Unit-2

SOLVING PROBLEMS BY SEARCHING

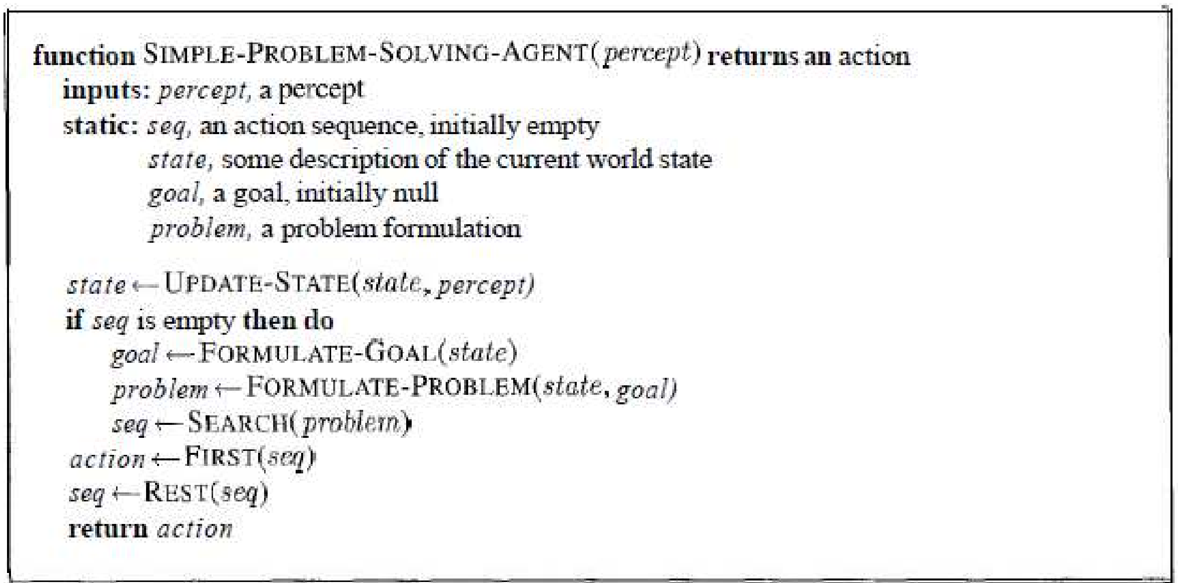
The simplest agents were the reflex agents, which base their actions on a direct mapping from states to actions. Such agents cannot operate well in environments for which this mapping would be too large to store and would take too long to learn.

Goal-based agents, on the other hand, can succeed by considering future actions and the desirability of their outcomes. *This kind of goal-based agent called a problem-solving agent. Problem-solving agents decide what to do by finding sequences of actions that lead to desirable states.*

Problem-Solving Agent

Intelligent agents are supposed to maximize their performance measure. In general, *an agent with several immediate options of unknown value can decide what to do by first examining different possible* sequences *of actions that lead to states of known value, and then choosing the best sequence.* This process of looking for such a sequence is called search.

A search algorithm takes a problem as input and returns a solution in the form of an action sequence. Once a solution is found, the actions it recommends can be carried out. This is called the execution phase. After formulating a goal and a problem to solve, the agent calls a search procedure to solve it. It then uses the solution to guide its actions, doing whatever the solution recommends as the next thing to do-typically, the first action of the sequence-and then removing that step from the sequence. Once the solution has been executed, the agent will formulate new goal.



```
function SIMPLE-PROBLEM-SOLVING-AGENT(percept) returns an action
   inputs: percept, a percept
   static: seq, an action sequence, initially empty
           state, some description of the current world state
           goal, a goal, initially null
           problem, a problem formulation

   state ← UPDATE-STATE(state, percept)
   if seq is empty then do
       goal ← FORMULATE-GOAL(state)
       problem ← FORMULATE-PROBLEM(state, goal)
       seq ← SEARCH(problem)
   action ← FIRST(seq)
   seq ← REST(seq)
   return action
```

Figure 3.1 A simple problem-solving agent. It first formulates a goal and a problem, searches for a sequence of actions that would solve the problem, and then executes the actions one at a time. When this is complete, it formulates another goal and starts over. Note that when it is executing the sequence it ignores its percepts: it assumes that the solution it has found will always work.

Well-defined problems and solutions

A problem can be defined formally by four components:

1) The **initial state** that the agent starts in. For example, the initial state for our agent in Romania might be described as In(Arad).

2) A description of the possible **actions** available to the agent. The most common formulation uses a **successor function.** For example, from the state In(Arad), the successor function for the Romaniaproblem would return { (*Go(Sibzu), In(Sibiu)), (Go(Timisoara), In(Tzmisoara)), (Go(Zerznd),In(Zerind)))* Together, the initial state and successor function implicitly define the **state space** of the problem-the set of all states reachable from the initial state. A **path** in the state space is a sequence of states connected by a sequence of actions.

3) The **goal test,** which determines whether a given state is a goal state. The agent's goal in Romania is the singleton set {In(Bucharest)).

4) A **path cost** function that assigns a numeric cost to each path. The problem-solving agent chooses a cost function that reflects its own performance measure.

A **solution** to a problem is a path from the initial state to a goal state. Solution quality is measured by the path cost function, and an **optimal solution** has the lowest path cost among all solutions.

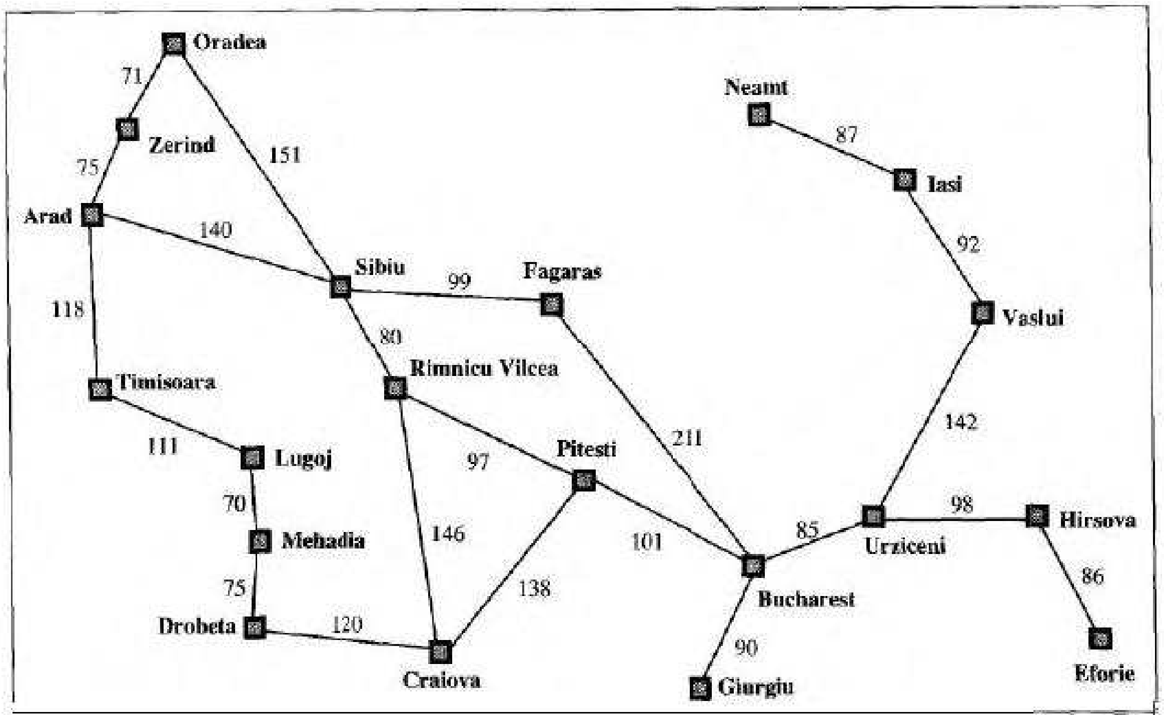


Figure 3.2 A simplified road map of part of Romania.

Example Problem

Toy problems

The first example we will examine is the vacuum world. This can be formulated as a problem as follows:

- ✓ **States:** The agent is in one of two locations, each of which might or might not contain dirt. Thus there are 2 x *22* = 8 possible world states.
- ✓ **Initial state:** Any state can be designated as the initial state.

- ✓ **Successor function:** This generates the legal states that result from trying the three actions (Left, ***Right,*** and ***Suck).*** The complete state space is shown in Figure 3.3.
- ✓ **Goal test:** This checks whether all the squares are clean.
- ✓ **Path cost:** Each step costs 1, so the path cost is the number of steps in the path.

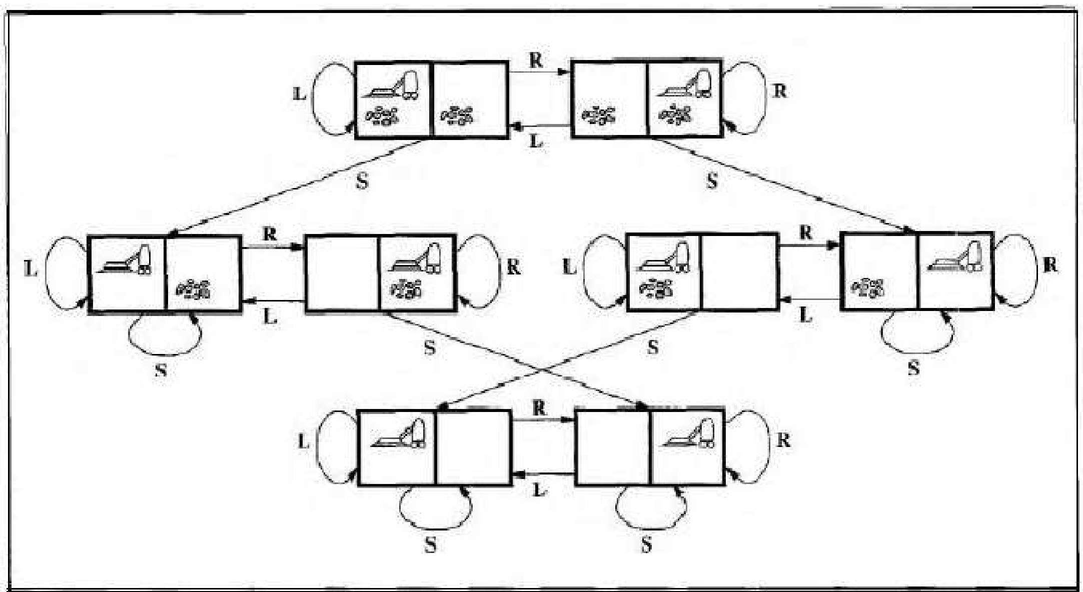


Figure 3.3 The state space for the vacuum world. Arcs denote actions: L = *Left*, R = *Right*, S = Suck.

The 8-puzzle Problem

The 8-puzzle consists of a 3 x 3 board with eight numbered tiles and a blank space. A tile adjacent to the blank space can slide into the space. The object is to reach a specified goal state, such as the one shown on the right of the figure. The standard formulation is as follows:

- ✓ **States:** A state description specifies the location of each of the eight tiles and the blank in one of the nine squares.
- ✓ **Initial state**: Any state can be designated as the initial state. Note that any given goal can be reached from exactly half of the possible initial states.
- ✓ **Successor function:** This generates the legal states that result from trying the four actions (blank moves Left, Right, Up, or Down).
- ✓ **Goal test:** This checks whether the state matches the goal configuration.
- ✓ **Path cost:** Each step costs 1, so the path cost is the number of steps in the path.

