Enhancing State-Space Tree Diagrams for Collaborative Problem Solving

Steven L. Tanimoto

University of Washington, Seattle WA 98195, USA,
tanimoto@cs.washington.edu,
WWW home page: http://www.cs.washington.edu/homes/tanimoto/

Abstract. State-space search methods in problem solving have often been illustrated using tree diagrams. We explore a set of issues related to coordination in collaborative problem solving and design, and we present a variety of interactive features for state-space search trees intended to facilitate such activity. Issues include how to show provenance of decisions, how to combine work and views produced separately, and how to represent work performed by computer agents. Some of the features have been implemented in a kit called TStar and a design tool called PRIME Designer.

1 Introduction

1.1 Motivation

Problem solving and design processes tend to confront more and more complex challenges each year. A solution to a single problem often requires expertise from several domains. For example, Boston's "Big Dig" required expertise from civil engineers, geologists, urban planners, and politicians, among many others.

A group at the University of Washington is studying the use of the classical theory of problem solving as a methodology in support of collaborative design. One part of the approach involves the use of interactive computer displays to anchor the shared experience of a team. This paper describes a set of issues and possible features to address those issues. The issues and features pertain to the use of interactive tree diagrams that show portions of a design space or solution space, and that support a team in managing their work.

1.2 State-Space Search Trees

Problem solving has been cast in the formal framework of "state space search" by early researchers in artificial intelligence, such as Newell and Simon (see Simon, 1969). A solver starts with an initial state, representing an absence of commitments to the particulars of a solution, and step by step, tries adding particular elements or modifying the current state, in an attempt to reach a "goal state". The goal state, or perhaps the sequence of steps taken to reach it, constitute a solution to the problem. The states visited in such a search can

usually be plotted as nodes of a graph. Each move from one state to another, in turn, corresponds to an edge of the graph. If we decide that each state is described by the sequence of operators applied, starting at the initial state, then the graph has the structure of a tree, and it can be drawn as a tree. Such diagrams have been used to illustrate textbooks of artificial intelligence, often in conjunction with classical puzzles such as the Eight puzzle (Nilsson, 1971) or the game of Tic-Tac-Toe as shown in Fig. 1. The use of state-space search trees in the design of image-processing procedures was discussed in the context of face-image analysis (Cinque et al, 2007) and that diagrammatic methodology is shown in Fig. 2.

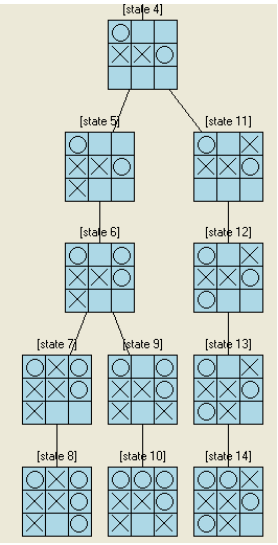

Fig. 1. A familiar kind of state-space search tree. This diagram corresponds to a partial search in the middle of a Tic-Tac-Toe game.

1.3 Issues

This paper is concerned with a set of issues that arise when one chooses to use state-space search tree diagrams as interactive representations of the progress of a design or problem-solving team. Some of these issues are the following.

Provenance. Each node represents a partial design or solution. Who contributed what to this design? From what other designs was it derived? Supports for querying and viewing provenance are needed.

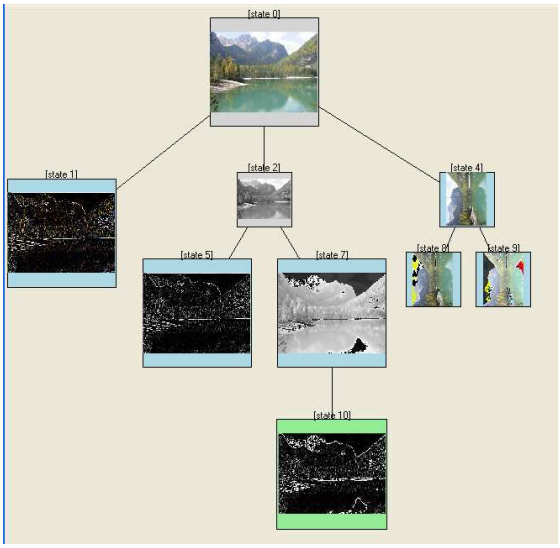

Fig. 2. Tree representing an exploration in a space of transformed images.

Negotiating Shared Views. A designer's personal view of a design tree reflects her priorities and preferences. In a shared view, priorities and preferences may collide, leading to the need for compromises. Algorithms can offer certain compromises automatically, but shared views that are most satisfactory to a group may require give and take. Facilities to manage such negotiation are needed.

