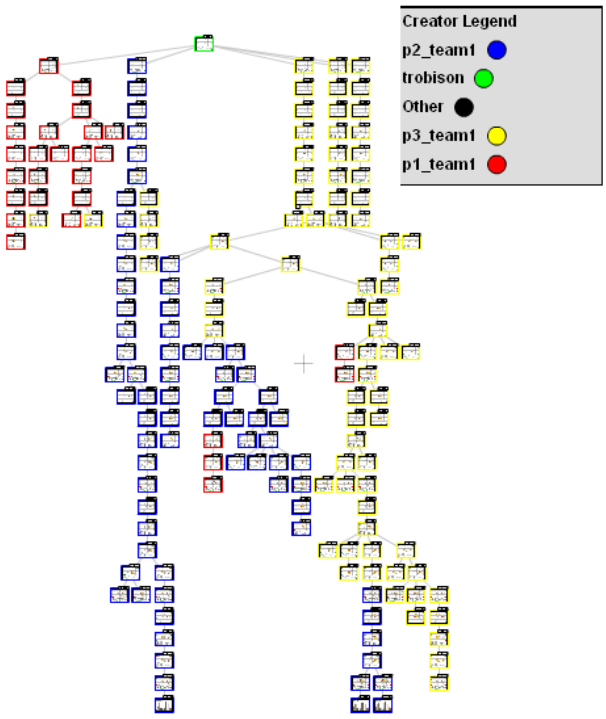


Figure 4.17: Team 1 - Solving-Session Tree (MC)

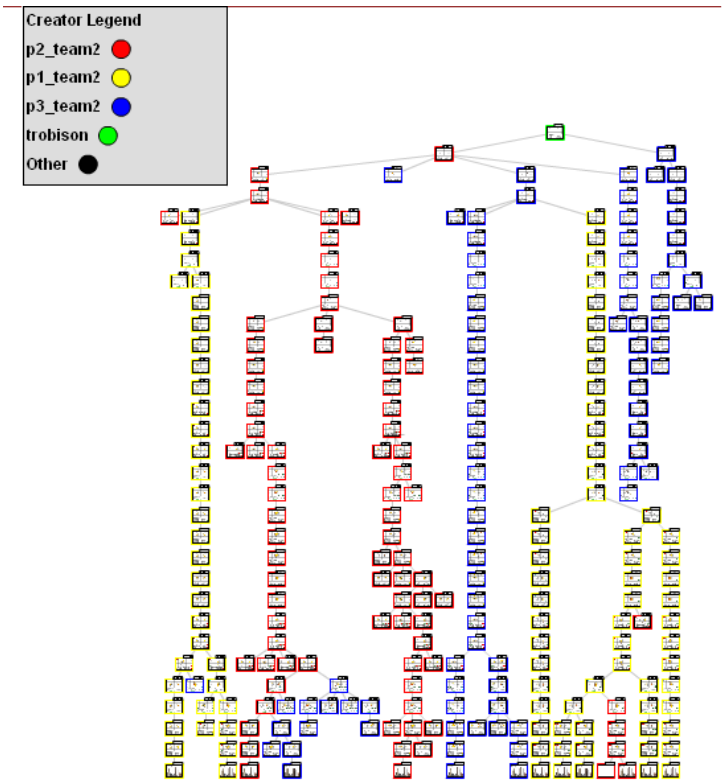


Figure 4.18: Team 2 - Solving-Session Tree (FC)

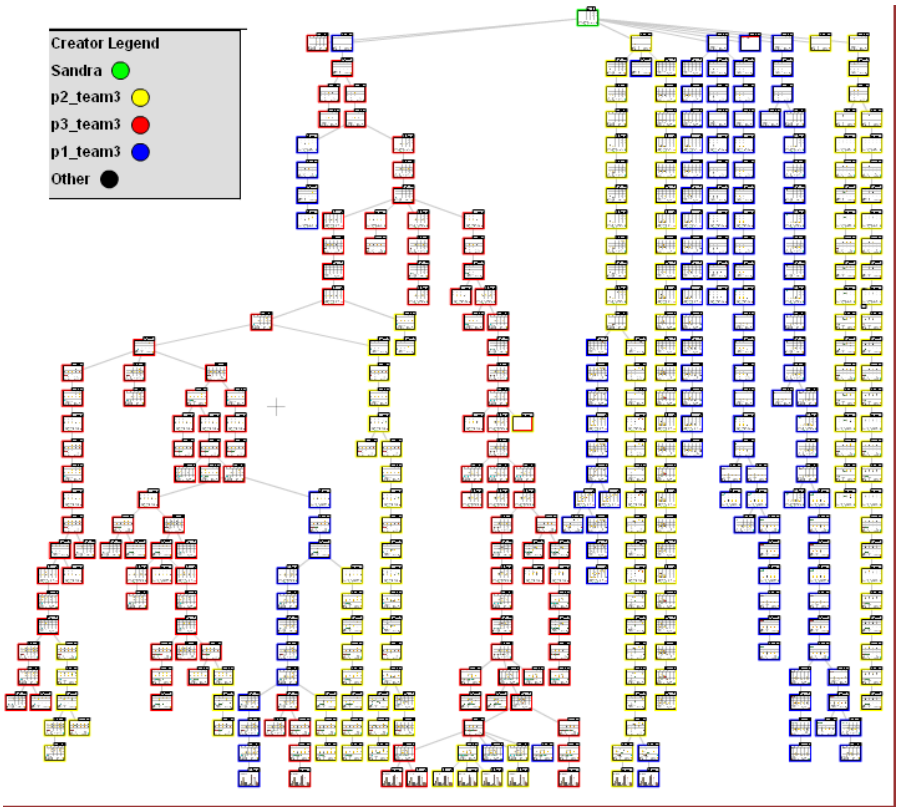


Figure 4.19: Team 3 - Solving-Session Tree (MC)

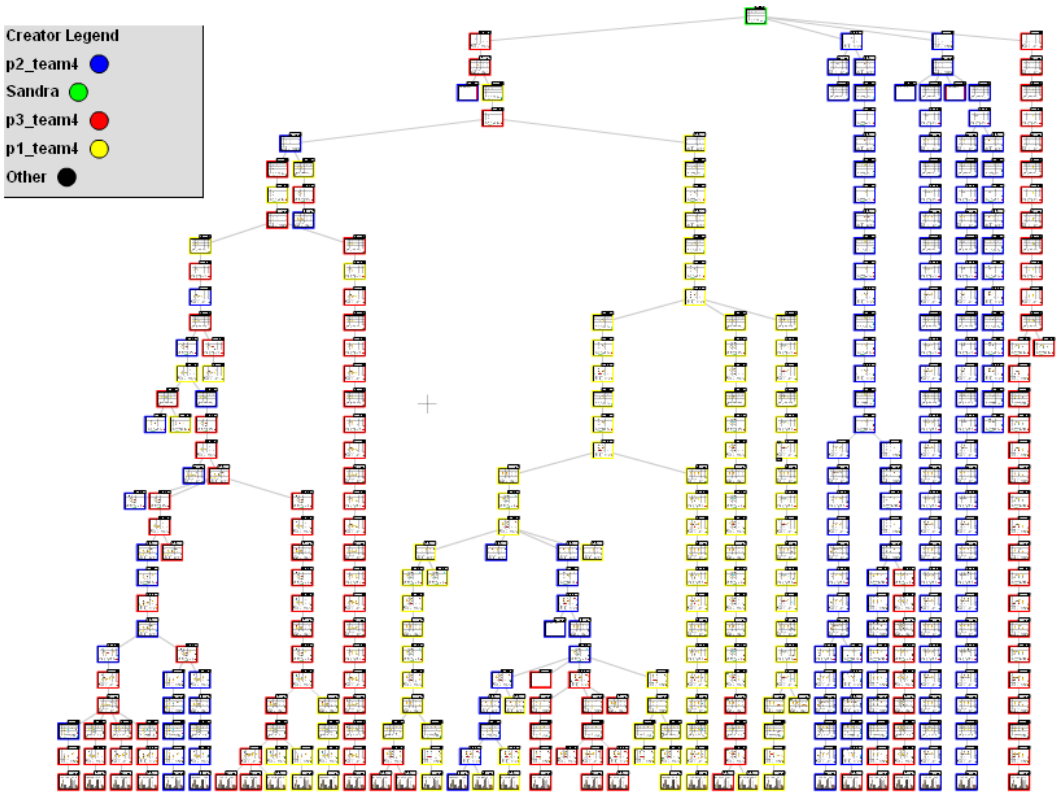


Figure 4.20: Team 4 - Solving-Session Tree (FC)

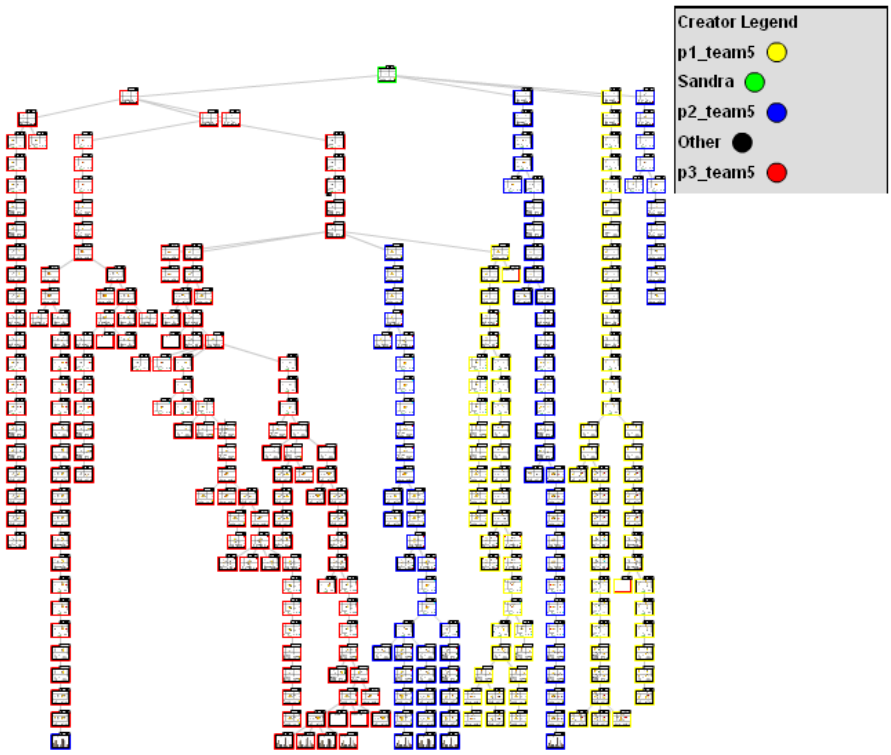


Figure 4.21: Team 5 - Solving-Session Tree (MC)

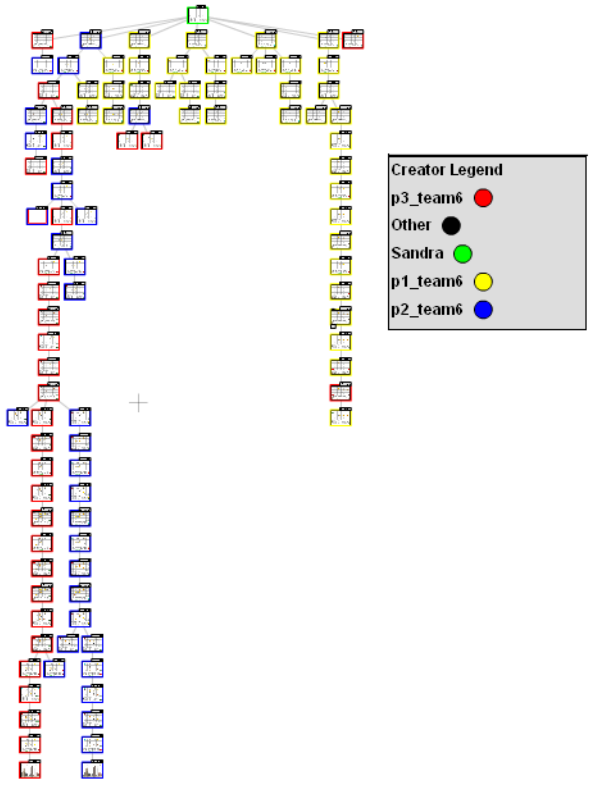


Figure 4.22: Team 6 - Solving-Session Tree (FC)

takes you to annotated state" (*p3-team1*)). Sometimes subjects would use the root as a place to store their non-node-specific annotations (suggested to them by the experimenters) but eventually they would end up conversing at whatever node they happened to be working on, most likely so that they would not have to jump back and forth between nodes. In at least one case, a subject used annotations specifically as a bookmark for herself, so that after refreshing the view and looking at others work, she could jump back to the nodes where she had been previously working. Most subjects, though, remarked that they did not make annotations for themselves, only to communicate with others. Because of the time pressure, they felt creating new nodes was more important.

As mentioned before, there were very few critical annotations, ranging from zero to two negative annotations per team. There are two reasons for this, mentioned by some of our subjects during interviews. First, subjects did not want to be seen as harsh or critical. Secondly, many subjects said that if a node is not promising, they didn't want to draw attention there, since the teams had limited time to work on their sessions; rather, you want to find the good branches and continue working off of those.

Finally, it should be noted that some of the subjects did not seem to realize that clicking on the annotations would make the viewport jump to center on the associated node. All subjects were taught to do this and confirmed that they could during the tutorial, yet from our interviews and from the annotation content itself, it became apparent some of them were unaware, or mostly likely forgot, about this functionality. (Sometimes, one teammate would annotate a node and tell others to look at it, and then another teammate would respond saying they "can't find" the node.) This could also have been due to the fact that they needed to refresh their screen to display the most recent nodes. In either case, it's possible that their annotation behavior would have been different had the subjects been able to use the jump-to-node annotation feature more reliably, and that we may have seen more node-specific annotations.

Individual Subject Behaviors

From our individual wrap-up interviews with the subjects and their answers to the Wrap-Up Questionnaire, we identified several behaviors that were characteristic of solvers in our system.

Tree navigation. The first set of behaviors involves tree navigation strategies for comparing nodes. The most common behavior was comparing nodes by scrolling vertically or horizontally. When we asked subjects how they decided what their next step should be when solving, many subjects told us they would pan horizontally across all nodes at a single depth level, to compare the *money*, *tax-rate*, or *income* across the nodes at that level. They would then select the node that contained the best state based on some combination those factors, and continue working using that node. It was such a common behavior that, during the interview, some subjects asked for a way to be able to easily navigate horizontally across nodes using keyboard shortcuts. This specific behavior could be due to the structure of the CitySim problem itself. Since the problem had an imposed limited depth of 31, and each level added exactly one more building to the city grid, all nodes at a level have the same number of buildings, and are directly comparable as a result. If, for example, CitySim allowed solvers to remove buildings at any arbitrary turn, comparable states with the same number of buildings might instead be on different levels, and make it harder to compare horizontally.

Another related, commonly reported behavior was to examine children of a single node, instead of looking at all existing nodes at a level. Subjects would apply several operators in quick succession to the same node, and then visually compare all these children and select the "best" one of them to continue working with. This local comparison behavior, unlike horizontally scanning an entire level, is probably one that would be common to any problem template.

Finally, some subjects said they navigated up and down a single branch, comparing each successive child node, to figure out their teammates' strategies on that branch. They would sometimes do this if, say their teammate *thumbs-up*-annotated a node as being promising. The subject would then go to the node, and then scroll up to examine what strategies made this a successful node. (CitySim solvers cannot simply examine the final node because building placement order affects the *money* value of a city, due to CitySim's particular set of mechanics.) Then the subject might go back to a branch they were working on, and apply that same technique. Hence, CoSolve's tree structure allowed subjects to learn from each other, since they could actually see their teammates' thought processes in creating each

node. In many design interfaces, users can only see the final product, rather than being able to learn from seeing the creator's decision process.

Solving styles. We also saw a difference between subjects in terms of solving styles. Some solvers were "planners" and others were "explorers." This was interesting to us as we initially considered CoSolve to be a platform for exploratory brainstorming, so we were surprised when some subjects thought of CoSolve nodes less of as "rough drafts" and more as "final versions."

The "explorers," who were the majority of the solvers, generally wanted to jump right in and start experimenting with CitySim. Sometimes they would start by just creating a straight branch all the way to the end state, to see what would happen. They would also exhibit behaviors such as creating several nodes and then selecting one to work off of, as mentioned earlier.

On the other hand, the "planners" took their time to consider each move before making it. After creating each node, they would switch between state views and carefully consider the best operator for their next move. Two subjects actually took out paper and pencil to write down notes and sketch out ideas, instead of just testing them in CoSolve itself as we would have expected, though they tended to do this only at the beginning, perhaps because they were not yet used to using CoSolve. Then they would start exhibiting more "explorer" behavior.

The fact that this difference exists between these two types of solvers leads us to think that there could be some fundamental difference in problem-solving personality, between people who prefer to experiment, brainstorm and select candidate solutions, versus people who prefer to carefully plan things out ahead of time. CoSolve, although designed for the former style, seemed to be able to support both without any issues.

Collaborative-Solving Behavior. Solvers also had different collaboration preferences. Some subjects were the lone wolves, who preferred to work on their own. This was a very common behavior, we would often see it in at least one member out of the three teammates.

There were several reasons subjects gave for this behavior. The first category of reasons

came from the subjects who said they wanted to explore and learn the puzzle mechanics first, before they felt comfortable working with others on it. One subject remarked, "I'm the kind of person who likes to explore first, like I need a lot of time to explore on my own, before I feel like I am confident enough to start sharing" (*p1-team5*). Wanting to explore to learn the game was the most common reason.

Still others stated that the time pressure kept them from wanting to explore options; they wanted to go straight down to the end state as quickly as possible, rather than spend a lot of time exploring and communicating with others. Another subject told us that he did not pay much attention to another teammate's work because he figured, "if I get a really high score, our whole team benefits" (*p2-team5*).

This strategy did not always work out for the teams. Several subjects remarked to us, or discovered during the interview, that another teammate actually had a better strategy that they missed out on because they weren't keeping track of others' work. Some mentioned wishing they had been more aware of others' actions or annotations, paid more attention to them initially, rather than ignoring them, because they later found them to be useful. Subject *p1-team1* stated, "I didn't really pay attention at the beginning, until I realized *p3*'s states were better. ... Motivation of knowing that *p3*'s suggestion worked helped me pay more attention later." So even for subjects with lone-wolf personalities, the ability to see other's progress in CoSolve, even while mostly working individually, helped them learn more strategies for solving the problem.

There also could have been other, situational reasons for the subjects behavior. Subject *p3-team3* said that since he did not know his teammates, he acted very differently than he normally would have, and that had he known these people, he would have been far more talkative. However, since he wasn't sure what these people were like or how they would react to his comments, he was more reluctant to communicate with them.

In Team 1, we also found a similar behavior, this time based on the perceived expertise of the team members. In this case, the three members were classmates and did know each other, and one subject stated that the "level of expertise between me and the other two were really different," the others were perceived to be "expert game players" and that this added to the pressure. So this subject wanted to spend some time learning the game alone,

and figured that the other two didn't really have time to teach the subject how to use the CitySim rules. This subject's collaborative activity consisted mostly of reading annotations and and trying to follow what the team was doing, rather than annotating or branching off others' nodes.

In cases where behavior is dependent on the particular makeup of the team, perhaps it is important to impose other structures to encourage teams to get to know one another, or to give subjects structured time to learn the problem on their own. Chapter 5, discusses these issues in more depth.

Other subjects were more collaboration-oriented. For example, *p3-team1* began the solving session by making copious use of the annotations to set up a plan and create intermediate goals for his team. Another subject, *p1-team2*, said he tried to "keep it [his branches] clean" because others were looking at it, so he wanted to make them easy for others to understand. He did so because he "didn't feel confident about describing deep strategy in annotation box, so [he] tried to make self-describing things." These types of subjects consciously tried to structure their actions towards collaboration.

One of the most common collaborative behaviors, that we saw in almost all teams, and even occasionally in some subjects that were more "lone-wolf" types, was a subject alerting others of his or her best node, after which the other teammates would start working there. Often subjects would ask for status updates from their teammates via annotations, i.e. "best node so far?" Or, subjects would decide themselves that a node was particularly worthy. In either case, a subject would annotate their high-scoring node, usually with a thumbs-up annotation. Then, if others saw that it was better than their best nodes, team members would stop what they were doing and go to that node, and begin applying operators starting from that node.

In general, we tended to see more collaboration, in terms of annotation, and planning at the beginning, after which the team members started to work on their own. Some subjects told us that they had difficulty keeping track of their teammates' activity and understanding what others were doing as the tree grew bigger. One interesting point some subjects brought up was the fear of "messing up other people's stuff." There was a sense of each teammates' ownership of branches and some felt that adding on to others' branches

would be interfering with their teammates' work. Interestingly, others said in the interviews that the tree structure was "great" because they could "do this without messing up other's work", since nothing is deleted or edited, only added on to.

Overall, almost all subjects said that the tree visualization was useful to them, both as a paradigm for individual solving–solvers can try different strategies without having to "undo" or edit an existing state–and for collaboration, because solvers can learn from others' work.

Full Consultant (FC) versus Minimal Consultant (MC)

Let's turn our attention now to the Full Consultant (FC) versus the Minimal Consultant (MC) groups. There was no significant difference in score and solution outcomes between the FC and MC groups, though there were differences in the collaborative process. FC users tended to create more neutral annotations than the MC users; on average, 76.5% of a FC user's annotations were neutral compared to 26.8% in the MC condition (two-tailed t-test, p=0.00041, significant). At the same time, the FC users had a lower percentage of positive annotations (19.7% versus 60.3%, p=0.00251, significant). There was no significant difference in negative annotations; both groups created very few of these.

We found that FC users did collaborate more actively with each other in creating a solution than MC users. To quantitatively capture the nature of each team's collaboration, we calculated metrics such as turn-taking and solution path inequality. Turn-taking was defined numerically as the percentage of nodes a user created that were the children of nodes created by other teammates, rather than by the user himself. Turn-taking by teams ranged from 2.2% to 24.8%, with a median of 7.5%. FC condition users had a higher rate of turn-taking than MC condition users (14.6% vs 6.0%, p=0.03, significant). However, the two most equal conditions in the FC condition still were not quite as equal as one would have thought. Team 4's SPI of 10.7 came from a distribution of 3, 8 and 19 nodes on the solution path from each subject, respectively. Team 6's 13.3 SPI was from a distribution of 0, 10 and 20 nodes from each subject.

Solution path inequality (SPI) measures each team's users' contributions on the solution path from the root of the tree to the leaf node containing the final highest-scoring state in

the tree. I define SPI as follows:

$$SPI = \frac{|n(A) - n(B)| + |n(A) - n(C)| + |n(B) - n(C)|}{3} \tag{4.1}$$

where $n(X)$ is the number of nodes team member X created along the final solution path. Perfect equality would be an SPI of 0, i.e. no inequality. Assuming the final solution path is 30 nodes long, this means each team member contributed 10 nodes to this path. An unequal path where one team member created all 30 nodes, and the other two both created zero on the solution path, would give a maximum SPI of 20. All other combinations range somewhere between 0 to 20–the smaller the node count difference between each member, the smaller the SPI.[1] Note though that this does not take into account where or when the turn-taking occurred along the solution path–teams could have alternated one node per member, or one member could have done 10, then another member the next 10, etc.

SPI ranged from 10.7 to 20. The median SPI was 15. MC teams had more inequality than FC (17.8 vs 12.7, p=0.03, significant). So we can presume the Full Consultant caused the subjects to pay more attention to their turn-taking, a hypothesis corroborated by our interview data. As previously noted, in general FC groups tended to work together more at the beginning than the MC group, though they quickly started splitting up.

Finally, in the post-activity questionnaire, we asked subjects to rate the CoSolve Consultant. Table 4.2 shows subjects' responses to these questions on a Likert scale from "strongly agree" to "strongly disagree." The responses shown are divided between the subjects whose teams used the Full Consultant (FC), and those who saw the Minimal Consultant (MC). None of the FC subjects thought the Consultant was difficult to use, and only one MC subject thought so (Q1). 77.8% of the FC subjects agreed or strongly agreed that the

[1] Robison [54] instead defines SPI as:

$$SPI = \sum_{i=1}^{n} |10 - count_{user_i}| \tag{4.2}$$

where n is the number of users on a team, and $count_{user_i}$ is the number of nodes $user_i$ created on the solution path. SPI=0 if all users contribute ten nodes; SPI=40 if a single user contributed all 30 nodes. If Team X's three subjects creates 0, 10 and 20 nodes respectively, then SPI=20. However, if Team Y creates 0, 15 and 15 nodes, then also SPI=20. These two distributions seem intuitively unequal to me. Hence, my version calculates SPI differently: Team X would have an SPI of $13.\bar{3}$ which is more "unequal" than Team Y's SPI of 10. However, even with Robison's definition, the FC group had less inequality than the MC group (SPI=21.4 out of 40 versus SPI=34 out of 40, p=0.04, significant).

Consultant was helpful, versus only 11.1% of the MC subjects (Q2). No one thought the Consultant was difficult to learn (Q3), and more FC subjects thought the Consultant was helpful for teamwork than the MC subjects (Q4, Q5).

Overall, it seems that the full CoSolve Consultant did slightly affect the subjects collaboration behavior, but did not have an effect on overall solution outcomes, either positive or negative.

Pretest versus Post-Test performance

As mentioned earlier, we administered a 12-question pretest before the tutorial began to determine their familiarity with tree concepts, and then a similar post-test after the activity. The tests had a multiple choice section testing graph-related vocabulary, and also had an abstract illustration of a tree and questions like "Find the root node of this tree" or "How many children does this node have?" etc. In the pre-test, subjects correctly answered a mean of 7.83 questions out of 12 total; the median score was 8. In the post-tests, we observed an increase in correct answers: 10.75 mean, 12 median. (Note: this data is for the last four teams, i.e. 12 subjects. Data from the first two teams, 6 subjects total, are not included due to changes in the tests). The FC subjects (n=6) showed a 3.67 increase in their mean scores compared to a 2.17 increase in mean scores for the MC (n=6), though the median score increase in both conditions is the same for both: a 3-question improvement. This suggests that participating in this activity has the potential to contribute to some learning gains in tree concepts, although it is unclear if this could be merely a short-term gain.

4.2.5 User Interface Evaluation

In both the post-activity wrap-up questionnaire (Appendix E) and the interviews (Appendix G), we asked our subjects to evaluate CoSolve's overall solving user interface.

Table 4.3 shows the Likert-scale responses to the questions on the wrap-up questionnaire. As shown in the table, none of the subjects felt the tree visualization interface was confusing (Q1), and 77.8% agreed or strongly agreed that the visualization was helpful (Q2). Our interviews with most of the subjects reflected this as well, e.g. one subject said "Definitely

Table 4.2: Subjects' Likert-scale responses regarding the CoSolve Consultant. Half of the teams were assigned to use the Full Consultant, and the other half to use the Minimal Consultant (as a control) but they all answered the same set of questions regarding the Consultant.

	Control (Minimal Consultant, n=9)					**Experimental (Full Consultant, n=9)**				
Question	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
Q1. The CoSolve Consultant was difficult to use	22.2%	44.4%	22.2%	11.1%	0.0%	22.2%	44.4%	33.3%	0.0%	0.0%
Q2. The CoSolve Consultant was helpful in playing the game	33.3%	11.1%	44.4%	11.1%	0.0%	0.0%	0.0%	22.2%	66.7%	11.1%
Q3. The CoSolve Consultant was difficult to learn	22.2%	44.4%	33.3%	0.0%	0.0%	22.2%	66.7%	11.1%	0.0%	0.0%
Q4. The CoSolve Consultant was helpful in understanding my teams problem-solving process	22.2%	33.3%	22.2%	22.2%	0.0%	11.1%	33.3%	22.2%	33.3%	0.0%
Q5. The CoSolve Consultant helped my team collaborate	11.1%	33.3%	44.4%	11.1%	0.0%	0.0%	33.3%	33.3%	33.3%	0.0%
Q6. The highlighting features of the CoSolve Consultant were not useful	44.4%	33.3%	22.2%	0.0%	0.0%	66.7%	11.1%	22.2%	0.0%	0.0%

the [tree] layout...that was great...if I had a couple good moves, and didn't know which one was good, I'd do all of them, and see the actual numbers instead of doing it in my head, and that was great, because otherwise I'd have to start over."

Regarding the usability of the entire CoSolve user interface itself, e.g. annotations, UI for applying operators, etc. (as opposed to just the tree visualization of a solving session), 83.4% of subjects agreed that the CoSolve interface was "easy to learn" (Q8), and the same percentage agreed that CoSolve was "helpful in playing the game" (Q7). So overall, subjects felt positively about using CoSolve itself for solving. In terms of teamwork, only 5.6% thought that CoSolve made collaboration more difficult (Q9). As for whether CoSolve actually "helped encourage collaboration" (Q4), most subjects either agreed (44.4%) or were neutral (44.4%). So it seems that while CoSolve's UI did not seem to harm collaboration, more could be done to help it. These issues will be discussed in Chapter 5.

The two main issues that subjects had with the user interface was maintaining awareness of their teammates' activity, and overall tree navigation. As for awareness, only 11.1% felt they were "always well aware" (Q11) and 22.2% were "aware...most of the time" (Q13) of what their teammates were doing, and although they did not think that CoSolve made collaboration more difficult (Q9), at the same time half (50%) disagreed that it was "easy to collaborate with my teammates". In the interviews, many subjects told us that they had a hard time figuring out what part of the tree other team members were currently working on.