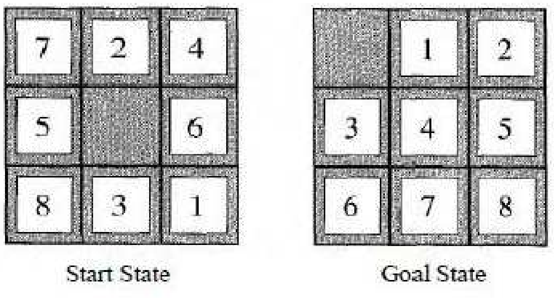


Figure 3.4 A typical instance of the 8-puzzle.

The 8-queens problem

The goal of the **8-queens problem** is to place eight queens on a chessboard such that no queen attacks any other. (A queen attacks any piece in the same row, column or diagonal.) There are two main kinds of formulation. An **incremental formulation** involves operators that *augment* the state description, starting with an empty state; for the 8-queens problem, this means that each action adds a queen to the state. A **complete-state formulation** starts with all 8 queens on the board and moves them around. In either case, the path cost is of no interest because only the final state counts. The first incremental formulation one might try is the following:

- ✓ **States:** Any arrangement of 0 to 8 queens on the board is a state.
- ✓ **Initial state:** No queens on the board.
- ✓ **Successor function:** Add a queen to any empty square.
- ✓ **Goal test: 8** queens are on the board, none attacked.

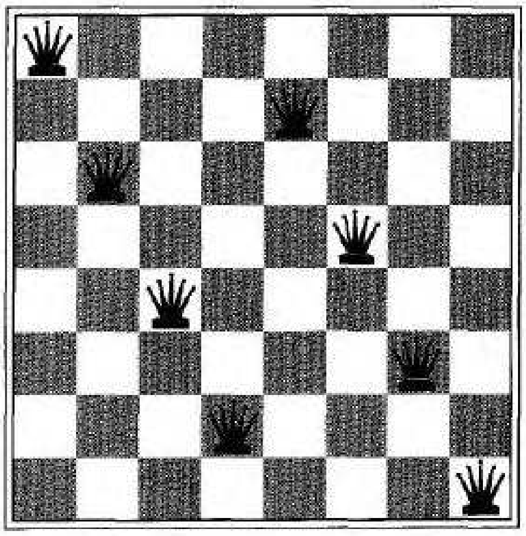


Figure 3.5 Almost a solution to the 8-queens problem. (Solution is left as an exercise.)

Real-world problems

Consider a simplified example of an airline travel problem specified as follows:

- ✓ **States:** Each is represented by a location (e.g., an airport) and the current time.
- ✓ **Initial state:** This is specified by the problem.
- ✓ **Successor function:** This returns the states resulting from taking any scheduled flight (perhaps further specified by seat class and location), leaving later than the current time plus the within-airport transit time, from the current airport to another.
- ✓ **Goal test:** Are we at the destination by some pre specified time?
- ✓ **Path cost:** This depends on monetary cost, waiting time, flight time, customs and immigration procedures, seat quality, time of day, type of airplane, frequent-flyer mileage awards, and so on.

The travelling salesperson problem (TSP)

The **travelling salesperson problem** (TSP) is a touring problem in which each city must be visited exactly once. The aim is to find the *shortest* tour. The problem is known to be NP hard, but an enormous amount of effort has been expended to improve the capabilities of TSP algorithms.

A VLSI layout problem

A VLSI layout problem requires positioning millions of components and connections on a chip to minimize area, minimize circuit delays, minimize stray capacitances, and maximize manufacturing yield. The layout problem comes after the logical design phase, and is usually split into two parts: cell layout and channel routing. In cell layout, the primitive components of the circuit are grouped into cells, each of which performs some recognized function. Each cell has a fixed footprint (size and shape) and requires a certain number of connections to each of the other cells. The aim is to place the cells on the chip so that they do not overlap and so that there is room for the connecting wires to be placed between the cells. Channel routing finds a specific route for each wire through the gaps between the cells. These search problems are extremely complex, but definitely worth solving.

Robot navigation is a generalization of the route-finding problem described earlier. Rather than a discrete set of routes, a robot can move in a continuous space with (in principle) an infinite set of possible actions and states.

SEARCHING FOR SOLUTIONS

The root of the search tree is a **search node** corresponding to the initial state, *In(Arad).* The first step is to test whether this is a goal state. Clearly it is not, but it is important to check so that we can solve trick problems like "starting in Arad, get to Arad." Because this is not a goal state, we need to consider some other states. This is done by **expanding** the current state; that is, applying the successor function to the current state, thereby **generating** a new set of states.
In this case, we get three new states: *In(Sibiu), In(Timisoara),* and *In(Zerind).* Now we must choose which of these three possibilities to consider further Suppose we choose Sibiu first. We check to see whether it is a goal state (it is not) and then expand it to get *In(Arad), In(Fagaras), In(Oradea),* and *In(RimnicuVi1cea).* We can then choose any of these four, or go back and choose Timisoara or Zerind. We continue choosing, testing, and expanding until either a solution is found or there are no more states to be expanded. The choice of which state to expand is determined by the **search strategy.**

There are many ways to represent nodes, but we will assume that a node is a data structure with five components:
- ✓ **STATE:** the state in the state space to which the node corresponds;
- ✓ **PARENT-NODE:** the node in the search tree that generated this node;
- ✓ **ACTION:** the action that was applied to the parent to generate the node;
- ✓ **PATH-COST:** the cost, traditionally denoted by *g(n),* of the path from the initial state to the node, as indicated by the parent pointers; and
- ✓ **DEPTH:** the number of steps along the path from the initial state.

A node is a bookkeeping data structure used to represent the search tree. A state corresponds to a configuration of the world. Thus, nodes are on particular paths, as defined by PARENT-NODE pointers, whereas states are not. Furthermore, two different nodes can contain the same world state, if that state is generated via two different search paths. We also need to represent the **collection of nodes that have been generated but not yet expandedthis collection is called the fringe or frontier.** Each element of the fringe is a **leaf node,** that is, a node with no successors in the tree.

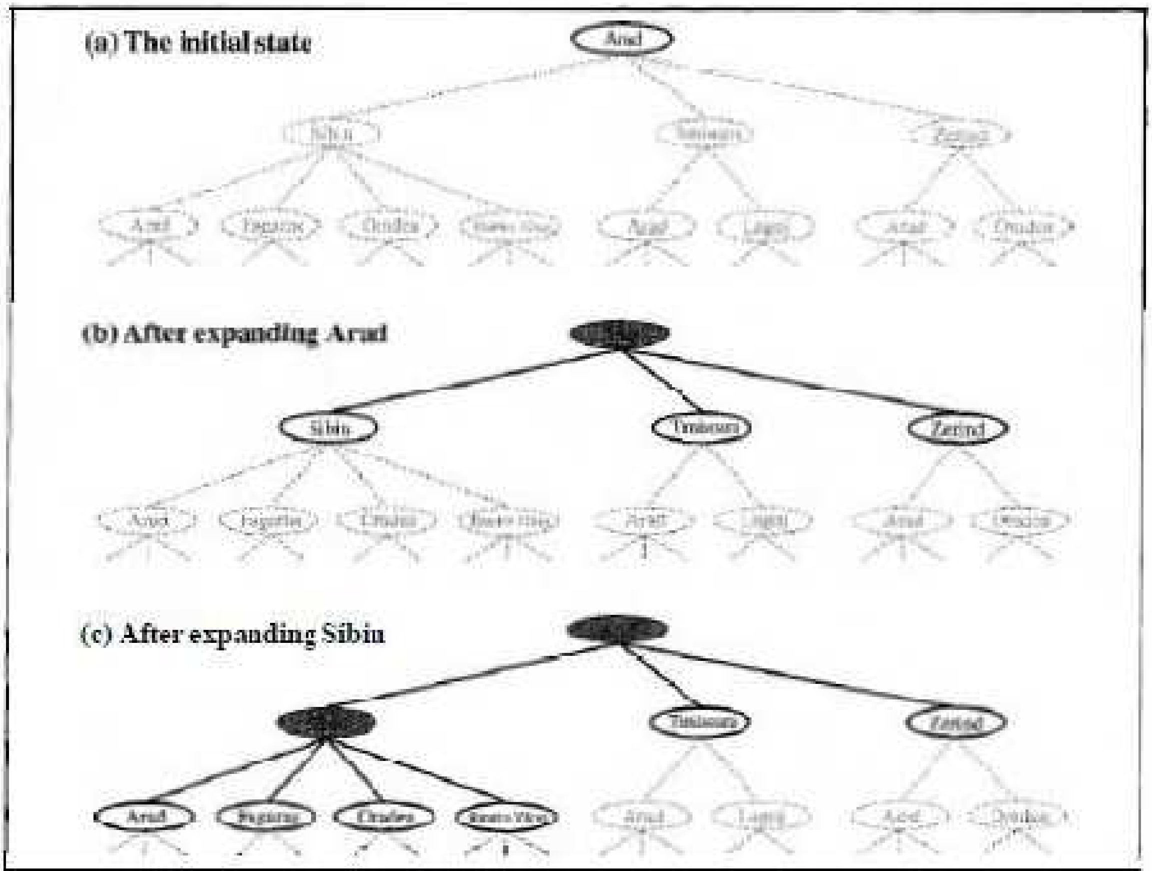
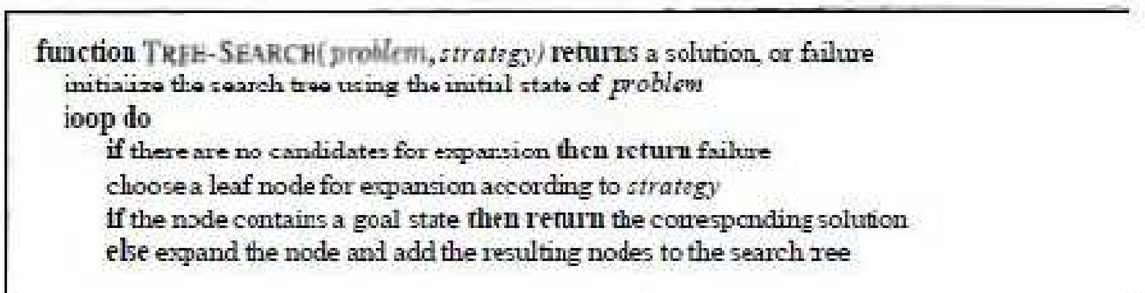


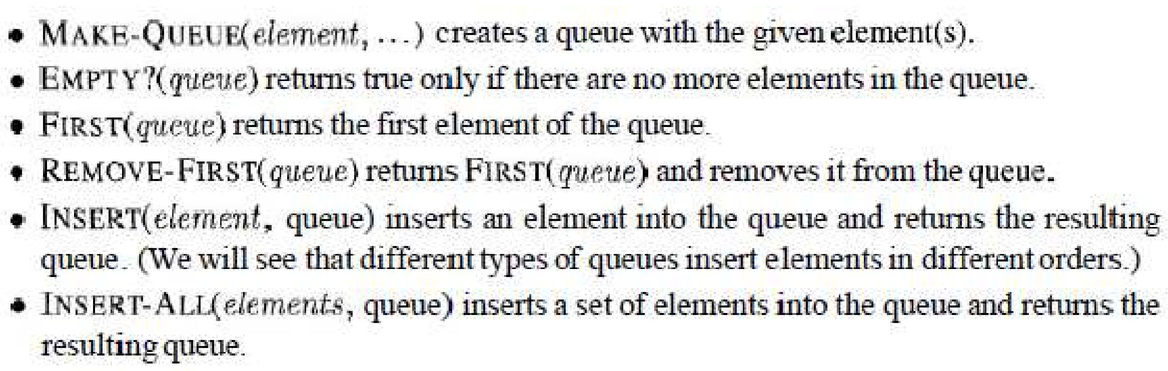
Figure 3.6 Partial search trees for finding a route from Arad to Bucharest. Nodes that have been expanded are shaded; nodes that have been generated but not yet expanded are outlined in bold; nodes that have not yet been generated are shown in faint dashed lines.



```
function TREE-SEARCH(problem, strategy) returns a solution, or failure
  initialize the search tree using the initial state of problem
  loop do
     if there are no candidates for expansion then return failure
     choose a leaf node for expansion according to strategy
     if the node contains a goal state then return the corresponding solution
     else expand the node and add the resulting nodes to the search tree
```

Figure 3.7 An informal description of the general tree-search algorithm.

