4

HEURISTIC SEARCH

The task that a symbol system is faced with, then, when it is presented with a problem and a problem space, is to use its limited processing resources to generate possible solutions, one after another, until it finds one that satisfies the problem defining test. If the symbol system had some control over the order in which potential solutions were generated, then it would be desirable to arrange this order of generation so that actual solutions would have a high likelihood of appearing early. A symbol system would exhibit intelligence to the extent that it succeeded in doing this. Intelligence for a system with limited processing resources consists in making wise choices of what to do next. . . .

—NEWELL AND SIMON, 1976, Turing Award Lecture

I been searchin' . . .
Searchin' . . . Oh yeah,
Searchin' every which-a-way . . .

—LIEBER AND STOLLER

4.0 Introduction

George Polya defines *heuristic* as "the study of the methods and rules of discovery and invention" (Polya 1945). This meaning can be traced to the term's Greek root, the verb *eurisco*, which means "I discover." When Archimedes emerged from his famous bath clutching the golden crown, he shouted "Eureka!" meaning "I have found it!". In state space search, *heuristics* are formalized as rules for choosing those branches in a state space that are most likely to lead to an acceptable problem solution.

AI problem solvers employ heuristics in two basic situations:

1. A problem may not have an exact solution because of inherent ambiguities in the problem statement or available data. Medical diagnosis is an example of this. A given set of symptoms may have several possible causes; doctors use heuristics

to choose the most likely diagnosis and formulate a plan of treatment. Vision is another example of an inexact problem. Visual scenes are often ambiguous, allowing multiple interpretations of the connectedness, extent, and orientation of objects. Optical illusions exemplify these ambiguities. Vision systems often use heuristics to select the most likely of several possible interpretations of as scene.

2. A problem may have an exact solution, but the computational cost of finding it may be prohibitive. In many problems (such as chess), state space growth is combinatorially explosive, with the number of possible states increasing exponentially or factorially with the depth of the search. In these cases, exhaustive, *brute-force* search techniques such as depth-first or breadth-first search may fail to find a solution within any practical length of time. Heuristics attack this complexity by guiding the search along the most "promising" path through the space. By eliminating unpromising states and their descendants from consideration, a heuristic algorithm can (its designer hopes) defeat this *combinatorial explosion* and find an acceptable solution.

Unfortunately, like all rules of discovery and invention, heuristics are fallible. A heuristic is only an informed guess of the next step to be taken in solving a problem. It is often based on experience or intuition. Because heuristics use limited information, such as knowledge of the present situation or descriptions of states currently on the open list, they are not always able to predict the exact behavior of the state space farther along in the search. A heuristic can lead a search algorithm to a suboptimal solution or fail to find any solution at all. This is an inherent limitation of heuristic search. It cannot be eliminated by "better" heuristics or more efficient search algorithms (Garey and Johnson 1979).

Heuristics and the design of algorithms to implement heuristic search have long been a core concern of artificial intelligence. Game playing and theorem proving are two of the oldest applications in artificial intelligence; both of these require heuristics to prune spaces of possible solutions. It is not feasible to examine every inference that can be made in a mathematics domain or every possible move that can be made on a chessboard. Heuristic search is often the only practical answer.

Expert systems research has affirmed the importance of heuristics as an essential component of problem solving. When a human expert solves a problem, he or she examines the available information and makes a decision. The "rules of thumb" that a human expert uses to solve problems efficiently are largely heuristic in nature. These heuristics are extracted and formalized by expert systems designers, as we see in Chapter 8.

It is useful to think of heuristic search from two perspectives: the heuristic measure and an algorithm that uses heuristics to search the state space. In Section 4.1, we implement heuristics with *hill-climbing* and *dynamic programming* algorithms. In Section 4.2 we present an algorithm for *best-first* search. The design and evaluation of the effectiveness of heuristics is presented in Section 4.3, and game playing heuristics in Section 4.4.

Consider heuristics in the game of tic-tac-toe, Figure II.5. The combinatorics for exhaustive search are high but not insurmountable. Each of the nine first moves has eight possible responses, which in turn have seven continuing moves, and so on. A simple analysis puts the total number of states for exhaustive search at $9 \times 8 \times 7 \times \cdots$ or 9!.

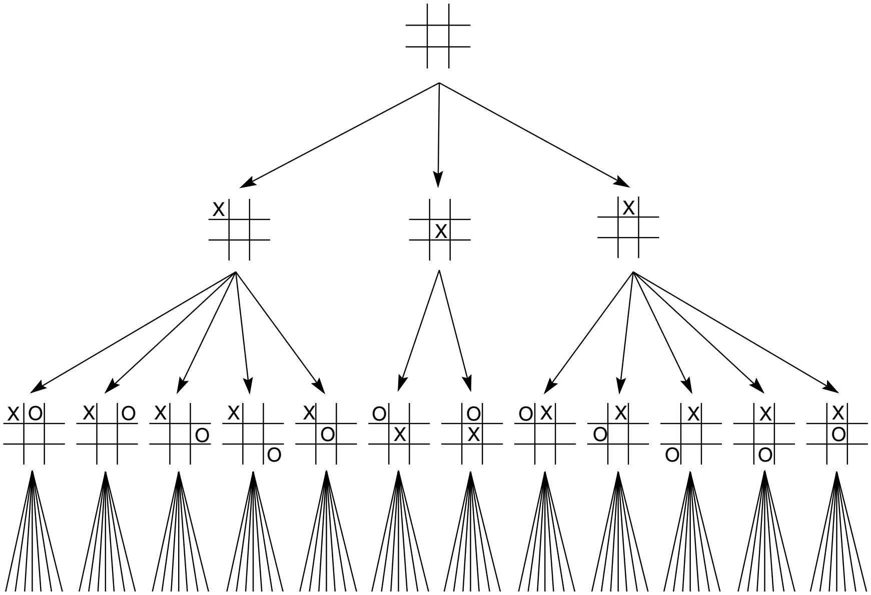
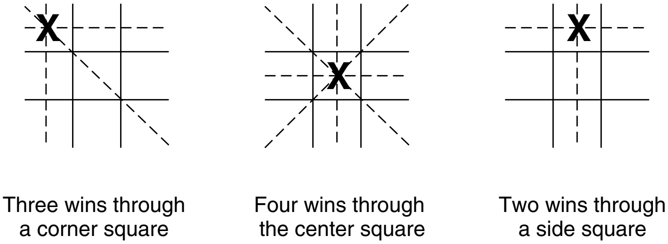


Figure 4.1 First three levels of the tic-tac-toe state space reduced by symmetry.

Symmetry reduction decreases the search space. Many problem configurations are actually equivalent under symmetric operations of the game board. Thus, there are not nine but really three initial moves: to a corner, to the center of a side, and to the center of the grid. Use of symmetry on the second level further reduces the number of paths through the space to 12 × 7!, as seen in Figure 4.1. Symmetries in a game space such as this may be described as mathematical invariants, that, when they exist, can often be used to tremendous advantage in reducing search.

A simple heuristic, however, can almost eliminate search entirely: we may move to the state in which X has the most winning opportunities. (The first three states in the tic-tac-toe game are so measured in Figure 4.2.) In case of states with equal numbers of potential wins, take the first such state found. The algorithm then selects and moves to the state with the highest number of opportunities. In this case X takes the center of the grid. Note that not only are the other two alternatives eliminated, but so are all their descendants. Two-thirds of the full space is pruned away with the first move, Figure 4.3.

After the first move, the opponent can choose either of two alternative moves (as seen in Figure 4.3). Whichever is chosen, the heuristic can be applied to the resulting state of the game, again using “most winning opportunities” to select among the possible moves. As search continues, each move evaluates the children of a single node; exhaustive search is not required. Figure 4.3 shows the reduced search after three steps in the game. States are marked with their heuristic values. Although not an exact calculation of search size for



Three wins through a corner square

Four wins through the center square

Two wins through a side square

Figure 4.2 The most wins heuristic applied to the first children in tic-tac-toe.

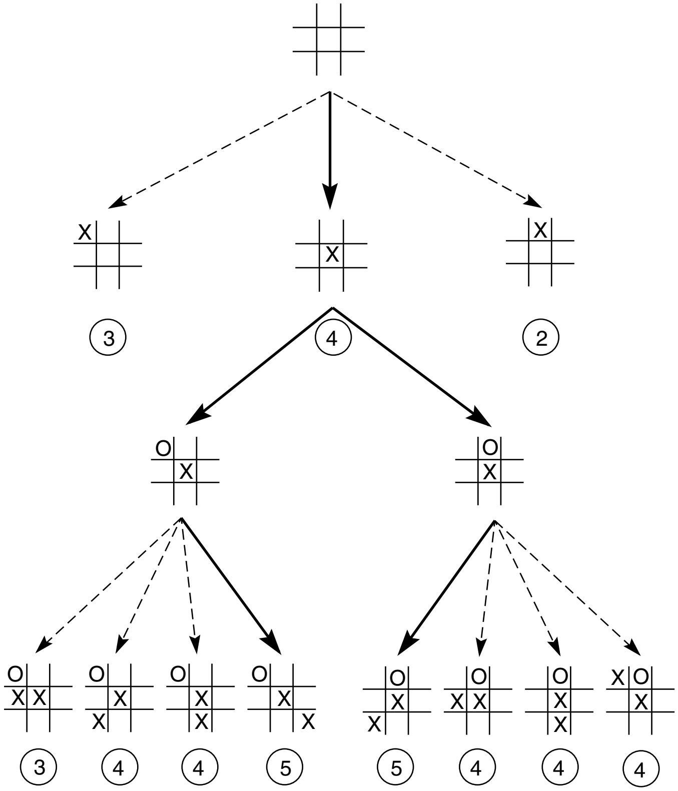


Figure 4.3 Heuristically reduced state space for tic-tac-toe.

this “most wins” strategy for tic-tac-toe, a crude upper bound can be computed by assuming a maximum of five moves in a game with five options per move. In reality, the number of states is smaller, as the board fills and reduces options. This crude bound of 25 states is an improvement of four orders of magnitude over 9!.

The next section presents two algorithms for implementing heuristic search: *hill-climbing* and *dynamic programming*. Section 4.2 uses the priority queue for *best-first* search. In Section 4.3 we discuss theoretical issues related to heuristic search, such as *admissibility* and *monotonicity.* Section 4.4 examines the use of *minimax* and *alpha-beta pruning* to apply heuristics to two-person games. The final section of Chapter 4 examines the complexity of heuristic search and reemphasizes its essential role in intelligent problem solving.

4.1 Hill-Climbing and Dynamic Programming

4.1.1 Hill-Climbing

The simplest way to implement heuristic search is through a procedure called *hill-climbing* (Pearl 1984). Hill-climbing strategies expand the current state of the search and evaluate its children. The "best" child is selected for further expansion; neither its siblings nor its parent are retained. Hill climbing is named for the strategy that might be used by an eager, but blind mountain climber: go uphill along the steepest possible path until you can go no farther up. Because it keeps no history, the algorithm cannot recover from failures of its strategy. An example of hill-climbing in tic-tac-toe was the "take the state with the most possible wins" that we demonstrated in Section 4.0.

A major problem of hill-climbing strategies is their tendency to become stuck at *local maxima*. If they reach a state that has a better evaluation than any of its children, the algorithm falters. If this state is not a goal, but just a local maximum, the algorithm may fail to find the best solution. That is, performance might well improve in a limited setting, but because of the shape of the entire space, it may never reach the overall best. An example of local maxima in games occurs in the 8-puzzle. Often, in order to move a particular tile to its destination, other tiles already in goal position need be moved out. This is necessary to solve the puzzle but temporarily worsens the board state. Because "better" need not be "best" in an absolute sense, search methods without backtracking or some other recovery mechanism are unable to distinguish between local and global maxima.

Figure 4.4 is an example of the local maximum dilemma. Suppose, exploring this search space, we arrive at state X, wanting to maximize state values. The evaluations of X's children, grand children, and great grandchildren demonstrate that hill-climbing can get confused even with multiple level look ahead. There are methods for getting around this problem, such as randomly perturbing the evaluation function, but in general there is no way of guaranteeing optimal performance with hill-climbing techniques. Samuel's (1959) checker program offers an interesting variant of the hill climbing algorithm.

Samuel's program was exceptional for its time, 1959, particularly given the limitations of the 1950s computers. Not only did Samuel's program apply heuristic search to checker playing, but it also implemented algorithms for optimal use of limited memory, as well as a simple form of learning. Indeed, it pioneered many of the techniques still used in game-playing and machine learning programs.