One possible reason for this is that, as the tree grew larger and solvers started working on different parts of the tree, it became harder to navigate the tree. Several subjects said they were reluctant to pan a very large tree for fear of losing their current working location; as a result, they did not look at their teammates' work as often. Another reason may be that CoSolve's "Refresh" indicator, while designed to avoid disrupting an individual user's view, does so at the expense of some group awareness. Visually, a parent node's children are equally spaced beneath the parent; as each child appeared, existing children are animated to spatially move to make room for the new child. This can be jarring if nodes are moving as a user works on them, and so we implemented the "Refresh" functionality described earlier to allow users to update their views on demand, rather than automatically. Also, in both

Table 4.3: Subjects' post-activity questionnaire responses, as percentages of total number of responses (n=18)

	Question	**Strongly Disagree**	**Disagree**	**Neutral**	**Agree**	**Strongly Agree**
Q1.	The tree visualization of the problem-solving space was confusing	33.30%	55.60%	11.10%	0.00%	0.00%
Q2.	The tree visualization of the problem-solving space was helpful	0.00%	11.10%	11.10%	50.00%	27.80%
Q3.	The tree visualization of the problem-solving space was difficult to navigate	0.00%	11.10%	38.90%	44.40%	5.60%
Q4.	The tree visualization of the problem-solving space helped encourage collaboration	0.00%	11.10%	44.40%	44.40%	0.00%
Q5.	Regarding navigation of the tree, I was often lost or disoriented	5.60%	38.90%	27.80%	22.20%	5.60%
Q6.	The CoSolve Interface was difficult to use	11.10%	44.40%	16.70%	22.20%	5.60%
Q7.	The CoSolve Interface was helpful in playing the game	0.00%	5.60%	11.10%	77.80%	5.60%
Q8.	The CoSolve Interface was easy to learn	0.00%	0.00%	16.70%	77.80%	5.60%
Q9.	The CoSolve interface made collaboration more difficult	16.70%	44.40%	33.30%	5.60%	0.00%
Q10.	The CoSolve interface was helpful in understanding my teams problem-solving process	0.00%	5.60%	38.90%	50.00%	5.60%
Q11.	I was always well aware of what my teammates had done	5.60%	61.10%	22.20%	11.10%	0.00%
Q12.	It was easy to collaborate with my teammates.	0.00%	50.00%	33.30%	16.70%	0.00%
Q13.	I was aware of what my team-mates were doing most of the time.	5.60%	50.00%	22.20%	11.10%	11.10%

the MC and FC versions, we implemented a Refresh Indicator, located in the Consultant menu (Figure 4.14, upper left, labeled "View is up-to-date"), that tells the user how many new nodes have been created, and not yet updated in the user's view, since the last refresh. The Refresh feature prevents interruptions of users' views, but may also decrease awareness of others' actions.

Subjects suggested many solutions they would have wished to see for addressing this issue. For example, one was to have a "mini-map" view of the tree, in one corner of the interface. That way, solvers could work on one part of the tree in the main view, but still have an idea of what the structure of the tree is like, and navigate, using the smaller, corner view of the entire tree. Another idea was to allow users to split the screen to show different parts of the tree on different sides of the screen. This way, a solver could work on one part of the tree, while also following what a teammate is doing on a different part of the tree.

These features would also have been useful for assisting with tree navigation, the other area subjects had the most trouble with. When asked whether the tree visualization "was difficult to navigate" (Q3) 50% either agreed or strongly agreed, and 38.9% were neutral. Subjects told us during their interviews that the panning and zooming controls were unpleasant to use, and they wanted some way to easily examine nodes either across a single level, or up and down from parent-to-children. Subjects were interested in seeing the differences between specific nodes, either to find the best node at a single level, or to find the best child of a given parent. Sometimes they wanted to go up or down a single branch to examine the process by which it was created. Currently, all these actions require panning repeatedly using the mouse cursor; many subjects mentioned they would have liked to either click a button, or use the arrow keys, to jump immediately straight up and down, or across nodes, on a node-by-node basis. Another issue with navigation was due to a failing in our zooming feature–the mouse wheel zoom was buggy, and this caused our subjects a lot of headache. Implementing node-by-node navigation, and working on overall tree navigation seems like the most important next step for improving solving tree navigation.

Finally, almost all subjects brought up the need for a conventional chat box or messaging feature. Many subjects wanted a chance to discuss overall strategy with their teammates before beginning, and while some teams used the root node to do so, this seemed awkward,

and grew difficult as the tree got larger, because subjects would have to jump back and forth between their current working node and the root in order to add general discussion. Sometimes, they would be uncertain exactly which node would be most appropriate to attach an annotation to. Instead, they would have liked a text area in the annotation list for general comments, since the annotation list was always accessible and would eliminate having to decide where to place an annotation.

4.2.6 Discussion

So finally, let's return to our initial research questions:

1. Are human users able to *understand* and successfully use state-space search for problem solving? Or, on the contrary, will they find such a representation too confusing or cumbersome to use?
2. *How* do these solvers use Cosolve-style systems for collaborative problem solving? What strategies do they employ?
3. How could such a framework aid solvers' understanding of their own collaborative problem-solving processes, and as a result, improve their collaborative efforts?

With regards to the first question, our results showed that subjects were able to use CoSolve to solve the CitySim problem, and from their questionnaire response data, we found that they felt CoSolve was easy to learn and use. Subjects found the tree-based solving interface helpful, and were additionally able to learn about tree structures by using CoSolve, as shown by the pretest and post-test results. In their interviews, many subjects said they enjoyed using the tree structure for solving because they "really liked that you could go up and see everything you did, unlike most games, where there's only 1 node, so being able to backtrack was helpful" (*p1-team3*). One subject, *p1-team1*, likened the tree structure to a new style of collaborative gameplay, remarking that he "really enjoyed it, I was surprised. I'd never done something like this, played a game in a new way, sort of a different style of playing a game that I've never experienced before, more like a bunch of people working together on a big jigsaw puzzle or something ... the moment I realized

other people had better ideas than I did, and being able to compare things in a row in this visualization to compare from each others ... then I cared about this [interface] a lot."

In terms of the second question, we were able to discern different team and individual subject behaviors from our interview data and directly through examining the solving session trees. For example, we saw how subjects made use of turn-taking and branching. We also examined collaborative behaviors, such as subjects learning from their teammates' branches, or subjects annotating nodes when they had found a good technique to share with others. Others would see that score, and then build branches from that node, perhaps applying their own techniques, enabling the team to get a higher score overall. CoSolve's ability to show everyone's work at once enabled solvers to learn. On the other hand, we also saw that some subjects preferred working alone, either as a general personal preference, or due to situational factors. We also identified areas of improvement for CoSolve's solving user interface, such as with the annotation/chat feature, and tree navigation.

Finally, we examined the third research question by implementing the CoSolve Consultant (Full Consultant), and testing it against a control condition version of the Consultant called the Minimal Consultant. We saw that the Full Consultant interface had a positive impact on users' team behavior, though not their team's actual solution scores. Users who had access to the Full Consultant were more effective collaborators: in those teams, each user contributed more equally to the solution than in teams where the users did not have access to it.

In this section, we have described the results of a user study which showed that overall, subjects find this novel problem-solving interface helpful, easy to learn and easy to use. We have also identified areas for improvement of the solving interfaces, and we have identified behaviors that emerged from this unique solving paradigm. We found that the tree structure allowed solvers to explore and share solutions in ways they would not have been able to in a non-tree based interface, and we also saw a contrast between different styles of collaboration, which we will address in Chapter 5.

Our focus so far has been on the use of CoSolve in a laboratory setting and to solve a fairly constrained and simple task, the CitySim problem. What about the use of CoSolve for solving more free-form, creative problems? Can CoSolve be used for open-ended problems,

problems without obvious goal criteria? In the next section, we will continue to discuss the CoSolve solving process, but in a different context: the design of educational games.

4.3 Using CoSolve for Design Problems

So far we have mostly observed the use of CoSolve to solve problems for which there is a clear evaluation function with which to analyze potential solutions. In CitySim, a city's state either has more money than another state, or it doesn't. But what about problems for which the solution is more subjective, for which there could be many acceptable solutions, and the goal state, and even the operators themselves, are not obvious? Or perhaps each member of the team of solvers have different ideas of what a good solution. One example of this type of problem is in *design*.

Herbert Simon pioneered the idea that design is in essence a problem-solving activity [1], and that designing is the process of searching for an ideal design in the "design space" of all possible designs. This is, of course, analogous to what CoSolve was created for: to let solvers create and search through a space of possible solutions. But design is different from traditional, puzzle-like problems. Design, by real humans and for real humans, can be a subjective matter, and trying to solve a design problem involves having conversations with other stakeholders and designers. There may not be a single design that *satisfies* all involved parties, and so the design process involves finding a solution in the design space that *satisfices*, such that the group can come to a consensus.

Our analysis thus far has demonstrated the use of CoSolve as part of a constrained, formal user study. Let's now look at how CoSolve can be used more freely as part of a larger solving process: the problem of designing an educational game. This study will differ from the CitySim study in two main ways. First, instead of solvers working on a single, *very poseable* problem that was posed for them, CoSolve is used in a more free-form manner as part of an educational-game design activity, where the designers iterated between posing and solving. Secondly, CoSolve was part of a larger design process that included face-to-face meetings and use of other communication tools.

This design exercise was undertaken by a team of six in the context of an graduate educational technology seminar at the University of Washington in 2010 and our observations

were published in [58], and a longer report can be found at [59].

Our design team consisted of the author, three more graduate students in computer science and two faculty members, one in architecture and one in computer science, participating in a seminar on the topic of problem solving and design, over the course of a 10-week academic term. The goal of the activity was to study design by actually designing two different games and playtesting them, to examine how communication tools can be used to aid this design process. The tools we used were Google Wave [60], CoSolve, and INFACT [61]. The two resulting games were called Eco-avelli and Go Atom.

We will not focus too much on the other two tools here, but let us briefly describe what they are. INFACT is an online, private-access threaded discussion forum providing asynchronous communication, with additional tools for educational assessment. We used only its basic system, including a feature called the curtain (a means to temporarily hide responses to a question, so participants can all respond before seeing the answers of others). The second tool we used was the now-defunct Google Wave [2]. Each "wave" was a group-editable, shared online document, whose features mixed those of the interactive documents in Google Docs, and the traditional, threaded message forums and chat rooms. Users are permitted to edit the contents of a wave simultaneously in a way that feels like typing in a chat session; others can see each character as it is typed, but they can also create hierarchical threaded posts and edit others' posts. A wave is like a wiki with a synchronous feel.

The ideas for both Eco-avelli and Go Atom were developed within our group during the seminar. Eco-avelli is a card game meant to model the political processes involved in the issue of global warming, and we used Google Wave and CoSolve for our communication. Go Atom is an elementary chemistry education game. For this game, we investigated using INFACT and CoSolve to support our design work. CoSolve's annotations facility was not fully implemented, in the form described earlier in this chapter, at the time of the study, so annotations weren't used in our design activity. In designing each of the games, we met face-to-face over several, roughly one-hour-long sessions, and we also used the online tools to communicate between sessions and during sessions. In each session, we completed one or two design iterations, beginning with a discussion of problems in the current version of the design, then brainstorming and deciding on a modified set of game rules, playing the

game, and analyzing how well the new rules enhanced or detracted from the game. We ended up creating three CoSolve problem templates[9] related to this design activity: "Go Atom Game Design", "Eco-avelli Card Game" and "State Space Design". We will talk first about designing Go Atom.

4.3.1 Go Atom

Game description. Go Atom involves basic organic chemistry concepts. Players are dealt a hand of Carbon, Oxygen, Hydrogen and Nitrogen atoms. An initial Carbon atom begins play. In turn, each player bonds an atom, from his own hand, to an atom currently in play, creating a molecule consistent with chemistry bonding rules. Players gain points for each atom or bond they add to the structure, and the game ends when no more bonds or atoms can be added that follow these chemistry bonding rules. The player who ends the game gets to claim the molecule, and may receive points for it.

Design Process. The design process began with initial game idea proposals from all team members, which were posted as messages to INFACT. Once we decided on the Go Atom game idea, additional messages were posted to INFACT to hash out the details of the design. Then, after we had come up with a basic idea, before the first face-to-face design session (Session 1), a problem template was created in CoSolve to describe the state space of the game design. In designing this game, we wanted to explore how changing different aspects of the rules might affect the gameplay, while maintaining our two goals of maximizing the playability of the game (i.e., is it fun, enjoyable, easy-to-learn?), and teaching chemistry concepts. We hypothesized that by changing our scoring scheme, we would be able to come up with the right gameplay dynamics to achieve this balance, so our CoSolve template reflects that.

Here is a list summarizing the state variables in our first iteration of Go Atom's design:

---

[9]The problem templates were primarily created by Rolfe Schmidt. They can be viewed online at http://cosolve.cs.washington.edu/problem/go-atom-game-design, http://cosolve.cs.washington.edu/problem/eco-avelli-card-game, and http://cosolve.cs.washington.edu/problem/state-space-design.

1. Scoring scheme for adding each type of atom and each bond. This was implemented as a Python dictionary within the state variable dictionary:

```
S = { 'add_score' : {"C": 0, "N": 0, "O": 0,  "H": 0, "Bond": 0} }
```

2. Scoring scheme for claiming completed molecules, including points for each atom type and each bond. This was also implemented as a Python dictionary, similar to the above.
3. Specification of the number and types of atoms in the initial hand of each player. Also implemented as a Python dictionary, similar to the above.
4. Whether or not initial hand is random. Implemented as a Boolean variable.
5. If random, minimum allocations of atoms in the initial hand. Implemented as a Python dictionary similar to the above.

The following operators were created to set these variables:

- Increase "claiming" score by 1 (Parameter prompt: Which game element do you want to increase the score of? C, N, O, H, or Bond?)
- Increase "adding" score by 1 (Parameter prompt: Which game element do you want to increase the score of? C, N, O, H, or Bond?)
- Increase initial hand by 1 (Parameter prompt: Which game element do you want to increase? C, N, O, H, or Bond?)
- Increase minimum allocation of random initial hand by 1
- Set initial hand to random
- Set initial hand to deterministic

At that point, the team initiated CoSolve solving sessions where they created state nodes containing possible sets of scoring rules for the game. After this, we met for a face-to-face design meeting. We held two face-to-face design sessions total, where we played through four design iterations: Iteration 1 and Iteration 2A & 2B (simultaneous iterations involving half the group in each iteration) during Session 1, and Iteration 3 during Session 2. In between the meetings we would plan what we would do in the next face-to-face meeting

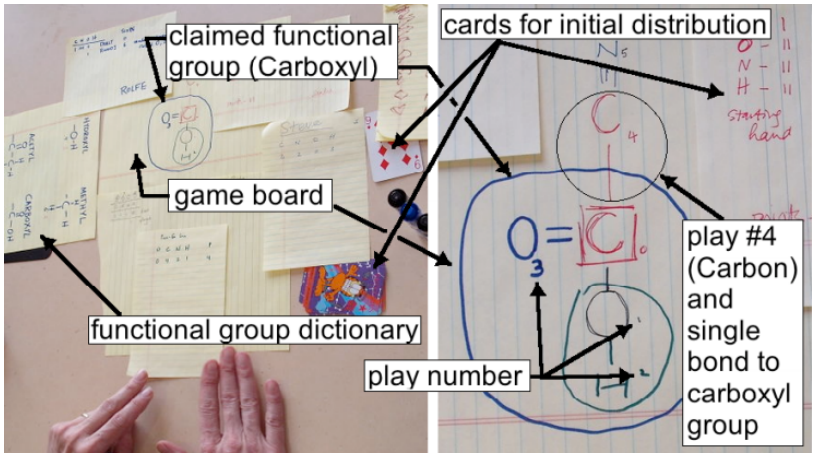


Figure 4.23: In the photo on the right, groups of atoms that form a functional group are circled by the player that won points for that group.

via more messages on INFACT. We also used INFACT for recording notes of our meetings during the session itself.

During these face-to-face meetings, we tested our game designs playtesting with paper prototypes. Figure 4.23 shows two photographs from Iteration 3. As seen in the photos, playing cards were used as a way to randomly pick the initial hand of atoms of each player. On the right, we can see a "play number" (turn number) at the bottom of each atom; we used these to record and identify each move made by each player during our play testing. Instead of, say, playing an entire game directly in CoSolve, which may have been inappropriate for a board game, or for a game at such an early design stage, we used CoSolve to help organize our game rules for each design iteration so that we could arrive at our face-to-face meetings ready to playtest, rather than spend a lot of time figuring out game mechanics.

Some iterations were devoted to fixing game-play issues, such as balancing scoring (e.g. Iteration 1). Others focused on the educational value of the game. For example, in Iteration 3, we brainstormed ideas to increase the chemistry content in the game. We decided on a Scrabble-like version of GoAtom, with points awarded for building molecules that were real chemical compounds, rather than simply any that fit the basic bonding rules. We tested this idea by adding a small dictionary of functional groups from organic chemistry, and awarded

extra points for building or identifying these groups. We found that this did increase our ability to identify functional groups, so we further discussed ways a more general dictionary of molecules could be developed for the game. This is an example of a design decision could not be accounted for in the initial CoSolve state space; it required adding additional variables to the state space, which meant we had to modify the original problem template by acting as posers, rather than just adjusting scoring values as solvers. This is an issue we will see repeated in our design of Eco-avelli.

4.3.2 Eco-avelli

Game description. The name Eco-avelli is a reference to how, in this game, Machiavellian techniques are at odds with sacrifices made for the greater good. We played the game with a regular deck of playing cards, and the game was intended to make players aware of the kinds of trade-offs nations must make in deciding how they will approach the issue of climate change.

Each player has a certain number of hidden as well as open cards in his or her hand, which are that player's "resources". Players take turns drawing cards from the deck and adding them to their hands. The more matching cards ("combos") a player has, the more points that player earns. We devised a scoring scheme that specified how many points each combo–single cards, pairs, three-of-a-kind, four-of-a-kind–would generate, representing the concept of countries being more powerful if they have monopolies on resources. However, on each turn, players are also allowed to request specific cards from each other, mimicking trade between countries.

Finally, there are also "convention" rounds, in which players vote to "ban" certain ranks of cards from being able to accumulate more points. The purpose of these convention rounds are to simulate climate change conventions where individual countries within the international community must negotiate what restrictions to put into place to limit carbon emissions for the greater good, while keeping their own country's economic interests in mind.

Design process. In this design process, we again tried to balance the two goals of playability, with faithfulness to the educational aspect of the game. There were five design sessions

of Eco-avelli in total, with eight iterations of the game during those sessions, including the first game session that was played before the official start of this study (Iteration 0). In Sessions 1 through 3, there were two design rounds each (Iterations 1 through 6), and in Session 4, there was one (Iteration 7). This time we used Google Wave for communication, and again used it both between meetings, and to take notes during face-to-face meetings. Most of our communication in Wave was in the form of recording what had taken place, the ideas that had been discussed, or the design decisions that had been made.

Here is an example of the kinds of state space variables used before beginning Iteration 1. This problem template was quite complex, with several nested dictionaries, so I will not list all the code details of each variable here. The ones listed as "rules" were usually created by listing all the possible rules under a related set of rules (e.g. rules for banning: majority vote, secret voting, open voting), and setting them each to be either true or false.

1. Size of each player's initial hand of cards, e.g.

```
S = {'Initial Hand': {'visible cards': 0,
                      'hidden cards': 3,
                      'bidding cards': 0}}
```

2. Scoring scheme for combos
3. Number of visible/hidden cards in a player's hand,
4. Rules for banning: effect of a ban
5. Rules for banning: majority vote? secret or open voting? etc.,
6. How are cards transferred between players, if allowed at all?

In most of our iterations, instead of simply modifying one of these variables, we again found that we often needed to add a whole new variable to the state space all together. For instance, after Iteration 1, we found that since a player could declare victory immediately after drawing, if other players had not had their turn that round, the game was unfair and ended too quickly. So we brainstormed a variety of new rules. Among them was a new rule which states you cannot declare victory immediately after your turn; you must wait until everyone has had another round of play. This is a rule that was not accounted for in the

original variables of the state space, and hence, a variable was created for it, the "declare victory after turn" variable, and the variable's value is initially set to false. However, once we included this variable in our state space, we never changed its value again, which reflected a shortcoming in the way we created the problem template–for this game design, we tended to simply want to add variables, rather than just change them.

In CoSolve, we created an initial problem template that described the state space as we viewed it after playing the game in Iteration 0 (before we identified the example state variables shown above). After Iteration 4, we found that we had increased the size of the state space considerably in our iterations of the game, and posted a revised list of state variables into Wave, and then translated this into a revised CoSolve problem template, which was considerably larger. We then attempted to record all our previous iterations using this new state space model. However, after doing so, we did not make use of this CoSolve template in later iterations, as we found our state space kept changing and it required us to keep modifying the initial problem template.

4.3.3 State Space Design

Our main observation in these design activities was that we were generally unable to anticipate the state variables to be used, and most design iterations caused us to create new state variables, rather than simply modify an existing one. So, as a result of our experiences, we designed a more general-use problem template, a "meta-template" that would enable us to add, remove and modify state variables themselves. The initial state of this template, simply, looks like this:

```
S = { 'state variables': {} }
```

The operators were as follows. Solvers could add several types of variables: integer, boolean, or enumerated type. Enumerated type allowed solvers to create their own variables types, e.g. "Days-Of-Week" and specify the possible values of that type, e.g. "Monday", "Tuesday", "Wednesday", etc.

1. Add Enumerated Type Variable

2. Add Integer Variable
3. Add Boolean Variable
4. Edit State Variable
5. Delete State Variable
6. Add Values to Enumerated Type
7. Delete Values from Enumerated Type

This template gave us greater freedom because of the ability to add variables at the solvers' discretion. We then used this template in later game design activities. For example, one design activity involved creating a flashcard memorization game. Flashcards were laid out to form a maze. Players must correctly remember and answer each flashcard to navigate and progress through the maze. Another example is an auction economics game, again using playing cards. We were able to describe both games using this same template, rather than having a new template for each game.

Summary

These game design studies were exploratory, and conducted with an earlier version of CoSolve from 2010 that was not as full-featured as the current version. As a result, secondary communications tools were used, and these studies do not provide much information on what role CoSolve annotations might have played in the design process. Nevertheless, these game design examples demonstrate another style of CoSolve solving, and how CoSolve can be used as a design tool. In this case, CoSolve was not used directly to generate and evaluate possible solutions (candidate designs), but instead used to support design processes in concert with other communication tools, and as a way to communicate outside of face-to-face meetings, and to take notes during meetings. Evaluation and the designs was still done outside of CoSolve during playtesting, unlike in problem templates such as CitySim, where each state was evaluated directly within the solving session interface. We saw how CoSolve's role in the process included both creating problem templates, and using solving sessions to record and communicate design ideas, and we have demonstrated how CoSolve can be viable and flexible tool for any kind of collaborative problem-solving activity.

Conclusion

In this chapter, we have discussed CoSolve's problem-solving in detail, both from a theoretical viewpoint and through two practical examples. In the CitySim user study, we tested CoSolve's solving interface and found that users were able to successfully use and understand it. We also were able to evaluate our solving user interface, as well as study solver behavior and collaborative strategies. Then we discussed a different way that CoSolve can be used for solving: as part of a design process that included other tools, as well as both creating problem templates and using solving sessions, to record and communicate design ideas. These examples have shown that CoSolve is a viable and flexible tool for collaborative problem-solving activity.

The CitySim study brought to our attention some of the collaboration imbalances that can occur in a CoSolve team solving situation, with some users being reluctant to participate fully for various personality or situational reasons, and others ignoring their teammates. In the following chapter, we will introduce collaborative roles, which we developed to help address these issues.

Chapter 5

COLLABORATION ROLES IN COSOLVE

In the previous chapter, we looked at how solvers behaved in a collaborative problem-solving activity. In general, solvers found the CoSolve interface helpful, but at the same time, we found that not all solvers were participating as fully as they could have been. For example, as mentioned before, one of the subjects thought the other two members of the team were more experienced with computer games, and felt that she would only slow the rest of the team down. This subject said, "[The team's communication] was through annotations...mostly I just read what they do or suggest, I wasn't really voicing what I thought much because I was just trying to follow what they were doing. That helped me understand the game, but I wasn't able to contribute too much to this whole thing."

This echoes some of what we've seen in the Polymath Project discussed in Section 2.5.1. Some of the participants felt intimidated by the status of some of the other mathematicians, others were unsure where they could contribute, and so these participants contributed very few comments. However, the study by Cranshaw and Kittur [37] found that even participants who posted relatively fewer comments still had an important impact on the final solution. Could it be that, had these barriers to participation in both situations been removed, the collaborative problem-solvers would have been able to come up with better or more efficient solutions?

With regard to unequal team member status, team members who perceive themselves to be of lower status may feel uncomfortable criticizing the ideas of higher-status members. Or they may feel uncomfortable suggesting new ideas, for fear that more expert teammates may find these ideas useless or incorrect. As for uncertainty about how best to contribute, without structure, some participants are unsure what to do at any given point–should they be brainstorming new ideas? Should they be reading through past ideas and commenting on good ones? Where in the problem-solving discussion can they be of most help? Is there a

way CoSolve could actively encourage participation by suggesting to users how they can be the most helpful? Additionally, is there a way CoSolve could encourage this participation to be more equitable–instead of just one or two people contributing to most of the process or the final solution, how can everyone be encouraged to participate equally?

Perhaps CoSolve could provide structure by assigning team work tasks to different members of a team. In terms of CoSolve, this might mean creating a certain number of nodes, or it might mean evaluating nodes and annotating them for the benefit of the team, which will then help node-creators find the best places to continue creating nodes. This will also increase communication between members in general, by encouraging them to look at other nodes. Additionally, this addresess the higher-status teammate issue as well. If CoSolve specifically assigns a solver to negatively annotate nodes, perhaps that solver would feel less reluctant to, say, criticize higher-status members, since everyone knows it is currently his assigned task to criticize everyone's ideas.

We divided these specifically-assigned tasks into user roles in CoSolve. From a social perspective, a role defines the relationship between members of a group. The general function of roles is to manage expectations between members of a group. From the point of view of software applications, users' roles have almost always been implemented as access control systems, i.e., as a way to specify permissions given to a user or group of users. These permissions control access to tasks (who can do what), objects (who can view, edit, create or delete what), and information (who can know what). This type of role manages expectations by providing users with information about what functions they can expect others to perform, as well as what functions they themselves should perform. This is an important function because it helps users predict others' behavior. It is also important for user interface (UI) design. In a large collaborative group, it can be difficult for users to maintain awareness of others' activity, making it difficult for them to coordinate their work. Having a set of roles can help the system determine what activity to alert users of; for instance, if a solver is trying to find the best place to apply operators, it would be helpful to see where the positive-annotation creator last created annotations.

Another way roles might increase participation is, once users have been assigned designated roles in the system, they may feel an increased sense of ownership and pride in their

work. Pride and ownership in one's work has shown to be an important aspect of collabs [48], and developing an identity in the community, as discussed earlier, often times in a particular area of expertise within an open source project, is important to one's progression in an OSS community (Section 2.5.2). Perhaps giving users specific tasks to be in charge of will encourage them to participate more.

To explore how we can increase and equalize participation in CoSolve, we conducted a user study to examine the impact that such collaboration roles might have in the problem-solving process. By "collaboration roles", we mean roles related to general teamwork and how the team interacts with each other. This is opposed to domain-specific roles; for example, a team of automotive engineers may include those whose domain-specific role is safety engineering, others may be experts in vehicle dynamics, or aerodynamics, or quality control, etc. In PRIME Designer, CoSolve's predecessor, described in Section 2.3.1, the PRIME design task was divided into several topic areas–architecture expert, music expert, image processing expert, and so on. So architecture solvers mainly applied operators related to architecture, for instance. In a preliminary user study [18], we observed that our division of roles caused an unequal balance of both work and mental effort. Some participants had a lot of easy work to do, others had very few, but extremely challenging, actual tasks to perform. So, in this CoSolve study, we chose instead to focus on collaboration roles, and let solvers divide up domain-specific tasks as they wish.

The collaboration roles in the CoSolve Roles system are:

- Brainstormer
  - Description: Brainstorm ideas and generate a large variety of alternatives and possible solutions
  - CoSolve Action: Apply operators to create new nodes
- Supporter
  - Description: Provide arguments or comments in support of ideas created by brainstormers
  - CoSolve Action: Provide evaluation of nodes and branches by adding positive (thumbs up) annotations to nodes

- Critic
  - Description: Provide arguments or comments in rejection of ideas created by brainstormers
  - CoSolve Action: Provide evaluation of nodes and branches by adding negative (thumbs down) annotations to nodes

The solving-session initiator can enable the CoSolve Roles system when she creates a new solving session, alternatively, roles can also be turned on any time during the session. When enabled, the CoSolve Roles system assigns roles to users and alternates which user is currently in which role. This system is described in more detail in Section 5.1.1.

Finally, we'd like to mention one more reason for exploring the use of roles in CoSolve. One of the motivating reasons for developing CoSolve was to use it to teach the process of problem-solving. It could be argued that, since in the case of both Polymath and CitySim, participants were able to come up with solutions, perhaps the issue of increasing participation is unimportant. However, we would like to encourage participation for the sake of engaging people in the problem-solving process itself, not just for the sake of the final solution. In the long run, involving more people in the problem-solving process will spread knowledge and expertise in problem-solving itself, and participants can improve their problem-solving skills.

We have now discussed some background and motivation for the CoSolve Roles system. In the next section, we will discuss the user study we conducted to examine what effect assigning such roles would have on the users' collaborative behavior. We explain the design of the CoSolve Roles system developed for the purposes of this study. Then, we present the results of the Roles user study and close with discussion on the use of roles in CoSolve.

5.1 Roles User Study

In our user study, we wanted to examine whether assigning collaboration roles to users in an environment such as CoSolve would affect their teamwork and solving behavior. In particular, we wanted to see if we could alleviate some of the collaboration issues we mentioned in the previous section. At the same time, since CoSolve is also meant to help solvers

learn about their own problem-solving behavior, we hoped that our roles system could help solvers reflect on their own collaborative behavior.

We conducted this experiment to answer the following research questions:

1. Can assigning users to collaboration roles encourage *more participation*, and possibly *more equitable* participation, in problem-solving environments?
2. Does assigning collaborative user roles in problem-solving environments result in overall better solutions to problems?
3. In what ways could engaging in an activity that assigns collaboration roles affect users' attitudes toward collaboration? Does it affect awareness of their team's collaborative activity, the quality of their teamwork, and the amount of collaboration that occurs?

To examine these issues, we developed a collaboration roles system, called CoSolve Roles, that we integrated into CoSolve. The details of this system are described in the following section, Section 5.1.1. Then we again had teams of three subjects work together to solve the CitySim problem, and compared their performance with a control group consisting of teams that used CoSolve without the roles system.