We will assume that the collection of nodes is implemented as a **queue.** The operations on a queue are as follows:



- MAKE-QUEUE(*element*, ...) creates a queue with the given element(s).
- EMPTY?(*queue*) returns true only if there are no more elements in the queue.
- FIRST(*queue*) returns the first element of the queue.
- REMOVE-FIRST(*queue*) returns FIRST(*queue*) and removes it from the queue.
- INSERT(*element*, queue) inserts an element into the queue and returns the resulting queue. (We will see that different types of queues insert elements in different orders.)
- INSERT-ALL(*elements*, queue) inserts a set of elements into the queue and returns the resulting queue.

Measuring problem-solving performance

The output of a problem-solving algorithm is either *failure* or a solution. We will evaluate an algorithm's performance in four ways:

- ✓ **Completeness:** Is the algorithm guaranteed to find a solution when there is one?
- ✓ **Optimality:** Does the strategy find the optimal solution?
- ✓ **Time complexity:** How long does it take to find a solution?
- ✓ **Space complexity:** How much memory is needed to perform the search?

Time and space complexity are measured in terms of

- ✓ b-maximum branching factor of the search tree
- ✓ d-depth of the least-cost solution
- ✓ m-maximum depth of the state space

UNINFORMED SEARCH STRATEGIES

An *uninformed* (a.k.a. *blind*, *brute-force*) search algorithm generates the search tree without using any domain specific knowledge. All they can do is generate successors and distinguish a goal state from a non-goal state. All search strategies are distinguished by the *order* in which nodes are expanded.

Breadth-first search

Breadth-first search is a simple strategy in which the root node is expanded first, then all the successors of the root node are expanded next, then their successors, and so on. In general, all the nodes are expanded at a given depth in the search tree before any nodes at the next level are expanded.

Breadth-first search can be implemented by calling TREE-SEARCH with an empty fringe that is a first-in-first-out (FIFO) queue, assuring that the nodes that are visited first will be expanded first. The FIFO queue puts all newly generated successors at the end of the queue, which means that shallow nodes are expanded before deeper nodes. First lesson is that, *the memory requirements are a bigger problem for breadth9rst search than is the execution time.* The second lesson is that the time requirements are still a major factor. In general, *exponential-complexity search problems cannot be solved by uninformed methods for any but the smallest instances.*

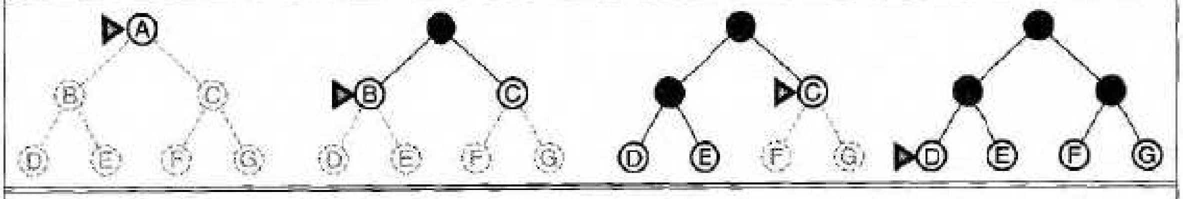


Figure 3.10 Breadth-first search on a simple binary tree. At each stage, the node to be expanded next is indicated by a marker.

Depth	Nodes	Time	Memory
2	1100	.11 seconds	1 megabyte
4	111,100	11 seconds	106 megabytes
6	10^7	19 minutes	10 gigabytes
8	10^9	31 hours	1 terabytes
10	10^{11}	129 days	101 terabytes
12	10^{13}	35 years	10 petabytes
14	10^{15}	3,523 years	1 exabyte

Figure 3.11 Time and memory requirements for breadth-first search. The numbers shown assume branching factor b = 10; 10,000 nodes/second; 1000 bytes/node.

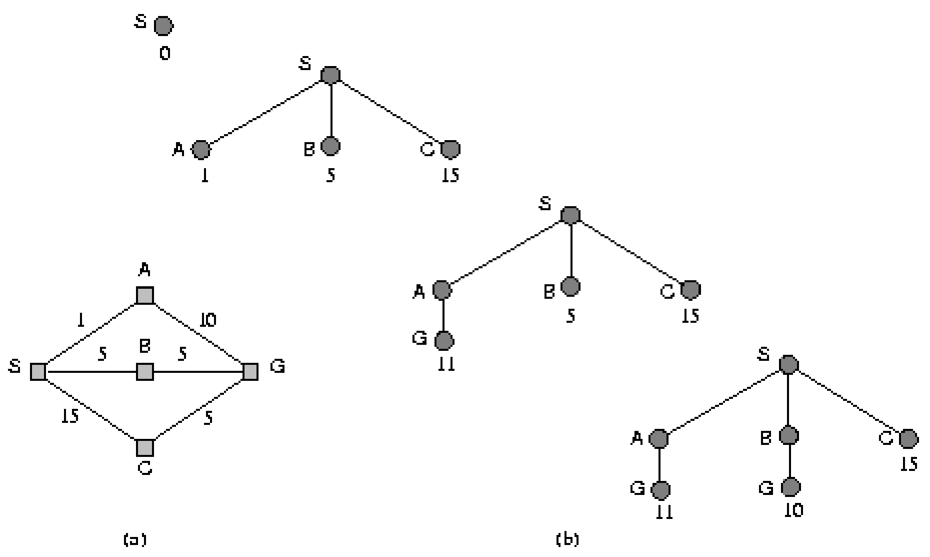
Properties of breadth-First search

- *Complete:* Yes (if b is finite)
- *Time:* $b + b^2 + b^3 + \ldots + b^d + b(b^d - 1) = O(b^{d+1})$, i.e., number of nodes generated is exponential in d
- *Space:* $O(b^{d+1})$ (keeps every node in memory)
- *Optimal:* Yes (if cost = 1 per step)

Space is the big problem; can easily generate nodes at 100MB/sec so 24hrs = 8604GB.

Uniform-cost search

Breadth-first search is optimal when all step costs are equal, because it always expands the *shallowest* unexpanded node. By a simple extension, we can find an algorithm that is optimal with any step cost function. Instead of expanding the shallowest node, **uniform-cost search** expands the node n with the *lowest path cost.* Note that if all step costs are equal, this is identical to breadth-first search.
Uniform-cost search does not care about the *number* of steps a path has, but only about their total cost. Therefore, it will get stuck in an infinite loop if it ever expands a node that has a zero-cost action leading back to the same state.



- *Complete:* Yes, if step cost $\geq \epsilon$
- *Time:* # of nodes with $g \leq$ cost of optimal solution, $O(b^{1+\lfloor C^*/\epsilon \rfloor})$
  where C^* is the cost of the optimal solution
- *Space:* # of nodes with $g \leq$ cost of optimal solution, $O(b^{1+\lfloor C^*/\epsilon \rfloor})$
- *Optimal:* Yes—nodes expanded in increasing order of $g(n)$

Depth-first search

Depth-first search *always expands the deepest node in the current fringe of the search tree*. The search proceeds immediately to the deepest level of the search tree, where the nodes have no successors. As those nodes are expanded, they are dropped from the fringe, so then the search "backs up" to the next shallowest node that still has unexplored successors.

This strategy can be implemented by TREE-SEARCH with a last-in-first-out (LIFO) queue, also known as a stack. As an alternative to the TREE-SEARCH implementation, it is common to implement depth-first search with a recursive function that calls itself on each of its children in turn.

Depth-first search has very modest memory requirements. It needs to store only a single path from the root to a leaf node, along with the remaining unexpanded sibling nodes for each node on the path. Once a node has been expanded, it can be removed from memory as soon as all its descendants have been fully explored.

The drawback of depth-first search is that it can make a wrong choice and get stuck going down a very long (or even infinite) path when a different choice would lead to a solution near the root of the search tree.