Samuel's program evaluated board states with a weighted sum of several different heuristic measures:

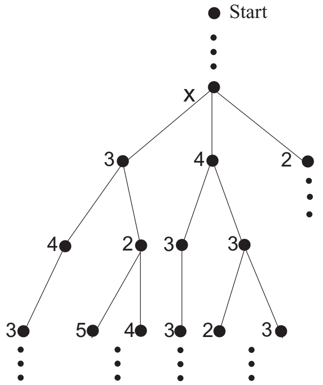


Figure 4.4 The local maximum problem for hill-climbing with 3-level look ahead.

$$\sum_i a_i x_i$$

The x_i in this sum represented features of the game board such as piece advantage, piece location, control of center board, opportunities to sacrifice pieces for advantage, and even a calculation of moments of inertia of one player's pieces about an axis of the board. The a_i coefficients of these x_i were specially tuned weights that tried to model the importance of that factor in the overall board evaluation. Thus, if piece advantage was more important than control of the center, the piece advantage coefficient would reflect this.

Samuel's program would look ahead in the search space the desired number of levels or *plies* (usually imposed by space and/or time limitations of the computer) and evaluate all the states at that level with the evaluation polynomial. Using a variation on *minimax* (Section 4.3), it propagated these values back up the graph. The checker player would then move to the best state; after the opponent's move, the process would be repeated for the new board state.

If the evaluation polynomial led to a losing series of moves, the program adjusted its coefficients in an attempt to improve performance. Evaluations with large coefficients were given most of the blame for losses and had their weights decreased, while smaller weights were increased to give these evaluations more influence. If the program won, the opposite was done. The program trained by playing either against a human partner or against another version of itself.

Samuel's program thus took a hill-climbing approach to learning, attempting to improve performance through local improvements on the evaluation polynomial. Samuel's checker player was able to improve its performance until it played a very good game of checkers. Samual addressed some of the limitations of hill climbing by checking the effectiveness of the individual weighted heuristic measures, and replacing the less effective. The program also retained certain interesting limitations. For example, because of a limited global strategy, it was vulnerable to evaluation functions leading to traps. The learning component of the program was also vulnerable to inconsistencies in opponent's play; for example, if the opponent used widely varying strategies, or simply played foolishly, the weights on the evaluation polynomial might begin to take on “random” values, leading to an overall degradation of performance.

4.1.2 Dynamic Programming

Dynamic programming (DP) is sometimes called the *forward-backward,* or, when using probabilities, the *Viterbi* algorithm. Creatred by Richard Bellman (1956), dynammic programming addresses the issue of restricted memory search in problems composed of multiple interacting and interrelated subproblems. DP keeps track of and reuses subproblems already searched and solved within the solution of the larger problem. An example of this would be the reuse the subseries solutions within the solution of the Fibonacci series. The technique of subproblem caching for reuse is sometimes called *memoizing* partial subgoal solutions. The result is an important algorithm often used for string matching, spell checking, and related areas in natural language processing (see Sections 9.4 and 14.4).

We demonstrate dynamic programming using two examples, both from text processing; first, finding an optimum global alignment of two strings of characters and, second, finding the minimum edit difference between two strings of characters. Suppose we wanted to find the best possible alignment for the characters in the strings BAADDCABDDA and BBADCBA. One optimal alignment, among several possible, would be:

```
BAADDCABDDA
BBA DC B  A
```

Dynamic programming requires a data structure to keep track of the subproblems related to the state currently being processed. We use an array. The size of the dimensions of this array, because of initialization requirements, is one more than the length of each string, in our example 12 by 8, as in Figure 4.5. The value of each row-column element of

	—	B	A	A	D	D	C	A	B	D	D	A
—	0	1	2	3	4	5	6	7	8	9	10	11
B	1	0										
B	2											
A	3											
D	4											
C	5											
B	6											
A	7											

Figure 4.5 The initialization stage and first step in completing the array for character alignment using dynamic programming.

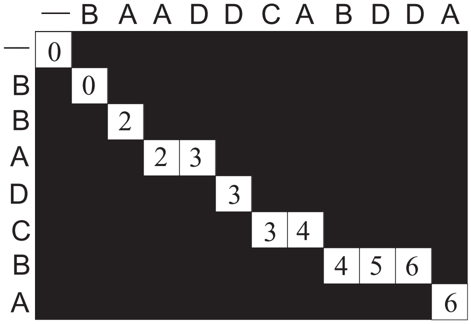
	—	B	A	A	D	D	C	A	B	D	D	A
—	0	1	2	3	4	5	6	7	8	9	10	11
B	1	0	1	2	3	4	5	6	7	8	9	10
B	2	1	2	3	4	5	6	7	6	7	8	9
A	3	2	1	2	3	4	5	6	7	8	9	8
D	4	3	2	3	2	3	4	5	6	7	8	9
C	5	4	3	4	3	4	3	4	5	6	7	8
B	6	5	6	5	4	5	4	5	4	5	6	7
A	7	6	5	4	5	6	5	4	5	6	7	6

Figure 4.6 The completed array reflecting the maximum alignment information for the strings.

the array reflects the global alignment success to that point in the matching process. There are three possible costs for creating the current state: if a character is shifted along in the shorter string for better possible alignment, the cost is 1 and is recorded by the array's *column* score. If a new character is inserted, the cost is 1 and reflected in the array's *row* score. If the characters to be aligned are different, shift and insert, the cost is 2; or if they are identical the cost is 0; this is reflected in the array's *diagonal*. Figure 4.5 shows initialization, the increasing +1s in the first row and column reflect the continued shifting or insertion of characters to the "_" or empty string.

In the *forward* stage of the dynamic programming algorithm, we fill the array from the upper left corner by considering the partial matching successes to the current point of the solution. That is, the value of the intersection of row x and column y, (x, y), is a function (for the minimum alignment problem, the minimum cost) of one of the three values in row x - 1 column y, row x - 1 column y - 1, or row x column y - 1. These three array locations hold the alignment information *up to* the present point of the solution. If there is a match of characters at location (x, y) add 0 to the value at location (x - 1, y - 1), if there is no match add 2 (for shift and insert). We add 1 by either shifting the shorter character string (add to the previous value of column y) or inserting a character (add to the previous value of row x). Continuing this approach produces the filled array of Figure 4.6. It can be observed that the minimum cost match often takes place near the upper-left to lower-right diagonal of the array.

Once the array is filled, we begin the *backward* stage of the algorithm that produces particular solutions. That is, from the best alignment possible, we go back through the array to select a specific solution alignment. We begin this process at the maximum row column value, in our example the 6 in row 7 column 12. From there we move back through the array, at each step selecting one of the immediate state's predecessors that produced the present state (from the *forward* stage), ie, one of either the diagonal, row, or column that produced that state. Whenever there is a forced decreasing difference, as in the 6 and 5 near the beginning of the trace back, we select the previous diagonal, as that is where the match came from; otherwise we use the value of the preceding row or column. The trace back of Figure 4.7, one of several possible, produces the optimal string alignment of the previous page.



	—	B	A	A	D	D	C	A	B	D	D	A
—	0											
B		0										
B			2									
A				2	3							
D						3						
C							3	4				
B									4	5	6	
A												6

Figure 4.7 A completed backward component of the dynamic programming example giving one (of several possible) string alignments.

In the second example use of DP, we consider the idea of the *minimum edit difference* between two strings. If we were building an intelligent spell checker, for example, we might want to determine what words from our set of correctly spelled words best approximate a certain unrecognized string of characters. A similar approach could be used to determine what known (strings of phones representing) words most closely matched a given string of phones. The next example of establishing the *edit distance* between two strings of characters is adapted from Jurafsky and Martin (2009).

Suppose you produce an unrecognized string of characters. The task of the spell checker is to produce an ordered list of the most likely words from the dictionary that you meant to type. The question is then, how can a difference be measured between pairs of strings of characters, ie, the string you just typed and the character strings of a dictionary. For your string, we want to produce an ordered list of possibly correct words in the dictionary. For each of these words, we want a numerical measure of how "different" each word is from your string.

A *minimum edit difference* between two strings can be specified as the number of character insertions, deletions, and replacements necessary to turn the first string, the source, into the second string, the target. This is sometimes called the *Levenshtein distance* (Jurafsky and Martin 2009). We now implement a dynamic programming search for determining the minimum edit difference between two strings of characters. We let intention be the source string and execution be the target. The edit cost for transforming the first string to the second is 1 for a character insertion or a deletion and 2 for a replacement (a deletion plus an insertion). We want to determine a minimum cost difference between these two strings.

Our array for subgoals will again be one character longer than each of the strings, in this case 10 by 10. The initialization is as in Figure 4.8 (a sequence of insertions is necessary, starting at the null string, to make either string resemble the other. The array location (2, 2) is 2, because a replacement (or making a delete plus an insert) is required to turn an i into an e.

	—	e	x	e	c	u	t	i	o	n
—	0	1	2	3	4	5	6	7	8	9
i	1	2								
n	2									
t	3									
e	4									
n	5									
t	6									
i	7									
o	8									
n	9									

Figure 4.8 Initialization of minimum edit difference matrix between *intention* and *execution* (adapted from Jurafsky and Martin 2000).

Figure 4.9 gives the full result of applying the dynamic programming algorithm to transform intention into execution. The value at each location (x, y) in the array is the cost of the minimum editing to that point plus the (minimum) cost of either an insertion, deletion, or replacement. Thus the cost of (x, y) is the minimum of the cost of (x - 1, y) plus cost of insertion, or cost (x - 1, y - 1) plus cost of replacement, or cost of (x, y - 1) plus the deletion cost. Pseudocode for his algorithm is a function taking the two strings (a source and a target) and their lengths and returning the minimum edit difference cost:

```
function dynamic (source, sl, target, tl) return cost (i, j);

create array cost(sl + 1, tl + 1)
cost (0,0) := 0                                                    % initialize
for i := 1 to sl + 1 do
    for j := 1 to tl + 1 do
        cost (i, j) := min [ cost (i - 1, j) + insertion cost $target_{i - 1}$          % add 1
                             cost (i - 1, j - 1) + replace cost $source_{j - 1}$ $target_{i - 1}$  % add 2
                             cost(i, j - 1) + delete cost $source_{j - 1}$ ]             % add 1
```

Using the results (bold) in Figure 4.9, the following edits will translate intention into execution with total edit cost of 8:

intention	
ntention	delete i, cost 1
etention	replace n with e, cost 2
exention	replace t with x, cost 2
exenution	insert u, cost 1
execution	replace n with c, cost 2

	—	e	x	e	c	u	t	i	o	n
—	0	1	2	3	4	5	6	7	8	9
i	1	**2**	3	4	5	6	7	8	9	10
n	2	3	**4**	5	6	7	8	9	10	11
t	3	4	**5**	6	7	8	9	10	11	12
e	4	5	6	**5**	**6**	7	8	9	10	11
n	5	6	7	6	7	**8**	9	10	11	12
t	6	7	8	7	8	9	**8**	9	10	11
i	7	8	9	8	9	10	9	**8**	9	10
o	8	9	10	9	10	11	10	9	**8**	9
n	9	10	11	10	11	12	11	10	9	**8**

Figure 4.9 Complete array of minimum edit difference between intention and execution (adapted from Jurafsky and Martin 2000).

In the spell check situation of proposing a cost-based ordered list of words for replacing an unrecognized string, the backward segment of the dynamic programming algorithm is not needed. Once the minimum edit measure is calculated for the set of related strings a prioritized order of alternatives is proposed from which the user chooses an appropriate string.

The justification for dynamic programming is the cost of time/space in computation. Dynamic programming, as seen in our examples has cost of n^2, where n is the length of the largest string; the cost in the worse case is n^3, if other related subproblems need to be considered (other row/column values) to determine the current state. Exhaustive search for comparing two strings is exponential, costing between 2^n and 3^n.

There are a number of obvious heuristics that can be used to prune the search in dynamic programming. First, useful solutions will usually lie around the upper left to lower right diagonal of the array; this leads to ignoring development of array extremes. Second, it can be useful to prune the search as it evolves, e.g., for edit distances passing a certain threshold, cut that solution path or even abandon the whole problem, i.e., the source string will be so distant from the target string of characters as to be useless. There is also a stochastic approach to the pattern comparison problem that we will see in Section 5.3.

4.2 The Best-First Search Algorithm

4.2.1 Implementing Best-First Search

In spite of their limitations, algorithms such as backtrack, hill climbing, and dynamic programming can be used effectively if their evaluation functions are sufficiently informative to avoid local maxima, dead ends, and related anomalies in a search space. In general, however, use of heuristic search requires a more flexible algorithm: this is provided by *best-first search*, where, with a priority queue, recovery from these situations is possible.

Like the depth-first and breadth-first search algorithms of Chapter 3, best-first search uses lists to maintain states: open to keep track of the current fringe of the search and closed to record states already visited. An added step in the algorithm orders the states on open according to some heuristic estimate of their "closeness" to a goal. Thus, each iteration of the loop considers the most "promising" state on the open list. The pseudocode for the function best_first_search appears below.

```
function best_first_search;

begin
  open := [Start];                                                        % initialize
  closed := [ ];
  while open ≠ [ ] do                                                     % states remain
    begin
      remove the leftmost state from open, call it X;
      if X = goal then return the path from Start to X
      else begin
              generate children of X;
              for each child of X do
              case
                  the child is not on open or closed:
                    begin
                        assign the child a heuristic value;
                        add the child to open
                    end;
                  the child is already on open:
                    if the child was reached by a shorter path
                    then give the state on open the shorter path
                  the child is already on closed:
                    if the child was reached by a shorter path then
                      begin
                        remove the state from closed;
                        add the child to open
                      end;
              end;                                                        % case
              put X on closed;
              re-order states on open by heuristic merit (best leftmost)
            end;
  return FAIL                                                             % open is empty
end.
```

At each iteration, best_first_search removes the first element from the open list. If it meets the goal conditions, the algorithm returns the solution path that led to the goal. Note that each state retains ancestor information to determine if it had previously been reached by a shorter path and to allow the algorithm to return the final solution path. (See Section 3.2.3.)

