Visualization Tools for Real-time Search Algorithms

Yoshitaka Kuwata[1] *& Paul R. Cohen*

Computer Science Technical Report 93-57

Abstract

Search methods are common mechanisms in problem solving. In many AI applications, they are used with heuristic functions to prune the search space and improve performance. In last three decades, much effort has been directed toward research on such heuristic functions and search methods by AI community. As it is very hard to build theoretical models for heuristic functions to predict their behavior, we can often only check their performance experimentally.

In practical applications, it is important to understand the search space and behavior of heuristic functions, otherwise, we cannot figure out what's going on in actual applications and cannot control them. These issues are critical, especially in the field of real-time problem solving, in which applications have time constraints and are required to finish processing within the given time interval.

In this report, visualization methods are introduced as tools to understand the search spaces and behavior of heuristic functions. As examples of the usefulness of visualization methods, A* and IDA* algorithms are represented in various forms. They can be used to debug practical applications that use heuristic functions.

---

[1] NTT Data Communications Systems, Development Section

1. Why Visualizations are Important in AI

Looking back on the history of science, observation tools often played a very important role in scientific discovery. In the ages before the invention of the telescope, for example, we could not observe planets in detail. Telescopes made it possible to measure accurate positions of the planets, and that made it possible to find Kepler's law, Newton's laws and so forth. The same story is told in other areas of science. The microscope, X-ray, spectrum analyzer and other observation tools all contributed to the progress of science. In other words, the history of science is also the history of the engineering of observation tools.

The story is the same in computer science. Today, computers are used to simulate a variety of problems in computer science; for example, computer simulations of the execution queue of an operating system. Queuing theory provides some guidance but the optimal queue length is often determined experimentally with computer simulations. In this example, computers are used as tools to study theories in computer science and they are also the target of study in computer science. When we develop theories of computer science, we also need to develop observation tools for them. Again, we cannot develop theories without observing phenomena.

In computer science, especially in artificial intelligence, the development of observation tools seems not to have been very focused. This paper describes a collection of visualization tools for search algorithms, because search algorithms are very common. To apply search mechanisms for problem solving, the problem space must be defined as a set of states, including an initial state and goal states, and a set of operators to move from one state to the other states. A solution corresponds to a series of operations moving from the initial state to a goal state.

We can analyze search algorithms theoretically or empirically. Theoretical analyses are sometimes better than experimental ones, because we don't need to collect data, and because the results are often more general. However, we usually need to rely on heuristic functions to solve big and complex problems in realistic time. Heuristic functions make analysis harder as they often don't have theoretical underpinning. In the case of chess, for example, thousands of suggestions exist in the books, but they don't have theoretical explanations. Instead, they came from humans' experiences and inspirations. In such cases, we need to rely on experimental analysis of the heuristics.

By applying visualization tools to search algorithms, we can observe phenomena caused by heuristic functions. These tools are also useful for debugging, verification and validation.

Search spaces are described in section 2 of this paper. Conventional techniques such as search tree representations and depth-number of node representations, and their advantages and disadvantages as visualizations are discussed in section 3. In section 4, visualizations of heuristic function are described with examples from the

Eight Puzzle. Time series and frequency analysis of search algorithms are discussed with examples in section 5.

2. Objects to Visualize in Search

2.1 Search Space

The most important thing in problem solving by search is to understand search spaces themselves. They vary from problem to problem. For example, backgammon has 10^{20} states in its search space and chess has a much bigger state space. The structure of search spaces is also completely different in various problems. In chess, large numbers of actions are possible in the first stage of the game but actions are limited in the endgame. On the other hand, in shogi, which is a two-player game similar to chess that allows reuse of dead pieces, the movements in the last stage aren't limited as in chess.[2] Therefore, the search spaces for the two games would be different, even though these games look similar.

The simplest way to represent a search space is to count the number of states in it. For example, we can compare two problems A and B, which have ten million states and one million states, respectively. We expect to take 10 times longer to solve problem A than to solve problem B. This is because we assumed problem A and B are equally difficult; i.e., they have the same search space structure. But it is possible to define a problem which has a bigger search space but is simpler to solve. Thus simple search state count alone is not sufficient to characterize a search problem.

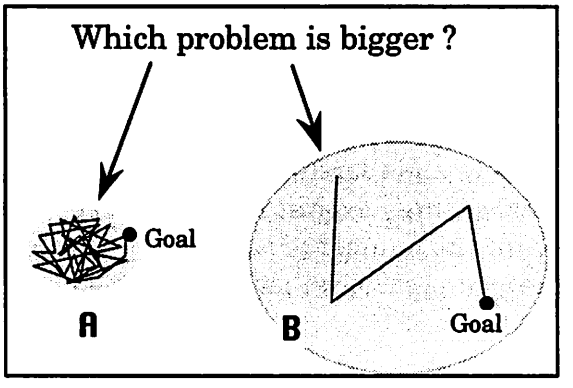


Figure 2.1.1 Two Problems The number of possible states in problem A is much smaller than in problem B. It takes more than 100 steps to reach the goal in problem A, but only 3 steps in problem B on average.

2.2 Shape of search space (Branching Factor and Depth)

Assuming we can represent problem spaces as search trees, the average branching factor and depth of search tree are useful characterizations. The average branching factor is the average number of successor states following each state. For

[2] Although possible actions in the endgame are fewer than in the first stage, shogi has more possible endgame actions than chess.

example, in the Eight Puzzle, the number of successor nodes[3] is either one, two, three or four. When we solve the puzzle allowing cyclic actions[4], the average branching factor is about 2.6. On the other hand, it is 2.3 without cyclic actions. These numbers were taken from actual execution traces of searches that have different successor functions. It is often very difficult to calculate the theoretical branching factor. Thus experimental values for branching factor[5] are often used. In general, the branching factor depends upon the successor functions[6] in the search method.

The depth of a search tree represents the length of a sequence of operations required to reach the terminal nodes. When we assume a search tree is completely balanced and has branching factor b and depth d, the number of nodes at level i is given by b^i, therefore the total number of nodes in the tree n is given by the following equation:

$$\begin{aligned} n &= 1 + b + b^2 + b^3 + \cdots + b^d \\ &= \sum_{i=0}^{d} b^i \end{aligned} \qquad (2.2.1)$$

If we assume a tree has different branching factor b_i at level i, equation (2.2.1)[7] would be the following:

$$\begin{aligned} n &= 1 + b_1 + b_1 \cdot b_2 + b_1 \cdot b_2 \cdot b_3 + \cdots + b_1 \cdot b_2 \cdot b_3 \cdots b_d \\ &= 1 + \sum_{i=1}^{d} \prod_{j=1}^{i} b_j \end{aligned} \qquad (2.2.2)$$

Equations (2.2.1) and (2.2.2) can be applied to a fully balanced tree only when theoretical branching factors and search depth are known. However, it is possible to determine *the effective branching factor* at level i (b_i) experimentally by actually searching. When we count the number of nodes at level i as n_i and at level $i+1$ as n_{i+1}, the average effective branching factor at level i is defined by the following equation:

$$b_i = \frac{n_{i+1}}{n_i} \qquad (2.2.3)$$

The change in effective branching factor shows the change of the *effective search space* which is defined as a result of the search space and of the heuristic function used to prune nodes. For example, we would observe large b at shallow levels but small b in

---

[3] Successor nodes in a search tree represent possible successive states.

[4] Cyclic actions are defined to generate the same states that appeared previously. For example, moving one piece back and forth many times corresponds to a cyclic action. We know such actions would not help in solving the puzzle.

[5] We refer to experimental branching factor as the "effective branching factor" in the rest of this paper.

[6] Successor functions are functions that return successor nodes.

[7] This equation still assumes the tree is balanced but has different branching factor at each level.

deep levels in the search of games like chess and shogi. In such games, there are a huge number of possible actions in the beginning stages but fewer at the end. Thus the expected branching factor in search at each level will vary.

Note also that equation (2.2.3) gives only the average branching factor at a certain level and does not tell us whether the search tree is balanced. The height of search trees is not necessarily the same at each branch in actual search, as one branch can reach a dead end before another. In such cases, the number of terminal nodes at level i will affect the calculation of branching factor at that level. Because terminal nodes have no successor nodes, their branching factor is 0, which decreases the average branching factor. When we are interested only in the nodes that have at least one successor node, we should exclude terminal nodes from our calculations. One example of the effect of terminal nodes on the average branching factor is shown as figure 2.1.2.

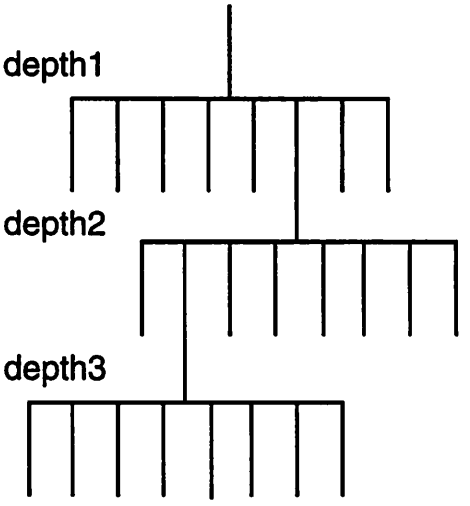


Figure 2.1.2 An example of an Unbalanced Tree In this tree, there is one node at each depth which has 8 successor nodes while all others are terminal nodes (branching factor = 0). The average branching factor at depth 2 and depth 3 is 1. Nodes which have 8 successor nodes contribute to increase the average branching factor.

This effect is also observed when we use a heuristic function to prune branches from a search tree. The pruned nodes can be regarded as a terminal node, therefore, the effective branching factors at each depth becomes smaller than the values given by the successor function. This effect is also observed when we are forced to stop search at a certain level. In this case, the observed branching factor at the lowest level would be zero.

2.3 Pruning and Heuristics

When it is known that some parts of the search space are not worth searching, the cost of search can be saved by excluding those areas from the solution space. For example, in the search algorithms used to solve the Eight Puzzle, we don't need to consider the previous state as a successor state, because it would cause meaningless

iterative actions. We know such actions would not help us solve the puzzle, therefore, we can prune them.

It is common also to use heuristics to choose the best successor state earlier than less worthy successors. If the problem is to find one acceptable solution, and not to search all of the state space, choosing the best action first will help us find the solution faster. Heuristics are used for two purposes:

1. to choose the best successor state
2. to omit subtrees where there is no need to search (pruning)

Visualizations must be capable of representing both of these purposes. For example, we can use branching factor analysis directly to know the effect of pruning. As the result of pruning, the average branching factor becomes smaller than the expected value. On the other hand, we can indirectly observe the quality of the path the heuristic function chooses from the sequence of heuristic values the function returns. In the case of $\hat{h}$ functions in A* search, for example, if the function is chooses a good path to the goal, the $\hat{h}$ value is expected to decrease.

A good visualization would represent these two purposes directly and separately.

2.4 Characteristics of search algorithms

As the goal of visualizations for search algorithms is to understand the algorithms and to predict their behaviors, visualization tools must have the ability to represent how search algorithms are working.

For example, **depth-first search**[8] and **breadth-first search**[9] behave differently. The former searches down to the deepest place first while the latter fully explores the shallowest level of the search tree first. We can decompose the characteristics of other search algorithms into depth-first components (deeper first) and breadth-first components (broader first). **Best-first search**[10] is a general strategy to search the best possible successor node first. If the evaluation function of a best-first search relies heavily on the cost of the path to reach the node, the search will be similar to breadth-first search. By contrast, if the evaluation function emphasizes the reachability of goals, the search would be similar to depth-first search. The characteristics of best-first search depend on the evaluation function. In general, in best-first search, we need to balance various cost factors in the evaluation function in a task-specific manner.

Evaluation functions typically must be carefully tuned by experiment. Decomposing search behavior into depth-first and breadth-first components is one way

---

[8] Pseudo-code of depth-first search is shown in Appendix A.1.

[9] Pseudo-code of breadth-first search is shown in Appendix A.1.

[10] Pseudo-code of best-first search is shown in Appendix A.1.

to characterize and understand search algorithms. In this paper, various visualization techniques are shown to accomplish this decomposition.

There are interactions between search algorithms and search spaces. Evaluation functions that are carefully tuned to one particular problem will not always work well for slightly different problems. For example, the right-hand method[11], or left hand method, are easy ways to solve simple maze problems. They are basically a depth-first search. Figure 2.1.3 is a simple example of two maze problems, which illustrates an interaction effect between search algorithms and search spaces. In case (A), the left-hand method works better than the right-hand method; in case (B) the right-hand method works better.

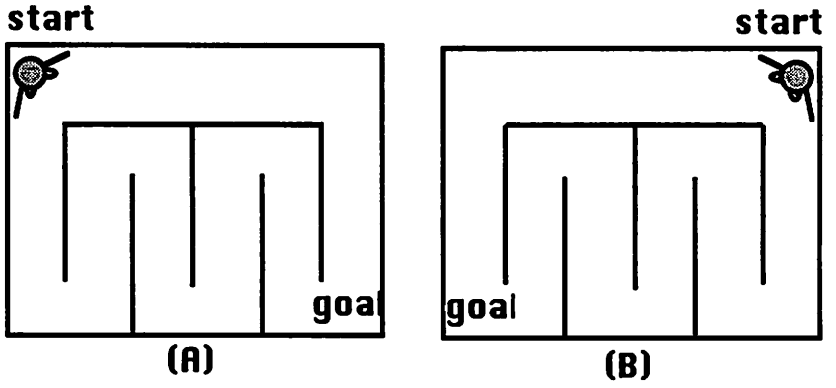


Figure 2.1.3 Two Maze Problems One simple example of the interaction of algorithm and search space. (A) is easily solved by left-hand method. On the other hand, in maze (B), which is the mirror image of (A), left-hand method is not a good solution.