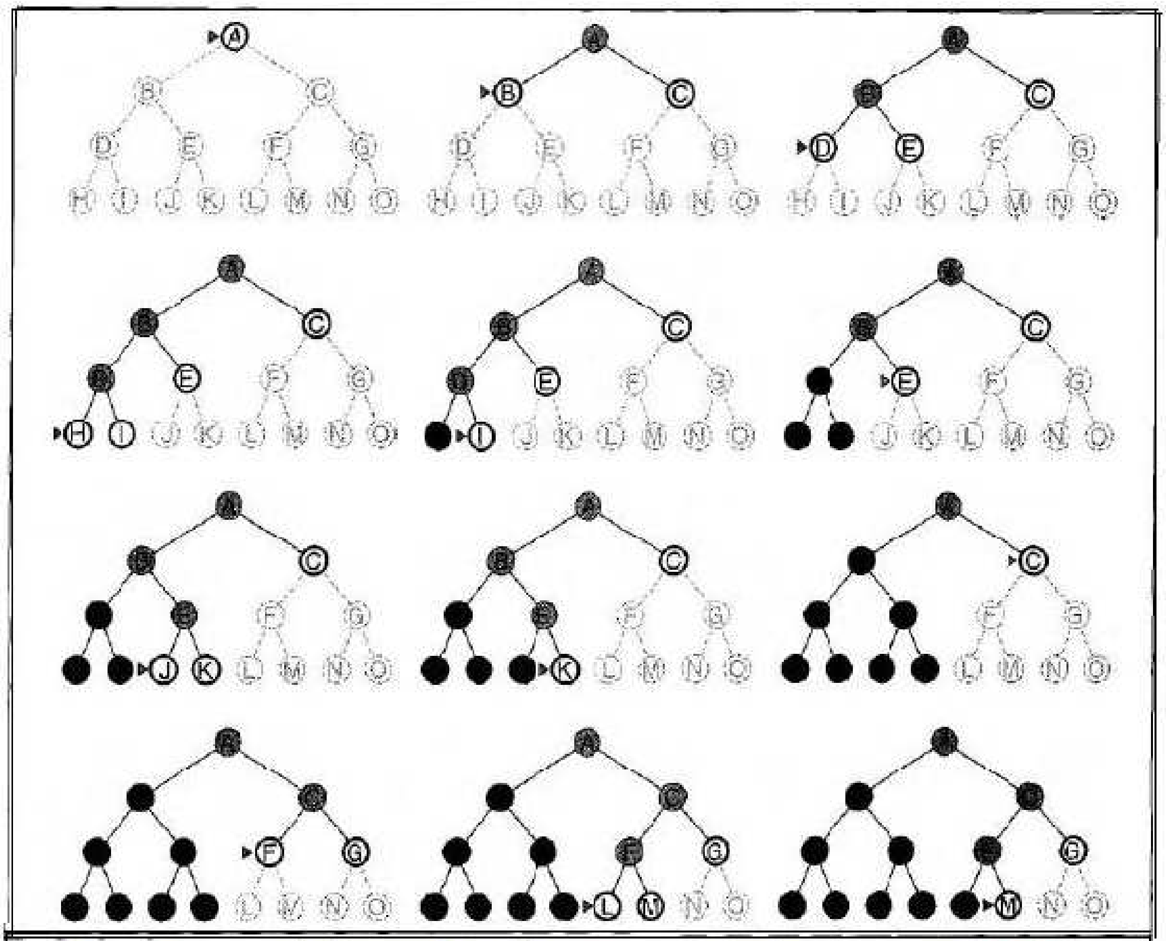
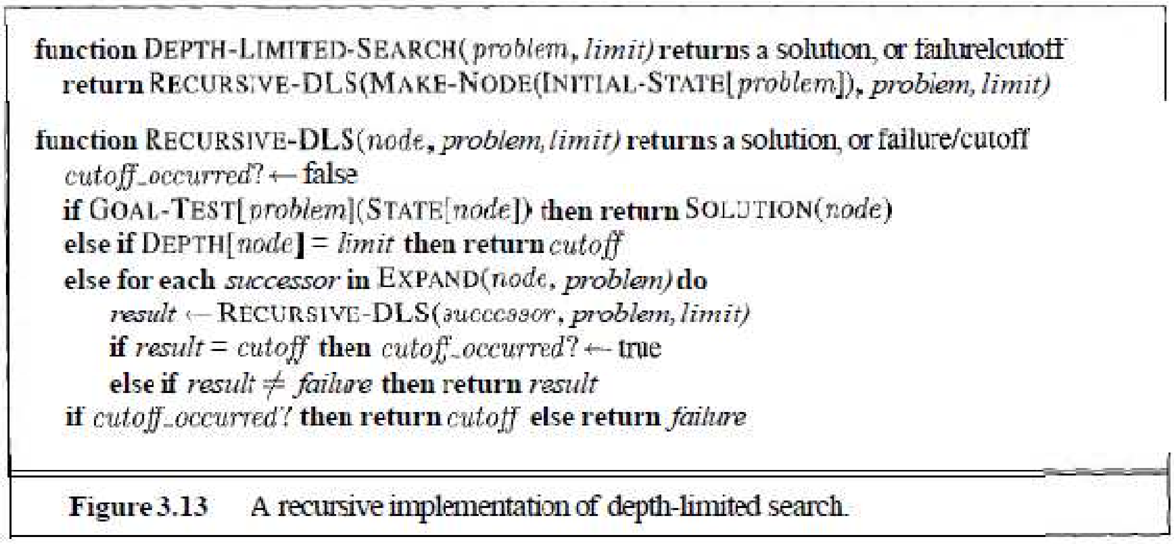


Figure 3.12 Depth-first search on a binary tree. Nodes that have been expanded and have no descendants in the fringe can be removed from memory; these are shown in black. Nodes at depth 3 are assumed to have no successors and M is the only goal node.

- *Complete:* No: fails in infinite-depth spaces, spaces with loops
  Modify to avoid repeated states along path
  $\Rightarrow$ complete in finite spaces
- *Time:* $O(b^m)$: terrible if m is much larger than d
  but if solutions are dense, may be much faster than breadth-first
- *Space:* $O(bm)$, i.e., linear space!
- *Optimal:* No

Depth-limited search

The problem of unbounded trees can be alleviated by supplying depth-first search with a predetermined depth limit *l*. That is, nodes at depth *l* are treated as if they have no successors. This approach is called **depth-limited search.** The depth limit solves the infinite-path problem.



```
function DEPTH-LIMITED-SEARCH(problem, limit) returns a solution, or failure/cutoff
  return RECURSIVE-DLS(MAKE-NODE(INITIAL-STATE[problem]), problem, limit)

function RECURSIVE-DLS(node, problem, limit) returns a solution, or failure/cutoff
  cutoff_occurred? ← false
  if GOAL-TEST[problem](STATE[node]) then return SOLUTION(node)
  else if DEPTH[node] = limit then return cutoff
  else for each successor in EXPAND(node, problem) do
      result ← RECURSIVE-DLS(successor, problem, limit)
      if result = cutoff then cutoff_occurred? ← true
      else if result ≠ failure then return result
  if cutoff_occurred? then return cutoff else return failure
```

Figure 3.13 A recursive implementation of depth-limited search.

Iterative deepening depth-first search

Iterative deepening search (or iterative deepening depth-first search) is a general strategy, often used in *combination with depth-first search that finds the best depth limit. It does this by gradually increasing the limit-first 0, then 1, then 2, and so on-until a goal is found. This will occur when the depth limit reaches d, the depth of the shallowest goal node*.

Iterative deepening combines the benefits of depth-first and breadth-first search. Like depthfirst

search, its *memory requirements are very modest*: O(bd) to be precise. Like breadth-first search, *it is complete when the branching factor is finite* and *optimal when the path cost is a non decreasing function of the depth of the node*. Iterative deepening search may seem wasteful, because states are *generated multiple times*.

In an iterative deepening search, the nodes on the bottom level (depth d) are generated once, those on the next to bottom level are generated twice, and so on, up to the children of the root, which are generated d times. So the total number of nodes generated is

$$N(\text{IDS}) = (d)b + (d-1)b^2 + \ldots + (1)b^d \,,$$

which gives a time complexity of $O(b^d)$. We can compare this to the nodes generated by a breadth-first search:

$$N(\text{BFS}) = b + b^2 + \ldots + b^d + (b^{d+1} - b) \,.$$

Notice that breadth-first search generates some nodes at depth $d+1$, whereas iterative deepening does not. The result is that iterative deepening is actually *faster* than breadth-first search, despite the repeated generation of states. For example, if $b = 10$ and $d = 5$, the numbers are

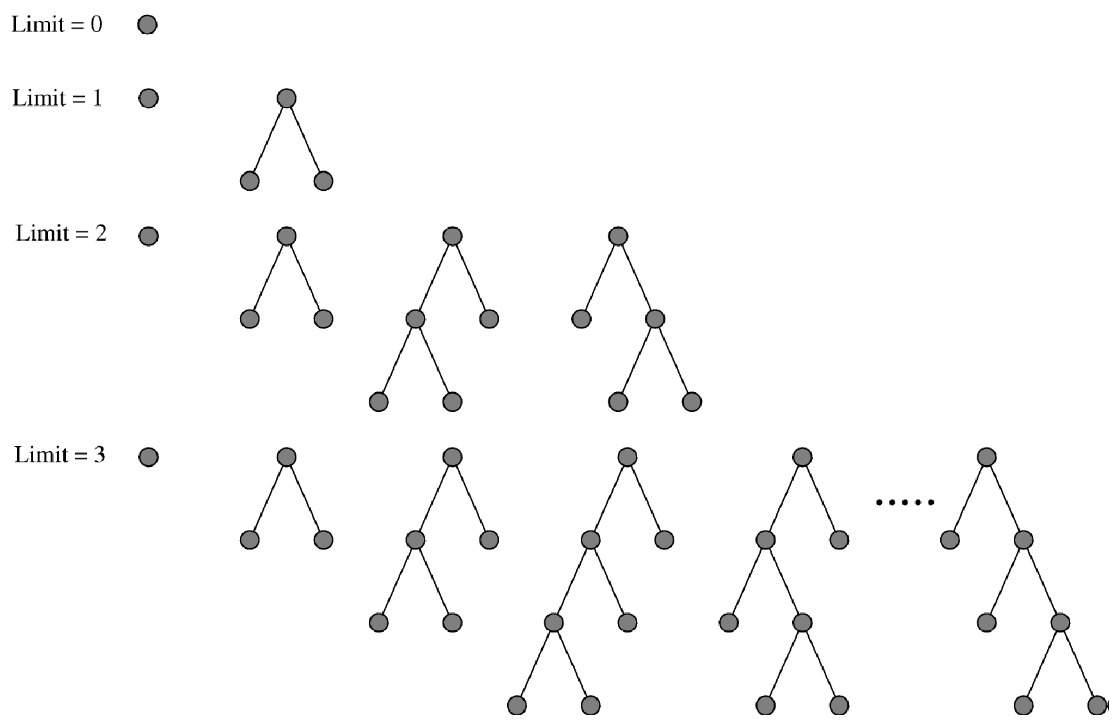
$$N(\text{IDS}) = 50 + 400 + 3,000 + 20,000 + 100,000 = 123,450$$
$$N(\text{BFS}) = 10 + 100 + 1,000 + 10,000 + 100,000 + 999,990 = 1,111,100$$

In general, iterative deepening is the preferred uninformed search method when there is a large search space and the depth of the solution is not known.

```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution, or failure
   inputs: problem, a problem

   for depth ← 0 to ∞ do
      result ← DEPTH-LIMITED-SEARCH(problem, depth)
      if result ≠ cutoff then return result
```

Figure 3.14 The iterative deepening search algorithm, which repeatedly applies depth-limited search with increasing limits. It terminates when a solution is found or if the depth-limited search returns *failure*, meaning that no solution exists.



Bidirectional search

The idea behind bidirectional search is to run two simultaneous searches-one forward from the initial state and the other backward from the goal, stopping when the two searches meeting the middle.

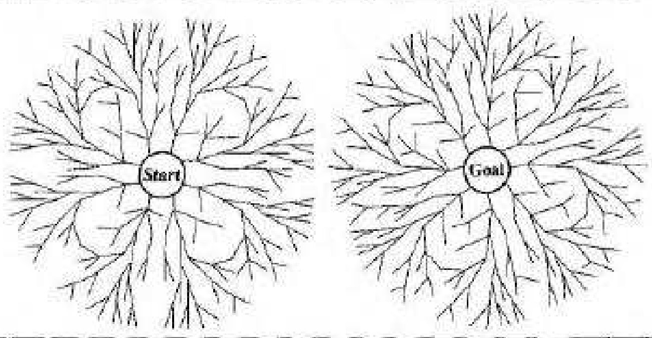
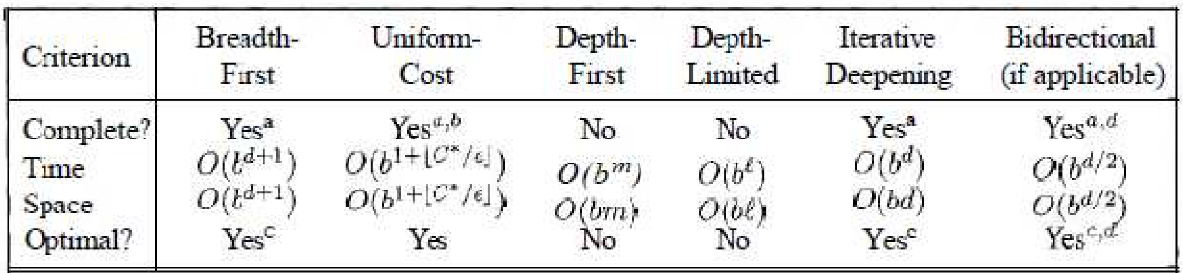


Figure 3.16 A schematic view of a bidirectional search that is about to succeed, when a branch from the start node meets a branch from the goal node.

