Livesolving: Enabling Collaborative Problem Solvers to Perform at Full Capacity

Steven L. Tanimoto
University of Washington
Seattle, WA 98195, USA
tanimoto@cs.washington.edu

ABSTRACT

Collaborative problem solving is a key methodology for tackling complex and/or contentious problems. The methodology is supported by computer and communication systems that bring human solvers together with computational agents and provide clear protocols for exploring and rating alternative solution approaches. However, these systems can be challenging to use due not only to the complexity of the problems being solved but the variety of abstractions involved in managing the solution process, e.g., problem representations, collaborations, and strategies. This paper offers new ideas to help the human users of such systems to learn and work more effectively. It also suggests how problem solving may sometimes be carried out in performance contexts similar to those of livecoding improvisational music. Most important here is the identification of seven forms of liveness in problem solving that may heighten a solving team's sense of engagement. Common themes among them are increasing solvers' awareness and minimizing latency between solver intentions and system responses. One of the seven livesolving forms involves solvers in tracing paths within problem-space graphs. This and the other six forms derive from experience with a system called CoSolve, developed at the University of Washington.

1. INTRODUCTION

1.1 Motivation

Problem solving has somewhat different interpretations in different disciplines. In mathematics it usually consists of finding "answers" in the form of numbers or mathematical expressions, and it sometimes means coming up with a multi-step proof of a formal proposition. In business administration, problem solving typically refers to the decision making that owners and CEOs do in order to manage their operations or expand their markets. In psychological counseling, problem solving refers to the resolution of personal relationship conflicts, family strife, or depression. In engineering, problem solving may require the design of a machine or part that will satisfy a set of specifications. In spite of having somewhat specialized connotations in particular fields like these, people solve problems almost continuously during their lives, some of them very minute and subconscious such as the avoidance of puddles and stones when walking, and some of them more deliberate, such as deciding what groceries or brands of items to buy at the supermarket. Our aim is to support solving in some of these areas, but primarily situations in which there is enough value in formalizing the problem that people will be willing to consciously think about the problem in its formalized representation. Puzzles often figure prominently in discussions of computer problem solving, and I use them here because they facilitate conceptual explanations. However, I mention a variety of other problems, such as understanding climate change, to clarify other points.

Three reasons for supporting problem solving are (1) to help get important problems solved such as climate change, poverty, or world peace, (2) to help teach people how to solve problems in new and potentially better ways, and (3) to facilitate the art and performance of problem solving -- e.g., "livesolving." In our own group, a principal motivation is to find new ways in which technology can further empower people in problem solving. This paper focuses on forms of problem solving that resemble livecoding or live programming, as these terms have come to be known. Liveness in problem solving promises to contribute to either the effectiveness with which human solvers can accomplish the solving itself or the effectiveness with which the humans can artfully demonstrate solving activity. Livesolving is a new avenue by which to improve people's ability to solve problems and to share the share the solving process.

1.2 Definition

Livesolving refers to problem solving activity performed under circumstances associated with live programming: (a) the use of computational affordances that respond in real-time, (b) the potential presence or virtual presence of “consumers” such as audience members at a performance, and (c) improvisational elements of creative synthesis and non-retractability. Whereas live programming typically involves the editing of program code, livesolving involves the manipulation of problem structures such as formulations of problems (“problem templates”), state variables, and fragments of solution paths. By supporting and studying livesolving, we seek a better understanding of best practices in the design of methodologies and tools to support humans in complex, creative work.

2. PRIOR WORK

2.1 Theory and Systems for Problem Solving:

When artificial intelligence research started in the 1950s, it paid particular attention to problem solving. The "General Problem Solver" (GPS) developed by Allen Newell, J. C. Shaw, and Herbert Simon (1959), involved articulating a theory of problem solving that was subsequently developed and used both for designing computer agents and for modeling human problem solving (Simon and Newell 1971; Newell and Simon 1972). I refer to the core aspects of the theory as the "Classical theory of Problem Solving" (and just "Classical Theory" below). Numerous researchers have contributed to the theory, and a number of books have explained the ideas well (e.g., Nilsson 1972, Pearl 1984). The primary motivation for all this work was to be able to build intelligent, automatic problem solving agents. The secondary motivation was to find a new theory for human problem solving.

Others, including my group, have had a different motivation: to support human problem solvers with appropriate computer technology. At the Swiss Federal Institute in Lausanne, a mixed-initiative system was developed that supported a human problem solver in the formulation and solution of problems involving combinatorial constraints (Pu and Lalanne 2002). In my own group, the focus has been on supporting teams of solvers who collaborate on the solution to problems, with the assistance of computers. A system we developed, called CoSolve, embodies the classical theory and gives a solving team web access to a shared session that includes an explored subset of a problem space (Fan et al 2012). Two user roles are supported by CoSolve: "solving" and "posing." The solvers work in small teams to construct "session tree" objects that represent realizations of portions of problem spaces. Figure 1 is part of a screen shot from CoSolve showing a session tree for a Towers-of-Hanoi puzzle. The posing role in CoSolve engages users called posers in "scaffolded programming" by providing online forms that hold fragments of Python code to represent the components of a problem formulation. Related to posing are the activities of understanding and transforming information (e.g., see Russell et al, 1993, and Mahyar and Tory, 2014, on sensemaking and Kearne et al, 2014, on ideation.)