Actual observations depend both on the characteristics of search and the search spaces. A good visualization would represent them independently and help determine the interaction effect between them.

[11] One can solve simple mazes, which have no islands, by keeping one hand to the wall and walking around the maze. "Right-hand" method refers to solving the problem by using the right-hand and turning around the maze clockwise, and vice versa for "left-hand" method.

3. Conventional Visualization tools

3.1 Search Trees, Graphs

It is very common to represent search spaces as trees[12]. In this representation, nodes in the tree correspond to states in the search. Search goes from the root node of the tree down to the leaf nodes – nodes which have no further successor nodes or which are desired states. The path from the root to the leaf nodes traces the steps from the start state to a goal state in the solution space. The following figure shows a tree representation of a search space.

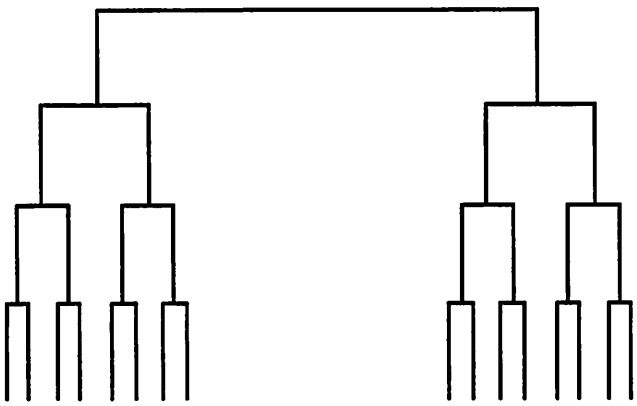


Figure 3.1.1 Example of Graph Representation of Search Tree Fully balanced tree with branching factor b=2 and height of tree = 4. There are 31 total nodes in this tree, including root node and leaf nodes.

In this representation, when we draw arcs as the same length and descendant nodes as the same distance, we can easily see the search depth and search width as the height and the width of the search tree, respectively.

It is also easy to show characteristics of search algorithms such as differences in search order. For example, depth-first search will pick up one path and continue searching down to the leaf node, which is a state the search algorithm was looking for, or which has no successors. In the former case, the search algorithm has the option to keep searching until it finds better solutions. To search further in the tree after reaching the leaf node, the search algorithm needs to look up other branches in the path. The simplest method is to search lowest alternative branches. This operation is known as *backtracking*. Depth-first search would continue backtracking until it finds a solution or searches all of the tree. The following figure shows the characteristics of depth-first search using tree representations.

[12] In general, a state space would be represented by general directed graphs. However, we can expand these graph representations by duplicate subtrees in a tree representation.

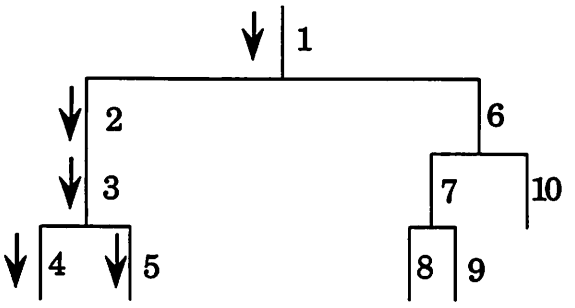


Figure 3.1.2 Depth-first Search in a Search Tree The order of search is labeled on each of the branches. In this example, the search algorithm checks the leftmost branches first, then goes to the right branches.

On the other hand, a breadth-first search algorithm searches the shallow level of the tree first, then increases the level of search one by one. Figure 3.1.3 shows the behavior of breadth-first search using the same representation as depth-first search.

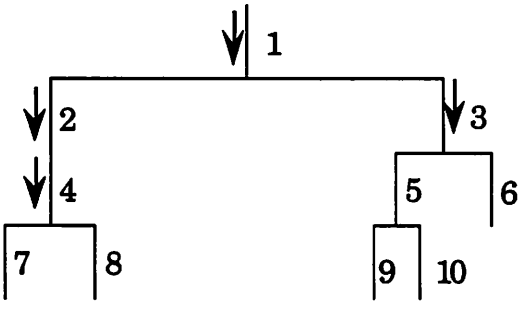


Figure 3.1.3 Breadth-first Search in a Search Tree The order of search is labeled on each of the branches. Breadth-first search checks nodes in the shallowest level first, then searches progressively deeper levels.

These representations are very intuitive and easy to understand, but they are useful mainly for small search spaces. When the tree becomes large, we cannot represent all search states as nodes in the search tree. Part of this difficulty depends on human perceptual capabilities. It is said to be hard for humans to handle more than 10 to 20 objects at once. The tree representation of the search space for a practical problem could be more than ten million states. Even if we could recognize patterns in such a large tree representation, we could not understand them in detail. To determine the details of these patterns, we need to look closely at individual nodes and paths inside the tree.

In the large tree in figure 3.1.4 (showing about two hundred nodes) we can recognize that the top leftmost branch has many more nodes than other two branches. However, we cannot determine the average branching factor of each branch from this representation.

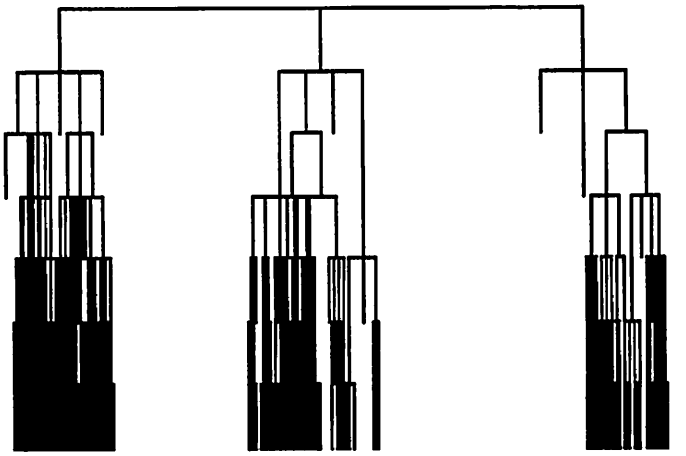


Figure 3.1.4 Large Tree This figure is made by randomly pruning subtrees from a fully balanced tree with b=6 and h=7. The resulting tree includes about 200 nodes.

This discussion is also valid for variations of tree representations such as 'AND-OR trees' and 'labeled trees.' They are very intuitive and easy to understand for small search spaces. Thus they help us understand the basis of search spaces and algorithms, but are less useful for analyzing large scale search spaces in detail.

3.2 Depth-number of node counts

The size of a search space is commonly represented as the number of nodes at each level.

Theoretical Depth-Node Counts

Figure 3.2.1 plots node counts for a search tree. The number of nodes at each level is given by b^i, where i is the depth of the search tree and b is the average branching factor. Depth-node counts for three values of b are shown in this figure.

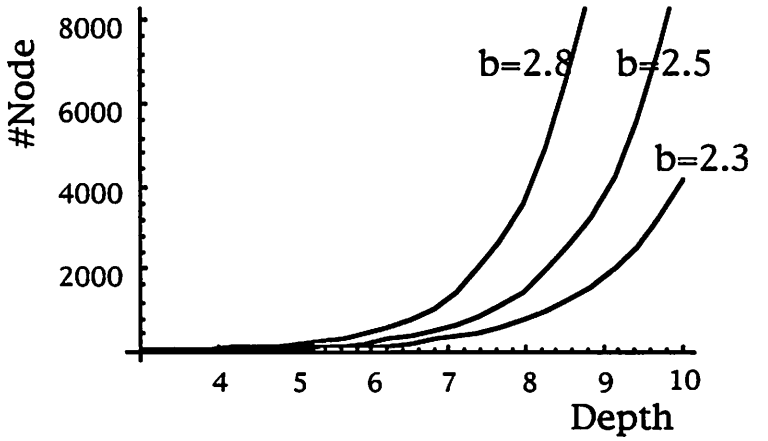


Figure 3.2.1 Node Count Representation of Search Trees This figure is based on equation (2.2.1). The number of nodes at successive depths increases exponentially.

Empirical Depth-node counts

It is also useful to apply the node count visualization to empirical data. The following figure shows actual data from Eight Puzzle search using three different search methods. One is pure depth-first search with a depth-bound, which corresponds to one iteration of IDA* search with a constant $\hat{h}$ function. The other two are IDA* search[13]: one uses a Manhattan distance function for $\hat{h}$ and the other uses a final position count function for $\hat{h}$.[14]

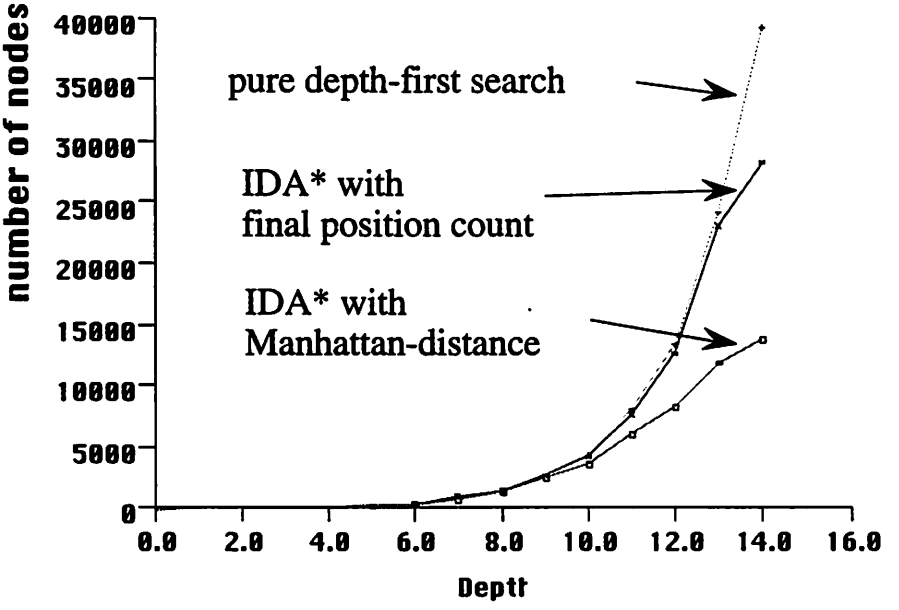


Figure 3.2.2 Empirical node counts from Eight Puzzle This figure shows the number of nodes expanded at each level of the search tree for three algorithms. As pure depth-first search doesn't prune nodes at all, the curve from pure depth-first search shows the total number of nodes at each level. By comparing with that curve, we can see how many nodes are pruned by the other two algorithms. If the number of nodes at one level is the same as pure depth-first search, then there is no pruning at that level.

The curve from pure depth-first search uses no pruning, therefore it represents all of the nodes at each depth. This curve grows exponentially as we have shown in the theoretical depth-node counts graph (figure 3.2.1).

Now we can check the effect of pruning by comparing pruned data with the pure depth-first search curve. The other two curves are almost identical to the no-pruning curve until depth = 9. This means the two algorithms expanded almost the same number of nodes until depth = 9. The curve from IDA* with the final position count function is different from no-pruning curve only from depth = 14, but IDA* with Manhattan distance has a lower growth ratio than the no-pruning curve. This shows that the former can prune nodes only at the deepest levels, but the latter can prune

[13] Pseudo-code of IDA* search is shown in Appendix A.3.

[14] These heuristic functions are described in Appendix A4 in detail.

nodes at shallow levels also. *We cannot know why these functions are different, given this representation, but we can know how they are different.* We need to introduce visualization of heuristics to know why these heuristic functions behave differently.

From equation (2.2.3) the average branching factor at each level for this data can be calculated. The result is shown in figure 3.2.3.

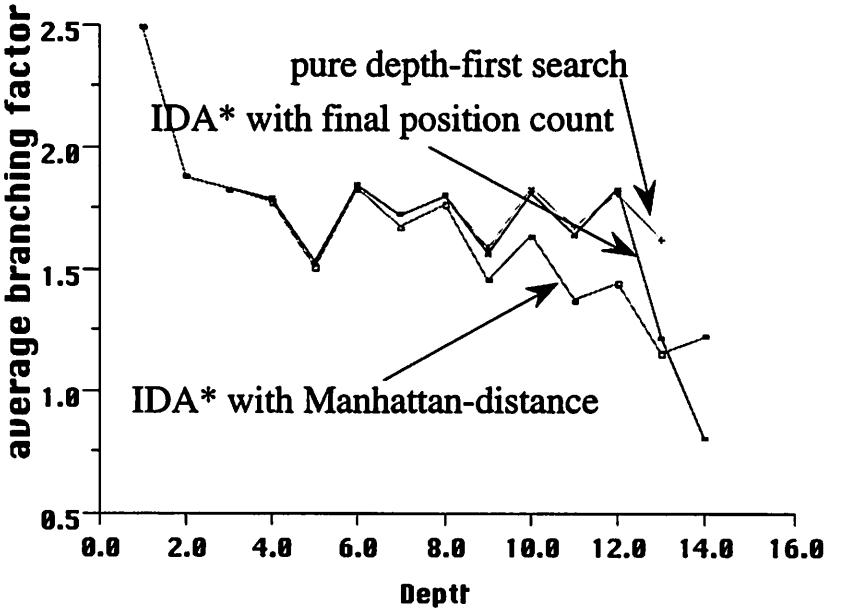


Figure 3.2.3 Empirical branching factor at each level of the Eight Puzzle This figure shows the empirically derived average branching factor for each algorithm.

