The HuGS Platform: A Toolkit for Interactive Optimization

Gunnar W. Klau[1], Neal Lesh[2], Joe Marks[2], Michael Mitzenmacher[3], Guy T. Schafer[4]

[1] Vienna University of Technology, Austria
guwek@ads.tuwien.ac.at
[2] Mitsubishi Electric Research Laboratories, 201 Broadway, Cambridge, MA, 02139
{lesh,marks}@merl.com
[3] Harvard University, Computer Science Department
michaelm@eecs.harvard.edu*
[4] Harvard University Extension School
gschafer@fas.harvard.edu

ABSTRACT

In this paper we develop a generalized approach to visualizing and controlling an optimization process. Our framework, called Human-Guided Search, actively involves people in the process of optimization. We provide simple and general visual metaphors that allow users to focus and constrain the exploration of the search space. We demonstrate that these metaphors apply to a wide variety of problems and optimization algorithms. Our software toolkit supports rapid development of human-guided search systems.

Our approach addresses many often-neglected aspects of optimization that are critical to providing people with practical solutions to their optimization problems. Users need to understand and trust the generated solutions in order to effectively implement, justify, and modify them. Furthermore, it is often impossible for users to specify, in advance, all appropriate constraints and selection criteria for their problem. Thus, automatic methods can only find solutions that are optimal with regard to an invariably over-simplified problem description. In contrast, human-in-the-loop optimization allows people to find and better understand solutions that reflect their knowledge of real-world constraints.

Finally, interactive optimization leverages people's abilities in areas in which humans currently outperform computers, such as visual perception, learning from experience, and strategic assessment. Given a good visualization of the problem, people can employ these skills to direct a computer search into the more promising regions of the search space.

The software we describe is written in Java and is available under a free research license for research or educational purposes.

*Supported in part by NSF CAREER Grant CCR-9983832 and an Alfred P. Sloan Research Fellowship. This work was done while visiting Mitsubishi Electric Research Laboratories.

1. INTRODUCTION

Research on designing systems to solve optimization problems, such as routing, layout, and scheduling problems, focuses primarily on developing automatic algorithms to search the exponentially large space of possible solutions more efficiently. Typically, the user's role in these systems is to specify the problem, including weights on predefined criteria for evaluating candidate solutions, and then initiate a computer search to find an optimal solution.

In this paper, we describe a generalized approach to visualizing and controlling an optimization process. Our framework, called Human-Guided Search (HuGS), actively involves people in the process of optimization. We provide simple and general visual metaphors that allow users to focus and constrain the exploration of the search space. We demonstrate that these metaphors apply to a wide variety of problems and optimization algorithms. We also describe middleware software that supports rapid development of such human-guided search systems and four applications we have built with it. This software is written in Java and available for research or educational purposes.[1]

Our approach addresses many often-neglected aspects of optimization that are essential for people to obtain usable solutions. Users must understand and trust the generated solutions to make effective use of them. Furthemore, it is often impossible for users to specify, in advance, all appropriate constraints and selection criteria for every possible scenario of a problem. Consider, for example, someone producing a monthly work schedule. She must understand the solution to convey it to the affected employees. Moreover, she must understand how to make modifications as new needs arise. Futhermore, she may have experience in evaluating candidate schedules that is difficult to convey to the computer. By participating in the construction of the work schedule, she can steer the computer towards an optimal schedule based on her real-world knowledge.

Additionally, human guidance of search can improve the quality of solutions for at least two different reasons. First, it leverages people's abilities in areas in which humans currently outperform computers, such as visual perception, learning from experience, and strategic assessment. Given a good visualization of the problem, people can employ these skills to direct a computer search into the more promising regions of

[1]Contact lesh@merl.com for details.

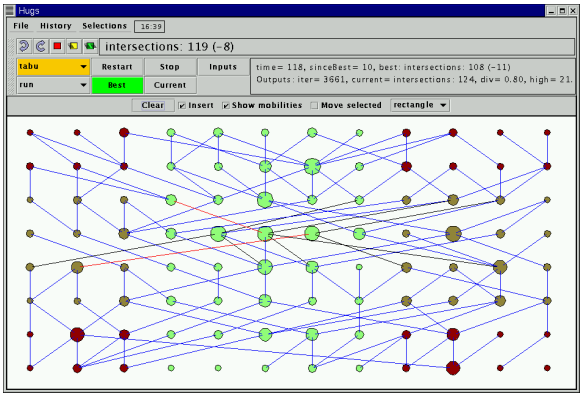


Figure 1: The Crossing Application.