CoSolve was designed and built after a previous prototype, called TSTAR, proved to have so much latency for sharing of solving activity within a team that solvers did not feel that they were really working closely together when solving (Tanimoto et al 2009). CoSolve, however it managed to reduce the latency, still had enough latency that the solvers would sometimes get annoyed, and we are working on a successor system in order to further reduce such latency. It is the latency issue, for the most part, that connects the topic of computer-supported problem solving with livecoding.

2.2 Forms of Live Programming:

Reducing the latency between coding (editing code) and execution (seeing the results) has been a fundamental goal of live programming (Tanimoto 1990, 2013). The various ways in which this takes form depend on the style of programming and the available technologies. Without going into details here, we can simply say that live programming has two key aspects: (1) editing the running program without stopping it, and (2) reducing the latency between code editing and execution of that code to an imperceptibly small interval of time.

Livecoding usually refers to a particular form of live programming in which the programming is part of a musical performance, the music being generated by the code as the code is modified. This kind of programming is a form of improvisational performance, and it would have this character even if the output were changed from music to dynamic visual art. Livecoding is of particular interest to those who study the psychology of computer programming, not only because it is an unusual context for programming, but because it puts the programmer rather than the language or the software at the center of attention (Blackwell and Collins 2005). Later in this paper, I'll consider the related case where the output is problem solving state, rather than music or graphics per se.

2.3 Cognitive State of Flow

Bederson (2004) has argued that human-computer interfaces that support cognitive tasks should be designed to keep the user in a psychological state of "flow." Such a state can be characterized as one of intense focus as a task progresses. In order to support such user focus, the interface must be designed to avoid undue latency and to anticipate the possible next states of user attention, providing some of the information that will be needed prior to any delays due to communication networks or computer processing.

3. FORMS OF LIVESOLVING

By considering the several ways in which users of our CoSolve system interact with it, one can identify corresponding ways to reduce the latency of those interactions. In this section, seven forms of livesolving are described that derive from this analysis.

CoSolve engages human solvers in posing problems, initiating solving sessions, and solving the problems. In this paper, I neither address the posing process nor the session initiation process, and thus the focus is on solving. Solving in CoSolve consists in the deliberate selection of existing states in a session, the selection of operators to apply to them, and in some cases, the selection or specification of parameters to those operators. One "turn" consists of selecting one state already in the session, selecting one operator, and if the operator takes parameters, selecting and/or writing the values of those parameters. When the user inputs this information, typically over the course of several seconds in time, the choices are relayed to the server which computes a new state and sends it back to the client. The new state, added to the session by the server, is then drawn on the client screen as a new part of the tree representing the session so far.

Here are seven forms of livesolving in the context of a system that supports collaborative problem solving. They are discussed in greater detail in later sections. Not supported in CoSolve (with the possible exception of #5, livesolving presentations), we are building them into a new system code-named "Quetzal." The order in which the forms of liveness are listed here may seem arbitrary, but corresponds to my estimates of importance to design of next-generation general solving systems.

The first promotes thinking strategically at the level of problem spaces, which without the immediacy of visualization and live interaction is conceptually difficult for solvers. The second dispatches with any unnecessary latency in adjustment of parameters, something needed in solutions of many problems. The next two can affect the relationships between a solver and her/her collaborators and/or audience members. Livesolving form number 5 is associated with the design of dynamic objects such as animations or computer programs. Form number 6 means that solving decisions that might be considered finished can be revisited without necessarily undoing all the work done since. Finally, livesolving form 7 directly supports an experience of flow that offers a potentially exciting new modality of solving that might be effective for some problems or as a training aid for human solvers.

1. "Drawing and co-drawing solution paths." The user traces a path (or two or more users co-draw a path) through a display of the problem-space graph. The system supports the drawing process and limits the trace to legal paths, according to the problem specification. The turn-taking time of CoSolve-style interaction is reduced to an almost imperceptible delay. The possible challenge for the user of staying on a legal path can be limited by the system through intelligent interpretation of the user's drawing intent, as explained later.

2. "Live parameter tuning." One part of the operator-application turn described above is the specification of parameter values. CoSolve requires that the user make a commitment to a particular parameter vector prior to operator application. This means that the user cannot easily make a series of small adjustments to a parameter value using feedback from the system. Live parameter tuning means that, upon operator selection, the system will display a new state showing the consequences of the current parameter values without requiring a commitment to those values, and furthermore, there is instant updating of the state and its display as the parameters are adjusted using appropriate input widgets such as sliders.

3. "Synchronous co-solving." Whereas CoSolve's group awareness feature allows each solver to become aware of other users' edits to the solving-session tree through small graphical indicators, actual updates to the user's session-tree display do not occur until the user requests them. On the other hand, with fully synchronous co-solving, any team member's edit to the solving-session tree is automatically propagated to each team member's view of the tree with only a small, deliberate delay introduced so that a smooth animation can provide continuity between the old and new views.

4. "Livesolving presentations." This is solving a problem, using computer technology, in front of an audience or video camera. While it doesn't require any new technology per se, it can benefit from a variety of standard technologies and affordances, such as means for highlighting, annotation, bookmarking, and linking to multimedia facilities.