Visualization of Opportunity. The members of a team may understand their own components of a design or solution, but it's often difficult to understand the potential interactions of the pieces. In some cases, separately designed pieces can be combined, and the diagram can indicate the compatibility of a piece for a particular role. Some opportunities result from the applicability of operators to newly computed states. Others result from the availability of resources in relation to defined tasks. Helping users see and evaluate opportunities is an important goal for collaborative design and problem-solving tools.

Work by Agents. A collaborative team traditionally consists of people. However, computer agents may be present and available to help out. Agents generally work very differently from people, and special affordances are required to define and display tasks for agents and the results of those tasks.

1.4 Previous Work

While much research has been done separately on problem solving, tree diagram layout, operations on trees, and interacting with trees, we mention in this section a mix of foundations and some work that bridges these topics.

Theory of Problem Solving. What we'll call the "classical theory of problem solving" was developed during the 1960s. An introduction can be found in Nilsson (1971). The important concepts for understanding the roles of the tree diagrams are presented below in section 2. The rationale for this somewhat reductionist view of design and problem solving was provided by Simon (1969).

Flexible Tree Layout. In order to display a large tree so that various parts of the tree can be distinguished, or so that a minimum or limited area is used, many methods have been proposed. A survey of tree display methods was presented by Hanrahan (2001). Popular methods include "hyperbolic trees" (Lamping et al, 1995), subdivision diagrams, and cluster diagrams. Usually, the goal is to display a large tree in such a way that individual nodes get their fair share of screen real estate.

Interactive Tree Display. A family of interactive techniques for trees has been developed in the context of operating-system file hierarchies. The best-known example of a hierarchical file system browser is the Microsoft Windows Explorer (not Internet Explorer). Interior nodes of the tree shown in Explorer correspond to folders in the file system, and the leaves correspond to individual files. The user can click on folder icons (situated at the nodes) to open and close the folders. When a folder that contains files is closed, it has a hotspot labeled with a plus (+) sign. Clicking on the plus sign opens the folder to reveal its contained files and subfolders. A more general set of affordances for hiding and revealing folders is described in a US patent (see Chu and Haynes, 2007). Here it is possible to have two nodes in a relationship where A is an ancestor of B, where A is closed, but B is visible and open. In such a view, all of A's descendants, except B and its descendants (and any other nodes explicitly opened) are hidden.

Interactive Layout. There have been several developments that have combined flexible layout with interactivity. A notable example is "Degree of Interest Trees" (Card and Nation, 2002). In such a tree, the user can indicate interest or disinterest in particular nodes of the tree by clicking on them, and the layout of the tree immediately adapts to the new distribution of interest, using a continuous animation to move from the current layout to the new layout.

2 Theoretical Background

Here we define the basic concepts on which our tree diagrams are based.

States and Moves. We assume that a problem, possibly a design problem, has been defined. We assume that a solution can be developed by starting with either a blank slate or an arbitrary configuration of elements (that we call the *initial state*) and taking a series of steps called *moves*. (The term "move" comes from the steps involved in solving puzzles and playing games like checkers.) Each move leads to a new state. A *state* represents a snapshot, a particular point of progress,

in the process of finding a solution. A state that satisfies the requirements for being a solution to the problem being solved is called a *goal state*. The set of all states that can, in principle, be reached from the initial state by making moves (possibly infinite sequences of moves)

Operators. A move comes about by applying a method called an *operator* to a given state. Any given operator may or may not be applicable to a particular state. For example, when designing a house, an operator to "connect the first and second floors with a staircase in the northwest corner of the house" is only applicable if both the first floor and second floor have already been created as parts of the design and no staircase already connects them in the northwest corner of the house. The applicability of an operator is determined by a predicate that is associated with it called its *precondition*. There are two kinds of operators: fixed operators and parameterized operators. A fixed operator doesn't require any additional specification in order to be applied. A parameterized operator requires more information in order to apply it. For example, an operator to "add a back deck" might require a (width, length) specification. The allowable parameters may depend on the state to which the operator is being applied.

Search Trees. In many puzzles, games, and real-world problem-solving situations, it is possible to arrive at a particular state via two or more different sequences of moves. If displayed nodes always corresponded one-to-one with states, the diagrams corresponding to such collections of states and moves would not necessarily be trees but directed graphs that might even contain cycles.

We will avoid dealing with cycles and multiple paths to the same state by defining a node not strictly in terms of an underlying state but in terms of the sequence of moves taken from the initial state. This ensures that the diagrams we work with will always be trees. Not only that, it respects the different provenances of nodes even when they represent equivalent states.