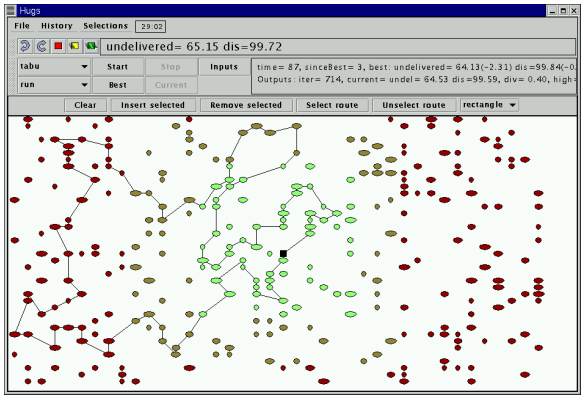


Figure 2: The Delivery Application.

the search space [2, 17, 20].

This paper is organized as follows. We first discuss our four current applications and introduce terminology to describe them uniformly. We then present our primary method of guiding search and discuss several examples of the various ways in which it can be used. We then present an overview of all user actions in our system, and then give an overview of our toolkit by describing how a new application can be made with it. We conclude with discussions of related and future work.

2. APPLICATIONS AND TERMINOLOGY

In this section, we briefly describe four applications of our system, which will serve as examples to describe the functionality of our system in the remainder of this paper. We also introduce abstractions that allow a uniform description of these and future applications.

The *Crossing* application is a graph layout problem [9] in which the goal is to arrange nodes so as to minimize the number of intersections between edges. The problem consists of m levels, each with n nodes and edges connecting nodes on adjacent levels. The goal is to re-arrange the nodes within their level (i.e., it is legal to move nodes horizontally but not vertically) so as to minimize the number of intersections between edges. An example is shown in Figure 1.

The *Delivery* application is a variation of the traveling-salesman problem in which there is no requirement to visit every city [10]. Instead the goal is to deliver as many packages as possible, without driving more than a given maximum distance. A problem consists of a set of customers, each at a fixed geographic location with a given number of requested deliveries, a starting point, and a maximum distance that can be traveled. In Figure 2, the starting point is the black square in the middle of the screen, and the customers are represented by ovals. The size of the oval is proportional to the number of orders the customer has placed. The primary goal is to fulfill as many requests as possible without exceeding the distance limitation. We treat this as a minimization problem by having the goal be to minimize the number of unfulfilled requests. The secondary goal is to minimize the distance traveled; but it is always better to fulfill more requests than to travel less. There are no time constraints or capacity limitations. The problem is quite challenging, however, because it involves both subset selection and route minimization.

The *Protein* application is a simplified version of the protein-folding problem, using the hydrophobic-hydrophilic model introduced by Dill [8]. In this model, the input is a sequence of amino acids representing a protein. Each amino acid is labeled as either hydrophobic or hydrophilic. (Under this model, the input could simply be a binary string representing whether each acid in the sequence is hydrophobic or hydrophilic.) The sequence of amino acids must be placed on a given geometry so that there are no overlapping amino acids and so that adjacent amino acids in the sequence are adjacent. Here we consider the geometry of the two-dimensional grid. The goal is to maximize the number of adjacent hydrophobic pairs, which corresponds to finding the minimum-energy configuration corresponding to the protein. Even with the restriction of counting only hydrophobic interactions on a two-dimensional lattice the problem is NP-hard [7]. Our framework easily applies to variations such as triangular grids or more complex models of interactions between adjacent amino acids. Even the best known heuristic algorithms do not appear capable, in general, of solving problems based on sequences of one hundred amino acids [4]. An example is shown in Figure 3.

