BlockTree – Pedagogical Information Visualization for Heuristic Search

David Furcy, Andrew Jungwirth, and **Thomas Naps**
University of Wisconsin Oshkosh
Department of Computer Science
Oshkosh, Wisconsin 54901
{furcyd,jungwa45,naps}@uwosh.edu

Abstract

BlockTree is an information visualization tool that helps students explore nuances of heuristic search algorithms, such as A* and WA*. The goal of such exploration is to achieve a level of understanding that is deeper than merely being able to trace the algorithm on an artificially small domain. Instead, BlockTree presents a single execution snapshot of a heuristic search tree encompassing up to hundreds of thousands of nodes and offers a robust GUI that encourages viewing such a tree from a variety of perspectives. By experimenting with the visualizations generated by the problem domains already incorporated into BlockTree, students are exposed to the same large-scale issues that AI practitioners face on a regular basis. Beyond these built-in problem domains, BlockTree offers an extensible object-oriented framework that can be used as the basis for programming assignments. As students implement additional domains in BlockTree, their solutions are automatically visualized in the same fashion as the built-in domains.

Motivation and learning objectives

The motivation for the development of the BlockTree tool[1] described in this paper can best be grounded in the well-known Bloom's taxonomy of levels of understanding (Bloom & Krathwohl 1956). This taxonomy structures a learner's depth of understanding along a linear progression of six increasingly sophisticated levels:

Level 1: *The knowledge level*. Characterized by mere factual recall with no real understanding of the deeper meaning behind these facts.

Level 2: *The comprehension level*. The learner is able to discern the meaning behind the facts.

Level 3: *The application level*. The learner can apply the learned material in specifically described new situations.

Level 4: *The analysis level*. The learner can identify the components of a complex problem and break the problem down into smaller parts.

Level 5: *The synthesis level*. The learner is able to generalize and draw new conclusions from the facts learned at lower levels.

[1]The BlockTree tool is available for download at *http://www.uwosh.edu/departments/computer_science/BlockTree/index.php*

Level 6: *The evaluation level*. The learner is able to compare and discriminate among different ideas and methods. By assessing the value of these ideas and methods, the learner is able to make choices based on reasoned arguments.

Mapping Bloom's taxonomy to heuristic search

In the particular context of the tool presented here, we want to distinguish student levels of understanding with respect to heuristic search algorithms. Our perspective on a mapping between Bloom's taxonomy and what we expect of our students in their comprehension of heuristic search is the following:

Knowledge Level: A student should know (memorize) that A* uses a priority queue data structure in which the priority $f(n)$ of a node n is given by the formula $g(n) + h(n)$.

Comprehension Level: A student should understand the operations of the priority queue ADT, as well as the meanings of the g- and h-values in a search problem.

Application Level: A student should be able to apply (trace) the A* algorithm to a new domain already described as a graph of nodes and associated g- and h-values.

Analysis Level: A student should be able to re-frame a given AI problem described in everyday terms into a graph search problem. This level involves being able not only to reformulate the problem, but also to identify both the states (that is, nodes) and the actions (edges) of the problem.

Synthesis Level: A student should be able to use the knowledge acquired at the lower levels to generate a new heuristic function. For example, having reformulated a task as a graph-search problem, a student should be able to apply their knowledge of the A* algorithm and its use of h-values to design a heuristic function such as the Manhattan Distance heuristic.