We define a *search tree* to be an acyclic, directed graph, with at least one node, the root, which corresponds to the initial state for a problem. Each node other than the root has a unique parent node from which it is derived by applying one operator (fixed or parameterized). The sequence of moves (or practically speaking, operators and any needed parameters) to get from the root to a node N identifies the node N. A search tree represents a portion of a state space. If the state space is finite and small, the search tree could possibly represent the whole space, but in general it will usually represent only a small fraction of the entire state space. It typically represents an "explored portion" of a state space.

The way that the tree is drawn can be modified to reflect the properties of nodes, operators, parameters, and to take into account viewing preferences and status of current activities.

Potential vs Realized Nodes. A search tree, representing only a portion of a state space, contains only "realized" nodes. The others are "potential nodes." A potential node corresponds to a sequence of operators that could be applied to the initial state to obtain a state for the node, but that has not yet been applied.

When a tree is grown, some potential nodes are converted into realized nodes by applying operators and thus making new moves.

A node schema corresponds to a set of potential and/or realized nodes all of which can be produced by a sequence of operators some of which may be parameterized operators whose parameters have not been specified. Changing any parameter of a parameterized operator in the sequence always changes the final node, and so the schema can correspond to a multiplicity of nodes. A schema can be considered a plan for exploration.

3 Application to Collaborative and Design

3.1 Rationale

We use the classical theory of problem solving outlined above for two reasons: (1) provide a common language for the problem solving process to design teams whose members represent different disciplines, and (2) help humans interact, via the computer interface, not only with each other, but with computational agents that perform design or problem-solving services.

3.2 Design Acts

For convenience, let's call our problem solving or design process simply "the design process," with the understanding that the methodology serves either purpose. We assume that the entire process of design can be broken down into small steps that we call *design acts*. There are several types of design acts, including communication acts, design steps, evaluation acts, and administrative acts.

Communication acts include writing and reading messages associated with the project. The messages may be annotations attached to a diagram, or they may be email or chat messages decoupled from precise design contexts.

Design steps are primarily applications of operators to existing nodes to produce new nodes in a search tree. Selecting a node or adjusting a selection, in order to apply an operator, may also be considered a design step.

An evaluation act is either a judgment or an application of an evaluation measure to a node or set of nodes. A judgment is a human-created quantitative or qualitative estimation of the value of a node on some scale with regard to some particular characteristic. An evaluation measure is a mathematical function that can be applied to a node to return a value (usually numeric). Such a value may correspond to the degree to which a partial design meets a particular criterion. For example, one evaluation measure for designs of houses is the number of rooms in the house. Another is the square footage of area in the designed house so far. A judgment might correspond to an aesthetic evaluation – how pleasing is this floor layout to the eye of designer Frank?

Administrative acts include actions such as the definition of tasks and subgoals for the design team, commitments of effort to tasks, and adjustments to views of the progress made so far. It may be difficult to separate administrative from communicative acts, since a commitment is usually accompanied by an expression of that commitment to others.

3.3 Roles of the Diagram

By using tree diagrams, it is possible to provide computer assistance, at some level, for each type of design act mentioned above. Many communication acts will relate to parts of a tree. Messages can refer to parts of a tree in two ways: (1) by naming a labeled node or by describing the path to it from the root, and (2) by being embedded in the node as an annotation. Messages can also refer to nodes via descriptions or expressions in a node-specification language. Such a language is a kind of query language with features for identifying nodes not only via their properties and annotations, but via their "kinship" relations.

Design steps are supported via interactive tree-building functionality. The selection of nodes and application of operators can be performed using a direct-manipulation style of construction. (An example of a design tree built with a tool called PRIME Designer is shown in Fig. 3).

Evaluation is supported via annotation and editing tools for judgments and via commands to apply built-in functions to sets of nodes.

Administrative acts such as task definition can often be expressed in terms of work to be done exploring different branches of a tree. Naming of subtrees or work associated with them can make use of node annotation facilities.

In addition to helping with specific design acts, the diagrams provide an easily browsable record of the history of the design process. They also provide contextual information for particular tasks. Thus a state-space search tree provides an organizing framework to a team for pursuit of solutions.

4 Layout and Visibility Affordances

State-space search trees have the potential to grow very large. In order to make large trees manageable on an interactive display, the following techniques are commonly used: scrolling, overall scaling, and opening/closing (revealing/hiding) of subtrees. We discuss variations on each of these techniques that can provide additional user control in the case of search trees.

4.1 Scrolling

Scrolling is the business of translating the viewport on the tree to locations of interest. It can be done manually by the user with scrollbars, or automatically by the program according to policies or special commands. Manual scrolling is well understood by users, but it can be difficult to navigate when the viewport is small relative to the entire diagram.