5. "States alive." In CoSolve, states in a session are shown as nodes in a tree, with static images as graphical illustrations of each state. Additional details of state information are available to users by clicking, after which JSON code is displayed. The restriction that states be presented in this manner means that certain kinds of problems, such as animation design problems, cannot be solved in the most natural manner. To improve that, states can be permitted to have dynamic displays. Thus, if the state represents a dynamic process, such as a computer program, then states-alive liveness will allow the user to observe, within a session tree or problem-space visualization, a collection of running animations, reflecting the running programs the states represent.

6. "Ancestor modification." In CoSolve, each turn of operator application is a commitment. States can only be added to a session, not deleted. Furthermore, existing states cannot be modified. That limitation simplifies session management in collaborative contexts, and often there is no problem having a few extra states around, representing evidence of solver exploration prior to finding a good path. The difficulty comes

up when a long chain of operator applications hangs off of an ancestor that needs a change. CoSolve treats the ancestor as immutable, and in this sense of no longer supporting interaction, "dead." With ancestor-modification liveness, edits to ancestors are permitted, and they have the effect of raising such states from the dead and letting them be modified. There are two alternative policies for this. One is that the ancestor is immediately cloned, with the original remaining immutable. The other policy is that there is no clone, and the ancestor itself will be modified. Modification may involve simply re-specifying or re-tuning the parameter values used in the operator that created the ancestor. However, it may also involve a change requiring the use of an entirely different operator and possibly new set of parameter values as well. In either case, a change to the ancestor may require either of two kinds of changes to its descendants: (a) updating of the state variables affected by the changed parameter(s), (b) pruning of descendant subtrees due to violations of operator preconditions by the new ancestor or by other updated states.

7. "Driving an agent." Whereas the first form of livesolving (drawing and co-drawing solution paths) achieves its dynamics by the users' hand or stylus motions, in driving-an-agent liveness, the solving process proceeds with a form of momentum (speed, direction in the problem space, restricted deceleration) that means steering decisions must be made at a rate that avoids the wrong turns and dead ends that would be made by the agent's default behavior. Computer agents for problem solving come in many forms, and the most obvious application of this form of liveness works with a depth-first-search agent. However, it can apply to almost any solving agent that can accept input over time. To make this form of liveness effective, a good display must be provided to the driver (the user or users) so that each decision can be made on the basis of the best information available at the current point in the search.

4. LIVESOLVING WITH FINITE PROBLEM SPACES

In this and the following two sections, I discuss the seven forms of livesolving in more detail. I have chosen to categorize four of them roughly according to the nature of the problems they might be applied to and the other three by their "social nature." However, this classification is primarily for convenience of presentation rather than something necessitated by the affordances themselves.

4.1 Characteristics of Finite Problem Spaces

A finite problem space is one in which the number of possible states for a solution process is bounded. For example, in a game of Tic-Tac-Toe, a reasonable formulation leads to a problem space with 5478 legal states. (This rules out states having more Os than Xs, and many others, but not duplicates modulo symmetries.) Although mathematical finiteness does not strictly imply enumerability by a computer in a reasonable amount of time, we will consider, in bringing liveness to solving these problems, than the space of states can be mechanically explored, fully, and that the time required to do this is taken care of in advance of the livesolver's encounter with the solving process. The livesolver's job might therefore be to demonstrate the selection of a good sequence of states from the problem's initial state to one of its goal states. The fact that the states have been precomputed before livesolving begins does not necessarily mean that the solution has been found in advance (although that is possible).

4.2 Drawing the Solution Path by Tracing over a Problem-Space Graph.

Our first form of livesolving, drawing and co-drawing solution paths, is a good fit for working in finite problem spaces. The finiteness means that it might be feasible to determine a state-space map for the problem in advance, and to offer it to the livesolver(s), as a substrate, for solving by drawing.

As an illustration, let's consider the Towers-of-Hanoi puzzle. Part of a CoSolve session tree for a 3-disk version of the puzzle is shown in Fig. 1. The initial state is at the root of the tree, and states created by making moves from it are shown underneath it. Now let's consider a version with 4 disks. The state space for this problem has $3^4 = 81$ states. The state-space graph for this problem contains one node for each state and an edge between nodes n_i and n_j (corresponding to states s_i and s_j) provided there are operators that transform state s_i into s_j and s_j into s_i. Various layouts for this graph are possible. Figure 2 shows a layout constructed by assigning to each node three barycentric coordinates determined by assigning values to the presence of the various disks on each of the three pegs. The smallest disk is counted with weight 1; the next with weight 2; the third with weight 4; and the fourth with weight 8. The initial state of the puzzle is plotted at the lower-left vertex of the equilateral triangle, because the weight of all disks is on the first peg. Similarly, the other two vertices of the large triangle correspond to states in which all the disks are on the middle peg or all the disks are on the right-hand peg.