If the tree is fully balanced and there is no pruning during search, the average branching factor is expected to be constant at every level and the plot is expected to be a flat line. In search without pruning, which is shown as pure depth-first search in the figure, the average branching factor has some structure: small branching factors are followed by large branching factors and vice versa. In the Eight Puzzle, the full search tree is likely to have a small number of moves followed by a large number of moves, and vice versa. For example, the center position has three possible moves that lead to states which have only two legal moves.

Time Trace of Node Count

It is also useful to observe the behavior of various search methods in a node count graph by plotting the history of searched nodes. Figure 3.2.4 represents the expected history of node counts in depth-first search. In this graph, the number of nodes at each level increases as time passes. Thus later curves are above earlier curves, as more nodes are searched at each depth. The curve grows **vertically** from bottom to top.

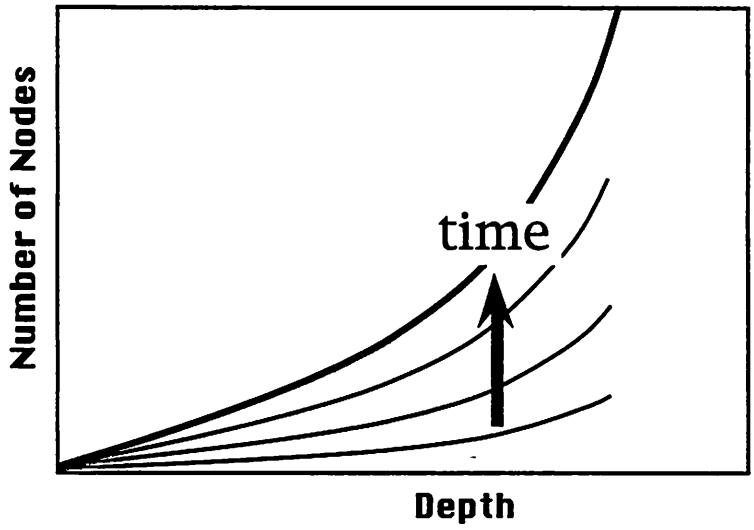


Figure 3.2.4 Time Trace Representation in Node Count (general depth-first search) This is the expected graph on time trace of node count in depth-first search. As time passes, the number of nodes visited by depth-first search increases at each level.

An actual depth-first search for the Eight Puzzle, shown in figure 3.2.5, is pretty much as expected. The number of nodes at each depth increases as time passes. If the search tree is not balanced at all, the number of nodes shown in this figure wouldn't increase proportional to time spent for search. We can conclude, then, that the search tree in Eight Puzzle is balanced.

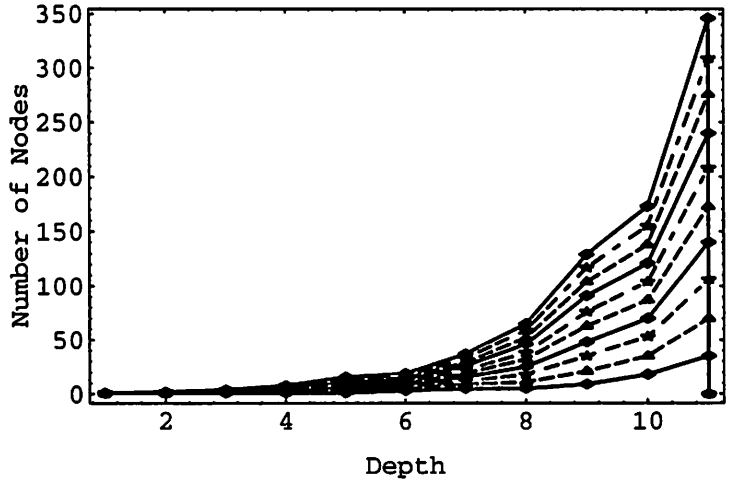


Figure 3.2.5 Time Trace Representation in Node Count (Depth-first Search on the Eight Puzzle) This graph was made from one instance of a time trace of depth-first search on the Eight Puzzle. Each curve represents the number of nodes at each depth in every 80 nodes expanded; i.e., the lowest curve is the plot of the first 80 nodes searched, the second lowest curve is from the first 160 nodes, and so on. The envelopes of these curves grow exponentially as the depth increases.

On the other hand, the graph of breadth-first search will be completely different from depth-first search in a node count representation with history. The following figure is an expected execution history of depth-first search. In this representation, the breadth-first search progresses **horizontally** from left to right. This is because the search moves from shallow levels to deeper levels progressively. If the search space is the same as for depth-first search, the envelope of the curve is also the same exponential curve as that of depth-first search.

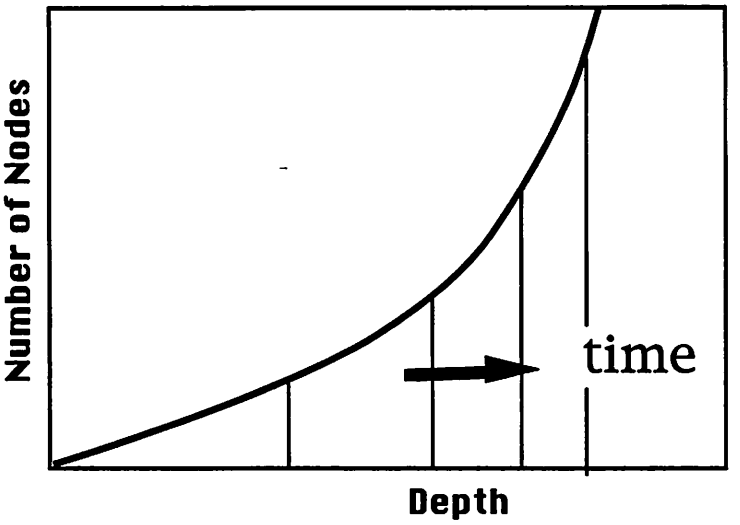


Figure 3.2.6 Time Trace Representation in Node Count (general breadth-first search) As time passes, the frontier of the search progresses from left to right horizontally. The envelope of the curve reflects the search space and is the same with depth-first curve.

The result of an actual breadth-first search on the Eight Puzzle is shown in figure 3.2.7.

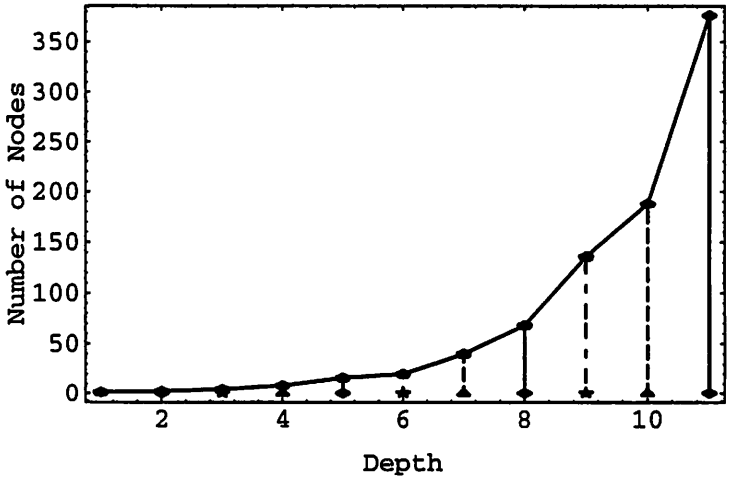


Figure 3.2.7 Time Trace Representation in Node Count (Breadth-first Search on Eight Puzzle) This graph was made from a time trace of breadth-first search on the Eight Puzzle. Each curve represents the number of nodes at each depth.

Other search algorithms mix breadth-first search and depth-first search. The following figure shows best-first search applied to the same search tree as figures 3.2.4 and 3.2.5. If the evaluation function used in best-first search relies on depth, the search is like depth-first search and the envelope grows vertically. If the function relies on breadth, the search becomes like breadth-first search and curve grows horizontally. In other words, *the vertical growth of the curve represents depth-first components in the search and the horizontal growth of the curve represents breadth-first components.*

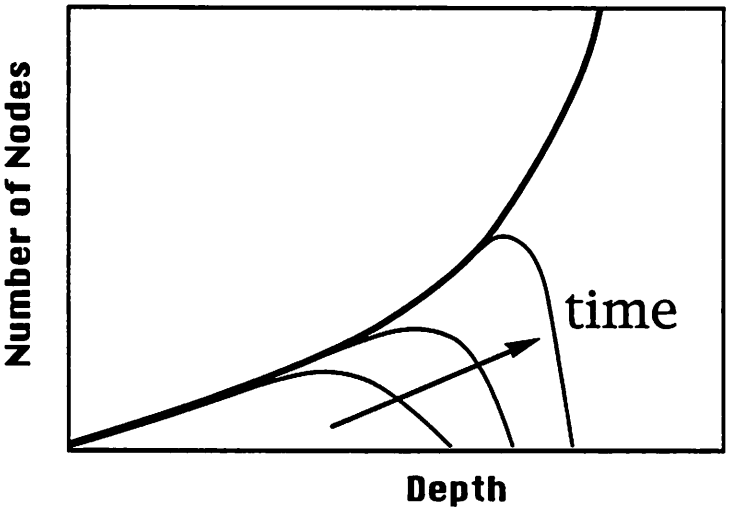


Figure 3.2.8 Time Trace Representation in Node Count (general best-first search) General best-first search has a depth-first component (vertical growth) and a breadth-first component (horizontal growth).

The result from IDA* search with Manhattan distance is shown in figure 3.2.9. Each curve corresponds to an iteration of IDA* search; i.e., the leftmost curve, which starts at depth 0 and reaches depth 3, is from the first iteration. The second leftmost curve is from the second iteration, which expanded nodes to level 4, and so forth. The eighth (last) iteration is shown as the top curve. In this case, a solution is found at the eighth iteration at level 21. The search horizon progresses horizontally in the later iteration of IDA* search. This fact shows that the heuristic function prunes many branches in deeper parts of the search tree.

For comparison, a non-pruning algorithm is also shown in this figure as an exponential curve. The search is very much like depth-first search (i.e., it grows rapidly in the horizontal direction) but expands fewer nodes than pure depth-first search. Especially in the last two iterations, the search progresses at more than ten levels at once. This means that the heuristic function can help choose the right branch and avoid searching bad paths.

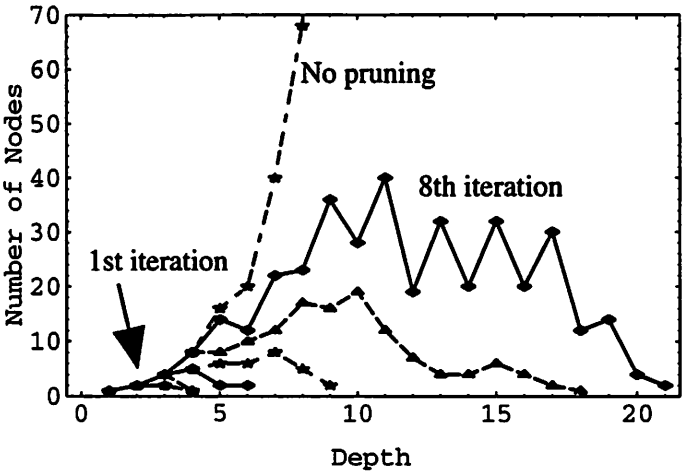


Figure 3.2.9 Time Trace Representation in Node Count (IDA* Search with Manhattan distance) One instance of a time trace of node count using IDA* search on the Eight Puzzle.

Figure 3.2.10 is another example from the puzzle with a different $\hat{h}$ function. In this case, the left-hand side of the envelope of the time trace follows exactly the exponential no-pruning curve, which means the algorithm expanded all nodes at each level without pruning. From comparison with the previous figures, this graph also shows this search includes more breadth-first components than the previous search algorithm, as the envelope of the curve grows faster vertically than horizontally (compare with figure 3.2.9). This means that the $\hat{h}$ function is not helping very much by pruning nodes, and as a result, search cannot go deeper into the search tree until it expands many more nodes at shallower depths.

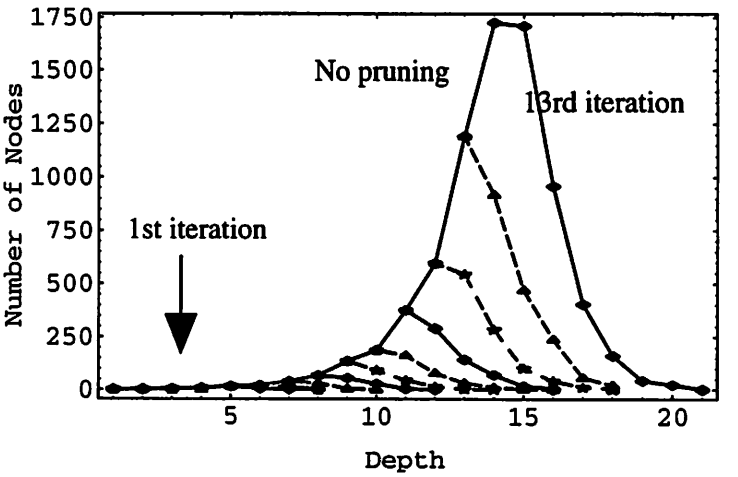


Figure 3.2.10 Time Trace Representation in Node Count (IDA* Search with final position count) Another instance of a time trace in node count using IDA* search on the Eight Puzzle. Note that the scale of the Y-axis is much bigger than in the previous two figures.

4. Visualization of Heuristics

4.1 Depth-Heuristic Function Representations

When heuristic functions are used for search, visualizations of the search give us many useful ideas about how to evaluate them. In many cases, we don't have theoretical expectations for heuristic functions. We need to determine how well they perform empirically, for which we can rely on visualizations of them.