Automatic scrolling "by policy" means that the program changes the location of the viewport when certain kinds of events occur. When a new node is created as a result of a user's instruction, automatically scrolling to the new node can be helpful. This facilitates inspection of the new node, and it anticipates the likely next step in search as one that builds further in the same direction, i.e., from the new node (i.e., depth-first).

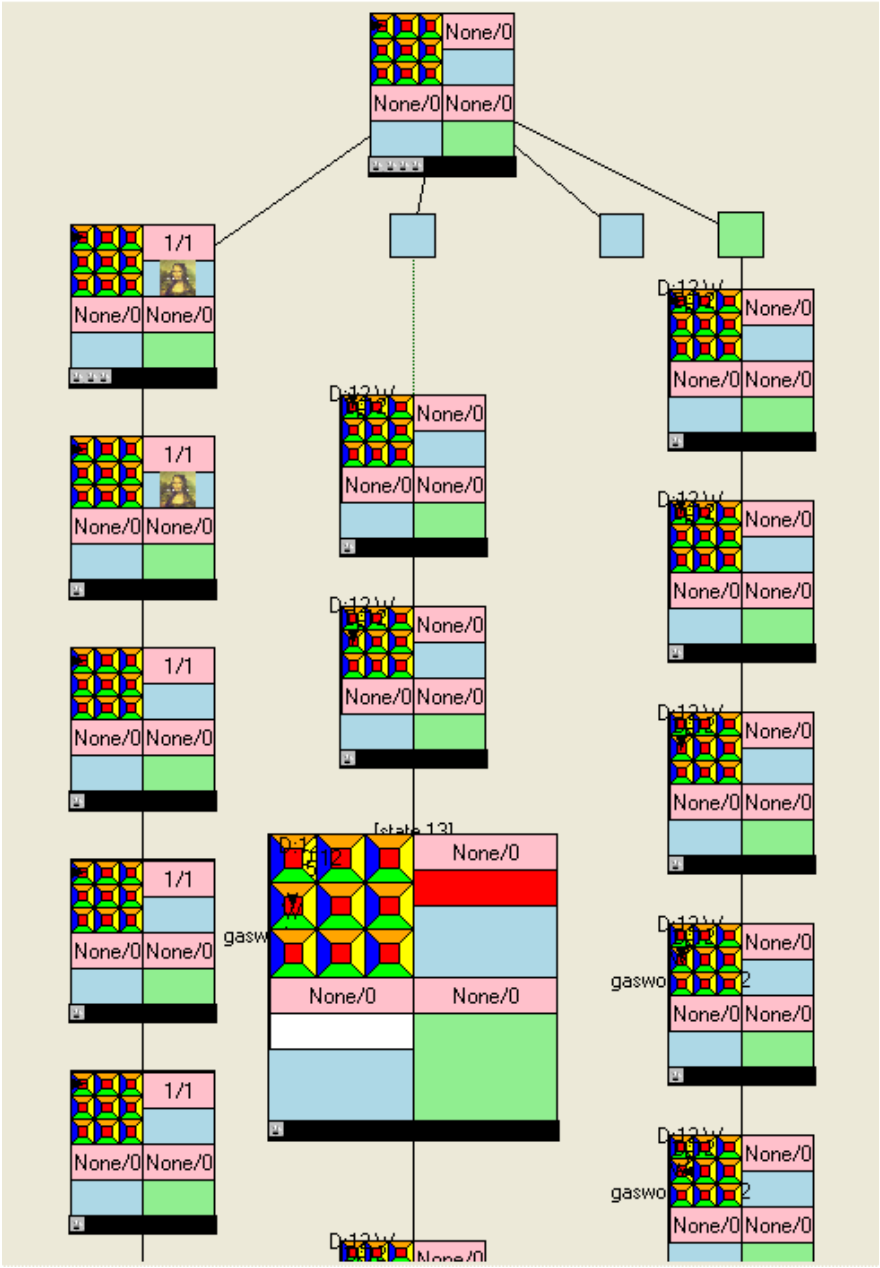

Fig. 3. Design tree showing multiple paths developed by separate members of a team. Also shown are "minified" nodes, an ellipsis under the first minified node, and an enlarged node. The four quadrants in each node correspond to different roles on the design team. This view is an "all-roles" view.

Automatic scrolling "by command" means that the program manages the adjustment of the viewport in order to obey a command from the user, such as "scroll to the root", or "scroll to the parent of node 113." Relative-motion commands can also be useful: "move to left sibling."

Scrolling is also assisted by birds-eye views, scrolling histories, and options to "snap to tree nodes" (analogous to snap-to-grid policies in drawing programs).