To answer the first research question, we will examine whether the roles participants performed more actions in the CoSolve system–created more nodes and annotations–than the control groups. To determine whether the subjects participated equally, we will also examine whether the overall number of actions performed by all the participants were roughly equal, in addition to whether they contributed equally to the final solution. For instance, if 30 nodes were created, did each subject create roughly ten of them?

To explore the second question regarding solution quality, we compared the resulting high-scoring CitySim solutions from the roles and the control groups. Finally, for the third research questions, we administered surveys to subjects before and after the activity, to uncover any attitude changes that may have occurred from using the roles system. Additionally, we compared subject attitudes between the roles and the control groups. We also examined data from subjects' interviews to determine affect and attitude of the solvers toward the roles interface and collaboration in general.

Next, we will describe the CoSolve Roles feature that we designed for this user study.

5.1.1 Roles User Interface Design and Implementation

The CoSolve Roles feature did not change any of the existing functionality in CoSolve solving sessions. CoSolve Roles can be enabled on a per-solving-session basis, and could be enabled by experimenters without affecting the solving session tree. When CoSolve Roles is turned on for a solving session, experimenters can access a special Roles Administration page for that solving session. This page allows experimenters specify which users in the system should be assigned roles for that solving session.

As for the solvers' user interface, CoSolve Roles only added one additional user interface feature, the Roles User Interface, or Roles UI, which was a box placed on the left side of the screen above the annotations, as shown in Figure 5.1. The Roles UI consists of a status area at the top of the box, and a set of three tabs that the user can switch between that shows the user's progress in his or her role. The three tabs are the "Current Role" tab (Figure 5.2), the "Overall In-Role" tab (Figure 5.3), and the "All Actions" tab (not used during this study). These will be explained in more detail later in this section.

In the user study, there were three roles, *Brainstormer*, *Critic*, and *Supporter*. We initially had a *Team Wrangler* role, whose job was to provide the team with direction and goals, and enforce role compliance, but we eliminated this role due to time and participant recruitment constraints. We felt that for such a small team, and for a study conducted during a short, in-lab session, a Team Wrangler was less necessary than for a solving session with a large team distributed over a period of several days or weeks.

If a user is assigned the Brainstormer role, his task is to help the team by generating ideas. In CoSolve, this means applying operators to create new nodes in the solving session tree. We will call such actions *in-role actions*. The job of Critics and Supporters, on the other hand, is to evaluate existing ideas for their flaws and merits, respectively. This translates into CoSolve as negatively or positively annotating nodes. A Critic in the Roles system should review nodes in the tree, looking for those nodes that are less promising as possible solutions, and create negative, or "thumbs down" annotations on them, and then in the annotation, provide the reasons why she is negatively annotating the node. A Supporter in the Roles system does the opposite, he looks for promising nodes and creates

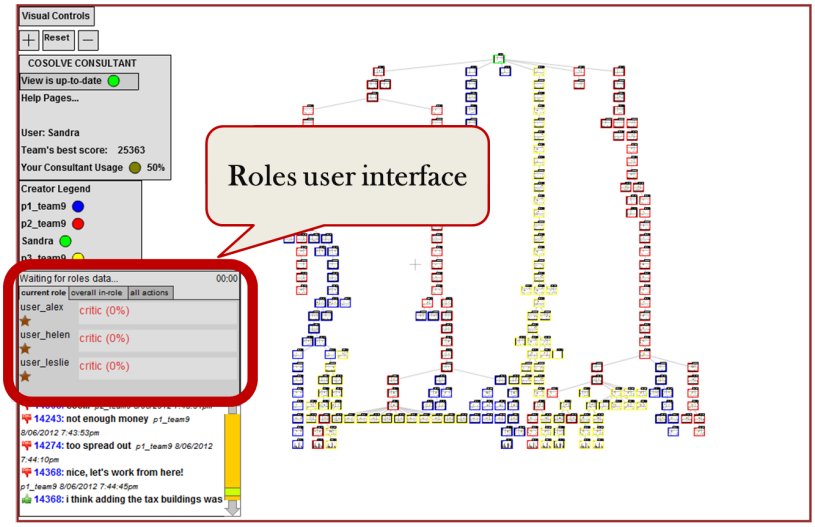


Figure 5.1: This figure shows the placement of the Roles User Interface within a solving session. The Roles UI box on the left, above the Annotations List, is the only visual difference in the user interface, compared with the regular Solving Session user interface. The Minimal Consultant, shown here, was used during the Roles User Study.

positive, "thumbs up" annotations, and explains his reasons for doing so in the text of the annotation. Hence, the in-role action for a Critic is to create negative annotations; the in-role action for a Supporter is to create positive annotations.

Our first design decision was how to assign roles to users. We wanted to make sure everyone had a chance to experience every role more than once, so that they would have a chance to create nodes based on feedback received, or to provide several rounds of feedback throughout the whole session. At the same time, we wanted to maintain a reasonable user-study session duration for the subjects, but we wanted subjects to have enough time to spend on each role before switching roles. We also wanted the assignment of roles to make sense, for instance, if a user is assigned to the Supporter or Critic roles at the start of the session, then that user has little to do because there are not enough nodes in the tree yet to annotate. This means that all users should start as Brainstormers, but in general, we wanted at least one person in each of the three roles at any given time so that there would be a sense of responsibility for being in that role; e.g., we want each user to feel that he or

she is the critic right now, and so that is his or her current responsibility to the team. Also, during the pilot studies, we found that if all users were in the same role, then when users were assigned Critic and Supporter roles, they did not have a constant supply of new nodes to annotate, and ended up not doing much during that time.

To try to satisfy all of these properties, we performed several pilot studies to find the right balance for assigning roles, and came up with the "role phase" schedule given in Table 5.1. We divided up the duration of the user study activity into phases, and inserted rest breaks for our users between sets of three phases. During the first fifteen-minute phase, all solvers are assigned to the Brainstormer role. This is the longest phase, and gives solvers a chance to familiarize themselves with the problem. Then during the next five-minute phase, all solvers are Supporters, so that they have a chance to try out adding positive annotations in the context of this solving session. For the third five-minute phase, all solvers are Critics, so that they could do the same for negative annotations. After these initial three phases, all users should have had a chance to learn the roles system as it relates to the problem they are solving, and the solving session has been seeded with some nodes and annotations for solvers to build from. From that point forward, the CoSolve Roles system rotates between the roles, such that there is always at least one user in each role, and a solver is never given the same role twice in a row (unless they have had at least a break).

Our next design decision was regarding how strict we should be with these roles, and how to enforce them. Strict roles would mean that solvers would not be able to perform any actions other than those associated with each solver's assigned roles. Flexible roles would mean that, although solvers are assigned a role, they do not have to act within the confines of the role unless they wish to do so. On the one hand, if the roles are completely flexible, users might ignore them completely. While this might be fine in practice since we wouldn't want to force solvers to deal with functionality they don't want to use, in this user study, we specifically wanted to test the effect of roles. Hence, we wanted to make sure the users actually paid attention to their roles. On the other hand, overly strict roles might frustrate users, especially given the very limited time they had during a user study session.

In the end we decided on a compromise. When a user is assigned a role, there is a *goal number of in-role actions* associated with that role phase. The user is permitted to perform

Table 5.1: Role phase schedule. PID is the Phase ID. This table shows the ordering of role phases and the goal number of in-role actions for each of the three subjects during the user study. Brainstormers had to create a goal number of nodes by applying operators. Critics and Supporters had to make a goal number of negative or positive annotations, respectively; these annotations did not have to be on unique nodes. Subjects were given five-minute rest breaks between sets of three phases.

		Participant 1		**Participant 2**		**Participant 3**	
PID	**Time**	**Role**	**Goal**	**Role**	**Goal**	**Role**	**Goal**
1	15 min	Brainstormer	10 nodes	Brainstormer	10 nodes	Brainstormer	10 nodes
2	5 min	Supporter	1 pos. anno.	Supporter	1 pos. anno.	Supporter	1 pos. anno.
3	5 min	Critic	1 neg. anno.	Critic	1 neg. anno.	Critic	1 neg. anno.
4	*5 min*	*BREAK (no roles)*					
5	8.3 min	Brainstormer	16 nodes	Critic	4 neg. annos.	Supporter	4 pos. annos.
6	8.3 min	Supporter	4 pos. annos.	Brainstormer	16 nodes	Critic	4 pos. annos.
7	8.3 min	Critic	4 neg. annos.	Supporter	4 pos. annos.	Brainstormer	16 nodes
8	*5 min*	*BREAK (no roles)*					
9	8.3 min	Supporter	4 pos. annos.	Brainstormer	16 nodes	Critic	4 neg. annos.
10	8.3 min	Critic	4 neg. annos.	Supporter	4 pos. annos.	Brainstormer	16 nodes
11	8.3 min	Brainstormer	16 nodes	Critic	4 neg. annos.	Supporter	4 pos. annos.

other actions during that phase, but extra incentives are provided for completing the goal number of in-role actions. For example, say a user is assigned to the Supporter role for a ten-minute phase. The goal number of in-role actions for that particular role phase might be five negative annotations. During the ten minutes that the user is assigned a role, she can create five negative annotations to fulfill her in-role actions goal, but may also perform other actions, such as creating nodes or positive annotations.

Then, to encourage in-role actions, we incentivized users to perform in-role actions in several ways. First of all, we told subjects that they would be given an additional CitySim points bonus, if they managed to complete their role phase goals most of the time. (This was the same incentive bonus as we gave to the Minimal Consultant and Full Consultant teams in the CitySim User Study, as described in Chapter 4.) Recall that the team with

the most points receives an additional gift card award at the end of the study. Secondly, we used visual user interface mechanics to provide a form of incentive. As a user performs in-role actions, a colored "roles progress" bar fills up (shown in the "Current Role" tab of the Roles UI in Figure 5.2). When a user has completed all the actions for a role phase, a greyed-out star changes to a bright yellow, "lit up" star next to their name. This star serves as a visual reinforcement "reward" for the user, in much the same way as a flashing lights and bells on a casino slot machine provides stimulation for a gambler. Additionally, the "Overall In-Role" tab in the Roles UI shows the stars that a user has "collected" over the course of all the phases in the study so far (Figure 5.3). All solvers on a team can see other members' progress bars and stars, providing an extra, social incentive for solvers to fulfill their roles: if a team member is not carrying his weight, everyone else knows it.

Finally, to find a reasonable goal number for in-role actions, we performed several pilot studies, and set the goal numbers accordingly as shown in Table 5.1. We found that it took significantly more time to create annotations, positive or negative, than it took to create new nodes. Brainstormers were usually able to create several nodes in the same period of time it took for Supporters or Critics to evaluate new nodes, find one that warranted an annotation, and then type in the annotation text and add the annotation. After some experimentation, we found that solvers could reasonable create new nodes at a rate of one per 30 seconds, and could create thoughtful annotations at a rate of one every two minutes, so in our final role phase schedule, we set the goal number of in-role actions according to these rates. This was the final way in which we attempted to make our roles structure stricter without actually restricting our user's actions: we made the schedule tight enough that a subject would not have much time left over for non-role (out-of-role) actions during a role phase. The exception to this is in the first three phases, when subjects were given more time so that they had a chance to get used to the system.

The Roles UI is shown in Figures 5.2 and 5.3. As mentioned earlier, the last tab, "All Actions" was not used for the user study, and its contents are not shown here. The status area at the top of the Roles UI, shows what the user's current role is, what the user's next role is, or "break" if the user is currently on break or has an upcoming break, and shows the time left for the current role phase. In the "Current Role" tab, the user is shown each

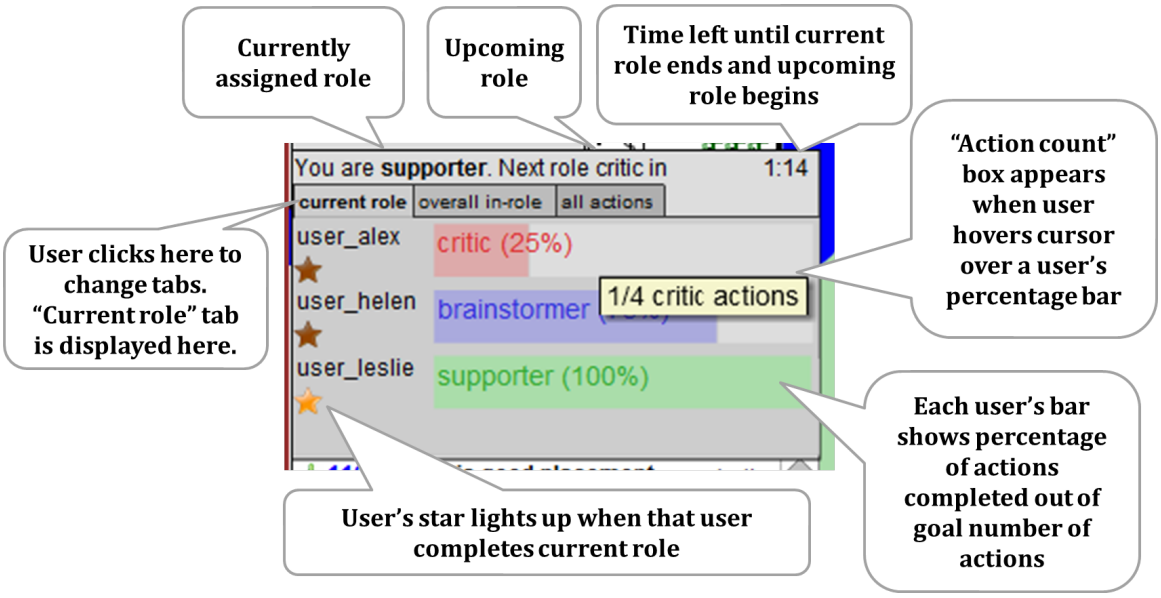


Figure 5.2: The "Current Role" tab is the Roles UI shows each user's progress in their assigned roles for the current role phase. As users complete in-role actions, the bars next to their names fill up until they reach their goal number of actions for their role. User may perform non-role actions in the solving session, but they do not fill up the bar. Once the bar fills up, the star next to the user's name is lit up. Hovering over a bar brings up a box that gives a count of actions performed over goal actions, e.g., "1/4 critic actions" for *user_alex* in the example above means that *user_alex* has completed one out of the four negative annotations that are the goal for his particular role.

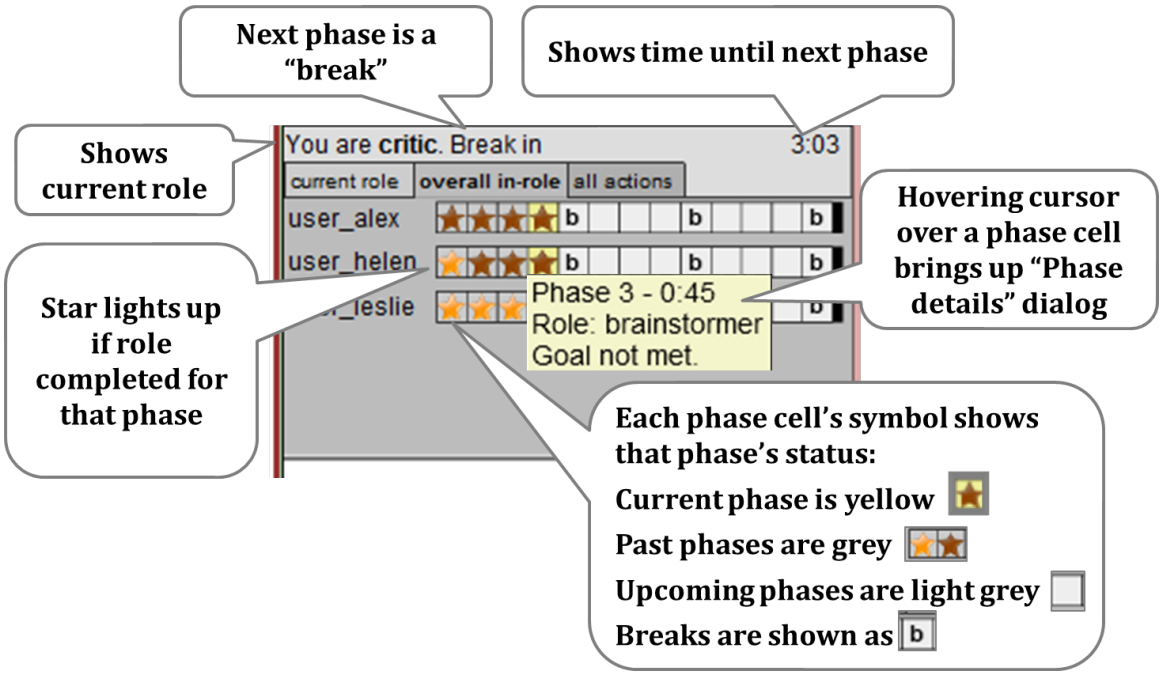


Figure 5.3: The "Overall In-Role" tab in the Roles UI shows all solvers' overall role performance for the entire user study solving session. Next to each username is a row of cells. (This screenshot is an example, and not from the actual user study.) Each of those cells represents one role phase for that user, except for rest breaks, which are shown as cells containing the letter "b." The current phase has yellow background, past phases have a dark grey background, phases that have not yet occurred have a light gray background. For past and current role phases, if the goal in-role action count was achieved by that user, the star is "lit up" in bright yellow, otherwise the star is grayed out. The mouse-over dialog box appears when the user mouses over any cell; this box contains information about which phase it is, what the associated role is, the total amount of time alloted for that phase (45 seconds here, because this session did not employ the user study role phase schedule), and the user's performance on the goal.

users' progress in the current role phase. Usernames are listed on the left, with a greyed out star under each name that lights up when the goal in-role action count is met for that user. On the right of each name is a percentage progress bar, showing that user's progress toward 100% completion of all in-role actions for that phase. This bar fills up as the user performs each in-role action; for example, if the Brainstormer creates one of the ten goal nodes for the role phase, 10% of the bar fills up, to give the user instant feedback and reinforcement for performing an in-role action. Mousing over the bar displays a dialog box that shows the exact number of completed in-role actions and goal in-role actions. This was done for the sake of the user study data collection; there was concern that we would not have enough information about how often the subjects actually looked at the Roles UI, so as a proxy, we decided to have a mouse-over dialog box, so that we could count the number of times solvers interacted with the Roles UI.

The "Overall In-Role" tab, shown in Figure 5.3, displays role progress over the entire user study solving session. Next to each user's name, a row of cells is displayed, representing all the role phases and breaks over the course of the entire solving session. As such, this tab would only be relevant for solving sessions that last for a known duration of time, such as a user study. In general, solving sessions presumably continue until the problem is solved, and the solvers would not be expected to know how long it would take a priori. (Or perhaps, during a solving session, the solvers could set aside a specified periods of time for using roles.)

Each cell in each user's row represent a single role phase, or a break, as shown in the figure. As each phase or break ends, its associated cell is grayed out. The current phase's or break's cell is highlighted by a yellow background. Stars in a cell indicated whether the phase goal was reached: lit up if the user met the phase in-role actions goal, dimmed if not.

Similar to the "Current Role" tab, mousing over a cell brings up a dialog box with more details about that cell's particular phase. This includes information about the phase number (the first phase of the session is "phase 1", the second phase is "phase 2" and so on), the time duration for that phase, what role the user is, was, or will be assigned for that phase, and whether or not the goal was met. If the phase has not occurred, the dialog box will state that the goal was not yet met. Otherwise, it will show text such as "Goal

met within 2:41; did 5 out of 4." Note that the user can exceed the goal number of in-role actions, this is so that the user can feel like they "get credit" for continuing to act in-role, even if they already exceeded the goal. The "Overall In-Role" tab's mouse-over dialog was implemented for the same reason as the "Current Role" tab's, we wished to use it to collect data on whether the users were paying attention to the UI.

Finally, thirty seconds before the role phase changes, the solving session user interface's background begins to pulse with a different color–red when the user's next role is Critic, blue when the next role is Brainstormer, and Green when the next role is Supporter. (Note that these colors are also mirrored in the "Current Role" tab progress bars.) This gives the user warning that their role is about to change, in addition to the information given by the countdown timer in the Roles UI showing time until the next phase. When that timer counts down to zero, it is time for a new role phase. The current role phase immediately ends and the new one automatically begins, without any user intervention. When this occurs, the solving session's background is switched back to white, and the Roles UI changes to display a role change notification, as shown in Figure 5.4. Although the new phase has already begun, subjects must click the notification to close it, otherwise they are unable to view the regular Roles UI information. This, again, was an intentional design choice that we made due to concerns that the user might not have noticed that the role has changed. In our design discussions, we initially considered blocking out the entire screen such that the user could not perform further actions without acknowledging the role change. However, we felt that this could possibly be frustrating if the user was in the middle of an action during the change. So we decided to have a smaller dialog box over the Roles UI instead. That way, the users need to click it to view additional information, and we as the experimenters have extra data about whether they chose to look at the Roles UI and whether they know there was a role phase change, but the users have the option to ignore it, or delay clicking it, if they are busy at the moment.

5.1.2 Study Procedure

Although we are describing the roles user study in a separate chapter, it was actually conducted as part of the CitySim User Study in Chapter 4. The user study procedure for

Figure 5.4: After the timer runs out on a user's role phase, the user's role automatically changes and the new role phase begins immediately. After the change, this dialog box appears in the Roles UI. The user is asked to click to acknowledge the role change. However, the new role phase, and the timer for the new role phase, has begun regardless of when the user clicks the box.

the Roles User Study is almost exactly the same as for the CitySim study, the CitySim problem template was used, and we recruited from the same pool of users. The CitySim User Study can really be thought of as one user study with three conditions: the Minimal Consultant, the Full Consultant, and the Roles UI. We are choosing to write about the results of this last condition, the Roles condition, as a separate chapter for clarity's sake.

In this study, we will compare the Roles condition with the Minimal Consultant condition (our control condition). For the Roles condition, we again had nine subjects, divided into three teams of three people each, the same as in the other conditions.

The only differences for this condition is as follows. First, instead of the tutorial material on the Minimal Consultant or the Full Consultant at the beginning of the user study session, subjects were instead given information on the Roles UI. During this tutorial, they were told about the three roles and what each roles' duties were, and showed what each of the

parts of the Roles UI meant. The rest of the introductory tutorial on CoSolve usage and CitySim rules was the same. Secondly, instead of being told that they would receive bonus points for clicking on the Full Consultant buttons, or on the Minimal Consultant's "Help Pages," subjects were told they would receive bonus points for meeting role phase goals. Then, during the actual activity, the Roles UI was enabled for each of the teams' solving sessions.

Additionally, during the wrap-up interview, they were asked additional questions regarding their usage of roles (listed in Appendix G). Also, in their wrap-up questionnaire, Consultant-related questions were replaced with Roles UI questions. This wrap-up questionnaire can be seen in Appendix F. Other than these differences, the Roles condition sessions were conducted in the same way as the other sessions.

5.1.3 Overall Results

We will continue the team identification and numbering system from Chapter 4; hence the three teams in the roles condition are Team 7, Team 8 and Team 9, and the three subjects in Team 7 are *p1_team7*, *p2_team7*, *p3_team7*, and so forth. We will compare them to the Minimal Consultant teams as a control condition; these teams will have the same identification numbers as in Chapter 4, i.e., Team 1, Team 3, Team 5. In this section, we will specifically identify which results were statistically significant; any other results presented can be assumed to be not statistically significant.

Roles performance

First, let's look at whether the subjects actually attempted to follow their assigned roles. Table 5.2 shows the number of phases for which each subject completed the goal actions for that phase, and also shows the phase ID (PID) of phases for which the subject did not meet the goal. (Breaks are included as "phases", but subjects took a rest break during that preiod.) Everyone met their phase goals for the first four role phases (phase 1, 2, 3 and 5), so it would seem that subjects started off fairly aware of their roles and the associated goals. Also, most of the subjects met the goals for most of the phases, with the exception

Table 5.2: This table shows the number of roles phases each subject completed, i.e., phases during which the subject met the goal in-role action count. The PIDs correspond to the PIDs in Table 5.1, PIDs 4 and 8 are rest breaks for the subjects.

	Number of phases in which goal was met	PID of phases in which goal was not met
p1_team7	8 / 9	9
p2_team7	7 / 9	6, 9
p3_team7	9 / 9	-
p1_team8	9 / 9	-
p2_team8	5 / 9	7, 9, 10, 11
p3_team8	9 / 9	-
p1_team9	9 / 9	-
p2_team9	7 / 9	6, 7
p3_team9	8 / 9	7

of *p2_team8*, who seems to have stopped paying attention to role goals in the second half of the activity. This is corroborated by our interview data with the subject. Additionally, in our interviews with other subjects, we generally found that at the beginning, subjects were aware of meeting their goals, but as time went on and they got more involved in the solving task, they paid less attention to them.

Table 5.3 shows the number of in-role and non-role solving actions they took over the course of the entire solving session. We consider node creation and annotation creation actions as *solving actions*. An in-role action occurs when a subject performs an action associated with their current role, e.g., creating a negative annotation when the subject is assigned to the Critic role. A non-role action occurs when a subject performs actions not associated with the current role, e.g., creating a positive annotation when the subject is assigned to the Critic or Brainstormer roles, or creating a neutral annotation at any time. Note that this includes in-role actions taken even after the role goal was completed. The data shows a large amount of variation in subjects' in-role action percentages. About three-

Table 5.3: In-role actions summary for roles condition teams. This table shows the number of in-role versus non-role actions performed by each person, for each of the subjects in the three roles condition teams, Team 7, 8 and 9.

	Team 7			Team 8			Team 9		
	p1	p2	p3	p1	p2	p3	p1	p2	p3
# of in-role actions	219	84	171	216	159	132	210	123	108
# of non-role actions	165	153	237	78	165	291	69	213	213
% of actions that are in-role	57.0%	35.4%	41.9%	73.5%	49.1%	31.2%	75.3%	36.6%	33.6%

quarters of the actions performed by subjects *p1_team8* and *p1_team9* were in-role, about half of *p1_team7*'s, *p3_team7*'s, and *p2_team8*'s actions were in-role, and about a third of the actions of the other four subject were in-role.

As Table 5.3 shows, on the whole, most subjects performed more non-role actions than in-role actions. Since we also saw that most of them met their role phase goals, this means they had enough time to complete additional work outside of their roles. So, next, let's look at time to goal completion per phase, in Table 5.4. Phases before the first break–phase 1, 2 and 3–are slightly different in that the goals were easier (as described in Table 5.1), so let's look only at phases after the first break. In this table, we see that subjects were generally faster to complete their goals in the last third of the activity, after the second break. This is the case even if we remove *p2_team8* from the average. One possibility for this is that subjects became more efficient at completing their roles as time went on.

Solving activity performance

Next, let's look at user performance in the Roles condition compared to the those of the Control condition (Minimal Consultant condition). Figures 5.5, 5.6, 5.7 show the solving session trees for the roles teams, teams 7, 8 and 9 respectively. The trees for the control teams were shown in Figures 4.17, 4.19 and 4.21. Team 7's tree looks fairly typical of the other trees we've seen, with a few long branches by a single users, and occasional branching turn-taking between users. Team 8's tree consisted of less turn-taking, and a lot of long

Table 5.4: Average time, in minutes, that users took to perform goal number of in-role actions per phase, for all phases where goal was met. (Phases during which a user did not meet the goal are not included here.)

Phase ID (PID)	Time to goal	Allotted time
1	5:59	15:00
2	2:00	5:00
3	2:06	5:00
4	break	5:00
5	7:16	8:20
6	6:40	8:20
7	6:37	8:20
8	break	5:00
9	4:47	8:20
10	5:17	8:20
11	6:35	8:20

straight branches by single users; again, this is not too unusual. Team 9's tree was slightly different, compared to previous trees we've seen in that there is initially not much turn-taking at the top, and mostly single branches, but later in the session, and near the bottom of the tree, there was much more turn-taking and branching.

Table 5.5 contains the data for actions performed by subjects in both conditions. (Teams 2, 4, and 6 are not listed, they were the Full Consultant condition described in Chapter 4.) With the exception of positive and negative annotations created, and overall annotations created, none of the condition differences we list below are statistically significant, but we will use this data as a reference to guide our qualitative analysis of user behavior.

The table is divided into several sections. The first set of rows in the table, the Nodes section, describe node creation behavior. Under the Nodes section, "Nodes created" is the number of nodes created by each user in the entire solving session. On average, control subjects created 99.3 nodes, roles subjects created 91.44. "Percentage of team total" under "Nodes" is the number of nodes created by the subject as a percentage of all the nodes created by their team. We also asked subjects, after the activity, how many nodes they

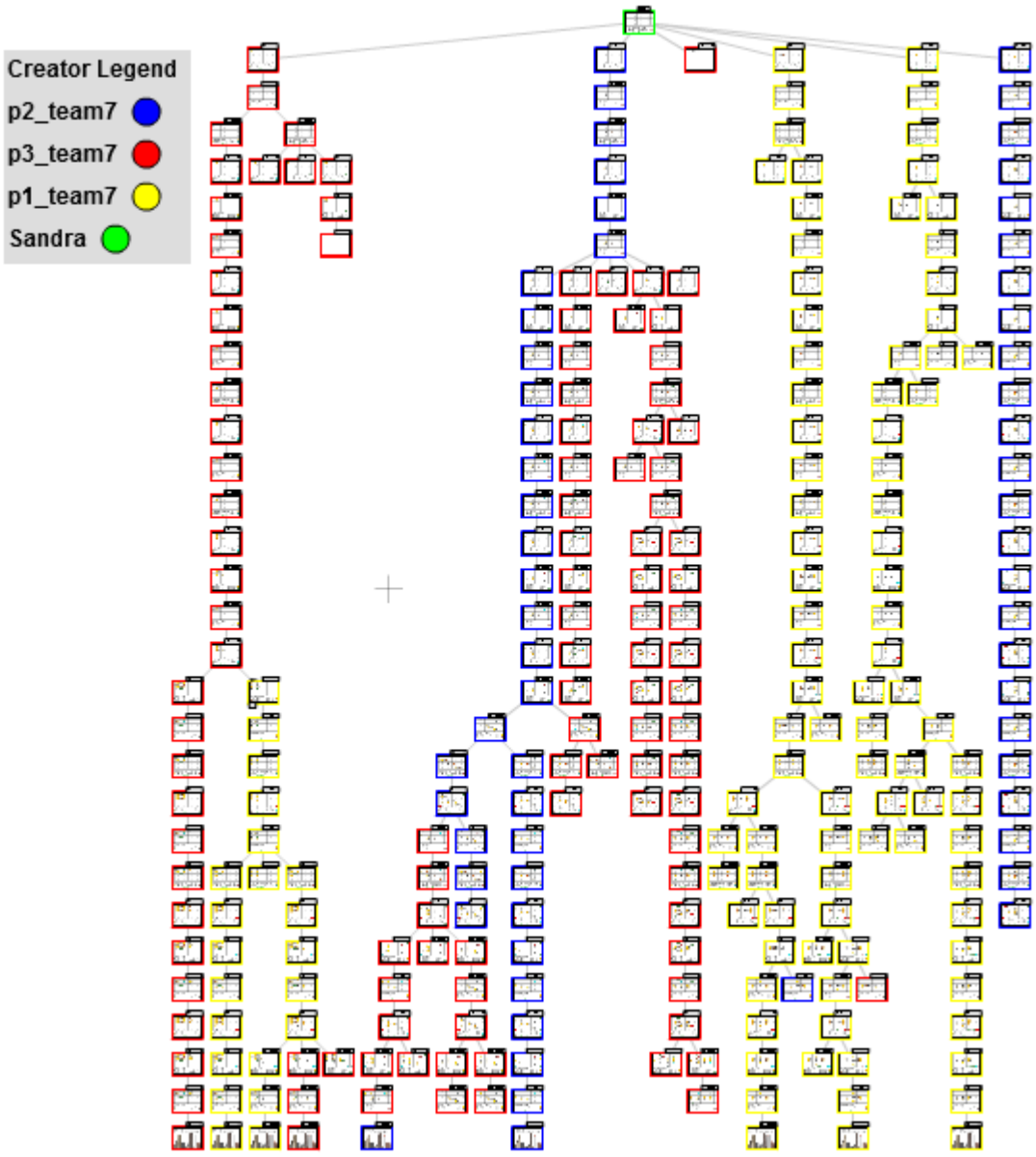


Figure 5.5: Team 7 - Solving-Session Tree.