The *Jobshop* application is a widely studied task-scheduling problem. The general problem is extremely broad: there is a set of n jobs to be processed on a set of m machines in some order so as to minimize some function on the completion times. In the variation we consider, each job is composed of m operations (one for each machine) that must be performed in a specified order, with the time for the ith job on the jth machine given by t_{ij}. Operations are not allowed to overlap on a machine, and the operations assigned to a given machine can be processed in any order. We seek to minimize the time that the last job finishes. (In other variations, there may be precedence constraints, so that certain jobs must be processed on certain machines before others. Alternatively, not every job may need to run on every machine. Our framework can easily be extended to handle such variations.) Of course this problem is NP-hard and indeed even cases where $n = m = 15$ are extremely difficult to solve in practice. For more on the history of this problem, see [1, 3]. An example of our application is shown in Figure 4.

We have found that these applications benefit from the use of a projected, tabletop display shown in Figure 5, which

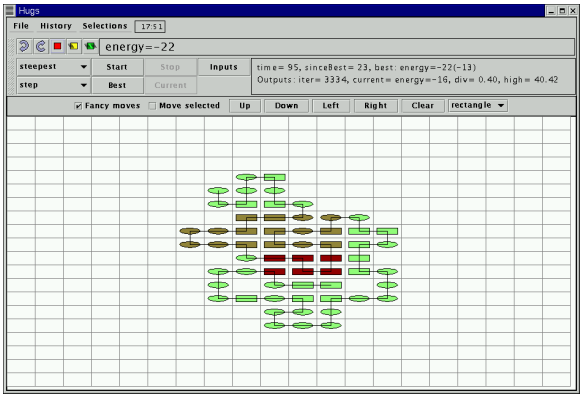


Figure 3: The Protein Application.

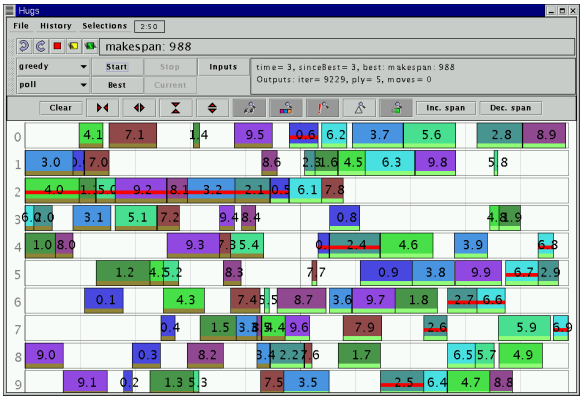


Figure 4: The Jobshop Application.

we call the Optimization Table. We project an image down onto a whiteboard. This allows users to annotate candidate solutions by drawing or placing tokens on the board. In addition, several users can comfortably use the system together.

2.1 Terminology

To describe these applications uniformly, we refer to *problems*, *solutions*, *moves*, and *elements*. A *problem* is an instance of the type of problem being optimized. For example, a Protein problem consists of a sequence of amino acids. A Delivery problem specifies the location and number of orders of each customer, the starting point, and the maximum allowed distance.

The goal of optimization is to find the best *solution* to the given problem. A Protein solution, for example, is a location in the 2D grid for each amino acid. A Delivery solution is a sequence of customers. We assume that for each application there is a method for comparing any two solutions and that for any two solutions, one is better than the other or they are equally good. As mentioned in the introduction, however, we assume that this total ordering merely approximates the real-world constraints and preferences known by the users. Additionally, for most applications, it is possible to create *infeasible* solutions which violate some of the constraints of the problem. For example, a Delivery solution

Figure 5: The Optimization Table.

may exceed the distance constraint, or a Protein solution may not describe a proper path.

For each application we have designed a set of possible *moves*, or transformations on solutions. Applying a move to a solution produces a new solution. For example, in the Jobshop, Crossing, and Delivery applications, one possible move is to swap two adjacent operations, nodes, or customers, respectively. For the Delivery applications, other types of moves include adding or removing customers from the current route.[2]

Finally, we assume that each problem contains a finite number of *elements*. The elements of Crossing are the nodes, the elements of Delivery are the customers, the elements of Protein are the amino acids, and the elements of Jobshop are the operations. Each move is defined as altering some subset of the elements of its problem. In general, it is straightforward to define the problems and solutions for the applications. However, there are often several options available for defining the set of possible moves and which elements they alter. Part of designing an effective interactive optimization system is choosing how to model the moves. As we discuss below, these decisions can have a significant impact on the behavior of our interactive optimization system. Similarly, designing a good set of transformations on solutions has a significant impact on fully automatic, heuristic search algorithms as well.