The *depth-heuristic function representation* is shown in the rest of this subsection. Figure 4.1.1 was made from a history of search in solving the Eight Puzzle problem. This figure is from one particular search. The X-axis represents $\hat{h}$-value (estimated h-value) given the current state, the Y-axis shows the depth of the current state from the initial state, and the gray level at each point represents the number of states at that level. The positions colored in black signify that no such state exists in the search space. White stands for many nodes[15]. In this figure, the $\hat{h}$ function was used only to evaluate each state; it wasn't used during search to prune successor nodes. Thus this figure represents the structure of the entire search space from the view of the $\hat{h}$ function. For example, the $\hat{h}$-value of the initial condition was 16. Then two nodes at $\hat{h}$-values 15 and 17 and level 2 are expanded, next $\hat{h}$-values 14, 16, and 18 at level 3, and so forth[16]. From this figure, we can observe that the number of successor nodes increases as the search progresses. In particular, a huge number of nodes have $\hat{h}$-values around 17 (white place at $\hat{h}$=17, level=12). The nodes distribute almost symmetrically with the center at $\hat{h}$=16.

In the actual IDA* search that uses this $\hat{h}$ function, nodes are searched in order of smaller $(\hat{h}+g)$-value[17]. Figure 4.1.2 is a visualization of search order using the same $\hat{h}$ function as in 4.1.1. White diagonal arrows from the bottom right to top left show groups of nodes which have the same $(\hat{h}+g)$-value. In the first iteration of IDA* search, the nodes on the leftmost arrow are searched, because they have the same $(\hat{h}+g)$-value as the initial threshold (16). In the second iteration, the threshold value is set to 18 and nodes on the second leftmost arrow are also searched, along with the nodes on the leftmost arrow. In each iteration of IDA* search, the search horizon progresses one arrow toward the left, as more nodes are searched at deeper levels.

---

[15] It is possible to use a 3D graph that shows the number of nodes on the Z-axis (instead of a 2D gray scale graph). The gray scale representation is chosen for its intuitiveness of representation. In this figure, gray level is determined by the number of nodes at each state and plotted on a logarithmic scale. (While the most frequent state has more than 20,000 nodes, it is practically impossible for humans to distinguish more than 10 gray levels simultaneously.)

[16] In this h-hat-function, the h-hat-value of the next state is one plus or one minus the current h-hat-value. As the result of this property, the depth h-hat-function representation has a checkerboard appearance.

[17] In this example, the depth from the root node was used as the g-value for IDA* search.

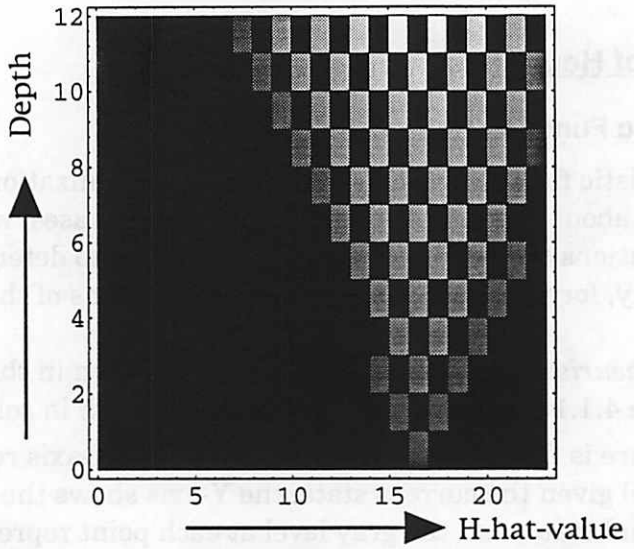


Figure 4.1.1 Example of Depth-Heuristic Function Visualization This figure is made from one instance of search on the Eight Puzzle by using the Manhattan distance with the blank space as $\hat{h}$ function. The X- and Y-axes represent $\hat{h}$-value and depth respectively. Each square in this plane represents the state corresponding to the values on the X-axis and the Y-axis. For example, the square next to the top right corner shows the state that has $\hat{h}$=22 at depth=12. The gray tone in each area represents the number of nodes at that state (black=none, white=many).

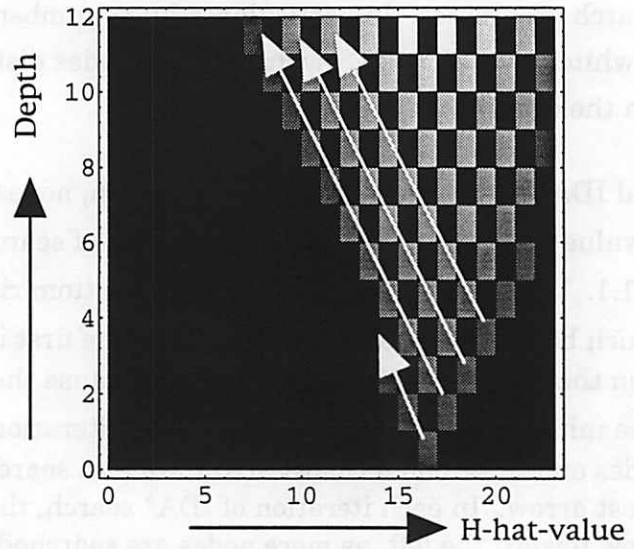


Figure 4.1.2 Search order of Figure 4.1.1 White arrows in this figure show states which have same $(\hat{h}+g)$-value. In the IDA* search algorithm, thresholds are set in each iteration, then search is done up to the threshold value. The search progresses from the leftmost arrow to right arrows in successive iterations.

Figure 4.1.3 is the same search as in figures 4.1.1 and 4.1.2 but the plot shows only actually searched nodes by using the $\hat{h}$ function for pruning[18]. The path that reached the first solution is shown by a jagged white arrow running from bottom right to top left. In this example, two actions that increased the $\hat{h}$-value were needed to reach the solution. These two actions are represented by the jagged parts of the arrow, and correspond to three iterations in IDA* search with threshold values 16, 18, and 20. In IDA* search, the threshold value was set to 16 first because the $\hat{h}$-value of the initial condition is 16. The successors that had bigger $(\hat{h}+g)$-values were not expanded in the first search iteration. Only nodes on the diagonal line from $\hat{h}$ =16 (nodes $\hat{h}$=15/d=1 and $\hat{h}$=14/d=2) were searched in the first iteration. In the second iteration, the threshold was set to 18 and IDA* searched to the nodes on the next diagonal line. In the third iteration, the goal node was found at d=20 and $\hat{h}$=0.

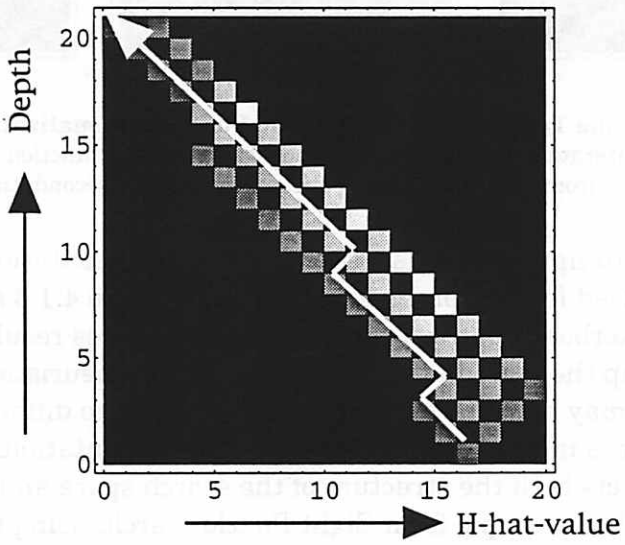


Figure 4.1.3 Example of Depth-Heuristic Function Visualization (from the Eight Puzzle, Manhattan distance with blank space) The same instance of depth-heuristic function shown in figures 4.1.1 and 4.1.2, but nodes actually searched are shown in this figure. Note that the nodes in the upper right of figure 4.1.1 are pruned and not searched. The white arrow shows a path to one solution found in this search.

Note that the nodes above the diagonal line have higher $(\hat{h}+g)$-values than the threshold value of the last iteration. The nodes shown above the line were expanded in the last iteration of IDA* search, which recognized that these successors need not to be searched. In this representation, we regard these nodes as searched, and they are plotted in the same figure because they were actually expanded.

It is intuitive to observe the progress of the search as a sequence of depth-heuristic function representations. Figure 4.1.4 is generated from the same instance of

[18] Note that the scale of the Y-axis is different from figure 4.1.1 and 4.1.2, which makes the slope of the search horizon line different. The slopes would be the same if we plotted them on the same scale.

the IDA* search shown in the previous figures. Each picture corresponds to one iteration of IDA* search. In the first iteration, in the leftmost picture, only a small portion of the space is searched. In the second iteration, the algorithm is allowed to search the next diagonal line shown in figure 4.1.2 but does not reach the goal. The goal is reached in the third iteration, shown in the rightmost picture.

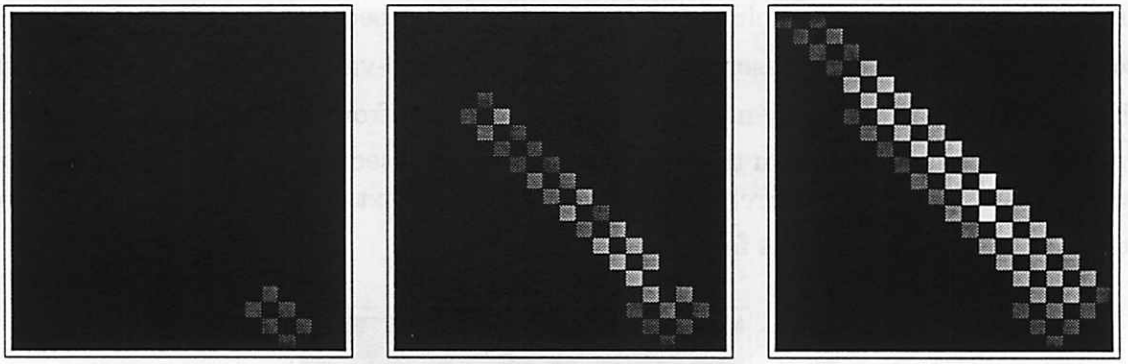


Figure 4.1.4 Time Trace of Depth-Heuristic Function Visualization Each figure is a snapshot of an iteration of IDA* search in the depth-heuristic function representation. The leftmost figure is from the first iteration, the center from the second, the right from the third.

We can regard figure 4.1.1 as a search space representation from the view of the heuristic function used for this problem solving, and figure 4.1.3 as the result of pruning. The checkerboard pattern shown in these pictures results from the heuristic function used to map the state in the problem space into heuristic values. Other heuristic functions may map states in the search space into different values, and thus into different patterns in depth-heuristic function representation. In other words, this representation reflects both the structure of the search space and the heuristic function. Figure 4.1.5 is another example from Eight Puzzle search, using the same initial condition but a slightly different heuristic function[19] to estimate the distance to the goal state. The search space has basically the same shape as with the previous $\hat{h}$ function, but because the values of successor states can be minus two, zero or plus two from the values of the current state, $\hat{h}$ can have only even values and the state space becomes evenly spaced vertical lines.

[19] In this example, the h-hat function used manhattan distance but didn't include the blank space in the count.

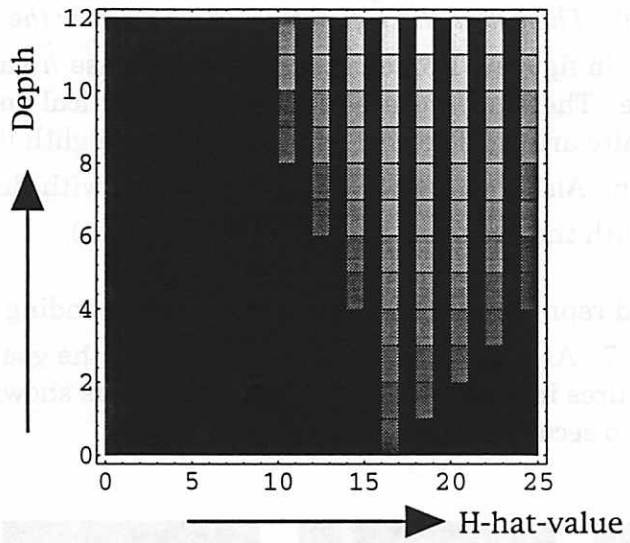


Figure 4.1.5 Example of Depth-Heuristic Function Visualization (from the Eight Puzzle, Manhattan distance without the blank space) All of the nodes up to depth=12 in the search tree are shown. The shape of the distribution of nodes is an inverted triangle similar the previous search. The striping is a product of the $\hat{h}$ function used.

Figure 4.1.6 shows nodes actually searched with the heuristic function shown in figure 4.1.5. This figure corresponds to figure 4.1.3. The path to reach the first solution is shown by the white diagonal arrow.