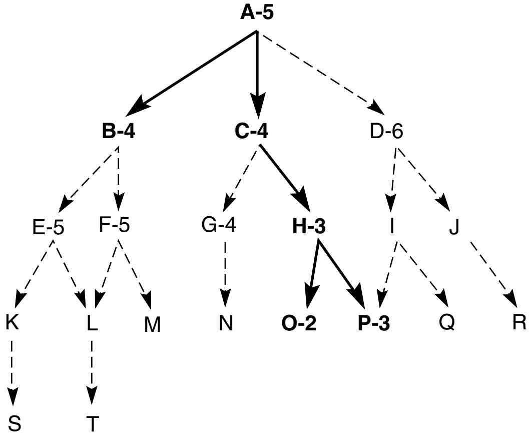


Figure 4.10 Heuristic search of a hypothetical state space.

If the first element on open is not a goal, the algorithm applies all matching production rules or operators to generate its descendants. If a child state is not on open or closed best_first_search applies a heuristic evaluation to that state, and the open list is sorted according to the heuristic values of those states. This brings the "best" states to the front of open. Note that because these estimates are heuristic in nature, the next "best" state to be examined may be from any level of the state space. When open is maintained as a sorted list, it is often referred to as a *priority queue*.

If a child state is already on open or closed, the algorithm checks to make sure that the state records the shorter of the two partial solution paths. Duplicate states are not retained. By updating the path history of nodes on open and closed when they are rediscovered, the algorithm will find a shortest path to a goal (within the states considered).

Figure 4.10 shows a hypothetical state space with heuristic evaluations attached to some of its states. The states with attached evaluations are those actually generated in best_first_search. The states expanded by the heuristic search algorithm are indicated in bold; note that it does not search all of the space. The goal of best-first search is to find the goal state by looking at as few states as possible; the more *informed* (Section 4.2.3) the heuristic, the fewer states are processed in finding the goal.

A trace of the execution of best_first_search on this graph appears below. Suppose P is the goal state in the graph of Figure 4.10. Because P is the goal, states along the path to P will tend to have lower heuristic values. The heuristic is fallible: the state O has a lower value than the goal itself and is examined first. Unlike hill climbing, which does not maintain a priority queue for the selection of "next" states, the algorithm recovers from this error and finds the correct goal.

1. open = [A5]; closed = [ ]
2. evaluate A5; open = [B4,C4,D6]; closed = [A5]

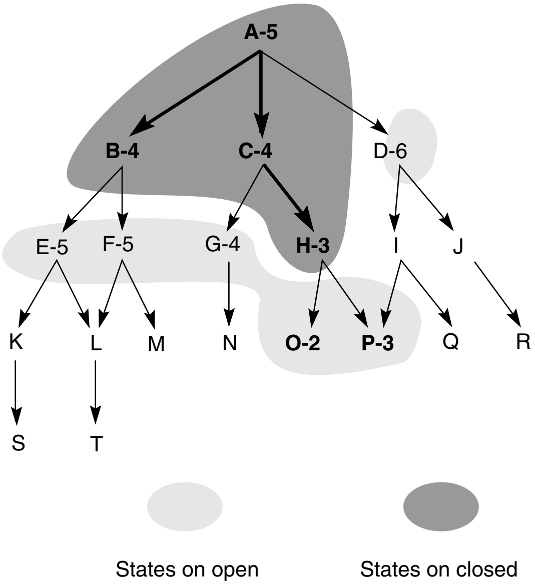


Figure 4.11 Heuristic search of a hypothetical state space with open and closed states highlighted.

3. evaluate B4; open = [C4,E5,F5,D6]; closed = [B4,A5]
4. evaluate C4; open = [H3,G4,E5,F5,D6]; closed = [C4,B4,A5]
5. evaluate H3; open = [O2,P3,G4,E5,F5,D6]; closed = [H3,C4,B4,A5]
6. evaluate O2; open = [P3,G4,E5,F5,D6]; closed = [O2,H3,C4,B4,A5]
7. evaluate P3; the solution is found!

Figure 4.11 shows the space as it appears after the fifth iteration of the while loop. The states contained in open and closed are indicated. open records the current frontier of the search and closed records states already considered. Note that the frontier of the search is highly uneven, reflecting the opportunistic nature of best-first search.

The best-first search algorithm selects the most promising state on open for further expansion. However, as it is using a heuristic that may prove erroneous, it does not abandon all the other states but maintains them on open. In the event a heuristic leads the search down a path that proves incorrect, the algorithm will eventually retrieve some previously generated, "next best" state from open and shift its focus to another part of the space. In the example of Figure 4.10, after the children of state B were found to have poor heuristic evaluations, the search shifted its focus to state C. The children of B were kept on open in case the algorithm needed to return to them later. In best_first_search, as in the algorithms of Chapter 3, the open list supports backtracking from paths that fail to produce a goal.

4.2.2 Implementing Heuristic Evaluation Functions

We next evaluate the performance of several different heuristics for solving the 8-puzzle. Figure 4.12 shows a start and goal state for the 8-puzzle, along with the first three states generated in the search.

The simplest heuristic counts the tiles out of place in each state when compared with the goal. This is intuitively appealing, because it would seem that, all else being equal, the state that had fewest tiles out of place is probably closer to the desired goal and would be the best to examine next.

However, this heuristic does not use all of the information available in a board configuration, because it does not take into account the distance the tiles must be moved. A "better" heuristic would sum all the distances by which the tiles are out of place, one for each square a tile must be moved to reach its position in the goal state.

Both of these heuristics can be criticized for failing to acknowledge the difficulty of tile reversals. That is, if two tiles are next to each other and the goal requires their being in opposite locations, it takes (several) more than two moves to put them back in place, as the tiles must "go around" each other (Figure 4.13).

A heuristic that takes this into account multiplies a small number (2, for example) times each direct tile reversal (where two adjacent tiles must be exchanged to be in the order of the goal). Figure 4.14 shows the result of applying each of these three heuristics to the three child states of Figure 4.12.

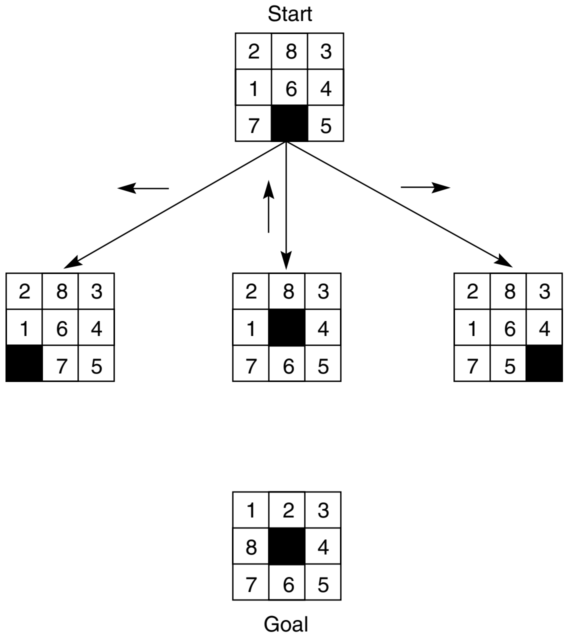


Figure 4.12 The start state, first moves, and goal state for an example 8-puzzle.

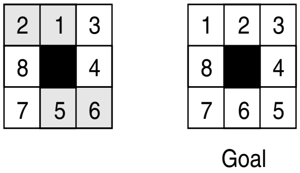
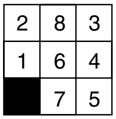
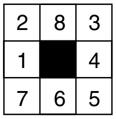
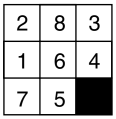


Figure 4.13 An 8-puzzle state with a goal and two reversals: 1 and 2, 5 and 6.

In Figure 4.14's summary of evaluation functions, the sum of distances heuristic does indeed seem to provide a more accurate estimate of the work to be done than the simple count of the number of tiles out of place. Also, the tile reversal heuristic fails to distinguish between these states, giving each an evaluation of 0. Although it is an intuitively appealing heuristic, it breaks down since none of these states have any direct reversals. A fourth heuristic, which may overcome the limitations of the tile reversal heuristic, adds the sum of the distances the tiles are out of place and 2 times the number of direct reversals.

This example illustrates the difficulty of devising good heuristics. Our goal is to use the limited information available in a single state descriptor to make intelligent choices. Each of the heuristics proposed above ignores some critical bit of information and is subject to improvement. The design of good heuristics is an empirical problem; judgment and intuition help, but the final measure of a heuristic must be its actual performance on problem instances.

If two states have the same or nearly the same heuristic evaluations, it is generally preferable to examine the state that is nearest to the root state of the graph. This state will have a greater probability of being on the *shortest* path to the goal. The distance from the

	Tiles out of place	Sum of distances out of place	2 x the number of direct tile reversals
	**5**	**6**	**0**
	**3**	**4**	**0**
	**5**	**6**	**0**

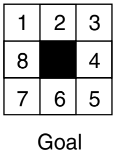


Figure 4.14 Three heuristics applied to states in the 8-puzzle.

starting state to its descendants can be measured by maintaining a depth count for each state. This count is 0 for the beginning state and is incremented by 1 for each level of the search. This depth measure can be added to the heuristic evaluation of each state to bias search in favor of states found shallower in the graph.

This makes our evaluation function, f, the sum of two components:

$$f(n) = g(n) + h(n)$$

where $g(n)$ measures the actual length of the path from any state n to the start state and $h(n)$ is a heuristic estimate of the distance from state n to a goal.

In the 8-puzzle, for example, we can let $h(n)$ be the number of tiles out of place. When this evaluation is applied to each of the child states in Figure 4.12, their f values are 6, 4, and 6, respectively, see Figure 4.15.

The full best-first search of the 8-puzzle graph, using f as defined above, appears in Figure 4.16. Each state is labeled with a letter and its heuristic weight, $f(n) = g(n) + h(n)$. The number at the top of each state indicates the order in which it was taken off the open list. Some states (h, g, b, d, n, k, and i) are not so numbered, because they were still on open when the algorithm terminates.

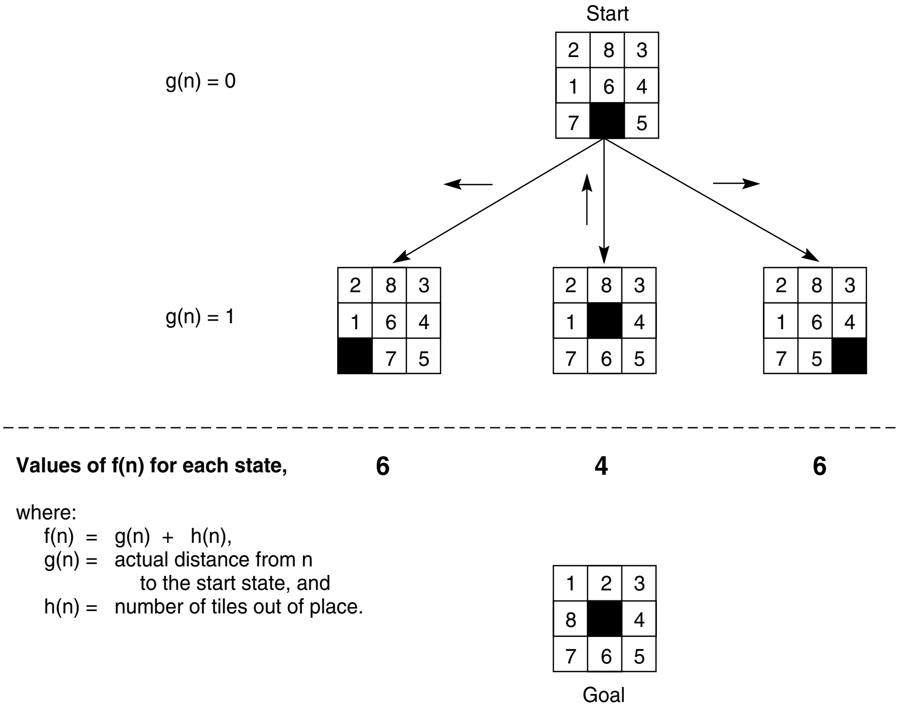


Figure 4.15 The heuristic f applied to states in the 8-puzzle.