3. VISUAL CONTROL OF SEARCH ALGORITHMS

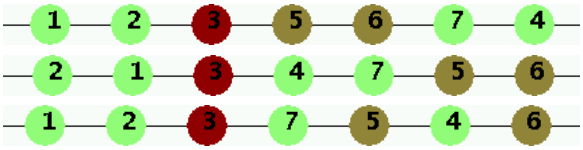
In this section we describe a general mechanism that allows users to visually annotate elements of a solution in order to guide a computer search to improve this solution. Each element can be assigned a high, medium, or low *mobility*. Roughly speaking, the search algorithm is focused on altering the high-mobility elements and is forbidden from altering any of the low-mobility elements. It can alter medium-mobility elements only in service of altering the high-mobility ones. The formal definition of mobility is that the search algorithm is only allowed to explore solutions that can be reached by applying a sequence of moves to the current solution such that each move alters at least one high-mobility element and does not alter any low-mobility elements. Thus, the semantics of mobilities in any application relies on the

[2]The moves in the Protein application are more complicated than the moves of the other applications and are beyond the scope of this paper.

set of possible moves that are defined for that application, as well as which elements of the problem each move is defined as altering.

We demonstrate the concept of mobility with a simple example. Suppose the problem contains seven elements and the possible solutions to this problem are all possible orderings of these elements. Further assume that the only allowed move is to swap two adjacent elements. A swap of two elements, of course, is defined as altering only those two elements. Suppose, the current solution is as follows, and we have assigned element 3 low mobility (shown in dark gray), element 5 and 6 medium mobility (shown in medium gray), and the rest of the elements have high mobility (shown in light gray):

A search algorithm would be allowed to swap any pair of adjacent elements only if at least one has high mobility and neither have low mobility. The search algorithm could thus explore the space of solutions that are reachable by a series of such swaps, including:



Note that setting element 3 to low mobility essentially divides the problem into two separate subproblems, both of which are much smaller than the original one. Also, while medium-mobility elements can change position, their relative order cannot be changed, e.g., it is impossible to move element 6 before element 5. In fact, for this simple example, there are only 12 possible solutions given these mobility settings. Without mobilities, there are 7!=5040 possible solutions. Thus, these mobilities reduce the search space by a factor of 420. This simple example demonstrates the different effects of each of the three mobility settings.

For more sophisticated moves, such as inserting an element into a new location, we distinguish between the *mover* element, e.g., the element being relocated, and the *moved* elements, e.g., the elements that are shifted to accommodate the relocation. The optimizer is only allowed to perform moves in which the mover element has high mobility and the moved elements have high or medium mobility.

In some cases, the question of which elements are altered by a move is a design choice for the developer. For example, in Delivery, inserting a customer into the route could reasonably be said to alter all customers in the route after the inserted customer, since they are now appear later in the route. This would mean that no new customers could be inserted before any customer with low mobility. Instead, we defined a move as altering only those customers which are removed or added to the route or have at least one new neighbor on the route after the move is performed.

Reducing the elements to three different types allows a natural visualization interface that provides the user a great deal of control. We describe two basic ways that the mobilities and the visual interface can be used to enhance the search for solutions. The first involves *focusing* the search on a restricted portion of the space. The second involves *modifying* a solution to pull the computer into a new part of the state space. We demonstrate the use of mobilities with examples from the Delivery application.

Suppose that the search algorithm runs for some time without focus, i.e., all elements are set to high mobility. The search is likely to quickly reach a state where there are few improving moves, and hence improvements will be difficult to achieve if the number of possible moves is sufficiently large. The users can assign mobilities to the customers in the problem to focus the computer search on areas which they think have the most potential for further optimization. Consider the three regions of customers shown in Figure 2 (differentiated by the same shades of gray used in the example above). One region of customers is set to low mobility, indicating that the users do not think these customers should be added, removed, or re-positioned in the current route. Another region of customers are given high mobility, indicating the users think the solution can be improved by altering these elements. The remaining customers have been given medium-mobility, so as not to overly constrain the movement of the high-mobility customers. By assigning the mobilities, the users have essentially defined a much smaller, and thus more tractable problem to work on. If the users have indeed identified promising regions of the search space, the computer will be more likely to improve the solution than with an unfocused search, given a fixed amount of computational effort.