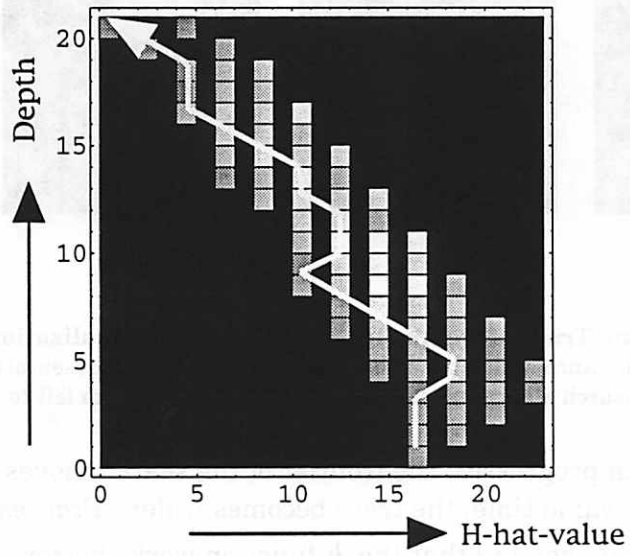


Figure 4.1.6 Example of Depth-Heuristic Function Visualization (from the Eight Puzzle, Manhattan distance without the blank space) This figure shows the nodes actually expanded in the search, which uses the $\hat{h}$ function shown in figure 4.1.5 for pruning.

By comparing figure 4.1.3 and 4.1.6, we can observe that the searched path in figure 4.1.6 is much wider than in figure 4.1.3. In the ideal case, an $\hat{h}$ function should return the exact estimation of the distance to the goal. Thus the path to the goal should be a straight line from the bottom right to the top left corner, meaning the area

searched is minimal. *The better the $\hat{h}$ function, the narrower the area shown in these figures, in general.* In figure 4.1.6, ten actions that increase $\hat{h}$ values are needed to reach the goal state. These actions are represented byvertical and right-leaning line segments in the white arrow. The goal was found in the eighth iteration of IDA* search with this $\hat{h}$ function. Also the number of nodes searched with this function is about twice as many as with the previous $\hat{h}$ function (figure 4.1.3).

An animated representation of this search, corresponding to figure 4.1.4, is shown in figure 4.1.7. As it took eight iterations to reach the goal with this $\hat{h}$ function, there are eight pictures in this figure. The first iteration is shown in the top left corner, the second in the top second left, and so on.

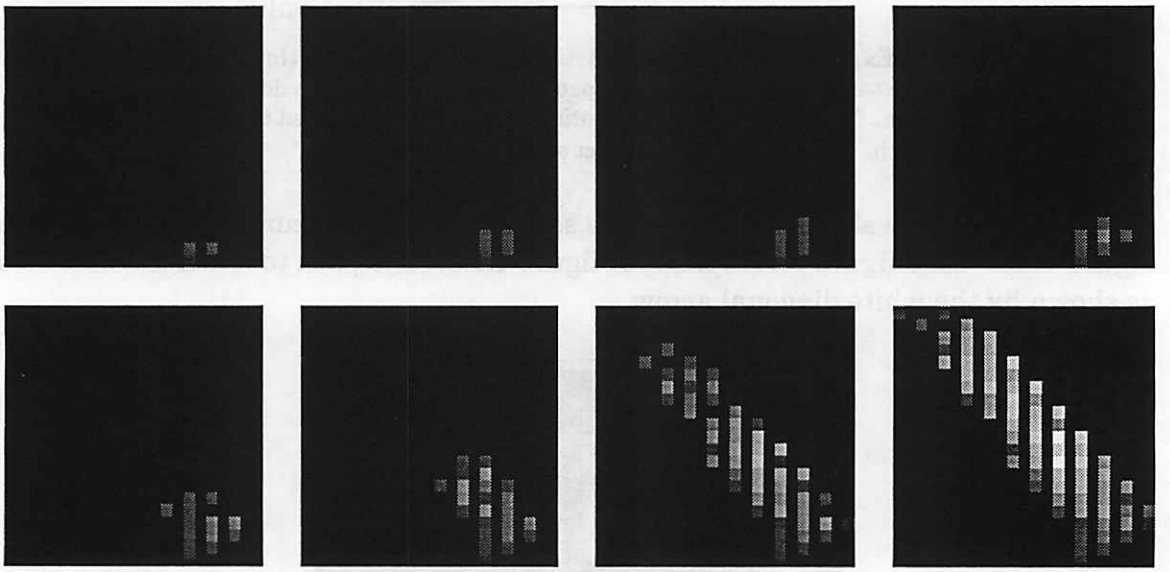


Figure 4.1.7 Time Trace of Depth-Heuristic Function Visualization (from the Eight Puzzle, Manhattan distance without the blank space) The same representation as figure 4.1.4 but with a different search algorithm. These figures are ordered from left to right and top to bottom.

As the search progresses, the frontier of the search moves toward the top left corner, while at the same time, the trace becomes wider. Progress is significant in the last two iterations, suggesting that the $\hat{h}$ function works better at deeper levels than in shallow levels.

5. Time Series Representations

5.1 Time Domain Representations

When we record search depth as the history of search, we can observe the behavior of search in the time domain, and we can apply time domain analysis techniques.

Figure 5.1.1 is an example from a time trace of general depth-first search. The X-axis is the order of search, which corresponds to time if we assume constant time to expand each node. The Y-axis is the depth of the searched node. In this example, the algorithm searched down to depth = 7 and continued visiting nodes at depth 6 and 7 several hundred times. Big spikes in this representation mean big backtracks. For example, a backtrack from depth 7 to depth 3 happened at time 130. Time 0 through time 130 corresponds to a search of one branch from level 3 if the search is done in depth-first order.

In the time trace of pure depth-first search, the distance between points in the series at one level represents time to determine one subtree below that level.

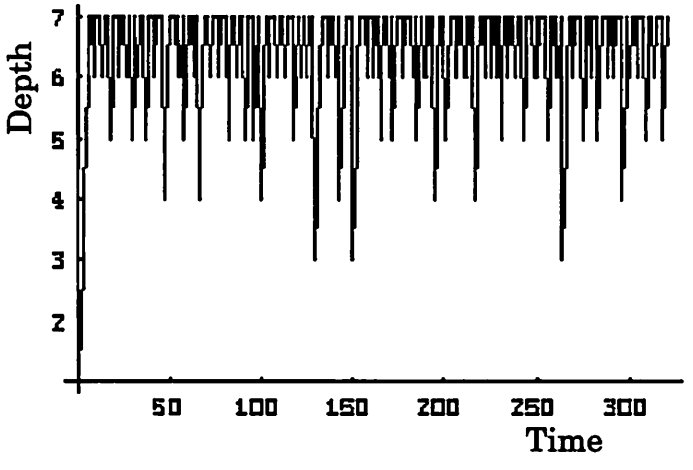


Figure 5.1.1 Time Trace of Depth-first Search The X- and Y- axes represent time and the depth visited at that time. In depth-first search, algorithm picks up one branch and tries to visit the deepest level of the branch first, then goes back and forth at the deeper levels. The big spikes in this representation signifies the termination of searching a subtree, with a large backtrack to the next subtree.

It is easy to see at which levels the algorithms spent time. For example, figure 5.1.2 represents the time trace of breadth-first search on the same search space as in figure 5.1.1. Breadth-first search visits all nodes at shallower levels first. The time spent at a level is proportional to the number of nodes at that level, so the shape of the time trace is completely different between depth-first search and breadth-first search. Depth-first algorithms have spiky time traces and breadth-first algorithms make stairs.

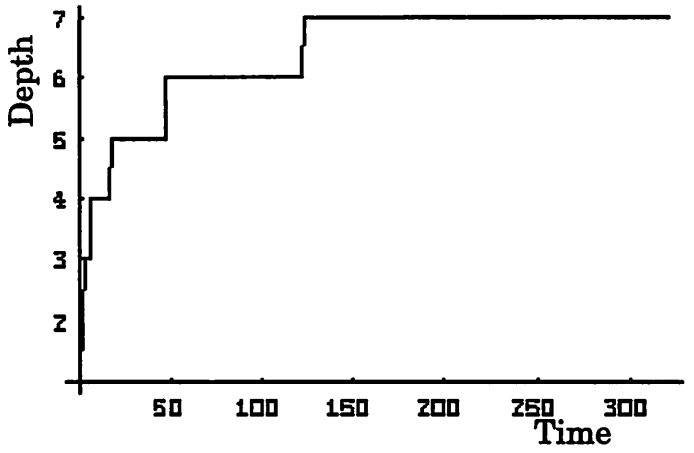


Figure 5.1.2 Time Trace of Breadth-first Search

However, it is not so easy to find patterns in time traces for algorithms other than pure breadth-first search and depth-first search. For example, figure 5.1.3 is the time trace of IDA* search for the problem discussed in the previous sections. It is very hard to tell much from this figure.

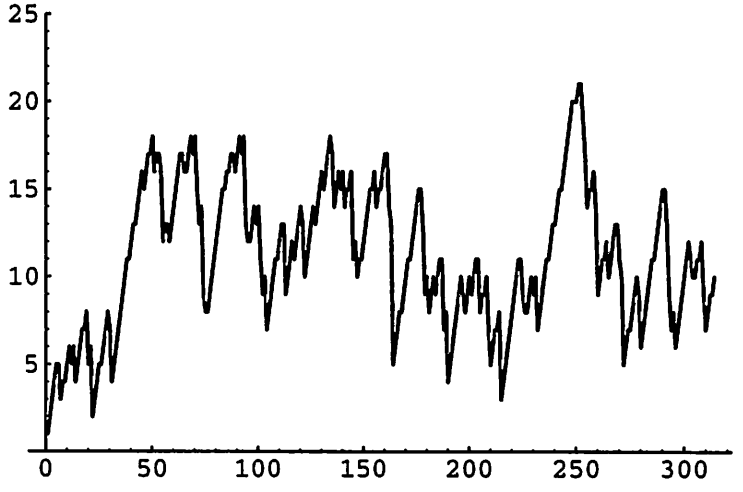


Figure 5.1.3 Time Trace of IDA* Search on Eight Puzzle

From the time trace of search algorithms we can get: 1) a rough idea of the behavior of the search algorithms; it is especially useful for pure depth-first search or breadth-first search, and 2) some idea of the frequency components of time trace; this is needed for the design of a smoothing filter (described the following section).

5.2 Frequency Domain

Time domain analysis is straightforward and very intuitive. Although it cannot distinguish two similar algorithms in detail, it is useful to see the search depth history. As the next step, we can apply fourier transforms to view the time trace of search algorithms in the frequency domain. By converting a sequence from the time domain into the frequency domain, we can observe iterative actions in the search algorithms more clearly. For example, visiting a node at depth 5, expanding three child nodes at depth 6 and then returning to a node at depth 5 corresponds an iteration of interval 4. If this happens often, it can produce a strong frequency component at frequency 1/4 Hz. Frequency components represent backtracking in search. In general, low frequency components represent large backtracks, which are likely to occur when returning to a shallow level of the search tree, and high frequency components come from minor backtracking at deeper levels.

Iterative components observed in the frequency domain can be caused by interactions between the structure of the search space and search algorithms. For example, the time trace of depth-first search and breadth-first search can include completely different frequency components even when applied to exactly the same search space. Thus, frequency analysis cannot separate the structure of the search space from characteristics of search algorithms.

In this subsection, the basis of the fourier transform is explained first. Then a smoothing technique for practical analysis is reviewed. To illustrate frequency analysis of search algorithms, we show frequency graphs for the Eight Puzzle and the traveling salesperson problem, with their interpretations.

5.2.1 Fourier Transform

The fourier transform is a common technique to transform a function in the time domain into the frequency domain. The basic idea is to regard an arbitrary wave form as the addition of simple sine waves. The transform decomposes the wave into sine waves, yielding the strength and phase of each decomposed sine wave. By adding all of the decomposed sine waves, we can reconstruct the original wave form. This is called the *inverse fourier transform.*

In the case where the original function is continuous, the following formulas are applied to calculate the fourier transform and inverse fourier transform.

$$\begin{cases} F(\omega) = \int_{-\infty}^{\infty} f(t) \cdot e^{-2\pi i \omega t} dt & \text{FourierTransform} \\ f(t) = \int_{-\infty}^{\infty} F(\omega) \cdot e^{2\pi i \omega t} d\omega & \text{InverseFourierTransform} \end{cases} \tag{5.2.1}$$

If the original function is in discrete form, equations (5.2.2) are used to calculate the fourier transform and inverse fourier transform. They are basically equivalent to the equations in (5.2.1) but assume a discrete number of wave forms.

$$\begin{cases} F(j) = \frac{1}{N}\sum_{k=0}^{N-1} f(t)\cdot e^{-2\pi ikj/T} & \text{FourierTransform} \\ f(k) = \sum_{j=0}^{N-1} F(j)\cdot e^{2\pi ikj/T} & \text{InverseFourierTransform} \end{cases} \qquad (5.2.2)$$

Note that in the case of the discrete fourier transform, the highest frequency depends upon the sample rate of the original function. If the sampling rate is 1 point per second, the highest frequency is 1/2Hz. For a sampling rate of 10 points per second, we can get a highest frequency of 5Hz[20].

As we are going to apply the fourier transform to time traces of search algorithms, we need to use the discrete fourier transform. Figure 5.2.1 shows a frequency domain spectrum from the node-depth history of a depth-first search.

