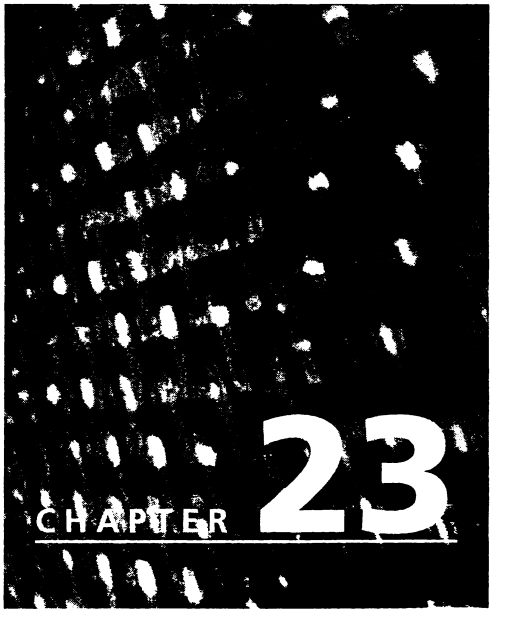


HEURISTIC SEARCH STRATEGIES: A*-SEARCH AND GAME TREES

heuristic: ... *providing aid or direction in the solution of a problem, but otherwise unjustified or incapable of justification. ... of or relating to exploratory problem-solving techniques that utilize self-learning techniques to improve performance.*

Webster's New Collegiate Dictionary

Search strategies such as backtracking, LIFO and FIFO branch-and-bound, breadth-first search, and depth-first search are blind in the sense that they do not look ahead, beyond a local neighborhood, when expanding a node. In this chapter, we show how using heuristics can help narrow the scope of otherwise blind searches. We introduce a type of heuristic search strategy known as A*-search, which is widely used in artificial intelligence (AI). We then discuss strategies for playing two-person games. The alpha-beta heuristic for two-person games is based on assigning a heuristic value to positions reached by looking ahead a certain fixed number of moves. Then an estimate for the best

move is obtained by working back to the current position using the so-called *minimax* strategy.

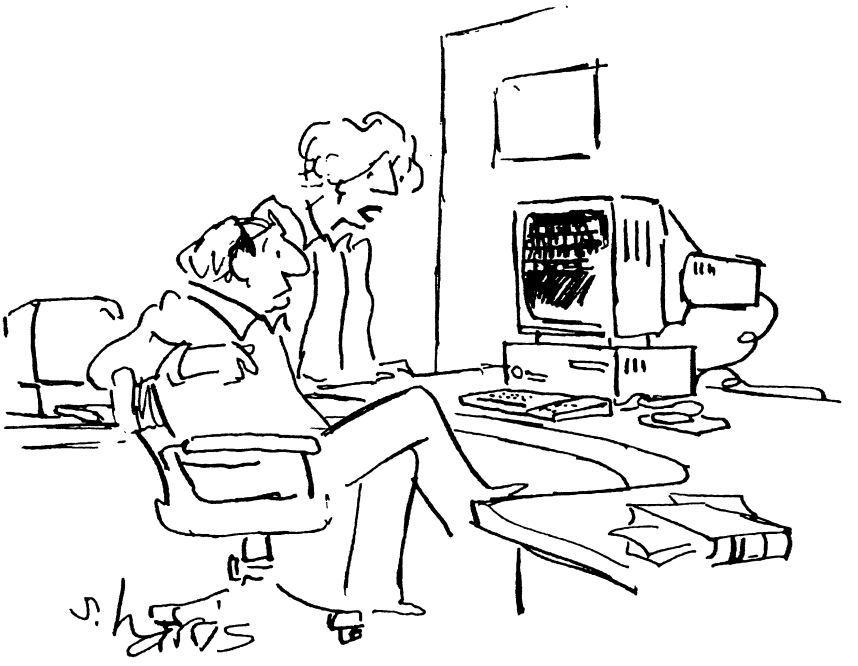
23.1 Artificial Intelligence: Production Systems

The subject of AI is concerned with designing algorithms that allow computers to emulate human behavior (see Figure 23.1). The major areas of AI include natural language processing, automatic programming and theorem proving, robotics, machine vision and pattern recognition, intelligent data retrieval, expert systems, and game playing. Certain activities that are child's play, such as using a natural language, present theoretically and computationally difficult problems that are beyond the reach of current technology. It is true that voice recognition computer programs are currently available that can properly interpret a limited set of spoken instructions. However, the time when we can carry out ordinary conversations with a computer, such as those between the spaceship crew and the computer Hal in the movie *2001: A Space Odyssey*, has yet to be fully realized.

Many problems in AI involve production systems. An AI production system is characterized by *system states* (also called *databases*), *production rules* that allow the system to change from one state to another, and a *control system* that manages the execution of the production rules and allows the system to evolve according to some desired scenario. For example, a system state might be the position of a robotic arm. A production rule allows the robotic arm to change its position. A control strategy is an algorithm that controls the movement of the arm from a given initial position to a final goal position. Here again, a two-year-old child can move his or her arms without much thinking, but designing an algorithm that allows a robot to accomplish the same thing is a complicated task.

Given any AI production system, there is a positive cost associated with the application of each production rule. A production system can be modeled as a positively weighted digraph, called a *state-space digraph,* where a node in the graph is the state of the system, and a directed edge from node v to node w is assigned the cost $Cost(v, w)$ of the production rule that transforms state v into state w. Given some initial state r (in which the root vertex is in the directed graph), we are interested in whether or not we can find a directed path from r to a goal state. A control system for the problem is then simply a search strategy for reaching a goal state starting from r. As usual, we wish to find control systems that perform searches efficiently.

FIGURE 23.1



23.2 8-Puzzle Game

We illustrate the ideas of an AI production system by the 8-puzzle game. The 8-puzzle game is a smaller version of the 15-puzzle game invented by Sam Lloyd in 1878. In the 8-puzzle game, there are eight tiles numbered 1 through 8 occupying eight of the nine cells in a 3×3 square board. The objective is to move from a given initial state in the board to a goal state. The only moves (production rules) allowed are to move a tile into an adjacent empty cell. It is convenient to characterize such a move as a movement of the empty cell. Thus, there are exactly four rules for moving the empty cell: move left, move right, move up, move

down. Of course, the only states allowing all four rules to be applied are when the empty cell is in the center. When the empty cell is at a corner location in the board, then only two of the four rules can be applied. The remaining locations allow the application of three of the four rules.

We will let D be the state-space digraph whose vertex set consists of all possible board configurations in the 8-puzzle game. In D, a directed edge (v, w) exists if a move in the game transforms v into w. Note that if (v, w) is an edge, then (w, v) is also an edge. Thus, D can be considered a state-space graph, where the two directed edges (v, w) and (w, v) are replaced with the single undirected edge $\{v, w\}$. A portion of the state-space graph is shown in Figure 23.2. Suppose we use the breadth-first search control strategy to search for a path leading from the initial state to the goal state. We assume that our control strategy always generates the children of a node in the following order: move left, move right, move up, move down. In Figure 23.3, we have taken an initial position that is only four moves from the goal state. However, 28 states (not counting the initial state) are generated by breadth-first search.

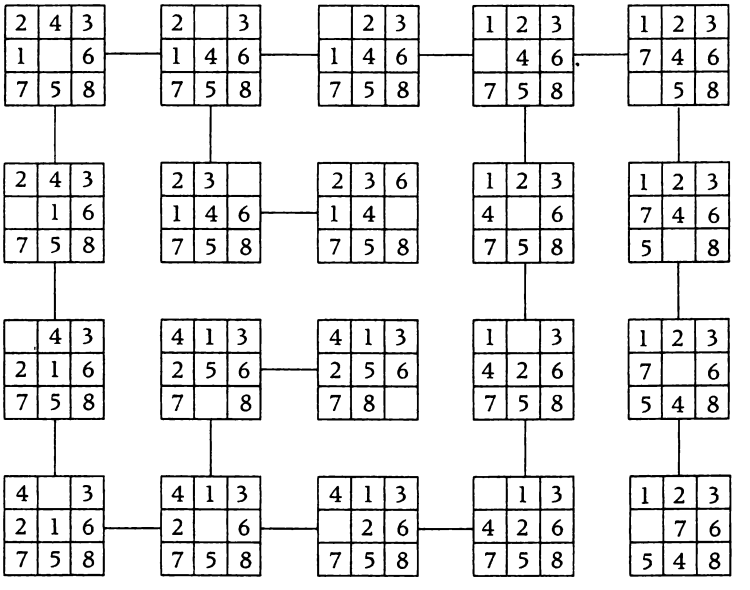


FIGURE 23.2

A portion of the state-space graph for the 8-puzzle game.

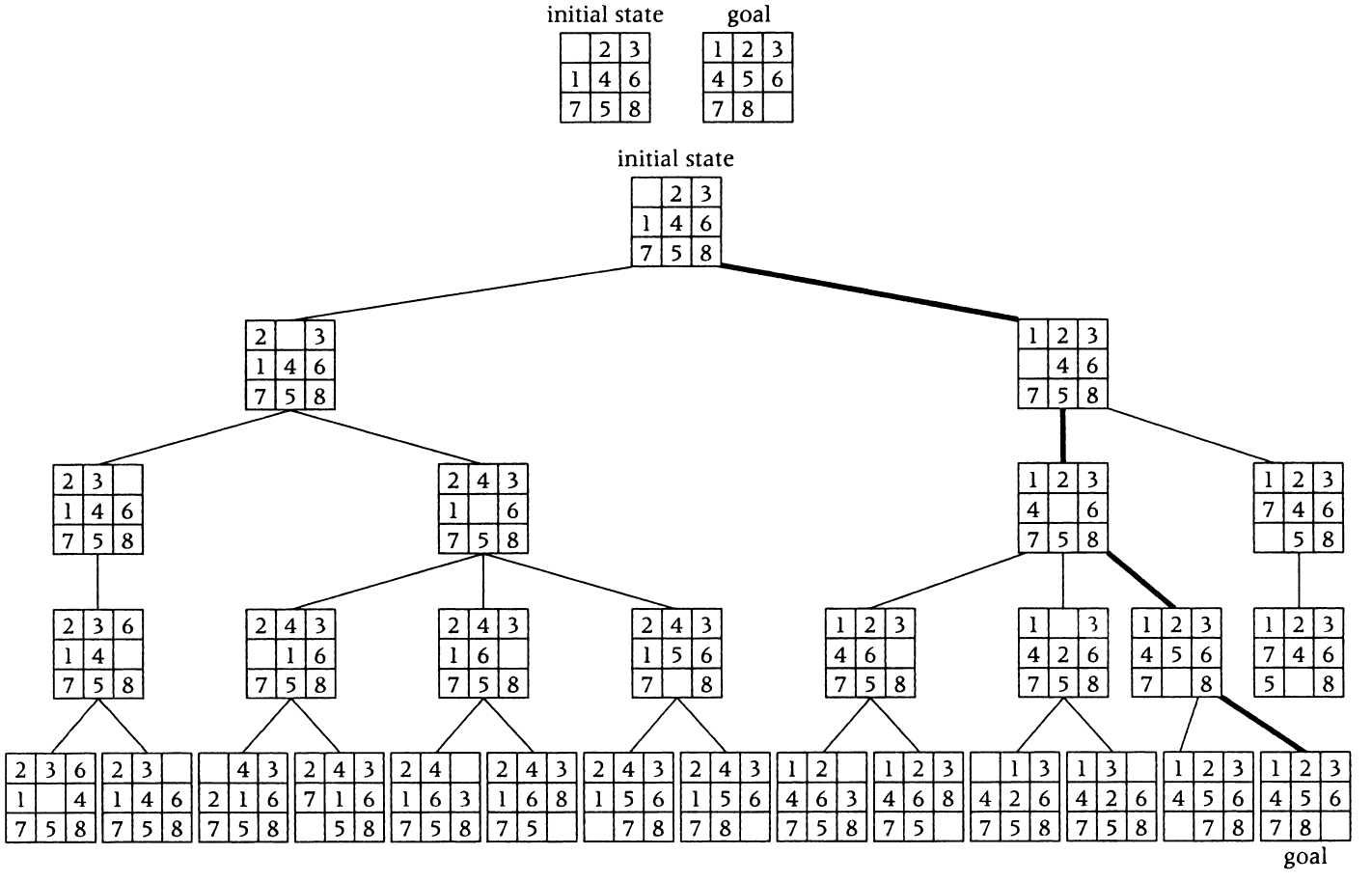


FIGURE 23.3

The states generated by the breadth-first search for a path from the given initial state to the goal state in the 8-puzzle game. The path to the goal is shown in bold.