The successive stages of open and closed that generate this graph are:

1. open = [a4];
   closed = [ ]
2. open = [c4, b6, d6];
   closed = [a4]
3. open = [e5, f5, b6, d6, g6];
   closed = [a4, c4]
4. open = [f5, h6, b6, d6, g6, i7];
   closed = [a4, c4, e5]
5. open = [j5, h6, b6, d6, g6, k7, i7];
   closed = [a4, c4, e5, f5]
6. open = [l5, h6, b6, d6, g6, k7, i7];
   closed = [a4, c4, e5, f5, j5]
7. open = [m5, h6, b6, d6, g6, n7, k7, i7];
   closed = [a4, c4, e5, f5, j5, l5]
8. success, m = goal!

In step 3, both e and f have a heuristic of 5. State e is examined first, producing children, h and i. Although h, the child of e, has the same number of tiles out of place as f, it is one level deeper in the space. The depth measure, g(n), causes the algorithm to select f for evaluation in step 4. The algorithm goes back to the shallower state and continues to the goal. The state space graph at this stage of the search, with open and closed highlighted, appears in Figure 4.17. Notice the opportunistic nature of best-first search.

In effect, the g(n) component of the evaluation function gives the search more of a breadth-first flavor. This prevents it from being misled by an erroneous evaluation: if a heuristic continuously returns "good" evaluations for states along a path that fails to reach a goal, the g value will grow to dominate h and force search back to a shorter solution path. This guarantees that the algorithm will not become permanently lost, descending an infinite branch. Section 4.3 examines the conditions under which best-first search using this evaluation function can actually be guaranteed to produce the shortest path to a goal.

To summarize:

1. Operations on states generate children of the state currently under examination.
2. Each new state is checked to see whether it has occurred before (is on either open or closed), thereby preventing loops.
3. Each state n is given an f value equal to the sum of its depth in the search space g(n) and a heuristic estimate of its distance to a goal h(n). The h value guides search toward heuristically promising states while the g value can prevent search from persisting indefinitely on a fruitless path.

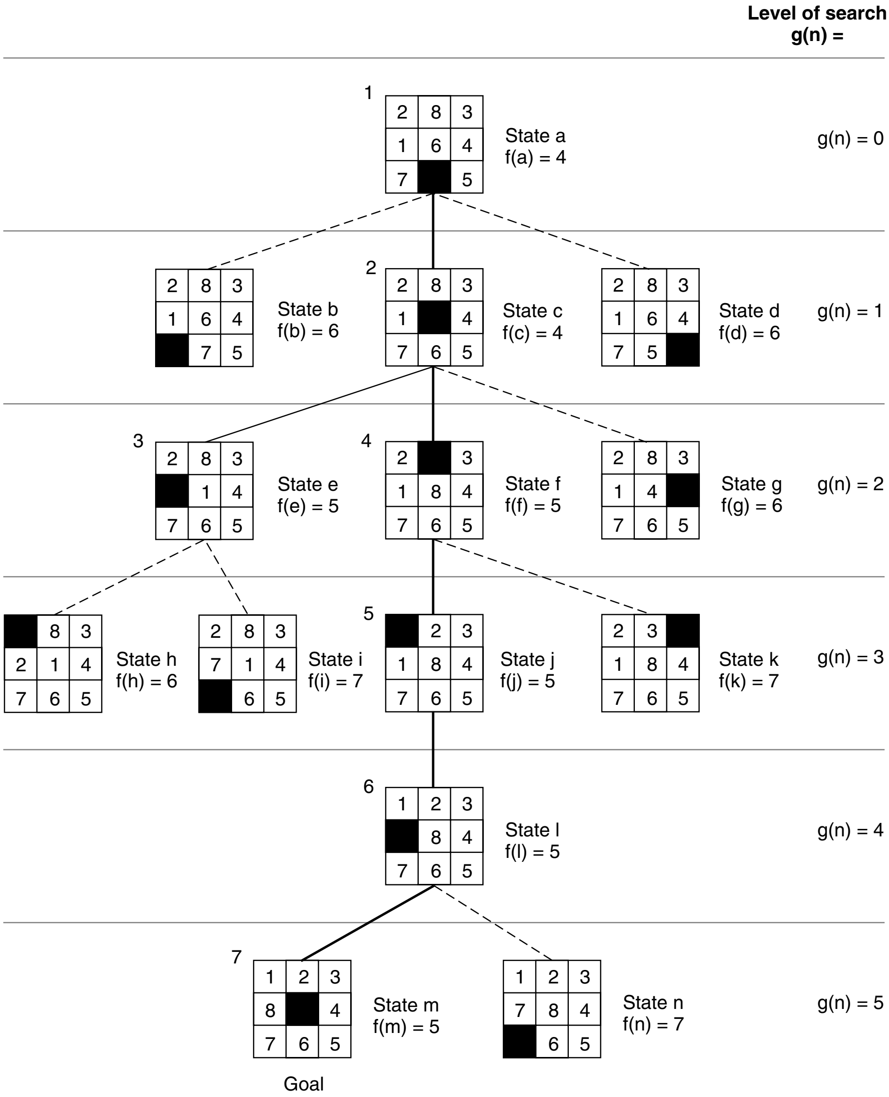


Figure 4.16 State space generated in heuristic search of the 8-puzzle graph.

4. States on open are sorted by their f values. By keeping all states on open until they are examined or a goal is found, the algorithm recovers from dead ends.

5. As an implementation point, the algorithm's efficiency can be improved through maintenance of the open and closed lists, perhaps as *heaps* or *leftist trees.*

Best-first search is a general algorithm for heuristically searching any state space graph (as were the breadth- and depth-first algorithms presented earlier). It is equally

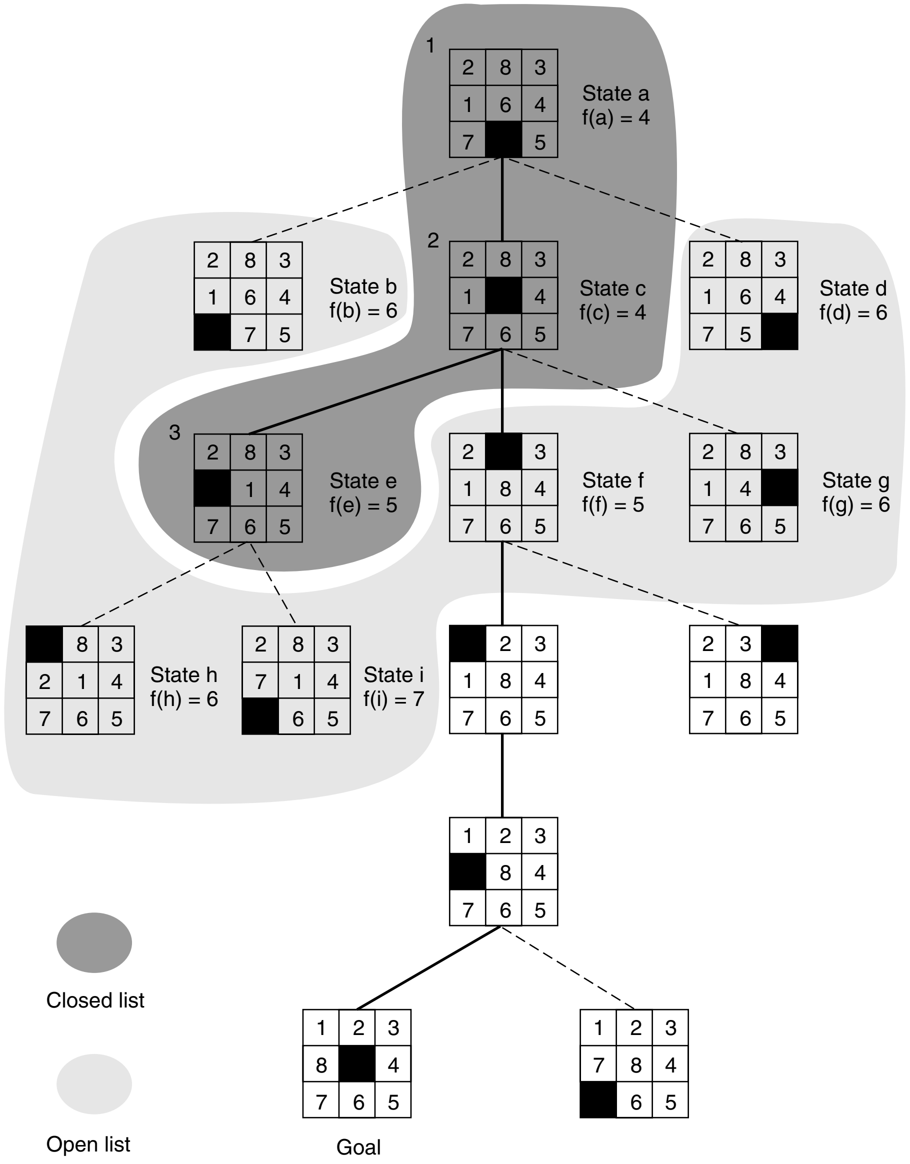


Figure 4.17 open and closed as they appear after the third iteration of heuristic search.

applicable to data- and goal-driven searches and supports a variety of heuristic evaluation functions. It will continue (Section 4.3) to provide a basis for examining the behavior of heuristic search. Because of its generality, best-first search can be used with a variety of heuristics, ranging from subjective estimates of state's "goodness" to sophisticated measures based on the probability of a state leading to a goal. Bayesian statistical measures (Chapters 5 and 9) offer an important example of this approach.

Another interesting approach to implementing heuristics is the use of confidence measures by expert systems to weigh the results of a rule. When human experts employ a heuristic, they are usually able to give some estimate of their confidence in its conclusions. Expert systems employ *confidence measures* to select the conclusions with the highest likelihood of success. States with extremely low confidences can be eliminated entirely. This approach to heuristic search is examined in the next section.

4.2.3 Heuristic Search and Expert Systems

Simple games such as the 8-puzzle are ideal vehicles for exploring the design and behavior of heuristic search algorithms for a number of reasons:

1. The search spaces are large enough to require heuristic pruning.
2. Most games are complex enough to suggest a rich variety of heuristic evaluations for comparison and analysis.
3. Games generally do not involve complex representational issues. A single node of the state space is just a board description and usually can be captured in a straightforward fashion. This allows researchers to focus on the behavior of the heuristic rather than the problems of knowledge representation.
4. Because each node of the state space has a common representation (e.g., a board description), a single heuristic may be applied throughout the search space. This contrasts with systems such as the financial advisor, where each node represents a different subgoal with its own distinct description.

More realistic problems greatly complicate the implementation and analysis of heuristic search by requiring multiple heuristics to deal with different situations in the problem space. However, the insights gained from simple games generalize to problems such as those found in expert systems applications, planning, intelligent control, and machine learning. Unlike the 8-puzzle, a single heuristic may not apply to each state in these domains. Instead, situation specific problem-solving heuristics are encoded in the syntax and content of individual problem solving operators. Each solution step incorporates its own heuristic that determines when it should be applied; the pattern matcher matches the appropriate operation (heuristic) with the relevant state in the space.

EXAMPLE 4.2.1: THE FINANCIAL ADVISOR, REVISITED

The use of heuristic measures to guide search is a general approach in AI. Consider again the financial advisor problem of Chapters 2 and 3, where the knowledge base was treated as a set of logical implications, whose conclusions are either true or false. In actuality, these rules are highly heuristic in nature. For example, one rule states that an individual with adequate savings and income should invest in stocks:

savings_account(adequate) $\wedge$ income(adequate) $\Rightarrow$ investment(stocks).

In reality, it is possible that such an individual may prefer the added security of a combination strategy or even that of placing all investment money in savings. Thus, the rule is a heuristic, and the problem solver should try to account for this uncertainty. We could take additional factors, such as the age of the investor and the long-term prospects for security and advancement in the investor's profession, into account to make the rules more informed and capable of finer distinctions. However, this does not change the fundamentally heuristic nature of financial advice.

One way in which expert systems have addressed this issue is to attach a numeric weight (called a *confidence measure* or *certainty factor*) to the conclusion of each rule. This measures the confidence that may be placed in their conclusions.

Each rule conclusion is given a confidence measure, say a real number between -1 and 1, with 1 corresponding to certainty (true) and -1 to a definite value of false. Values in between reflect varying confidence in the conclusion. For example, the preceding rule may be given a confidence of, say, 0.8, reflecting a small possibility that it may not be correct. Other conclusions may be drawn with different confidence weights:

savings_account(adequate) $\wedge$ income(adequate) $\Rightarrow$ investment(stocks)
with confidence = 0.8.

savings_account(adequate) $\wedge$ income(adequate) $\Rightarrow$ investment(combination)
with confidence = 0.5.

savings_account(adequate) $\wedge$ income(adequate) $\Rightarrow$ investment(savings)
with confidence = 0.1.