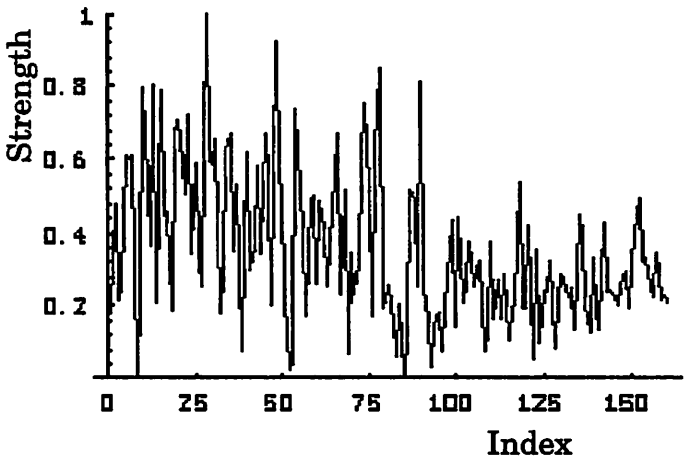


Figure 5.2.1 Frequency Distribution of depth-first search in figure 5.1.1

In this frequency distribution graph, the X-axis represents frequency (or interval) and the Y-axis represents relative strength. By simply applying equation (5.2.2) to the discrete time series of search, the frequency components are calculated as complex numbers. In this graph, the absolute values of each frequency component are calculated first, then they are standardized by dividing by the maximum component value. Note also that the zero frequency, which is referred to as D.C. level in electrical engineering, is omitted because it represents simply the average node depth.

[20] Sampling theory states that the theoretical highest frequency is half of the sampling rate.

In this example, 320 data points are used for the calculation. Thus 160 in the X-axis corresponds to 160/320 = 0.5 Hz or interval 2. Similarly 107 in the middle of the X-axis corresponds to iterations about interval 320/107 = 3, 80 is from interval 320/80 = 4, and so on. Interval 4 is caused by the iterative search of 3 child nodes. For example, depth-first search from a level 4 node followed by three leaf nodes, which is the sequence of 4,5,5,5,4,5,5,5,... in the time trace, can cause frequency components at the interval 4. In general in depth-first search, frequency components at interval *i* correspond to search on the node which has *i-1* child nodes.

As this example came from discrete numbers, the other interval components between these values cannot technically exist, though they may exist in practice. They are called *harmonics* and are caused by decomposing non-sine waves into sine waves in the fourier transform.

The frequency distribution shown in figure 5.2.3 was generated from the time trace of depth-first search on a fully balanced tree with b=3. There is a big clear peak at about 75, which corresponds to interval 310/75 ≈ 4 in this distribution. If the search trees are balanced and have exactly the same branching factors in each level, frequency distributions tend to have clear peaks, though they still have harmonics.

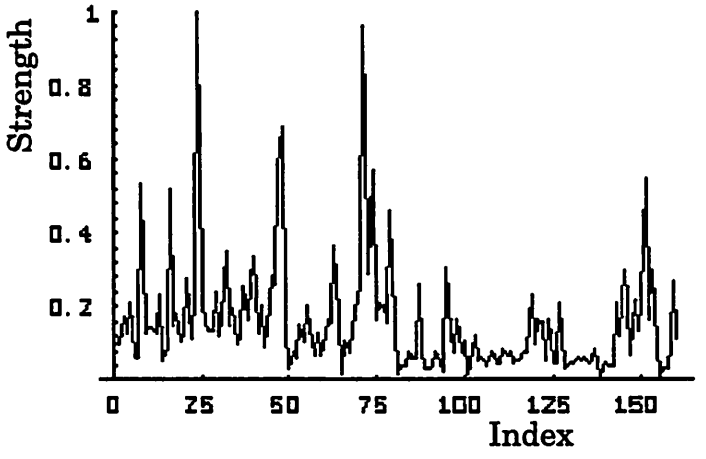


Figure 5.2.3 Frequency Distribution of Depth-first Search on a Fully Balanced Tree

5.2.2 Smoothing Filter

In the fourier analysis of real search histories, noise in the original function makes analysis difficult. We can apply a smoothing filter to the frequency distribution to reduce noise and to make the distribution clear.

Although we can design any filter in practice, the most common is the mean filter with a fixed window size. It works as a low-pass (high-cut) filter because the noise is often in the high range of frequency distributions. Equation (5.2.3) is used as the simple mean filter of window size 2.

$$MeanSmoothed[a(i)] = \frac{a(i)+a(i+1)}{2} \qquad (5.2.3)$$

In general, the mean smoothing filter with window size w is described by the following formula:

$$MeanSmoothed[a(i),w] = \frac{1}{W}\sum_{k=i-w/2}^{i+w/2} a(i) \qquad (5.2.4)$$

Note here that applying a mean filter with window size 2 twice to a function f is equivalent to applying a quarter-half-quarter filter to the function f once.

We can also design a high-pass filter or other filters, but for smoothing purposes, only low-pass filters make sense.[21] We should look for the filter that most clearly reveals the frequency components.

It is important that smoothing filters not be applied to time domain data (before fourier transform), but rather to the frequency domain (after fourier transform). If applied to the time domain, a filter reduces high frequency components of the original wave form. As a result, the frequency distribution would be different from the original form. Applying a low pass filter to the time domain is equivalent to ignoring the high frequencies in the frequency domain. Because we are interested in trend in the frequency domain, we should apply the smoothing filter to the frequency distribution.

The following graph was generated by applying a mean smoothing filter four times to the frequency distribution shown in figure 5.2.1. Note that spikes in the unsmoothed distribution have been reduced by smoothing so that patterns in the frequency spectrum become clearer. To get an idea which frequency components really do exist and which do not, we need to check the figures from the time domain.

By smoothing, frequency components at intervals 3 and 4, located at index 107 and 80 respectively, have been enhanced. However, it is still difficult to distinguish other components between these levels that are caused by harmonics of low frequency components.

Note here that the smoothing filter must be designed so that it doesn't filter out components which interest us. Thus designing these filters requires knowledge of the frequency components that should exist in the observations. We need to check the time trace of the search first in order to get some idea which part of the frequency components are important.

---

[21] We can use the same fourier technique to check the frequency response of any filter. This is the most common usage of the fourier transform in electrical engineering.

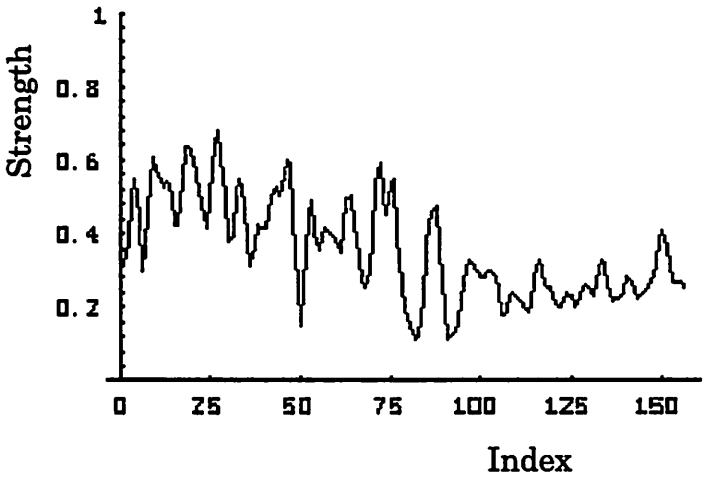


Figure 5.2.3 Smoothed frequency distribution of depth-first search with mean filter
A smoothing filter is applied to the frequency distribution shown in figure 5.2.1. The smoothing filter can reduce spikes, revealing the main components of the distribution.

5.2.3 Examples from Fourier Analysis of Search Algorithms

A Simple Example of Fourier Transform on Search

The following figure was generated from the history of a breadth-first search. We should see no backtracking in this trace because the algorithm scans all nodes from shallower to deeper levels once and never visits the same level again. Therefore, there should be no frequency components in the frequency domain. In practical applications of the fourier transform, it is assumed that the wave form is sampled from an infinite sequence, and furthermore, that the rest of the wave is exactly the same as the observed part. This assumption causes us to find low frequency components even though the original function has no iterative components. As a result, the frequency distribution has only low frequency components, which are basically noise.

This result should be contrasted to the depth-first search shown in figure 5.2.5. Here, lots of backtracking causes high-frequency components in the frequency domain. Therefore, the frequency distribution of depth-first search includes more high frequency components than breadth-first search. In general, high frequency components correspond to backtracking with a short period, which can only occur in the lowest levels of a search tree. Low frequency components correspond to long period backtracking, which can occur between high levels of search trees.

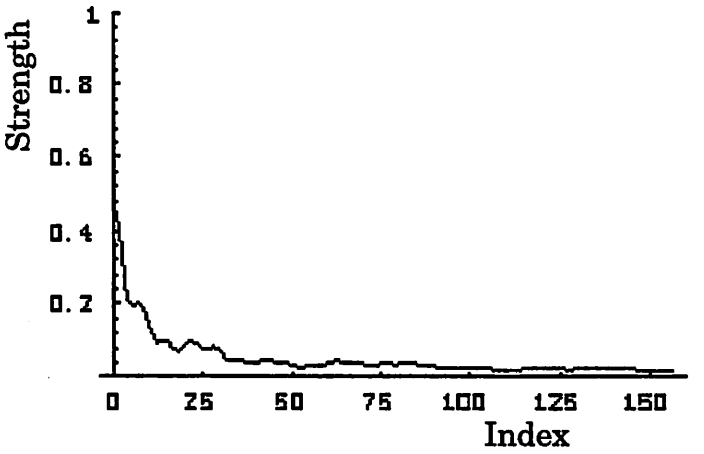
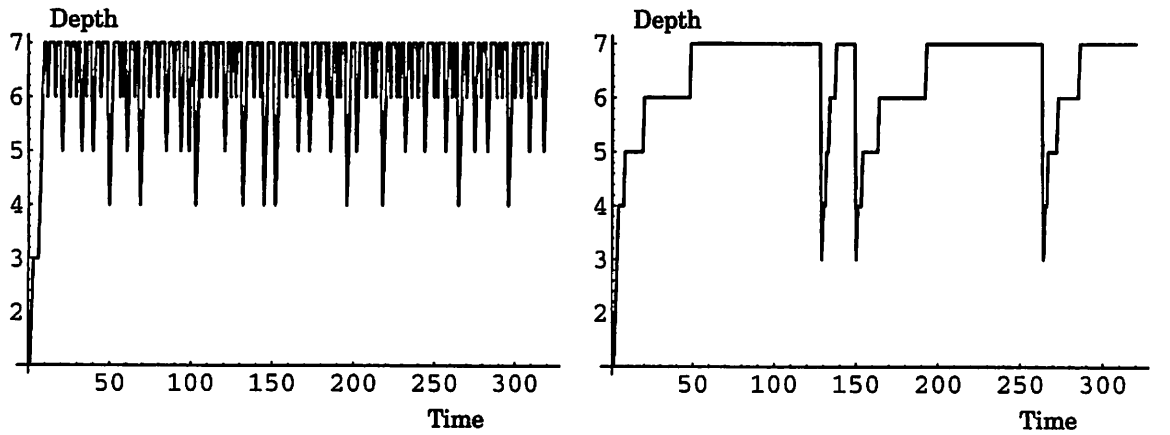


Figure 5.2.4 Frequency Distribution of Breadth-first Search[22] Only low frequency components exit in this distribution, which is called iterative noise. (This phenomenon results from using a finite number of sample points.)

Frequency Distribution of Combined Search

To demonstrate the utility of the fourier transform, the figures in 5.2.5 are created from combined-search algorithms. The Breath-Depth algorithm, shown as (A), uses breadth-first search until it reaches depth 4, then uses depth-first search. On the contrary, the Depth-Breadth algorithm in (B) uses depth-first search until level 4, and then uses breadth-first search.



(A) Breadth-Depth Search (B) Depth-Breadth Search

Figure 5.2.5 Time Trace of Breadth-Depth Search and Depth-Breadth Search Time trace of two combined search algorithms. The Breath-Depth algorithm, shown as (A), uses breadth-first search until it reaches depth 4, then uses depth-first search. In contrast, the Depth-Breadth algorithm in (B) uses depth-first search until level 4, then breadth-first search.

[22] A mean smooth filter was applied four times to generate this graph.

In the time domain, it is difficult to compare two time traces of search depth directly. Even though we can calculate the correlation of two time traces by changing the offset of one curve, we need to compare this correlation *for each possible offset* in order to find the best value. For example, imagine the case shown in figure 5.2.6. The same algorithm is used in two almost identical search trees; one search tree has depth = 4 (shown in A) and the other also has depth = 4 but one extra branch at depth 2 (shown in B). Because of the extra branch, the time trace of the second search tree has a delay, thus we will find the best correlation with an unknown offset. As we don't have any idea about this offset value in general, we need to iterate this calculation many times. This situation will become even more complicated when we compare time traces from dissimilar trees.