Evaluation Level: A student should be able to compare the strengths and weaknesses of different solutions. In the context of A* search, this means evaluating different heuristic functions in terms of their accuracy, admissibility, and computational cost. Similarly, a student should be able to compare the strengths and weaknesses of different

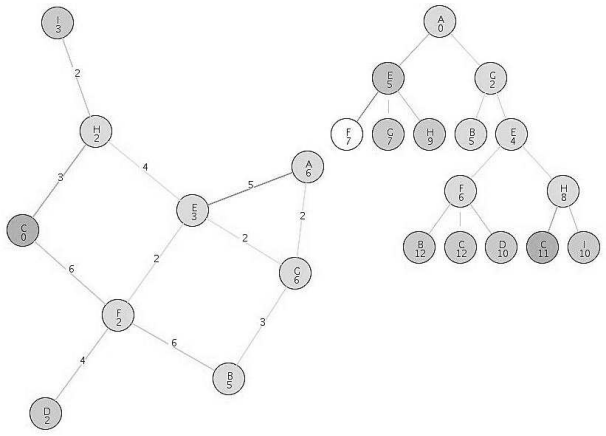

Figure 1: Algorithm visualization on a small search tree

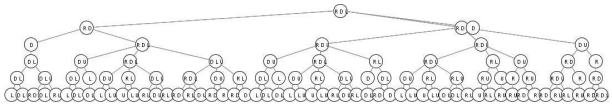

Figure 2: Standard tree visualization of a larger tree

search algorithms, for example, blind search techniques such as breadth-first search (BFS) versus heuristic search techniques such as A* or Weighted A* (WA*).

Visualization for deeper understanding

The benefits of *algorithm visualization*, in which the successive steps of an algorithm are graphically portrayed to help students learn an algorithm as a "recipe", have been well documented (Naps 2005). For instance, the JHAVÉ visualization environment, available on the Web at *http://jhave.org* provides an infinite reservoir of small example graphs on which students can practice tracing the A* search algorithm (see Figure 1).

However, by our mapping of Bloom's taxonomy to heuristic search, such small-scale visualizations emphasize only the lower levels of the taxonomy. Simply being able to trace the steps of an algorithm is a pre-requisite for the deeper levels of understanding in the taxonomy. As educators, we are always concerned with developing our students' understanding at these deeper levels. BlockTree helps us accomplish this by using large-scale *information visualization* rather than algorithm visualization. Instead of showing execution steps from a small search space at a micro-level, BlockTree captures in one picture the entire execution of a heuristic search conducted over a relatively large search space (several hundred thousand nodes for the 8-Puzzle). Students can explore that picture from a variety of perspectives to discover answers to questions such as:

- What are the differences among breadth-first, A*, and WA* searches[2] when looking for a low-cost solution?
- What are the differences among these three algorithms in terms of efficiency? That is, how many nodes did each algorithm have to expand before it found the solution?
- How do we change the behavior of the A* algorithm by using different heuristics?
- In the WA* algorithm, how do the number of nodes expanded and the length of the solution path vary as we increase or decrease the weight multiplier?

Students' abilities to cope with the conceptual nuances implicit in questions like these establish a much higher metric for their understanding on the Bloom scale. Indeed, the comparisons we have them address on substantial search spaces – differences in algorithms, heuristics, and tuning of weights – require many aspects of Bloom's synthesis and evaluation levels.

Visualization for programming assignments

BlockTree's collection of built-in problems, algorithms, and heuristics help us design exploratory exercises in which the synthesis and evaluation levels of understanding can be pursued. However, the analysis level requires that, given a new problem, the student can break it down into components amenable to solution by heuristic search. Toward that end, we provide an object-oriented framework that allows students to extend BlockTree by writing Java classes. Extending BlockTree can be done in three ways:

- Develop a new heuristic for a provided puzzle and algorithm.
- Develop a new algorithm for a provided puzzle, *e.g.*, Dijkstra's algorithm or beam search.
- Develop an implementation of a new puzzle, *e.g.*, Rush Hour. Here the student must demonstrate a complete understanding at Bloom's analysis level. The student must decide how to represent a state of the puzzle, how to generate successors from that state, and how to portray a graphic rendition of that state.

Because BlockTree is based on sound software engineering principles, such extension of the framework can be done in a fashion that minimizes the amount of coding students have to do before they actually begin exploring the effectiveness of their solution. This is important because, as students quickly learn, just getting code to work correctly is only half of the battle in AI programming. Once it works, considerable experimentation must be done to "tune it" for better performance.