These rules reflect the common investment advice that although an individual with adequate savings and income would be most strongly advised to invest in stocks, there is some possibility that a combination strategy should be pursued and a slight chance that they may want to continue investing in savings. Heuristic search algorithms can use these certainty factors in a number of ways. For example, the results of all applicable rules could be produced along with their associated confidences. This exhaustive search of all possibilities may be appropriate in domains such as medicine. Alternatively, the program might return only the result with the strongest confidence value, if alternative solutions are not of interest. This can allow the program to ignore other rules, radically pruning the search space. A more conservative pruning strategy could ignore rules that draw a conclusion with a confidence less than a certain value, 0.2 for example.

A number of important issues must be addressed in using confidence measures to weight rule conclusions. What does it really mean to have a "numeric confidence measure"? For example, how are the confidences handled if the conclusion of one rule is used as the premise of others? How are confidences combined in the event that more than one rule draws the same conclusion? How are the proper confidence measures assigned to rules in the first place? These issues are discussed in more detail in Chapter 8.

4.3 Admissibility, Monotonicity, and Informedness

We may evaluate the behavior of heuristics along a number of dimensions. For instance, we may desire a solution and also require the algorithm to find the shortest path to the goal. This could be important when an application might have an excessive cost for extra solution steps, such as planning a path for an autonomous robot through a dangerous environment. Heuristics that find the shortest path to a goal whenever it exists are said to be *admissible*. In other applications a minimal solution path might not be as important as overall problem-solving efficiency (see Figure 4.28).

We may want to ask whether any better heuristics are available. In what sense is one heuristic "better" than another? This is the *informedness* of a heuristic.

When a state is discovered by using heuristic search, is there any guarantee that the same state won't be found later in the search at a cheaper cost (with a shorter path from the start state)? This is the property of *monotonicity.* The answers to these and other questions related to the effectiveness of heuristics make up the content of this section.

4.3.1 Admissibility Measures

A search algorithm is *admissible* if it is guaranteed to find a minimal path to a solution whenever such a path exists. Breadth-first search is an admissible search strategy. Because it looks at every state at level n of the graph before considering any state at the level $n + 1$, any goal nodes are found along the shortest possible path. Unfortunately, breadth-first search is often too inefficient for practical use.

Using the evaluation function $f(n) = g(n) + h(n)$ that was introduced in the last section, we may characterize a class of admissible heuristic search strategies. If n is a node in the state space graph, $g(n)$ measures the depth at which that state has been found in the graph, and $h(n)$ is the heuristic estimate of the distance from n to a goal. In this sense $f(n)$ estimates the total cost of the path from the start state through n to the goal state. In determining the properties of admissible heuristics, we define an evaluation function f^*:

$$f^*(n) = g^*(n) + h^*(n)$$

where $g^*(n)$ is the cost of the *shortest* path from the start to node n and h^* returns the *actual* cost of the shortest path from n to the goal. It follows that $f^*(n)$ is the actual cost of the optimal path from a start node to a goal node that passes through node n.

As we will see, when we employ best_first_search with the evaluation function f^*, the resulting search strategy is admissible. Although *oracles* such as f^* do not exist for most real problems, we would like the evaluation function f to be a close estimate of f^*. In algorithm A, $g(n)$, the cost of the current path to state n, is a reasonable estimate of g^*, but they may not be equal: $g(n) \geq g^*(n)$. These are equal only if the graph search has discovered the optimal path to state n.

Similarly, we replace $h^*(n)$ with $h(n)$, a heuristic estimate of the minimal cost to a goal state. Although we usually may not compute h^*, it is often possible to determine whether or not the heuristic estimate, $h(n)$, is bounded from above by $h^*(n)$, i.e., is always less than or equal to the actual cost of a minimal path. If algorithm A uses an evaluation function f in which $h(n) \leq h^*(n)$, it is called *algorithm A**.

DEFINITION

ALGORITHM A, ADMISSIBILITY, ALGORITHM A*

Consider the evaluation function $f(n) = g(n) + h(n)$, where

n is any state encountered in the search.

$g(n)$ is the cost of n from the start state.

$h(n)$ is the heuristic estimate of the cost of going from n to a goal.

If this evaluation function is used with the best_first_search algorithm of Section 4.1, the result is called *algorithm A*.

A search algorithm is *admissible* if, for any graph, it always terminates in the optimal solution path whenever a path from the start to a goal state exists.

If algorithm A is used with an evaluation function in which $h(n)$ is less than or equal to the cost of the minimal path from n to the goal, the resulting search algorithm is called *algorithm* A* (pronounced "A STAR").

It is now possible to state a property of A* algorithms:

All A* algorithms are admissible.

The admissibility of A* algorithms is a theorem. An exercise at the end of the chapter gives directions for developing its proof (see also Nilsson 1980, p 76-78). The theorem says that any A* algorithm, i.e., one that uses a heuristic $h(n)$ such that $h(n) \leq h^*(n)$ for all n, is guaranteed to find the minimal path from n to the goal, if such a path exists.

Note that breadth-first search may be characterized as an A* algorithm in which $f(n) = g(n) + 0$. The decision for considering a state is based solely on its distance from the start state. We will show (Section 4.3.3) that the set of nodes considered by an A* algorithm is a subset of the states examined in breadth-first search.

Several heuristics from the 8-puzzle provide examples of A* algorithms. Although we may not be able to compute the value of $h^*(n)$ for the 8-puzzle, we may determine when a heuristic is bounded from above by the actual cost of the shortest path to a goal.

For instance, the heuristic of counting the number of tiles not in the goal position is certainly less than or equal to the number of moves required to move them to their goal position. Thus, this heuristic is admissible and guarantees an optimal (or shortest) solution path when that path exists. The sum of the direct distances of tiles out of place is also less than or equal to the minimum actual path. Using small multipliers for direct tile reversals gives an admissible heuristic.

This approach to proving admissibility of 8-puzzle heuristics may be applied to any heuristic search problem. Even though the actual cost of the shortest path to a goal may not always be computed, we can often prove that a heuristic is bounded from above by this value. When this can be done, the resulting search will terminate in the discovery of the shortest path to the goal, when such a path exists.

4.3.2 Monotonicity

Recall that the definition of A* algorithms did not require that $g(n) = g^*(n)$. This means that admissible heuristics may initially reach non-goal states along a suboptimal path, as long as the algorithm eventually finds an optimal path to all states on the path to a goal. It is natural to ask if there are heuristics that are "locally admissible," i.e., that consistently find the minimal path to each state they encounter in the search. This property is called *monotonicity.*

DEFINITION

MONOTONICITY

A heuristic function h is monotone if

1. For all states n_i and n_j, where n_j is a descendant of n_i,

   $$h(n_i) - h(n_j) \leq cost(n_i,n_j),$$

   where $cost(n_i,n_j)$ is the actual cost (in number of moves) of going from state n_i to n_j.

2. The heuristic evaluation of the goal state is zero, or $h(Goal) = 0$.

One way of describing the monotone property is that the search space is everywhere locally consistent with the heuristic employed. The difference between the heuristic measure for a state and any one of its successors is bound by the actual cost of going between that state and its successor. This is to say that the heuristic is everywhere admissible, reaching each state along the shortest path from its ancestors.

If the graph search algorithm for best-first search is used with a monotonic heuristic, an important step may be omitted. Because the heuristic finds the shortest path to any state the first time that state is discovered, when a state is encountered a second time, it is not necessary to check whether the new path is shorter. It won't be! This allows any state that is rediscovered in the space to be dropped immediately without updating the path information retained on open or closed.

When using a monotonic heuristic, as the search moves through the space, the heuristic measure for each state n is replaced by the actual cost for generating that piece of the path to n. Because the actual cost is equal to or larger than the heuristic in each instance, f will not decrease; i.e., f is monotonically nondecreasing (hence the name).

A simple argument can show that any monotonic heuristic is admissible. This argument considers any path in the space as a sequence of states $s_1, s_2, \ldots, s_g$, where s_1 is the start state and s_g is the goal. For the sequence of moves in this arbitrarily selected path:

s_1 to s_2	$h(s_1) - h(s_2) \leq cost(s_1,s_2)$	by monotone property
s_2 to s_3	$h(s_2) - h(s_3) \leq cost(s_2,s_3)$	by monotone property
s_3 to s_4	$h(s_3) - h(s_4) \leq cost(s_3,s_4)$	by monotone property
.	. . .	by monotone property
.	. . .	by monotone property
s_{g-1} to s_g	$h(s_{g-1}) - h(s_g) \leq cost(s_{g-1},s_g)$	by monotone property

Summing each column and using the monotone property of $h(s_g) = 0$:

path s_1 to s_g $\quad h(s_1) \leq cost(s_1,s_g)$

This means that monotone heuristic h is A^* and admissible. It is left as an exercise whether or not the admissibility property of a heuristic implies monotonicity.

4.3.3 When One Heuristic Is Better: More Informed Heuristics

The final issue of this subsection compares two heuristics' ability to find the minimal path. An interesting case occurs when the heuristics are A^*.

DEFINITION

INFORMEDNESS

For two A^* heuristics h_1 and h_2, if $h_1(n) \leq h_2(n)$, for all states n in the search space, heuristic h_2 is said to be *more informed* than h_1.

We can use this definition to compare the heuristics proposed for solving the 8-puzzle. As pointed out previously, breadth-first search is equivalent to the A^* algorithm with heuristic h_1 such that $h_1(x) = 0$ for all states x. This is, trivially, less than h^*. We have also shown that h_2, the number of tiles out of place with respect to the goal state, is a lower bound for h^*. In this case $h_1 \leq h_2 \leq h^*$. It follows that the "number of tiles out of place" heuristic is more informed than breadth-first search. Figure 4.18 compares the spaces searched by these two heuristics. Both h_1 and h_2 find the optimal path, but h_2 evaluates many fewer states in the process.

Similarly, we can argue that the heuristic that calculates the sum of the direct distances by which all the tiles are out of place is again more informed than the calculation of the number of tiles that are out of place with respect to the goal state, and indeed this is the case. One can visualize a sequence of search spaces, each smaller than the previous one, converging on the direct optimal path solution.

If a heuristic h_2 is more informed than h_1, then the set of states examined by h_2 is a subset of those expanded by h_1. This can be verified by assuming the opposite (that there is at least one state expanded by h_2 and not by h_1). But since h_2 is more informed than h_1, for all n, $h_2(n)$ † $h_1(n)$, and both are bounded above by h^*, our assumption is contradictory.

In general, then, the more informed an A* algorithm, the less of the space it needs to expand to get the optimal solution. We must be careful, however, that the computations

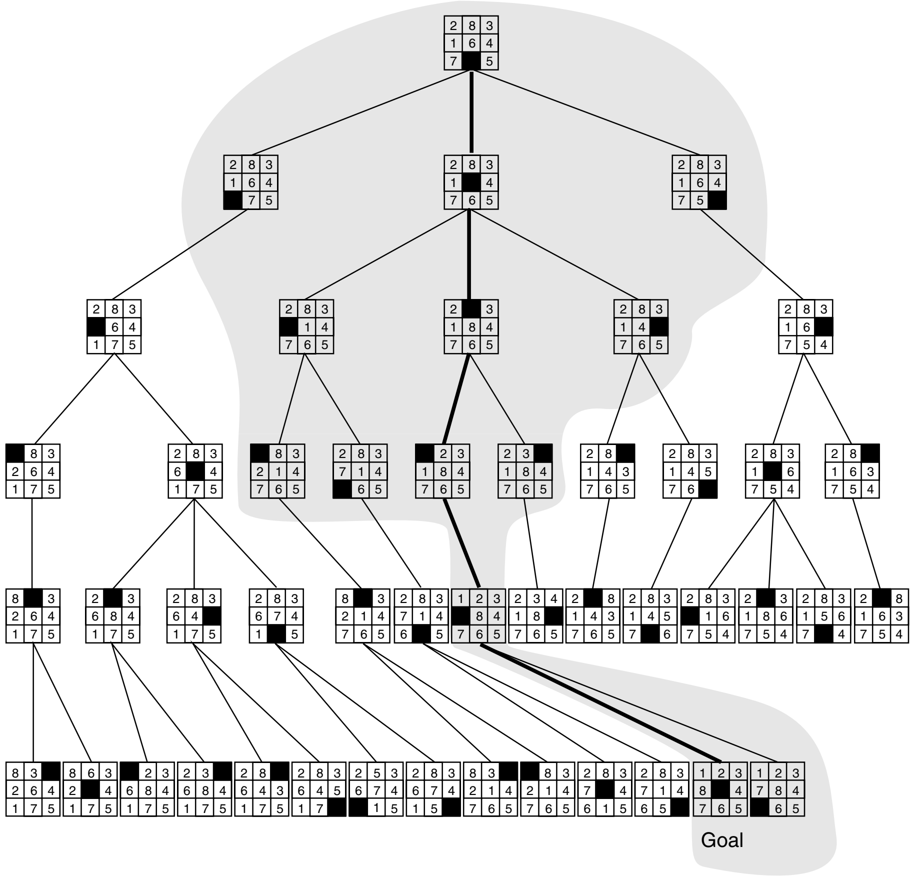


Figure 4.18 Comparison of state space searched using heuristic search with space searched by breadth-first search. The portion of the graph searched heuristically is shaded. The optimal solution path is in bold. Heuristic used is f(n) = g(n) + h(n) where h(n) is tiles out of place.

necessary to employ the more informed heuristic are not so inefficient as to offset the gains from reducing the number of states searched.