As we describe below, users can also manually modify the current solution. Modifying the current solution can be particularly useful when the search seems to be caught in a local minimum. In such a case, it may be extremely difficult for the search program to move away from this local minimum in order to find a better solution. The user, however, can force the search engine to explore new areas of the search space by manually modifying the solution and restarting the search algorithm.

The mobility metaphor greatly enhances this ability. For example, in the Delivery problem, the search algorithm often finds a solution that makes deliveries to a small set of customers that are close together but fairly far from the rest of the customers on the route. Often, the human user can see that the distance driven to reach these customers could probably be better spent servicing customers closer to the main route. For many search strategies it may be difficult to escape such a local minimum, since it requires removing several deliveries from the current path, any one of which only reduces the amount driven by a small amount. The user can force the issue by manually removing all of these deliveries from the path and setting their mobility to low, effectively preventing them from being restored to the path. In this way the user can pull the computer into a new region of the search space.

The ability to modify the current solution and introduce mobilities is useful for introducing real-world constraints as well. For example, suppose the user notices that an important customer is not on the current route. Even though the customer is far from the current route, the user wants to try to fulfill this important customer's request, unless it means losing too many other customers. The addition of the new customer may greatly increase the length of the route, causing the current solution to be infeasible. If the user were to invoke the search algorithm without use of mobilities, the algorithm would simply remove the new customer,

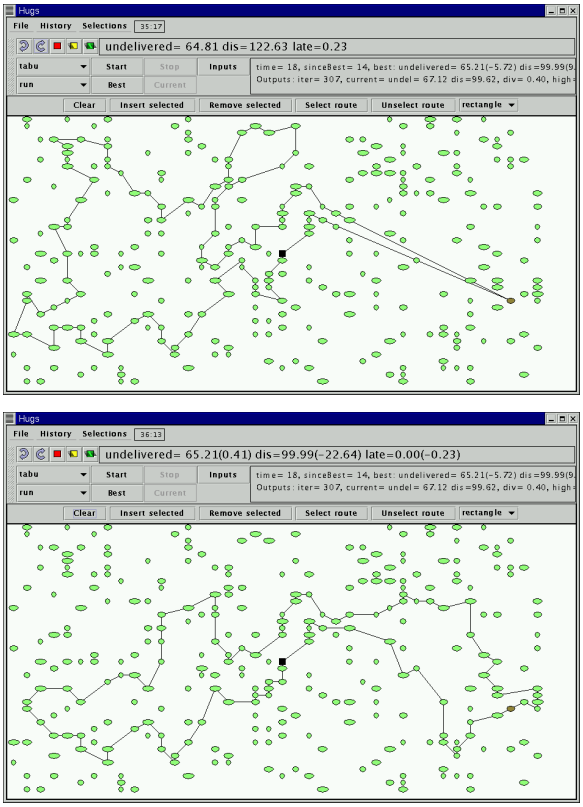


Figure 6: Use of Mobilities to Add Real-World Constraints. The top screen shot shows the Delivery application after a user has added an important customer to the route. The resulting route is too long, and thus infeasible. The user sets the mobility of the new customer to medium, so that it cannot be removed from the route and then re-invokes the search algorithm which produces the solution shown in the lower screen shot, which is feasible and optimized with the constraint posted by the user.

because this would immediately make the solution feasible again. However, as shown in Figure 6 the user sets the new customer to medium (or low), forcing the computer to re-optimize the solution without removing this customer. It is very likely that the result of the computer search will be vastly better than the user could have found by manually adding and removing customers herself.

A further advantage of this visual framework is that it is easy to take advantage of human ability to cluster. Specifically, our user interface provides controls to set the mobility of a large number of elements simultaneously in a way that naturally corresponds to visual clusters. In the Delivery problem, for example, a rectangular region can be selected, and the mobility of all elements in the region set simultaneously. Similarly, in the Protein problem a region in the two-dimensional grid can be selected and the mobility of all amino acids in that region can be set.