Moreover, because the student's code hooks directly into the information visualization capabilities of BlockTree, the exploration they do with their implementations is done in the context of the same GUI and visual feedback that we have had them use in exploring the puzzles already built into BlockTree. Today's students are multi-media oriented. Hence, as noted in (Guzdial & Soloway 2002), giving them a programming assignment in which their output is expressed

[2]WA* is a version of A* in which the priority of each node n is given by $f(n) = g(n) + wh(n)$. The weight multiplier w is greater than or equal to 1 in order to make the search more greedy. A* is the special case of WA* when $w = 1$.

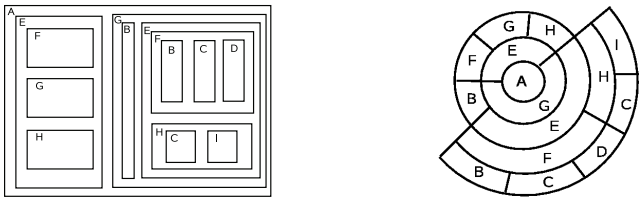

Figure 3: Treemap (left) and Sunburst (right) views of tree from Figure 1

in hard-to-decipher textual form is a sure way to foster disinterest. By leveraging off of the graphics provided by BlockTree, we provide a very natural motivation for them.

BlockTree visualization in context

Figure 1 shows that the standard hierarchical graph picture of a search tree, with circular nodes connected by edges, is adequate for artificially small trees. However, Figure 2 shows that, even for a moderately-sized tree with a branching factor of three and depth five, the amount of space used to show the hierarchical structure of the tree renders the contents of nodes totally illegible. Moreover, this illegibility is pervasive over the entire tree. Even at the upper levels of the tree, where very few nodes appear, the content of the nodes cannot be determined.

The inadequacy of standard tree diagrams in portraying large trees has been addressed by the information visualization community through the development of *treemaps* (Shneiderman 1992). In a treemap, the root of a (sub)tree is portrayed as a partitioned rectangle that encloses all of its subtrees. The greater the size of a subtree, the greater the space allocated to it in its partitioned area. A treemap view of the tree from Figure 1 appears on the left in Figure 3.

Unlike standard tree diagrams, treemaps are space-filling visualizations that have been used to great advantage in visualizing disk storage in hierarchical file systems. However, concurring with observations in (Coulom 2002), we found that treemaps fail to capture important structural information when visualizing heuristic search algorithms. First, treemaps tend to obscure the hierarchical structure of large trees with many levels, making it hard, for example, to identify all the nodes at a given level in the tree. Second, treemaps are designed to fill up the whole space of the enclosing rectangle (the root node of the tree), thus preventing the natural use of empty space to visualize the effect of pruning in heuristic search.

The *Sunburst* visualization tool (Stasko 2000) addresses both issues with a circular space-filling technique that places the root in the center and groups all other nodes by levels in concentric circles around the root. The sunburst view of the tree in Figure 1 appears on the right in Figure 3. Unfortunately, by their circular nature, sunburst trees lose the traditional top-down view of heuristic search trees in which the children of a node always appear below it in the tree. Furthermore, the computation of hundreds of thousands of circular slices using trigonometric functions on floating point numbers may significantly hamper the continuous interac-

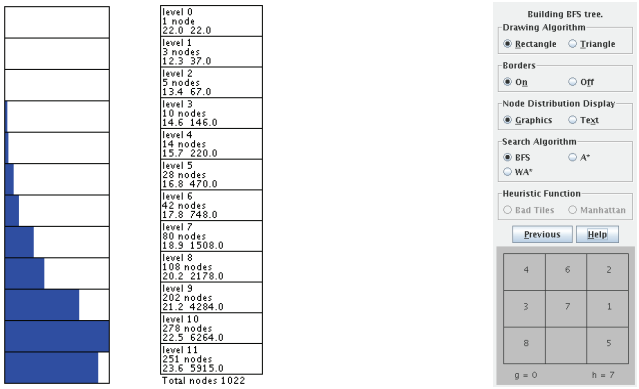

Figure 4: Graphical and Textual views of node distribution information (left) and configuration panel (right)

tions we expect our students to engage in when using a pedagogical tool.