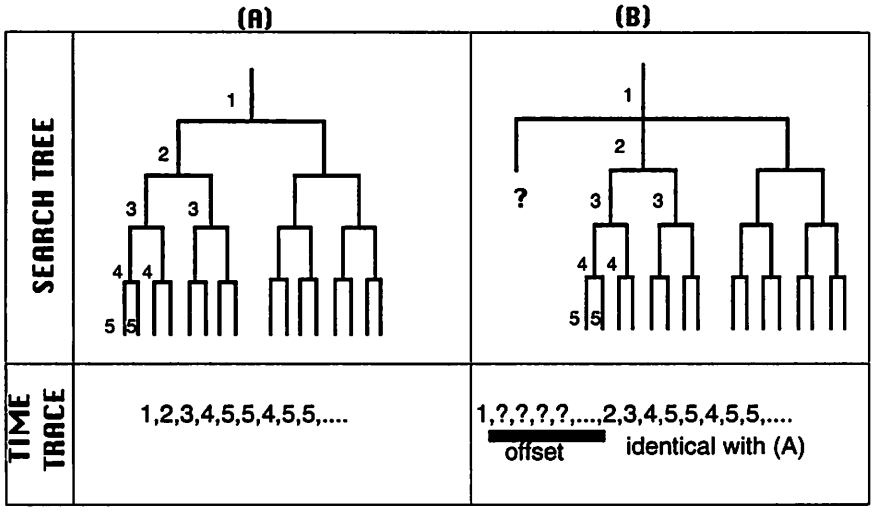
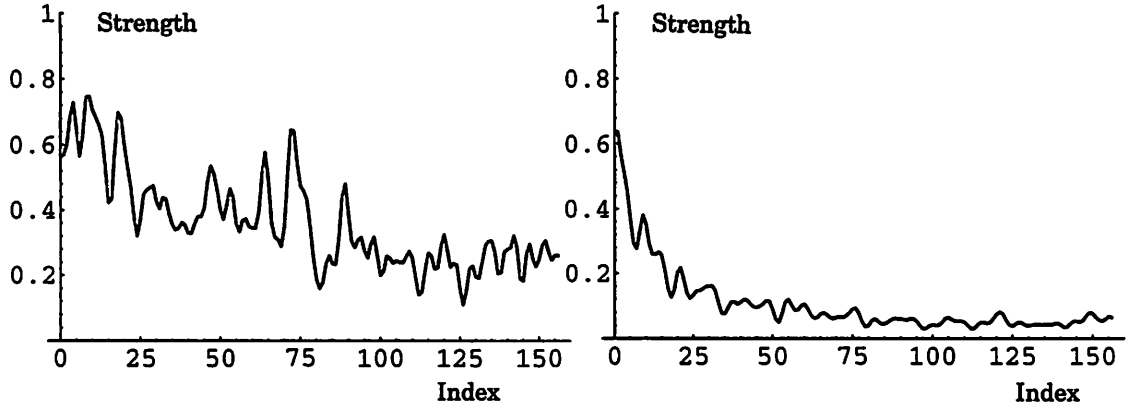


Figure 5.2.6 Problem of the correlation of time traces (B) has almost identical search tree with (A) but has an extra branch at level 2. As a result of this extra branch, the time traces must be offset to find the best correlation.

Once time traces are converted into frequency distributions they are directly comparable – a major benefit for the use of frequency analysis.

By using the frequency distribution representation we can easily discuss breadth-first and depth-first components of these two search algorithms. The distributions are shown in figure 5.2.7.

The frequency distribution of Depth-Breadth search shown in figure 5.2.7(A) has strong spikes at the same places as the pure depth-first search algorithms shown in figure 5.2.2. Thus we can predict a similarity between Depth-Breadth search and pure depth-first search. The frequency distribution of Breadth-Depth search, shown as figure 5.2.7(B), has only low frequency components and is similar to pure breadth-first search shown in figure 5.2.3.



(A) Breadth-Depth Search (B) Depth-Breadth Search

Figure 5.2.7 Frequency Distributions of Breadth-Depth Search and Depth-Breadth Search Frequencies included in the Breadth-Depth search in (A) include many more high frequency components, which were generated by depth-first components. The distribution from the Depth-Breadth search in (B) has only low components. These are basically noise caused by errors of transformation and harmonics. This distribution is very similar to those produced by pure breadth-first search algorithms.

Frequency Distribution of IDA* Search on Eight Puzzle

As another example, the following figures represent fourier transforms of IDA* search on the Eight Puzzle. A mean smoothing filter was applied four times in the frequency domain.

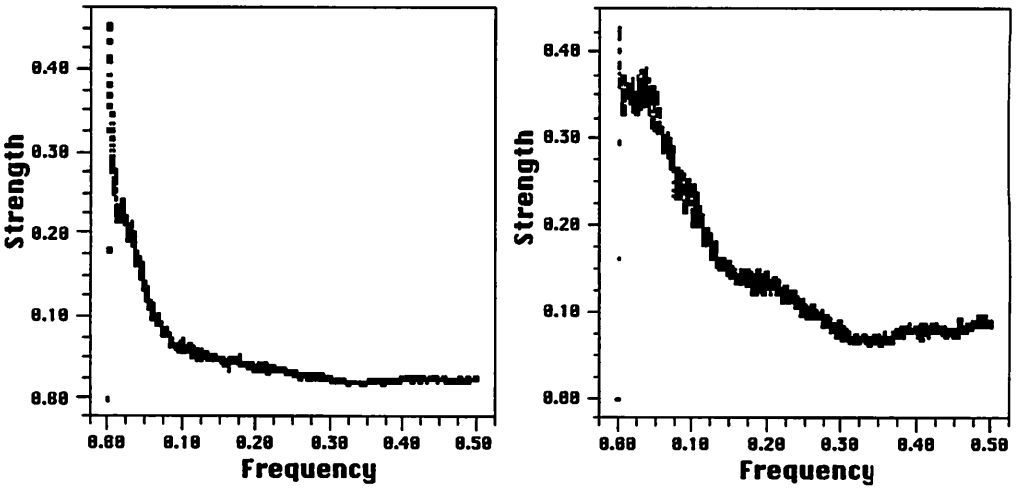


Figure 5.2.8 Frequency Distribution of IDA* Search on the Eight Puzzle (A) with Manhattan distance (B) with count final position

Looking at these two figures we can compare the behavior of the two heuristic functions. The frequency distribution from IDA* search with Manhattan distance (A) only includes lower frequencies, especially when compared with final position count (B).

This indicates less backtracking in lower levels of the tree. The search works by going down to the deeper levels, then does a little backtracking followed by a return to higher levels. The heuristic function helps by pruning bad branches in early stages of the search, and as a result, many big backtracking jumps are observed. On the contrary, in the case of the count final position function, shown in (B), the frequency distribution includes relatively many high frequency components. This means more backtracking occurs in both high levels and low levels, from which we can conclude that the heuristic function didn't help much to prune branches.

Figure 5.2.9 is the frequency distribution of pure depth-first search on the Eight Puzzle. The number of successors is either 1, 2, or 3 and depends on the current state, thus the branching factor of the puzzle is not constant. Therefore, the distribution is more scattered than the distributions in figure 5.2.8.

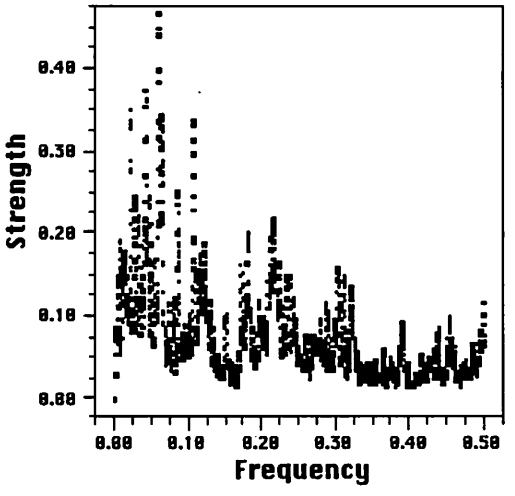


Figure 5.2.9 Frequency Distribution of Pure Depth-first Search on the Eight Puzzle
There are strong peaks at branches of 3, 4 and 5.

5.3 Interval Counting Methods

The interval counting representation is an alternative way to view the intervals of wave forms. Because the fourier analysis of time traces has the problem of harmonics, we cannot get a clear picture from fourier analysis without some knowledge about which frequencies to use in a smoothing filter. In contrast, the interval counting method can be used without any knowledge of frequencies. Using numbers, this method combines the representation of sub-tree and branching factors from frequency analysis (though it only works if the search is done in depth-first order).

Suppose we have a time trace of search depth in some algorithm shown as figure 5.3.1. We can measure the intervals between occurrences of the same search depth. For example, depth 4 appears four times in this search and these intervals are 2, 1 and 1.

We can count the intervals between different search depths and fill out a table, as shown in table 5.3.1.

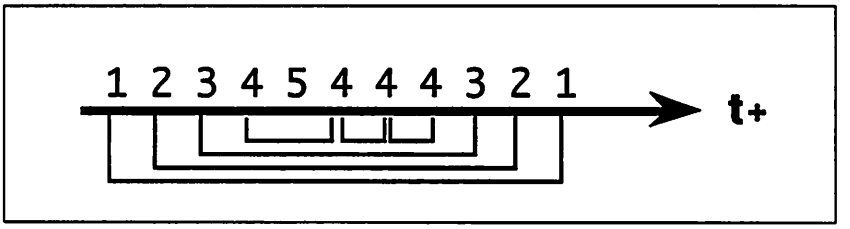


Figure 5.3.1 Interval Counting Method The numbers on the arrow are depths of nodes visited. To calculate the interval for each level, it is necessary to find the next occurrence of a node at that level in the sequence, and count the number of nodes from the initial occurrence to the next. For example, 2 is found at t=2 and t=10, thus the interval of 2 is 8.

Table 5.3.1 Example of Interval Counting Table

Interval	Depth = 1	Depth = 2	Depth = 3	Depth = 4	Depth = 5	Depth = 6
2				**2**		
3				**1**		
4						
5						
6						
7			**1**			
8						
9		**1**				
10						

In the fourier transform, the intervals are counted between all pairs of the same depth, which makes it difficult to interpret periodicity in their behaviors. For example, for depth 4 in figure 5.3.1, we will find frequency components of interval 2 (twice), 3 (twice), 4 (once), and 5 (once) from the fourier transform. The only interesting frequency components in this list are neighboring levels because we are interested in the smallest intervals and interval 5 isn't one of the smallest intervals of depth 4 here. Only interval 2 (twice) and 3 (once) should be counted, and the other components should be ignored, as shown in figure 5.3.2.

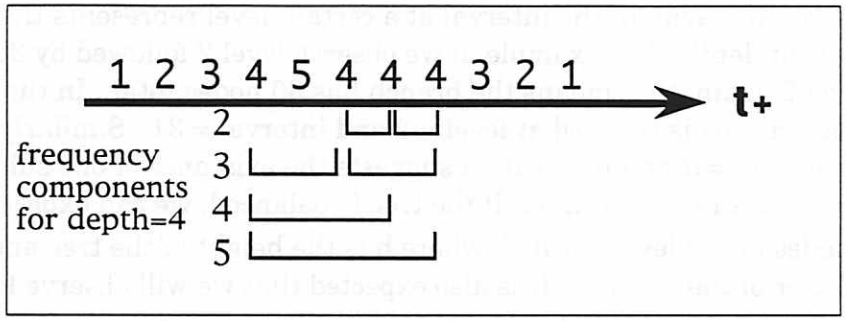


Figure 5.3.2 Frequency Components for depth=4 An analogy to the fourier transform is found in interval counting. In a fourier transform, all combinations of the interval shown in this figure would be counted, which would obscure rather than reveal the frequency components.

The following two figures are examples of interval counting from depth-first search and breadth-first search, graphed as three-dimensional bar-plots. In these figures, the X-axis represents the interval, the Y-axis represents the level and the Z-axis represents frequency of appearances in the histories of the search. Note that a logarithmic function was applied to frequencies to make small components visible on the same scale with the bigger values, which are of different magnitude.

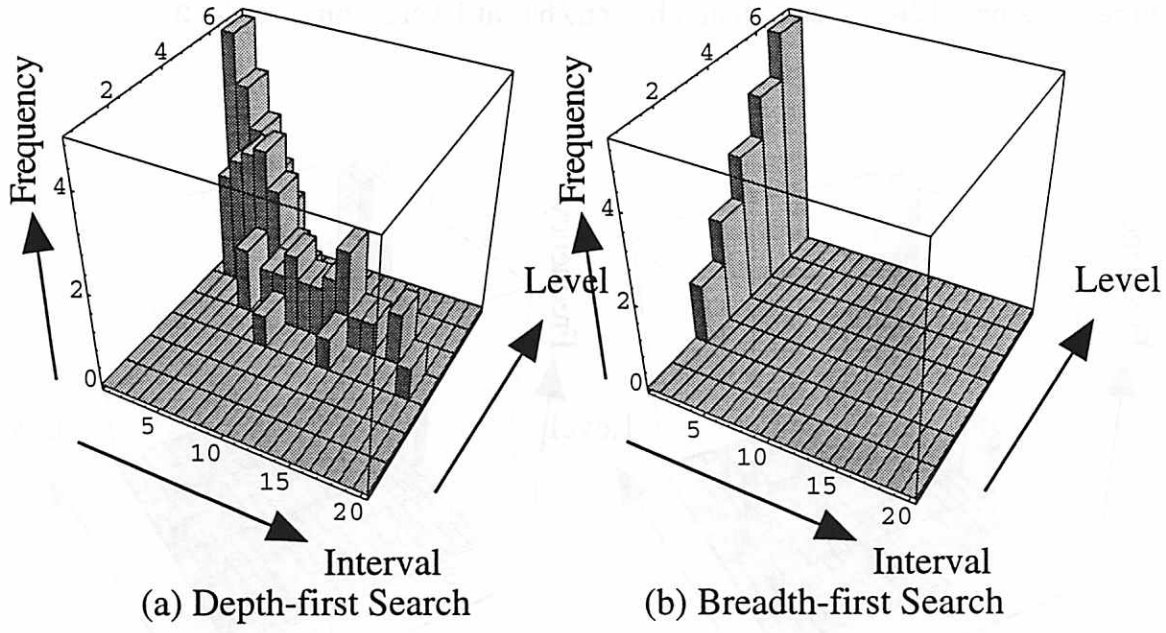


Figure 5.3.3 Three Dimensional Bar Plot of (a) Depth-first Search, (b) Breadth-first Search in a randomly pruned tree In these graphs, only the counts which are less than 20 intervals are shown. There are lots of long interval components in the depth-first search shown as (a).[23] On the contrary, in the breadth-first search graph, only components which have an interval of 2 exist.

