Module 2

PROBLEM-SOLVING

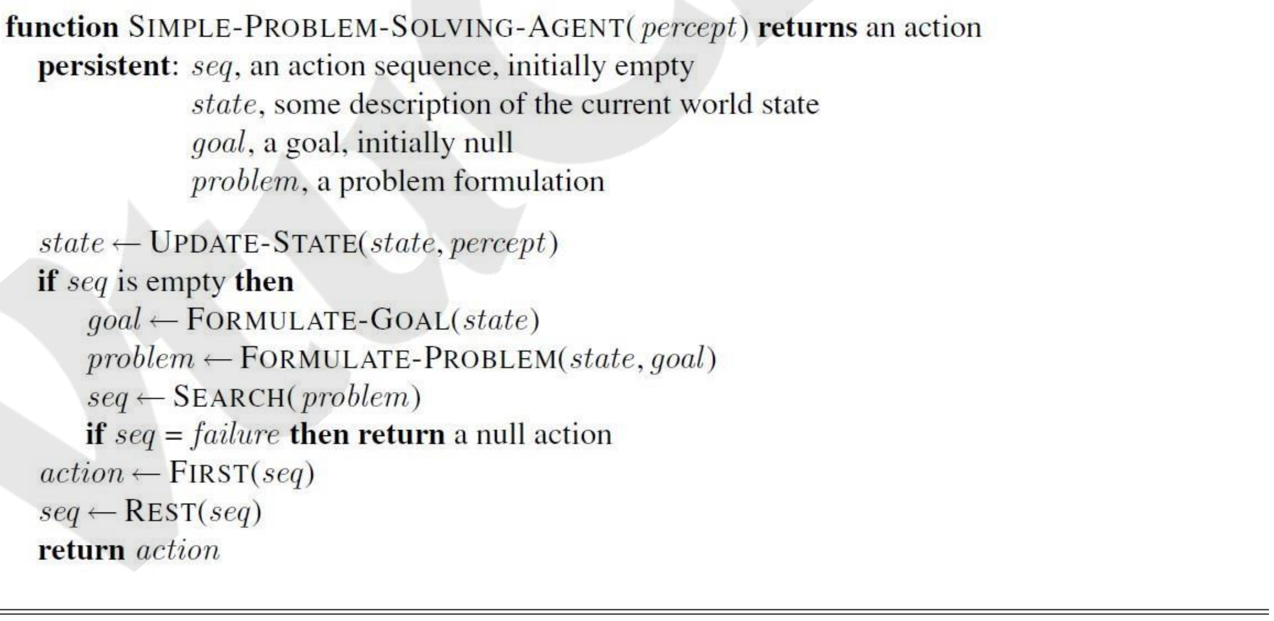
1. PROBLEM-SOLVING AGENTS

Goal formulation, based on the current situation and the agent's performance measure, is the first step in problem solving.

Problem formulation is the process of deciding what actions and states to consider, given a goal.

An agent with several immediate options of unknown value can decide what to do by first examining future *actions that eventually lead to states of known value.*