In general, the number of nodes generated by a breadth-first search for the n-puzzle game is exponential in the minimum number of moves required to reach a goal. Thus, we must look for a better search strategy to solve the problem for initial states requiring many moves to reach the goal. We now describe such a strategy.

23.3 A*-Search

Given a root vertex r and a set of goal states in a state-space digraph, an A*-search is a strategy for finding a shortest path from r to a nearest goal. Such a path is called an *optimal path*. An A*-search finds an optimal path using a generalization of Dijkstra's algorithm. In the discussion of Dijkstra's algorithm in Chapter 12, we maintained an array $Dist[0{:}n-1]$. At each stage, the vertex v minimizing $Dist[v]$ over all vertices not in the tree was added to T. The values of $Dist[w]$ were then updated for all vertices w in the out-neighborhood of v. The operations performed on the array $Dist[0{:}n-1]$ were essentially those of a priority queue. To aid in the description of the A*-search strategy, we now give a high-level description of Dijkstra's algorithm based on maintaining a priority queue of vertices. We also modify Dijkstra's algorithm to terminate once a goal is dequeued.

We denote the priority of a vertex v in the queue by $g(v)$, where the smaller values of g have higher priority. At initialization, the root vertex r is enqueued with priority $g(r) = 0$. When a vertex v is enqueued, a parent pointer from v to $Parent(v)$ is also stored. The parent pointers determine a tree. We call the subtree of the tree spanned by the vertices that have been dequeued the *dequeued tree* T. The tree T contains a shortest path in D from r to each vertex in T.

At each stage in the algorithm, a vertex v in the priority queue is dequeued, and the out-neighbors of v are examined. If an out-neighbor w of v is already in the tree T, then nothing is done to w. If the out-neighbor w has not been enqueued, then it is enqueued with priority $g(w) = g(v) + c(v, w)$, and a parent pointer from w to v is set. Finally, if the out-neighbor w is already on the queue, then the priority of vertex w is updated to $g(w) = \min\{g(w), g(v) + c(v, w)\}$. If $g(w)$ is changed to $g(v) + c(v, w)$, then the parent pointer of w is reset to point to v. The algorithm terminates once a goal is dequeued. The path in the final tree T from r to the goal is an optimal path. The action of the algorithm is shown in Figure 23.4 for a sample digraph.

Dijkstra's algorithm is too inefficient for most AI applications because the shortest-path tree can grow to be huge, even exponentially large. The reason for

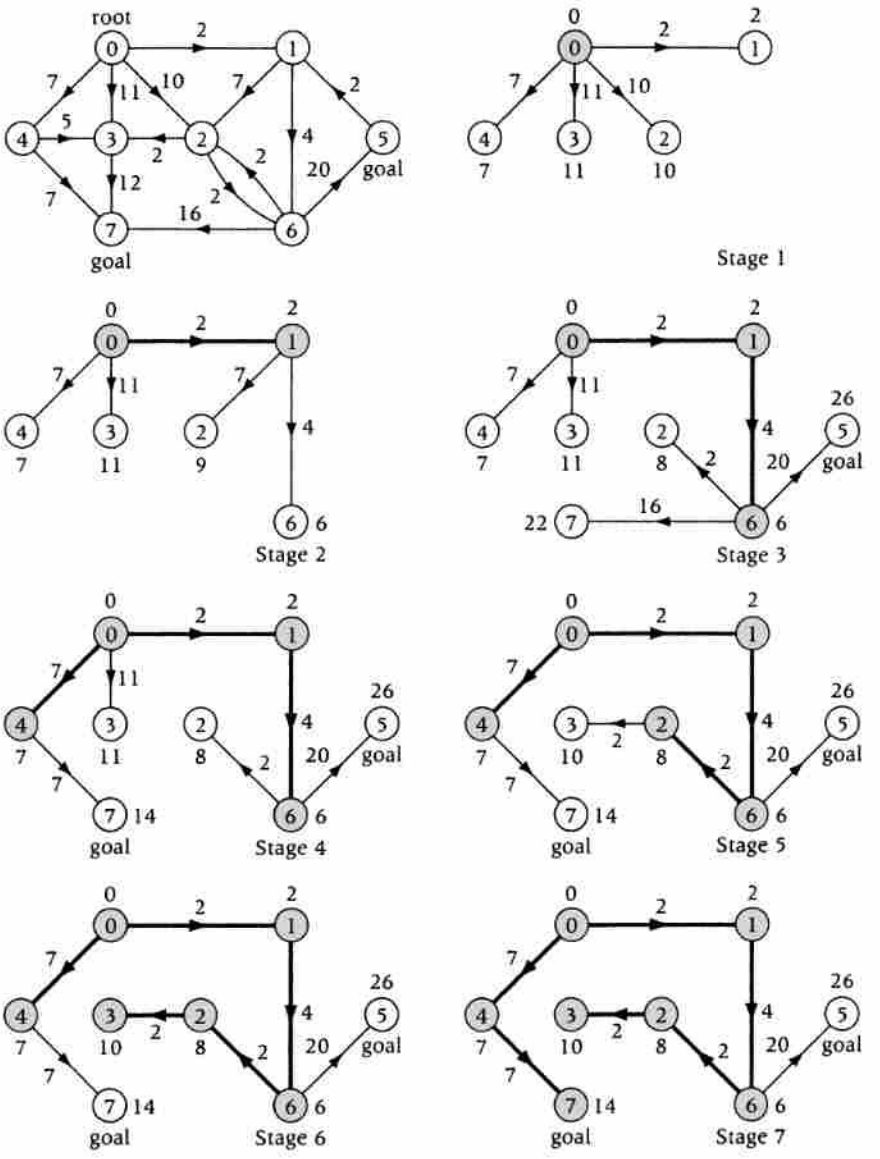


FIGURE 23.4

The action of Dijkstra's algorithm is shown for a sample weighted digraph D. The distance $g(v)$ is shown outside each node v. The vertices and the edges of the dequeued (shortest-path) tree are shaded. The priority queue at each stage consists of vertices w not in the dequeued tree, where the priority of w is $g(w)$.

its inefficiency is that only local information is assumed when looking ahead to a goal. No global information is used that can help the shortest-path tree send out branches in a promising direction. In other words, in Dijkstra's algorithm, the shortest-path tree tends to grow fat (a "shotgun approach" to a goal) rather than grow skinny (a "beeline" approach to a goal).

23.3.1 Heuristics

When information beyond merely the costs of the edges in the digraph is available, Dijkstra's algorithm can be improved so that the shortest-path tree is less expansive and the search is more efficient. The idea is that the priority value $g(v)$ of a vertex v in the queue, which is the cumulative distance (cost) from the root r via the current path determined by the parent pointers, can be replaced by an overall estimate of the cost of the shortest path from r to a goal constrained to go through v. In Dijkstra's algorithm, the priority value of v is $g(v)$, but now we define the priority value of v to be the *cost function*

$$f(v) = g(v) + h(v), \qquad \textbf{(23.3.1)}$$

where $h(v)$ is some estimate of the cost of a shortest path from v to a goal vertex. Because the shortest path from v to a goal has not been found, the best that we can do is use a *heuristic* value for $h(v)$.

When no restriction is placed on the heuristic h in Formula (23.3.1), h is merely a heuristic for a greedy algorithm. When the vertices in the dequeued tree T are reexamined and their parent pointers updated when shorter paths for them are found, the algorithm based on (23.3.1) is called an *A-search*. There is no guarantee that the first path found to a goal is optimal using an A-search. When an A-search uses a heuristic $h(v)$ that is a lower bound of the cost of the shortest path from v to a goal, then the algorithm is called an *A*-search*. We assume that an A*-search terminates when it dequeues a goal, or when the queue is empty. The proof of the following theorem is left to the exercises.

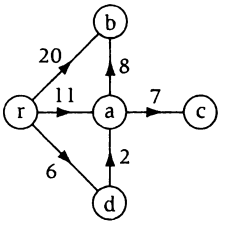
Theorem 23.3.1 Given a positively weighted digraph G (finite or infinite), if a goal is reachable from a root vertex r, then an A*-search terminates by finding an optimal path from r to a goal. □

In an A*-search, when a node v is dequeued, some of its neighbors may already be in the tree T. Unlike Dijkstra's algorithm, an A*-search must check these neighbors to see if shorter paths to them now exist via the vertex v just dequeued. If shorter paths are found, then T must be adjusted to account for them (see Figure 23.5).

Considerable computational cost may be incurred by an A*-search when adjusting the dequeued tree T. This computational cost can be avoided by placing a rather natural and mild restriction on the heuristic h used by A*-search. The heuristic value $h(v)$ is an estimate of the cost of going from v to the nearest goal.

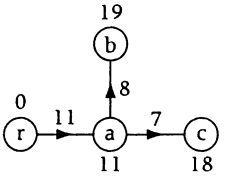
FIGURE 23.5

The number shown outside each node v in T is the value $g(v)$. This example shows how the dequeued tree T can change during an A*-search. Note that the values of g at vertices a, b and c required updating.

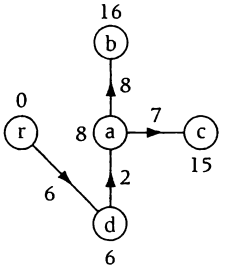


v	r	a	b	c	d
$h(v)$	–	23	20	15	29

Portion of state space induced by vertices r, a, b, c, d, and an associated function h



The dequeued tree T after vertices r, a, b, c have been dequeued



The dequeued tree T after d has been dequeued

If the edge (v, w) exists, then one estimate is $c(v, w) + h(w)$. The restriction on a heuristic, called the *monotone restriction*, says that $h(v)$ should be at least as good as this estimate.