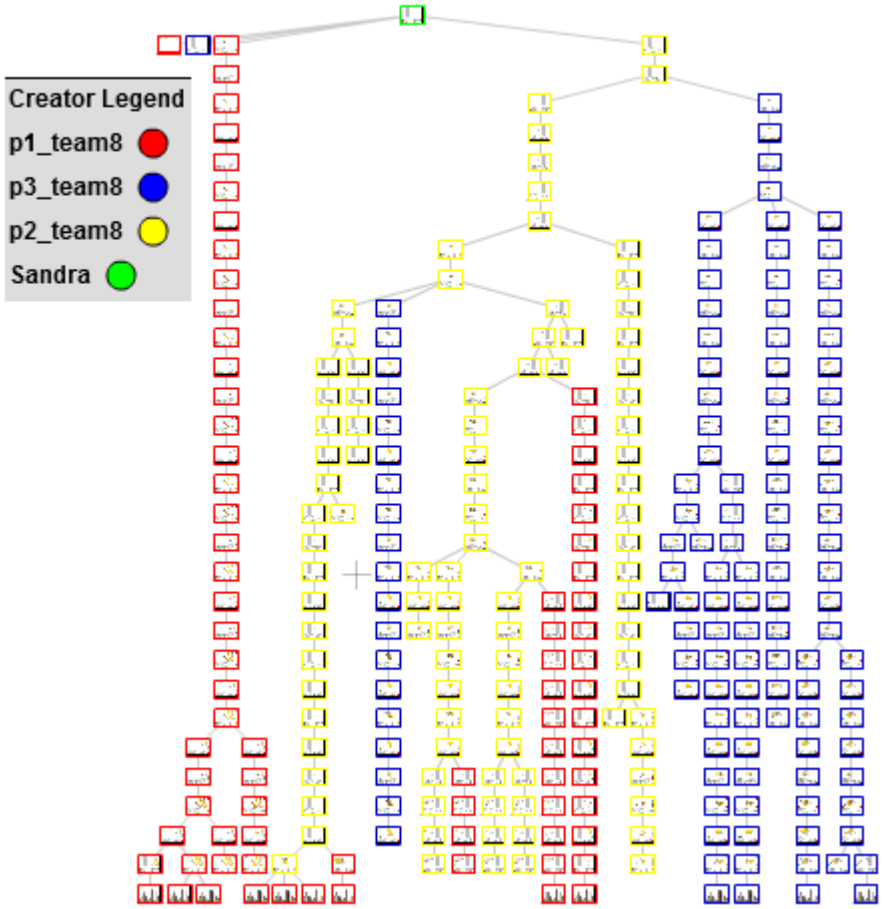


Figure 5.6: Team 8 - Solving-Session Tree.

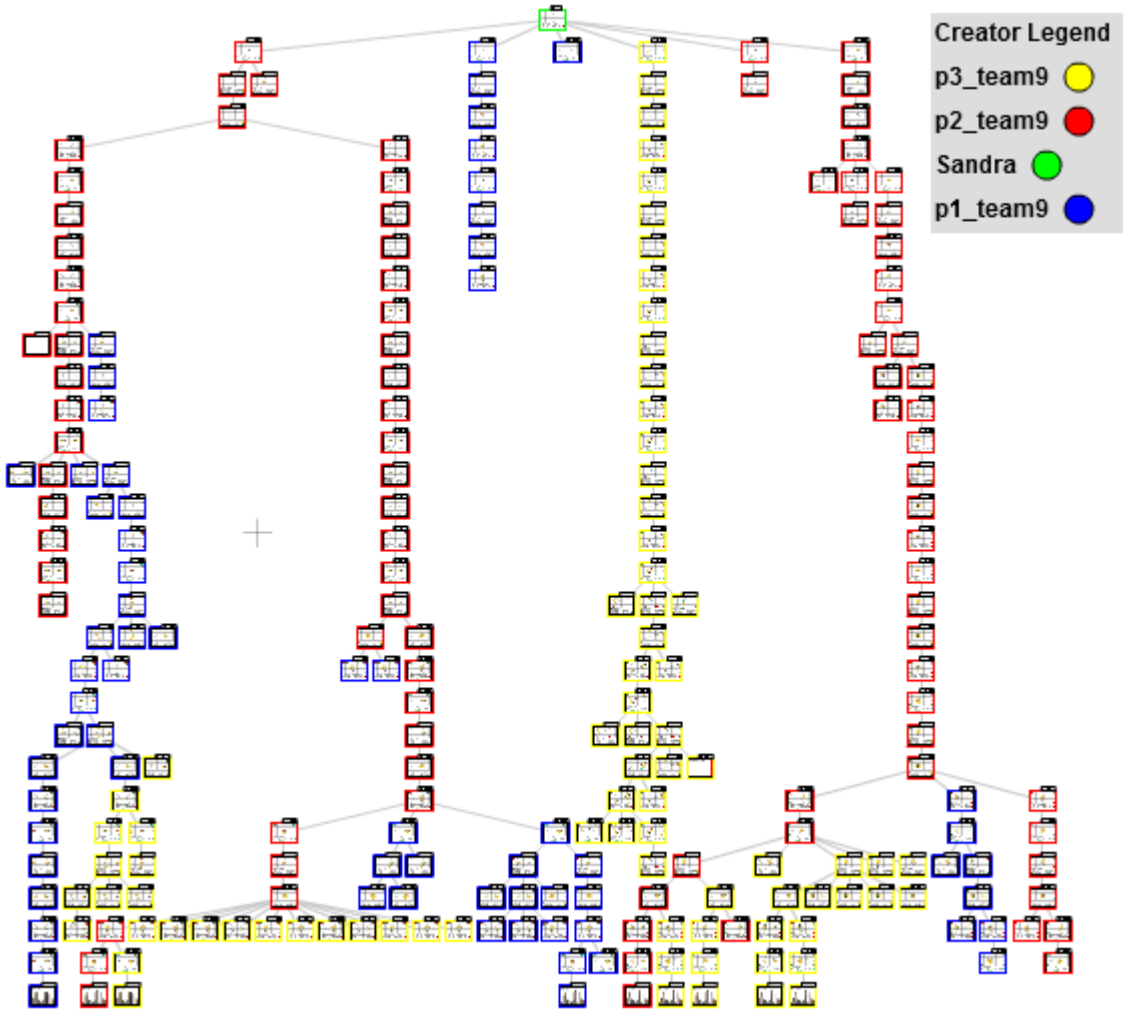


Figure 5.7: Team 9 - Solving-Session Tree.

thought they created, as a percentage of all the nodes their team created; these guesses are listed in the "Percentage guess" row.

The equivalent rows under the Annotations section of the table have the equivalent meaning. Below that, the Annotations by Type section is mostly self-explanatory; "neutral" annotations are annotations for which the subject explicitly selected the "neutral" annotation type. "Null" annotations are annotations for which the subject did not select any type. Some of the data in these Annotations-related sections are statistically significant, as will be described later.

The next section describes total solving actions. We define this value to be the sum of nodes created and annotations created, and represent the total amount of "work" each participant did. On average, Control subjects performed 105.9 actions each, and Roles subjects performed an average of 111.4. Finally, the CitySim scores are as described in Chapter 4. The team's score is the highest-scoring node for that team's session, and the user high score is the highest-scoring node created by that user. All of the Control condition scores are higher than the Roles condition scores.

The table contains a lot of per-user data that can be more easily understood when represented visually, so we have also included plots of the data[1]. These are the remaining figures in this section, starting from Figure 5.8. These plots are read as follows. The x-axis shows the teams; the control teams are T1, T3 and T5 for Team 1, Team 3 and Team 5 on the left, and the roles teams are T7, T8 and T9 (Team 7, etc.) on the right. The y-axis shows the metric we are using, for instance, number of nodes created by each user. Stacked above each team's label are three dots, each representing that metric's value for one of the three users in the team. The mean value for each team is denoted by the horizontal line above each team.

As an example of how to read these plots, let's take the first plot in Figure 5.8 shows the nodes created by each user, compartmentalized into teams. Visually, we see that the control teams seem to have a larger spread of nodes created; Team 1's subjects on average created fewer nodes than all other teams, Team 3 and Team 5 created more. The roles teams' node

[1]These plots were created by Julie Michelman, a graduate student in the UW Department of Statistics, as part of a statistical consulting assignment.

Table 5.5: Overall performance data for roles and control (MC) conditions. The "guess" row data are users' guesses at the percentage of node or annotations out of the team's total that each user thought he created after the activity finished. "Solving action" count is the sum of nodes created and annotations created.

	Control									Roles								
	Team 1			Team 3			Team 5			Team 7			Team 8			Team 9		
	p1	p2	p3	p1	p2	p3	p1	p2	p3	p1	p2	p3	p1	p2	p3	p1	p2	p3
NODES																		
Nodes created	28	73	76	126	149	123	86	84	149	111	61	112	79	97	122	66	94	81
Percentage of team total	0.16	0.41	0.43	0.32	0.37	0.31	0.27	0.26	0.47	0.39	0.21	0.39	0.27	0.33	0.41	0.27	0.39	0.34
Percentage guess	0.15	0.333	0.5	0.25	0.275	0.333	0.25	0.333	0.4	0.33	0.33	0.37	0.3	0.3	0.3	0.2	0.5	0.33
Nodes on Solution Path	0	6	24	1	2	27	0	0	30	0	6	23	0	2	28	0	25	5
Soln. Path Inequality (SPI)	16			17.333			20			15.333			18.667			16.667		
ANNOTATIONS																		
Annotations created	3	12	23	11	3	11	0	2	3	17	18	24	19	11	19	27	20	25
Percentage of team total	0.08	0.32	0.61	0.44	0.12	0.44	0.00	0.40	0.60	0.29	0.31	0.41	0.39	0.22	0.39	0.38	0.28	0.35
Percentage guess	0.1	0.5	0.75	0.5	0.225	0.333	0	0.5	0.5	0.33	0.33	0.4	0.3	0.2	0.33	0.33	0.225	0.33
ANNOS. BY TYPE																		
null annotations	1	7	11	0	0	0	0	0	0	1	0	0	0	3	0	3	0	1
neutral annotations	0	0	1	4	0	3	0	0	0	0	0	0	0	0	0	0	1	0
positive annotations	1	5	11	7	3	8	0	1	3	7	9	15	10	2	9	15	10	13
negative annotations	1	0	0	0	0	0	0	1	0	9	9	9	9	6	10	9	9	11
TOTAL ACTIONS																		
Total Solving Actions	31	85	99	137	152	134	86	86	152	128	79	136	98	108	141	93	114	106
CITYSIM SCORES																		
Team Score	33087			26576			26865			21945			21779			25363		
User High Score	8349	33087	28432	22024	26576	26550	13362	25080	26865	19807	13672	21945	20378	19782	21779	20864	24936	25363

counts are mostly in between these. Also we can see that for the Control teams, there tends to be two subjects per team who create roughly the same number of nodes (two vertically-stacked dots that are close together), and one teammate who creates significantly more, or fewer. For example, in Team 1, two subjects both created almost 80 nodes each, and one subject created less than 40. On the other hand, out of the roles teams, Team 7 exhibits this behavior, but Team 8 and Team 9 are more spread out in terms of node creation per subject.

From the plots in this figure, we can see that there is no node creation difference between the two conditions. Also, looking at the null and neutral annotations, there isn't a clear difference in the averages of the two condition, though we do seem team outliers: Team 1 has many more null annotations, and Team 3 has created a couple more neutral annotations than most of the other teams. With annotations in general, however, and with positive and negative annotations individually, we see a definite difference in the averages. All of the roles teams' averages are above all of the control teams' averages. These are all statistically significant differences (p=0.033). So we can conclude that, since most subjects complied with their role assignments, they did indeed create more annotations related to the Critic and Supporter roles, though they did not create more of any other types of annotations. Particularly dramatic is the difference in negative annotations; without an explicit Critic role, subjects tend not to create any negative annotations as shown in the previous user study. With an explicit Critic role, subjects tend to create only exactly as many as they need to fulfill the Critic role (9 negative annotations over the entire session).

In Figure 5.9, we see metrics related to equitable participation. Let's look at the first plot, which shows "Nodes on Solution Path." For all the teams, the majority of the nodes on the solution path were created by one teammate, so there is no difference between the two conditions. Indeed, the average SPI[2] for the control teams is 17.8 for Control, and 16.9 for Roles–statistically, there was no difference. As for subjects' guesses on how equally they all participated, the Roles subjects did tend to think that they worked more equally, both in terms of nodes and annotations, with the exception of Team 9. From Table 5.5, we see that

[2]Recall that SPI (Solution Path Inequality) ranges from 0 to 20. The higher the value, the more inequality between the team members' node counts on the solution path.

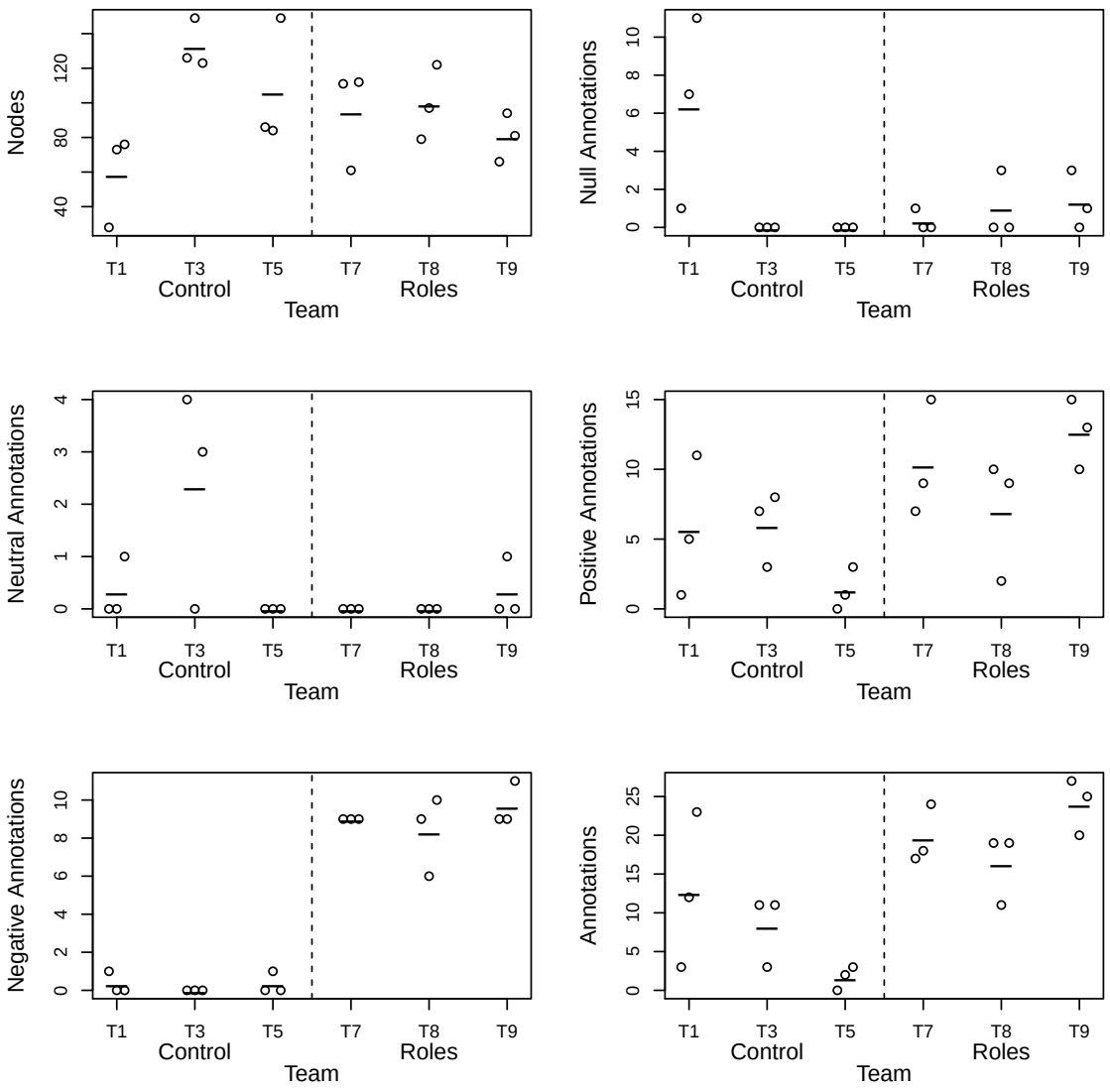


Figure 5.8: Participation metrics: solving actions by user, by team. These plots show nodes created, annotations created, and annotation types created.

p2_team9 overestimated his contribution, and *p1_team9* underestimated her contribution, but in actuality, as we can see from the Nodes plot in Figure 5.8, their node creation counts are actually closer to each other than other teams'.

In Table 5.10, we see subjects' guesses compared to their actual contributions. Here, the numbers on the plot represent each of the three subjects on a team. Numbers along the diagonal line represent subjects that guessed their percentages correctly. For example, the three number "1's" on the plot represent each of the three members of Team 1. In the nodes proportion graph on the left, we see that one member of Team 1, in the bottom left corner, guessed that they created very few nodes compared to the rest of their team, and that this matches reality. The "1" near the top of the plot shows another member of Team 1, who guessed that they created around 50% (i.e., 0.5) of their team's nodes, but they actually created around 40%. The final member of Team 1 is somewhere near the middle, also creating around 40%, but guessing that they created 33%. Comparing the nodes versus the annotations proportion plots, it seems that annotations were created more equally overall, especially for the Roles condition teams (7,8,9), and that subjects' guesses regarding annotations are more accurate than their node guessses.

Finally, let's look at Figure 5.11 to examine each user's highest-scoring node. In general, there appears to be more variation in each users' scores in the Control teams; the Roles teams' members had scores closer to each others'. However, gaming experience seems to be a stronger factor. Based on subjects' answers in the background questionnaire, we found a moderate positive correlation between familiarity with multi-player/city-simulation games and CitySim high score ($r=0.53$, $n=27$,$p=0.0045$). Additionally, the two particularly high-scorers in Team 1 are graduate students who research computer games, so it is not unreasonable to hypothesize that their experience contributed to their scores.

Collaboration attitude

Next, we will present the data regarding subjects' attitude changes toward collaborative activities, as reported by the subjects in their Background Questionnaires (Appendix B) and Wrap-Up Questionnaires (Appendix E). In each questionnaire, subjects responded to

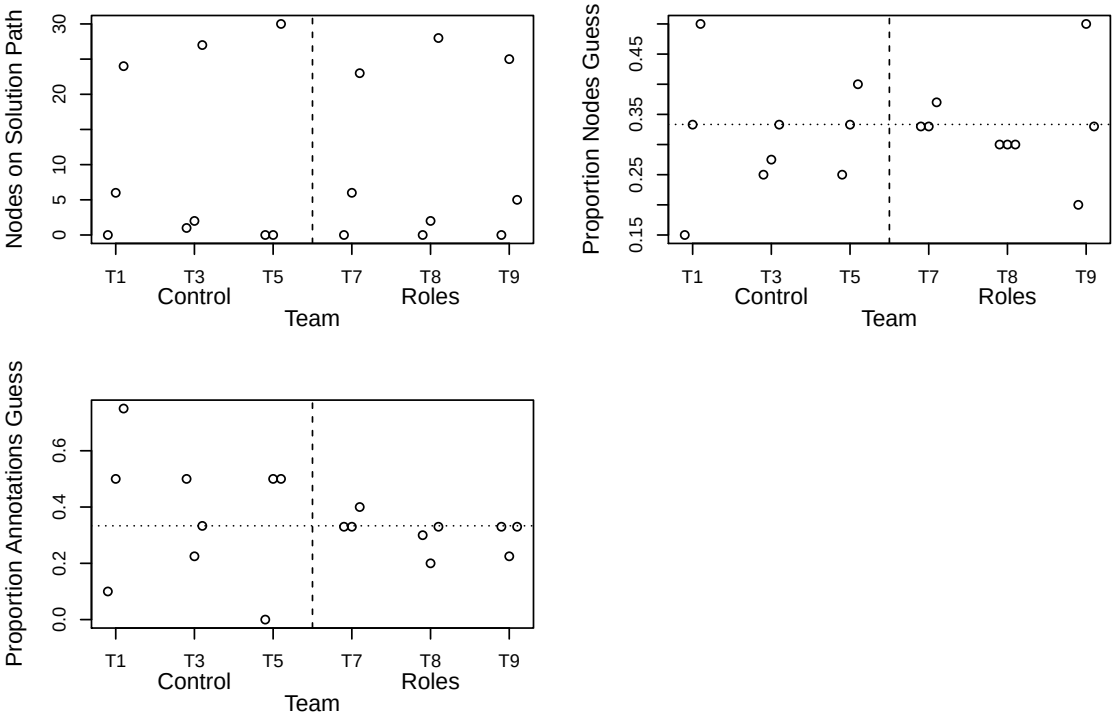


Figure 5.9: Equitable participation metrics. These plots show how many nodes each user created on the solution path, and the post-activity percentage (proportion) of nodes or annotations each user estimated that they created out of the total their team created.

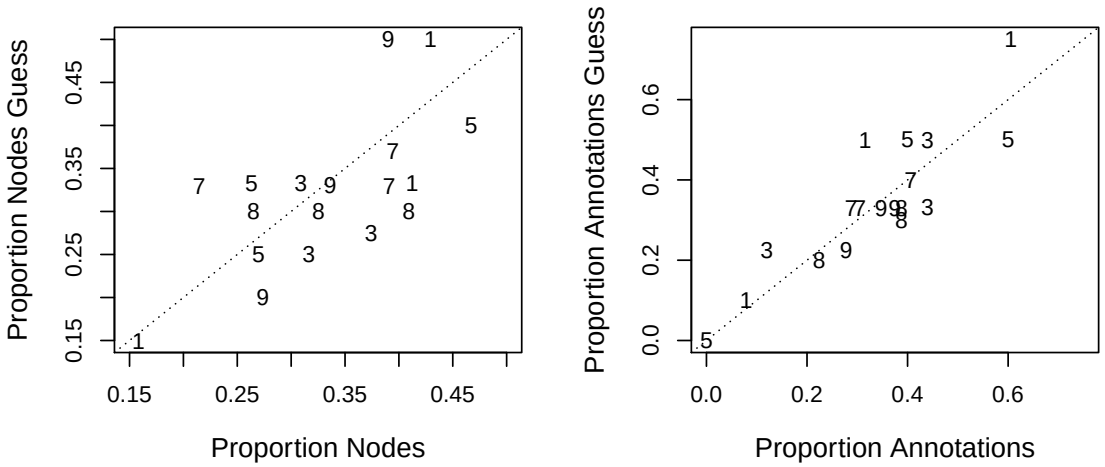


Figure 5.10: Subjects' guesses about their activity percentages (y-axis), versus their actual percentage contribution to their teams (x-axis). Each number in a plot represents a team member in that team, for instance, the three 1's in the plot represent each of the three members of Team 1, and their guesses as it relates to their actual activity percentages.

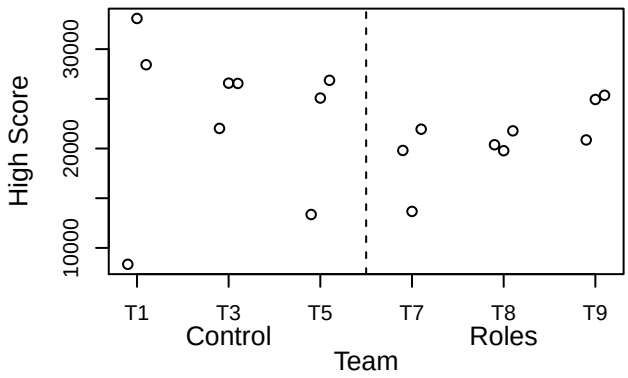


Figure 5.11: User high scores, by team. A user's high score in the highest-scoring CitySim node they created during the solving session.

each of the following questions on a scale from 1 (Strongly Disagree) to 5 (Strongly Agree):

- I enjoy working in a group or team.
- On most projects, I work better if I am in a group, rather than alone.
- Given a choice, I would prefer to work on projects in a group, rather than on my own.
- When working on a group project, I often feel free to share my opinions.
- In group project discussions, it is important to me that everyone feels they have a chance to participate in the discussion.
- When trying to complete a group project, it is more important to me that everyone has a chance to share their opinion, rather than others agreeing with my opinion.
- When working in a group, I often feel like I do more work than my group members.
- I am often hesitant or reluctant to participate or share my thoughts in *online* group discussions.
- In *online* group discussions, I express my opinion equally as often as other group members.
- In *online* group discussions, I often wish I could participate more.
- When discussing a project *online*, I am sometimes uncertain how to best participate.
- I believe working in a group results in a better solution than working alone.

Two subjects, *p3_team8* and *p1_team9*, marked "n/a" for the "online" questions, reporting that they had never participated in online group discussions, and so their answers were dropped. Figures 5.12 and 5.13 show before-activity and after-activity changes in the Likert scale values for these questions. As can be seen from these plots, there is not much difference between the two conditions for any of the questions. However, for "Share Opinion in Group" in Figure 5.12, we can see that the change for the roles condition is always positive or zero, and the change for control condition is always negative or zero. Although the effect is not statistically significant, it could suggest that being assigned roles that required them to share their opinions helped subjects feel less hesitant to participate in group discussions than they might otherwise have felt after normally participating in a group. Also, for "Uncertain how to participate" (Figure 5.13), many subjects in the control group were more uncertain after the activity, but all the subjects in the roles group were either the same or less uncertain after the activity. So perhaps having roles helped subjects feel like they knew where they could contribute.

Next, in the Wrap-Up Questionnaire, we directly asked subjects if their attitude changed, again on the same Likert scale. The results are shown in Figure 5.14, the questions are:

- I have a better understanding of my own problem-solving process now than before playing the game.
- My attitude towards working in a group has changed during the course of this activity.

Finally, Figure 5.15 presents data on subjects' overall attitude change as measured by a composite attitude score[3]. This score is calculated as follows (accounting appropriately for each question's directionality):

```
AttitudeScore = enjoyGroup + workBetter + preferGroup + shareOpinion
              + allParticipate + allOpinions + (6 - moreWork)
              + (6 - webReluctant) + webEqual + (6 - webWishMore)
              + (6 - webUncertain) + betterSolution
```

[3]This score was calculated by Julie Michelman.