The process of looking for a sequence of actions that reaches the goal 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. Thus, a simple "formulate, search, execute" design for the agent, as shown in Figure 3.1. 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 a new goal.



```
function SIMPLE-PROBLEM-SOLVING-AGENT(percept) returns an action
  persistent: 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
      goal ← FORMULATE-GOAL(state)
      problem ← FORMULATE-PROBLEM(state, goal)
      seq ← SEARCH(problem)
      if seq = failure then return a null action
  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.

1.1 Well-defined problems and solutions

A problem can be defined formally by five components: - (Explained with example Romania)

- The **initial** state that the agent starts in_ For example, the initial state for our agent in Romania might be described as *In(A rad)*.

A description of the possible actions available to the agent Given a particular state *s,* ACTIONS(s) returns the set of actions that can be executed in *s. We* say that each of these actions is **applicable** in
s. For example, from the state *Ir.(Arad),* the applicable actions are { *Go(Sibiu), Go(Timisoara), Go(Zerim1)}.*

- A description of what each action does; the formal name for this is the **transition model,** specified by a function REsuur(s, a) that returns the state that results from doing action a in state *s.* We also use the term **successor** to refer to any state reachable from a given state by a single action.2 For example, we have

$$\text{RESULT}(In(Arad\,), Go(Zerind\,)) \;=\; In(Zerind\,)\;\;.$$

  - Together, the initial state, actions, and transition model implicitly define the **state space** of the problem—the set of all states reachable from the initial state by any sequence of actions.
  - The state space forms a directed network or **graph** in which the nodes are states and the links between nodes are actions.
  - {The map of Romania shown in Figure 3.2 can be interpreted as a state-space graph if we view each road as standing for two driving actions, one in each direction.)
  - A **path** in the state space is a sequence of states connected by a sequence of actions.

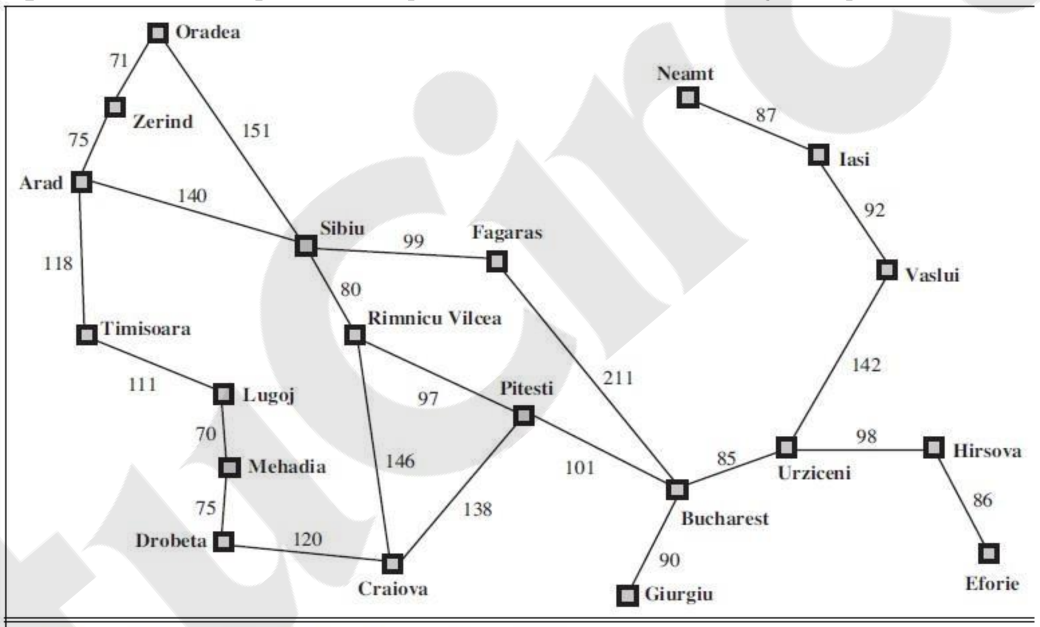


Figure 3.2 A simplified road map of part of Romania.

- The **goal test,** which determines whether a given state is a goal state. Sometimes there is an explicit set of possible goal states, and the test simply checks whether the given state is one of them. The agent's goal in Romania is the singleton set { *In(Bucharest)}.*

- **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. For the agent trying to get to Bucharest, time is of the essence, so the cost of a path might be its length in kilometers.

Assume that the cost of a path can be described as the *guns* of the costs of the individual actions along the path 3 The **step cost of taking** action a in state *s* to reach state *s'* is denoted by *e(s,* **a, s').** The step costs for Romania are shown in Figure 3.2 as route distances.

Formulating problems.

A formulation of the problem of getting to Bucharest in terms of the initial state, actions, transition model, goal test, and path cost. This formulation seems reasonable, but it is still a *model*—an abstract mathematical description—and not the real thing. Compare the simple state description we have chosen, *In(Arad)*, where the state of the world includes so many things: the traveling companions, the current radio program, the scenery out of the window, the proximity of law enforcement officers, the distance to the next rest stop, the condition of the road, the weather, and so on. All these considerations are left out of our state descriptions because they are irrelevant to the problem of finding a route to Bucharest.

- The process of removing detail from a representation is called **abstraction**.
- Abstract the actions themselves.
- A driving action has many effects.
- Besides changing the location of the vehicle and its occupants, it takes up time, consumes fuel, generates pollution, and changes the agent.
- Our formulation takes into account only the change in location. Also, there are many actions that we omit altogether: turning on the radio, looking out of the window, slowing down fur law enforcement officers, and so on.
- we don't specify actions at the level of "turn steering wheel to the left by one degree."

Now consider a solution to the abstract problem: for example. The path from Arad to Sibiu to Rimnicu Vilcea to Pitesti to Bucharest. This abstract solution corresponds to a large number of more detailed paths.

For example, we could drive with the radio on between Sibiu and Rimnicu Vilcea, and then switch it off for the rest of the trip.

- The abstraction is *valid* if we can expand any abstract solution into a solution in the more detailed world; a sufficient condition is that for every detailed state that is "in Arad." there is a detailed path to some state that is "in Sibiu," and so on.
- The abstraction is *useful* if carrying out each of the actions in the solution is easier than the original problem; in this case they are easy enough that they can be carried out without further search or planning by an average driving agent.
- The choice of a good abstraction thus involves removing as much detail as possible while retaining validity and ensuring that **the** abstract actions are easy to carry out. Were it not for the ability to construct useful abstractions, intelligent agents would be completely swamped by the real world.