4. OVERVIEW OF USER ACTIONS

We now describe the full range of user actions in the HuGS framework. In our applications, the system always maintains a single, current working solution which is displayed to the users. (Potentially, other solutions could be displayed as well, though currently we do not do this.) The users try to improve the current solution by performing the following three actions:

1. manually choose a move to be applied to the current solution,
2. invoke, monitor, and halt a focused search for a better solution,
3. revert to a previous or precomputed solution.

We now describe each type of action. The users can manually modify the current solution by performing any of the possible moves defined for the current application on the current solution, such as repositioning an operation in the Jobshop application. In many of our applications, a single user action on the GUI can invoke several moves. In the Delivery application, for example, the user can select any number of customers and remove them all with a single button press.

Users can also invoke a computer search for a better solution. The search algorithm starts from the current solution and explores the space of solutions that can be reached by applying moves which are allowed given the mobility assignments as described above. The users can invoke a variety of different search algorithms. Currently, we provide steepest-descent and greedy exhaustive search algorithms and a popular heuristic algorithm, called tabu search [12, 13]. Both exhaustive algorithms first evaluate all legal moves, then all combinations of two legal moves, and then all combinations of three moves and so forth. The steepest-descent algorithm keeps searching deeper and deeper for the move that most improves the current solution. The greedy algorithm immediately makes any move which improves the current solution and then restarts its search to try to improve the solution that results from applying that move. The tabu algorithm repeatedly makes the best possible move from the current solution, which may yield a worse solution. A variety of mechanisms are used to prevent tabu from immediately backtracking and to prefer altering elements that have not been altered recently. Our initial experience has been that tabu search outperforms exhaustive search but it seems useful to provide multiple search algorithms to the users.

After the users have invoked a search algorithm, they can monitor its progress in order to decide when to halt it. A text display shows the score of the best solution the search has found and how many seconds ago this solution was found. At any time, the user can query the search algorithm for either the best solution found so far or the current solution it is considering. This solution becomes the current visualized solution of the system. While the search is running the user can modify the current visualized solution or reassign mobility values to problem elements. The user can restart the search from these current settings, or halt the search.

While the search algorithm is running, the users can select from a variety of search-visualization modes. The most efficient mode is to let the search algorithm run in the background without updating the current visualized solution.

The users can also observe the search more directly. The users can put the search into "auto" mode, in which every solution the search considers is displayed, or "poll" mode in which the computer is polled periodically for its current solution, or "step" mode in which the computer waits for the user to press a button before moving on to the next solution it considers. These modes are useful for developing applications as well as for learning about how the system and search algorithms work.

Finally, the third type of user action is to revert to a previous solution. The system maintains a history of previous solutions, which can be browsed and adopted by the users. The GUI also provides menu commands to quickly undo or redo recent moves, as well as revert to the best solution seen so far. Additionally, the users can browse and adopt a set of solutions that were precomputed by the search algorithms prior to the interactive optimization session.

5. CREATING A NEW APPLICATION

We now discuss what is required to produce an interactive optimization application for a new problem using our software platform, with emphasis on what is required of the visualization component.

The majority of the code in these applications is shared by all of them, and thus could also be used by a new application. Generic code is used to maintain the current working solution, the mobilities, and the history. The file Input/Output, including saving and loading of problems and solutions, and logging user behavior, are also performed by generic code. Furthermore, all our applications use the same implementation of the search algorithms and the GUI's for invoking and monitoring them. Our implementation of the tabu search algorithm functions by modifying the mobility assignments. Thus, there is no additional burden on the developer of a new application in order to be able to use tabu search.

Of course, the developer of a new application must define what a problem is and what a solution is for that application.[3] Each problem instance needs to implement a function that returns all the elements of that problem. Each solution instance must be able to return an object which represents the *score* of that solution. We have built generic components for representing scores with integer or double numbers, but for some applications a new score object must be written. For example, the score of the Delivery application consists of both the percentage of unfulfilled requests and the distance traveled, and so required the creation of a new type of Score object. An instance of a score object must be able to compare itself to another instance of a score and decide if it is better, worse, or equal to that instance.