Our BlockTree technique combines the advantages of both standard and sunburst trees in naturally reflecting the top-down, hierarchical structure of a tree with some of the space-filling advantages of treemaps. BlockTree uses a rectangular view like treemaps but retains the hierarchical perspective of a standard tree diagram. The basic layout rule is that the horizontal space allocated to all of a node's children is the same as what is allocated to the parent. Figure 6 (left) shows a BlockTree view of the search tree created by a breadth-first search of the 8-puzzle whose initial state is depicted in the configuration panel of Figure 4.

User interaction with BlockTree

The window for our BlockTree visualization tool is divided into three separate panes, as shown by the numeric labels in Figure 6 (left). These panes are the statistics pane (1), the BlockTree pane (2), and the configuration pane (3).

The **statistics pane** shows how the nodes are distributed among levels of the tree. Using the *Node Distribution Display* selection box on the configuration pane, the user can toggle between the graphical display and the textual display. Figure 4 shows both displays. The graphical display shows the node density at each level using horizontal bars. At the level with the greatest number of nodes, the bar is completely filled (level 10 in Figure 4). The bars for each of the other levels in the tree are filled corresponding to the number of nodes in that level divided by this maximum number of nodes. Thus, the bars reflect the number of nodes per level as a percentage of the level with the greatest number of nodes. In the textual display, several pieces of information are provided for each level in the tree, including the depth of the level and the number of nodes in that level.

The **BlockTree pane** provides various mechanisms for interacting with the tree. Clicking (either mouse button) and dragging within this pane moves the tree's graphics within the pane. This is useful if the tree is deep enough that the lower nodes are clipped at the bottom of the window. The statistics pane also moves vertically along with the

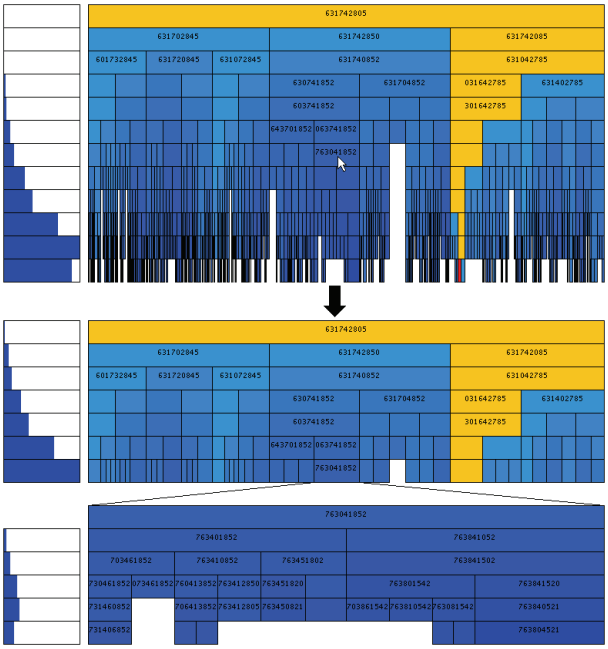

Figure 5: Left clicking to expand node

tree so that each section of the statistics pane remains properly aligned with the corresponding level in the tree. Left-clicking on a node in the BlockTree pane causes the node to expand as shown in Figure 5 [3]. The clicked node expands outward to fill the entire width of the tree and is displayed similarly to the root node. Notice that the children of all other nodes in the same level as the node that was clicked are no longer shown after the node has been expanded. The statistics pane also changes to reflect only the nodes that are currently visible. After a node is clicked, left-clicking on any other node below it causes that node to be expanded as well. In this way, multiple expansions can be made at the same time to display the lower levels of a deep tree – hence filling the space below a node in a way that the viewer decides is most advantageous for their exploration.