2. EXAMPLE PROBLEMS (Toy problems)

A toy problem is intended to illustrate or exercise various problem-solving methods. It can be given a concise, exact description and hence is usable by different researchers to compare the performance of algorithms. A real-world problem is one whose solutions people actually care about. Such problems tend not to have a single agreed-upon description, but we can give the general flavour of their formulations.

2.1. Vacuum world

The State space for the vacuum world is shown in figure3.3. Vacuum world problem can be formulated as a problem as follows:

- **States**: The state is determined by both the agent location and the dirt locations. The agent is in one of two locations, each of which might or might not contain dirt. Thus, there are $2 \times 2^2 = 8$ possible world states. A larger environment with n locations has $n \cdot 2^n$ states.

- **Initial state**: Any state can be designated as the initial state.
- **Actions**: In this simple environment, each state has just three actions: *Left*, *Right*, and *Suck*. Larger environments might also include *Up* and *Down*.
- **Transition model**: The actions have their expected effects, except that moving *Left* inthe leftmost square, moving *Right* in the rightmost square, and *Suck*ing in a clean squarehave no effect. 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.

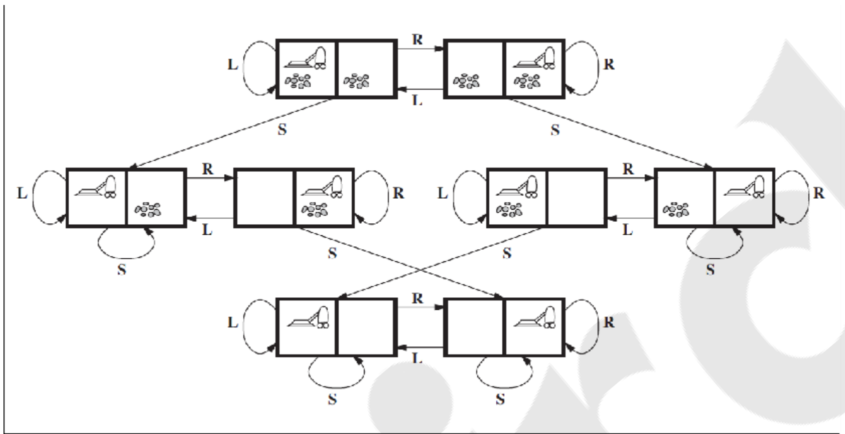


Figure 3.3 The state space for the vacuum world. Links denote actions: L = *Left*, R = *Right*, S = *Suck*.

2.2. 8-puzzle Problem

The **8-puzzle,** an instance of which is shown in Figure 3.4, consists of a 3 x3 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 tight of the figure. The standard formulation is as follows:

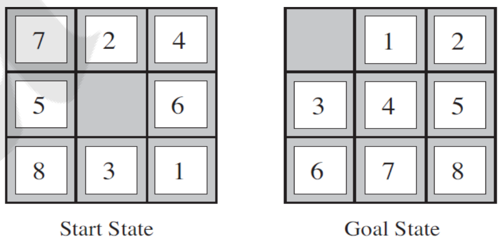


Figure 3.4 A typical instance of the 8-puzzle.

States: A state description specifies the location of each of the eight Ides 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 he reached Front exactly half of the possible initial states.
- **Actions:** The simplest formulation defines the actions as movements of the blank space *Left*, *Right*, *Up*, or Down. Different subsets of these are possible depending on where the blank is.
- **Transition model:** Given a state and action, this returns the resulting state; for example, if we apply *Left* to start state in Figure 3.4, the resulting state has the 5 and **the blank** switched.

- **Goal test:** This checks whether the state matches the goal configuration shown in Figure 3.4. (Other goal configurations are possible.)
- **Path cost: Each step** costs 1, so the path cost is the number of steps in the path.

The 8-puzzle belongs to the family of **sliding-block puzzles, which are** often used as test problems for new search algorithms in AI. This family is known to be NP-complete, so one does not expect to find methods significantly better in the worst case than the search algorithms described in this chapter and the next. The 8-puzzle has 91/2 =181, 440 reachable states and is easily solved. The 15-puzzle (on **a** 4 x 4 board) **has around 1.3 trillion** states, and random instances can be solved optimally in a few milliseconds by the best search algorithms. The 24-puzzle (on a 5 x 5 board) has around 1025 states, and random instances take several hours to solve optimally.

2.3. 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.) Figure 3.5 shows an attempted solution that fails; the queen in the rightmost column is attacked by the queen at the top left.

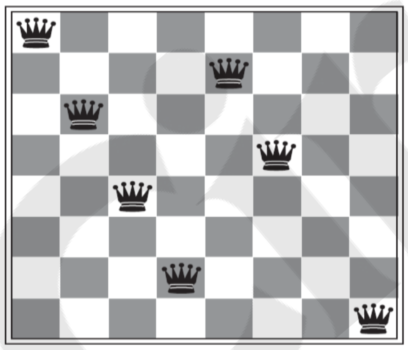


Figure 3.5 Almost a solution to the 8-queens problem. (Solution is left as an exercise.)

Although efficient special-purpose algorithms exist for this problem and for the whole n-queens family, it remains a useful test problem for search algorithms.
There are two main kinds of formulation.

1. 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.
2. **A complete-state formulation** starts with all 8 queens on the board and moves them amend. 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.
- **Actions**: Add a queen to any empty square.
- **Transition model**: Returns the board with a queen added to the specified square.
- **Goal test**: 8 queens are on the board, none attacked.

In this formulation, have 69 • 63 • • • 57 1.8 x 1014 possible sequences to investigate.
A better formulation would prohibit placing a queen in any square that is already attacked:

- **States: All** possible arrangements of n queens (0 < *rt* < 8), one per column in the leftmost *n*. columns, with no queen attacking another.

- **Actions:** Add a queen to **any** square in the leftmost empty column such that it is not attacked by any other queen.

This formulation reduces the 8-queens state space from 1.8×10^{14} to just 2,057, and solutionsare easy to find.

2.4. Toy problem devised by Donald Knuth

Our final toy problem was devised by Donald Knuth (1964) and illustrates how infinite state spaces can arise. Knuth conjectured that, starting with the number 4, a sequence of factorial, square root, and floor operations will reach any desired positive integer. For example, we can reach 5 from 4 as follows:

$$\left\lfloor \sqrt{\sqrt{\sqrt{\sqrt{\sqrt{(4!)!}}}}} \right\rfloor = 5 \,.$$

The problem definition is very simple:

- States: Positive numbers.
- Initial state: 4.
- Actions: Apply factorial, square root, or floor operation (factorial for integers only).
- Transition model: As given by the mathematical definitions of the operations.