4.2 Scaling

Overall scaling, equivalent to zooming, is an essential viewing technique for large trees. A challenge for designers of visualizations is providing controls so that users can perform *differential scaling*, so that different parts of the tree can have different scale factors. A form of differential scaling can be found in dynamic tree layout methods such as hyperbolic trees (Lamping et al, 1995) and "degree-of-interest" trees (Card and Nation, 2002). These methods are convenient for giving prominence to regions of a tree, a region being a subtree or the nodes in a graph neighborhood of a given node.

Another approach to differential scaling allows the user to associate, with any node or nodes s/he deems special, a particular scale factor for that node. That factor is then used as a relative factor, and zooming of the overall view maintains the differential. Our TStar software supports this technique.

4.3 Hidden Nodes

While scrolling and scaling are often considered to be geometric operations, they are also means of information hiding, since scrolling is a way of showing some and hiding other information in a diagram, and scaling makes some details visible or invisible and shows more or less of the area of the entire diagram.

Now we describe two additional forms of information hiding, one of which we call "hidden nodes" and the other "ellipsis." They are closely related. They differ in three respects: how affected portions of the diagram are rendered, what kinds of sets of nodes can be hidden, and how the user interacts with them. We make these distinctions in order to provide a rich set of view controls to users, and to suggest additional possibilities for dealing with the complex but important issue of working with state-space tree diagrams.

We define a hidden node to be an invisible, but fully computed node object, that is neither rendered on the screen nor has any effect on the rendering of the rest of the tree. From the standpoint of screen appearance, a hidden node may just as well be a node that was never created or that has been deleted. If it has any children, they, too, must be hidden. In our TStar system, the user can hide a subtree in order to create a simpler view, or unhide a subtree to (re)include it in the diagram. Any overhead that would be associated with deleting or recreating corresponding states is avoided in the hiding and unhiding operations.

In order for the user to know that a hidden node exists in a tree, a small indication is available, on demand, in the form of a highlight on nodes having hidden children. Hiding the root is not permitted in TStar.

4.4 Ellipsis

A node *in ellipsis* is a tree node that is not displayed but is represented by an ellipsis indication. One or more nodes, in an ancestor chain, can be represented by a single ellipsis indication. An *ellipsis* is a collection of all the nodes in ellipsis that are represented by the same ellipsis indication.

An ellipsis therefore represents a set of nodes in a tree whose existence is important enough to indicate, but whose details and whose number are sufficiently unimportant that they are hidden. For example, the nodes in an ellipsis may have an important descendant, displayed in detail, but not have any details of their own worth showing.

The technique of ellipsis gives us a means to show nodes that have been explored but which we show in a very abbreviated form.

While we have just defined an ellipsis as a static set, in practice the user may work with dynamic ellipses. A basic dynamic ellipsis is a changing set of nodes in ellipsis, and a user-controlled dynamic ellipsis is a basic dynamic ellipsis with a boolean variable "currently-in-ellipsis" that can be set true or false by the user to adjust the visibility of the nodes without losing the grouping.

In principle a single node might be an element of zero or more separate user-controlled dynamic ellipses, but if a node belongs to two or more, and their variable values conflict, a priority mechanism must be used to determine the status of the node's visibility. One simple priority mechanism is event-based: when the variable associated with an ellipsis changes, update all its nodes to be consistent with it, at the possible expense of consistency with other ellipses. Another system would involve assigning numeric priority values to each user-controlled dynamic ellipsis. Our system TStar currently implements basic dynamic ellipses but not user-controlled dynamic ellipses or any priority systems.

The traditional means of displaying an ellipsis in text, formulas, and many kinds of diagrams is three dots in a line: "...". In a tree, an ellipsis can be indicated by using a dotted line for a path containing nodes in ellipsis. An example of this can be seen in Fig. 3 under the first small blue node.

There is a layout issue for ellipses. How long should the dotted line be? On the one hand, a reason for using ellipsis is to save space, and this suggests "as short as possible." However, it's important that the line communicate the ellipsis, and so if it's dotted, three or more dots should be visible. If the line leads to a fully visible node, then the length of the line might be suggestive of the depth of that node. If the node chain in ellipsis is hundreds of levels long, and the rendered line is the same length as a single normal tree edge, that might be misleading. One layout choice could be that an ellipsis line should be 1.5 times as long as a normal edge, and that if space permits, a numeric label next to the line should indicate the actual length of the path represented by the ellipsis. The factor of 1.5 will intentionally tend to misalign the visible descendants of the nodes in ellipsis from nodes on the left and right, reminding the viewer that they are not at the same depth.