Computer chess programs provide an interesting example of this trade-off. One school of thought uses simple heuristics and relies on computer speed to search deeply into the search space. These programs often use specialized hardware for state evaluation to increase the depth of the search. Another school relies on more sophisticated heuristics to reduce the number of board states searched. These heuristics include calculations of piece advantages, control of board geography, possible attack strategies, passed pawns, and so on. Calculation of heuristics itself may involve exponential complexity (an issue discussed in Section 4.5). Since the total time for the first 40 moves of the game is limited it is important to optimize this trade-off between search and heuristic evaluation. The optimal blend of blind search and heuristics remains an open empirical question in computer chess, as can be seen in the Gary Kasparov Deep Blue battle (Hsu 2002).

4.4 Using Heuristics in Games

At that time two opposing concepts of the game called forth commentary and discussion. The foremost players distinguished two principal types of Game, the formal and the psychological . . .

—Hermann Hesse, “*Magister Ludi*” (The Glass Bead Game)

4.4.1 The Minimax Procedure on Exhaustively Searchable Graphs

Games have always been an important application area for heuristic algorithms. Two-person games are more complicated than simple puzzles because of the existence of a “hostile” and essentially unpredictable opponent. Thus, they provide some interesting opportunities for developing heuristics, as well as greater difficulties in developing search algorithms.

First we consider games whose state space is small enough to be exhaustively searched; here the problem is systematically searching the space of possible moves and countermoves by the opponent. Then we look at games in which it is either impossible or undesirable to exhaustively search the game graph. Because only a portion of the state space can be generated and searched, the game player must use heuristics to guide play along a path to a winning state.

We first consider a variant of the game *nim*, whose state space may be exhaustively searched. To play this game, a number of tokens are placed on a table between the two opponents; at each move, the player must divide a pile of tokens into two nonempty piles of different sizes. Thus, 6 tokens may be divided into piles of 5 and 1 or 4 and 2, but not 3 and 3. The first player who can no longer make a move loses the game. For a reasonable number of tokens, the state space can be exhaustively searched. Figure 4.19 illustrates the space for a game with 7 tokens.

In playing games whose state space may be exhaustively delineated, the primary difficulty is in accounting for the actions of the opponent. A simple way to handle this

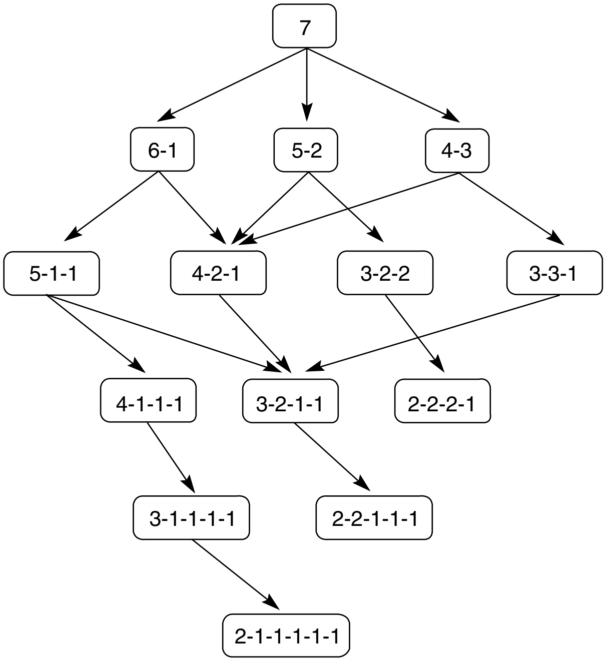


Figure 4.19 State space for a variant of nim. Each state partitions the seven matches into one or more piles.

assumes that your opponent uses the same knowledge of the state space as you use and applies that knowledge in a consistent effort to win the game. Although this assumption has its limitations (which are discussed in Section 4.4.2), it provides a reasonable basis for predicting an opponent's behavior. *Minimax* searches the game space under this assumption.

The opponents in a game are referred to as MIN and MAX. Although this is partly for historical reasons, the significance of these names is straightforward: MAX represents the player trying to win, or to MAXimize her advantage. MIN is the opponent who attempts to MINimize MAX's score. We assume that MIN uses the same information and always attempts to move to a state that is worst for MAX.

In implementing minimax, we label each level in the search space according to whose move it is at that point in the game, MIN or MAX. In the example of Figure 4.20, MIN is allowed to move first. Each leaf node is given a value of 1 or 0, depending on whether it is a win for MAX or for MIN. Minimax propagates these values up the graph through successive parent nodes according to the rule:

If the parent state is a MAX node, give it the maximum value among its children.

If the parent is a MIN node, give it the minimum value of its children.

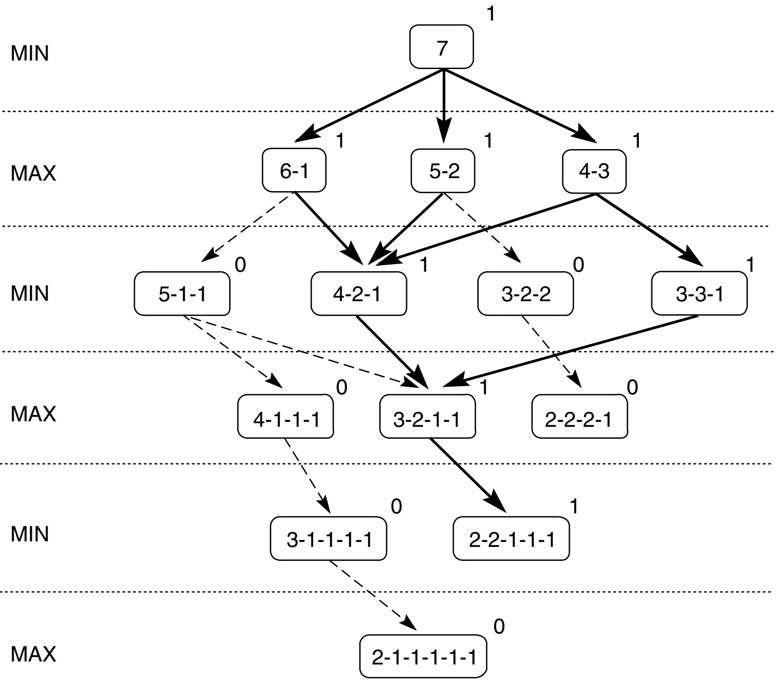


Figure 4.20 Exhaustive minimax for the game of nim. Bold lines indicate forced win for MAX. Each node is marked with its derived value (0 or 1) under minimax.

The value that is thus assigned to each state indicates the value of the best state that this player can hope to achieve (assuming the opponent plays as predicted by the minimax algorithm). These derived values are used to choose among possible moves. The result of applying minimax to the state space graph for nim appears in Figure 4.20.

The values of the leaf nodes are propagated up the graph using minimax. Because all of MIN's possible first moves lead to nodes with a derived value of 1, the second player, MAX, always can force the game to a win, regardless of MIN's first move. MIN could win only if MAX played foolishly. In Figure 4.20, MIN may choose any of the first move alternatives, with the resulting win paths for MAX in bold arrows.

Although there are games where it is possible to search the state space exhaustively, most interesting games do not. We examine fixed depth search next.

4.4.2 Minimaxing to Fixed Ply Depth

In applying minimax to more complicated games, it is seldom possible to expand the state space graph out to the leaf nodes. Instead, the state space is searched to a predefined number of levels, as determined by available resources of time and memory. This strategy is called an n-*ply look-ahead*, where n is the number of levels explored. As the leaves of this subgraph are not final states of the game, it is not possible to give them values that reflect a win or a loss. Instead, each node is given a value according to some heuristic evaluation function. The value that is propagated back to the root node is not an indication of whether or not a win can be achieved (as in the previous example) but is simply the heuristic value of the best state that can be reached in n moves from the root. Look-ahead increases the power of a heuristic by allowing it to be applied over a greater area of the state space. Minimax consolidates these separate evaluations for the ancestor state.

In a game of conflict, each player attempts to overcome the other, so many game heuristics directly measure the advantage of one player over another. In checkers or chess, piece advantage is important, so a simple heuristic might take the difference in the number of pieces belonging to MAX and MIN and try to maximize the difference between these piece measures. A more sophisticated strategy might assign different values to the pieces, depending on their value (e.g., queen vs. pawn or king vs. ordinary checker) or location on the board. Most games provide limitless opportunities for designing heuristics.

Game graphs are searched by level, or *ply*. As we saw in Figure 4.20, MAX and MIN alternately select moves. Each move by a player defines a new ply of the graph. Game playing programs typically look ahead a fixed ply depth, determined by the space/time limitations of the computer. The states on that ply are measured heuristically and the values are propagated back up the graph using minimax. The search algorithm then uses these *derived values* to select among possible next moves.

After assigning an evaluation to each state on the selected ply, the program propagates a value up to each parent state. If the parent is on a MIN level, the minimum value of the children is backed up. If the parent is a MAX node, minimax assigns it the maximum value of its children.

Maximizing for MAX parents and minimizing for MIN, the values go back up the graph to the children of the current state. These values are then used by the current state to select among its children. Figure 4.21 shows minimax on a hypothetical state space with a four-ply look-ahead.

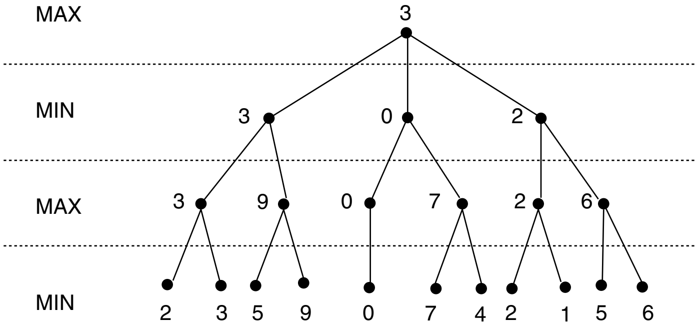


Figure 4.21 Minimax to a hypothetical state space. Leaf states show heuristic values; internal states show backed-up values.

We can make several final points about the minimax procedure. First, and most important, evaluations to any (previously decided) fixed ply depth may be seriously misleading. When a heuristic is applied with limited look-ahead, it is possible the depth of the look-ahead may not detect that a heuristically promising path leads to a bad situation later in the game. If your opponent in chess offers a rook as a lure to take your queen, and the evaluation only looks ahead to the ply where the rook is offered, the evaluation is going to be biased toward this state. Unfortunately, selection of the state may cause the entire game to be lost! This is referred to as the *horizon effect*. It is usually countered by searching several plies deeper from states that look exceptionally good. This selective deepening of search in important areas will not make the horizon effect go away, however. The search must stop somewhere and will be blind to states beyond that point.

There is another effect that occurs in minimaxing on the basis of heuristic evaluations. The evaluations that take place deep in the space can be biased by their very depth (Pearl 1984). In the same way that the average of products differs from the product of averages, the estimate of minimax (which is what we desire) is different from the minimax of estimates (which is what we are doing). In this sense, deeper search with evaluation and minimax normally does, but need not always, mean better search. Further discussion of these issues and possible remedies may be found in Pearl (1984).

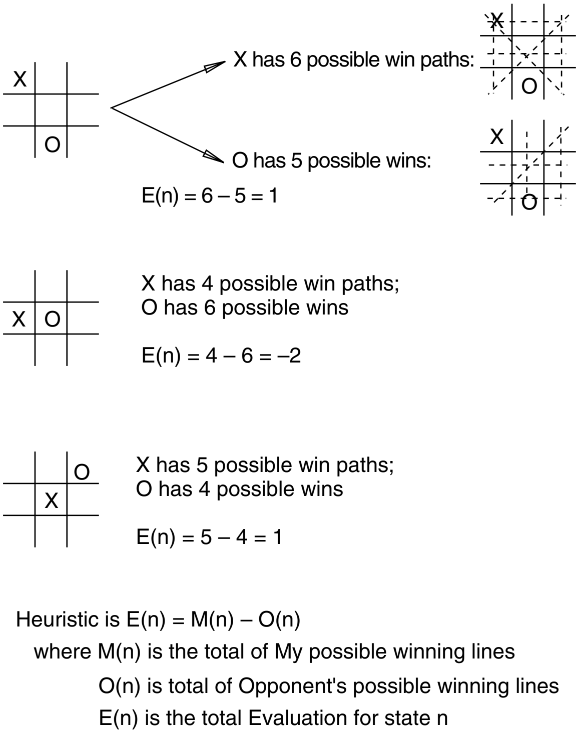

X
O
X has 6 possible win paths:
O has 5 possible wins:
E(n) = 6 – 5 = 1
X has 4 possible win paths;
O has 6 possible wins
E(n) = 4 – 6 = –2
X has 5 possible win paths;
O has 4 possible wins
E(n) = 5 – 4 = 1
Heuristic is E(n) = M(n) – O(n)
where M(n) is the total of My possible winning lines
O(n) is total of Opponent's possible winning lines
E(n) is the total Evaluation for state n

Figure 4.22 Heuristic measuring conflict applied to states of tic-tac-toe.