Additionally, a developer must define a set of moves which can be applied to solutions in this application. For some applications, there might be several different types of moves. Additionally, at a minimum, the user must provide a function for generating all possible single moves for a given solution. This is sufficient for the search algorithms to consider any legal combinations of moves. However, for efficiency, a developer might provide functions for directly generating combinations of moves.

[3]This involves defining classes which implement Java interfaces for a Problem class and a Solution class.

5.1 Visualization Component

Each application requires a domain-specific visualization component. From the point of view of the system, the visualization component has only three responsibilities. First, it must report any manual moves made by the user. These moves will be applied to the current solution that the system maintains. Second, the visualization component must have an update function which, when called, triggers it to display the current solution and mobilities maintained by the system. Third, the system must allow the users to select and unselect elements of the problem. The system will query the visualization component for the list of currently selected elements in order to maintain the mobilities. The users can, for example, set all the selected elements to a particular mobility, as well as reset all elements to any particular mobility.

In all of our applications, we have found it useful to visually indicate which elements have changed compared to the previous solution that was displayed. For example, we highlight which customers have changed their condition in Delivery, or which nodes have changed locations in Crossing. This is important because when the computer produces a new solution, it can be difficult (and yet important) for a user to identify what has changed.

Beyond the above-mentioned system requirements, of course, the usefulness of any interactive optimization system depends on the quality of the visualization. This challenge differs from application to application. One unusual feature of designing visualizations for interactive optimizations, however, is that the quality of the visualization can be measured (albeit, imperfectly). It is reasonable to compare two visualizations by running a series of experiments in which people work on the same problems for the same amount of time, with the only difference being which of the two visualizations is used. If people are able to produce more optimal solutions with one visualization than another, it can be said to be superior for this task. While this does not measure all the benefits that one visualization might offer over another, it does quantify the performance of the visualization on one of its most important tasks.

All of the applications described in this paper consider problems with, at most, several hundred elements. A future research direction is to explore techniques for applying interactive optimization to larger problems, in which people cannot view all the elements at once on the computer screen.

6. RELATED WORK, FUTURE WORK, AND CONCLUSIONS

The work described in this paper builds on previous work on the Human-Guided Search paradigm [2]. The framework for user actions and the idea of high, medium, and low mobilities has been previously described in the context of a particular application (capacitated vehicle routing with time windows). The novel aspects of this paper include the uniform approach to interactive optimization, the description of the Java software, the four applications, and the interaction mechanisms for monitoring the search algorithms while they are in progress (e.g., the ability to query for the current and best solutions, and the various modes of visualizing the search). In related work, our human-guidable tabu search algorithm is described in detail in [15]. This work also provides experimental confirmation that people can effectively

guide our tabu search algorithm.

Interactive optimization systems have been built for a variety of applications, including space-shuttle scheduling [6], graph drawing [17], graph partitioning [16], vehicle routing [23, 5, 2], and constraint-based drawing [18, 14, 11, 19]. To our knowledge, however, no single interactive optimization approach has previously been applied to such a varied set of optimization problems as we have described here.

An approach similar to HuGS was used to create an interactive graph-drawing system, which allows users to post constraints on the final solution, as well as to perform manual moves and to freeze graph elements in their current position [17]. Other research has explored alternative methods for dividing the work between human and computer in cooperative optimization or design. In the space-shuttle scheduling application [6], for example, the computer detects and resolves conflicts introduced by the user's refinements to a schedule. In interactive constraint-based drawing applications e.g., [18, 14, 11, 19], the user imposes geometric or topological constraints on a nascent drawing such that subsequent user manipulation is constrained to useful areas. The interactive-evolution paradigm, which has primarily been applied to design problems, offers a different type of cooperation [21, 22]. In this approach, the computer generates successive populations of novel designs based on previous ones, and the user selects which of the new designs to accept and which to reject. Thus novel designs evolve, subject to user-supplied selection criteria. This paradigm has found use in various computer-graphics applications. The HuGS paradigm differs significantly from the iterative-repair, constraint-based, and interactive-evolution paradigms in affording the user great involvement and greater control of the optimization/design process.

The primary contribution of this work has been to present a general approach for visualizing and controlling an interactive search process. We have shown how this approach can be applied to four diverse applications. Our future work includes developing extensions of our system which allow distributed human-guided search in that multiple people can collaboratively work on the same optimization problem from remote locations, as well as take advantage of distributed computational resources.