Ellipses not only hide node details and reduce clutter (see, e.g., Taylor et al, 2006) but they save space by reducing the amount of space allocated to the nodes

in ellipsis. The space savings in the vertical direction depend on the number of levels of nodes in the ellipsis, and the savings in the horizontal direction depend on the maximum width of subgraph consisting of the nodes in the ellipsis in relation to the width of visible descendant subtree(s) of the nodes in the ellipsis. Obviously, the savings also depend on the tree layout algorithm involved.

5 Operations on Trees

While the most basic operations on state-space search trees involve applying operators to grow the trees, there are additional operations that become important in the context of particular applications. In collaborative design, for example, there are operations having to do with sharing work done and creating common views for communication and coordination. We describe some of these operations in order to discuss insights about related affordances for interaction and display.

5.1 Topological Operations

We can divide our tree operations into two categories. The first category is "topological." These operations involve modifying the structure of the trees involved, typically adding or removing nodes. The other category — "view oriented" — is concerned primarily with the management of visibility, scale factors, and annotations, especially when multiple users interact.

Merging. A fundamental operation on two trees constructed from a common starting state is *basic merging*. In this operation, a new tree is constructed by essentially combining the two roots into a single root of the new tree, and carrying over all the other nodes and edges. Any "duplicate" nodes (other than the root) are not merged, but separate copies are carried over in the new tree. As a justification for the utility of this operation, consider the following example. Two team members, Sue and Joe, independently explore possible sequences of steps from the same starting point. Some of these efforts duplicate the other's, and some don't. Each step taken is represented by an edge and the node it leads to. Each node other than the root is annotated with either "Sue" or "Joe". After their efforts are finished, a combined tree is created by performing a basic merge of the two trees. In the combined tree there is a copy of every node labeled "Sue" and a copy of every node labeled "Joe". Thus all the work performed separately is represented and obvious in the new tree. The provenance cues are preserved.

Another merging operation is *path-equivalence merging*. If a node N_a in tree A is derived from its root by the same operator sequence that derives a node N_b in tree B, and their roots are considered equivalent, then these two nodes are *path equivalent*. The tree C resulting from the path-equivalence merging of A and B is a tree in which every node of A and every node of B is represented by a path-equivalent node in C, but there are no two nodes in C path equivalent to each other. The annotations and other non-topological attributes of path-equivalent nodes must also be merged, but this process depends on particular view-merging

criteria, discussed later. Provenance is still represented in the result, but less clearly than with basic merging.

Path Extraction. Extraction is a means to isolate one chain of design steps from side chains. The input consists of a single tree and a designated node in that tree. All the nodes on the path from the root to the designated node are copied into a new tree, with their corresponding edges.

A variation of this operation is a segment extraction in which two nodes of the input tree are designated: a "top" node and a "bottom" node. The result of extraction is a path that typically does not contain the root but starts at the top node and extends to the bottom node. It is important that the top node be an ancestor of the bottom node for this to be an applicable operation. Our TStar/PRIME-Designer system does not implement segment extraction, because it assumes that all results of tree operators are trees rooted at nodes corresponding to the same initial state.

Path Merging. Whereas basic merging brings the information represented in a pair of compatible trees together into a single new tree, it does not combine the work of multiple individuals in the usual sense. In order to obtain a single new state that represents work done along two separate paths (e.g., by individual people) a new path needs to be formed from the two existing paths. This can only be performed if all preconditions of the involved operators continue to be satisfied at each point where an operator is to be applied.

Let us suppose that we have a tree formed by basic merging that represents the work done by Sue and by Joe. Sue's best work is represented by a node N_S and Joe's by N_J. The sequence of operators applied by Sue to transform the root's state into the state associated with N_S is the sequence $\mathcal{Q}_S = O_{S1}, O_{S2}, \ldots, O_{Sk_S}$, and the sequence applied by Joe to transform the root's state into the state associated with N_J is the sequence $\mathcal{Q}_J = O_{J1}, O_{J2}, \ldots, O_{Jk_J}$.

The result of merging $\mathcal{Q}_S$ and $\mathcal{Q}_J$ is defined or undefined, depending on whether the precondition of each and every O_{Ji} is satisfied by the state resulting from applying first the sequence $\mathcal{Q}_S$ and then all of O_{J1} up to $O_{J(i-1)}$. If all steps can be taken, then the result is defined, and the new path contains $k_S + k_J$ edges. The state corresponding to the last node in the path embodies each selected design act of Sue and each selected design act of Joe.

When a tree contains paths that can be merged (with a defined result), it is possible to cue the user, either proactively or on demand, with highlights. By clicking on a highlighted node, a user can ask for specifics about the path-merging opportunity. If the system can evaluate the possible benefits of such possible merges, for example by counting the numbers of edges in the resulting paths, then it can render a "visualization of opportunity," e.g., with color-coding.