Comparing uninformed search strategies

Figure 3.17 compares search strategies in terms of the four evaluation criteria set forth in Section 3.4.



Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Complete?	Yes[a]	Yes[a,b]	No	No	Yes[a]	Yes[a,d]
Time	$O(b^{d+1})$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^{d+1})$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(bm)$	$O(b\ell)$	$O(bd)$	$O(b^{d/2})$
Optimal?	Yes[c]	Yes	No	No	Yes[c]	Yes[c,d]

Figure 3.17 Evaluation of search strategies. b is the branching factor; d is the depth of the shallowest solution; m is the maximum depth of the search tree; l is the depth limit. Superscript caveats are as follows: [a] complete if b is finite; [b] complete if step costs $\geq \epsilon$ for positive ϵ; [c] optimal if step costs are all identical; [d] if both directions use breadth-first search.

INFORMED (HEURISTIC) SEARCH STRATEGIES

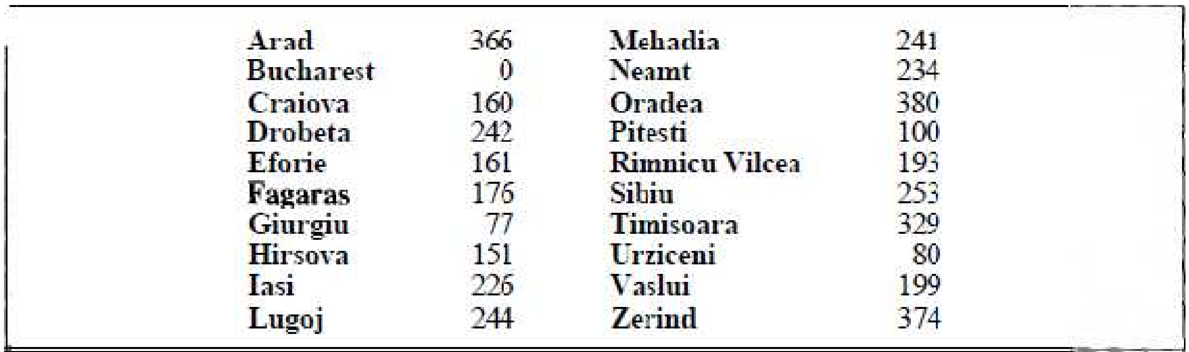
An **informed search** strategy one that uses problem-specific knowledge beyond the definition of the problem itself can find solutions more efficiently than an uninformed strategy. **A** key component of these algorithms is a heuristic function denoted by h (n):
h (n) = estimated cost of the cheapest path from node *n* to a goal node.

Greedy best-first search

Greedy best-first search tries to expand the node that is closest to the goal, on the: grounds that this is likely to lead to a solution quickly. Thus, it evaluates nodes by using just the heuristic function: f *(n)* = *h(n).*

Let us see how this works for route-finding problems in Romania, using the straight line distance heuristic, which we will call h_{SLD}. If the goal is Bucharest, we will need to know the straight-line distances to Bucharest,

It suffers from the same defects as depth-first search-it is not optimal, and it is incomplete (because it can start down an infinite path and never return to try other possibilities). The worst-case time and space complexity is O(bm), where m is the maximum depth of the search space. With a good heuristic function, however, the complexity can be reduced substantially.



Arad	366	Mehadia	241
Bucharest	0	Neamt	234
Craiova	160	Oradea	380
Drobeta	242	Pitesti	100
Eforie	161	Rimnicu Vilcea	193
Fagaras	176	Sibiu	253
Giurgiu	77	Timisoara	329
Hirsova	151	Urziceni	80
Iasi	226	Vaslui	199
Lugoj	244	Zerind	374

Figure 4.1 Values of h_{SLD}—straight-line distances to Bucharest.

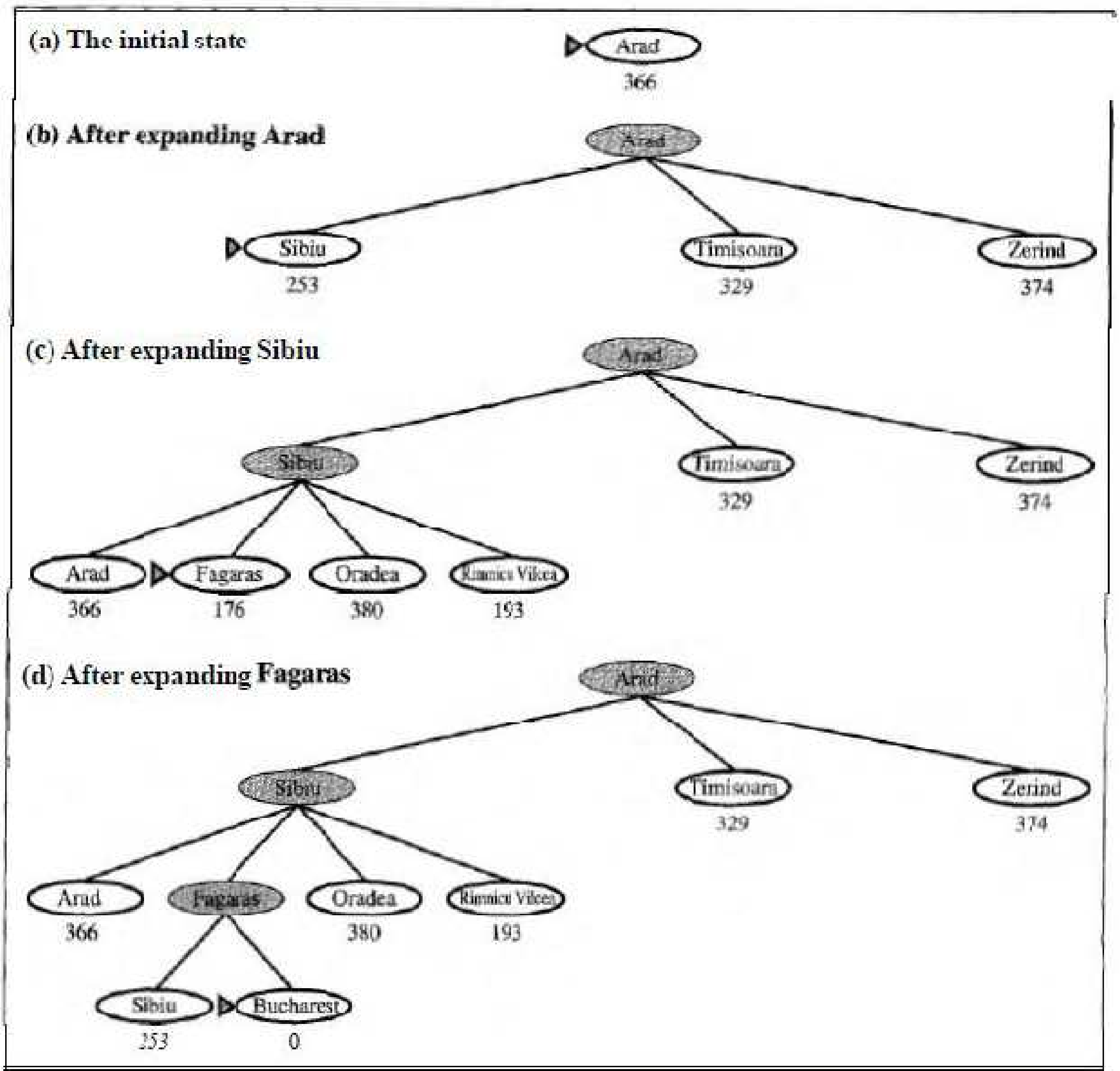


Figure 4.2 Stages in a greedy best-first search for Bucharest using the straight-line distance heuristic h_{SLD}. Nodes are labeled with their h-values.

A* search

The most widely-known form of best-first search is called **A*** search (pronounced "A-star Search"). It evaluates nodes by combining *g (n),* the cost to reach the node, and *h (n.),* the cost to get from the node to the goal:
f (n)=g (n) + h (n)
g (n) = the path cost from the start node to node *n,*
h (n) = the estimated cost of the cheapest path from n to the goal,
f (n) = the estimated cost of the cheapest solution through n.
A* search is both **complete and optimal.** In this case, **A* is optimal if *h(n)* is an admissible heuristic-that is, provided that *h(n) never overestimates* the cost to reach the goal**. Admissible heuristics are by nature optimistic, because they think the cost of solving the problem is less than it actually is. Since *g(n)* is the exact cost to reach *n,* we have as immediate consequence that *f (n)* never overestimates the true cost of a solution through n.

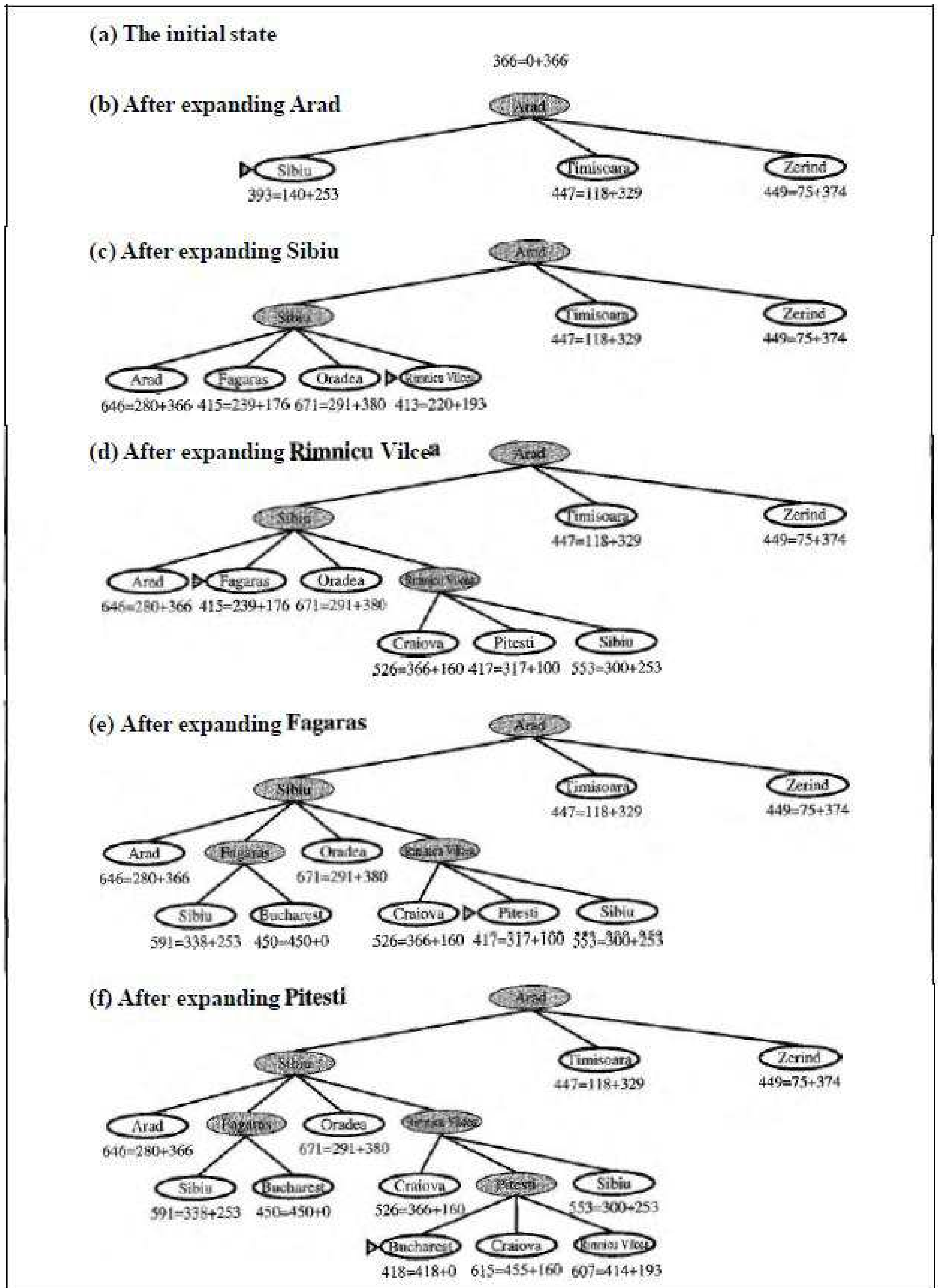


Figure 4.3 Stages in an A* search for Bucharest. Nodes are labeled with f = g + h. The h values are the straight-line distances to Bucharest taken from Figure 4.1.