7. REFERENCES

[1] E. Aarts, P. van Laarhoven, J. Lenstra, and N. Ulder. A computational study of local search algorithms for job-shop scheduling. *ORSA Journal on Computing*, 6(2):118–125, 1994.

[2] D. Anderson, E. Anderson, N. Lesh, J. Marks, B. Mirtich, D. Ratajczak, and K. Ryall. Human-guided simple search. In *Proc. of AAAI 2000*, pages 209–216, 2000.

[3] D. Applegate and W. Cook. A computational study of the job-shop scheduling problem. *ORSA Journal on Computing*, 3(2):149–156, 1991.

[4] U. Bastoalla, H. Frauenkron, E. Gerstner, P. Grassberger, and W. Nadler. Testing a new Monte Carlo algorithm for protein folding. *Proteins: Structure, Function, and Genetics*, 32:52–66, 1998.

[5] John W. Bracklow, William W. Graham, Stephan M. Hassler, Ken E. Peck, and Warren B. Powell. Interactive optimization improves service and performance for Yellow Freight system. *INTERFACES*, 22(1):147–172, 1992.

[6] S. Chien, G. Rabideau, J. Willis, and T. Mann. Automating planning and scheduling of shuttle payload operations. *J. Artificial Intelligence*, 114:239–255, 1999.

[7] P. Crescenzi, D. Goldman, C. Papadimitriou, A. Piccolboni, and M. Yannakakis. On the complexity of protein folding. *Journal of Computational Biology*, 5(3):409–422, 1998.

[8] A. K. Dill. Theory for the folding and stability of globular proteins. *Biochemistry*, 24:1501, 1985.

[9] P. Eades and N. C. Wormald. Edge crossings in drawings of bipartite graphs. *Algorithmica*, 11:379–403, 1994.

[10] D. Feillet, P. Dejax, and M. Gendreau. The selective Traveling Salesman Problem and extensions: an overview. TR CRT-2001-25, Laboratoire Productique Logistique, Ecole Centrale Paris, 2001.

[11] Michael Gleicher and Andrew Witkin. Drawing with constraints. *Visual Computer*, 11:39–51, 1994.

[12] F. Glover. Future paths for integer programming and links to artificial intelligence. *Computers & Operations Research*, 13:533–549, 1986.

[13] A. Hertz, E. Taillard, and D. de Werra. Tabu search. In E. Aarts and J. K. Lenstra, editors, *Local Search in Combinatorial Optimization*, chapter 5, pages 121–136. John Wiley & Sons, 1997.

[14] A. Heydon and G. Nelson. The Juno-2 constraint-based drawing editor. *Digital Systems Research Center Research Report 131a*, December 1994.

[15] G. W. Klau, N. Lesh, J. Marks, and M. Mitzenmacher. Human-guided Tabu search. *In submission*, 2002.

[16] N. Lesh, J. Marks, and M. Patrignani. Interactive partitioning. *Graph Drawing*, pages 31–36, 2000.

[17] H.A.D. do Nascimento and P. Eades. User hints for directed graph drawing. *To appear in Proc. of the Symposium on Graph Drawing*, 2001.

[18] Greg Nelson. Juno, a constraint based graphics system. *Computer Graphics (Proc. of SIGGRAPH '85)*, 19(3):235–243, July 1985.

[19] K. Ryall, J. Marks, and S. Shieber. Glide: An interactive system for graph drawing. In *Proc. of the 1997 ACM SIGGRAPH Symposium on User Interface Software and Technology (UIST '97)*, pages 97–104, Banff, Canada, October 1997.

[20] S. Scott, N. Lesh, and G. W. Klau. Investigating human-computer optimization. *To appear in Proc. of CHI 2002*, 2002.

[21] Karl Sims. Artificial evolution for computer graphics. *Comp. Graphics (Proc. of SIGGRAPH '91)*, 25(3):319–328, July 1991.

[22] Stephen Todd and William Latham. *Evolutionary Art and Computers*. Academic Press, 1992.

[23] C.D.J Waters. Interactive vehicle routeing. *Journal of Operational Research Society*, 35(9):821–826, 1984.