In concluding the discussion of minimax, we present an application to tic-tac-toe (Section 4.0), adapted from Nilsson (1980). A slightly more complex heuristic is used, one that attempts to measure the conflict in the game. The heuristic takes a state that is to be measured, counts the total winning lines open to MAX, and then subtracts the total number of winning lines open to MIN. The search attempts to maximize this difference. If a state is a forced win for MAX, it is evaluated as +∞; a forced win for MIN as –∞. Figure 4.22 shows this heuristic applied to several sample states.

Figures 4.23, 4.24, and 4.25 demonstrate the heuristic of Figure 4.22 in a two-ply minimax. These figures show the heuristic evaluation, minimax backup, and MAX's move, with some type of tiebreaker applied to moves of equal value, from Nilsson (1971).

4.4.3 The Alpha-Beta Procedure

Straight minimax requires a two-pass analysis of the search space, the first to descend to the ply depth and there apply the heuristic and the second to propagate values back up the tree. Minimax pursues all branches in the space, including many that could be ignored or pruned by a more intelligent algorithm. Researchers in game playing developed a class of search techniques called *alpha-beta* pruning, first proposed in the late 1950s (Newell and Simon 1976), to improve search efficiency in two-person games (Pearl 1984).

The idea for alpha-beta search is simple: rather than searching the entire space to the ply depth, alpha-beta search proceeds in a depth-first fashion. Two values, called *alpha* and *beta*, are created during the search. The alpha value, associated with MAX nodes, can never decrease, and the beta value, associated with MIN nodes, can never increase. Suppose a MAX node's alpha value is 6. Then MAX need not consider any backed-up value

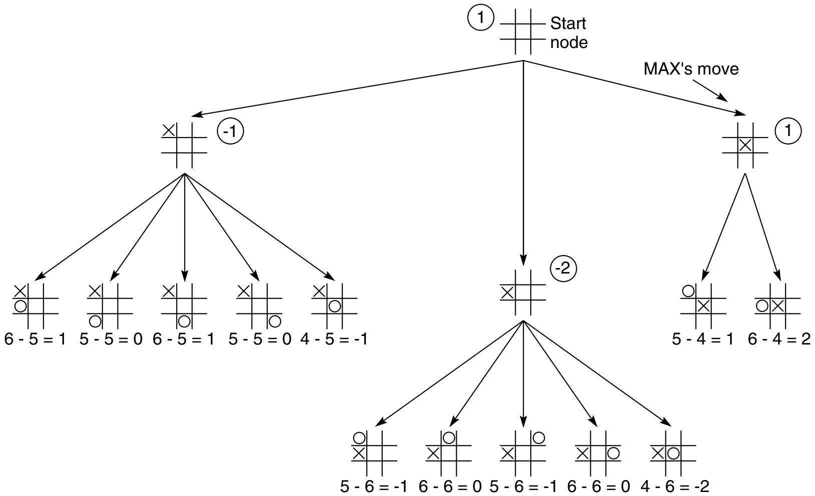


Figure 4.23 Two-ply minimax applied to the opening move of tic-tac-toe, from Nilsson (1971).

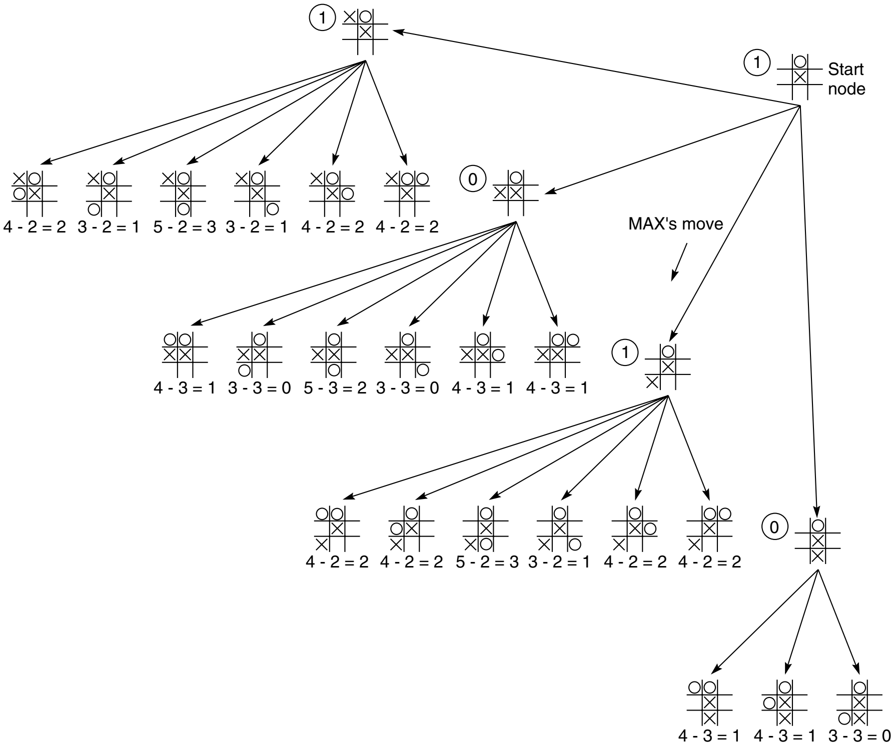


Figure 4.24 Two-ply minimax and one of two possible MAX second moves, from Nilsson (1971).

less than or equal to 6 that is associated with any MIN node below it. Alpha is the worst that MAX can "score" given that MIN will also do its "best". Similarly, if MIN has beta value 6, it does not need to consider any MAX node below that has a value of 6 or more.

To begin alpha-beta search, we descend to full ply depth in a depth-first fashion and apply our heuristic evaluation to a state and all its siblings. Assume these are MIN nodes. The maximum of these MIN values is then backed up to the parent (a MAX node, just as in minimax). This value is then offered to the grandparent of these MINs as a potential beta cutoff.

Next, the algorithm descends to other grandchildren and terminates exploration of their parent if any of their values is equal to or larger than this beta value. Similar procedures can be described for alpha pruning over the grandchildren of a MAX node.

Two rules for terminating search, based on alpha and beta values, are:

1. Search can be stopped below any MIN node having a beta value less than or equal to the alpha value of any of its MAX node ancestors.
2. Search can be stopped below any MAX node having an alpha value greater than or equal to the beta value of any of its MIN node ancestors.

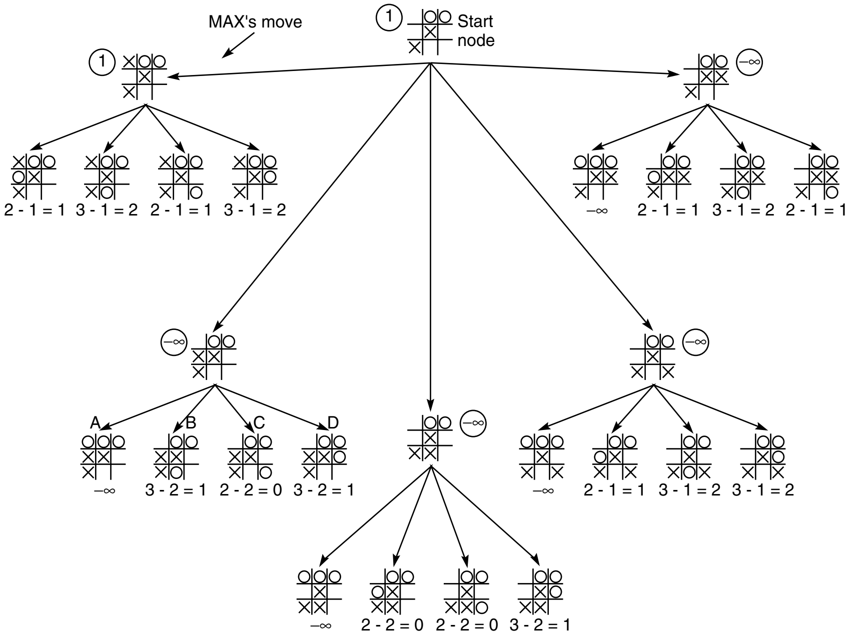


Figure 4.25 Two-ply minimax applied to X s move near the end of the game, from Nilsson (1971).

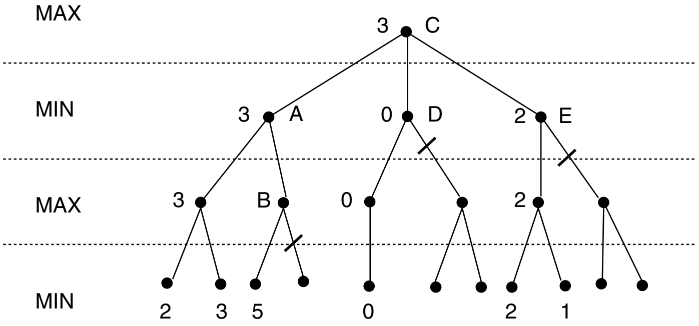
Alpha-beta pruning thus expresses a relation between nodes at ply n and nodes at ply $n + 2$ under which entire subtrees rooted at level $n + 1$ can be eliminated from consideration. As an example, Figure 4.26 takes the space of Figure 4.21 and applies alpha-beta pruning. Note that the resulting backed-up value is identical to the minimax result and the search saving over minimax is considerable.

With a fortuitous ordering of states in the search space, alpha-beta can effectively double the depth of the search space considered with a fixed space/time computer commitment (Nilsson 1980). If there is a particular unfortunate ordering, alpha-beta searches no more of the space than normal minimax; however, the search is done in only one pass.

4.5 Complexity Issues

The most difficult aspect of combinatorial problems is that the "explosion" often takes place without program designers realizing that it is happening. Because most human activity, computational and otherwise, takes place in a linear-time world, we have difficulty appreciating exponential growth. We hear the complaint: "If only I had a larger (or faster or highly parallel) computer my problem would be solved." Such claims, often made in the aftermath of the explosion, are usually rubbish. The problem wasn't understood properly and/or appropriate steps were not taken to address the combinatorics of the situation.

The full extent of combinatorial growth staggers the imagination. It has been estimated that the number of states produced by a full search of the space of possible chess



A has $\beta = 3$ (A will be no larger than 3)
B is β pruned, since $5 > 3$
C has $\alpha = 3$ (C will be no smaller than 3)
D is α pruned, since $0 < 3$
E is α pruned, since $2 < 3$
C is 3

Figure 4.26 Alpha-beta pruning applied to state space of Figure 4.21. States without numbers are not evaluated.

moves is about 10^{120}. This is not “just another large number;” it is comparable to the number of molecules in the universe or the number of nanoseconds since the “big bang”.

Several measures have been developed to help calculate complexity. One of these is the *branching factor* of a space. We define branching factor as the average number of branches (children) that are expanded from any state in the space. The number of states at depth n of the search is equal to the branching factor raised to the nth power. Once the branching factor is computed for a space, it is possible to estimate the search cost to generate a path of any particular length. Figure 4.27 gives the relationship between B (branching), L (path length), and T (total states in the search) for small values. The figure is logarithmic in T, so L is not the “straight” line it looks in the graph.

Several examples using this figure show how bad things can get. If the branching factor is 2, it takes a search of about 100 states to examine all paths that extend six levels deep into the search space. It takes a search of about 10,000 states to consider paths 12 moves deep. If the branching can be cut down to 1.5 (by some heuristic), then a path twice as long can be examined for the same number of states searched.

The mathematical formula that produced the relationships of Figure 4.27 is:

$$T = B + B^2 + B^3 + \cdots + B^L$$

with T total states, L path length, and B branching factor. This equation reduces to:

$$T = B(B^L - 1)/(B - 1)$$

Measuring a search space is usually an empirical process done by considerable playing with a problem and testing its variants. Suppose, for example, we wish to establish the branching factor of the 8-puzzle. We calculate the total number of possible moves: 2 from each corner for a total of 8 corner moves, 3 from the center of each side for a total of 12, and 4 from the center of the grid for a grand total of 24. This divided by 9, the different number of possible locations of the blank, gives an average branching factor of 2.67. As can be seen in Figure 4.27, this is not very good for a deep search. If we eliminate moves directly back to a parent state (already built into the search algorithms of this chapter) there is one move fewer from each state. This gives a branching factor of 1.67, a considerable improvement, which might (in some state spaces) make exhaustive search possible.

As we considered in Chapter 3, the complexity cost of an algorithm can also be measured by the sizes of the open and closed lists. One method of keeping the size of open reasonable is to save on open only a few of the (heuristically) best states. This can produce a better focused search but has the danger of possibly eliminating the best, or even the only, solution path. This technique of maintaining a size bound on open, often a function of the number of steps expected for the search, is called *beam search*.