DEFINITION 23.3.1 A heuristic $h(v)$ for an A*-search for a given digraph with cost function c on the edges is said to satisfy the *monotone restriction*, if

$$\begin{aligned} &h(v) \leq c(v, w) + h(w), && \text{whenever the edge } (v, w) \text{ exists,} \\ &h(v) = 0, && \text{whenever } v \text{ is a goal.} \end{aligned} \qquad \textbf{(23.3.2)}$$

If h satisfies the monotone restriction, then we merely say that h is monotone.

If h is a monotone heuristic, then $h(v)$ is a lower bound of the cost of the shortest path from v to a goal. The following proposition states that an A*-search using a monotone heuristic does not need to update parent pointers of a vertex v already in the dequeued tree T, because the path in T from r to v is already a shortest path in D from r to v. The result is consistent with the fact that the A*-search algorithm reduces to Dijkstra's algorithm when $h(v) \equiv 0$ (and the identically zero function trivially satisfies the monotone restriction).

oposition 23.3.2 Suppose an A*-search uses a monotone heuristic. Then the dequeued tree T is a shortest-path tree in the state-space digraph D. In particular, parent pointers for vertices in the dequeued tree T never need updating.

PROOF

For any vertex $v \in V$, let $g^*(v)$ denote the length of the shortest path in D from r to v (so that $g(v) \geq g^*(v)$ at every stage in the execution of an A*-search). We wish to show that $g(v) = g^*(v)$ at the time when v is dequeued. If $v = r$, then we have $g(v) = g^*(v) = 0$; thus, we can suppose that $v \neq r$. Let $P = v_0, v_1, \ldots, v_j$ be a shortest path in D from $r = v_0$ to $v = v_j$. Let vertex v_k be the last vertex in P such that $v_0, v_1, \ldots, v_k$ were all in the tree T when v was dequeued (v_k exists since $r = v_0 \in T$). Then v_{k+1} was in the queue Q at the time when v was dequeued. For any pair of consecutive vertices v_i, v_{i+1} in P, using the monotone restriction, we have

$$g^*(v_i) + h(v_i) \leq g^*(v_i) + h(v_{i+1}) + c(v_i, v_{i+1}). \tag{23.3.3}$$

Now v_i and v_{i+1} are in a shortest path in G, so that

$$g^*(v_{i+1}) = g^*(v_i) + c(v_i, v_{i+1}). \tag{23.3.4}$$

Substituting Formula (23.3.4) in Formula (23.3.3), we obtain

$$g^*(v_i) + h(v_i) \leq g^*(v_{i+1}) + h(v_{i+1}). \tag{23.3.5}$$

Iterating Formula (23.3.5) and using the transitivity of $\leq$ yields

$$g^*(v_{k+1}) + h(v_{k+1}) \leq g^*(v_j) + h(v_j) = g^*(v) + h(v). \tag{23.3.6}$$

Now v_{k+1} is on a shortest path P, and $v_0, v_1, \ldots, v_k$ all belong to T, so that $g(v_{k+1}) = g^*(v_{k+1})$. Hence, Formula (23.3.6) implies

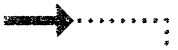
$$f(v_{k+1}) = g(v_{k+1}) + h(v_{k+1}) \leq g^*(v) + h(v) \leq g(v) + h(v) = f(v) \tag{23.3.7}$$

Thus, we must have had $g(v) = g^*(v)$ when v was dequeued; otherwise, $f(v_{k+1}) < f(v)$, and v would not have been dequeued in preference to v_{k+1}. ■

The following proposition helps explain the terminology *monotone restriction*. We leave the proof of Proposition 23.3.3 as an exercise.

Proposition 23.3.3 The f-values of the vertices dequeued by an A*-search using a monotone heuristic are nondecreasing. □

In any problem using an A*-search, the digraph and associated cost function are either implicitly or explicitly input to the algorithm. Here we give examples of both scenarios. When the digraph is very large, it is usually implicitly defined, and only the part of the digraph generated by the execution of the A*-search is made explicit. The following is a high-level description of A*-search using a monotone heuristic. At any given point in the execution of procedure *A*-SearchMH*, T is a subtree of D rooted at the root vertex r containing a path from r to each vertex that has been dequeued by the algorithm. Assuming that a path from r to a goal exists, Theorem 23.3.1 and Proposition 23.3.2 show that when *A*-SearchMH* terminates after dequeuing a goal, the corresponding path to the goal is optimal.



```
procedure A*-SearchMH(D, c, r, GoalSet, h, T)
Input:  D = (V, E) (a digraph, either implicitly or explicitly defined)
            c (a positive cost function on E)
            r (a root vertex in D)
            h (a heuristic function satisfying monotone restriction)
            GoalSet (a set of goal vertices in D)
Output: a shortest-path out-tree T rooted at r containing an optimal path to a goal
        vertex, if one exists
    Q (a priority queue of vertices, with v having priority value f(v) = g(v) + h(v), where
    g(v) is the cost of shortest path P(v) from r to v currently generated. Q also contains
    a parent pointer from v to w ∈ T, where edge (w, v) belongs to P(v))
    while Q is not empty do
        dequeue vertex v in Q with minimum priority value f(v)
        add vertex v to T using parent pointer
        if v ∈ GoalSet then
            return
        endif
        for all vertices w ∉ T and adjacent to v do
            if w ∉ Q then
                enqueue w with parent v and priority value f(w) = g(w) + h(w)
                where g(w) = g(v) + c(v, w)
```

```
            else
                if f(w) ≥ g(v) + c(v, w) + h(w) then
                    reset parent pointer of w to v and update priority value of
                    w to f(w) = g(w) + h(w), where g(w) = g(v) + c(v, w)
                endif
            endif
        endfor
    endwhile
    return "failure"
end A*-SearchMH
```

The action of procedure *A*-SearchMH* is illustrated in Figure 23.6 for a sample digraph *D*.

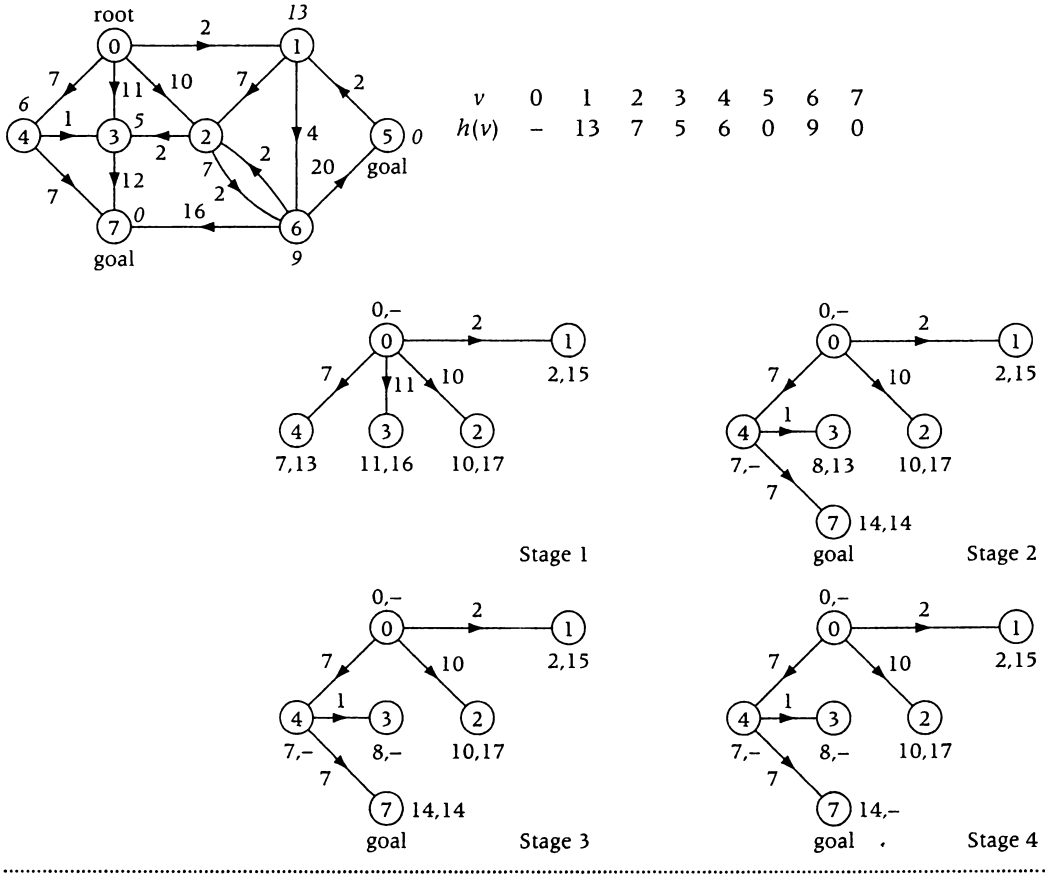


FIGURE 23.6

The action of A*-search with root vertex $r = 0$ and a monotone heuristic $h(v)$ is shown for the same weighted digraph *D* as in Figure 22.4.

We now illustrate procedure *A*-SearchMH* with two examples. First, we revisit the 8-puzzle problem. Then we consider the problem of finding shortest paths between cities in the United States using the freeway system. In the 8-puzzle problem, the state-space graph G is implicitly defined. In the freeway problem, the state-space graph is explicitly input to the algorithm.

23.3.2 A*-Search and the 8-Puzzle Game

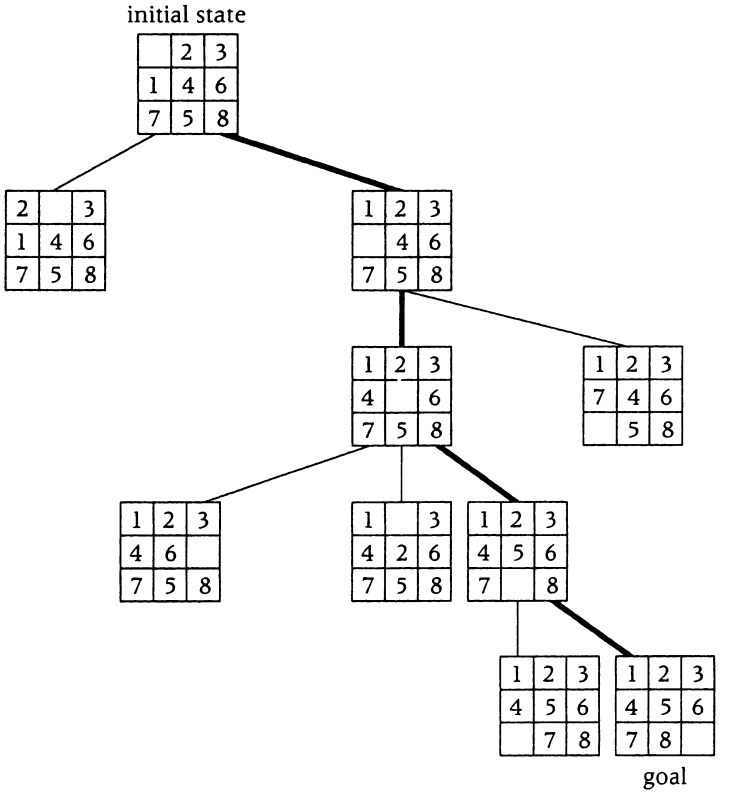
Consider the heuristic

$$h(v) = \text{ the number of tiles not in correct cell in the state } v.$$

It is easy to verify that $h(v)$ satisfies the monotone restriction. Using $h(v)$, Figure 23.7 shows the shortest-path tree generated by the A*-search for the same input as shown in Figure 23.3.