Right-clicking on a node in the tree shows a detailed view of that node. This view, added under the buttons on the configuration pane, displays a graphical representation of the corresponding state (see Figure 4). The node that was right-clicked is also highlighted in another color to indicate that the detailed view is currently showing its state. Holding the shift key while right-clicking on a node rebuilds the tree using that node as the new root: The current settings from the configuration pane are used to construct a new tree rooted at the indicated node, and this new tree is drawn in the BlockTree pane. It is also possible to zoom in or out on the whole tree using the mouse wheel.

The **configuration pane**, shown in detail in Figure 4, allows the user to set the options for both the BlockTree and statistics panes. At the top is a text field that relays messages to the user when the program is processing user commands. Below it is the *Drawing Algorithm* selection box which allows the user to switch between the BlockTree view described in this paper, and a Triangle view, which we do not discuss in this paper for lack of space. The *Borders* selection box is used to toggle on and off the black borders around the nodes in the tree. In some cases, turning the borders off makes it possible to see the colors of the nodes deeper in the tree than when the borders are on. The *Node Distribution Display* selection box switches between the textual and graphical views in the statistics pane, as described above. Clicking one of the radio buttons within the *Search Algorithm* selection pane changes the search algorithm used to construct the tree. Some search algorithms may require user input. In this case, an input box pops up for the user to enter the required algorithm parameter(s). Similarly, the *Heuristic Function* box allows the user to change the heuristic function that is used to evaluate nodes when searching the tree. When a new search algorithm or heuristic function is selected, the tree is rebuilt from the current root using the selected algorithm and heuristic function, and the new tree is drawn in the BlockTree pane. The available search algorithms and heuristic functions are dependent on and dynamically updated according to the domain that is being visualized.

Every time the user rebuilds the tree from a new root using the shift+right click command, the old root is stored in a stack. Clicking the *Previous* button redraws the tree from the most recent previous root using the currently selected algorithm, heuristic function, and rendering settings. This feature may be useful for thoroughly exploring a domain, algorithm, or heuristic function.

When the *Help* button is clicked, the system opens a separate window (not shown) to describe the available user commands. Finally, underneath these buttons is the detail panel that displays a graphical representation of a node when it is right-clicked in the BlockTree pane.

Visualizing heuristic search with BlockTrees

Our BlockTree visualization package is a general and extensible pedagogical tool for heuristic search. While students and instructors can easily plug in their own problem domains, our package also readily includes several implemented domains, namely the flashlight, sliding-tile, and Rush Hour domains, to be used for in-class demonstrations, individual practice exercises, or programming assignments. In the interest of space, this paper only refers to the sliding-tile domain with 8 tiles. In addition, our BlockTree package facilitates the comparison of several heuristic (or h) functions for each domain. Here, we focus on a single 8-Puzzle problem instance characterized by the initial state shown at the bottom of Figure 4 (right). Figures 6 and 7 contain the BlockTree visualizations of (*i.e.*, all the expanded nodes in) the search trees rooted at this state and built with BFS, A* search with the misplaced tiles heuristic, A* search with the Manhattan Distance heuristic, and WA* search with the Manhattan Distance heuristic and $w = 2$.

As discussed above, our BlockTree package targets several, often higher levels in Bloom's taxonomy. In one of

[3] In Figures 5-7, each tree node contains its state's tile numbers in a canonical order. Orange nodes are on the solution path.