Suppose a suboptimal goal node *G2* appears on the fringe, and let the cost of the optimal solution be C*. Then, because *G2 i*s suboptimal and because *h (G2=)* 0 (true for any goal node), we know

$$f(G2) = g(G2) + h(G2) = g(G2) > C^*$$

If h *(n)* does not overestimate the cost of completing the solution path, then we know that

$$f(n) = g(n) + h(n) <= C^* .$$

Now we have shown that f *(n)* <=C* < f *(G2)* so *G2* will not be expanded and A* must return an optimal solution.

A heuristic *h (n)* is consistent if, for every node *n* and every successor *n'* of *n* generated by any action a, the estimated cost of reaching the goal from *n* is no greater than the step cost of getting to *n'* plus the estimated cost of reaching the goal from *n':*

$$h(n) <= c(n, a, n') + h(n).$$

Another important consequence of consistency is the following: if *h (n) is consistent, then the values* off *(n) along any path are no decreasing.* The proof follows directly from the definition of consistency. Suppose *n'* is a successor of *n;* then *g(n')= g(n)+ c(n,a , n')* for some *a,* and we have

$$f(n') = g(n') + h(n') = g(n) + c(n,a,n') + h(n') \geq g(n) + h(n) = f(n).$$

It follows that the sequence of nodes expanded by A* using GRAPH-SEARCH is in non decreasing order of f (n). Hence, the first goal node selected for expansion must be an optimal Solution, since all later nodes will be at least as expensive.

LOCAL SEARCH ALGORITHMS AND OPTIMIZATION PROBLEMS

Local search algorithms operate using a single **current node** (rather than multiple paths) and generally move only to neighbours of that node. Typically, the paths followed by the searc are not retained. Although local search algorithms are not systematic, they have two key advantages: (1) they use very little memory—usually a constant amount; and (2) they can often find reasonable solutions in large or infinite (continuous) state spaces for which systematic algorithms are unsuitable. In addition to finding goals, local search algorithms are useful for solving pure **optimization problems,** in which the aim is to find the best state according to an **objective function.**

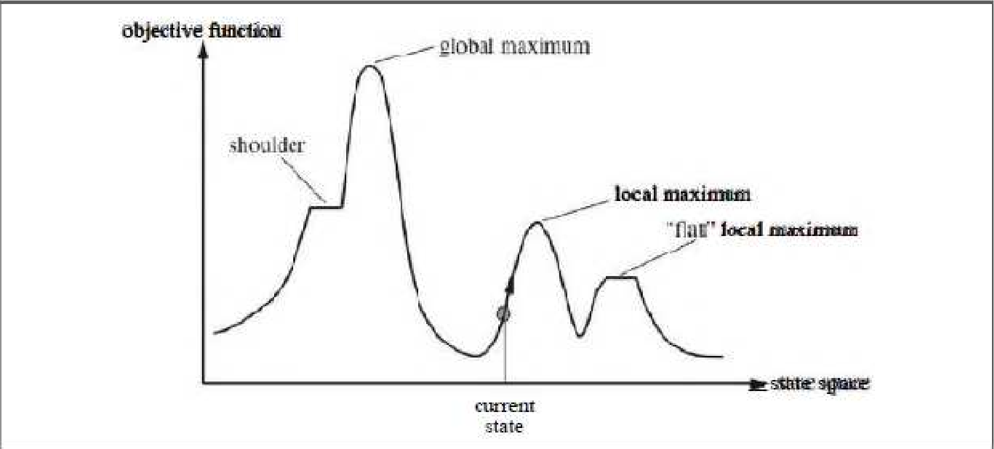


Figure 4.1 A one-dimensional state-space landscape in which elevation corresponds to the objective function. The aim is to find the global maximum. Hill-climbing search modifies the current state to try to improve it, as shown by the arrow. The various topographic features are defined in the text.

A landscape has both "location" (defined by the state) and "elevation" (defined by the value of the heuristic cost function or objective function). If elevation corresponds to cost, then the aim is to find the lowest valley—a **global minimum;** if elevation corresponds to an objective function, then the aim is to find the highest peak—a **global maximum.** Local search algorithms explore this landscape. A **complete** local search algorithm always finds a goal if one exists; **an optimal** algorithm always finds a global minimum/maximum.

Hill-climbing search

It is simply a loop that continually moves in the direction of increasing value—that is, uphill. It terminates when it reaches a "peak" where no neighbour has a higher value. The algorithm does not maintain a search tree, so the data structure for the current node need only record the state and the value of the objective function. Hill climbing does not look ahead beyond the immediate neighbours of the current state.

```
function HILL-CLIMBING(problem) returns a state that is a local maximum

  current 4— MAKE-NODE(problem.INITIAL-STATE)
  loop do
      neighbor     a highest-valued successor of current
      if neighbor. VALUE <= current.VALUE then return current.STATE
      current ← neighbor
```

Figure 4.2 The hill climbing search algorithm, which is the most basic local search tech nique. At each step the current node is replaced by the best neighbor; in this version, that means the neighbor with the highest VALUE, but if a heuristic cost estimate *h is* used, we would find the neighbor with the lowest h.

Unfortunately, hill climbing often gets stuck for the following reasons:

- **Local maxima:** a local maximum is a peak that is higher than each of its neighbouring states but lower than the global maximum. Hill-climbing algorithms that reach the vicinity of a local maximum will be drawn upward toward the peak but will then be stuck with nowhere else to go.
- **Ridges:** Ridges result in a sequence of local maxima that is very difficult for greedy algorithms to navigate
- **Plateaux:** a plateau is a flat area of the state-space landscape. It can be a flat local maximum, from which no uphill exit exists, or a shoulder, from which progress is possible.

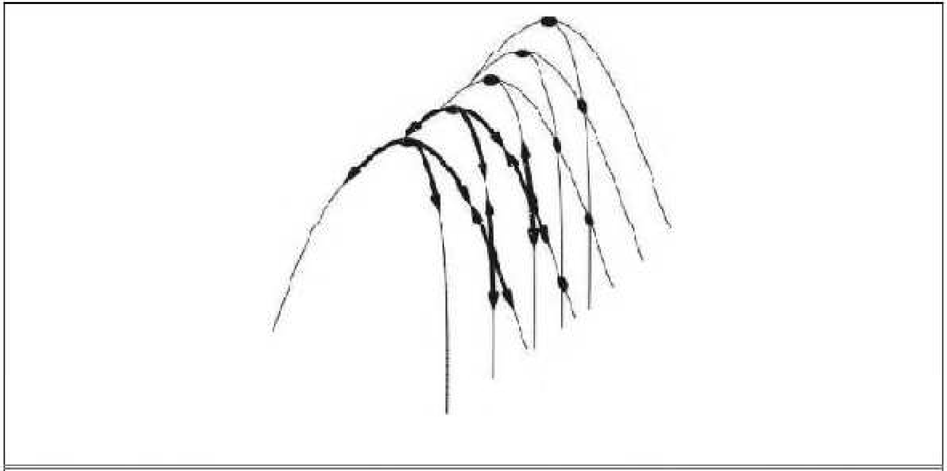
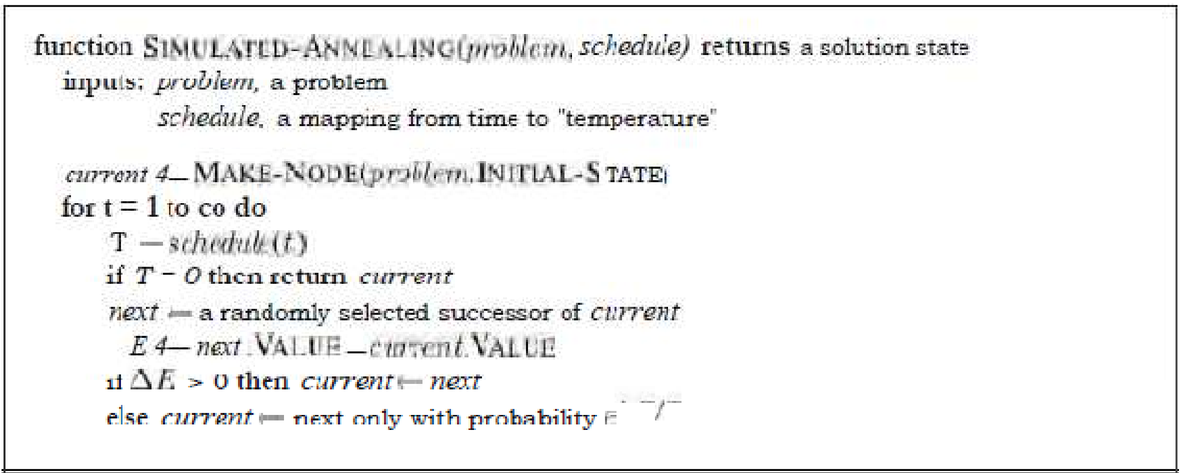


Figure 4.4 Illustration of why ridges cause difficulties for hill climbing. The grid of states (dark circles) is superimposed on a ridge rising from left to right, creating a sequence of local maxima that are not directly connected to each other. From each local maximum, all the available actions point downhill.

Simulated annealing

A hill-climbing algorithm that *never* makes "downhill" moves toward states with lower value (or higher cost) is guaranteed to be incomplete, because it can get stuck on a local maxi-mum. In contrast, a purely random walk—that is, moving to a successor chosen uniformly at random from the set of successors—is complete but extremely inefficient. Therefore, it seems reasonable to try to combine hill climbing with a random walk in some way that yields both efficiency and completeness. **Simulated annealing** is such **an** algorithm.



```
function SIMULATED-ANNEALING(problem, schedule) returns a solution state
  inputs: problem, a problem
          schedule, a mapping from time to "temperature"

  current 4— MAKE-NODE(problem.INITIAL-STATE)
  for t = 1 to co do
      T — schedule(t)
      if T - O then return current
      next ← a randomly selected successor of current
        E 4— next.VALUE — current.VALUE
      if ΔE > 0 then current ← next
      else current ← next only with probability e [illegible]
```

Figure 4.5 The simulated annealing algorithm, a version of stochastic hill climbing where some downhill moves are allowed. Downhill moves are accepted readily early in the anneal-ing schedule and then less often as time goes on. The *schedule* input determines the value of the temperature *T* as a function of time.