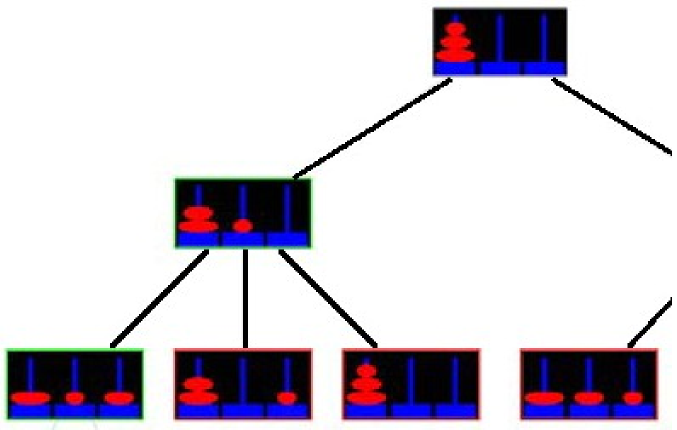


Figure 1. Portion of a CoSolve session tree for a 3-disk Towers of Hanoi puzzle. A solver generates one state at a time by making a selection of a node and an operator.

There are six operators in a standard Towers-of-Hanoi problem formulation. Each is of the form "Move the topmost disk from Peg i to Peg j." In Figure 1, the spatial direction in which a node transition is made corresponds directly to the operator involved. For example, from the initial state, moving the smallest disk from Peg 1 onto Peg 2 is shown by the line segment going from the node marked 0, diagonally up and to the right.

Livesolving with this state-space graph is tricky, because the solver would like to move directly from the lower-left vertex to the lower-right vertex, and yet no direct path exists. In fact, the shortest

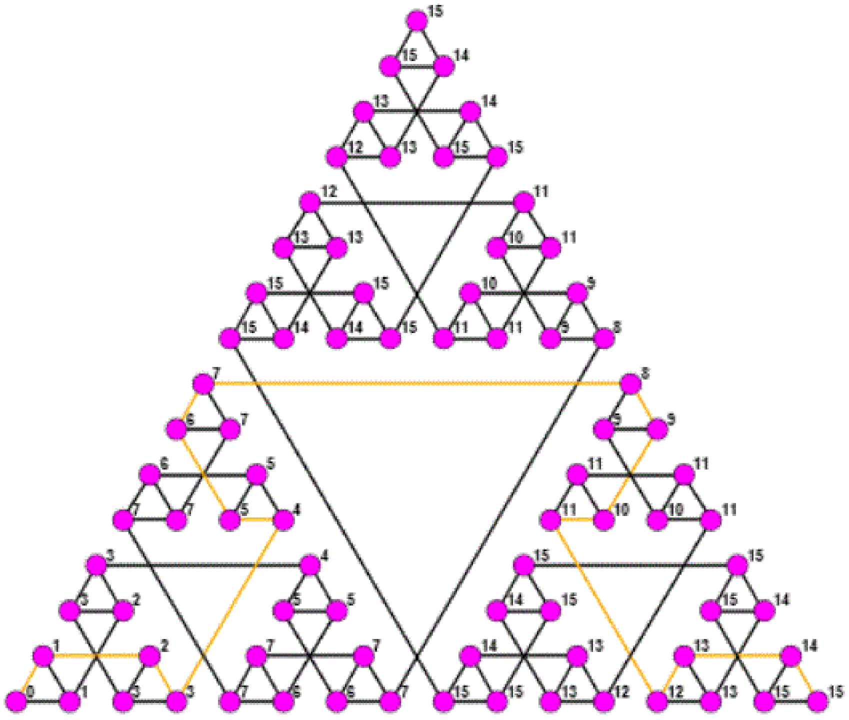


Figure 2. State-space graph for the 4-disk Towers-of-Hanoi puzzle used as a substrate for livesolving by drawing and co-drawing solution paths. A shortest solution path (the "golden path") is shown.

solution path involves some counterintuitive moves of disks from right to left now and then. Thus, there is an opportunity for livesolvers to develop skill at solving these puzzles and demonstrate the skill using this mode of livesolving, much as Rubik-cube experts like to show how quickly they can spin the faces from a random position and get all the marks on each face of the cube to have a uniform color.

One of the challenges for those who formulate problems is to come up with appropriate graph layouts (that we call problem-space visualizations) to permit this sort of solving by drawing. Finiteness of the problem space is a help, but that is not sufficient to make such layout easy. We discuss the challenges of layout further in the section on Livesolving in Infinite Problem Spaces.

4.3 "States-Alive" Liveness

In a puzzle such as the Towers of Hanoi, each state represents a static snapshot -- a possible configuration of disks on pegs that might reached in the course of attempting to solve. A static image is a suitable graphical representation for a solver working on the problem. On the other hand, there are problems that involve dynamic entities, such as the problem of designing an animation clip or creating a computer program that simulates bouncing balls on a table. With states alive, each state displayed in a solving session is backed by its own generative process, typically a running program operating in a separate thread of control. Particularly when the solver is comparing alternative states in order to select one for additional development, seeing the alternatives, "in action" can potentially improve the quality of the decision and have other benefits, such as making more apparent the properties of the states, not only for the solver, but for a livesolver's audience, if there is one.

One of the simplest examples of states-alive liveness occurs when the problem is to design an animated GIF image, and alternatives are displayed in their natural form -- as animated GIFs. An obvious potential drawback is that such displays, with multiple animated GIF images, can be annoying to users, because they tend to be visually busy and distracting, especially when the important decisions about which state to expand, etc., have already been made. The answer should be that the states-alive facility comes with suitable controls for disabling and enabling the live displays that go with such states.