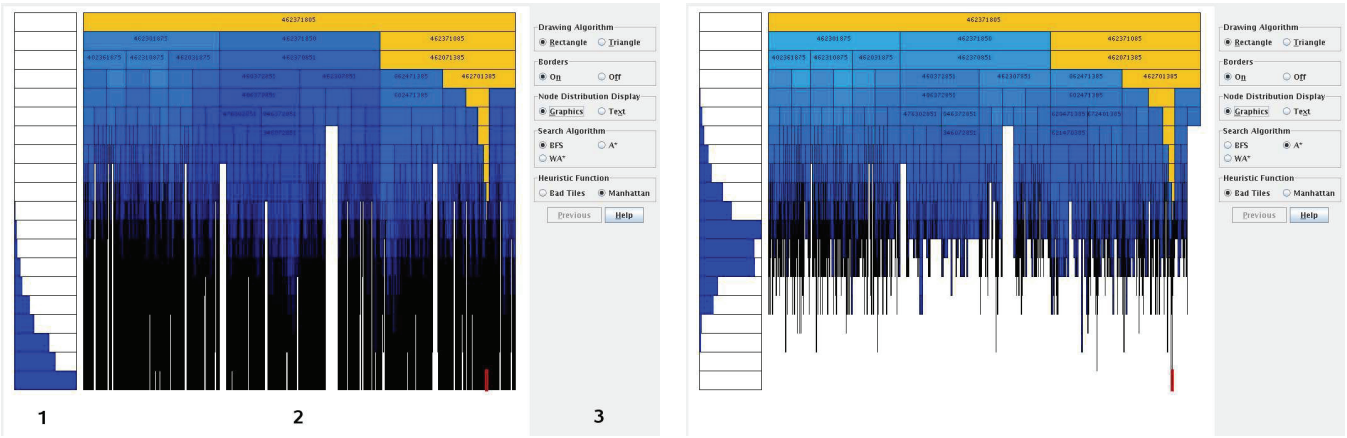

Figure 6: 8-Puzzle search trees generated by BFS (left) and A* with the misplaced tiles heuristic (right)

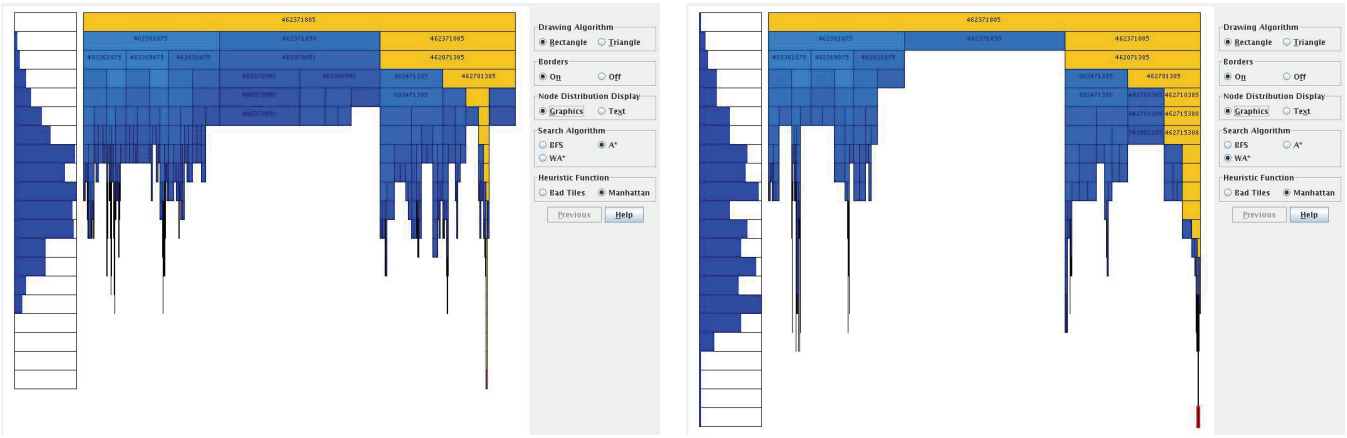

Figure 7: 8-Puzzle search trees generated by A* (left) and WA* (right), both with the Manhattan Distance heuristic

their AI programming assignments, our students must design and implement the most informed heuristic function they can devise for a given domain and search algorithm. This task, which draws on several levels of understanding with special relevance to Bloom's synthesis and evaluation levels, is facilitated by BlockTree in a number of ways.