Local beam search

The **local beam search** algorithm keeps track of *k* states rather than **just one.** It begins with *k* randomly generated states. At each step, all the successors of all *k* states are generated. If anyone is a goal, the algorithm halts. Otherwise, it selects the *k* best successors from the complete list and repeats.

At first sight, a local beam search with *k* states might seem to be nothing more than running *k* random restarts in parallel instead of in sequence. **In** fact, the two algorithms are quite different. In a random-restart search, each search process runs independently of the others. *In a local beam search, useful information is passed among the parallel search threads.*

Adversarial search:-Adversarial search is used in games where one player's attempts to maximize their fitness (win) is opposed by another player.

Search For Game

Game-playing programs developed by AI researchers since the beginning of the modern AI era Ex. Programs playing chess, checkers, etc

Specifics of the game search:

- Sequences of player's decisions **we control**
- Decisions of other player(s) **we do not control**

Contingency problem: many possible opponents' moves must be ―covered‖ by the solution. Opponent's behaviour introduces an uncertainty in to the game. We do not know exactly what the response is going to be.

• **Rational opponent** – maximizes it own **utility (payoff) function**

Types of game problems:

– **Adversarial games:**

• Win of one player is a loss of the other

– **Cooperative games:**

• Players have common interests and utility function

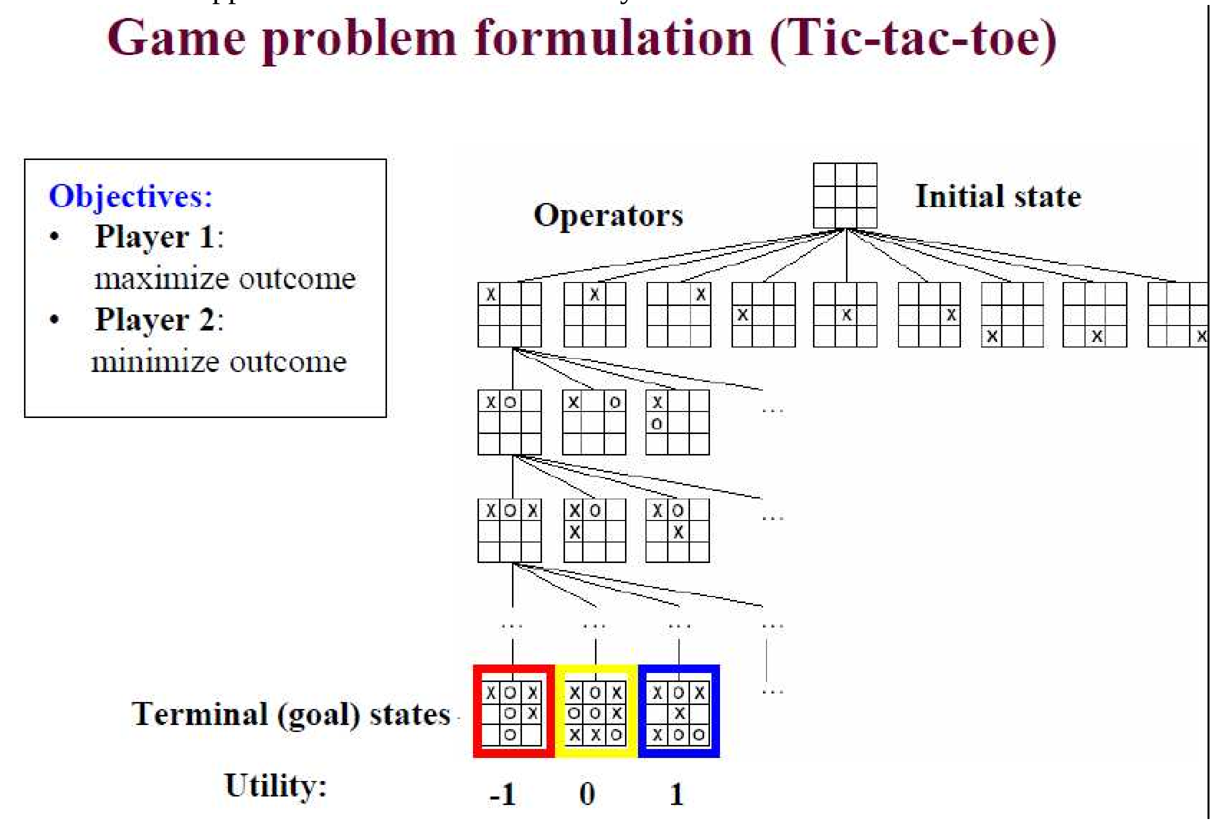
Game search problem

• **Game problem formulation:**

– **Initial state**: initial board position + info whose move it is
– **Operators:** legal moves a player can make
– **Goal (terminal test):** determines when the game is over
– **Utility (payoff) function**: measures the outcome of the game and its desirability

• **Search objective:**

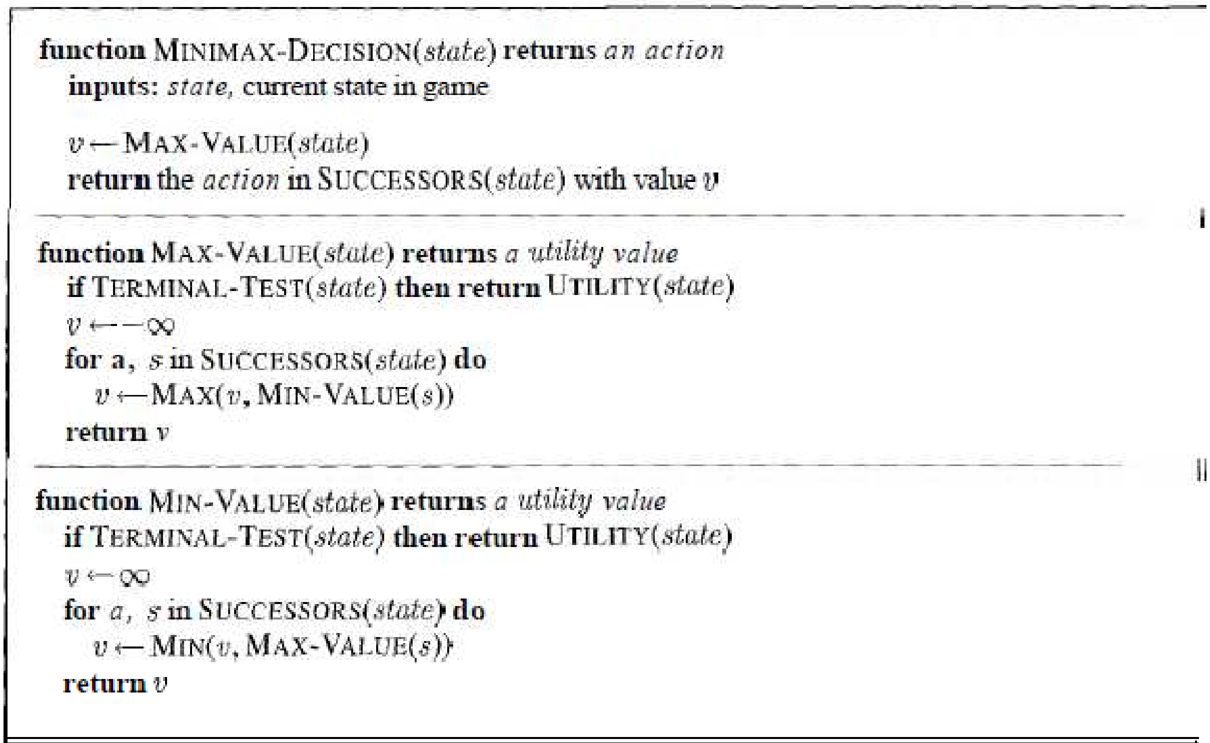
– find the sequence of player's decisions (moves) maximizing its utility (payoff)
– Consider the opponent's moves and their utility


Game problem formulation (Tic-tac-toe)
Objectives:
• Player 1: maximize outcome
• Player 2: minimize outcome
Operators
Initial state
Terminal (goal) states
Utility:
-1
0
1

The minimax algorithm

The minimax algorithm computes the minimax decision from the current state. It uses a simple recursive computation of the minimax values of each successor state, directly implementing the defining equations. The recursion proceeds all the way down to the leaves of the tree, and then the minimax values are **backed up** through the tree as the recursion unwinds.

The minimax algorithm performs a complete depth-first exploration of the game tree. If the maximum depth of the tree is m and there are *b* legal moves at each point, then the time complexity of the minimax algorithm is *0*(b_m). The space complexity is *0(bm.)* for an algorithm that generates all actions at once, or *0(m)* for an algorithm that generates actions one at a time.



```
function MINIMAX-DECISION(state) returns an action
   inputs: state, current state in game

   v ← MAX-VALUE(state)
   return the action in SUCCESSORS(state) with value v
```

```
function MAX-VALUE(state) returns a utility value
   if TERMINAL-TEST(state) then return UTILITY(state)
   v ← −∞
   for a, s in SUCCESSORS(state) do
      v ← MAX(v, MIN-VALUE(s))
   return v
```

```
function MIN-VALUE(state) returns a utility value
   if TERMINAL-TEST(state) then return UTILITY(state)
   v ← ∞
   for a, s in SUCCESSORS(state) do
      v ← MIN(v, MAX-VALUE(s))
   return v
```

Figure 6.3 An algorithm for calculating minimax decisions. It returns the action corresponding to the best possible move, that is, the move that leads to the outcome with the best utility, under the assumption that the opponent plays to minimize utility. The functions *MAX-VALUE* and *MIN-VALUE* go through the whole game tree, all the way to the leaves, to determine the backed-up value of a state.

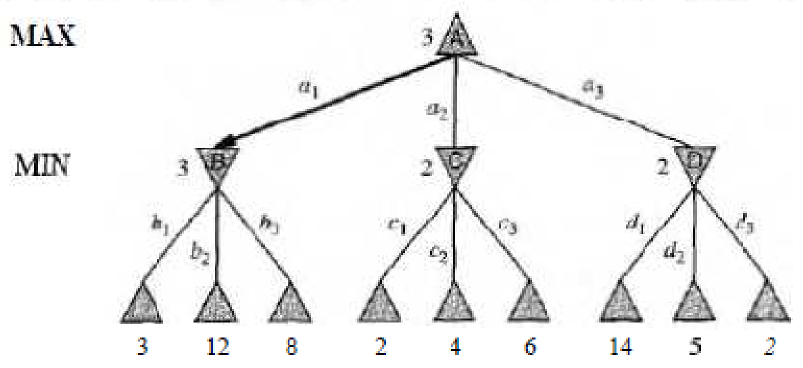


Figure 6.2 A two-ply game tree. The △ nodes are "MAX nodes," in which it is MAX's turn to move, and the ▽ nodes are "MIN nodes." The terminal nodes show the utility values for MAX; the other nodes are labeled with their minimax values. MAX's best move at the root is a_1, because it leads to the successor with the highest minimax value, and MIN's best reply is b_1, because it leads to the successor with the lowest minimax value.