- Goal test: State is the desired positive integer.

2. 5. Real-world problems

Consider the airline travel problems that must be solved by a travel-planning Web site:

- States: Each state obviously includes a location (e.g., an airport) and the current time. Furthermore, because the cost of an action (a flight segment) may depend on previous segments, their fare bases, and their status as domestic or international, the state must record extra information about these "historical" aspects.
- Initial state: This is specified by the user's query.
- Actions: Take any flight from the current location, in any seat class, leaving after the current time, leaving enough time for within-airport transfer if needed.
- Transition model: The state resulting from taking a flight will have the flight's destination as the current location and the flight's arrival time as the current time.
- Goal test: Are we at the final destination specified by the user?
- 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.

Touring problems are closely related to route-finding problems, but with an important difference. Consider, for example, the problem "Visit every city in Figure 3.2 at least once, starting and ending in Bucharest" As with route finding, the actions correspond to trips between adjacent cities. The state space, however, is quite different. Each state must include not just the current location but also the *set of cities the agent has visited.* So the initial state would be *In (Bucharest), Visit ed({Bueharest}),* a typical intermedi-ate state would be *fn(Vaslui), Visqed({Bucharest, Urziceni , Vaslui}),* and the goal test would check whether the agent is in Bucharest and all 20 cities have been visited.

The **traveling 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. In addition to planning trips for traveling salespersons, these algorithms have been used for tasks such as planning movements of automatic circuit-board drills and of stocking machines on shop floors.

Robot navigation is a generalization of the route-finding problem described earlier. Rather than following a discrete set of routes, a robot can move in a continuous space with (in principle) an infinite set of possible actions and states. For a circular robot moving on a flat surface, the space is essentially two-dimensional. When the robot has arms and legs or wheels that must also be controlled, the search space becomes many- dimensional. Advanced techniques are required just to make the search space finite.

3. SEARCHING FOR SOLUTIONS

After formulating some problems, need to solve them. A solution is an action sequence, so search algorithms work by considering various possible action sequences. The possible action sequences starting at the initial state form a search **tree** with the initial state at the root; the branches are actions and the **nodes** correspond to states in the state space of the problem.
Difference between the state space and the search tree.

- state space: states + actions
- search tree: nodes + actions

Figure 3.6 shows the fast few steps in growing the search tree for finding a route from Arad to Bucharest.

- The root node of the tree corresponds to the initial state, *In(Arad).*
- The first step is to test whether this is a goal state.
- Then we need to consider taking various actions. do this by expanding the current state; that is, applying each legal action to the current state.
- Thereby **generating** a new set of states. In this case, add three branches from the **parent node** *In(Arad)* leading to three new **child nodes:** *in(Sibw), In(Timisaara),* and *In(Zerind).* Then choose which of these three possibilities to consider farther.

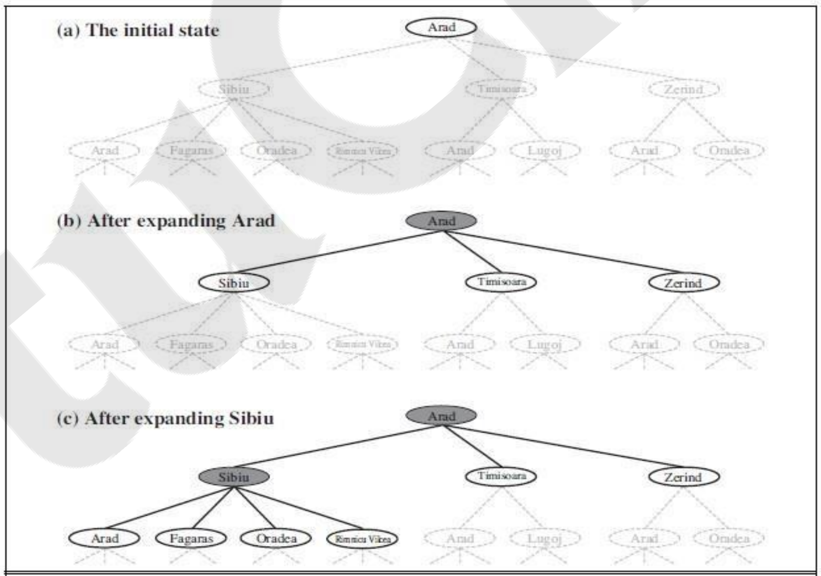


Figure 3.6 Partial search trees for finding a route from Arad to Bucharest. Nodes that have been expanded are shaded; nodes that have been generated but not yet expanded are outlined in bold; nodes that have not yet been generated are shown in faint dashed lines.

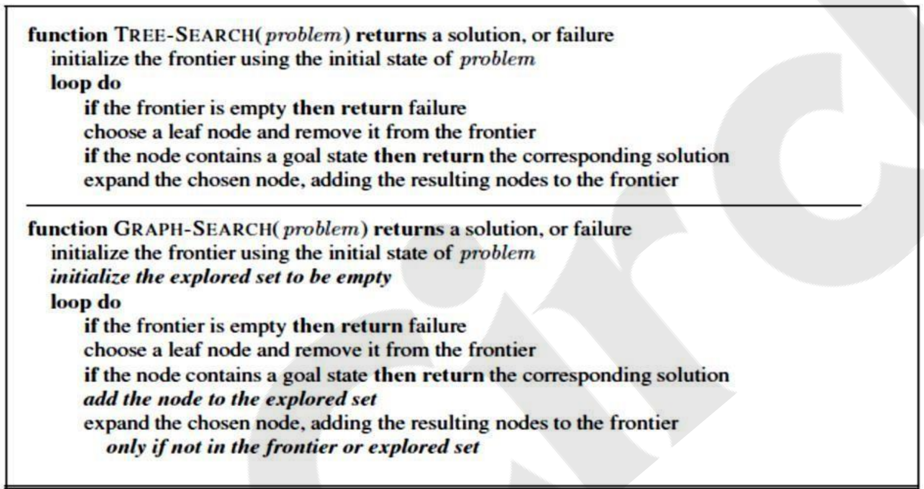
The general TREE-SEARCH algorithm is shown informally in Figure 3.7. Search algorithms all share this basic structure; they vary primarily according to how they choose which state to expand next—the so-called **search strategy.**

Algorithms that forget their history are doomed to repeat it. The way to avoid exploring redundant paths is to remember where one has been. To do this, we augment the TREE-SEARCH algorithm with a data structure called the **explored set** (also known as the **closed list**), which remembers every expanded node. Newly generated nodes that match previously generated nodes—ones in the explored set or the frontier—can be discarded instead of being added to the frontier.

The new algorithm, called GRAPH-SEARCH, is shown informally in Figure 3.7. The search tree constructed by the GRAPH-SEARCH algorithm contains at most one copy of each state, so we can think of it as growing a tree directly on the state-space graph, as shown in Figure 3.8.

The frontier **separates** the state-space graph into the explored region and the unexplored region, so that every path from the initial state to an unexplored state has to pass through a state in the frontier is illustrated in Figure 3.9.

As every step moves a state from the frontier into the explored region while moving some states from the unexplored region into the frontier, we see that the algorithm is *systematically* examining the states in the state space, one by one, until it finds a solution.