In the attempt to bring down the branching of a search or otherwise constrain the search space, we presented the notion of *more informed* heuristics. The more informed the search, the less the space that must be searched to get the minimal path solution. As we pointed out in Section 4.4, the computational costs of the additional information needed to further cut down the search space may not always be acceptable. In solving problems on a computer, it is not enough to find a minimum path. We must also minimize total cpu costs.

Figure 4.28, taken from an analysis by Nilsson (1980), is an informal attempt to get at these issues. The "informedness" coordinate marks the amount of information costs included in the evaluation heuristic that are intended to improve performance. The cpu coordinate marks the cpu costs for implementing state evaluation and other aspects of the search. As the information included in the heuristic increases, the cpu cost of the heuristic increases. Similarly, as the heuristic gets more informed, the cpu cost of evaluating states gets smaller, because fewer states are considered. The critical cost, however, is the total cost of computing the heuristic PLUS evaluating states, and it is usually desirable that this cost be minimized.

Finally, heuristic search of and/or graphs is an important area of concern, as the state spaces for expert systems are often of this form. The fully general search of these structures is made up of many of the components already discussed in this and the preceding chapter. Because all and children must be searched to find a goal, the heuristic estimate of the cost of searching an and node is the sum of the estimates of searching the children.

There are many further heuristic issues, however, besides the numerical evaluation of individual and states, in the study of and/or graphs, such as are used in knowledge based systems. For instance, if the satisfaction of a set of and children is required for solving a parent state, which child should be considered first? The state most costly to evaluate? The state most likely to fail? The state the human expert considers first? The decision is important both for computational efficiency as well as overall cost, e.g., in medical or other diagnostic tests, of the knowledge system. These, as well as other related heuristic issues are visited again in Chapter 8.

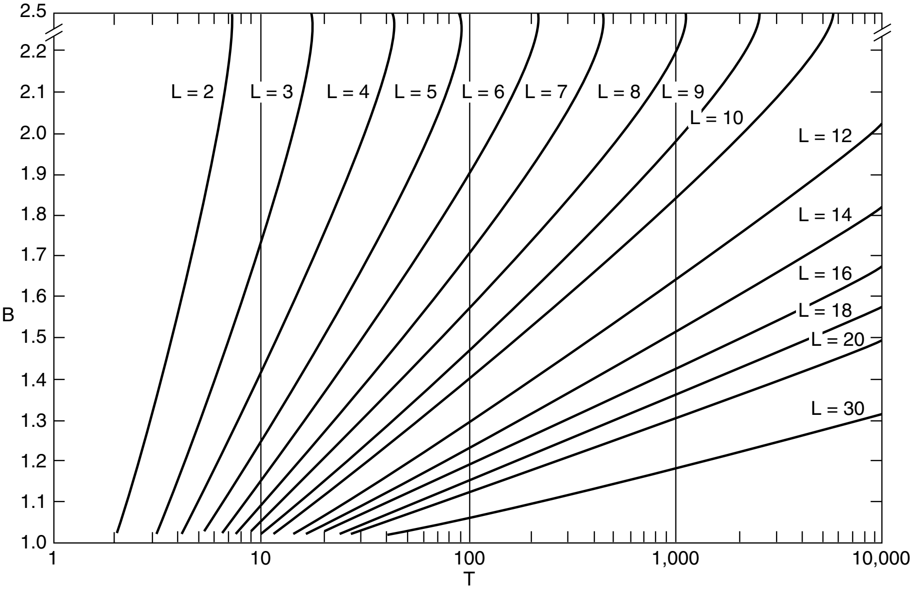


Figure 4.27 Number of nodes generated as a function of branching factor, B, for various lengths, L, of solution paths. The relating equation is: $T = B(B^L - 1)/(B - 1)$, adapted from Nilsson (1980).

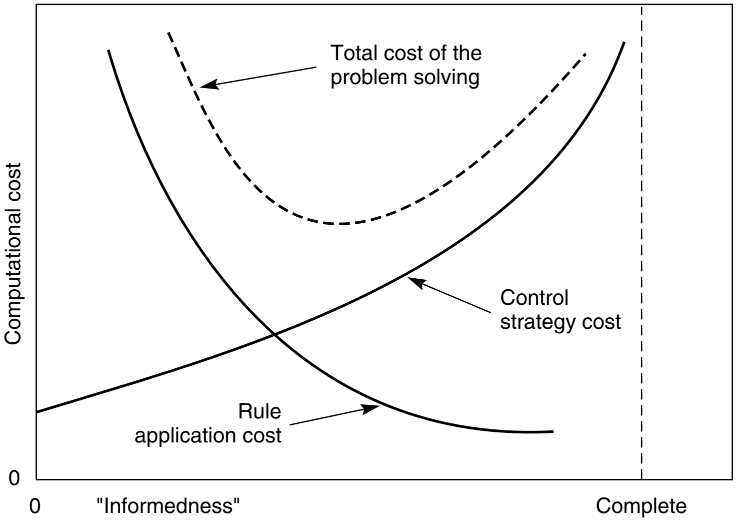


Figure 4.28 Informal plot of cost of searching and cost of computing heuristic evaluation against informedness of heuristic, adapted from Nilsson (1980).

4.6 Epilogue and References

The search spaces for interesting problems tend to grow exponentially; heuristic search is a primary tool for managing this combinatorial complexity. Various control strategies for implementing heuristic search were presented in this chapter.

We began the chapter with two traditional algorithms, both of them inherited from the discipline of operations research, hill-climbing and dynamic programming. We recommend reading the paper by Arthur Samuel (1959) discussing his checker playing program and its sophisticated use of hill-climbing and minimax search. Samuel also presents early interesting examples of a sophisticated memory management system and a program that is able to learn. Bellman's (1956) design of algorithms for dynamic programming remains important in in areas such as natural language processing where it is necessary to compare strings of characters, words, or phonemes. Dynamic programing is often called the forward/backward or Viterbi algorithms. For impotant examples of use of dynamic programming for language analysis see Jurafsky and Martin (2009) and Chapter 15.

We next presented heuristics in the context of traditional state space search. We presented the A and A* algorithms for implementing best-first search. Heuristic search was demonstrated using simple games such as the 8-puzzle and extended to the more complex problem spaces generated by rule-based expert systems (Chapter 8). The chapter also applied heuristic search to two-person games, using look-ahead with minimax and alpha-beta pruning to try to predict the behavior of the opponent. After discussing A* algorithms, we analyzed their behavior, considering properties including admissibility, monotonicity, and informedness.

The discipline of complexity theory has essential ramifications for virtually every branch of computer science, especially the analysis of state space growth and heuristic pruning. Complexity theory examines the inherent complexity of problems (as opposed to algorithms applied to these problems). The key conjecture in complexity theory is that there exists a class of inherently intractable problems. This class, referred to as NP-hard (Nondeterministically Polynomial), consists of problems that may not be solved in less than exponential time without resorting to the use of heuristics. Almost all interesting search problems belong to this class. We especially recommend *Computers and Intractability* by Michael R. Garey and David S. Johnson (1979) and *Algorithms from P to NP, Vol. I: Design and Efficiency* by Bernard Moret and Henry Shapiro (1991) for discussing these issues.

The book *Heuristics* by Judea Pearl (1984) provides a comprehensive treatment of the design and analysis of heuristic algorithms. R. E. Korf (1987, 1998, 1999, 2004) continues research on search algorithms, including an analysis of iterative deepening and the development of the IDA* algorithm. IDA* integrates iterative deepening with A* to obtain linear bounds on open for heuristic search. Chess and other game playing programs have held an abiding interest across the history of AI (Hsu 2002), with results often presented and discussed at the annual conferences.

We are indebted to Nils Nilsson (1980) for the approach and many of the examples of this chapter.

4.7 Exercises

1. Extend the the “most wins” heuristic for tic-tac-toe two plys deeper in the search space of Figure 4.3. What is the total number of states examined using this heuristic? Would the traditional hill-climbing algorithm work in this situation? Why?

2. Use the backward component of the dynamic programing algorithm to find another optimal alignment of the characters of Figure 4.6. How many optimal alignments are there?

3. With the Levenshtein metric of Section 4.1.2, use dynamic programming to determine the minimum edit distance from source strings sensation and excitation to target string execution.

4. Give a heuristic that a block-stacking program might use to solve problems of the form “stack block X on block Y.” Is it admissible? Monotonic?

5. The sliding-tile puzzle consists of three black tiles, three white tiles, and an empty space in the configuration shown in Figure 4.29.
   The puzzle has two legal moves with associated costs:

   > A tile may move into an adjacent empty location. This has a cost of 1. A tile can hop over one or two other tiles into the empty position. This has a cost equal to the number of tiles jumped over.

   The goal is to have all the white tiles to the left of all the black tiles. The position of the blank is not important.

   a. Analyze the state space with respect to complexity and looping.
   b. Propose a heuristic for solving this problem and analyze it with respect to admissibility, monotonicity, and informedness.

6. Compare the three 8-puzzle heuristics of Figure 4.14 with the heuristic of adding the sum of distances out of place to 2 times the number of direct reversals. Compare them in terms of:

   a. Accuracy in estimating distance to a goal. This requires that you first derive the shortest path solution and use it as a standard.
   b. Informedness. Which heuristic most effectively prunes the state space?
   c. Are any of these three 8-puzzle heuristics monotonic?
   d. Admissibility. Which of these heuristics are bounded from above by the actual cost of a path to the goal? Either prove your conclusions for the general case or give a counterexample.

7. a. As presented in the text, best-first search uses the closed list to implement loop detection. What would be the effect of eliminating this test and relying on the depth test, g(n), to detect loops? Compare the efficiencies of the two approaches.

B	B	B		W	W	W

Figure 4.29 The sliding block puzzle.

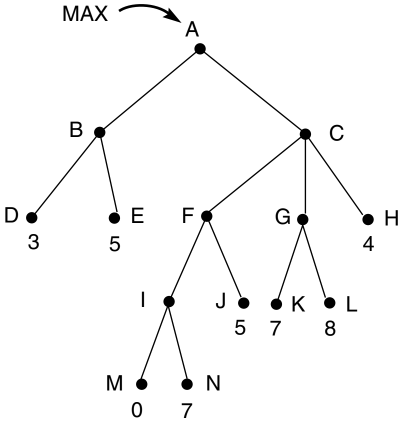


Figure 4.30

b. best_first_search does not test a state to see whether it is a goal until it is removed from the open list. This test could be performed when new states are generated. What effect would doing so have on the efficiency of the algorithm? Admissibility?

8. Prove A* is admissible. Hint: the proof should show that:
   a. A* search will terminate.
   b. During its execution there is always a node on open that lies on an optimal path to the goal.
   c. If there is a path to a goal, A* will terminate by finding the optimal path.

9. Does admissibility imply monotonicity of a heuristic? If not, can you describe when admissibility would imply monotonicity?

10. Prove that the set of states expanded by algorithm A* is a subset of those examined by breadth-first search.

11. Prove that more informed heuristics develop the same or less of the search space. Hint: formalize the argument presented in Section 4.3.3.

12. A Caesar cipher is an encryption scheme based on cyclic permutations of the alphabet, with the i-th letter of the alphabet replaced by the (i + n)-th letter of the alphabet. For example, in a Caesar cipher with a shift of 4, “Caesar” would be encrypted as “Geiwev.”
    a. Give three heuristics that might be used for solving Caesar ciphers.
    b. In a simple substitution cipher, each letter is replaced by another letter under some arbitrary one-to-one mapping. Which of the heuristics proposed for the Caesar cipher may be used to solve substitution ciphers? Explain. (Thanks to Don Morrison for this problem.)

13. Perform minimax on the tree shown in Figure 4.30.

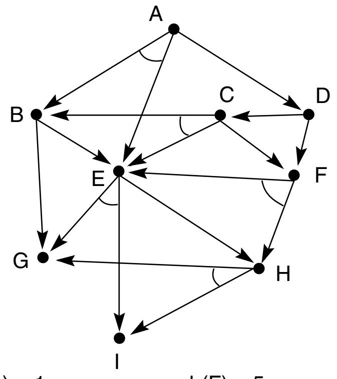
14. Perform a left-to-right alpha-beta prune on the tree of Exercise 13. Perform a right-to-left prune on the same tree. Discuss why a different pruning occurs.

15. Consider three-dimensional tic-tac-toe. Discuss representational issues; analyze the complexity of the state space. Propose a heuristic for playing this game.

16. Perform alpha-beta pruning on the tic-tac-toe search of Figures 4.23, 4.24, and 4.25. How many leaf nodes can be eliminated in each case?

17. a. Create an algorithm for heuristically searching and/or graphs. Note that all descendants of an and node must be solved to solve the parent. Thus, in computing heuristic estimates of costs to a goal, the estimate of the cost to solve an and node must be at least the sum of the estimates to solve the different branches.

    b. Use this algorithm to search the graph in Figure 4.31.



h(A) = 1
h(B) = 2
h(C) = 5
h(D) = 3
h(E) = 2
h(F) = 5
h(G) = 1
h(H) = 3
h(I) = 1

Figure 4.31