First, we encourage students to go through an iterative process whereby they implement and evaluate their design using BlockTrees to visualize the pruning enabled by their heuristic. BFS (see Figure 6 (left)) performs no pruning and constitutes the baseline performance: Its BlockTree view is essentially free of empty spaces since BFS expands all nodes up to the level of the goal node.[4] In contrast, the A* search shown in Figure 6 (right) contains a much larger proportion of empty space, a direct visual indication of the reduced search effort (time) and memory consumption (space), since all the algorithms we discuss share the property that they store in memory all the nodes they expand.

Second, BlockTrees help students evaluate the accuracy of their new heuristic. For example, a comparison of Figures 6 (right) and 7 (left) reveals that Manhattan Distance is a more accurate heuristic in this case since it enables more pruning by A*. Students are encouraged to interleave design and visualization episodes to iteratively improve their heuristic. Students may also use the interactive features of the BlockTree package to navigate the tree, identify subtrees in which their heuristic performs poorly, understand its weaknesses, and further improve it.

Third, students can compare the efficiency of different heuristic search algorithms. For example, Figure 7 (right) shows that WA* reduces the search effort (and memory requirements) significantly over A* with the same heuristic function. Beyond visual feedback, the BlockTree package helps students quantify the performance improvements: The textual statistics pane contains the exact number of expanded nodes, namely 41,371 nodes for BFS, 2,534 nodes for A* with the misplaced tiles heuristic, 399 for A* with the Manhattan Distance heuristic, and 279 nodes for WA* with the same heuristic.

Fourth, the graphical statistics pane helps students gain a qualitative appreciation for the effect of heuristic pruning. The left pane in Figure 6 (left) portrays the combinatorial explosion of the search tree with the largest level at the bottom, since no pruning occurs in BFS. In contrast, Figure 7 (left) shows how the A* search tree bulges in the middle and narrows again toward the bottom, closer to the goal, where the heuristic function is typically more accurate and the effects of pruning are more pronounced.

Finally, BlockTrees facilitate the comparative study of the quality of the solutions found by various algorithms. The

[4] The few empty spaces in the BlockTree pane of Figure 6 (left) are not due to pruning, but to either 1) tie-breaking at the last level, or 2) the fact that some states may be reached via several paths (or *transpositions*), which BFS eliminates.

perceptive student will note that the paths found by BFS and A* with both heuristics contain 18 moves or levels in the tree. However, WA*'s solution path is 20 moves long. A natural question to ask the student is why this happens even when A* and WA* use the same heuristic. If students remember (knowledge level) that both BFS and A* with an under-estimating heuristic are guaranteed to find a shortest solution, they should infer (application and synthesis levels) that the shortest path in this problem has to be 18 moves long. Thoughtful students will figure out that the only way to explain the longer solution path found by WA* (see Figure 7 (right)) is the added weight on the h values: A weight strictly larger than 1 may cause the resulting values to overestimate the true distance to the goal. Students will come out of this exercise understanding (at the highest level of Bloom's taxonomy) the trade-off exhibited by WA* between search effort and memory consumption on the one hand, and solution quality on the other. Figures not included in the paper for lack of space also show how this trade-off evolves for larger and larger values of the weight multiplier.