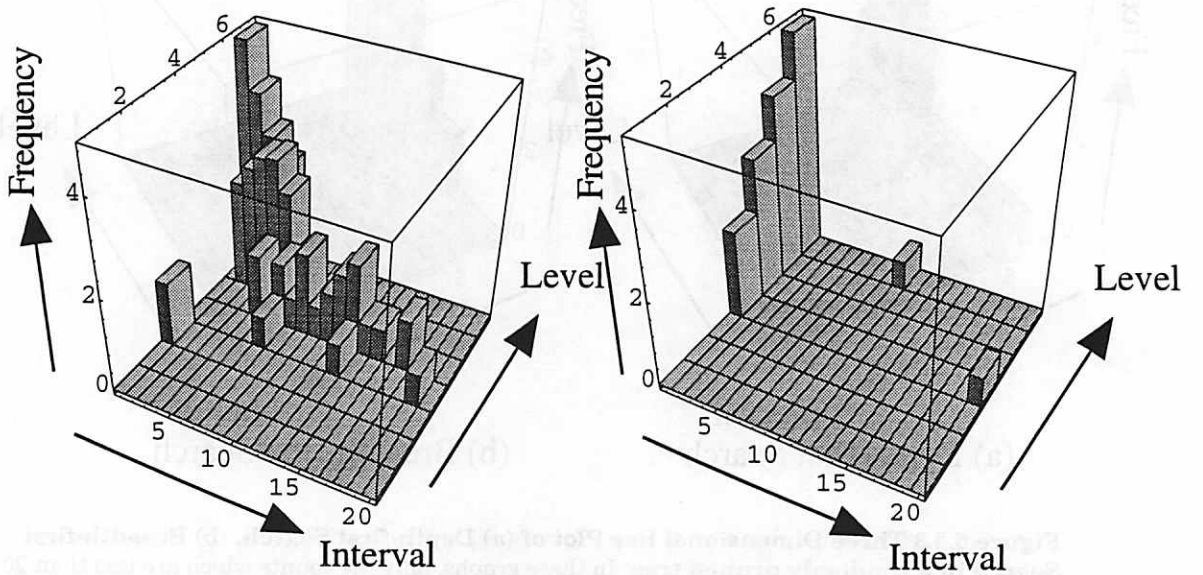
[23] Note that counts smaller than depth 3 in depth-first search have very big interval s and are located in the far right positions of this plot.

In depth-first search, the interval at a certain level represents the number of nodes below that depth. For example, if we observe level 2 followed by 30 lower levels, and then level 2 again, that means the branch has 30 nodes total. In the interval counting method, this is counted at level = 2 and interval = 31. Similarly, the observation of level = d and interval = i suggests the existance of one subtree at level d which has i-1 nodes in the subtree. If the tree is balanced, we can expect that the total number of nodes below level i is b^{h-i}, where h is the height of the tree and b is the branching factor of the subtree. It is also expected that we will observe level = d and interval = i, b^i times in a fully balanced tree with the same parameters.

In breadth-first search, we see no depth other than interval 2 using interval counting because breadth-first search progresses level by level, yielding only intervals of 2. In this graph, the heights of bars of interval 2 represent exactly the same number of nodes at each level.

The following figures show interval counting of the time traces for the search algorithms described in section 5.1.

Breadth-Depth-first search in figure 5.3.4(a) uses breadth-first search down to level 3, and then switches to depth-first search. Thus the interval counting graph becomes exactly the same graph as depth-first search below level 3. Above level 2 it is the same as breadth-first search and has one bar at level 2 and interval 2.



(a) Breadth-Depth-first Search (b) Depth-Breadth-first Search

Figure 5.3.4 Three Dimensional Bar Plot of (a) Breadth-depth-first search, (b) Depth-breadth-first search in a randomly pruned tree

On the other hand, Depth-Breadth-first search in figure 5.3.4(b) lacks the bar of breadth-first search at interval 2. It has bars at interval 2 above level 3 instead, which represents breadth-first search components. This means the search uses breadth-first search methods below level 3. Two small bars in a bigger interval area were generated by counting the same level of a different branch. These small components are regarded as ghost images and can be avoided by not counting an interval in the same level but different branches.

6. Summary of Representations of Search

7. Conclusions

Theoretical analysis techniques are better than experimental analysis. Sometimes, it is very difficult to build theoretical models and analyze them when problems become complex, or heuristic functions are used to solve the problems. In the analysis of search algorithms with heuristic functions, it is very hard to formalize the problem and to solve it by theoretically even in a small problem like the Eight Puzzle. Visualization tools for the search algorithms are important in these cases. They can also be applied to debugging search algorithms in practice.

Time domain analysis of search algorithms are shown in three forms:

(a) time trace to see at a glance the behavior of search

(b) frequency domain to analyze iterative action of search algorithms

(c) interval counting to decompose search algorithms into depth and breadth components.

Although these tools cannot be directly applied to arbitrary cases, the effort to collect such tools and to generalize them is very important. We need to continue developing such tools as part of the study of algorithms.

A. Search Algorithms used in Examples

A.1 Pseudo code of Depth-first, Breadth-first and Best-first Search

```
/* General Search Function
searches a tree defined by node_list and function successor() in the
order of strategy() function */
general_search(node_list, strategy)
{
     /* Check if there is a node */
     if( empty(node_list) )
          return (NOT_FOUND);

     current_node = first(node_list);
     alternative_nodes = rest(node_list);

     /* Check whether the current node is a goal */
     if( goalp(current_node) )
          return(current_node);

     /* Expand successor nodes */
     successor_node = successors(current_node);

     /* Re-calculate search order
     strategy() function returns the order of successor nodes */
     node_order = strategy(alternative_nodes, successor_node);

     /* Call self recursively */
     return (general_search(node_order, strategy));
}

/* Depth-first Search
function depth_first performs depth-first search over the tree defined
by node_list and successor() function
 */
depth_first(node_list)
{
     general_search(node_list, *depth_first_order());
}
depth_first_order(node_list, successor_list)
{
     return(append(successor_list, node_list));
}

/* Breadth-first Search
function breadth_first performs breadth-first search over the tree
defined by node_list and successor() function
 */
breadth_first(node_list)
{
     general_search(node_list, *breadth_first_order());
}
breadth_first_order(node_list, successor_list)
{
     return(append(node_list, successor_list));
}
```

```
/* Best-first Search
function best_first performs best-first search over the tree defined
by node_list and successor() function
 */
best_first(node_list)
{
     general_search(node_list, *best_first_order());
}
best_first_order(node_list, successor_list)
{
     return(sort_by_priority(append(node_list, successor_list)));
}
```

A.2 Pseudo code of A* Search

```
/*
function A* performs A* search over the tree defined by node_list and
successor() function
*/
a*(node_list)
{
     /* Check if there is a node */
     if( empty(node_list) )
          return (NOT_FOUND);

     current_node = first(node_list);
     alternative_nodes = rest(node_list);

     /* Check whether the current node is a goal */
     if( goalp(current_node) )
          return(current_node);

     /* Expand successor nodes */
     successor_node = successors(current_node);
     all_successors = append(alternative_nodes, successor_node);

     /* Re-calculate search order */
     node_order = sort(all_successors, f($node));

     /* Call self recursively */
     return (a*(node_order));
}

f(node)
{
     return (g(node) + h(node));
}

g(node)
{
     /* use depth from root as f value */
     return depth(node);
}
/* Heuristic Function */
h(node)
{
     return manhattan_distance(node);
}
```

A.3 Pseudo code of IDA* Search

```
/*
function ida* performs ida* search until MAX_TH over the tree defined
by root_node and successor() function
*/
ida*(root_node)
{
     /* initial threshold */
     th = f(root_node);

     /* id search */
     loop {
          smallest_f = +INFINITY;
          th = dfida*(root_node, th);
     } until (goalp(th) || th > MAX_TH);

     if (goalp(th))
          return (th);
     else
          return (NOT_FOUND);
}

/* This global variable is used to record the smallest f value in one
iteration. */
global variable smallest_f = +INFINITY;

/*
function dfida* performs depth-first ida* search over the tree defined
by node_list and successor() function
*/
dfida*(current_node, threshold)
{
     /* Check if there is a node */
     if( empty(current_node) )
          return (smallest_f);

     /* Check whether the current node is a goal */
     if( goalp(current_node) )
          return (current_node);

     current_f = f(current_node);

     /* Check threshold level */
     if ( current_f > threshold) {
          /* record for the next iteration */
          if (current_f < smallest_f )
              smallest_f = current_f;
     } else {
          /* Perform depth-first search of successor nodes */
          foreach (successor in successors(current_node)) {
              results = dfida*(successor, threshold)
              if (goalp(results))
                    return (results);
          }
     }
     return (smallest_f_value);
}
```

A4. The Eight Puzzle and Heuristic functions used in examples

The Eight Puzzle has 8 pieces labeled from 1 to 8, and one blank space in the 3 x 3 square board. As the initial condition, a randomly arranged state is given. The goal of this puzzle is to reach the state shown below. One can move one piece neighboring the blank space to the blank space at a time. Other movements, such as swapping two pieces, are illegal. A solution with fewer moves than another solution is considered better.

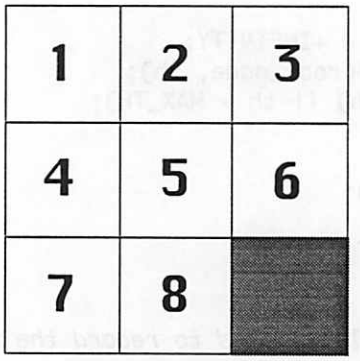


Figure A4.1 Eight Puzzle

In A* and IDA* search algorithms, heuristic functions are used to estimate the distance between the current state and the goal state.

Manhattan distance

Manhattan distance is given by the sum of the difference between the final position and the current position. Two slightly different functions are used to demonstrate the visualization of the heuristic function versus the depth of search in section 4. The following two formulas define these functions. The first function considers the blank space (i=9) as a part of the calculation of distance; the second function ignores the blank space.

$$\text{ManhattanDistance1} = \sum_{i=1}^{9}\left(|x_i - x_f| + |y_i - y_f|\right) \qquad \text{(A4.1)}$$

$$\text{ManhattanDistance2} = \sum_{i=1}^{8}\left(|x_i - x_f| + |y_i - y_f|\right) \qquad \text{(A4.2)}$$

Note that in definition 1, one move can change the value of the Manhatten distance of this function by +2, 0, or -2. Definition 2 changes the Manhatten distance by +1 or -1 with each move.

Count Final Position

Another definition of measurement used in section 4 is called 'count final position', which is defined as follows:

$$\text{Count Final Position1} = \sum_{i=1}^{9}\left(|x_i \neq x_f| \vee |y_i \neq y_f|\right) \quad \text{(A4.3)}$$

This function returns an integer between 0 and 9 and can change by +1, 0 or -1 in one move. This function is a poor estimator for distance because moves that decrease function value are not necessarily good ones.

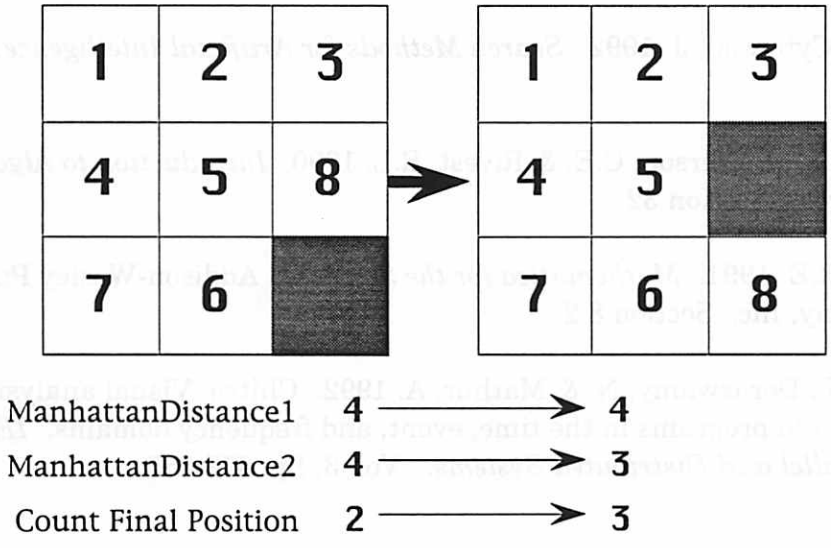


Figure A4.2 Example of Heuristic Values

In this example, the move shown is necessary to reach the goal. However, according to *ManhattanDistance1* the estimated distance from the current state to the solution state remains unchanged; according to *CountFinalPosition*, the estimated distance actually increases!

References

Korf, R.E. 1987. Real-Time Heuristic Search: First Results. *Proceedings of the Sixth National Conference on Artificial Intelligence*. Morgan Kaufmann. Pp. 133-138.

Korf, R.E. 1988. Real-Time Heuristic Search: New Results. *Proceedings of the Seventh National Conference on Artificial Intelligence*. Morgan Kaufmann. Pp. 139-144

Bolc, L. & Cytowski, J. 1992. *Search Methods for Artificial Intelligence*. Academic Press.

Cormen, T.H., Leiserson, C.E. & Rivest, R.L. 1990. *Introduction to Algorithms*. The MIT Press. Section 32.

Crandall, R.E. 1991. *Mathematica for the Sciences*. Addison-Wesley Publishing Company, Inc. Section 8.2.

Abrams, M., Doraswamy, N. & Mathur, A. 1992. Chitra: Visual analysis of parallel and distributed programs in the time, event, and frequency domains. *IEEE Transactions on Parallel and Distributed Systems.* Vol. 3, Pp. 672-685.