The use of separate threads of computation to support each of multiple states being shown simultaneously on the screen could easily consume available computation cycles or memory. Thus it is likely to work best when relatively few states will be displayed in this mode at a time. Finiteness of the problem space may help, at the very least, and policies or controls that limit the number of live states to a dynamically changing working set, would be appropriate. For example the live displays may be limited to one state and its immediate children, or to the states along one short path in the state-space graph.

4.4 Live Feedback During Solving

Crucial to maintaining a sense of flow to a user is providing timely feedback. In problem solving with CoSolve, the system generates a state display image each time the user creates a new state. With solving by drawing, the user should be receiving at least two forms of feedback: the display of the path as just drawn, plus the state visualization for the state most recently reached on the path. The time between user selection or specification of a new current state and its subsequent display is "state-display latency." By reducing this time to a few milliseconds, a system can better support a solver's experience of being in the flow. The finiteness of the problem space suggests that the state visualizations could all be precomputed and stored as properties of the nodes of the graph. This would allow showing these visualizations with low state-display latency, even if the user draws paths through the graph quickly.

In addition, feedback about the current node reached may include the display of values computed from the state: heuristic evaluation function values, for example. Even if these values or the state visualizations themselves are not precomputed, they may be computed lazily, scheduled for computation as soon as the node is added to a path under consideration. Yet the livesolver is not required to wait for them before exploring further. When computing resources are scarce, these visualizations should be computed strategically, to maximize the sense of orientation in the state space. For example, every other state along the path might be displayed if not every state can be.

5. LIVESOLVING IN INFINITE PROBLEM SPACES

5.1 Working with Parameters of Operators.

If the states of a problem have continuous variables in them, such as a temperature or speed, then the problem's state space is inherently infinite[1]. The space could be made finite by limiting such variables to a fixed set of values, but that may not always be appropriate. A consequence of having continuous variables in operator parameters is that the selection of appropriate parameter values can easily become a "flow killer" in a problem solving system. In order to avoid this, *live parameter tuning* should be supported. To use this, a user selects an operator and the system may respond by creating a new state using default values of the parameters. At the same time, controls such as sliders, dials, or image maps appear, as appropriate for each of the parameters required by the operator. The user then adjusts the parameter values, watching the new state update immediately to reflect the adjustments. A variation of this involves having the system present not just one new state, but a sample set of new states corresponding to a default set of values for each of the parameters. For example, an operator that adds a piece of furniture to a living-room layout may have three parameters: furniture type, x position, and y position. The x and y positions are values from a continuous range; the system can provide a sample set of (x, y) pairs that are generated as a cartesian product of a sample set for x with a sample set for y.

[1] Of course, real number are restricted in most computers to a finite set of rationals, but this is an artifact of implementation rather than problem modeling.

In addition to the use of default parameters and fixed sample parameter vectors, intelligent agents may take a solver's specification for a sampling distribution to use as the basis for parameter value sampling. An example of this is the specification of a "beam" of parameter values: a collection of tightly spaced values within a narrow subrange of the parameter's full allowable range.

5.2 Ancestor Modification

A variation of the live parameter adjustment affordance when creating new states (as described above) is to permit this in more cases. One of these is to change the parameters of an old state. If the state has no children, this is equivalent to what we have already discussed. If the state has children, then these children will generally be affected by any change to this or any ancestor. For each descendant of a modified state, any of several cases may apply: (a) no change, either because that influence of the change is blocked by some other ancestor, between the changed one and the descendant in question, (b) changes to the descendant propagate down, with no violations of operator preconditions on the path between the ancestor and the descendant, or (c) at some state between the ancestor and the descendant, the precondition for the operator originally used to create the state is now violated. In this third case, the logical update to the session tree is to remove the subtree whose root's operator precondition now is violated. However, that can be quite disruptive to the solving session, particularly if that deletion happens as an unintended accident of parameter adjustment. Flow may be better maintained by avoiding the possibility of such an accident. This can be done by automatically cloning any ancestor that has been selected for modification, along with all its descendants. If this policy is too expensive in terms of state duplication, a less costly cloning policy is to clone only when a descendant is about to be deleted, and then to remove the clone if the parameter is adjusted back in a way that removes the precondition violation. While ancestor modification complicates the implementation of the system, it offers substantial flexibility to the solver, allowing the "resurrection of otherwise unmodifiable states from the dead."

5.3 Path Drawing Without Precomputed Graphs

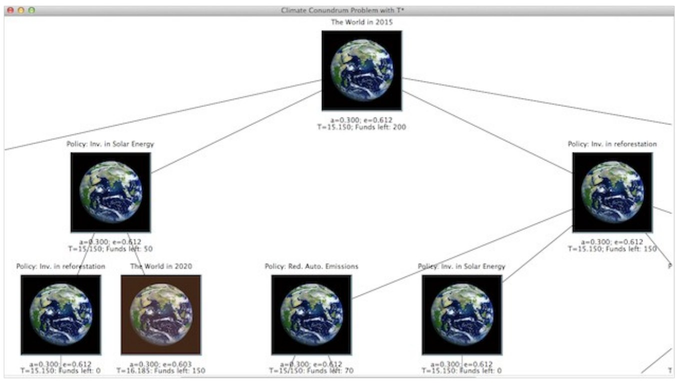
A problem space that is infinite cannot have its entire graph realized in advance or at any time. However, a portion of it may be computed in advance, and more of it may be computed during a solving session. The following challenges may arise:

The time required for the system to apply an operator and produce a new state may cause enough latency to threaten the user's sense of flow. When this is the case, the sense of liveness can sometimes be saved by (a) suitable display of partial information, progressively as the new state is computed, or (b) covering the latency by posing questions to the user or offering more readily available alternatives.