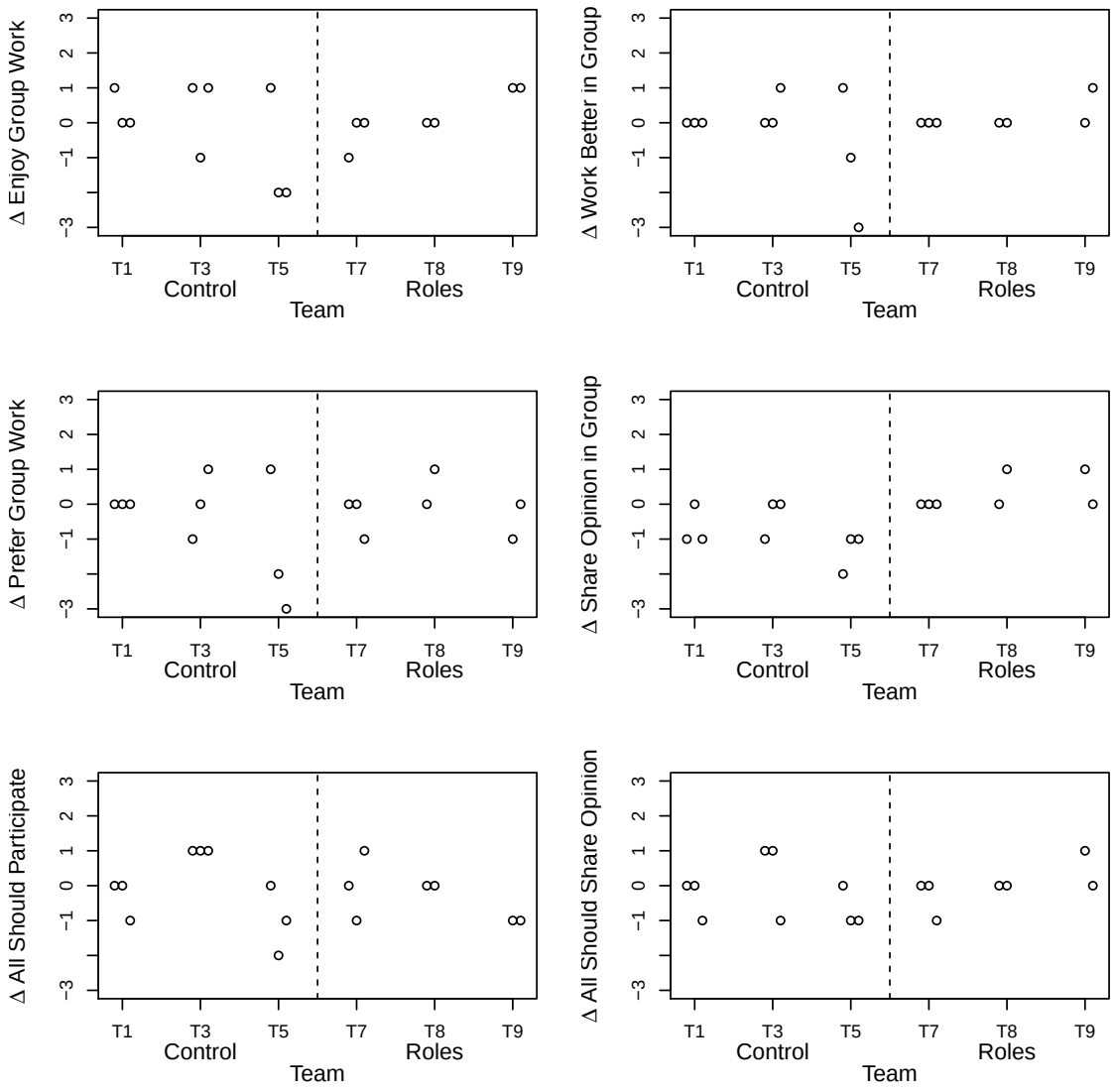


Figure 5.12: Change in attitude variables before and after activity. One subject from Team 8 and one from Team 9 were dropped, as they replied "n/a" to some of the questions.

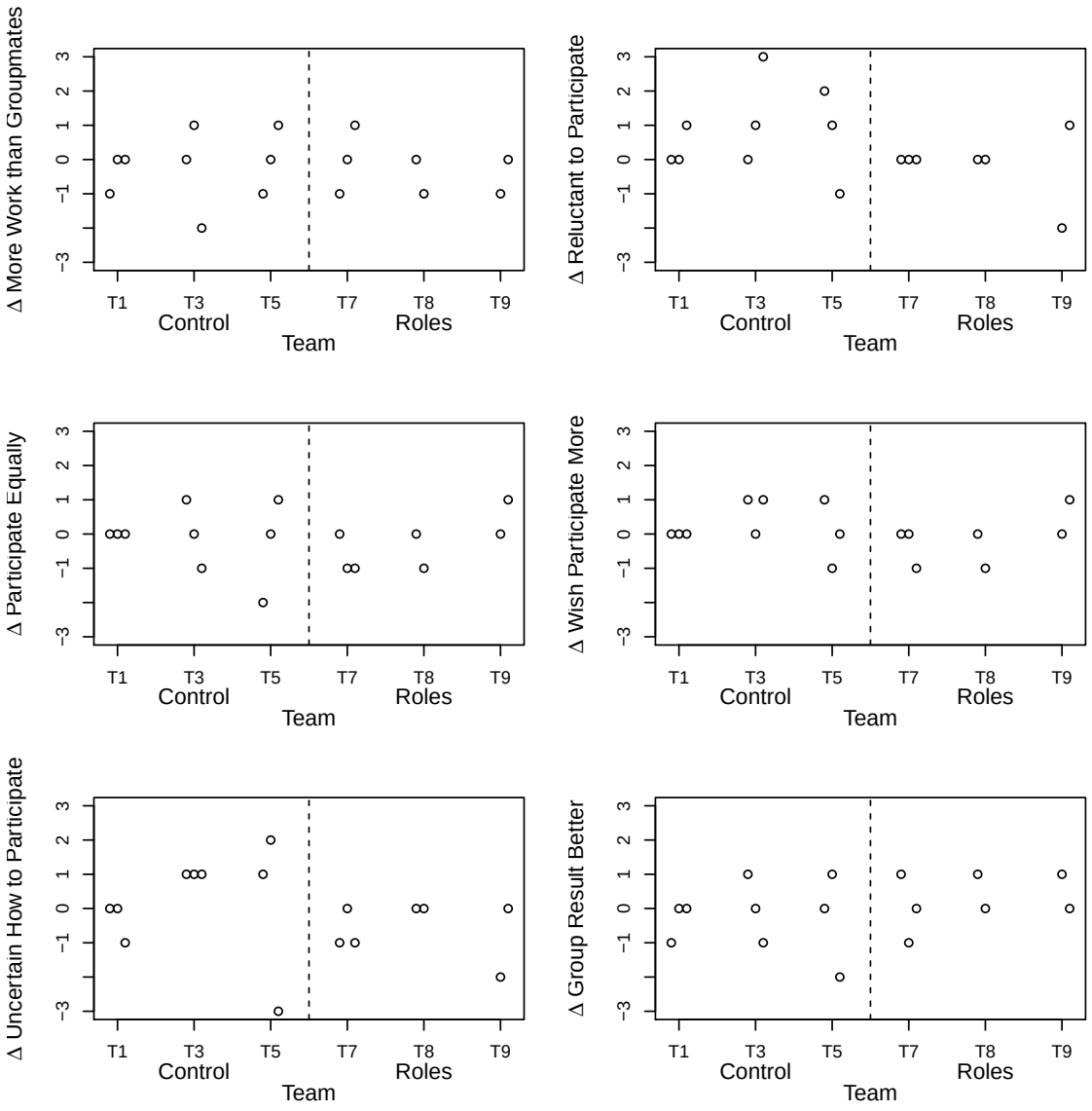


Figure 5.13: Change in attitude variables before and after activity, cont.

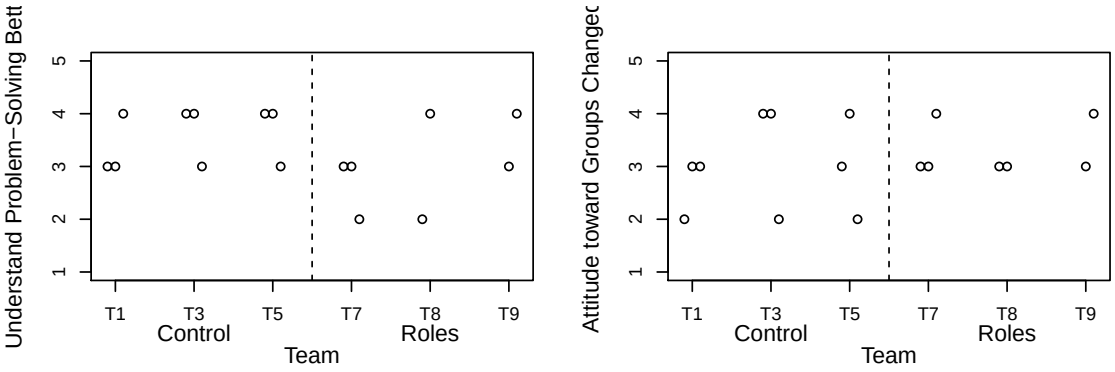


Figure 5.14: Self-reported change in attitude after activity.

Again, there is not a strong attitude change in either group here. However, more of the roles subjects are above the diagonal line than the control subjects, again suggesting that more roles subjects experienced some change in their attitude toward collaboration, as opposed to the control subjects.

User Interface

Finally, let's look at data related to the Roles UI. Table 5.6 shows user activity in the Roles UI area, specifically the number of times users clicked on tabs and hovered their cursors to view the extra dialog box information shown in Figures 5.2 and 5.3, and clicked on the role acknowledgment dialog, an example of which is shown in Figure 5.4. We notice that users rarely changed tabs, and generally did not often look at the mouse-over dialog information. So they did not often click or interact with the Roles UI area. However, we know that they at least looked at it, because they almost always clicked the role change acknowledgment (there were 12 such acknowledgments in total, because subjects had nine role phases and three breaks, and dialogs were presented to acknowledge the breaks as well, though the twelfth "break" was really the end of the entire activity). Recall that the acknowledgment dialog covers the entire UI, to see the Roles UI information, they had to uncover it by clicking the acknowledgment dialog.

From this, we can tell that solvers at least looked at the roles UI area once per phase.

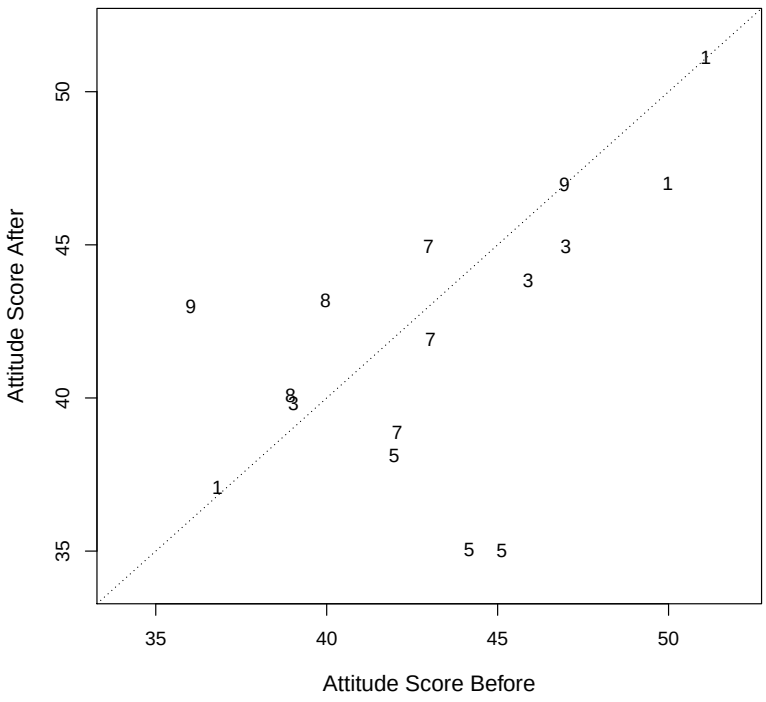


Figure 5.15: Composite attitude score before and after activity, for roles (Teams 7, 8, 9) and control (Team 1, 3, 5) team subjects.

We can also discern that the extra dialog information was either not helpful, or at least, not helpful enough to warrant taking attention away from the solving task.

Next, we asked subjects in their Wrap-Up Questionnaires to rate, again on the same Likert scale, the user interface. Roles subjects were asked to rate the roles interface, and control subjects were asked to rate the Minimal Consultant interface. These results are shown in Figure 5.16. Again, there is no appreciable difference between conditions for any of the metrics. In both conditions, most people disagreed that the interface was "hard to use." None of the subjects agreed with the statement that the interface was "hard to learn." So we can tell from this data that the Roles UI was not overly complicated for our subjects. Also, only one subject (*p1_team8*) strongly disagreed that the Roles UI helped

Table 5.6: This table shows the number of times each user viewed the Current Role bar dialog box, the Overall Role bar dialog box, clicked on each of the tabs, and clicked to acknowledge role changes.

	Team 7			Team 8			Team 9		
User UI Behaviors	p1	p2	p3	p1	p2	p3	p1	p2	p3
mouseover-current-role-bar	2	11	4	9	8	1	2	3	2
mouseover-overall-role-bar	0	0	0	0	0	0	1	0	2
click-acknowledge-role-change	11	12	12	9	11	9	11	11	12
click-overall-tab	1	6	1	0	1	0	2	0	2
click-current-tab	1	3	0	1	0	0	1	0	2

users collaborate; the rest were either neutral or agreed. While we can't tell from such a small sample whether *p1_team8* is unusual or not, at least for the majority of our subjects, the Roles UI was helpful for collaboration.

5.1.4 Discussion

Our first research question was regarding how roles can encourage more participation and more equitable participation. In our study, we consider participation to be the creation of nodes and annotations. Although we found that the subjects in the roles condition performed slightly more solving actions, this difference was not statistically significant. We did find a stastitically significant difference in the number of annotations created. When assigned to roles requiring them to create annotations, and incentivized to do so, subjects did indeed create more annotations, though they tended to do the minimum required, at least for negative annotations. So in that sense, there was more "participation" than in the control group. As for whether or not subjects would continue to perform in-role actions beyond their required goals, this differed from subject to subject, described in the previous section.

On interviewing the subjects, though, several told us that many of their annotations

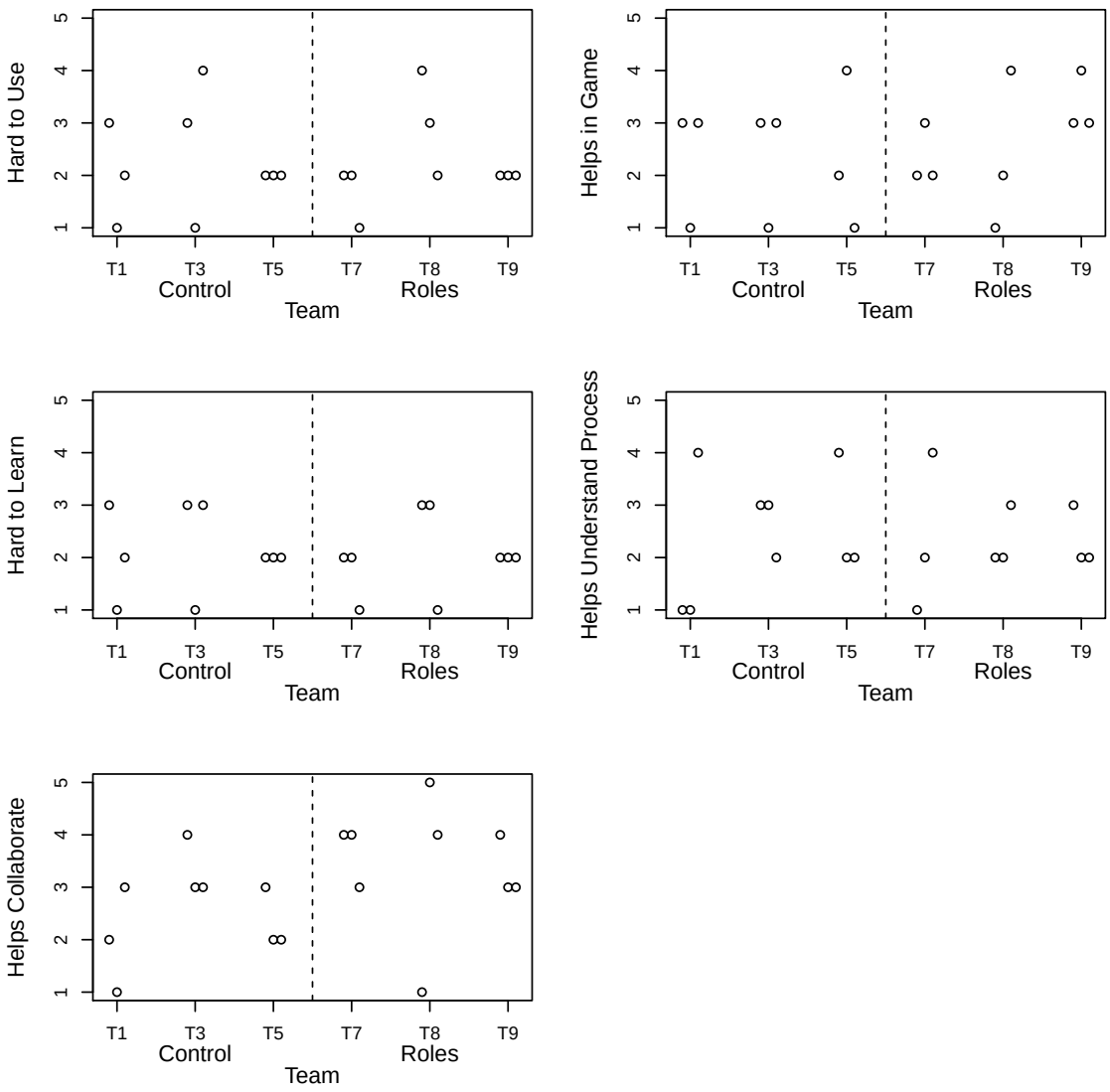


Figure 5.16: Roles subjects rated the Roles UI, control subjects rated the Minimal Consultant. Answers are on a Likert scale; 1 is Strongly Disagree, 5 is Strongly Agree.

were not genuine; they were created mostly to fulfill the role. Some are obviously so, for example, while in the Supporter role, one subject's positive annotation was simply the word "tired" (*p2_team9*). Others are more difficult to tell, a lot of annotations consist of simply "good" or "cool" or "not the best" without any more details; these seem likely candidates for "fake" annotations. So although there were more annotation actions, these were not necessarily meaningful actions. Hence, it would seem that, technically there was increased participation, but this may not have been meaningful participation.

As for the question of whether there was more equitable participation, we again saw that there was no significant difference between the two conditions. Roles subjects did report that they felt they participated more equally more often than control subjects reported this, in their work percentage "guesses". Additionally, while there was no difference in roles, there was more inequality in annotations in the control subjects than the roles subjects. However, as we noted before, subjects may not have been annotating genuinely.

Another factor to look at equitable participation was whether subjects contributed equally to the solution path. While we found no statistical difference, we did find an interesting effect. In all of the Roles teams, the final solution was made up of nodes created by the *p2* and *p3* subjects of each team, and never the *p1* subject. This was not always the case for the control (Minimal Consultant) teams, nor for the Full Consultant teams discussed in Chapter 4.

We conjecture that this actually had to do with the physical space in which we conducted the study. Due to building renovations, we had to use several different computer laboratories to run the study. It happened that the Roles conditions all took place in the same lab, and none of the other conditions' sessions occurred in this lab. In the Roles condition's lab, the three computers we were assigned for our study were placed in a row along a long table. Subject identifiers were assigned randomly, but then we always seated them *p1*, *p2* and *p3* from left to right at the tables. The seats for *p2* and *p3* were actually just a little bit closer together than *p1* and *p2*'s seats were, by perhaps about one foot. The difference was small enough that we as researchers did not notice this initially. However, upon interviewing subjects, and upon sitting in the seats ourselves, we noticed it was easier for *p2* and *p3* to see each others' screens than to see *p1*'s screen. Although we told subjects

to act as though they were doing the activity remotely and could not communicate with each other physically, they potentially could have seen what their neighbors were doing on their computers. In the other labs, we had been allowed use of the entire laboratory, so we spread out our subjects such that they were never sitting directly next to each other. This could potentially have been the reason that these two subjects in each team ended up working more closely together on what ended up being the final solution path for their team, and hence been a confounding variable in this study.

Our second research question was whether roles would lead to a better overall solution; this does not seem to generally be the case. We did see that all the roles teams' final CitySim scores were lower than all the control teams' final CitySim scores, so if we had a larger sample size, we may have been able to find that roles consistently did worse. Since, on average, control teams created more nodes, and roles teams created more annotations, it would seem that creating nodes, or "brainstorming" was more important to solving this problem than evaluation and interaction between the team members. However, if solvers are coordinating and tagging their nodes via annotations, wouldn't that prevent duplication of nodes and ideas, so that subjects could work more efficiently and have more "ideas" (i.e., nodes) overall?

There are several potential reasons why the roles teams had lower scores. One reason could be due to the creation of "fake" annotations. Since roles subjects reported having done "fake" annotations just to fulfill their roles, the time spent doing the annotations was wasted time that could have been spent more meaningfully brainstorming, i.e., creating potential solution nodes. Many subjects said in their interviews that when their roles changed to supporter or critic, they would try to create as many annotations as possible to "get it out of the way" and then go back to whatever they were actually working on. Subject *p2_team9*, actually said that whenever his role switched to something other than Brainstormer, it would make him "unhappy" because he had to go do something he didn't want to do. Most did say they first attempted to make meaningful comments, but when they could not find any places to do meaningful comments, they would simply do what they needed to get the role goal finished. So, many of the annotations were simply distractions, both for the Supporter or Critic who needed to create them unnecessarily, taking time away

from what they were currently working on, and for their teammates, who had to wade through irrelevant annotations. One way to solve this might be to not have switched roles in timed phases, but to simply have an indicator of the three types of roles. Subjects could see their performance in each of the roles, but could work on whichever they wished at whichever time they liked. Perhaps this would prevent breaking subjects' train of thought, but still visually make them aware of the fact that there were other things they could be working on, if they reached a lull in their current work.

A related reason is the time pressure. Since the subjects felt quite a time crunch, not only were they not necessarily creating genuine annotations, they may not have had time to read and react to others' annotations thoughtfully, and to evaluate parts of the tree that were not their own. Had the activity been conducted over, say, the course of several days, and the subjects allowed to log in and work on the activity at their leisure, perhaps they would have taken more time to look at others' nodes and annotations and learn from them. As such, because of the limited time available, a couple of the subjects–though not all–felt like they individually needed to try and get the highest score they could. One roles subject, *p2_team8*, said he didn't bother to pay attention to the roles after a while because he saw that his team's high score was higher than any he was able to achieve himself, and this "scared [him] a little." Instead of thinking, "great, my team is winning" or "let me work on the high scoring node, or find out why my teammates are doing better" he said that decided this meant he wasn't doing well enough and needed to work harder on his own to beat his teammates score, doing what he called "self individualistic optimization." In the interview, the subject says, "it was up to me to beat that score. And it did come into my head that I should have gone and inspected it, but I felt that the time limit really forced me to keep building this thing [the branch] because I invested so much into it."

For other subjects, simply seeing the roles phase timer generated much stress. The control condition, which saw the Minimal Consultant, had no timers on their interfaces. To indicate to roles subjects how much time they had left in a role phase, we displayed a timer counting down the phase's remaining time. Some subjects specifically told us that they saw this, and felt a lot of pressure. We perhaps should have had some sort of equivalent timer in the Minimal Consultant to control for this.

Another reason could be due to the nature of the CitySim problem and applying the operators. Perhaps CitySim had so few operators, and they were so easy to apply, that it was easier to create many nodes and then hope one of them will be high-scoring, than to write annotations about which nodes were most promising. With a problem template that had a much larger state-space or more complicated operator parameters, perhaps then, annotations would be more useful, as it would be more expensive to decide which operators to apply.

Finally, it could simply be that working alone and creating as many ideas as possible is always more important than trying to coordinate with others, and that in general, the overhead of communicating with others is not worth the cost of that time being spent working out new solutions. In this case, perhaps instead of annotating manually, CoSolve itself could just evaluate and alert solvers to the highest scoring nodes, without the users having to take explicit actions.

In general though, most subjects did not feel that collaboration was a bad thing, which leads us to our third research question, regarding how roles assignment affects users' attitudes and feelings toward collaboration, and their teamwork on the task in general. Most of the attitude measurements showed no change before and after the activity. For two of the items, "express opinion equally as often as other group members" and "uncertain how to participate," all the subjects either felt no change, or felt that they expressed their opinions equally and became less uncertain about how to participate. So with a larger sample size, we could potentially see a statistically significant difference in these participation attitudes. There was nothing to indicate that subjects felt worse about collaboration after the activity.

In the interviews, the majority of the roles subjects told us that without roles, there would definitely have been less collaboration. Subjects said that they would be more isolated and focused on themselves instead of others, they would not know how to leverage what others were working on. Even *p2_team8*, who completed the fewest role phases because he was most concerned with "self individualistic optimization" said that without roles, "I think it'd be madness. I think we'd collaborate even less. Yeah, I definitely felt roles initially forced me to take those actions; without them there would be fewer of those interactions. Seemed like interaction is important for a winning strategy."

A few of the subjects felt differently about how roles affected their collaboration, however. One subject, *p2_team9*, was particularly vocal about his dislike for roles. He remarked throughout the interview that they were "uncomfortable" and that it was "frustrating" being forced into roles. "Honestly, roles got in the way....I wanted to do my own stuff....So yeah, [roles] did affect my behavior, but perhaps in a negative way because I felt really forced into it, I was like, damn, now I have to, like, put thumbs-up." He also one of the few subjects to say that he "procrastinated" on roles. Instead of trying to get the role out of the way by finishing it earlier like most subjects, he waited until the role phase was ending to attempt to finish the role.

Other subjects simply felt that roles were unnecessary or distracting. Subject *p3_team8* said that without roles, there would be more brainstorming, but subjects would still communicate when they find something that works. This subject also felt that there would be less negative annotations, because "if it's not working, people know, there's no need to talk about it." So this subject seemed to think that collaboration would still have occurred without roles, there would just be less unneccessary forced collaboration. Subject *p1_team9* also said that annotations would have been *more* helpful without roles, because there'd be fewer fake, "filler" annotations.

Furthermore, many roles subjects mentioned in their interviews that they wanted some time to coordinate their activity before the session, preferably in-person. Some subjects mentioned that they wished they could have used a headset to talk to their teammates. And, as many subject mentioned even in the Minimal (control) and Full Consultant conditions, a conventional chatbox was the most desired additional feature. So it seems that the teams wished to collaborate more, but the annotation functionality was inadequate for their needs.

There could also be physical effects from the setup of the room in which the study took place. One of our Roles subjects also brought up that they felt the layout of the room made collaboration awkward–subjects were seated directly side-by-side next to each other, yet asked to refrain from speaking out loud, so as to simulate an environment where the users were in different locations. This subject said that collaboration was very unnatural knowing that the person you wanted to talk to was right beside you, but you had communicate with them only by annotation. Perhaps if subjects were literally in different locations, subjects

would have experienced less frustration, because it would seem more natural to communicate via the interface.

There *were* instances of communications that solvers felt positively about. For example, *p1_team9* explained, "When I posted about population lagging behind...p2 commented 'thanks' and so that was encouraging." That p2 subject is the same subject who told us annotations made him "unhappy" and roles were frustrating, but he completely independently in his interview, also remembered and brought up this incident, saying that *p1_team9*'s remark was helpful. Subjects also brought up instances where their teammates told them to "take a look" at various nodes, and this was helpful. Many subjects said they paid attention to positive annotations to find high-scoring nodes to work on, and usually felt negative comments were less helpful–they simply marked nodes that should be ignored.

On the other hand, one subject, *p3_team9*, remarked that it was helpful to receive critiques from others to know where she might be going wrong. Indeed, both her teammates had said that they saw her working on an incorrect strategy. As a result, one teammate, *p2_team9*, who had a good strategy, simply ignored her. The other teammate, *p1_team0* noticed *p2_team9*'s strategy, switched over to it, and during the Critic role, would negatively annotate *p3_team8*'s nodes "hoping they would come and work on p2's [branches]." This apparently worked, as not only did *p3_team9* eventually stopped working on her own and started working on others' nodes, she actually also created the final high-scoring nodes off of one of *p2_team9*'s branches. This team, Team 9, was also a little unusual, in that they started out with less turn-taking and branching than the other teams, but they ended up with much more of both behaviors in the end, as can be seen in their solving session tree, Figure 5.7, and that this is the behavior (the solution path) that lead up to their high-scoring node. In many of the other teams in the entire CitySim study, often the final high scoring node was a product of some turn-taking at the beginning, and a single-user long branch at the end. It seems then that for this team at least, collaboration helped their problem-solving process.

Overall, we did find instances where roles were helpful for subjects, and many of the subjects felt that roles increased their collaboration. Some of the subjects however, were frustrated by the forced role assignments, and nearly all the subjects told us that they

felt "time crunch" that caused creating annotations to feel like a distraction. It seems that a redesign of a roles system that is more flexible, and less time-focused, could have helped; this will be discussed in the future work section in Chapter 7. Our analysis found a statistically significant increase in number of annotations by the roles subjects, which is at least an affirmative sanity check of whether the subjects actually followed the roles, but we were unable to discern any statistically significant differences in increased participation and equitable participation in the two conditions due to small sample size, and possibly also due to confounding factors such as the user study's physical location. Additionally, many of the annotations made by roles' subjects may not have been genuine. We also saw that the roles' teams' solutions tended to be lower-scoring than the control teams'; it's unclear if this would have continued to be the case if a different problem template was used or if the subjects were given more time. However, from examining subjects' solving behavior, and from their interview data, we were able to discern the different feelings and attitudes that being assigned roles engendered in our subjects. Additionally, most subjects reported to us that they did indeed feel they collaborated more, and kept track of others' work more, due to being assigned to roles.

Conclusion

In this chapter, we have discussed some of the issues and imbalances with participation in collaborative problem-solving environments such as CoSolve. We attempted to address this issue by designing a system of participatory collaboration roles for our solvers. Overall, subjects reported that they felt roles helped them collaborate more, but we did not see statistically significant increases in collaborative measures, other than increased annotations. However, subjects felt frustrated by the constraints of the strict roles system, and the roles teams scored worse than the control team on the CitySim task. Perhaps a more flexible system for increasing participation would have been a more efficient way to address the issue. We will discuss ideas for future work in Chapter 7.

We have now seen examples of the many ways CoSolve can be used for problem-posing and problem-solving. In the next chapter, we will address how the CoSolve system came to be, by discussing the design and inner workings of the CoSolve's implementation.

Chapter 6

DESIGN AND IMPLEMENTATION OF COSOLVE

In this chapter, we will cover the technical challenges of CoSolve's implementation and design. There are several interesting issues in the design of CoSolve. The first is in the design of the problem-posing system. How can we design a posing system that would be both flexible and powerful enough to pose a wide range of problems, while also being easy for posers to use, and secure enough to allow our posers to execute arbitrary posing code? Another issue concerns how to perform the asynchronous execution of problem template code during an active solving session. The problem template's code is created by posers before any solving sessions are created, but during a solving session, each operator application, or operator preconditions, needs to be computed separately as needed. How should we store and execute these procedures? These and other issues surrounding CoSolve's development will be discussed in this chapter.

First, we will give an overview of the CoSolve system architecture. We will highlight some implementation details, and discuss CoSolve's API (Application Programming Interface) which allows other applications to access CoSolve solving functionality, and is currently being used to implement different kinds of CoSolve solving user interfaces. Finally, we will then address the design rationale and the trade-offs that we faced in creating CoSolve.