FIGURE 23.7

Shortest-path tree generated by the A*-search for the 8-puzzle game with the same initial state and goal state as in Figure 23.3.



Note that the A*-search only generated 9 states compared to the 28 states generated by the breadth-first search for the same input. Other monotone heuristics exhibit even better behavior in general than the one used in Figure 23.7. For example, for any given state, the sum of the Manhattan distances (vertical steps plus horizontal steps) from the tiles to their proper positions in the goal state exhibits good performance.

23.3.3 Shortest Paths in the Freeway System

Our second example of an A*-search is for the problem of finding a shortest path on freeways between two cities in the continental United States (see Figure 23.8). The heuristic $h(v)$ we use will be a lower bound of the geographical (great-circle) distance between v and the destination city t. We assume that the distances between adjacent cities is available to the algorithm via a suitable adjacency cost matrix. The lower-bound estimate is computed using the longitude and latitude of each city, which we assume are both input to the algorithm as additional information. A lower bound of 50 miles is used for the longitude distance of one degree apart. A lower bound of 70 miles is used for the latitude distance of one degree apart. The square root of the sum of the squares of the longitude distance and the latitude distance between cities v and t is used as $h(v)$. The heuristic $h(v)$ is monotone because the cost (mileage on a freeway between adjacent cities) cannot be smaller than the geographical distance between them. Indeed, if the cost between adjacent cities v and w is $c(v, w)$, then $h(v)$, $h(w)$, and $c(v, w)$ form an almost planar triangle, and we have $h(v) \leq h(w) + c(v, w)$, which is the monotone restriction.

Figure 23.8 shows the graph of the United States as input to a Prolog program, as well as a shortest path from Cincinnati to Houston. Figure 23.9 shows the shortest-path tree generated by Dijkstra's algorithm ($h \equiv 0$), whereas Figure 23.10 shows the shortest-path tree generated by the A*-search. The number of vertices in the shortest path between Cincinnati and Houston is 9. The number

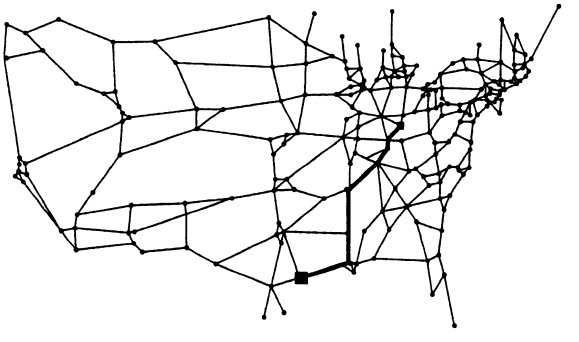


FIGURE 23.8
U.S. freeway system.

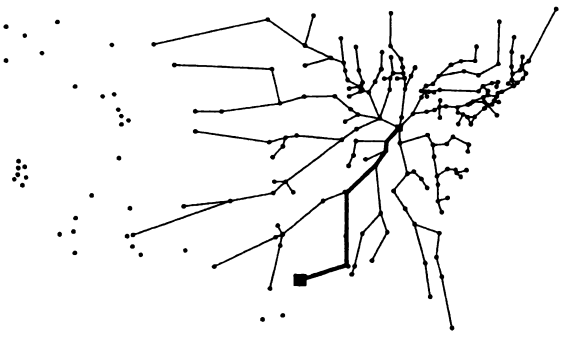


FIGURE 23.9

Shortest-path tree generated by Dijkstra's algorithm.

of vertices expanded when the heuristic is used is 34, compared with 213 vertices when expanded by Dijkstra's algorithm. Thus, the portion of the graph expanded using the A*-search is only 16 percent of that expanded using Dijkstra's algorithm for this example. The contrast between the trees grown in Figure 23.9 and in Figure 23.10 shows rather nicely the difference between the "shotgun" approach to a goal made by Dijkstra's algorithm versus the "beeline" approach made by A*-search.

The selection of a good heuristic is crucial to the success of an A*-search. The closer $h(v)$ is to the actual cost of the shortest path from v to a goal, the fewer nodes will be expanded during the A*-search. However, determining heuristics close to the actual cost is usually too expensive computationally, because determining close estimates is as hard as the original problem. It sometimes speeds the search to use a function for h that does not have the lower-bound property—that is, using an A-search instead of an A*-search. For example, there are better heuristics leading to A-searches for the 8-puzzle game than the monotone heuristic $h(v)$ that was the basis for our A*-search.

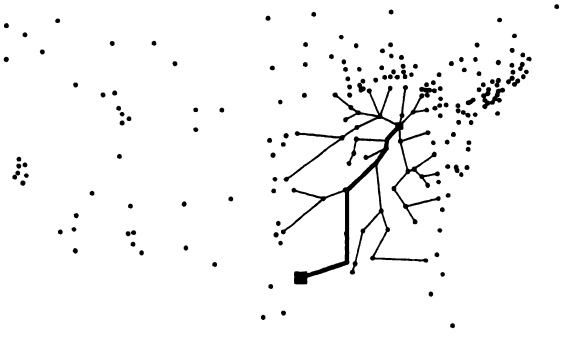


FIGURE 23.10

Shortest-path tree generated by A*-search.

23.4 Least-Cost Branch-and-Bound

Least-cost branch-and-bound is basically an A*-search applied to a state-space tree with the additional use of a bounding function. We use the same notation when describing the state-space tree T as we used in Chapter 10. Least-cost branch-and-bound applies to problems involving minimizing an objective function φ over each solution state in the state-space tree. The cost $c(v)$ of a given node $v = (x_1, \ldots, x_k)$ in the state-space tree is taken to be a lower-bound estimate for

$$\varphi^*(v) = \min\{\varphi(w) | w \in T_v \text{ and } w \text{ is a solution state}\}, \quad \textbf{(23.4.1)}$$

where T_v is the subtree of the state-space tree rooted at v, so that

$$c(v) \leq \varphi^*(v). \quad \textbf{(23.4.2)}$$

Often, the cost function c has the same form as with an A*-search,

$$c(v) = g(v) + h(v), \quad \textbf{(23.4.3)}$$

where $g(v)$ is the cost associated with going from the root to the node v, and $h(v)$ is a heuristic lower-bound estimate of the incremental cost of going from v to a solution state v^* in T_v where φ is minimized (over T_v). Typically, $g(v)$ is $\varphi(v)$, where $\varphi(v)$ is an extension of the objective function φ to all problem states.

For example, in the coin-changing problem (see Chapter 7), $g(v)$ is the number of coins used in the problem state v. A natural heuristic $h(v)$ is obtained in a manner similar to the greedy method. Let $r(v)$ denote the remaining change required. We use as many of the largest-denomination coin as possible, then as many of the next-largest-denomination coin as possible, and continue in this manner as long as we do not exceed $r(v)$. The number of coins so obtained is our heuristic $h(v)$ used for the lower bound $c(v) = g(v) + h(v)$.

As a second example, consider the 0/1 knapsack problem formulated as a minimization problem (see Chapter 10). We then have

$$g(x_1, \ldots, x_k) = \mathit{LeftOutValue}(x_1, \ldots, x_k)$$

and

$$h(x_1, \ldots, x_k) = -\mathit{Greedy}(C', B'),$$

where *LeftOutValue*$(x_1, \ldots, x_k)$ is the sum of the values of the objects not in the set $\{b_{x_1}, \ldots, b_{x_k}\}$, B' is the set of objects $\{b_{x_k+1}, \ldots, b_n\}$, $C' = C - (w_{x_1} + \ldots + w_{x_k})$, and *Greedy*$(C', B')$ is the value of the greedy solution to the (C', B') knapsack problem.

In least-cost branch-and-bound, the live nodes in the state-space tree are maintained as a priority queue with respect to the cost function c. In contrast to our method in A*-search, here we maintain a global variable *UB*, which is the smallest value of the objective function over all solution states already generated. Then a node v can be bounded if $c(v) \geq UB$; moreover, we have the following key fact.

Key Fact

Given a lower-bound cost function, if a node of least cost among the live nodes is bounded, then the algorithm can terminate, having already generated an optimal solution state (goal node).

The following paradigm for a least-cost branch-and-bound search strategy uses the same notation and implementation details for the state-space tree as in Chapter 10.

```
procedure LeastCostBranchAndBound
Input:  function D_k(x_1, . . . , x_{k-1}) (determining state-space tree T associated with the
        given problem)
        objective function φ defined on the solution states of T
        cost function c(v) such that:
        c(v) ≤ φ*(v) = min{φ(w) | w ∈ T_v and w is a solution state}
Output: a solution state (goal) where φ is minimized
    LiveNodes is initialized to be empty
    AllocateTreeNode(Root)
    Root→Parent ← null
    AddPriorityQueue(LiveNodes, Root)          //add root to priority queue of live
                                                 nodes
    Goal ← Root                                //initialize goal to root
    UB ← ∞
    Found ← .false.
    while LiveNodes is not empty .and. .not. Found do
        Select(LiveNodes, E-node, k)           //select E-node of smallest cost
                                                 from live nodes
        if c(E-node) ≥ UB then                 //Goal points to optimal solution
                                                 state
            Found ← .true.
```

```
        else
            if E-node is a solution state and φ(E-Node) < UB then
                                                    //update UB
                UB ← φ(E-Node)
                Goal ← E-Node
            endif
            for each X[k] ∈ D_k(E-node) do          //for each child of the E-node do
                if c(X[k]) < UB .and. .not. StaticBounded (X[1], . . . , X[k]) then
                    AllocateTreeNode(Child)
                    Child→Info ← X[k]
                    Child→Parent ← E-node
                    AddPriorityQueue(LiveNodes, Child)
                                                    //add child to list of live nodes
                endif
            endfor
        endif
    endwhile
    Path(Goal)                                      //output path from goal node to root
end BranchAndBound
```

23.5 Game Trees

Since the invention of the electronic computer, there has been interest in computerized strategies for playing two-player games. For example, particular interest has been focused on designing computer programs to play chess. The first computer programs written for playing chess were not very sophisticated, partly because early computers were not powerful enough to store the vast amounts of information necessary to play a good game. Computer programs implemented on levels for such games as backgammon and computer programs for playing bridge, have exhibited excellent performance.

Computerized game-playing strategies are usually based on the efficient partial search of the enormous game tree modeling all possible legal moves for a given two-player game. The size of the game tree of all possible moves is generally much too large to admit complete searches. Thus, computerized game-playing strategies use heuristics to estimate the value of various moves based on looking down a limited number of levels in the game tree.