Writing new "plug-ins" for BlockTree

Students may extend BlockTree in three ways: adding a heuristic, adding a search algorithm, and solving a new puzzle. Each of these corresponds to extending an abstract base class or interface. To create a puzzle portrayed by BlockTree, the student must extend the BTNode class.

```
public abstract class BTNode {
    protected static Heuristic heuristic;
    public static void setHeuristic(Heuristic h);
    public void updateH();
    public abstract BTNode[] expand();
    public abstract boolean isLeaf();
    public abstract JPanel getDetail();  }
```

A BTNode has a heuristic evaluator that is established using the *setHeuristic* method. *updateH* is the method used to update the heuristic value of the node using the current heuristic. The abstract methods *expand* and *isLeaf* are used to obtain the successors of a node or to indicate that there are none. By writing the *expand* method, the student must determine the appropriate data structures to use in representing the state of the puzzle – thereby illustrating understanding at the analysis level of Bloom's taxonomy. The *getDetail* method provides the small graphic rendition of a puzzle state that is shown in the configuration panel (Figure 4).

To add a new heuristic, the student must write a class that implements the *Heuristic* interface.

```
public interface Heuristic {
    public int calculateH(BTNode node);  }
```

The *calculateH* method of the heuristic is called by the *updateH* method in the *BTNode* class each time the heuristic is evaluated.

Finally, to implement a new search algorithm, the student must extend the *Algorithm* abstract base class.

```
public abstract class Algorithm {
    public abstract BTNode run(BTNode start);  }
```

The abstract method *run(BTNode start)* performs the algorithm from the given start state and must be implemented for any classes that extend the *Algorithm* class.

Anecdotal student reactions and future plans

In our first experience using BlockTree for our junior-senior level AI course, we crafted an assignment in which students first had to answer a series of conceptual questions using the built-in flashlight and sliding-tile puzzle domains. Then, for the programming portion of the assignment, students completed the implementation of a heuristic for the Rush Hour puzzle. After the assignment was completed, we asked them to fill out a survey regarding their reactions to BlockTree. Overall their comments were very favorable, but we also observed that some students expressed a certain amount of impatience at having to cope with the sheer amount of information incorporated into the different views presented by BlockTree's display. In retrospect, this was not surprising. Information visualization has been described as "a way to answer questions you didn't know you had." (Plaisant 2004) Undergraduate students are not used to this mode of learning, but those who are patient enough to persist are often rewarded. At least one student came to this realization, commenting "Once I understood exactly what everything represented, it was very helpful." Based on such encouraging feedback, we will continue to refine and use BlockTree for our coverage of heuristic search in the AI course. We also plan to complete an extension that will provide similar capabilities for exploration of adversarial search in game trees.

Acknowledgments

We gratefully acknowledge financial support from the National Science Foundation DUE Award #0341148. Insight and inspiration came from John Stasko of Georgia Tech, whose remarks in a conversation held at a visualization workshop in the summer of 2006 helped provide the conceptual design behind BlockTree.

References

Bloom, B. S., and Krathwohl, D. R. 1956. *Taxonomy of Educational Objectives; the Classification of Educational Goals, Handbook I: Cognitive Domain*. Addison-Wesley.

Coulom, R. 2002. Treemaps for Search-tree Visualization. In Uiterwijk, J. W. H. M., ed., *The Seventh Computer Olympiad Computer-Games Workshop Proceedings*.

Guzdial, M., and Soloway, E. 2002. Teaching the Nintendo Generation to Program. *Commun. ACM* 45(4):17–21.

Naps, T. L. 2005. JHAVÉ – Supporting Algorithm Visualization Engagement. *IEEE Computer Graphics and Applications* 25(5).

Plaisant, C. 2004. The Challenge of Information Visualization Evaluation. In *AVI '04: Proceedings of the working conference on Advanced visual interfaces*, 109–116. New York, NY, USA: ACM Press.

Shneiderman, B. 1992. Tree Visualization with Treemaps: A 2-D Space-filling Approach. *ACM Transactions on Graphics* 11(1):92–99.

Stasko, J. 2000. An Evaluation of Space-filling Information Visualizations for Depicting Hierarchical Structures. *Int. J. Hum.-Comput. Stud.* 53(5):663–694.