6.1 Overview of CoSolve's System Architecture

CoSolve is a website built with Drupal, an open-source content management system (CMS) installed on a LAMP (Linux, Apache, MySQL, PHP) stack. We chose to use Drupal as our web framework rather than build our own website from scratch so that we could quickly prototype CoSolve as a collaborative user environment–out of the box, Drupal provides functionality for user accounts and authentication, user permission access roles, storing and generating webpage content on the fly, etc. Additionally, Drupal's core functionality

can be extended by installing additional Drupal "modules." The Drupal community also has a large collection of "contributed" modules; that is, modules that other Drupal users have written, and released to the public, to provide extra functionality to Drupal, and we make use of many of these modules in CoSolve. We investigated using other CMSes and web frameworks, but at the time Drupal had a comparatively larger library of contributed modules than other systems. For this reason, we felt Drupal would be a good choice for prototyping CoSolve.

CoSolve's system architecture can be seen in Figure 6.1. At the base are the default components of a Drupal website: the Drupal "core" modules, written in PHP, and a MySQL database that stores all the content of a Drupal site. From there, we installed several Drupal "contributed" modules. The figure shows two of the most important ones, the *Services* module and the *Content Construction Kit* module, both of which we will talk more about momentarily. Some of the other Drupal contributed modules we use, not shown in Figure 6.1, include the Views module (used for generating different views of our content data), AMFPHP module (used by the web services for communicating between Flash and PHP), and the Organic Groups module (used to create "user groups" for access permissions).

The *Content Creation Kit* (CCK) module is used extensively to create the database schema for much of CoSolve. CCK allows you quickly and interactively create rich new "content types" in Drupal, beyond the basic content type functionality of Drupal. For CoSolve, we created the "Problem Template," "Operator," "Solving Session" and "Annotation" content types in CCK. The fields contained in each of these types were described in the chapter on problem posing (Ch. 3) and on problem solving (Ch. 4). Based on fields we specified, CCK automatically generates the basic web forms for posers to create or edit Problem Templates and Operators on the CoSolve website, and for solvers to create new Solving Sessions, or edit Solving Session settings through the web interface. CCK takes the user input from these web forms and stores and maintains this content in the Drupal database.

The actual creation of solving session trees and the execution of posers' Problem Template and Operator code are done by the *TStar-C* module, a custom Drupal module that

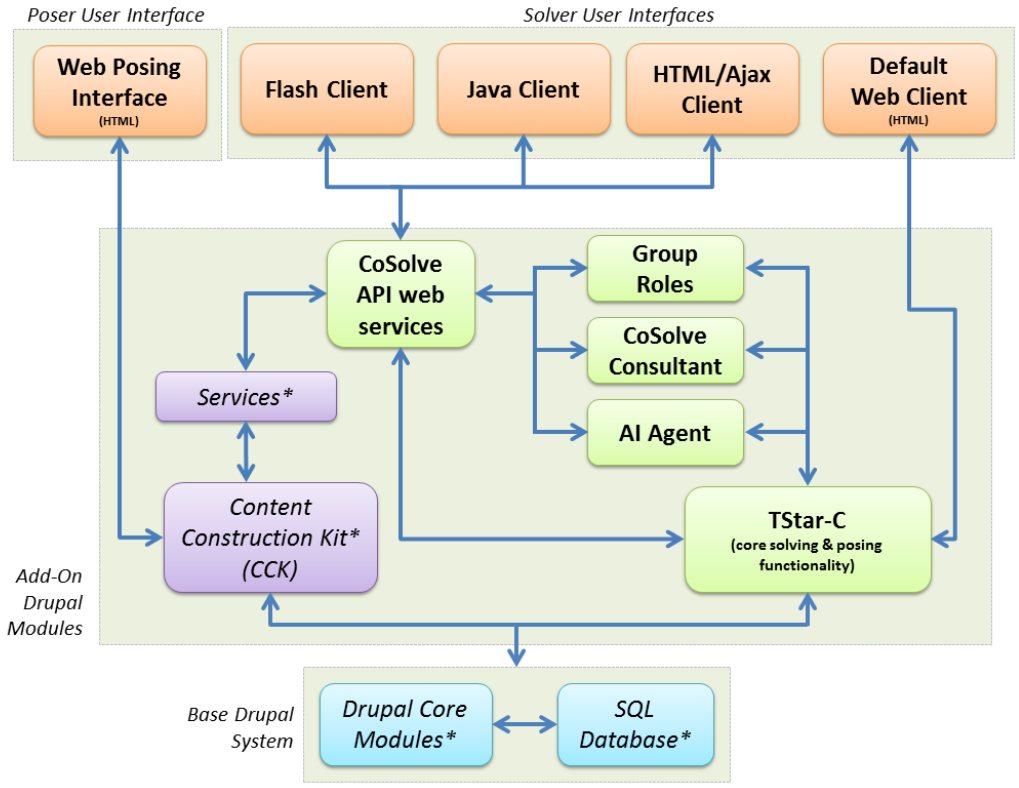


Figure 6.1: CoSolve System Architecture. Asterisks indicate components not created by our research group. Blue boxes are components that are part of core Drupal. The purple boxes indicate Drupal add-on modules contributed by the Drupal community that we make use of in CoSolve. Green components are custom Drupal modules that we created for CoSolve, these modules comprise CoSolve's main functionality. Orange components represent CoSolve user interfaces.

we built[1]. The TStar-C module then directly displays these solving session trees in HTML format on a web page through the default web client. However, CoSolve extends the aforementioned Services module to provide a CoSolve API so that others may create their own clients to access CoSolve solving session trees created by TStar-C. This API will be described in Section 6.3. Examples of client user interfaces include the Flash client described

[1]The module is actually named simply *TStar*, but we will call it *TStar-C* (TStar-CoSolve) here to avoid confusion with the earlier TStar system created by Steven L. Tanimoto.

in Chapter 4, a Java client[2], and an HTML/Ajax client[3]. The Flash client is actually now the most commonly used client, and the default web client is provided as a fallback when Flash is not available or enabled.

Additional custom modules include the AI Agent module[4] and the CoSolve Consultant module[5], both described in Chapter 4, and the collaborative Group Roles module, described in Chapter 5. All of these modules directly communicate with the TStar-C module, and can be accessed by the solving user interfaces via the CoSolve API web services.

Now that we have given a brief overview of the general components involved in the CoSolve system, in the next section, we will explain how the TStar-C module turns posers' problem template code into interactive solving session trees.

6.2 TStar-C Implementation Details

Let's now look at the details of how the TStar-C module composes together the problem poser's code into solving functionality for the problem solvers. After we discuss the implementation, in the next section we will discuss the some of the rationale for our implementation decisions.

When posers create Problem Templates and their associated Operators, they use web forms generated by CCK to input their Python code (described in Section 3.1). CCK takes this text input, and stores each *poser snippet*–e.g. "State Initalization" code, "Visualization" code, Operator "Precondition" code–directly into separate fields in the SQL database, as is, and associating them with the problem template with which they belong. At this point, none of this code is actually run, only stored into the appropriate database fields.

When a solver then creates a new instance of a Solving Session from a poser's Problem Template, then both the CCK and the TStar-C modules are involved. First, CCK generates the web form for the Solving Session instance's creator to specify the name, description, user group, and settings for this solving session (described in Section 4.1). CCK then creates a

---

[2]Implemented by Christopher Clark

[3]Implemented by David Broderick.

[4]Implemented by Robert Thompson. Actually named simply *agent* module in module code.

[5]Implemented by Tyler Robison. Actually named *cellophane* module in the module code.

new unique solving session identifier (`sid`) to identify this session, and stores the session's settings and `sid` in the database.

Once this session is created and stored in the database by CCK, TStar-C begins its work of constructing the tree for this solving session. It does so by composing together the different, relevant poser snippets–written earlier and stored in the database by CCK–associated with this problem template, inserting wrapper code around it, and writing this code out to a Python-executable file called a *tstar-file*. Then, TStar-C then executes the tstar-file and uses the results to create a new solving session tree node, called a TStar node, and stores it into the database, which is then accessed and retrieved by solving session user interfaces. These TStar nodes are stored in their own database table, the *tstar_nodes* table, which stores the data for each of the actual nodes in a solving session tree. The database table's fields are described in Table 6.1.

Let's go into more detail about how these tstar-files work. Not all of the problem template code is run at once in these files, only the relevant poser snippets are included in each file, as needed. These poser snippets are able to communicate with each other because they all use the same CoSolve-reserved variables names, e.g. `S, D, PARAMETER, IMG_HIGH, IMG_LOW`. There are several *execution phases* for which TStar-C creates and runs these tstar-files. These *execution phases* are:

Data - Occurs once, at the beginning of a solving session, before the Initialization phase. The *Common Data* code is run at this time, which generates any data that will remain constant for the rest of the solving session (for example, generating a random maze that the solvers will try to navigate through in the solving session, etc.)

Initialization - Occurs only when a new solving session is created. Primary purpose is to execute the *State Initialization* poser snippet, which then creates the root node of the solving session tree.

Transformation - Occurs every time a solver applies an Operator to a node. Runs the *State Transformation* code on the node the operator was applied to, creating a transformed state which is then stored as a new TStar node.

Precondition - Occurs after the Transformation phase, i.e. immediately after the creation

Table 6.1: Fields that comprise each TStar node, in the *tstar_node* database table.

Field	Description
tnid (int)	Primary key. Auto-generated unique identifier for this TStar node.
uid (int)	User ID of user who created this node, i.e. applied the operator to create this node.
timestamp (int)	Creation time, i.e. date and time this node was created.
sid (int)	Solving Session ID of the solving session that this node is part of.
state (text)	A string representing the state of the `S` dictionary variable, e.g. `S={ 'x':42, 'y':["red","yellow","blue"] }`
type (text)	Denotes what form the *state* field assumes. Currently, all nodes have `python dictionary` as their *type*, but presumably an image, URL, or binary object of some sort could also store information that represents the state, in which case *type* might be `image`, `URL`, etc.
parent (int)	The *tnid* of the parent of this node, in the solving session tree. This references another node in this table. The root of a tree has a *parent* value of `null`.
operator_nid (int)	The ID of the operator that had been applied to create this node.
operator_vid (int)	The version ID of the operator that had been applied to create this node. (Most objects in the system can have versions. Let's say a solver applies an operator to create node A. Later, the poser edits the code for the operator, and another solver applies the operator and creates another new node B. A and B will have the same *operator_nid*, but different *operator_vid*s.)
preconditions (text)	A list of *operator_nid*s of the operators that can be applied to this node, based on the preconditions of each of the operators in this problem template. e.g. `[324,123,889]`
operator_parameter_value (text)	String representation of the value of the parameter that the solver entered when applying the operator to create this node, e.g. if the operator to create this node were "Place building at this location" for some template, the parameter value would be a representation of the location that the solver then enters, perhaps coordinates like `(9,9)` depending on the template.
operator_parameter_type (int)	An integer, one of 0, 1, 2 or 3, representing the different possible types of parameters CoSolve can accept (i.e. text string, file input, multiple parameters, or mouse click location). Represents what type of parameter the *operator_parameter_value* for this node stores.

of a new TStar node. Evaluates the *Precondition* code of every Operator associated with this problem template, and has a result of either True or False for each, representing whether or not that Operator is valid and may be applied to the state in the new TStar node.

Visualization - Also occurs after the Transformation phase, i.e. after the creation of a new TStar node. The poser snippet containing the *Visualization* code is now executed, based on the state of the newly-created TStar node. TStar-C then takes the output of the *Visualization code* and writes it to an image file on the server, which solving interfaces can then access and display.

During each of these phases, TStar-C retrieves the relevant poser snippets from the database and writes a single tstar-file to be executed. The typical tstar-file is structured as follows:

1. Identifying information for the file, comments, headers.
2. `__phase__` variable is set to be `INIT`, `TRANS`, `VIS`, etc. (for each of the phases, Initialization, Transformation, Visualization, etc.). E.g., for *Precondition* phase:

   ```
   __phase__='PRECOND'
   ```
3. *Common Data* code, which creates and initializes the `D` dictionary variable. Included during any phase except the *Data* phase.
4. State dictionary variable for the appropriate node, e.g.

   ```
   S = {'x': 3, 'y': 5}
   ```

   This *state* variable is retrieved directly from the *tstar_node* database and written here. Only placed here if in the *Transformation, Precondition, Visualization* phases, because otherwise, the `S` state variable has not yet been initialized.
5. If in the *Transformation phase*, parameters entered by solvers is parsed and their values placed in a `PARAMETER` variable here.
6. *Common Code* inserted in its entirety. Notice that the Common Code can make use of any of the variables now defined earlier in the file. For example, depending on the

phase, the `D`, `S`, and `PARAMETER` variables can be accessed, as well as the `__phase__` variable, if the poser wished to have code that executes only during certain phases.

7. At this point, poser snippets are included based on which *execution phase* the file is being created for:

   - *Data* phase: *Common Data* poser snippet
   - *Initialization* phase: *State Initialization* poser snippet
   - *Visualization* phase: *Visualization* poser snippet
   - *Transformation* phase: A transformation phase occurs when a solver applies an operator. So the *State Transformation* poser snippet for that operator will be included here.
   - *Precondition* phase: TStar-C will retrieve the *Precondition* poser snippets for all of the operators associated with this problem template. Recall that these are in the form of a boolean expression. TStar-C creates a list[6] with key-value pairs, with each operator ID as the key and the true/false value of the *precondition* expression, e.g.

     ```
     # assume operators contains list of the operator
     # objects from the database with nids 123 and 432
     preconditions = []
     preconditions.append(operators[0].nid, eval(operators[0].preconditionSnippet))
     preconditions.append(operators[1].nid, eval(operators[1].preconditionSnippet))
     # preconditions variable would then evaluate to, e.g.:
     # [(123, True), (432, False)]
     ```

   Notice again, for all of these phases, that functions or variables defined in *Common Code*, *Common Data* and other included poser snippets can now be used in whatever snippet was included in this section.

8. Output results, depending on the phase:

   - *Initialization*, *State Transformation* phases: Output the state dictionary variable:

     ```
     print S
     ```

   - *Data* phase: Output the Data variable:

     ```
     print D
     ```

[6]Code shown here is an illustrative example of what happens, and is not the actual code.

- *Visualization* phase: Takes the contents of the `IMG_HIGH` and `IMG_LOW` variables created by the poser in the Visualization code and writes them out to an image file on the web server, which can be retrieved by its URL based on its problem ID, session ID (sid) and TStar node ID (tnid), e.g.:

  ```
  IMG_HIGH.save("solvingsession/problem756/session3611/state10826-high.png")
  IMG_LOW.save("solvingsession/problem756/session3611/state10826-low.png")
  ```

  No other output is explicitly printed.
- *Precondition* phase: Takes the `preconditions` list created earlier, filters out those with a `False` value, and prints out the *operator_nid* of the remaining items. Note that this would return a list of only the operators that are valid to apply to this node to create new children:

  ```
  print [item[0] for item in preconditions if item[1]]
  ```

An *Initialization* tstar-file and a *Data* tstar-file is created for each solving session, and a *Precondition* tstar-file, *Transformation* tstar-file and *Visualization* tstar-file is created for each TStar node that is created. Then, when the TStar-C module executes the appropriate tstar-file, it captures the output print stream, if any, from that file. As shown above, this output will be either state information, common data, or a list of precondition-valid operators. TStar-C takes this information, and stores it in the appropriate database, either as solving session information in the case of common data, or in the *tstar_node* database table for state and preconditions.

TStar-C does not directly handle any user interaction or tree visualization, other than the default web page, which statically displays the image files for the tree, and has only a rudimentary form for applying operators. Instead, TStar-C stores all its information in the database, and this information can be accessed by user interface clients through the CoSolve API, described next.

6.3 CoSolve Web Services API

We will give only a brief summary of the CoSolve Web Services API here, as the full documentation for developers is available online.[7]

[7] All available CoSolve services is listed and described at at `http://cosolve.cs.washington.edu/admin/build/services`, but users need an approved CoSolve developer account to access and use these services.

CoSolve's services API provides a collection of methods that can be used by remote applications (clients) created outside of CoSolve's Drupal modules to access information about solving sessions, apply operators, create and view annotations. The Tstar, Group Roles, CoSolve Consultant, and AI Agent modules all make their own service methods available for clients to use. Currently, the API only allows access to solving sessions, and does not yet allow remote applications to create and edit problem templates.

To use the API, clients authenticate to the CoSolve server with a CoSolve solver's user account, and make methods calls to the CoSolve web service over HTTP. For instance, to apply an operator to a node in a solving session for some template, the client would call the `tstar.applyOperator` method, and pass it the following arguments: session authentication key, solving session ID (`sid`), problem template ID (`pid`), ID of the operator to be applied (`operator_nid`), TStar node ID of the state the operator is to be applied to (`tnid`), and a string or array of any parameters to be passed to the operator. CoSolve's API then returns an object containing information about the newly-created TStar node, as listed in Table 6.1 and stored in the *tstar_node* database, and including URLs of the location of the visualization images on the server, to be displayed by the client.

The Flash client interface uses this API to make AMF (Actionscript remoting) calls to this service, retrieving information to display the UI described in Section 4.1. The actual formatting of the display is done within the Flash client's Actionscript code, and in particular, we use Flare visualization toolkit to display the tree[8]. Because TStar-C does not specify the display of the tree, client UIs that make use of the CoSolve API can display the tree however they like. For example, the Java client, created by Christopher Clark, displays a single node at a time rather than the entire tree, and has functionality for moving node-by-node up and down a branch. In this way, solvers can interact with the same solving session tree in different ways, depending on the best way to visualize a particular problem or on solvers' personal preferences, and it is easy to create clients for CoSolve on other platforms, like mobile devices, for instance.

---

[8]Flare (http://flare.prefuse.org/) is the Actionscript version of Jeffrey Heer's Prefuse framework [62].

6.4 Design Rationale and Trade-offs

Now that we've seen the inner workings of CoSolve, let's look at the design rationale behind some of the technical decisions we had to make implementing CoSolve.

Our first challenge was in trying to decide how to create a problem posing system. We knew we wanted to give posers enough power and flexibility to create useful problem templates, so we wanted them to have the ability to use a full-fledged programming language. On the other hand, we were concerned about the security of our system–allowing users to input and run arbitrary code on our server is a recipe for disaster. A malicious poser could wreak much havoc, but even an innocent user could unintentionally write code that, say, has an infinite loop, or some other error. Additionally, we wanted to design a posing system that would be easy for novices to learn, but could still hook in to the rest of CoSolve's functionality.

We address these issues in several ways. First, we created two user statuses, the "poser" status and the "solver" status, analogous to being a CoSolve problem poser or a CoSolve problem solver, described in earlier chapters. [9] All user accounts on CoSolve are initially "solver" status accounts, meaning they can create and participate in solving sessions, but are not allowed to write any code for Problem Templates or Operators. "Solver" status users can later request to become "poser" status users, and which will give their user account all the same access permissions as for a "solver," but now they can also create Problem Template and Operators, and input code for poser snippets. We, as the administrators of the CoSolve system, can then approve their request, and in this way, screen potential hackers, and provide assistance to novice posers. Also, as a precaution, files such as tstar-files and image visualizations can only be written to a specific directory for a particular solving session. For the most part, our system has worked without incident. Despite many spam accounts being created and posting content on CoSolve, we have managed to keep out malicious code so far. We also investigated sandboxing tstar-file execution, which we would still like to implement as future work, such that tstar-file execution that has gone astray

[9] Actually, these are called "roles" both in Drupal, and in CoSolve, but we will call them "statuses" here to differentiate between these permissions-based access roles, and the collaborative group roles.

can be shut down without affecting the rest of the system.

Secondly, and more interestingly, we developed a system of scaffolding our problem templates, as explained in Section 3.1, and of using CoSolve-reserved variable names for important CoSolve problem template objects, but other than these restrictions, posers were allowed to write any Python code they wished. Problem templates are divided up into specific sections, with descriptions of each in the web interface, and with starter code in each snippet, so that novice posers already have a working problem template to begin with. To make these snippets work together, they each refer to the same set of CoSolve variables (the `S` state dictionary, etc.), and starter examples of their use are provided in the scaffolded code. Scaffolding our templates and having this snippet structure makes CoSolve posing easier to use and hopefully helps reduce errors for novices. As seen by the many templates already created by a large number of posers, CoSolve's posing system seems to work sufficiently well thus far.

The next issue we faced was how to take code written by posers and actually create solving sessions out of them. The problem here was that we didn't want solving sessions to be continuously running problem template code the way a normal program would continuously run after starting up. We, as CoSolve developers, wanted to create the solving session and implement the solving user interaction so that posers would not have to bother with explicitly creating a solving session tree, creating menus for operators, etc. We wanted to the CoSolve system to take care of these things for the poser, so that the poser can focus on problem formulation, not Python code details.

This is why we came up with separate poser snippets, that are then run according to different *execution phases*. Posers need to simply write enough code to answer a question for each snippet, for example, for an operator's *State Transformation* code, the question would be, "Which values of this state are changed, and change to what, when this operator is applied?" rather than worry about states in the rest of the tree, or about UI details, or parsing parameter input. Then, since we have separated out solving computation into *execution phases*, at any given time, we need only worry about computing the relevant parts of the code as needed, rather than the entire thing, and different parts of the tree can be created independently by different users as they wish. Note also that this asynchronous

method of state-space-tree search is different from the traditional AI model, which generally searches the entire tree at once.

There are a few implementation decisions for which it is uncertain whether the best choice was made, and we would like to highlight those decisions, and their alternatives, now, in hopes that this discussion can help in future implementations of similar systems.

The first we would like to discuss is the pre-computation of operator preconditions. When a TStar node is first created, all of the problem template's operators are checked and their preconditions computed, and those that are true are stored along with the TStar node. The advantage of this is that, since these preconditions are computed once, right when the node is created, we don't have to compute them every time a solver opens the operator menu, to decide which operators to show, and this seemed a reasonable choice at first. In practice, when a poser is developing a problem template, the poser tests its by creating and running a test solving session. Since the preconditions for an existing node are not re-computed at any point, the poser must create a new session every time he or she edits the precondition code. Secondly, the poser may edit other parts of the code in such a way that the precondition code no longer has the same meaning as it did before, so a previously valid operator might not be a valid choice any more for the same node.

In such a case, it may seem obvious that preconditions for existing nodes should simply be recomputed any time the problem template is edited. However, from a solver's perspective, nodes should remain consistent throughout a tree. Assume a solver applies operator X to the root node to create a child node. Then she continues to apply other operators, creating a branch of descendants beginning with that child node. What if, after these descendants are created, the preconditions are changed such that operator X is no longer an available operator for the state at the root node? What should happen to the descendants already created? Should they be removed? This would violate the idea of CoSolve serving as a record of solving activity, and is would also be confusing for the solver. For these reasons, we decided to precompute operator preconditions, and maintain them throughout the life of a node.

Another difficult implementation decision we faced was in how to safely and efficiently execute the tstar-files, and whether the composed poser snippets should even be stored in

files, or stored on the server at all. Each file is only run once and writing files to disk is expensive in terms of space and time. One option would have been to execute the Python code directly from within the PHP code in the TStar-C module right away, rather than storing the code. However, we decided to store the code in tstar-files in the end for several reasons. First, we wanted to have a record of code executed on the server, both to fix any server errors if needed, and also potentially as a source of data for future research. Secondly, the files are useful to posers when debugging problem templates: when TStar-C receives a Python error, it displays the tstar-file to the poser, along with the error message, and the line number in the tstar-file of the code that generated the error. However, CoSolve is very slow, to the point of affecting user behavior, as mentioned in Chapter 4, and avoiding writing extra files to disk could potentially speed up the system.

The last implementation decision we would like discuss involves the use of Drupal to build CoSolve. Drupal itself tends to be very slow, because, although its infrastructure makes it easy to extend by adding custom modules, the way in which it is designed means that to generate a single page, many functions within many of those modules have to be checked and potentially called. Drupal sites usually solve this problem by implementing abundant amounts of caching, but in our custom CoSolve modules, new states need to be generated all the time, and thus, they cannot be cached. We could perhaps have bypassed this problem in the first place by not using a large pre-built framework, that comes with plentiful unneeded functionality, and simply rolling our own custom site, but that may have taken more development time.

Additionally, another decision we made was to use our own custom database table for *tstar_nodes*. Implementing it with CCK would have taken less development time, and also would have allowed us to use the default CCK web services rather than have to create our own API, and kept our code more consistent (Solving Session, Problem Template, Operator, and Annotation are all content types implemented in CCK). However, we thought that creating our own table would allow us to streamline the code execution process, and avoid the overhead that the monolithic CCK creates. At the same time, CCK does implement its own caching, so it's unclear if implementing TStar nodes as a CCK content type would have ended up being more efficient in the end.

Related to the topic of improving CoSolve's efficiency and speed is the idea of off-loading intensive computation to other servers. As it currently stands, all TStar nodes in the system use Python dictionaries to represent their states, which is why the *type* field of all TStar nodes in the *tstar_nodes* table is "`Python dictionary`". However, states could also be stored using alternative representations, for example, the *type* field could be set to a URL, and the *state* field could instead store a URL to a location on another server that perhaps can generate a more efficient state representation, or do more intensive operations appropriate for problem templates that involve, say, scientific simulations.

Conclusion

This chapter has presented an overview of the components in the CoSolve system, explained the details of how CoSolve's unique posing-solving interaction works, and addressed the implementation trade-offs involved in designing the CoSolve system. We have now concluded our discussion of CoSolve's implementation, and design rationale. The next chapter concludes this thesis with a summary of the work on CoSolve presented here, and outline directions for future work in state-space-based, collaborative problem-solving systems.

Chapter 7

CONCLUSIONS AND FUTURE WORK

Online collaborative problem-solving has the potential to engage people from all around the globe in working together to find solutions to problems, big or small. From students working together remotely to solve mathematics problems, to neighbors designing a new community center, to potentially even everyday citizens from around the world discussing socio-political issues such a global climate change, the Internet has given us new opportunities for communication in problem-solving.

In particular, the rise of social technologies for the web has led to the creation of new tools for collaboration. The idea of crowdsourcing problem-solving has gained momentum in the past decade, both commercially, e.g. Amazon's Mechanical Turk (MTurk) [14] and Innocentive [15], and in academia, e.g. the Polymath Project [63]. MTurk allows users to hire remote workers on the web for small jobs. This structured approach has the advantage of providing huge numbers of workers with well-defined tasks; however more open-ended, creative, problem-solving tasks are less appropriate in this system. Additionally, many crowdsourcing systems are not directly collaborative; solvers either have no contact with one another, as in MTurk, or solvers compete with each other, as in Innocentive, rather than work together.

At the other extreme are loosely organized, very creative, problem-solving projects such as those found in Substepr [64], or in the successful Polymath project [32] [37], which has gathered mathematicians together to cooperatively solve research problems in their field through blog entries, threaded online comments, and wiki pages. However, new participants have to sift through a great deal of past discussion, and it may be difficult for them to understand what state the project is in, and where one can best contribute.

Is there a middle ground where collaborative problem-solving can occur, while allowing a structured process and manageable, defined tasks? To explore this idea, we developed

CoSolve, a web-based collaborative problem-solving environment, which uses state-space search to provide structure while offering tools to enhance collaboration. State-space search models problems as a tree of states representing possible solutions, or steps toward a solution, in a problem space. Operators can be applied to states to transform them into new states, i.e. new potential solutions or steps toward solutions.

Our decision to use state-space search was inspired by Herbert Simon's ideas in the *Sciences of the Artificial* [1], and in this dissertation, we have provided an implementation and investigation of how this method can be used in practice with real users. We realize that using state-space search as a model for problem solving, and especially for design, can be a controversial issue, as there are potentially flaws in the restrictiveness of this approach. For example, in terms of design, Schön's Reflective Practitioner theory [65] argues that the ability to make design decisions is due to experience, and learning from one's experiences. This generates a kind of intuition about design in one's professional field, that perhaps could not be captured in state operators. However, we think that our choice of state-space search as a model is appropriate for our research because we feel that developing a structured approach can be useful both as a way to study the problem-solving and design process, and potentially as way to help learners who may be unfamiliar with problem-solving or with a certain domain. Examining a CoSolve problem-solving process may yield insights similar to how the Polymath project has educated novice mathematicians on how to do mathematics research by making the process open and available for all to see [32]. Additionally, with a structured approach we can take advantage of using AI agents in combination with human intuition.

In this dissertation, we have presented CoSolve, an online environment that allows users to collaboratively create search trees that model the problem space they wish to explore. We created a problem-posing framework for CoSolve that allows users to specify the problems they wish to solve, and explored how problem-posers utilize this functionality (Chapter 3). We also conducted a formal user study to investigate how problem-solvers use the CoSolve solving interface, and an informal design activity to study whether CoSolve is feasible for open-ended, creative problems (Chapter 4). We also investigated how to increase participation in collaborative problem-solving by building and evaluating a system of user roles

(Chapter 5). This chapter will summarize our research findings and present ideas for future work in each of these areas.

7.1 Problem Posing

7.1.1 Summary

CoSolve is a problem-solving environment, but users need to first specify the problem they wish to solve. These users, called *problem-posers*, can do so by creating *problem templates* using CoSolve's web-based, scaffolded, *problem-posing interface*. Using this interface, problem-posers can edit snippets of Python code that represent the state structure and operators of a problem. This interface is a framework that provides scaffolding for our problem-posers' problem specification–instead of writing, say, an entire executable program, they simply specify parts of their problem space, and CoSolve composes these together with additional "glue" code to allow solvers to create solving session trees. In this way, although problem-posers still need to have some programming knowledge, the interface provides enough scaffolding such that intermediate-level programmers are able to successfully create problem templates.

After this problem-posing framework was developed, we evaluated it by examining problem templates created by students in several offerings of an undergraduate artificial intelligence class. We found that students were able to successfully specify a variety of problems with our interface. We also examined the different ways in which students worked with the interface and structured their code. In this dissertation, we also discussed the scope of problem-posing in CoSolve, and introduced the notion *poseability*–how readily a problem can be turned into a state-space problem template. Problems with clear state variables, operators, and goal criteria are the most easily poseable. "Wicked" problems are more challenging to pose, and creating problem templates for such problems will usually require a simplification or modification of the actual problem. For these types of problems, we see CoSolve as medium for encouraging communication and conversation between stakeholders and a way to generate awareness and educate people about the issues surrounding a problem.