Game trees can become quite large even for simple games. For example, consider the game tree associated with tic-tac-toe on a 3×3 board (see Chapter 10). Suppose two players, A and B, are placing Xs and Os, respectively, and player A moves first. Player A has nine possible choices to place the first X. (Of course, using symmetries of the square, only three of these moves are nonequivalent. We

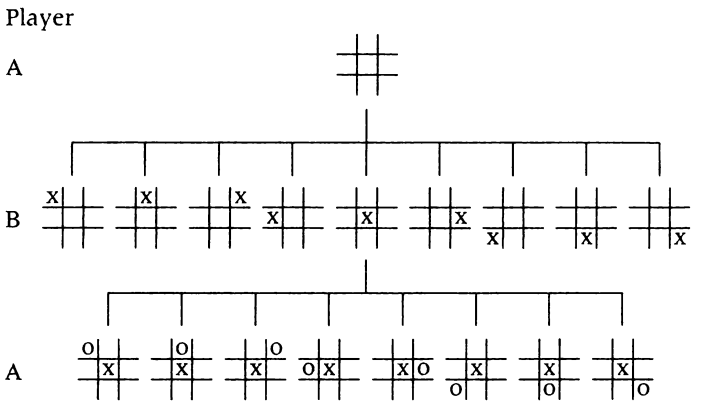


FIGURE 23.11

A portion of the game tree for tic-tac-toe on the 3 × 3 board. Children of a single node at level 1 are shown.

choose to ignore this reduction for the moment.) After player A moves, then player B has eight possible choices for placing the first O. Continuing in this fashion, we see that the game tree has an upper bound of 9! = 362,880 nodes, although the actual number of nodes is smaller because the game terminates whenever a player achieves three Xs or three Os in a row. Figure 23.11 shows all the nodes at levels 0 and 1 of the game tree, but only that portion of the nodes at level 2 corresponding to the eight possible moves from a particular single node at level 1. Figure 23.12 shows all the nodes in the levels 0, 1, and 2 of the pruned game tree that results by pruning symmetric board configurations.

Now consider the game tree modeling an arbitrary game between two players, A and B, who alternately make moves, with each player having complete

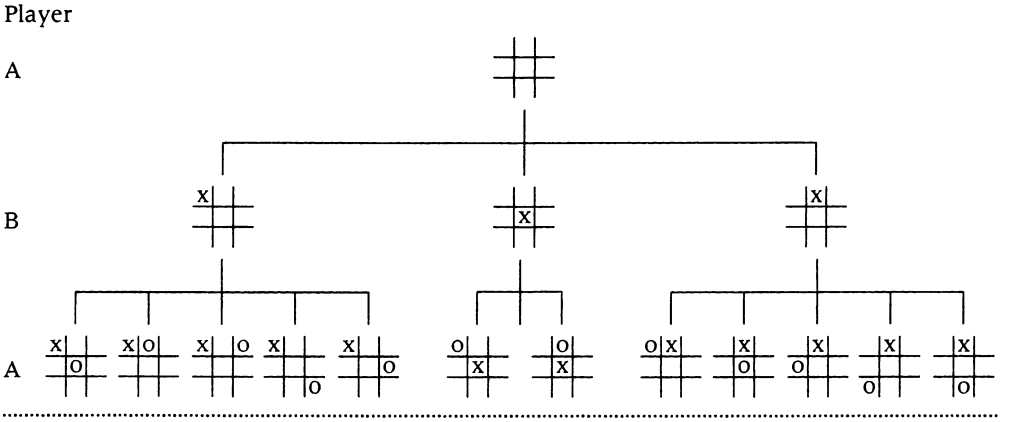


FIGURE 23.12

A portion of the game for tic-tac-toe on the 3 × 3 board pruned by symmetric board configurations. All nodes at the first three levels are shown.

knowledge of the moves of the other (a *perfect information* game). The root of the game tree corresponds to the opening move in the game, which is assumed to be made by player A. Nodes at even levels in the game tree correspond to configurations where it is A's move and are called *A-nodes*. Nodes at odd levels correspond to configurations where it is B's move and are called *B-nodes*. The children of an A-node (respectively, B-node) correspond to all admissible moves available to player A (respectively, player B) from the node. Similar to the game tree for tic-tac-toe, for general games we assume a particular ordering of all admissible moves from a given node, so our game trees are always ordered trees. A leaf node in the game tree is called a *terminal* node and corresponds to the end of the game. In general, a terminal node corresponds to a win, loss, or tie for player A, although certain games such as nim cannot end in a tie.

Given a game tree, the value to player A is assigned to each terminal node (outcome of the game). We also assume that the game is a *zero-sum* game, so that the value to player B of a terminal node is the negative of its value to player A. *However, until further notice, when we speak of the value of a node it is always the value to player A.* We wish to design an algorithm that determines player A's optimal first move. In other words, we want to determine the move that player A should make so that the game will end at a terminal node of maximum value for player A, assuming that each player plays perfectly. Of course, if player B does not play perfectly, then the outcome for player A could be even better.

To analyze the entire game, it is enough to design a procedure to determine the optimal opening move. Indeed, after player A makes the opening move, we would simply repeat (from player B's point of view) the optimal strategy at the game subtree rooted at the node corresponding to this move, and so forth.

If the game tree is small enough to be completely traversed in a reasonable amount of time, then there is a simple *minimax procedure* for player A to determine the optimal opening move. We simply perform a postorder traversal of the game tree, in which a visit at an A-node corresponds to identifying a child of maximum value for a move from the A-node and assigning that value to the A-node. Similarly, a visit at a B-node corresponds to identifying a child of minimum value for a move from the B-node and assigning that value to the B-node.

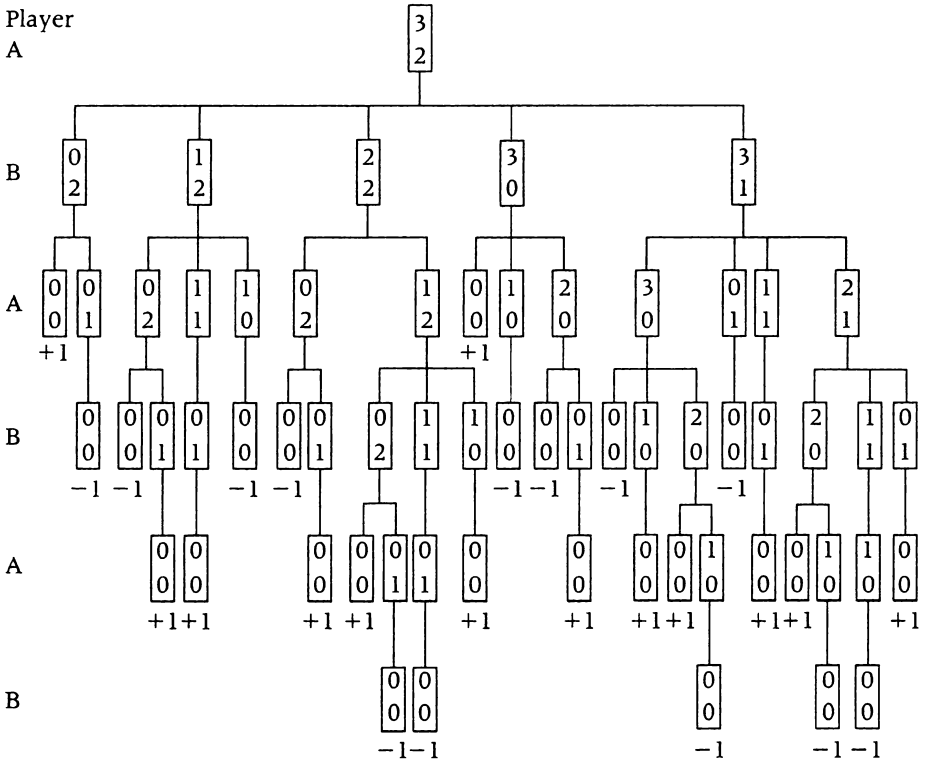
The minimax strategy is nothing more than the definition of perfect play for the two players.

Note that postorder traversal is necessary because the value of each child of a node must be determined before the value of the node itself can be determined. When the postorder traversal is complete, the opening move, together with the value of the game to player A, will be determined.

We illustrate the minimax procedure for the game tree corresponding to a small instance of the game of nim. In the general game of nim, there are n piles of sticks, where the i^{th} pile contains m_i sticks, $i = 1, \ldots, n$. Each player alternately chooses a nonempty pile and removes some or all of the sticks from this pile. There is usually a restriction made on how many sticks a player is allowed to remove in a given move. The last player to remove a stick loses. For large $m_1 + m_2 + \cdots + m_n$, the game tree modeling nim would be enormous. To keep things in sight, consider the instance $n = 2, m_1 = 3, m_2 = 2$. The game tree for this instance is shown in Figure 23.13, where the numbers inside each node correspond to the number of sticks left in each pile. Thus, terminal nodes correspond to 0, 0. We assign the value of +1 to a terminal A-node (A wins) and −1 to a terminal B-node (B wins). In Figure 23.14, we have done some pruning of the complete game tree to eliminate generating symmetric child configurations of the two nodes [1, 1] and [2, 2]. For example, in the complete game tree, the node [2, 2] generates the four nodes [0, 2], [1, 2], [2, 0], and [2, 1]. Using symmetry, we need only display the first two nodes in Figure 23.13 when drawing the (pruned) game tree.

FIGURE 23.13

Game tree for [3, 2] nim, pruned to eliminate symmetric children of [1, 1] and [2, 2]. Terminal node values are +1 when A wins and −1 when B wins.



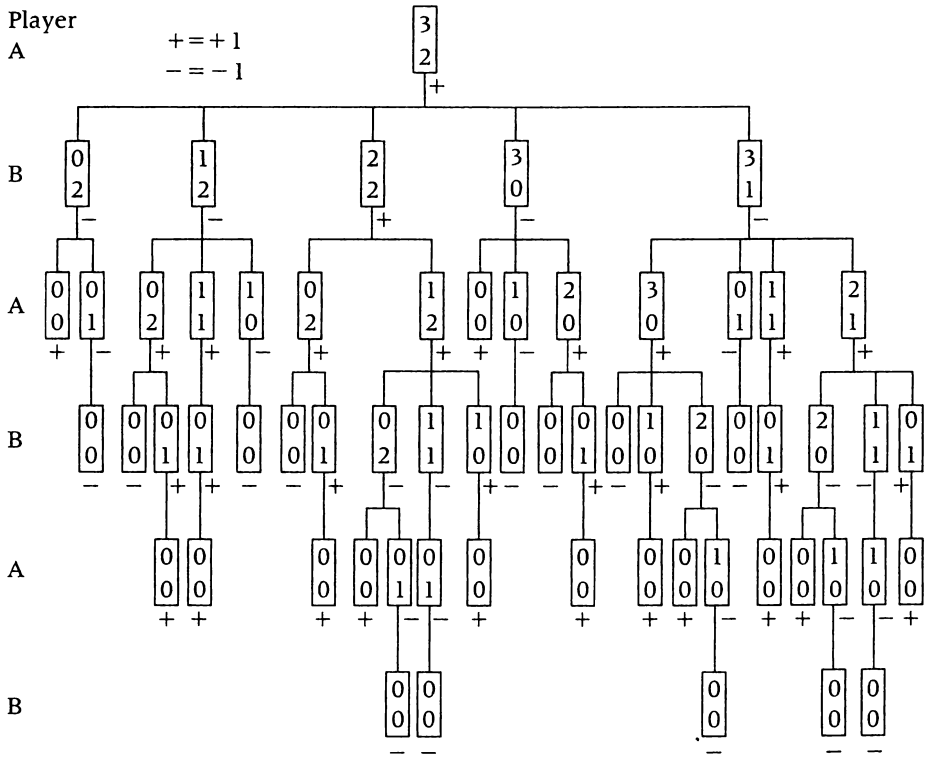


FIGURE 23.14

Values + = +1 and − = −1 are assigned to each node by the minimax postorder traversal of the [3, 2] game of nim.