```
function TREE-SEARCH(problem) returns a solution, or failure
  initialize the frontier using the initial state of problem
  loop do
      if the frontier is empty then return failure
      choose a leaf node and remove it from the frontier
      if the node contains a goal state then return the corresponding solution
      expand the chosen node, adding the resulting nodes to the frontier

function GRAPH-SEARCH(problem) returns a solution, or failure
  initialize the frontier using the initial state of problem
  initialize the explored set to be empty
  loop do
      if the frontier is empty then return failure
      choose a leaf node and remove it from the frontier
      if the node contains a goal state then return the corresponding solution
      add the node to the explored set
      expand the chosen node, adding the resulting nodes to the frontier
         only if not in the frontier or explored set
```

Figure 3.7 An informal description of the general tree-search and graph-search algorithms. The parts of GRAPH-SEARCH marked in bold italic are the additions needed to handle repeated states.

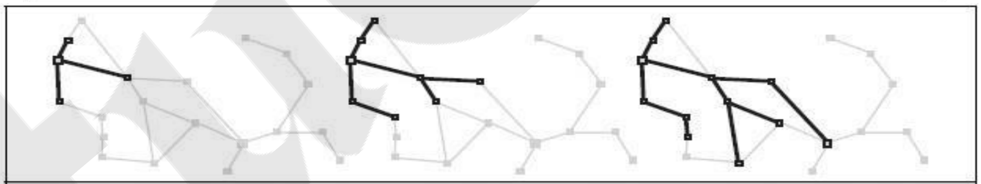


Figure 3.8 A sequence of search trees generated by a graph search on the Romania problem of Figure 3.2. At each stage, we have extended each path by one step. Notice that at the third stage, the northernmost city (Oradea) has become a dead end: both of its successors are already explored via other paths.

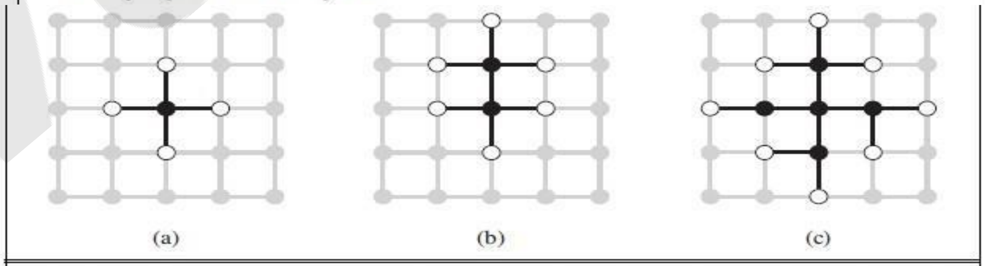


Figure 3.9 The separation property of GRAPH-SEARCH, illustrated on a rectangular-grid problem. The frontier (white nodes) always separates the explored region of the state space (black nodes) from the unexplored region (gray nodes). In (a), just the root has been expanded. In (b), one leaf node has been expanded. In (c), the remaining successors of the root have been expanded in clockwise order.

3.1. Infrastructure for search algorithms

Search algorithms require a data structure to keep track of the search tree that is being constructed. For each node n of the tree, we have a structure that contains four components:

- *n*.STATE: the state in the state space to which the node corresponds;
- *n*.PARENT: the node in the search tree that generated this node;
- *n*.ACTION: the action that was applied to the parent to generate the node;
- *n*.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.

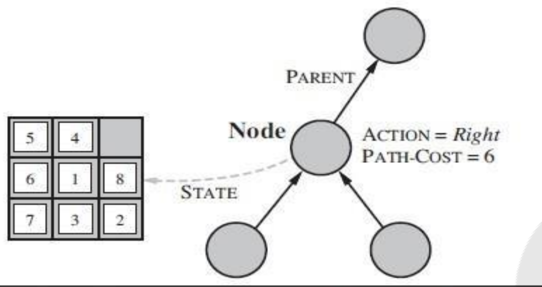


Figure 3.10 Nodes are the data structures from which the search tree is constructed. Each has a parent, a state, and various bookkeeping fields. Arrows point from child to parent.

Given the components for a parent node, it is easy to see how to compute the necessary components for a child node. The function CHILD-NODE takes a parent node and an action and returns the resulting child node:

```
function CHILD-NODE(problem, parent, action) returns a node
  return a node with
     STATE = problem.RESULT(parent.STATE, action),
     PARENT = parent, ACTION = action,
     PATH-COST = parent.PATH-COST + problem.STEP-COST(parent.STATE, action)
```

The node data structure is depicted in Figure 3.10 shows how the PARENT pointers string the nodes together into a tree structure. These pointers also allow the solution path to be extracted when a goal node is found;
Use the SOLUTION function to return the sequence of actions obtained by following parent pointers back to the root.

- A node is a bookkeepingdata 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 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.

Now that nodes need to put them somewhere. The frontier needs to be stored in such a way that the search algorithm can easily choose the next node to expand according to its preferred strategy. The appropriate data structure for this is a queue. The operations on a queue are as follows:

- EMPTY?(*queue*) returns true only if there are no more elements in the queue.
- POP(*queue*) removes the first element of the queue and returns it.
- INSERT(*element*, *queue*) inserts an element and returns the resulting queue.

3.2. Measuring problem-solving performance

Before we get into the design of specific search algorithms, we need to consider the criteria that might be used to choose among them. We can 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?

The typical measure is the size of the state space graph, $|V| + |E|$, where V is the set of vertices (nodes) of the graph and E is the set of edges (links).

Complexity is expressed in terms of three quantities:

- b, the branching factor or maximum number of successors of any node;
- d, the depth of the shallowest goal node (i.e., the number of steps along the path from the root); and
- m, the maximum length of any path in the state space.

Time is often measured in terms of the number of nodes generated during the search, and space in terms of the maximum number of nodes stored in memory.

4. UNINFORMED SEARCH STRATEGIES

UNINFORMED SEARCH STRATEGIES means that the strategies have no additional information about states beyond that provided in the problem definition. All they can do is generate successors and distinguish a goal state from a non-goal state.

4.1. 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 is an instance of the general graph-search algorithm (Figure 3.7) inwhich the *shallowest* unexpanded node is chosen for expansion. This is achieved very simplyby using a FIFO queue for the frontier.
- Thus, new nodes (which are always deeper than theirparents) go to the back of the queue, and old nodes, which are shallower than the new nodes,get expanded first.
- There is one slight tweak on the general graph-search algorithm, which isthat the goal test is applied to each node when it is *generated* rather than when it is selected forexpansion.
- Figure 3.12 shows the progress of the search on asimple binary tree.

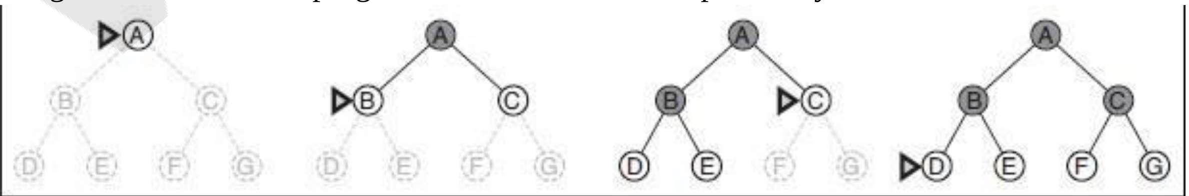
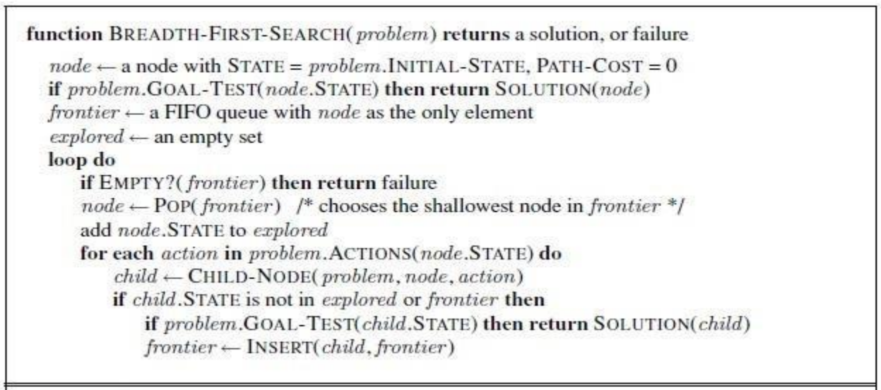


Figure 3.12 Breadth-first search on a simple binary tree. At each stage, the node to be expanded next is indicated by a marker.