Path Simplification. In some applications, the set of operators permits the underlying state graph for a search tree to contain cycles. As a simple example, if operator A means "Add component X to the design" and operator B means "Remove component X from the design", then A followed by B brings us back to

a state equivalent to where we began. Path simplification is any transformation of a path that produces a shorter path resulting in an equivalent state.

Path Reorganization to Enable Merging. In order for path merging to be possible, it must be the case that the preconditions of the operators along the second path continue to be satisfied as the merging is performed. However, that is not a sufficient condition to assure that the resulting state represents the intentions of the designers. The operator sequences must also be "non-interfering." The first sequence must not leave any state elements in a situation where those changes disrupt the correct application of operators in the sequence. An example is a selection operator that when applied, marks a particular state component as a target for a subsequent operator. If the first sequence changes the default selection, and the second sequence alters the currently selected element, an unintended state results.

This problem can be addressed in several ways. One is to leave it intentionally unfixed in order to train designers not to step on each other's toes. Another is to provide operator sets that have an orthogonality property, so that the subsets used by separate designers cannot interfere with each other. Another way is to allow potential interference in the operator sets, but to detect potential interference with particular sequences on demand and help resolve conflicts. One way to resolve conflicts is to provide "state restoration" (manually or automatically), adding additional links at the end of the first sequence, so that all selections or modes within states are back to their initial values. Another method is to compute an optimal ordering of the given design steps that will produce the desired target state with as few steps as possible. This may mean, for example, an irregular interleaving of operations from the first sequence with operations from the second sequence.

This is a special case of the general problem of reconciling conflicting changes to shared objects in collaborative editing, a common problem in software engineering teams. In software engineering, tools such as visual configuration managers have made it easier to identify and resolve conflicts (Accurev, 2008).

5.2 View-Oriented Operations.

While zooming, minifying, and editing visibility properties of nodes can be called "view editing," views are also affected by "view combining" and "view conversion." These are now briefly described.

View Merging. When two members of a team merge their trees via basic merging, very few conflicts can arise. The trees share only the root, and the root can neither be put into ellipsis nor hidden, at least in our example system.

Conflicts are another story when the trees are merged according to node equivalence based on operator sequences. Although a node N_a in tree A is considered path-equivalent to a node N_b in tree B if they are both derived from their (assumed equivalent) roots via the same operator sequence, they have lots of attributes that are by no means equivalent. They may share different authorship,

different annotations, different memberships in ellipses, they may be hidden or not, and they may have different scale factors associated with them.

Attributes of nodes such as authorship, creating timestamps, and descriptive annotations can usually be "unioned" – as entries on lists, the lists can be concatenated to preserve everything.

View properties such as node scale factors can also be unioned (kept along for dynamically changing the view), but an actual scale factor for the current view must also be determined. Three different methods are suggested, to be selected by the user at the time view merging is invoked, for different purposes. These methods are, briefly, "min," "max," and "diff." The min method is useful when each view represents work done to reduce scale factors on nodes judged to be unimportant, but each tree represents an incomplete job. The combined view will represent the totality of this work by the two team members. The max method is useful in the complementary situation, where each member has spent some effort to identify particularly important parts of the tree and has put in higher-than-normal scale factors for those nodes. The diff method is a means to highlight those parts of a combined tree in which the ratings of two team members (as expressed by node-specific scale factors) differ notably. Such a method takes the absolute value of the difference of the scale factors and maps it monotonically into a standard range of scale factors (0.1 to 2.0).

View Conversion. View conversion is the process of taking one view and modifying it automatically to meet a stated set of preferences (usually of someone else than the view's originator). This might involve rescaling of nodes meeting certain criteria, putting certain kinds of nodes into hiding or ellipsis, and turning on or off certain kinds of annotations. We haven't yet addressed this issue.

6 Issues

Several interesting questions have arisen in the course of this project. First is the question, To what extent can views be considered first-class objects?

6.1 Views as first-class objects

A first-class object is one that can be created, combined with others, saved, restored, and communicated. A system that supports such objects should provide facilities for the operations. While the primary activity of a problem solver or designer is to create solutions, there are many other aspects to their work. One of these is managing the process, including coordinating with others. A view of work done so far can be a valuable tool for such management. Systems such as TStar read, write, and edit files that represent trees and their views together. Originally designed as a kind of tree manipulation facility, the view-related affordances have gained in importance as the applications attempted with TStar have become more complex.