Figure 23.14 shows the results of a postorder traversal of the game tree in Figure 23.13, where visiting a node executes the minimax procedure described previously. In Figure 23.14, we show the value (to player A) of each node outside the node. Note that the [3, 2] game of nim is a win for player A. The complete traversal of the game tree shows that [2, 2] is a unique opening move that guarantees a win for player A. If a single winner strategy is desired, then the postorder traversal could be terminated as soon as it is determined that the root node has value +1 (with the opening move [2, 2]). Of course, similar termination can be done in any game where we simply have the values +1, 0, and −1 for win, tie, and loss, respectively.

A game in which a complete traversal of the game tree is feasible is usually too small to be interesting. Even the ordinary game of 3 × 3 tic-tac-toe has a rather large game tree. For games like chess, the game tree has been estimated to contain more than 10^{100} nodes. Thus, rather than attempting to traverse the entire game tree when determining an optimal move, in practice the minimax procedure is usually limited to looking ahead a fixed number of levels r in the game tree (*r-level search*). Terminal nodes encountered within r levels are assigned the

value of the outcome of the game corresponding to this node. Nonterminal nodes at level r are assigned some estimate of the value of the node based on the best available knowledge, typically using some heuristic. Nodes that look more promising are given higher values. Of course, the better the estimate, the better the strategy generated by the r-level search.

Suppose we consider 3×3 tic-tac-toe with a two-level search. None of the nodes in the first two levels is a terminal node, so we need to come up with some estimate of the value of each node at level 2. A natural choice would be to assign to a given node (configuration of the board) the number of winning lines completable in Xs minus the number of winning lines completable by Os. For example, Figure 23.15 shows the value of each node at level 2 in the game tree (pruned by symmetries). The figure also shows the result of applying the two-level search (minimax procedure) to the game tree for 3×3 tic-tac-toe, which gives the values of -1, 1, and -2 to the nodes C_1, C_2, and C_3 at level 1, respectively, and gives the root node a value of 1.

We see from Figure 23.15 that player A's opening move would be to place an X in the center position. Then the minimax procedure is continued from the subtree rooted at the latter node. Unfortunately, continuing with the same two-level heuristic search method may lead to a loss for player A (see Exercise 23.24). The fairly obvious fix to this problem is to assign an appropriately large value to terminal positions when encountered in the search We simply give terminal nodes that are wins for player A (that is, three Xs in a row) any value greater than 8, which is the total number of winning lines for the 3×3 game. For example, we could assign the value 9 to terminal nodes that are wins for player A and -9 to terminal nodes that are wins for player B. Terminal nodes corresponding to tie games (cat's games) are assigned the value 0. With the values 9,

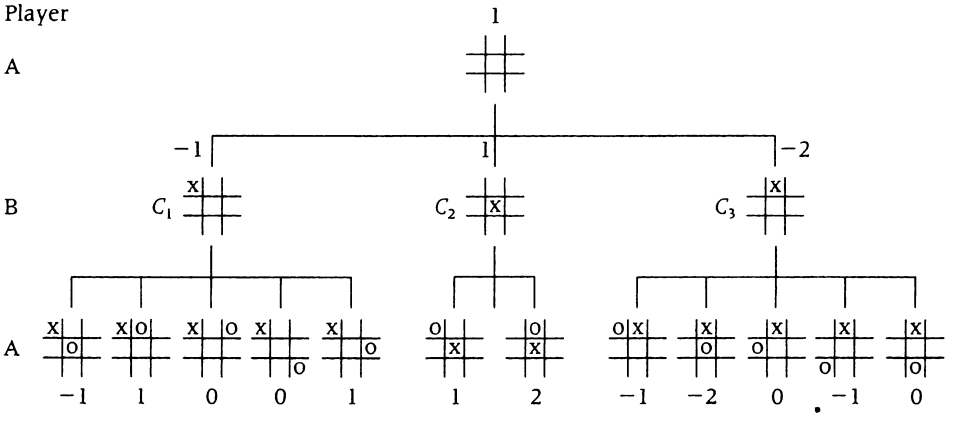


FIGURE 23.15

Value of each node at level 2 as computed using the number of winning lines completable by Xs minus the number of winning lines completable by Os. The values of nodes at levels 1 and 0 are computed using the minimax strategy.

0, and −9 so assigned to terminal nodes, a two-level search leads to a tie game. A tie game for the 3 × 3 board is the best that either player can hope for when both players play perfectly.

There is a heuristic strategy called *alpha-beta pruning* that can result in a significant reduction in the amount of nodes required to visit during an n-level search and still correctly compute the value of a given node. The easiest way to explain alpha-beta pruning is by example. Consider again the two-level search made in the game tree in Figure 23.15. After returning to the root from the middle child C_2, we know that player A can make a move to a node having value 1. Then we move to the third child C_3 of the root and begin visiting the children of C_3 (grandchildren of the root). The first child of C_3 has value −1, so we can immediately cut off our examination of the remaining children of C_3. The reason is simple: The value of C_3 is the minimum value of its children, so the value −1 of the first child of C_3 places an upper bound of −1 on the value of C_3. Because the lower bound on the value of the root is already known to be 1, the value of C_3 cannot possibly affect the value of the root. The cutoff just described is illustrated in Figure 23.16. A cutoff of the search of the grandchildren of an A-node (respectively, B-node) is called *alpha-cutoff* (respectively, *beta-cutoff*).

We formalize the notion of alpha-beta pruning as follows. A lower bound for the value of an A-node is called an *alpha value* of the A-node. Note that during an r-level search, an alpha value of a parent A-node is determined when we return to the A-node from its first child, and the alpha value can be updated, as appropriate, when we return from subsequent children. For example, after returning

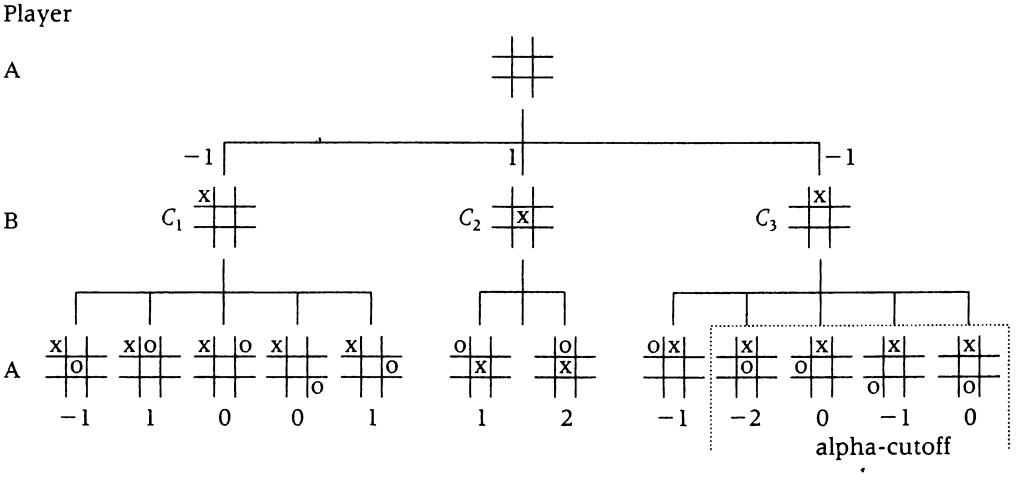


FIGURE 23.16

An alpha-cutoff.

to the root from child C_1 in Figure 23.16, we knew that -1 was an alpha value of the root. However, on returning to the root from the second child C_2, we could update the alpha value of the root to 1.

In general, suppose during an r-level search we are examining the children of the i^{th} child of C_i, where the parent of C_i is an A-node X. If we encounter a child of C_i (grandchild of X) whose value is not larger than an alpha value of X, then we can cut off (alpha-cutoff) our search of the remaining children of C_i, because the value of C_i cannot affect the value of X.

An entirely symmetric discussion holds for B-nodes. Specifically, an upper bound for the value of a B-node is called a *beta value* of the B-node. Given any grandparent B-node Y, if during an r-level search we encounter a child (grandchild of Y) of the i^{th} child D_i of Y whose value is not smaller than a beta value of Y, then we can cut off (beta-cutoff) our search of the remaining children of D_i, because the value of D_i cannot affect the value of Y.

Figure 23.17 illustrates a sample game tree and the effect of alpha-beta pruning for a complete search (that is, a three-level search) of the tree. To illustrate the dynamic nature of alpha and beta values, in Figure 23.17a, we show the indicated alpha and beta values of nodes just after returning to the node X from its second child. The value inside a given node is either the actual value of the node or an alpha or beta value as appropriate. Those nodes containing an alpha or a beta value are flagged as such. A value in a node that is shown as * means that the value is irrelevant because the node is never reached due to alpha-beta pruning. To emphasize the stage of the search in Figure 23.17a, no values are shown inside the nodes of the subtree rooted at the third child of X. In Figure 23.17b, values are supplied for the nodes in the latter subtree, where we show the results of the completed search.

When writing pseudocode implementing the minimax procedure, it is convenient to consider the value of a B-node to be the value to player B, not A. In other words, we simply change the signs of the values given to B-nodes in our previous discussion. These changes simplify the pseudocode by turning the minimax procedure into a max procedure. Note that in the max procedure, the value of either an A-node or a B-node is the maximum of the negatives of the values of their children. Thus, the identical max procedure is executed at an A-node or a B-node.

In the new scenario, our cutoff rule takes the same form whether we are examining the children of an A-node or those of a B-node. In either case, suppose *LB* is a lower bound for the value of the node v whose children are being evaluated, and suppose *ParentValue* is a lower bound for the value of the parent node. If we ever determine that $-LB \leq ParentValue$, then we can cut off further examination of the children of v because the value of v cannot affect the value of the parent of v.