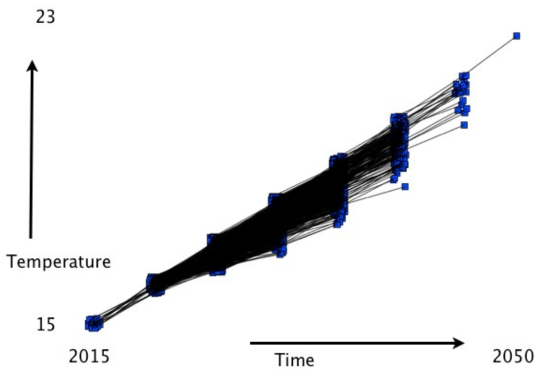
The mapping of states to layout positions may cause collisions among realized states. While the Towers of Hanoi graph of Figure 2 has a clear minimum spacing between adjacent nodes, in an infinite problem space, this may be difficult, if not impossible to achieve. The fact that parameters of operators, and state variables themselves, may take on values in continuous ranges (e.g., real numbers between 0.0 and 1.0), means that states may differ by arbitrarily small differences in any state variable. If the designer of a problem-space layout wishes to have a minimum separation of nodes on the screen of 25 pixels, then two fairly similar states s_1 and s_3 might be positioned at $x_1 = 100$ pixels and $x_2 = 125$ pixels, respectively. If s_1 and s_3 differ simply because there is a state variable v whose value in s_1 is v_1 and whose value in s_3 is v_3, (where $v_3 - v_1$ is close to zero), there still may exist a state s_2 having its value of v half way between v_1 and v_3. Now where should the layout place the display of s_2? It could dynamically adjust the layout and double the spacing between s_1 and s_3, in order to place s_2 directly between them without violating the minimum spacing constraint. However, this raises the problem of layouts changing during the solving process, which may increase the cognitive load on the solver. If layout positions are not to change during the session, then collisions on the screen are likely, as the realized portion of the infinite state space grows, and interface mechanisms are required to allow seeing and selecting any of the items that pile up on each other. One approach for this is to have multiple layouts or projections, with the user in charge of moving among different views of the same problem space. Another approach is to allow the layout to be locally dynamic, with techniques such as force layout, but overall mostly static.

Hand-in-hand with more sophisticated visualization techniques for the problem space are more sophisticated capabilities for the drawing tool supporting the livesolving. As the user draws a path through the 2D view of the problem space, the path must intelligently adapt to be a legal path through the actual problem space which may be of arbitrary dimension.

System policies supporting intelligent drawing assistance may include on-the-fly search, so that the state pointed to by the solver is automatically connected by a shortest path to the nearest landmark state, unless the distance from such a state exceeds a threshold set in the current drawing policy. Landmark states are defined by the solver by clicking on them and marking them during the course of solving. Another drawing policy that can be used is for the system to deal with drawing ambiguity (i.e., which state the user intends to move to next) by maintaining a set of nodes to represent the possible next state; the ambiguity may be gradually removed through additional user hints, or the


Climate Conundrum Problem with T*
The World in 2015
a=0.300; e=0.612
T=15.150; Funds left: 200
Policy: Inv. in Solar Energy
a=0.300; e=0.612
T=15.150; Funds left: 50
Policy: Inv. in reforestation
a=0.300; e=0.612
T=15.150; Funds left: 150
Policy: Inv. in reforestation
a=0.300; e=0.612
T=15.150; Funds left: 0
The World in 2020
a=0.300; e=0.603
T=16.185; Funds left: 150
Policy: Red. Auto. Emissions
a=0.300; e=0.612
T=15.150; Funds left: 70
Policy: Inv. in Solar Energy
a=0.300; e=0.612
T=15.150; Funds left: 0

(a)



(b)

Figure 3. Visualizations for the Climate Conundrum problem: (a) session tree, and (b) problem-space graph. The graph layout supports solvers' understanding of progress towards the goal, but it is poorly suited to path drawing, due to its densities of nodes and edges.

ambiguity may be permitted to persist, with the understanding that the system will eventually end up with multiple possible candidate solution paths.

Even if a problem-space visualization fails to provide a clear sense of direction or sense of proximity to any landmarks in the problem space, the ability to explore the space by drawing a path can still have appeal to livesolvers. At the very least, a graphical representation of the paths explored can provide a kind of visual session history whose topology (e.g., tree structure) indicates alternatives tried and points at which backtracking or branching occurred.

5.4 Effects of Dynamic Realization of the Problem Space.

Real-time solving with full state displays may be impractical, due to high state-display latency in complex problems. To keep the solver in the flow, it may be possible to use techniques such as (a) planning via homomorphic problem spaces (Pearl 1984), (b) low resolution visualizations on fully computed states, or (c) sampling policies for visualizations – e.g., every other state on a path is visualized, or every *n*th is visualized, or a state is visualized to degree *f*(*k*) if its importance exceeds *k*. Alternatively, visualizations may be computed lazily with different priorities, so that with enough time, a full path is shown in detail, but rapid work by a human will be supported by having the most-important visualization and state-computation work performed first.