```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  frontier ← a FIFO queue with node as the only element
  explored ← an empty set
  loop do
      if EMPTY?(frontier) then return failure
      node ← POP(frontier)  /* chooses the shallowest node in frontier */
      add node.STATE to explored
      for each action in problem.ACTIONS(node.STATE) do
          child ← CHILD-NODE(problem, node, action)
          if child.STATE is not in explored or frontier then
              if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
              frontier ← INSERT(child, frontier)
```

Figure 3.11 Breadth-first search on a graph.

- **Time Complexity:** In the worst case, it is the last node generated at that level. Then the total number of nodes generated is
  $b + b^2 + b^3 + \cdots + b^d = O(b^d)$.
- Breadth-first search is optimal because it always expands the *shallowest* unexpanded node.
- The memory requirements are a bigger problem for breadth-first search than is execution time.
- Exponential-complexity search problems cannot be solved by uniformed methods for any but the smallest instances.

Advantages:

- BFS will provide a solution if any solution exists.
- If there are more than one solutions for a given problem, then BFS will provide the minimal solution which requires the least number of steps.

Disadvantages:

- It requires lots of memory since each level of the tree must be saved into memory to expand the next level.
- *BFS needs lots of time if the solution is far away from the root node.*

4.2 Depth-first search:

Depth-first search always expands the *deepest* node in the current frontier of the search tree.The progress of the search is illustrated in Figure 3.16.

- 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 frontier, so then the search “backs up” to the next deepest node that still has unexplored successors.
- Depth-first search uses a LIFO queue. A LIFO queue means that the most recently generated node is chosen for expansion.
- This must be the deepest unexpanded node because it is one deeper than its parent—which, in turn, was the deepest unexpanded node when it was selected.

The **time complexity** of depth-first graph search is bounded by the size of the state space (which may be infinite). Generate all of the $O(b^m)$ nodes in the search tree, where m is the maximum depth of any node.

Advantages:

- DFS requires very less memory as it only needs to store a stack of the nodes on the path from root node to the current node.

- It takes less time to reach to the goal node than BFS algorithm (if it traverses in the right path).

Disadvantages:

- There is the possibility that many states keep re occurring, and there is no guarantee of finding the solution.
- DFS algorithm goes for deep down searching and sometime it may go to the infinite loop.

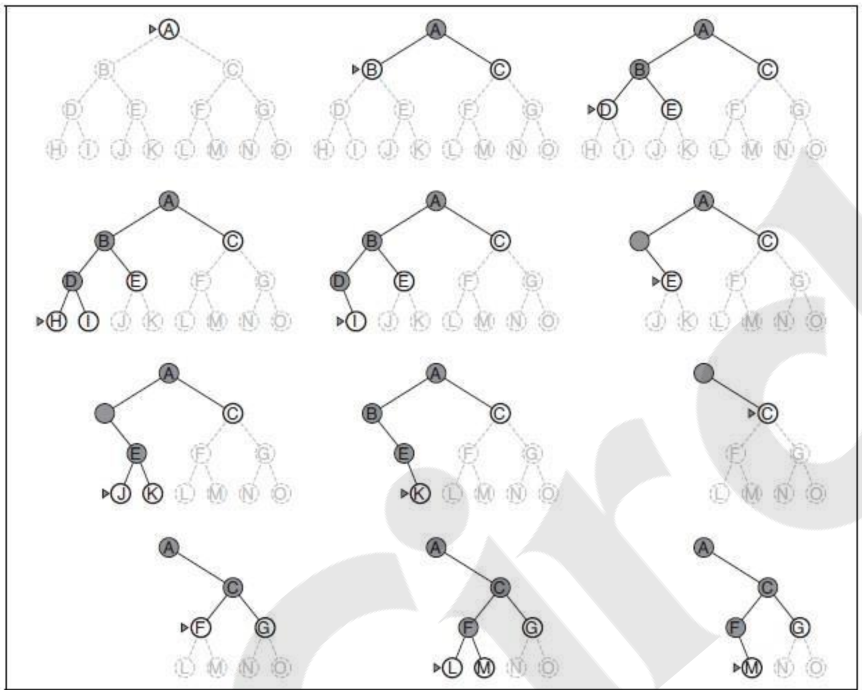


Figure 3.16 Depth-first search on a binary tree. The unexplored region is shown in light gray. Explored nodes with no descendants in the frontier are removed from memory. Nodes at depth 3 have no successors and M is the only goal node.

4.3. Iterative deepening depth-first search:

Iterative deepening search (IDS) (or iterative deepening depth-first search) is a general strategy,often used in combination with depth-first tree search, that finds the best depth limit. It doesthis 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.

The algorithm is shown in Figure 3.18. Iterative deepening combines the benefits of depth-first and breadth-first search.

- Like depth-first search, its memory requirements are modest: $O(bd)$to be precise.
- Like breadth-first search, it is complete when the branching factor is finite andoptimal when the path cost is a non decreasing function of the depth of the node.