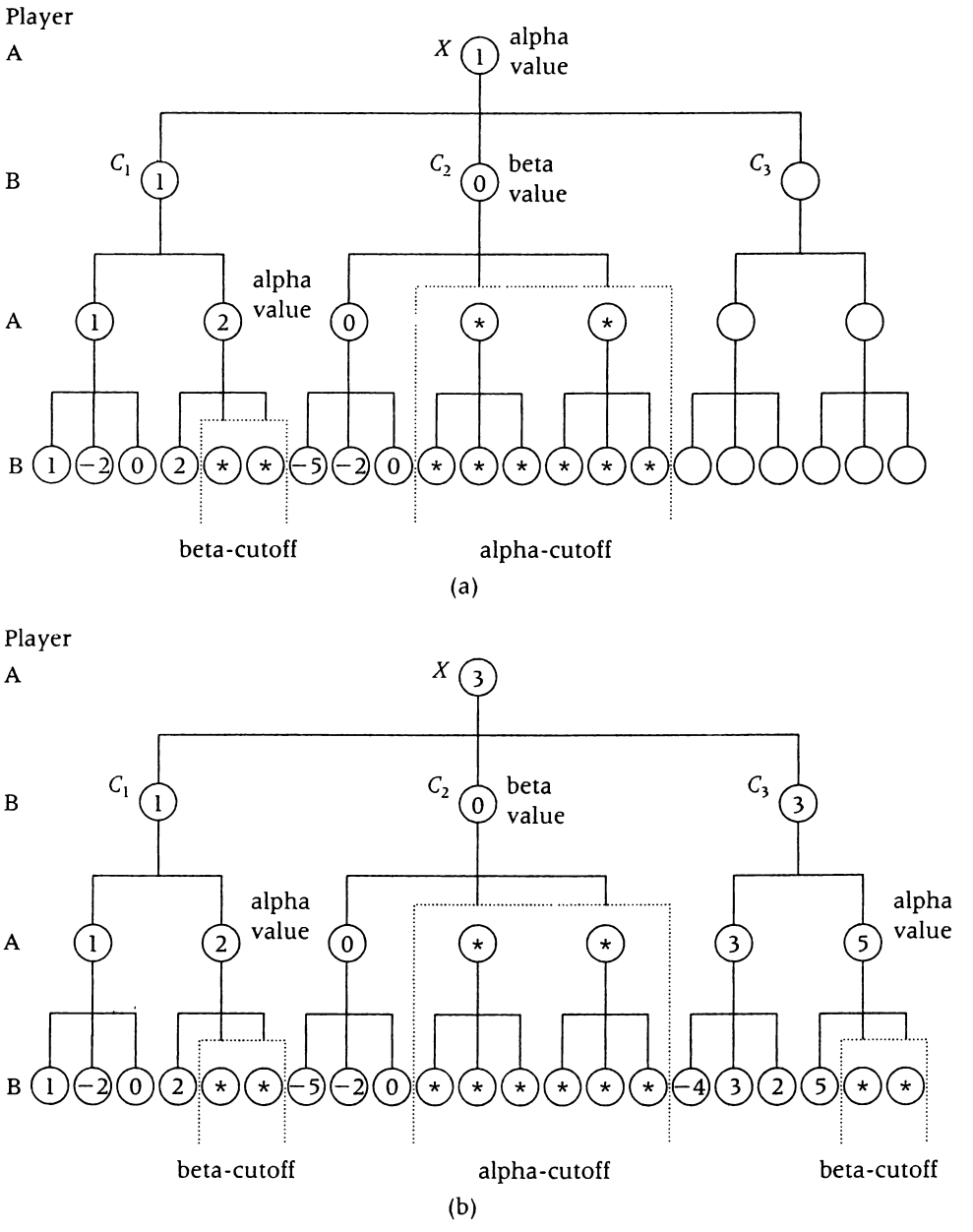


FIGURE 23.17

(a) The results of the search just after returning to X from its second child C_2; b) the results of the completed search.

The following pseudocode for the recursive function *ABNodeValue* returns the value of a node X in the game tree using a *NumLevels*-search, where the parameter *ParentValue* is a lower bound for the value of the parent of X. If X is a terminal node, or if *NumLevels* = 0, then we assume that X has been given an appropriately defined value (denoted by *Val*(X)) as described earlier. Given a

node X and an integer r, the value of X would be calculated using an r-search by invoking *ABNodeValue* initially with arguments X, r, ∞.

```
function ABNodeValue(X, NumLevels, ParentValue) recursive
Input:   X (a node in the game tree having children C1, C2, . . . , Ck),
         NumLevels (number of levels to search)
         ParentValue (lower bound on the value of the parent of X)
Output:  returns the value of X
   if X is a terminal node .or. NumLevels = 0 then
      return(Val(X))
   else
      LB ← −NodeValue(C1, NumLevels − 1, ∞) //initial lower bound for value of X
      for i ← 2 to k do
         if LB ≥ ParentValue then                    //cutoff
            return(LB)
         else
            LB ← max(LB, −NodeValue(Ci, NumLevels − 1, −LB)
         endif
      endfor
   endif
   return(LB)
end ABNodeValue
```

When measuring the efficiency of *ABNodeValue*, the quantity of interest is the number of nodes cut off from the straight minimax r-search of the game tree that does not use the cutoff rule. Of course, not much can be said in general unless some assumptions are made about the regularity of the game tree. Even with strong regularity conditions imposed on the game tree, the analysis is difficult. We merely state one result in this direction. Perl has shown that for game trees in which each parent has the same number of children, and in which the terminal nodes are randomly ordered, *ABNodeValue* permits a search depth greater by a factor of 4/3 than that allowed by the straight minimax procedure in the same amount of time.

Alpha-beta pruning can be enhanced by adding an additional parameter *NodeValueLowBnd* into the algorithm *ABNodeValue*. *NodeValueLowBnd* is maintained as a lower bound on the value on the input parameter X. Additional pruning of the game tree results from the following key fact.

If the value of a grandchild of *X* is not larger than *NodeValueLowBnd*, then all remaining children of the grandchild can be pruned.

Whereas alpha-beta pruning only uses information from the parent, *NodeValueLowBnd* carries information deep into the tree, and the resulting evaluation of the game tree is called *deep* alpha-beta pruning. We leave the design of the recursive function for deep alpha-beta pruning as an exercise.

23.6 Closing Remarks

Seeking solutions to the 8-puzzle game or finding a shortest-length trip along a freeway system are examples of what has been called *single-agent* problems. In general, an A*-search is better suited for a large-scale problem in which the *entire solution* is sought in a reasonable length of time (and then saved for future reference) than for a real-time problem in which the first step (and each successive step) in the solution must be computed very quickly. For example, it might be acceptable for a computer to take weeks or even months to solve a highly important single-agent problem because its solution would then be known and usable in real time thereafter.

An ordinary A*-search as applied to a single-agent problem usually finds the entire path to a goal before even the first move from the starting position is definitely known. Hence, using an A*-search for a single-agent problem becomes too costly for a large-scale application where the optimal decisions along the way must be made quickly and in advance of the final solution. For example, you might be under a short time constraint to make each move in a game like the 8-puzzle game, rather than simply wanting to determine the *entire solution* in a larger but more reasonable length of time.

While most single-agent problems are not subject to intermediate real-time constraints, two-player games usually are. Chess, for example, usually restricts the amount of time a player has to make the next move. Moreover, the game tree for chess is so enormous that generating complete solutions is out of the question. To make real-time decisions, the alpha-beta heuristic is based on attempting to evaluate moves in a limited *search horizon*—that is, looking ahead a fixed number of moves.

For a single-agent problem, a heuristic search method called *real-time A*-search* combines the A*-search strategy with a limited look-ahead search horizon. A real-time A*-search uses an analog of minimax alpha-beta pruning called minimin *alpha pruning*. Alpha pruning drastically improves the efficiency of A*-search without affecting the decisions made. Like an A*-search, a real-time A*-search can find the entire solution to a fairly large-scale problem in a reasonable amount of time. However, a real-time A*-search has the advantage of generating the optimal moves along the way quickly and before the entire solution is known. Refer to the references for further information on the real-time A*-search.

Suppose that a perfect-information game involving alternate moves by two players A and B must end in a finite number of moves and a win for one of the

two players. Then one of the two players must have a *winning strategy*—that is, a strategy that guarantees a win regardless of the moves made by the other player. The reason is simple: If neither player had a winning strategy, there would be a sequence of alternate moves made by A and B that never ends in a loss for either player. Because that sequence would not terminate, we would obtain a contradiction of the finiteness assumption of the game.

A two-person *positional* game is determined by a collection of sets A_i, $i = 1, \ldots, m$. The players alternately choose an element (which they keep) from $\cup_{i=1}^{n} A_i$. The first player to choose *all* the elements from one of the sets wins. Tic-tac-toe is an example of a positional game, where the sets A_i are the winning lines in the board. For positional games that cannot end in a tie, such as the $3 \times 3 \times 3$ game of tic-tac-toe, the first player always has a winning strategy. Indeed, we have just seen that one of the two players must have a winning strategy, so suppose it is the second player. Then the first player makes a random opening move, and thereafter assumes the role of the second player (basically ignoring the opening move). More precisely, when the first player moves, he chooses the move dictated by the second player's winning strategy, or moves randomly if he has previously made this move. Since having made an extra move in a positional game cannot possibly hurt, the first player is thus led to a win! This contradicts the assumption that the second player has a winning strategy and shows that the first player has a winning strategy. The same argument shows that if there is a winning strategy for a positional game, then it must belong to the first player.

For positional games that cannot end in a tie, the fact that the first player has a winning strategy does not mean that there is an efficient algorithm to generate the strategy. Also, for positional games that *can* end in a tie (given, perhaps, imperfect play), there still might exist a winning strategy for the first player. For example, tie positions exist for the $4 \times 4 \times 4$ game of tic-tac-toe (winning sets being four in a row). However, it was conjectured for a long time that the first player has a winning strategy in $4 \times 4 \times 4$ tic-tac-toe. This conjecture was finally established by Patashnik using clever bounding arguments that allowed a pruning of the enormous game tree for $4 \times 4 \times 4$ tic-tac-toe, reducing it to a size that was amendable to computer search.

References and Suggestions for Further Reading

Kanal, L., and V. Kumar, eds. *Search in Artificial Intelligence*. New York: Springer-Verlag, 1988. Contains numerous articles on search in artificial intelligence, including a discussion on the optimality of the A*-search.

Korf, R. E. "Real-Time Heuristic Search," *Artificial Intelligence* 42 (1990): 189–211. A paper devoted to the real-time A*-search.

Nilsson, N. J. *Principles of Artificial Intelligence.* Palo Alto, CA: Tioga, 1980. A detailed account of the A*-search, which was originally developed by Hart, Nilsson, and Raphael.

Pearl, J. *Heuristics: Intelligent Search Strategies for Computer Problem Solving.* Reading, MA: Addison-Wesley, 1984. A text devoted to heuristic searching.

Two books on artificial intelligence that contain extensive discussions of search strategies and game playing:

Rich, E., and K. Knight. Artificial Intelligence. 2nd ed. New York: McGraw-Hill, 1991.

Russell, S. J., and P. Norvig. Artificial Intelligence: A Modern Approach. Englewood Cliffs, NJ: Prentice Hall, 1995.

Patashnik, O. "Qubic: 4 × 4 × 4 Tic-Tac-Toe," Mathematics Magazine 53 (1980): 202–223. Survey discussion of n-dimensional tic-tac-toe, as well as the proof that the 4 × 4 × 4 is a first-player win.

Two papers containing detailed analyses of alpha-beta and deep alpha-beta pruning:

Baudet, G. "An Analysis of the Full Alpha-Beta Pruning Algorithm," Proceedings of the 10th Annual ACM Symposium on Theories of Computing, San Diego, CA: Association for Computing Machinery, 1978, pp. 296–313.