6. SOCIAL LIVESOLVING

In this section, I discuss three of the aforementioned livesolving forms: synchronous co-solving, livesolving presentations, and driving an agent.

6.1 Synchronous Co-solving

Synchronous co-solving is collaborative solving activity in which a single representation of the solving session is shared among the solvers. When solvers work from geographically distant locations, the shared representation is accessed via client views that must be updated by the system. CoSolve's mechanism for these updates pushes notifications to solvers that changes have been made and then updates a solver's view if and when the solver clicks to request the update. Our new system Quetzal, on the other hand, pushes the new views themselves as soon as any team member makes a change. The potential exists for such changes to be disrupting to individual solvers, as, of course, too many cooks can spoil a broth. This is true for groupware in general (Gutwin et al 1995). But were synchronous co-solving not to be supported, close collaboration would be more difficult. We compensate, in Quetzal, for the potential disruption, by smoothly animating each of the changes to the session-tree view over a 2-second period. The potential disruption can also be reduced by clearly labeling incoming nodes as new and identifying their authors using avatars, names, or color coding.

If a solving environment gives to agents the status of "team member," then the potential for disruption is amplified. Agents can move too quickly to create new material in a solving session, and also, there is a high likelihood of their work being considered to be mostly garbage. Consequently, special affordances are required for managing the contributions of agents, such as running them in sandboxed subsessions. New or novice team members can also be managed with such devices.

6.2 Livesolving Presentations

Livesolving presentations share several elements with livecoding performances. First, the presence of an audience demands that there be an effect element of communication during the activity; the audience is expected to derive something from the experience, be it entertainment, education, or other information. Furthermore, livecoding is traditionally improvisational. In order for livesolving to also be improvisational, the tools must support extemporaneous decision making and short response times. In music, live-performance improvisation involves real-time commitment to musical elements that cannot be retracted (unlike studio recording session in which mistakes can be corrected). A jazz musician who hits a "wrong" note (wrong in the sense of unintentional or off-sounding) will typically turn this bug into a feature by developing it, recontextualizing it, and perhaps, gracefully abandoning it. A livesolver who arrives at a dead end in the problem space should likewise make the bug into a feature by building a narrative context around the dead-end that makes it a natural part of exploring the problem space. The audience might judge the performance either in terms of speed to solution or in terms of "narrative richness" or honesty in exposing the solvers' thought process. Computer support for this form of livesolving may include the display of graphical signals to the audience that new states are being explored (e.g., green) backtracking is occurring after reaching a dead-end (red) or that unforced backtracking is occurring (e.g., orange). Music livecoders often include colored code highlighting in their editors to reflect re-evaluation, syntax errors, or other conditions.

6.3 Driving an Agent

Driving an agent is listed here under social livesolving, but it may or may not involve relationships that feel social. The relationship between a driver and a car is not traditionally a social one. However, with automatic driving, the user and driving agent may have a relationship involving communication in spoken English, which may seem social. Other social relationships in driving exist, too: driver-navigator, driver-backseat driver. In the case of direction-finding, the navigator in a car may be a person or a Garmin GPS unit. In livesolving, agents may be making decisions about the order in which states of a problem space are explored, but a user driving that agent may steer the agent to alter the default decisions. When the operators take parameters, the choice of parameter values may be a large enough task that it can be shared by multiple drivers. One livesolver determines the type of furniture to be placed in the living room while another selects an (x,y) position for it by clicking or touching on the room plan. Computer support for driving an agent means offering interfaces to livesolvers appropriate to the particular kind of agent being driven.

7. DISCUSSION

In this section I address three questions. First is "Where did the forms of livesolving come from?". The second is "What can make solving live?" The reason to ask this is to offer a review of issues already raised, but in a slightly different light. Third is "How does livesolving compare with livecoding?" This question is prompted by the theme of the conference.

7.1 Where did the forms of livesolving come from?

After our group built two tools, TSTAR and CoSolve in order to facilitate deliberate, human problem solving, the question arose of what more we could do, through providing computer technology, in order to support human problem solving. Our two approaches involved (1) looking for ways to reduce latency in our current computer-mediated solving process, and (2) looking for alternative kinds of solving experiences that could be supported by the same classical theory and underlying support engine. The ideas took shape in the course of trying to answer the other two questions in this section.

7.2 What Can Make Solving Live?

Solving a problem, with help from others, including computers, can feel live when the solver experiences the kind of feedback expected when working among a group of engaged individuals. Having a conversation in which one's partner responds promptly and thoughtfully is one example. Working with a tool that performs its assistance without undue delay is another. The goal of our work is to enable solvers not only to experience liveness, but to achieve cognitive flow -- a state of mind in which mental transitions from one idea to another proceed at a rate that is fast enough to inhibit distractions from setting in, yet slow enough to permit comprehension and the development of good judgment. The seven forms of liveness presented above offer some means to this end. However, there are surely many other factors that affect the ability to achieve flow that crop up in reality such as the solver's freedom from external anxieties, the time available, his or her motivation to solve the problem at hand. We also require that the problem be formulated according to the classical theory, and the formulation process is a topic for future work.