Trees are first class objects in many contexts – in language parsing systems and compilers, in hierarchical database systems, in phylogenetic information management systems, etc. Tree views can be considered to be annotated trees (where the annotations include layout and visibility directives). File formats for hierarchical data include XML and many application-specific variants. The proposal that views of state-space search trees be considered first-class objects does not seem unusual against that background. But the assumption that they are first-class suggests further work in exploring how views can better be automatically combined, customized, or synthesized for particular purposes.

6.2 Educational Issues

A major motivation for using state-space search trees as an interface is to help non-computer-scientist users understand the state-space approach to problem solving and design. How much should users learn, and how much can they be expected to learn in a limited time? What additional affordances in the interface would make the concepts of automatic search, evaluation, preconditions, and combination of work easier to understand? Possible answers might include embedded tutorials, or additional labels or controls.

6.3 Representing the Work of Agents

Agents can explore large numbers of states, but most states tend to be relatively uninteresting. This calls for display methods that succinctly summarize searches of large subtrees. Agents also blur the boundary between explored states and unexplored states. If an agent has explored the state, has it really been explored? It depends on *how* the agent explores the state as well as the extent to which the user understands what the agent has "seen." One degree of exploration has been reached when a set of states is realized – the states have been computed. But that may be of absolutely no consequence to a user if there is no "end product" of the realization – no additional nodes displayed on the screen, no reports of solutions found or promising nodes reached.

This suggests a need for new ways of rendering subtrees to show different degrees or forms of exploration and results of exploration. If evaluation functions are computed on nodes, color coding according to value is a natural approach. Subtrees currently being analyzed may be shown with flashing or other animation to indicate the locus of activity.

6.4 Large Teams

Another set of issues relates to the size and structure of teams. In scaling up to large numbers of collaborators, more controls are needed related to the modification of shared resources. Secure check-in and review processes may be needed to ensure that integrity of critical resources such as key components, master public views, etc., is maintained. Node annotations related to subteam responsibilities and permissions gain importance.

7 Acknowledgments

Thanks to Brian Johnson, Richard Karpen, Stefano Levialdi, Tyler Robison, and Linda Shapiro for comments on parts of the software described here. Partial support under NSF Grant 0613550 is gratefully acknowledged.

References

1. Accurev, Inc. Software configuration management. http://www.accurev.com/accurev.html. (2008).
2. Berry, L., Bartram, L., and Booth, K. S. Role-based control of shared application views. *Proceedings of the 18th Annual ACM Symposium on User Interface Software and Technology*, Seattle, WA. (2005). 23–32.
3. Card, S., and Nation, D. Degree-of-interest trees: A component of an attention-reactive user interface. *Proc. International Conference on Advanced Visual Interfaces (AVI02)*, Trento, Italy. ACM Press. (2002).
4. Chu, H., and Haynes, T. R. Methods, Systems and Computer Program Products for Controlling Tree Diagram Graphical User Interfaces and/or For Partially Collapsing Tree Diagrams. US Patent 20070198930.
5. Cinque, L., Sellers Canizares, S., and Tanimoto, S. L. Application of a transparent interface methodology to image processing. *J. Vis. Lang. Comput.* **18(5)** (2007) 504–512.
6. Dewan, P. Architectures for collaborative applications. In Beaudouin-Lafon, M. (ed.), *Trends in Software: Computer Supported Coop. Work*, **7**. (1998). 165–194.
7. Du Boulay, B., O'Shea, T., and Monk, J. The black box inside the glass box: Presenting computing concepts to novices. *International Journal of Human Computer Studies*, **51: 2**. (1999). 265–277.
8. Hanrahan, P. To Draw A Tree. Presented at the IEEE Symposium on Information Visualization. http://graphics.stanford.edu/ hanrahan/talks/todrawatree/. (2001).
9. Kobsa, A. User experiments with tree visualization systems. *Proceedings of the IEEE Symposium on Information Visualization 2004*. (2004). 9–16.
10. Lamping, J., Rao, R., and Pirolli, P. A focus+context technique based on hyperbolic geometry for visualizing large hierarchies. *Proc. ACM Conf. Human Factors in Computing Systems*. (1995). 401–408.
11. Levialdi, S., and Tanimoto, S. L. A transparent interface to state-space search programs. In Kraemer, E., Burnett, M. M., Diehl, S. (eds.): *Proc. of the ACM 2006 Symposium on Software Visualization (SOFTVIS 2006)*, Brighton, UK, September 4-5. (2006) 151–152.
12. Nilsson, N. *Problem-Solving Methods in Artificial Intelligence*. New York: McGraw-Hill. (1971).
13. Simon, H. *The Sciences of the Artificial*. Cambridge, MA: MIT Press. (1969).
14. Taylor, J., Fish, A., Howse, J., and John, C. Exploring the Notion of Clutter in Euler Diagrams. *Proc. DIAGRAMS 2006*.