Knuth, D. "An Analysis of Alpha-Beta Cutoffs," Artificial Intelligence 6 (1975): 293–323.

Berlekamp, E. R., J. H. Conway, and R. K. Guy. Winning Ways, for Your Mathematical Plays. Vol. I, II. New York: Academic Press, 1982. Covers strategies for a host of games.

EXERCISES

Section 23.2 8-Puzzle Game

23.1 Draw the first three levels of the state-space tree generated by a breadth-first search for the 8-puzzle game with the following initial and goal states:

goal

1	2	3
4	5	6
7	8	

initial state

4	2	7
1		6
3	5	8

23.2 The $(n^2 - 1)$-puzzle is a generalization of the 8-puzzle to the $n \times n$ board. The goal position is where the tiles are in row-major order (with the empty space in the lower-right corner). For $k \in \{1, \ldots, n^2\}$, let $L(k)$ denote the number of tiles t, $t < k$, such that the position of t comes after k in the row-major order in the initial arrangement (the empty space is considered as tile n^2). Show that a necessary and sufficient condition that the goal can be reached is that

$$\sum_{k=1}^{n^2} L(k) \equiv i + j \pmod 2, \qquad \textbf{(23.3.4)}$$

where (i, j) is the position of the empty space in the initial arrangement.

Section 23.3 A*-Search

23.3 Show that if h is a monotone heuristic, then $h(v)$ is a lower bound of the cost of the shortest path from v to a (nearest) goal.

23.4 Prove Proposition 23.3.3.

23.5 Design an algorithm for an A*-search using a heuristic that is not necessarily monotone.

23.6 For the 8-puzzle game, consider the following heuristic:

$h(v)$ = the number of tiles not in correct cell in the state v.

Show that $h(v)$ satisfies the monotone restriction.

23.7 For the 8-puzzle game, consider the following heuristic:

$h(v)$ = the sum of the Manhattan distances (vertical steps plus horizontal steps) from the tiles to their proper positions in the goal state.

Show that h satisfies the monotone restriction.

23.8 For the 8-puzzle game, let h be the monotone heuristic defined in Exercise 23.6. For the following initial and goal states, draw the states generated by making the first three moves in the game using an A*-search with the priority function $f(v) = g(v) + h(v)$ [$g(v)$ is the number of moves made from the initial state to v]. When enqueuing states, assume that the

(possible) moves of the empty tile are ordered as follows: move left, move right, move up, move down. Label each state v with its f-value.

goal

1	2	3
4	5	6
7	8	

initial state

4	2	7
1		6
3	5	8

23.9 Repeat Exercise 23.8 for the heuristic h defined in Exercise 23.7.

23.10 Write a program for the n-puzzle game using the Manhattan distance heuristic. Test your program for $n = 8$ and $n = 15$.

23.11 Can you find better heuristics (not necessarily lower bounds) for the n-puzzle game than the Manhattan distance heuristic? Test your heuristic empirically for $n = 8$ and $n = 15$.

23.12 Prove Theorem 23.3.1

Section 23.4 Least-Cost Branch-and-Bound

23.13 Show that the heuristic $h(v)$ given in Section 23.4 for the coin-changing problem is a lower bound for the minimum number of additional coins required to make correct change from the given problem state v.

23.14 Write a program implementing a least-cost branch-and-bound solution to the coin-changing problem.

23.15 Design a heuristic and a least-cost branch-and-bound algorithm for the variation of the coin-changing problem in which we have a limited number of coins of each denomination. Assume the number of coins of each denomination is input along with the denominations.

23.16 Draw the portion of the variable-tuple state-space tree generated by least-cost branch-and-bound for the instance of the 0/1 knapsack problem given in Figure 10.13 in Chapter 10, using the heuristic given in Section 23.4. Label each node with the value of $c(v)$ and the current value of UB.

23.17 Repeat Exercise 23.16 for the fixed-tuple state-space tree.

23.18 Write a program implementing a least-cost branch-and-bound solution to the 0/1 knapsack problem.

23.19 Given the complete digraph $\hat{K}_n$ with vertices $0, 1, \ldots, n-1$ and a nonnegative cost matrix $C = (c_{ij})$ for its edges (we set $c_{ij} = \infty$ if $i = j$ or if the edge ij does not exist), a traveling salesman tour starting at vertex 0 corresponds to a sequence of vertices $0, i_1, i_2, \ldots, i_{n-1}, 0$, where $i_1, i_2, \ldots, i_{n-1}$ is

a permutation of $1, \ldots, n-1$. Consider a state-space tree T for the traveling salesman problem (finding a minimum-cost tour) where a node at level k in T corresponds to a simple path containing $k+1$ vertices, starting with vertex 0. Thus, T has depth n, and leaf nodes correspond to a sequence of choices $i_1, i_2, \ldots, i_{n-1}$, determining the tour $0, i_1, i_2, \ldots, i_{n-1}, 0$.

We now describe a cost function $c(v)$ for a least-cost branch-and-bound algorithm for the traveling salesman problem. The definition of $c(v)$ is based on the notion of a reduced-cost matrix. A row (or column) of a nonnegative cost matrix is said to be *reduced* if it contains at least one zero. A nonnegative cost matrix is *reduced* if each row and column of the matrix is reduced (except for rows and columns whose elements are all equal to ∞). Given the cost matrix C, an associated reduced-cost matrix C_r is constructed as follows. First, reduce each row by subtracting the minimum entry in the row from each element in the row. In the resulting matrix, repeat this process for each column. We define $c(r)$ to be the total amount subtracted. The following example illustrates C_r for a sample C.

$$C = \begin{pmatrix} \infty & 23 & 9 & 32 & 12 \\ 21 & \infty & 2 & 16 & 4 \\ 4 & 8 & \infty & 20 & 6 \\ 15 & 10 & 4 & \infty & 2 \\ 9 & 5 & 8 & 10 & \infty \end{pmatrix} \qquad C_r = \begin{pmatrix} \infty & 14 & 0 & 18 & 3 \\ 19 & \infty & 0 & 9 & 2 \\ 0 & 4 & \infty & 11 & 2 \\ 13 & 8 & 2 & \infty & 0 \\ 4 & 0 & 3 & 0 & \infty \end{pmatrix} \qquad c(r) = 27$$

More generally, we define (inductively on the levels of T) a reduced-cost matrix for each nonleaf node by suitably reducing the cost matrix C_u associated with the parent node u of v. Suppose u corresponds to a path ending at vertex i, and v corresponds to adding the edge ij to this path. We then change all the entries in row i and column j of C_u to ∞, as well as the entry in the j^{th} row and first column. We then perform the same subtracting operation on the resulting matrix as we did when computing C_r. Let s_v denote the total amount subtracted, and define $c(v) = c(u) + C_u(i, j) + s_v$.

For leaf nodes v, $c(v)$ is defined as the cost of the tour determined by v.

a. Show that $c(r)$ is a lower bound for the minimum cost of a tour.

b. More generally, show that $c(v) \leq \phi^*(v)$ = the minimum cost over all tours determined by the leaf nodes of the subtree of T rooted at v.

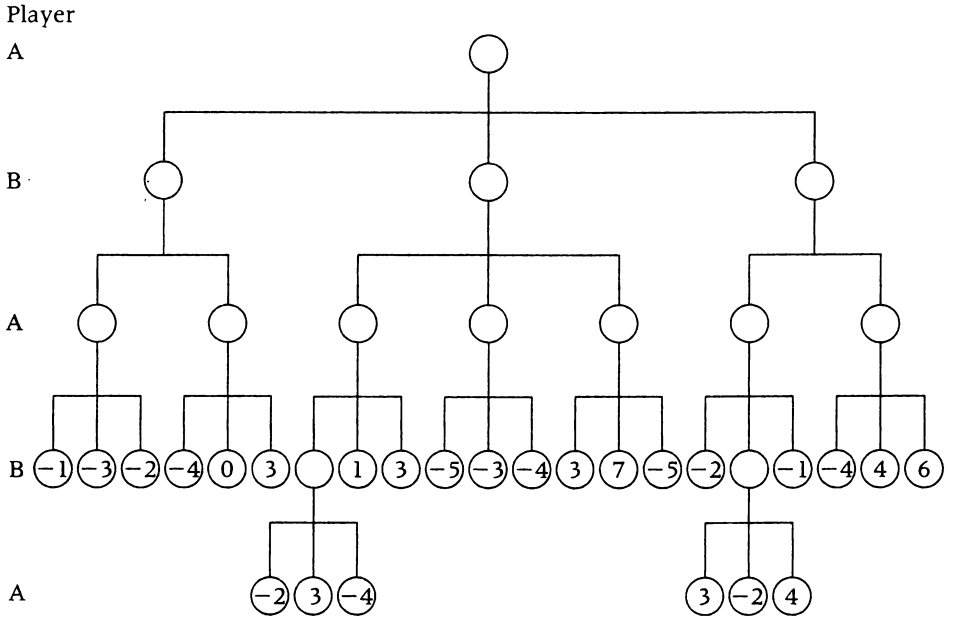
c. Part (b) shows that $c(v)$ is suitable for *LeastCostBranchAndBound*. Design and give pseudocode for *LeastCost BranchAndBound* implementing $c(v)$.

23.20 Draw the portion of the state-space tree T generated by the least-cost branch-and-bound discussed in the Exercise 23.19 for the cost matrix illustrated in that exercise. Label each node v with its cost value $c(v)$. Also, write out the reduced matrix associated with each node generated.

23.21 Discuss other state-space trees and associated cost functions $c(v)$ for the traveling salesman problem.

Section 23.5 Game Trees

23.22 Consider the two-person zero-sum game shown in the figure below. The values in the leaf nodes are values to player A. Use the minimax strategy (postorder traversal) to determine the value of the game to player A. Show clearly where alpha-cutoff and beta-cutoff occur, as well as (final) actual values, alpha values, and beta values of all nodes reached in the traversal.



23.23 Rewrite the recursive function *NodeValue* as a recursive procedure that has the same input parameters, *Y*, *NumLevels*, *ParentValue*, but now returns in output parameters the value V of Y and the child C_i, whose value is $-V$.

23.24 Find a sequence of admissible moves for the two-level heuristic search illustrated in Figure 23.16 that leads to a loss for player A in 3×3 tic-tac-toe.

23.25 Show that by assigning the values 9, 0, and -9 to terminal nodes that are wins, ties, or losses, respectively, for player A, the two-level search illustrated in Figure 23.16 never leads to a loss for player A.

23.26 Because there are no tie positions in the $3 \times 3 \times 3$ tic-tac-toe game, the first player has a winning strategy. Find a winning strategy for the first player.

23.27 Design a recursive function *DABNodeValue*(*X*, *NumLevels*, *ParentValue*, *NodeValueLowBnd*) for deep alpha-beta pruning. The initial invocation of *DABNodeValue* should have *ParentValue* $= \infty$ and *NodeValueLowBnd* $= -\infty$.

23.28 Redo Exercise 23.22 for deep alpha-beta pruning. Indicate any pruned nodes that were not pruned by alpha-beta pruning.