7.1.2 Future Work

This section discusses several possibilities for future work in problem-posing:

Solving session initiation parameters. In Section 3.2.2, we discussed the idea of meta-problem templates, i.e. problem templates that have operators that allow solvers to provide further problem specification within the solving interface itself, rather than the problem-poser doing so in the problem template code. I also discussed related problem templates in which problem specification changes between instances of solving sessions. For example, a "maze" problem template may randomly generate a new maze for each solving session that is created from that template. These practices suggest that users wish to create general problem templates that have the flexibility to solve multiple, related problems. One possibility to address this is to allow solvers, when initializing new solving sessions, to set certain parameters for their particular solving session. These options would be specified by the problem posers, and the solving-session initiators supply or select the details. For instance, for mathematics education, a general geometry construction problem template could be prepared by curriculum specialists. Then, a teacher could instantiate solving sessions for groups of students, and could select specific geometry problems ("bisect a line", "create an equilateral triangle") for each particular solving session he wishes to create.

Problem-poser debugging facilities. The problem-posing interface is usable, but still in an early stage of development. As such, there are not many features to support problem-posers' debugging processes. One of the students we talked to suggested adding the ability to debug on a state-by-state basis. Currently, when a poser's Python code generates an error, the error messages and code line numbers are presented for the session overall, and it can be tedious to track down the specific state that caused the error. Also, new solving sessions have to be generated after changing code to see the changes, or to recreate errors, which can also be time-consuming. The ability to visually attach the error messages to the states that generated them, or to "refresh" solving session states by re-generating them using the same sequence of operators, would be helpful for our problem posers.

IDE plug-in for CoSolve. In our study, we found that some problem posers preferred to use the web interface, others preferred to use a separate text editor or IDE (Integrated Development Environment) entirely, while the majority preferred to go between both, writing some code in the web interface, and writing some in a separate IDE and then moving it into the CoSolve for testing. Our initial intention in creating the web interface is to make it easy for novice programmers to pose problems in CoSolve, without having to set up or install anything. For more advanced users who prefer to use an IDE, one possibility for future work would be to develop an Eclipse plug-in that will help users by filling in CoSolve Python code snippets and recognizing CoSolve keywords. This plug-in would also connect to the CoSolve server and allow problem-posers to test their problem templates by generating solving sessions from their IDE directly.

Exploration of visualization editing options. In our analysis of students' problem templates, we found that the problem template area that usually contained the most lines of code was the Visualization area. To make problem posing easier, we could explore ways in which default visualizations could be provided, based on the state structure specification. For instance, if a state contains several numeric variables, these could be represented in a bar chart, with the length of each bar representing the magnitude of the corresponding variable. Problem posers would have the option of customizing the visualizations, e.g. selecting appropriate chart types, combining visualization types, choosing color schemes, specifying the possible range of the value of a variable, etc. In this way, we hope to explore ways to make it easier for problem-posers to create visualizations of their states.

Problem-posing interface for non-programmers. Problem-posing in CoSolve currently requires some programming knowledge. To make CoSolve more accessible, we would like to investigate creating a posing interface for non-programmers. One possibility would be to modify the existing web posing interface to allow posers to create problem templates entirely by selecting options from menus. For example, state variables could be created by selecting a variable type from a set of choices (e.g. number, text, etc.). Posers could create operators by composing together a set of pre-defined operations (e.g. changing a

value; arithmetic operations on numbers, etc.). We are interested in discovering what the strengths and limitations of such an interface would be, and at minimum, such an interface would be useful for teaching novice users how to pose a problem in CoSolve.

7.2 Problem Solving

7.2.1 Summary

The next component of CoSolve is the problem-solving process itself. In this process, solvers begin with a root node of a tree. This root node represents the initial state of the unsolved problem. By applying operators to a state, solvers can generate new states to represent possible solutions, or to discover a sequence of states that lead to a solution state. The CoSolve solving interface allows solvers to directly apply these operators to generate states visually, and to interact with and manipulate views of the state-space tree. This is different from the traditional implementations of state-space search, in which the state-space tree is represented in the system internally, instead of being exposed to the user. In this way, CoSolve incorporates human intuition and creativity as a heuristic in the state-space search process. Additionally, we provide extra functionality to support collaboration, for example, the ability to create textual node annotations as a way for solvers to communicate with each other regarding the nodes.

We conducted a formal user study to see whether human subjects are able to make use of the tree visualization of a problem space, and whether they are able to collaborate using the interface. We found that subjects were indeed able to use CoSolve for successfully solving a stylized urban simulation problem called CitySim, and that they found the visualization easy to learn and helpful in solving CitySim. However, we found that they had difficulty maintaining awareness of their teammates' activities and difficulty navigating the tree. We discuss ideas for addressing these issues in the next section on future work.

Finally, we examined the feasibility of using CoSolve for open-ended, creative problems by using CoSolve to design educational games. We created two face-to-face educational games, and found that we were able to model our game designs using CoSolve by creating a general state-space design template.

7.2.2 Future Work

We have identified several possibilities for future work in the area of tree visualizations of problem state-spaces:

Group awareness features. In our user study, we found that many of our subjects felt they had a difficult time maintaining awareness of where their teammates were working in the tree. As discussed in Section 2.4, plentiful research exists on the topic of group awareness in computer-supported collaborative work (e.g. Dourish and Belotti [19], Gutwin and Greenberg [20]). In particular, for tree visualizations, our subjects felt that the ability to see their teammates' viewports overlaid onto the tree, or to see a mini-map of the entire tree while working on their own zoomed-in part of the tree, would be helpful. Additionally, we could explore ways of displaying a status area that would show their teammates' most recent actions, and allow them to jump to those areas of the tree.

Tree navigation features. We also identified an issue with general tree navigation in the interface. We wanted to encourage users to explore their teammates' contributions to the tree; however as the tree grew larger, subjects became more and more reluctant to pan around to view different parts of the tree because they were afraid of losing their current place in the tree and not being able to find it again. Our subjects mentioned several ideas we would like to explore. One is a "back button" to jump back to previously viewed nodes. Many subjects also mentioned that they tended to view nodes either across a single level, or down a single branch, and that shortcut keys for navigating level-wise or branch-wise would be helpful. Another idea we could potentially examine is creating personal annotations that are not shared with teammates. Some subjects already mentioned that they created annotations as notes to themselves; creating *personal* annotations would mean that solvers don't have to worry about cluttering up the team's annotations list with their personal bookmarks.

Identifying identical states. As mentioned in Section 4.1.1, one of the goals of CoSolve's tree visualization is to provide a record, or history, of the problem-solving process. As such,

multiple nodes may be created with identical state values, but they are displayed as separate, unique nodes in CoSolve because they may differ in parentage. However, it may be useful to solvers to know when there are identical states in the tree. We would like to explore ways of alerting solvers in such situations, and perhaps merging related branches.

Accessibility issues in tree visualization CoSolve's main feature is its tree visualization, but is there any way we could make the solving interface accessible for users using non-visual interfaces, such as screen readers? One area for potential future work is to find ways to address collaboratively exploring nodes, navigating between nodes, and understanding a tree's structure via an audio or other alternative interface.

7.3 Collaboration and Communication

7.3.1 Summary

Finally, we wanted to explore ways to encourage collaboration in CoSolve by increasing participation, and increasing equitable participation. To do this, we developed the CoSolve Roles system, which assigns users to one of three roles: brainstormer, supporter and critic. The brainstormer creates new nodes as a way of exploring new ideas. The supporter looks for nodes that are potentially on the solution path and creates "positive" annotations to alert collaborators of these nodes. The critic, on the other hand, evaluates nodes to find ones that are less promising, and marks them as such with "negative" annotations. We then performed a user study evaluating the effectiveness of the roles system versus a control group. We did not find statistically significant differences in measurable participation metrics, but in subject interviews, we found that roles subjects felt they collaborated more due to roles than they would have without roles. At the same time, we identified several issues with our study and the roles system, which we would like to address in our future work.

7.3.2 Future Work

Our ideas for future work on collaboration in problem-solving include:

Longer-duration solving sessions. The subjects in our study only had an hour and a half to solve the problem they were given. Subjects mentioned that they felt an immense

amount of time pressure, and so, may not have taken the time to look at their teammates' nodes and annotations. This artificial time restriction was implemented for logistical reasons, but a longer term study, or solving sessions over a longer duration, would provide us with data on how subjects would behave in a more natural solving session. Subjects would have more time to review their teammates' actions; perhaps in this context, we would see a greater difference in the effect of roles.

Larger teams. Because the teams in our study were fairly small, it may have been easier for our solvers to collaborate than it would have been with larger teams. Roles may prove more useful in situations where there are more team members. Additionally, CoSolve's goal is to eventually support large-scale collaboration. One area for future work would be to investigate whether a roles system affects collaboration in large teams, of ten or more.

Flexible roles. Most of our subjects mentioned that the roles system was extremely stressful and frustrating. Having to switch roles often (every 8.3 minutes) may have broken subjects' focus, and the role-phase countdown timer may have added to their stress. We would like to redesign roles to be more flexible, in order to minimize solvers' stress and frustration. There are several options for doing so. We could make the timed phases longer (i.e. an hour, a day, a week), or we could eliminate timed phases altogether, and instead just show solvers their balance of activity, e.g. ten percent of a solver's activity is creating positive annotations, thirty percent is creating new nodes, etc. In this way, solvers can be made aware of the fact that there are other actions they could be performing, and see what others are doing.

Different annotation types. Subjects in our *solving* study told us that negative annotations were not useful because they did not wish to draw attention to unsuccessful nodes; it was far more fruitful to point out promising nodes. We could potentially investigate having only positive annotations rather than both negative and positive. We could also investigate having different types of annotations, for example, "question" annotations for asking questions about nodes. We could examine whether not having annotation types at

all might change their collaborative behavior.

Chatbox. Subjects in both the CitySim solving study and the roles study mentioned the need to communicate outside of the tree structure. In particular, many mentioned including a generic chat box that would allow subjects to communicate without the need to attach an annotation to a node.

Providing collaboration information. Finally, to encourage collaboration, we could provide more information on the team's collaborative process. Ways to do this include showing users how often others have created child states and branches off of their nodes, and how often others have annotated their work, and potentially correlate these to the team's progress toward a solution. Similar ideas have been explored in the CoSolve Consultant [54], but such information was often hidden behind a series of windows. By directly putting collaboration metrics upfront, and explicitly relating it to the team's progress, solvers can see how well their collaboration is–or potentially, is not–serving them.

7.4 Conclusion

This dissertation presents our research into helping users work together to solve problems. We have built CoSolve as an online environment to explore our ideas on collaborative problem-solving. However, we also hope that CoSolve can encourage a culture of problem-solving. The goal of CoSolve is both to help users solve problems, as well as to educate users about how to solve a problem. By explicitly breaking down the problem-solving process into understandable, structured components, we can make this process more transparent, and potentially more accessible. We have found that CoSolve is able to support a variety of constrained and open-ended problems, and that solvers are successfully able to collaborate using this model. We hope that our research can be a foundation for further exploration into possibilities for online collaborative problem-solving.

BIBLIOGRAPHY

[1] H. Simon, *The Sciences of the Artificial.* Cambridge, Massachusetts: MIT Press, 1969.

[2] S. B. Fan, T. Robison, and S. L. Tanimoto, "CoSolve: A system for engaging users in computer-supported collaborative problem solving," in *VL/HCC*, M. Erwig, G. Stapleton, and G. Costagliola, Eds. IEEE, 2012, pp. 205–212.

[3] A. Newell, J. C. Shaw, and H. A. Simon, "Report on a general problem solving program," in *Proceedings of the International Conference on Information Processing*, 1960.

[4] J. E. Laird, A. Newell, and P. S. Rosenbloom, "Soar: an architecture for general intelligence," *Artificial Intelligence*, vol. 33, no. 1, pp. 1–64, Sep. 1987. [Online]. Available: http://dx.doi.org/10.1016/0004-3702(87)90050-6

[5] J. R. Anderson, "ACT: A simple theory of complex cognition," *American Psychologist*, vol. 51, pp. 355–365, 1996.

[6] E. Gallopoulos, E. Houstis, and J. Rice, "Computer as thinker/doer: problem-solving environments for computational science," *Computational Science Engineering, IEEE*, vol. 1, no. 2, pp. 11 –23, summer 1994.

[7] M. Vass, N. Allen, C. A. Shaffer, N. Ramakrishnan, L. T. Watson, and J. J. Tyson, "The JigCell Model Builder and Run Manager," *Bioinformatics*, vol. 20, no. 18, pp. 3680–3681, 2004. [Online]. Available: http://bioinformatics.oxfordjournals.org/content/20/18/3680.abstract

[8] S. Wang, M. Armstrong, J. Ni, and Y. Liu, "GISolve: a grid-based problem solving environment for computationally intensive geographic information analysis," in *Challenges of Large Applications in Distributed Environments, 2005. CLADE 2005. Proceedings*, July 2005, pp. 3 – 12.

[9] J. R. Rice and R. F. Boisvert, *Solving Elliptic Problems Using ELLPACK.* Springer-Verlag, 2012.

[10] H. Kobashi, S. Kawata, Y. Manage, M. Matsumoto, H. Usami, and D. Barada, "A Meta Problem Solving Environment (PSE)," in *Computer Sciences and Convergence Information Technology (ICCIT), 2010 5th International Conference on*, 30 2010-dec. 2 2010, pp. 253 –259.

[11] GalaxyZoo. [Online]. Available: http://www.galaxyzoo.org/

[12] SETILive. [Online]. Available: http://setilive.org/

[13] FoldIt. [Online]. Available: http://fold.it/portal/

[14] Amazon Mechanical Turk. [Online]. Available: http://www.mturk.com/mturk

[15] InnoCentive. [Online]. Available: http://www.innocentive.com

[16] S. Tanimoto and S. Levialdi, "A transparent interface to state-space search programs," in *Proc. ACM Software Visualization*, 2006, pp. 151–152.

[17] S. L. Tanimoto, "Enhancing State-Space Tree Diagrams for Collaborative Problem Solving," *Lecture Notes In Artificial Intelligence; Vol. 5223*, 2008. [Online]. Available: http://portal.acm.org/citation.cfm?id=1432543

[18] S. L. Tanimoto, T. Robison, and S. B. Fan, "A Game-Building Environment for Research in Collaborative Design," in *IEEE Symposium Computational Intelligence and Gameson*. Milano, Italy: IEEE Comput. Soc, Sep. 2009, pp. 96–103. [Online]. Available: http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=5286489

[19] P. Dourish and V. Bellotti, "Awareness and coordination in shared workspaces," in *Proceedings of the 1992 ACM conference on Computer-supported cooperative work - CSCW '92*. New York, New York, USA: ACM Press, 1992, pp. 107–114. [Online]. Available: http://portal.acm.org/citation.cfm?doid=143457.143468

[20] C. Gutwin and S. Greenberg, "Design for Individuals, Design for Groups: Tradeoffs Between Power and Workspace Awareness," in *Proceedings of the Conference on Computer-Supported Cooperative Work*. Seattle, WA: ACM, 1998, pp. 207–216.

[21] P. Dewan, P. Agarwal, G. Shroff, and R. Hegde, "Mixed-focus collaboration without compromising individual or group work," in *Proceedings of the 2nd ACM SIGCHI Symposium on Engineering Interactive Computing Systems - EICS '10*. New York, New York, USA: ACM Press, Jun. 2010, p. 225. [Online]. Available: http://dl.acm.org/citation.cfm?id=1822018.1822054

[22] K. S. Park, A. Kapoor, and J. Leigh, "Lessons learned from employing multiple perspectives in a collaborative virtual environment for visualizing scientific data," in *Proceedings of the Third International Conference on Collaborative Virtual Environments - CVE '00*. New York, New York, USA: ACM Press, Sep. 2000, pp. 73–82. [Online]. Available: http://dl.acm.org/citation.cfm?id=351006.351015

[23] I. Heldal, D. Roberts, L. Bråthe, and R. Wolff, "Presence, Creativity and Collaborative Work in Virtual Environments," in *Proceedings of the 12th International Conference on Human-Computer Interaction: Interaction Design and Usability (HCI 2007)*, J. A. Jacko, Ed. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 802–811.

[24] D. D. Suthers, R. Vatrapu, R. Medina, S. Joseph, and N. Dwyer, "Beyond threaded discussion: Representational guidance in asynchronous collaborative learning environments," *Computers & Education*, vol. 50, no. 4, pp. 1103–1127, May 2008. [Online]. Available: http://linkinghub.elsevier.com/retrieve/pii/S0360131506001655

[25] C. Gutwin and S. Greenberg, "A Descriptive Framework of Workspace Awareness for Real-Time Groupware," *Computer Supported Cooperative Work*, vol. 11, no. 3, 2002. [Online]. Available: http://portal.acm.org/citation.cfm?id=586339

[26] Basecamp. [Online]. Available: http://basecamp.com

[27] GoogleDocs. [Online]. Available: http://docs.google.com

[28] R. Butler, B., Sproull, L., Kiesler, S., Kraut, "Community effort in online groups: Who does the work and why?" in *Leadership at a distance*, L. Weisband, S. & Atwater, Ed. Erlbaum, 2002, pp. 171–193. [Online]. Available: http://www.communitylab.org/?q=node/102

[29] T. Gowers, "Is Massively Collaborative Mathematics Possible?" 2009. [Online]. Available: http://gowers.wordpress.com/2009/01/27/is-massively-collaborative-mathematics-possible/

[30] ——, "A Combinatorial Approach to Density Hales-Jewett," 2009. [Online]. Available: http://gowers.wordpress.com/2009/02/01/a-combinatorial-approach-to-density-hales-jewett/

[31] ——, "Why this particular problem?" [Online]. Available: http://gowers.wordpress.com/2009/02/01/why-this-particular-problem/

[32] T. Gowers and M. Nielsen, "Massively collaborative mathematics." *Nature*, vol. 461, no. 7266, pp. 879–81, Oct. 2009. [Online]. Available: http://www.nature.com/nature/journal/v461/n7266/full/461879a.html

[33] T. Gowers, "Polymath1 and Open Collaborative Mathematics," 2009. [Online]. Available: http://gowers.wordpress.com/2009/03/10/polymath1-and-open-collaborative-mathematics/

[34] M. Nielsen, "The Polymath project: scope of participation," 2009. [Online]. Available: http://michaelnielsen.org/blog/the-polymath-project-scope-of-participation/

[35] B. Shneiderman, "Human-Centered Agenda for Discovery and Innovation: Statement to President's Council of Advisors on Science and Technology (PCAST) Subcommittee on Networking and Information Technology," Department of Computer Science, University of Maryland, College Park, College Park, MD, USA, Tech. Rep., 2006. [Online]. Available: http://www.cs.umd.edu/~ben

[36] T. Gowers, "Can Polymath Be Scaled Up?" 2009. [Online]. Available: http://gowers.wordpress.com/2009/03/24/can-polymath-be-scaled-up/

[37] J. Cranshaw and A. Kittur, "The polymath project: lessons from a successful online collaboration in mathematics," in *Proceedings of the 2011 Annual Conference on Computer Human Interaction*, Vancouver, BC, 2011, pp. 1865–1874. [Online]. Available: http://dl.acm.org/citation.cfm?id=1979213

[38] K. Luther and A. Bruckman, "Leadership in online creative collaboration," in *Proceedings of the ACM 2008 conference on Computer supported cooperative work - CSCW '08*. New York, New York, USA: ACM Press, 2008, p. 343. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1460563.1460619

[39] Wikipedia, "Wikipedia:About - Editorial Administration, Oversight, Management," 2009. [Online]. Available: http://en.wikipedia.org/wiki/Wikipedia:About#Editorial_administration.2C_oversight.2C_and_management

[40] H. T. Welser, D. Cosley, G. Kossinets, A. Lin, F. Dokshin, G. Gay, and M. Smith, "Finding social roles in Wikipedia," 2008, presented at the Annual Meeting of the American Sociological Association. [Online]. Available: http://www.allacademic.com/meta/p243019_index.html

[41] S. L. Bryant, A. Forte, and A. Bruckman, "Becoming Wikipedian: Transformation of Participation in a Collaborative Online Encyclopedia," in *Proceedings of GROUP: International Conference on Supporting Group Work*, Sanibel Island, FL, 2005, pp. 1–10.

[42] E. S. Raymond, *The Cathedral and the Bazaar*, 1st ed., T. O'Reilly, Ed. Sebastopol, CA, USA: O'Reilly & Associates, Inc., 1999.

[43] N. Ducheneaut, "Socialization in an Open Source Software Community: A Socio-Technical Analysis," *Computer Supported Cooperative Work (CSCW)*, vol. 14, no. 4, pp. 323–368, 2005. [Online]. Available: http://www.springerlink.com/index/10.1007/s10606-005-9000-1

[44] K. Nakakoji, Y. Yamamoto, Y. Nishinaka, K. Kishida, and Y. Ye, "Evolution patterns of open-source software systems and communities," *Proceedings of the international workshop on Principles of software evolution - IWPSE '02*, no. January 2001, p. 76, 2002. [Online]. Available: http://portal.acm.org/citation.cfm?doid=512035.512055

[45] F. Barcellini, F. Détienne, J.-M. Burkhardt, and W. Sack, "A socio-cognitive analysis of online design discussions in an Open Source Software community," *Interacting with Computers*, vol. 20, no. 1, pp. 141–165, 2008. [Online]. Available: http://linkinghub.elsevier.com/retrieve/pii/S0953543807000793

[46] F. Barcellini, F. Détienne, and J.-M. Burkhardt, "Participation in online interaction spaces: Design-use mediation in an Open Source Software community," *International Journal of Industrial Ergonomics*, vol. 39, no. 3, pp. 533–540, 2009. [Online]. Available: http://linkinghub.elsevier.com/retrieve/pii/S0169814108001637

[47] G. van Rossum, "Origin of BDFL." [Online]. Available: http://www.artima.com/weblogs/viewpost.jsp?thread=235725

[48] K. Luther, "Supporting and transforming leadership in online creative collaboration," *Conference on Supporting Group Work*, p. 1, 2009. [Online]. Available: http://portal.acm.org/citation.cfm?id=1531674.1531735

[49] Getting Started with Rails: Getting Up and Running Quickly with Scaffolding. [Online]. Available: http://guides.rubyonrails.org/getting_started.html#getting-up-and-running-quickly-with-scaffolding

[50] B. Sherin, B. J. Reiser, and D. Edelson, "Scaffolding analysis: Extending the scaffolding metaphor to learning artifacts," *Journal of the Learning Sciences*, vol. 13, pp. 387–421, 2004.

[51] H. W. J. Rittel and M. M. Webber, "Dilemmas in a General Theory of Planning," *Policy Sciences*, vol. 4, pp. 155–169, 1973.

[52] G. Polya, *How to Solve It: A New Aspect of Mathematical Method.* Princeton University Press, 1945.

[53] D. H. Jonassen, *Learning to solve problems: An instructional design guide*, Wiley, Ed. Pfeiffer, 2004, vol. 24, no. 2. [Online]. Available: http://eu.wiley.com/WileyCDA/WileyTitle/productCd-0787964379.html

[54] T. S. Robison, "Opening Up the Collaborative Problem-Solving Process to Solvers," Ph.D. dissertation, University of Washington, 2012.

[55] N. J. Nilsson, *Problem-Solving Methods in Artificial Intelligence.* McGraw-Hill, 1971.

[56] "SimCity," Computer Game. Maxis, California, 1989.

[57] J. Lew, "Making City Planning a Game," New York Times, 15, Jun. 1989.

[58] S. B. Fan, B. R. Johnson, Y.-E. Liu, T. S. Robison, R. R. Schmidt, and S. L. Tanimoto, "Analyzing a Process of Collaborative Game Design Involving Online Tools," in *2010 IEEE Symposium on Visual Languages and Human-Centric Computing.* IEEE, Sep. 2010, pp. 75–78. [Online]. Available: http://www.computer.org/portal/web/csdl/doi/10.1109/VLHCC.2010.19

[59] S. B. Fan, B. Johnson, Y.-E. Liu, T. Robison, R. Schmidt, and S. Tanimoto. (2010) Analyzing a process of collaborative game design involving online tools (long version). [Online]. Available: http://www.cs.washington.edu/ole/collab-design-long-2010.pdf

[60] B. Parr. (May 28, 2009) Google Wave: A complete guide. [Online]. Available: http://mashable.com/2009/05/28/google-wave-guide/

[61] S. Tanimoto, A. Carlson, J. Husted, E. Hunt, J. Larsson, D. Madigan, and J. Minstrell, "Text forum features for small group discussions with facet-based pedagogy," in *Computer-Supported Collaborative Learning*, 2002.

[62] J. Heer, "Prefuse: a software framework for interactive information visualization," M.S. Thesis, Computer Science Division, University of California, 2004.

[63] The Polymath Blog. [Online]. Available: http://polymathprojects.org

[64] Substepr. [Online]. Available: http://substepr.com

[65] D. A. Schön, "The Structure of Reflection in Action," in *The Reflective Practitioner: How Professionals Think In Action*, 1984.

Appendix A

CODE LISTING: "TOWERS OF HANOI" COSOLVE PROBLEM TEMPLATE

This is the entire code listing for the Towers of Hanoi problem template, and is a slightly modified version of the original version created by Richard Rice `http://cosolve.cs.washington.edu/problem/towers-hanoi`. Each section below goes into the corresponding text boxes in the posing interface web form. Detailed explanation of this code can be found in Section 3.1.2.

A.1 Problem Description

Solve the Towers of Hanoi puzzle. We are given three pegs upon which disks can be stacked. No two disks are the same size and no larger disk may be stacked on top of a smaller one. Play starts with a single stack of disks on peg one. A single move moves one disk from one peg to another. The goal is to use multiple moves to shift all the disks to peg two.

A.2 Problem Template Code

Listing A.1: Common Data

```
D = {'numberOfDisks' : 3}
```

Listing A.2: State Initialization

```
S = {'peg' : [list(range(D['numberOfDisks'],0,-1)), [], []]}
```

Listing A.3: Common Code

```
# move top disk between pegs
def move(from_peg,to_peg):
  disk = S['peg'][from_peg].pop() # removes last item from the list
  S['peg'][to_peg].append(disk) # adds it the the end of the list

```

```
# check whether move between pegs is valid
def isValidMove(from_peg,to_peg):
  result = False
  if S['peg'][from_peg]: # if list is has items, i.e. there is a disk on
      the from_peg
    if S['peg'][to_peg]: # if the to_peg has disks on it
      result = S['peg'][from_peg][-1] < S['peg'][to_peg][-1] # check top
          disks on pegs
    else:
    result = True
  return result
```

Listing A.4: Visualization code

```
import PIL, Image, ImageDraw
width = 400;
height = 200;
im = Image.new("RGB", (width,height))
draw = ImageDraw.Draw(im)
peg_area_width = width / len(S['peg']) # notice we can access S directly
    here. The len function will allow us to get the number of pegs in our
    state variable S['peg']
post_width = peg_area_width * 0.1
post_top_y = height / 5
post_bottom_y = height * 4 / 5
disk_height = ((height*2/5) / D['numberOfDisks']) - 5; # Using Common
    Data's 'numberOfDisks' to calculate a variable height for the disks
    based on how many disks there are
peg_color = (0,0,255) # this is RBG, so pegs will be blue
disk_color = (255,0,0) # disks are red
for peg in range(len(S['peg'])): # and also accessing S here
location_x = peg * peg_area_width
topLeftX = location_x + (peg_area_width * 0.45)
bottomRightX = topLeftX + post_width;
draw.rectangle((topLeftX,post_top_y,bottomRightX,post_bottom_y),
fill=peg_color) # post
draw.rectangle((location_x+5,
```

```
post_bottom_y,
location_x+peg_area_width-5,
post_bottom_y+post_width),
fill=peg_color) # base
# Now draw disks, from bottom up
bottom_y = post_bottom_y
for disk in S['peg'][peg]: # Here we access S again, this time to get the
    disks from each peg
disk_width = (disk * (peg_area_width-10) / D['numberOfDisks'])
x1 = (location_x + (peg_area_width - disk_width) / 2)
y1 = bottom_y-(disk_height*(D['numberOfDisks']-disk)) - 2
x2 = x1 + disk_width
y2 = y1-disk_height + 2
draw.rectangle(( x1,y1,x2,y2 ),
fill=disk_color)

IMG_HIGH = im
IMG_LOW = im
```

A.3 Operator Code

The following shows the name, description, precondition code, and state transformation code, for each of the operators in the Towers of Hanoi problem template.

Name: Move the top disk on the first post to the third post.

Description: Move disk from peg one to peg three.

Listing A.5: Precondition "Move the top disk on the first post to the third post."

```
PRECONDITION = isValidMove(0,2) # this function is defined in  Common Code
```

Listing A.6: State Transformation "Move the top disk on the first post to the third post."

```
move(0,2) # this function is defined in Common Code
```

Name: Move the top disk on the second post to the first post.

Description: Move disk from peg two to peg one.

Listing A.7: Precondition "Move the top disk on the second post to the first post."

```
1 PRECONDITION = isValidMove(1,0)
```

Listing A.8: State Transformation "Move the top disk on the second post to the first post."

```
1 move(1,0)
```

Name: Move the top disk on the third post to the second post.

Description: Move disk from peg three to peg two.

Listing A.9: Precondition "Move the top disk on the third post to the second post."

```
1 PRECONDITION = isValidMove(2,1)
```

Listing A.10: State Transformation "Move the top disk on the third post to the second post."

```
1 move(2,1)
```

Name: Move the top disk on the third post to the first post.

Description: Move disk from peg one to peg three.

Listing A.11: Precondition "Move the top disk on the third post to the first post."

```
1 PRECONDITION = isValidMove(2,1)
```

Listing A.12: State Transformation "Move the top disk on the third post to the first post."

```
1 move(2,1)
```

Name: Move the top disk on the second post to the third post.

Description: Move disk from peg one to peg three.

Listing A.13: Precondition "Move the top disk on the second post to the third post."

```
1 PRECONDITION = isValidMove(1,2)
```

Listing A.14: State Transformation "Move the top disk on the second post to the third post."

```
1 move(1,2)
```

Name: Move the top disk on the first post to the second post.

Description: Move disk from peg one to peg three.