Optimal decisions in multiplayer games

First, we need to replace the single value for each node with a vector of values. For example, in a three-player game with players A, B, and *C*, a vector *(v_A,v_B,v_C)* is associated with each node. For terminal states, this vector gives the utility of the state from each player's viewpoint.

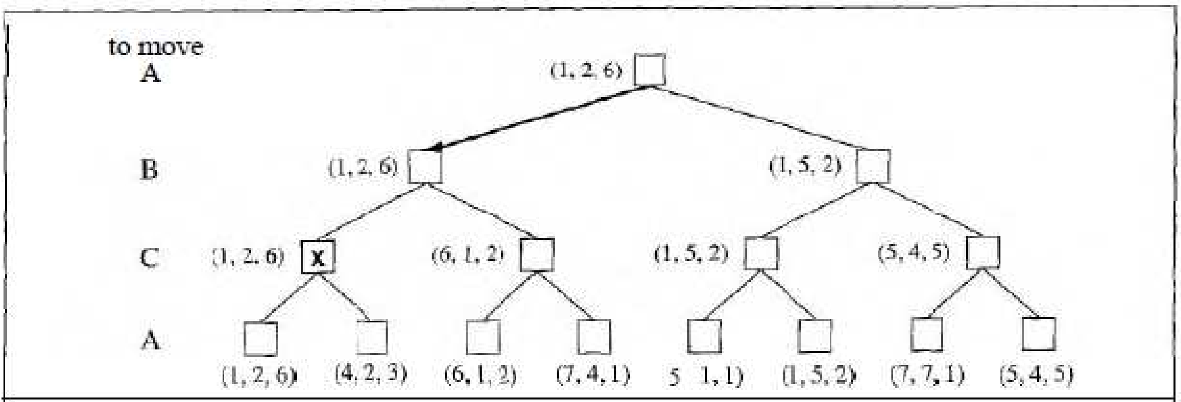
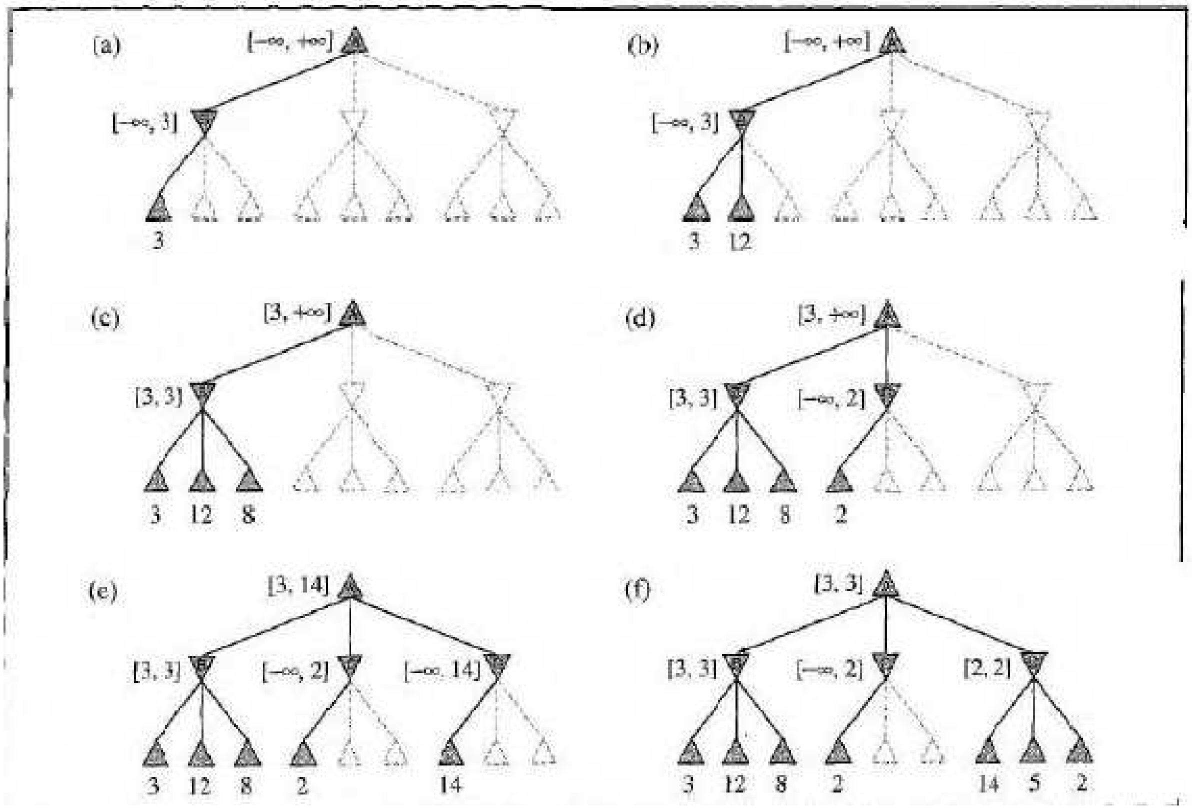


Figure 6.4 The first three ply of a game tree with three players (A, *B*, C). Each node is labeled with values from the viewpoint of each player. The best move is marked at the root.

ALPHA—BETA PRUNING

The problem with minimax search is that the number of game states it has to examine is exponential in the depth of the tree. The trick is that it is possible to compute the correct minimax decision without looking at every node in the game tree.

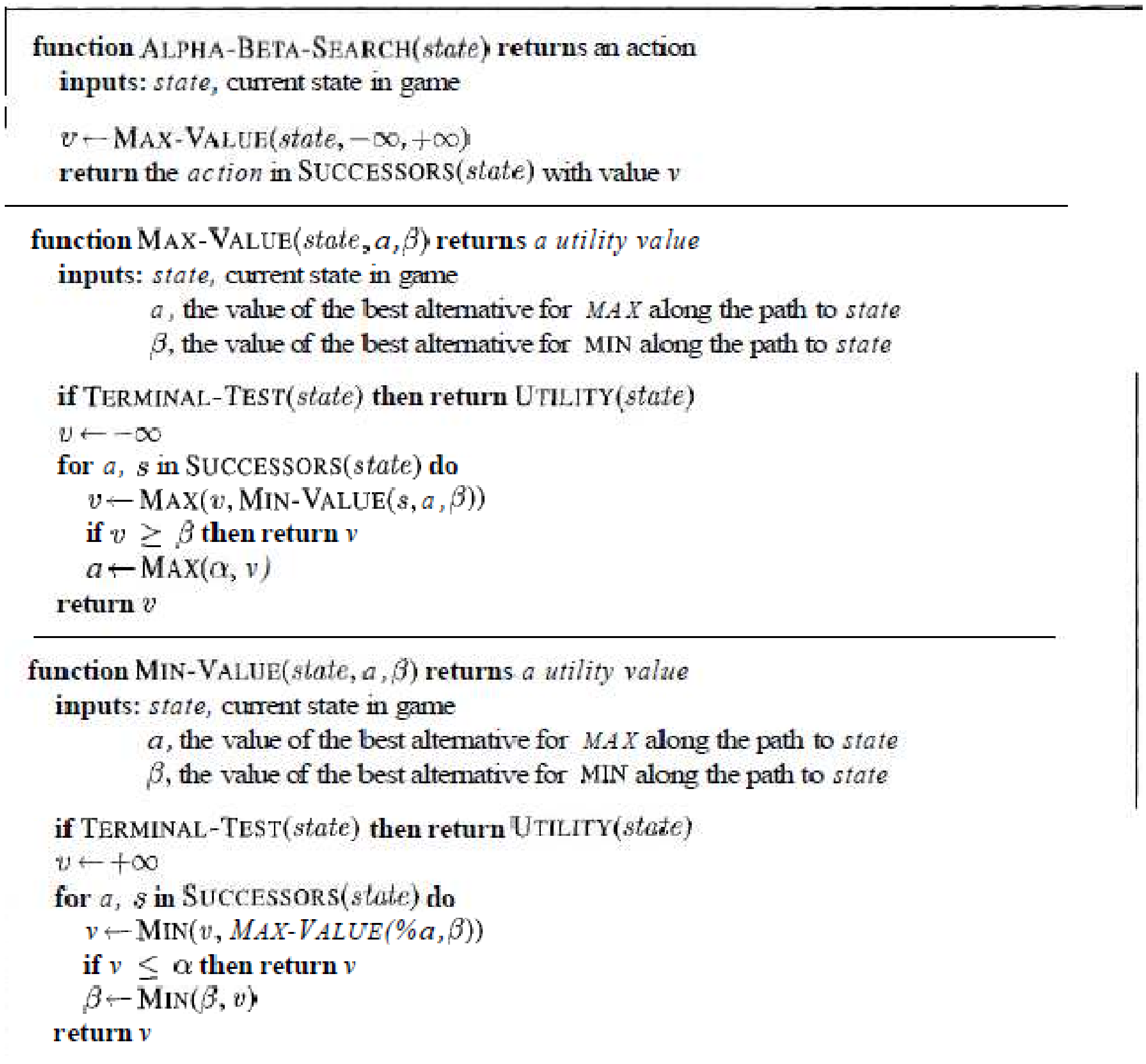
Alpha–beta pruning is a search algorithm that seeks to decrease the number of nodes that are evaluated by the minimax algorithm in its search tree. It stops completely evaluating a move when at least one possibility has been found that proves the move to be worse than a previously examined move. Such moves need not be evaluated further. When applied to a standard minimax tree, it returns the same move as minimax would, but prunes away branches that cannot possibly influence the final decision.



Alpha-beta pruning gets its name from the following two parameters that describe bounds on the backed.-up values that appear anywhere along the path:

- α = the value of the best (i.e., highest-value) choice we have found so far at any choice point along the path for MAX.
- β = the value of the best (i.e., lowest-value) choice we have found so far at any choice point along the path for MIN.

Alpha-beta search updates the values of α and β as it goes along and prunes the remaining branches at a node (i.e., terminates the recursive call) as soon as the value of the current node is known to be worse than the current *a* or *p* value for MAX or MIN, respectively.



```
function ALPHA-BETA-SEARCH(state) returns an action
  inputs: state, current state in game

  $v \leftarrow$ MAX-VALUE(state, $-\infty$, $+\infty$)
  return the action in SUCCESSORS(state) with value v

function MAX-VALUE(state, a, $\beta$) returns a utility value
  inputs: state, current state in game
          a, the value of the best alternative for MAX along the path to state
          $\beta$, the value of the best alternative for MIN along the path to state

  if TERMINAL-TEST(state) then return UTILITY(state)
  $v \leftarrow -\infty$
  for a, s in SUCCESSORS(state) do
     $v \leftarrow$ MAX(v, MIN-VALUE(s, a, $\beta$))
     if $v \geq \beta$ then return v
     a $\leftarrow$ MAX($\alpha$, v)
  return v

function MIN-VALUE(state, a, $\beta$) returns a utility value
  inputs: state, current state in game
          a, the value of the best alternative for MAX along the path to state
          $\beta$, the value of the best alternative for MIN along the path to state

  if TERMINAL-TEST(state) then return UTILITY(state)
  $v \leftarrow +\infty$
  for a, s in SUCCESSORS(state) do
     $v \leftarrow$ MIN(v, MAX-VALUE(%a, $\beta$))
     if $v \leq \alpha$ then return v
     $\beta \leftarrow$ MIN($\beta$, v)
  return v
```

Fig: - The alpha-beta search algorithm.