7.3 Comparing Livesolving and Livecoding

Livesolving and livecoding, as presented here, are closely related. An obvious difference is that livecoding depends on code editing as its fundamental activity. Livesolving depends on problem-space exploration via operator application. From a theoretical perspective, they are equipotent. Livecoding in a Turing-complete language can result in an arbitrary computable function. Livesolving in a suitable formulated problem space can result in the composition of an arbitrary computational object, such as any computable function. Either can be done in a performance context, and either can be used to generate music or any other computable process. The difference, then, is in the style of interaction that the solver or coder has with the system. The ways of thinking about the process, also, tend to differ. In a typical livecoding performance, there is not any goal to end up with a masterpiece of code at the end of a performance; rather the code evolves in order to generate the music required at each point in time, and at the end of the performance, the final version of the code may simply generate silence. In livesolving, however, there usually is an understood goal state, and there are criteria for success and failure related to it.

8. CONCLUSIONS

Problem solving using computers is an activity similar to programming in that it involves computational thinking, yet different from programming because of the structures and affordances typically provided. Exploiting the classical theory of problem solving, existing systems such as the University of Washington's CoSolve facility have shown how computational resources can be used by human solving teams to solve problems in ways that garner certain advantages. These systems, however, don't adequately support solvers in reaching their full potential as solvers. Here, I have presented seven forms of liveness in problem solving to help overcome those obstacles. Some of these forms involve how solvers specify the exploration of the problem space, while others involve the ways in which they interact with other solvers on the team or with an audience. Future work on livesolving will involve evaluation, comparisons to software design patterns, extensions to posing, the design of agents and agent interfaces, and enabling the scaling of facilities for large teams, complex problems, and more intelligent and fast computational support.

Acknowledgments

Thanks go to the CoSolve developers, especially Sandra Fan, Tyler Robison, and Rob Thompson. Thanks go also to Alan Blackwell, Sam Aaron, Luke Church, Advait Sarkar, Henri Angellino, Helmut Prendinger, and Tim Bell for supporting the development of these ideas during my 2013-2014 sabbatical. I'd also like to express my appreciation to the reviewers for their thoughtful comments.

REFERENCES

Bederson, Benjamin. 2004. "Interfaces for Staying in the Flow." *Ubiquity.* ACM. (September).

Blackwell, Alan and Collins, Nick. 2005. "The Programming Language as Musical Instrument." *Proc. PPIG05* (Psychology of Programming Interest Group), pp.120-130.

Fan, Sandra B., Robison, Tyler, and Tanimoto, Steven L., 2012. "CoSolve: A System for Engaging Users in Computer-supported Collaborative Problem Solving." *Proc. International Symposium on Visual Languages and Human-Centric Computing*, Innsbruck, Sept., pp.205-212.

Gutwin, Carl; Stark, Gwen; and Greenberg, Saul. 1995. "Support for Group Awareness in Educational Groupware." *Proc. Conference on Computer Supported Collaborative Learning*, pp. 147-156, Bloomington, Indiana, October 17-20, Lawrence Erlbaum Associates.

Kearne, Andruid; Webb, Andrew M.; Smith, Steven M.; Linder, Rhema; Lupfer, Nic; Qu, Yin; Moeller, Jon; and Damaraju, Sashikanth. 2014. "Using Metrics of Curation to Evaluate Information-Based Ideation." *ACM Transactions on Computer-Human Interaction*, Vol. 21, No. 3.

Mahyar, Narges and Tory, Menlanie. 2014. "Supporting Communication and Coordination in Collaborative Sensemaking." *IEEE Trans. Visualization and Computer Graphics*, Vol. 20, No. 12, pp.1633-1642.

Newell, Allen; Shaw, J.C.; and Simon, Herbert. 1959. "Report on a General Problem-Solving Program." *Proc. International Conference on Information Processing*. pp.256-264.

Newell, Allen; and Simon, Herbert. 1972. *Human Problem Solving.* Englewood-Cliffs, NJ: Prentice-Hall.

Nilsson, Nils J. 1971. *Problem Solving Methods in Artificial Intelligence*, New York: McGraw-Hill.

Pearl, Judea. 1984. *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison Wesley.

Pu, Pearl, and Lalanne, Denis. 2002. "Design Visual Thinking Tools for Mixed Initiative Systems." *Proc. 7th International Conference on Intelligent User Interfaces* (IUI '02). ACM, New York, NY, USA, 119-126.

Russell, Daniel M.; Stefik, Mark J.;, Pirolli, Peter; and Cart, Stuart K. 1993. "The Cost Structure of Sensemaking." *Proc. INTERCHI '93*, pp.269-276.

Simon, Herbert A., and Newell, Allen, 1971. "Human Problem Solving: The State of the Theory in 1970." *American Psychologist*, 1971, pp.145-159.

Tanimoto, Steven L. 1990. "VIVA: A Visual Language for Image Processing." *Journal of Visual Languages and Computing*, (1), 2, pp.127-139.

Tanimoto, Steven L.; Robison, Tyler; and Fan, Sandra. 2009. "A Game-building Environment for Research in Collaborative Design." *Proc. International Symposium on Artificial Intelligence in Games.*

Tanimoto, Steven L. 2013. "A Perspective on the Evolution of Live Programming." *Proc. LIVE 2013, Workshop on Live Programming*, IEEE Computer Society.