Listing A.15: Precondition "Move the top disk on the first post to the second post."

```
1 PRECONDITION = isValidMove(0,1)
```

Listing A.16: State Transformation "Move the top disk on the first post to the second post."

```
1 move(0,1)
```

Appendix B

CITYSIM USER STUDY: BACKGROUND QUESTIONNAIRE

Background Questionnaire Participant ID _____

	Please indicate how much experience you have with the following:	***Never***	***A little / Tried 1-2 times***	***Some experience***	***Fairly experienced***	***Pro / Very Experienced***
1	Using computers for common tasks, like checking your email, surfing the web, installing programs, etc.	1	2	3	4	5
2	Using design software such as Photoshop for images, DreamWeaver for web pages, Flash for animations or similar software? Include any software for creating music, editing movies, etc.; anything you consider design *In the questions below, list each software and mark your experience with it. Use the back for extra space as needed.*	1	2	3	4	5
3	Design Software:	1	2	3	4	5
4	Design Software:	1	2	3	4	5
5	Studying/using Graph theory (nodes, vertices, edges, breadth-first search, etc.)	1	2	3	4	5
6	Studying/using State-space Search (AI technique)	1	2	3	4	5
7	Playing ***single-player*** computer or video games	1	2	3	4	5
8	Playing ***multi-player*** computer or video games	1	2	3	4	5
9	Playing games in the SimCity series, or any type of city-building simulation game? *If you have experience with such games, please list which ones:*	1	2	3	4	5
10	Participating in online social networks, Twitter, Facebook, etc. *Approx hours per week spent on social networks?* ____	1	2	3	4	5
11	**Reading** online bulletin/message boards, forums, mailing lists, and other group discussion forums, such as UW Catalyst's GoPost	1	2	3	4	5
12	**Posting to** in online bulletin/message boards, forums, mailing lists, and other group discussion forums**,** such as UW Catalyst's GoPost	1	2	3	4	5
13	Using online collaboration tools (collaboration features in Google Docs, etc.), or contributing to online collaborative projects (open source software, Wikipedia, etc.) *If you have experience with them, please list which ones:*	1	2	3	4	5

Background Questionnaire

Participant ID _____

	Please indicate how strongly you agree with the following statements:	***Strongly Disagree***	***Disagree***	***Neutral***	***Agree***	***Strongly Agree***
1	I enjoy working in a group or team.	1	2	3	4	5
2	On most projects, I work better if I am in a group, rather than alone.	1	2	3	4	5
3	Given a choice, I would prefer to work on projects in a group, rather than on my own.	1	2	3	4	5
4	When working on a group project, I often feel free to share my opinions.	1	2	3	4	5
5	In group project ***discussions***, it is important to me that everyone feels they have a chance to participate in the discussion.	1	2	3	4	5
6	When trying to complete a group project, it is more important to me that everyone has a chance to share their opinion, rather than others agreeing with my opinion.	1	2	3	4	5
7	When working in a group, I often feel like I do more work than my group members.	1	2	3	4	5
8	I am often hesitant or reluctant to participate or share my thoughts in ***face-to-face*** group discussions.	1	2	3	4	5
9	I am often hesitant or reluctant to participate or share my thoughts in ***online*** group discussions. (check if you have never participated in an online group discussion, i.e. “n/a” [_])	1	2	3	4	5
10	In ***face-to-face*** group discussions, I express my opinion equally as often as other group members.	1	2	3	4	5
11	In ***online*** group discussions, I express my opinion equally as often as other group members. (Check if n/a [_])	1	2	3	4	5
12	In ***face-to-face*** group discussions, I often wish I could participate more.	1	2	3	4	5
13	In ***online*** group discussions, I often wish I could participate more. (Check if n/a [_])	1	2	3	4	5
14	When discussing a project ***face-to-face***, I am sometimes uncertain how to best participate.	1	2	3	4	5
15	When discussing a project **online**, I am sometimes uncertain how to best participate. (check if n/a [_])	1	2	3	4	5
16	I believe working in a group results in a better solution than working alone.	1	2	3	4	5

Do you consider yourself: [_] Female [_] Male Your age in years: _____

If you are a student…

…what year in school are you (Autumn 2012)? Grad / Undergrad, Year: ____

…what are you studying? _____, OR undecided

If you are not a student but are/were employed in some way, what is your current (or last) job/career position?

_____

Appendix C

CITYSIM USER STUDY: PRETEST/POST-TEST

CoSolve Test

Participant ID_____

Is this the pretest or the posttest? _____

Instructions

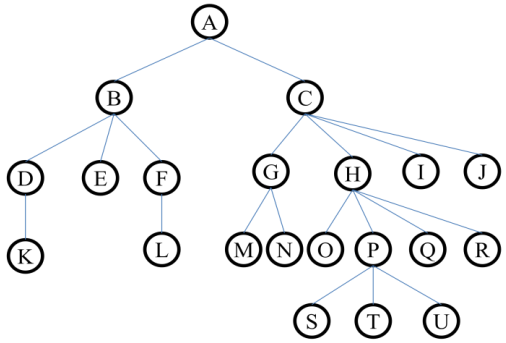
Match each item in the left column below with the best description of it in the context of collaborative problem-solving using a tool like CoSolve. The descriptions are found in the right column; write the description's letter (a, b, c, etc.) in the blank in the left column. If you don't know, it's fine to leave an item blank. Not all of the descriptions will be used.

Term	Description
1. ____ State	a. The node containing the initial state of the problem.
2. ____ Operator	b. A message sent to a user from a CoSolve administrator.
3. ____ Parent	c. Junction in a solving tree representing a moment in the problem-solving process–whether unsolved, partially solved or fully solved.
4. ____ Root	d. A message created by a user and placed on a CoSolve node.
5. ____ Annotation	e. Can create a new node from a previous node.
6. ____ Level	f. All nodes at the same depth.
	g. The result of a user error.
	h. A node to which an operator has been applied.
	i. A unique solution to a problem.
	j. The deletion of a node.

CoSolve Test

Participant ID_____

The next few questions refer to the following graph. Refer to the node by the letters printed inside them.



Question 8. What is the root of this tree?

Question 9. How many children does 'H' have?

Question 10. If 'S' is the highest scoring node, what is the solution path for this tree?

Question 11. What is 'F's parent?

Question 12. List 'G's children.

Question 13. What is the depth of node 'R', assuming the root's depth is 0?

Question 14. Do you have any other comments regarding your answers?

Appendix D

CITYSIM USER STUDY: TUTORIAL SCRIPT

"Before we start, I'd like to remind you that this is a research prototype, so it is still buggy, and quite slow. You will experience some lag on the system. Please try your best to complete the activity in spite of it.

1) Researcher1 opens new, blank citysim solving session. DO NOT let subjects open their session yet.

"This rectangular box here represents the beginning **STATE** of our city.

We see a grid of the city layout in the top half, and some information in the bottom half.

Nothing is in our city so far, and we see that we have $X money right now, in the bottom left, and that we have several types of buildings we can place, and they each have a cost. We'll discuss what they each do in a moment, but let's just try it out.

What building should we try first, **P1**? We are going to place the building at 8,1 just off of the upper right"

2) Researcher demonstrates adding the new building to the grid:

"To add a building, we mouseover the state, then click 'operators', then select the building we want. Then we need only click on the grid to place the building.

Let's place this ___ building here."

[while building is being created]

3) teach them TERMINOLOGY

"What we just did was to apply an **OPERATOR** to our initial state. Operators transform a state into a different **STATE**. "

[building done]

"Now we see that instead of just adding the building onto the grid in this box, CoSolve is display a totally new state underneath it.

*We can also call these states **NODES**, because, as you can see, they are connected together by a line.*

*We call this original node the **PARENT**, and the new node created is called the **CHILD.**"*

4) Get users to log in

"Okay, now you'll have a chance to try it out also.

Log into CoSolve with the information given, and go to this webpage URL. Researcher2 will help you.

Raise your hand when you have gotten there."

[wait for everyone to be done]

5) Get users to add nodes.

*"Now, **P2**, I'd like you to apply an operator to the child we just created to create another child.*
Please put a Commercial building at 1,1.
*Everyone else, please look at this gray box on the left. This is the **Consultant** tool. The first line says "View is up to date" and shows a green light. After **P2** applies the operator, you will notice it change. Let's watch it change"*

[wait for NEW NODE indicator]

"*Okay, the Consultant tool says there's been a new node created.*

So let's all click this Refresh button and now we'll see the new node.

*Raise your hand when you see the new node **P2** created appear."*

[refresh the screen, wait for finish]

*"Now, **P3**, I want you to add another commercial building, anywhere you like, and we will all wait to see it and refresh our screen."*

6)concept of money

*"Uh-oh, now we have an error in the new **CHILD** state that was created! We can't afford this building! Why not, **P3**?"*

[Wait for answer]

*"The answer is, we can't afford $10 to buy a Commercial building, since we only had $n money in the **PARENT** state that we created this error **CHILD NODE** off of.*

I can pan by dragging to see the parent again."

[Demonstrate PANNING to see the Parent State!]

7) Annotations

"Now, we want to make a note for ourselves that this node failed because we ran out of money. We can make notes about nodes through Annotations.

P1, could you scroll over the error node to get the Add Annotations menu, like this,

and then write "Tried to add commercial building." as the title and "Could not afford it." as the body, then click "thumbs-down" and add the annotation?

The Thumbs Down will tell us that this node is something we don't want to do."

"As you can see, the annotation appeared in the bottom left, and when you click on it, it will bring you to the node.

So if you find a good node while playing the game, you can thumbs-down, or thumbs-up, it to make note of it for yourself and everyone else.

If you thumbs-down a node, you are saying that you don't think it's on the path to a good solution.

Also, the three of you are in the same room, but this study is actually meant to simulate interaction taking place by people in different locations and possibly at different times, and that's why we'd like to ask you guys to communicate only through the interface rather than actually talking out loud. "

8) demonstrate backtracking.

*"So to correct our error, we need to **BACKTRACK**, and buy a cheaper building instead, say a Residential Building.*

*Now normally, when you are playing a game, you only see the **CURRENT STATE** of your game.*

But in CoSolve, every single move you make, whether it's a good move OR an error like we just saw, is saved.

You cannot delete anything. This has the advantage of allowing you to go back and try something again.

*So let's go back to the **PARENT** state of the error node, and then create a new **CHILD** by applying the 'Place Residential Building' **OPERATOR**.*

***P2,** can you try creating the new **CHILD** off of this **PARENT** by placing the Residential building at 8,8?"*

[wait for it]

“Great, thanks!”

5) Demonstrate CitySim strategy

"Now let's talk about the CitySim game itself. We've been placing buildings without knowing what they do.

Each building has an effect, and we can see these effects by using the State View feature.

*You can go to the menu here and select one of the state views. Currently we are in **BULDING** view because we can see the buildings.*

Let's look at the HAPPINESS view on the state P1 just created."

[demonstrate this]

"People like shopping and eating so the happiness of people is higher in the grid cells at and around the commercial building. Commercial buildings bring up happiness by 1.

*In **JOB** VIEW, we see they also provide jobs in the cells around the Commercial building, because people work in the shops.*

*If we go to the **POPULATION** View at the bottom, we'll see that there's population in the area around where P1 just put the Residential Building.*

*However, when we go to **EMPLOYMENT** view, there's nothing! So we have jobs and we have population, but since they don't overlap, these people are unemployed.*

Since they are unemployed, they can't pay taxes to our city, and so our city doesn't gain any income, the income value is listed as 0 here.

We need to try a different strategy."

6) ROOT/RESET/ZOOM

"Let's start over from the beginning. In CoSolve, this is easy to do because all your steps are saved.

*To go back to the empty state at the beginning, which is what we call the **ROOT** node, click on the **RESET** button in the upper left.*

Everyone do this now and raise your hand when finished."

[wait for hands]

"Here next to RESET, you can also ZOOM in and zoom out using these controls as well, the + and - in the upper left. If you zoom out, you'll see

*that you have something that's shaped like an upside down tree. We call this a **PROBLEM-SOLVING TREE.***

*Now after you hit the RESET button, you should see the **ROOT** node.*

*If we want to start over, we'll just create another **CHILD** from this **ROOT** node, rather than trying to delete something.*

*So now, **P3**, can you place a **residential** building somewhere on the grid starting from the **ROOT** node?*

*Then **P1**, after you see this new node, add an **Industrial** Building right next to it.*

*Finally, P2, tell us what the money and income values for the newest **CHILD** state are after this.*

I will be watching from my computer screen here."

[wait for them to finish, watch them]

"Okay, thank you P3. That's right the money and the income values are $17 and $12.

Let's see why the income is $12.

*We'll look at our **state** view to see **Jobs** are 2 around the Industrial building, and **Population** is 1 around the Residential building, and where they overlap is 1 because the 1 person there is being employed in one of the 2 jobs in that grid cell. All together, 6 people are employed, as can be seen in the **Employment** view.*

Now let's look at our TAX RATE. It is $2, so each of those 6 employed people pay $2, so your city will gain $12 on the next turn.

This is definitely on the right track, we finally have income, so I want to make note of this node.

***P1**, mouse over this node, go to 'Add Annotation' like this, and tell us that this is a good node by making a comment and clicking the **Thumbs Up** button and add the annotation.*

*Since we are thumbs-up-ing this node, we are saying that we think this node is on the **SOLUTION PATH**. Let me zoom out so that you can see the **problem-solving tree.**"*

[Zoom Out]

*"See, we have this path that was annotated negatively, then this one that is annotated positively, to show whether we think the node is on the **SOLUTION PATH.** When we annotate, we don't have to be sure that a node is on the solution path, we just want to note what our current thinking of the node is. We can add other annotations onto it later if we find that it's a good path or a bad path.***"**

"Now say we want to employ even more people, by building an Industrial Building on the other side to employ the unemployed people."

[apply Industrial to other side of Residential]

"Now we see that our money is now $28 because we had $17 and we bought an industrial building for $1, but we received an income of $12 from the last turn. So $28 = $17 - 1 + 12.

*However, look at our income! It's still the same even though we should have more people employed now. Why is this? Well, in the **Jobs** view, we definitely have more jobs...I'm going to **ZOOM** out so that you can see both at once"*

[show Jobs view]

"the 4s in the center are from overlapping jobs, so there are 4 jobs in those squares, and the RESIDENTIAL view is the same, but the employment is different! Those squares in the middle no longer have employment, though the squares around it do, which is why we still have an income of 12. Why is this?

*Well, let's look at the **happiness** view. The happiness in those squares is -2. In CitySim, when the happiness of a neighborhood block is -2 or less, people are unhappy and leave their jobs, so you don't get any income from them! Industrial Buildings, bring jobs but also causes unhappiness because people don't like the factory pollution and such. So this is not such a promising state. P2, can you annotate this with a thumbs down? "*

[wait for p2 to do this]

"also, if you can't see an annotation, you can also click in the upper left annotation button of each node, here. "

*[*show them how to toggle annotations]

"Let's try to increase happiness by placing a theater building at 4,5. This increases surrounding happiness by 2, so now we have more people employed."

6) FINISH

"So you should now know enough to get started with the game, but there are a few more building types that you will need to know. You can click in the Consultant under Help Pages to see what each building does."

[show help pages]

" (The theater increases happiness around it, so you could use it to counter the Industrial building in the example here).

The Tax Bureau and Dept of Roads are kind of unusual.

Tax Bureau increases your tax rate so you collect more taxes from each employed square, but you decrease happiness for the ENTIRE board, rather than just nearby.

The Dept of Roads expands the area of influence of all the other buildings in the city, so instead of a 9x9 area around each building, you have a 5x5 area around each building. If you build another Dept of Roads, you'll have a 7x7 area, then a 9x9 area, etc.

However, the cost of the Dept of Roads will increase each time you build it, so you can't afford to keep building them. It is very important so you should probably think about including it in your city, hint hint."

7) DONE, TALK ABOUT CONTROL/EXPERIMENTAL CONDITIONS (The Consultant Slides, etc.) & REWARD

Appendix E

CITYSIM USER STUDY: WRAP-UP QUESTIONNAIRE

Participant ID: _____

Wrap-up Questionnaire

Part A. For each of the statements in the boxes below, select on a scale of 1-5 how much you agree or disagree with that statement. Here are some descriptions of terms for clarification:

- *The Citysim Game* refers to just the city building puzzle game itself, independent of CoSolve.
- *The CoSolve Consultant* refers to the toolbox shown below the magnification buttons on the left of your CoSolve solving session; it exists to provide information to problem solvers, and includes node highlighting, but does not include the Annotation List.
- *The CoSolve Interface* refers to the interactions for solving a problem in CoSolve, including panning and zooming controls and interactions for applying operators and annotations.
- *The tree visualization of the problem-solving space* refers to how each state in the process is shown as a node, connected to a parent and possibly children. This does ***not*** include the CoSolve Consultant and could pertain to games or problems ***other*** than the CitySim Game.

	Statement	***Strongly Disagree***	***Disagree***	***Neutral***	***Agree***	***Strongly Agree***
1	*The tree visualization of the problem-solving space* was confusing	1	2	3	4	5
2	*The tree visualization of the problem-solving space* was helpful	1	2	3	4	5
3	*The tree visualization of the problem-solving space* was difficult to navigate	1	2	3	4	5
4	*The tree visualization of the problem-solving space* helped encourage collaboration	1	2	3	4	5
5	Regarding navigation of the tree, I was often lost or disoriented	1	2	3	4	5
6	I fully understand the rules of *the CitySim Game*	1	2	3	4	5
7	I often felt overwhelmed with the number of choices at any given point.	1	2	3	4	5
8	It was difficult to do well in *the CitySim Game*	1	2	3	4	5
9	I believe that my team-mates understood *the CitySim Game* better than I did	1	2	3	4	5
10	*The CitySim Game* was fun	1	2	3	4	5

Participant ID: _____

11	*The CoSolve Interface* was difficult to use	1	2	3	4	5
12	*The CoSolve Interface* was helpful in playing the game	1	2	3	4	5
13	*The CoSolve Interface* was easy to learn	1	2	3	4	5
14	*The CoSolve interface* made collaboration more difficult	1	2	3	4	5
15	*The CoSolve interface* was helpful in understanding my team's problem-solving process	1	2	3	4	5
16	My teammates and I each did an equal portion of the work	1	2	3	4	5
17	I felt like I was competing with members of my team	1	2	3	4	5
18	I was always well aware of what my teammates had done	1	2	3	4	5
19	It was easy to collaborate with my teammates.	1	2	3	4	5
20	I was aware of what my team-mates were doing most of the time.	1	2	3	4	5
21	I had a difficult time keeping up with my team-mates.	1	2	3	4	5
22	I wish my team-mates would have collaborated more.	1	2	3	4	5
23	*The CoSolve Consultant* was difficult to use	1	2	3	4	5
24	*The CoSolve Consultant* was helpful in playing the game	1	2	3	4	5
25	*The CoSolve Consultant* was difficult to learn	1	2	3	4	5
26	*The CoSolve Consultant* was helpful in understanding my team's problem-solving process	1	2	3	4	5
27	*The CoSolve Consultant* helped my team collaborate	1	2	3	4	5
28	The highlighting features of *the CoSolve Consultant* were not useful	1	2	3	4	5
29	I would have liked more time in which to complete the CitySim activity	1	2	3	4	5
30	I needed less time than I was given to complete the CitySim activity	1	2	3	4	5
31	My team-members and I communicated effectively regarding our design process	1	2	3	4	5
32	I have a better understanding of my own problem-solving process now than before playing the game	1	2	3	4	5

Participant ID: _____

33	I enjoy working in a group or team.	1	2	3	4	5
34	On most projects, I work better if I am in a group, rather than alone.	1	2	3	4	5
35	Given a choice, I would prefer to work on projects in a group, rather than on my own.	1	2	3	4	5
36	When working on a group project, I often feel free to share my opinions.	1	2	3	4	5
37	In group project discussions, it is important to me that everyone feels they have a chance to participate in the discussion.	1	2	3	4	5
38	When trying to complete a group project, it is more important to me that everyone has a chance to share their opinion, rather than others agreeing with my opinion.	1	2	3	4	5
39	When working in a group, I often feel like I do more work than my group members.	1	2	3	4	5
40	I am often hesitant or reluctant to participate or share my thoughts in online group discussions.	1	2	3	4	5
41	In online group discussions, I express my opinion equally as often as other group members.	1	2	3	4	5
42	In online group discussions, I often wish I could participate more.	1	2	3	4	5
43	When discussing a project online, I am sometimes uncertain how to best participate.	1	2	3	4	5
44	I believe working in a group results in a better solution than working alone.	1	2	3	4	5
45	My attitude towards working in a group has changed during the course of this activity.	1	2	3	4	5

Part B.
For the following items, please write a sentence or two giving your answer and an explanation.

1. For this study you created a solution for a SimCity like game. Describe the game: what the player does, what the goal is, etc.

2. If you were to start a similar exercise from scratch, would you prefer to work alone or with a team (say, of randomly chosen participants)? Explain.

Participant ID: _____

3. Are you satisfied with the quality of the interaction between yourself and your team-mates? Why or why not? What would you change?

4. Describe your team's collaborative process.

5. Roughly what percentage of the nodes created were yours, when compared to those of your team-mates? Explain.

6. Roughly what percentage of the annotations created were yours, when compared to those of your team-mates? Explain.

7. Roughly what percentage of the overall work (ideas, experimentation, time-spent, nodes/annotations created, etc.) was your contribution, compared to those of your team-mates? Explain.

8. If you had performed this activity by yourself, without any team-mates, do you think your score would have been higher, lower or the same? If higher or lower, how much? Explain.

9. If your team could have communicated verbally, do you think your score would have been higher, lower or the same? If higher or lower, how much? Explain.

10. Did this process help you learn about your problem-solving style? If so, what/how?

11. Did this process help you learn about your collaboration style? If so, what/how?

Appendix F

ROLES USER STUDY: WRAP-UP QUESTIONNAIRE

Participant ID: _____

Wrap-up Questionnaire

Part A. For each of the statements in the boxes below, select on a scale of 1-5 how much you agree or disagree with that statement. Here are some descriptions of terms for clarification:

- *The Citysim Game* refers to just the city building puzzle game itself, independent of CoSolve.
- *The Roles Interface* refers to the box shown in the bottom left above the annotations box.
- *The Roles System* refers to the general roles system (phases, goals, timing, etc.)
- *The CoSolve Interface* refers to the interactions for solving a problem in CoSolve, including panning and zooming controls and interactions for applying operators and annotations.
- *The tree visualization of the problem-solving space* refers to how each state in the process is shown as a node, connected to a parent and possibly children. This does ***not*** include the CoSolve Consultant and could pertain to games or problems ***other*** than the CitySim Game.

	Statement	***Strongly Disagree***	***Disagree***	***Neutral***	***Agree***	***Strongly Agree***
1	*The tree visualization of the problem-solving space* was confusing	1	2	3	4	5
2	*The tree visualization of the problem-solving space* was helpful	1	2	3	4	5
3	*The tree visualization of the problem-solving space* was difficult to navigate	1	2	3	4	5
4	*The tree visualization of the problem-solving space* helped encourage collaboration	1	2	3	4	5
5	Regarding navigation of the tree, I was often lost or disoriented	1	2	3	4	5
6	I fully understand the rules of *the CitySim Game*	1	2	3	4	5
7	I often felt overwhelmed with the number of choices at any given point.	1	2	3	4	5
8	It was difficult to do well in *the CitySim Game*	1	2	3	4	5
9	I believe that my team-mates understood *the CitySim Game* better than I did	1	2	3	4	5
10	*The CitySim Game* was fun	1	2	3	4	5
11	*The CoSolve Interface* was difficult to use	1	2	3	4	5
12	*The CoSolve Interface* was helpful in playing the game	1	2	3	4	5

Participant ID: _____

13	*The CoSolve Interface* was easy to learn	1	2	3	4	5
14	*The CoSolve interface* made collaboration more difficult	1	2	3	4	5
15	*The CoSolve interface* was helpful in understanding my team's problem-solving process	1	2	3	4	5
16	My teammates and I each did an equal portion of the work	1	2	3	4	5
17	I felt like I was competing with members of my team	1	2	3	4	5
18	I was always well aware of what my teammates had done	1	2	3	4	5
19	It was easy to collaborate with my teammates.	1	2	3	4	5
20	I was aware of what my team-mates were doing most of the time.	1	2	3	4	5
21	I had a difficult time keeping up with my team-mates.	1	2	3	4	5
22	I wish my team-mates would have collaborated more.	1	2	3	4	5
23	*The Roles Interface* was difficult to use	1	2	3	4	5
24	*The Roles Interface* was helpful in playing the game	1	2	3	4	5
25	*The Roles Interface* was difficult to learn	1	2	3	4	5
26	*The Roles Interface* was helpful in understanding my team's problem-solving process	1	2	3	4	5
27	*The Roles Interface* helped my team collaborate	1	2	3	4	5
28	*The Roles System* was difficult to understand	1	2	3	4	5
29	*The Roles System* was helpful in playing the game	1	2	3	4	5
30	*The Roles System* was difficult to learn	1	2	3	4	5
31	*The Roles System* was helpful in understanding my team's problem-solving process	1	2	3	4	5
32	*The Roles System* helped my team collaborate	1	2	3	4	5
33	The highlighting features of *the CoSolve Interface* were not useful	1	2	3	4	5
34	I would have liked more time in which to complete the CitySim activity	1	2	3	4	5

Participant ID: _____

35	I needed less time than I was given to complete the CitySim activity	1	2	3	4	5
36	My team-members and I communicated effectively regarding our design process	1	2	3	4	5
37	I have a better understanding of my own problem-solving process now than before playing the game	1	2	3	4	5
38	I enjoy working in a group or team.	1	2	3	4	5
39	On most projects, I work better if I am in a group, rather than alone.	1	2	3	4	5
40	Given a choice, I would prefer to work on projects in a group, rather than on my own.	1	2	3	4	5
41	When working on a group project, I often feel free to share my opinions.	1	2	3	4	5
42	In group project discussions, it is important to me that everyone feels they have a chance to participate in the discussion.	1	2	3	4	5
43	When trying to complete a group project, it is more important to me that everyone has a chance to share their opinion, rather than others agreeing with my opinion.	1	2	3	4	5
44	When working in a group, I often feel like I do more work than my group members.	1	2	3	4	5
45	I am often hesitant or reluctant to participate or share my thoughts in online group discussions.	1	2	3	4	5
46	In online group discussions, I express my opinion equally as often as other group members.	1	2	3	4	5
47	In online group discussions, I often wish I could participate more.	1	2	3	4	5
48	When discussing a project online, I am sometimes uncertain how to best participate.	1	2	3	4	5
49	I believe working in a group results in a better solution than working alone.	1	2	3	4	5
50	My attitude towards working in a group has changed during the course of this activity.	1	2	3	4	5

Part B.

For the following items, please write a sentence or two giving your answer and an explanation.

1. For this study you created a solution for a SimCity like game. Describe the game: what the player does, what the goal is, etc.

Participant ID: _____

2. If you were to start a similar exercise from scratch, would you prefer to work alone or with a team (say, of randomly chosen participants)? Explain.

3. Are you satisfied with the quality of the interaction between yourself and your team-mates? Why or why not? What would you change?

4. Describe your team's collaborative process.

5. Roughly what percentage of the nodes created were yours, when compared to those of your team-mates? Explain.

6. Roughly what percentage of the annotations created were yours, when compared to those of your team-mates? Explain.

7. Roughly what percentage of the overall work (ideas, experimentation, time-spent, nodes/annotations created, etc.) was your contribution, compared to those of your team-mates? Explain.

8. If you had performed this activity by yourself, without any team-mates, do you think your score would have been higher, lower or the same? If higher or lower, how much? Explain.

9. If your team could have communicated verbally, do you think your score would have been higher, lower or the same? If higher or lower, how much? Explain.

10. Favorite Role? Least Favorite? Why?

11. Did this process help you learn about your collaboration style? If so, what/how?

Appendix G

CITYSIM USER STUDY: INTERVIEW QUESTIONS

Interview Questions

1. [get them to guide you through their tree] Walk through your tree and describe the design process to us. Describe what the 'story' is behind each part of the tree; what was your thought process when building this branch?
   **For 4 sections (branches or general regions) of the tree, try to get the following answers. Do for 2 sections created by this user, and 2 by others
      **What the thinking/story behind this part of the tree was
      **Whether it worked out or not; why or why not
      **What was learned
   {CitySim User Study Questions, Roles Study optional:}
2. If you were to start a similar exercise from scratch – a similar game playing exercise – how would you go about it this time? Assume you were working with the same team members as before.
3. Were you happy with the solution you and your team came up with? Is there more you would have liked to have done?
4. What criteria did you use to evaluate your team's progress? Specifically, 1) what your team had done and 2) what needed to be done?
5. Describe your various reasons for creating new nodes; for instance: to get a higher score, or to experiment with a new technique.
6. Describe your various reasons for creating new annotations; for instance, as a reminder to yourself, or as a message for a team-mate.
7. Describe how your group members communicated with each other.
   {Roles User Study Questions ONLY:}
8. Were you usually aware of what role you were in?
9. How about what role your teammates were in?
10. Did your roles affect your behavior? How so? For example, did you spend most of your time thinking about / trying to complete your role, or did you mostly ignore it? Did you try to complete it quickly, or did you spread it out over the role phase?
11. What was your strategy, thoughts, or feelings when you were in each of the following roles:
    a. Brainstormer
    b. Critic
    c. Supporter
12. What about when your teammates were in each of those roles? How do you think it affected their behavior, and your attitude towards their behavior?
13. How would you have behaved differently without a roles system?
14. Thoughts on the Current Role tab? How often did you use it?
15. Thoughts on the Overall Role tab? How often did you use it?
    {For both user studies:}
16. What parts of the flash program were helpful in solving the problem? List the 3 most helpful aspects and why?
17. What parts of the flash program were harmful or distracting in solving the problem? List the 3 most harmful or distracting aspects and why?
18. What additional features can you think of that would make problem-solving in CoSolve easier/better?
19. What additional features can you think of that would make collaboration in CoSolve easier/better?

VITA

Sandra B. Fan (sbfan@cs.uw.edu) received her B.A. in Information and Computer Science from the University of Hawaii at Mānoa in 2002. She received her M.S. in Computer Science and Engineering from the University of Washington in 2006. She worked as a software engineer at Healia, a Bellevue, WA-based consumer health internet company, from 2007 to 2008 before returning to the UW and completing her PhD in 2013.