```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution, or failure
  for depth = 0 to ∞ do
    result ← DEPTH-LIMITED-SEARCH(problem, depth)
    if result ≠ cutoff then return result
```

Figure 3.18 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.

Advantages: It combines the benefits of BFS and DFS search algorithm in terms of fast search and memory efficiency.
Disadvantages: The main drawback of IDS is 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 in the worst case is

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

which gives a **time complexity of $O(b^d)$**—asymptotically the same as breadth-first search. There is some extra cost for generating the upper levels multiple times, but it is not large.

Forexample, if $b = 10$ and $d = 5$, the numbers are

$$N(\text{IDS}) = 50 + 400 + 3,000 + 20,000 + 100,000 = 123,450$$
$$N(\text{BFS}) = 10 + 100 + 1,000 + 10,000 + 100,000 = 111,110 .$$

Figure 3.19shows four iterations of ITERATIVE-DEEPENING-SEARCH on a binary search tree, where the solution is found on the fourth iteration.

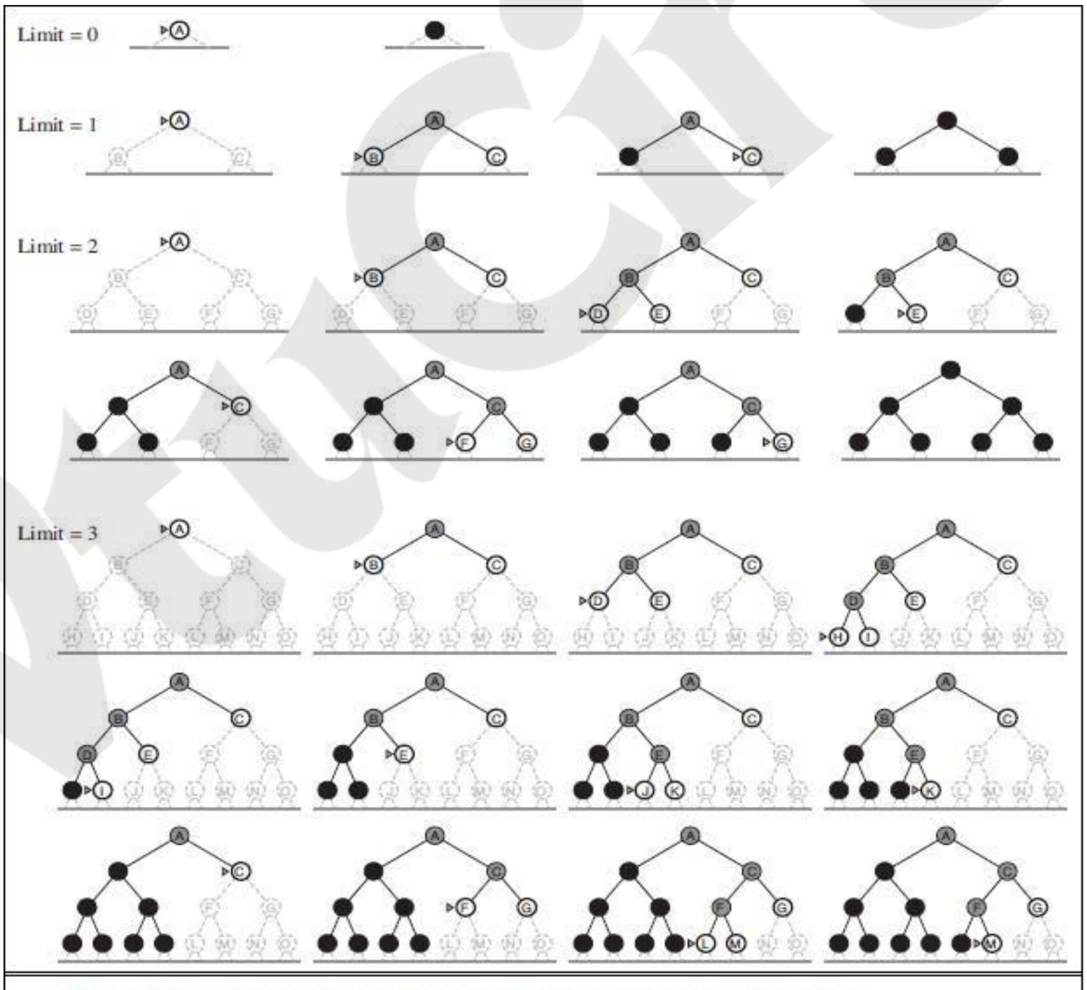


Figure 3.19 Four iterations of iterative deepening search on a binary tree.

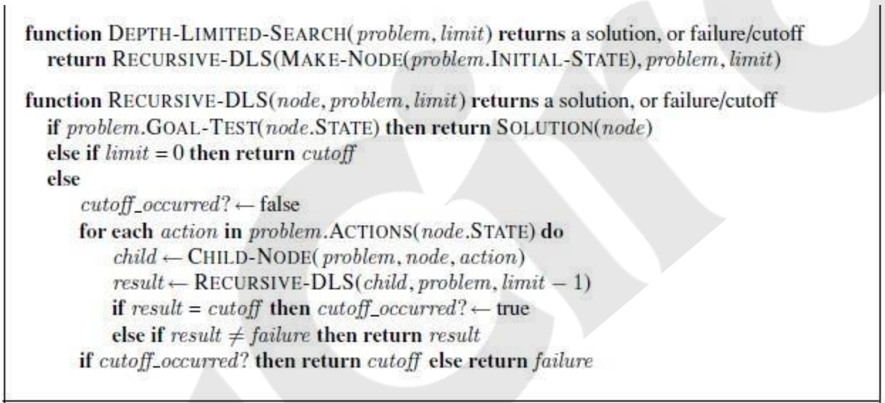
4.4. Depth-limited search

The failure of depth-first search in infinite state spaces can be alleviated by supplying depth-first search with a predetermined depth limit *P*. That is, nodes at depth *E* are treated as if they have no successors. This approach is called **depth-limited search.**

- The depth limit solves the infinite-path problem.
- Unfortunately, it also introduces an additional source of incompleteness if we choose $l < d$, that is, the shallowest goal is beyond the depth limit. (This is likely when *d* is unknown.)
- Depth-limited search will also be non optimal if we choose Q > *d.*
- Its **time complexity** is $O(b^l)$ and its space complexity is *O(bl).*
- Depth-first search can be viewed as a special case of depth-limited search with $l \rightarrow \infty$.
- Depth-limited search can be implemented as a simple modification to the general tree-or graph-search algorithm.

Alternatively, it can be implemented as a simple recursive algorithm as shown in Figure 3.17.
Notice that depth-limited search can terminate with two kinds of failure:

- the standard *failure* value indicates no solution;
- the *cutoff* value indicates no solution within the depth limit.



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

function RECURSIVE-DLS(node, problem, limit) returns a solution, or failure/cutoff
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  else if limit = 0 then return cutoff
  else
     cutoff_occurred? ← false
     for each action in problem.ACTIONS(node.STATE) do
        child ← CHILD-NODE(problem, node, action)
        result ← RECURSIVE-DLS(child, problem, limit − 1)
        if result = cutoff then cutoff_occurred? ← true
        else if result ≠ failure then return result
     if cutoff_occurred? then return cutoff else return failure
```

Figure 3.17 A recursive implementation of depth-limited tree search.

4.5. Uniform-cost search

Uniform-cost search expands the node n with the lowest path cost g(n).

- This is done by storing the frontier as a priority queue ordered by q.
- The algorithm is shown in Figure 3.14.

In addition to the ordering of the queue by path cost, there are two other significant differences from breadth-first search.

- The first is that the goal test is applied to a node when it is selected for expansion rather than when it is first generated. The reason is that the first goal node that is generated may be on a suboptimal path.
- The second difference is that a test is added in case a better path is found to a node currently on the frontier.

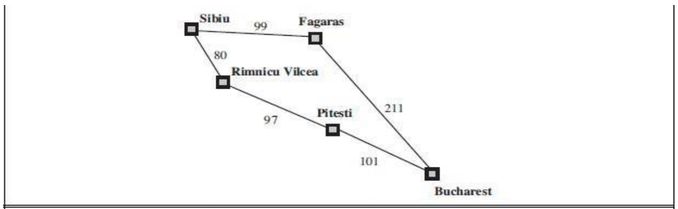
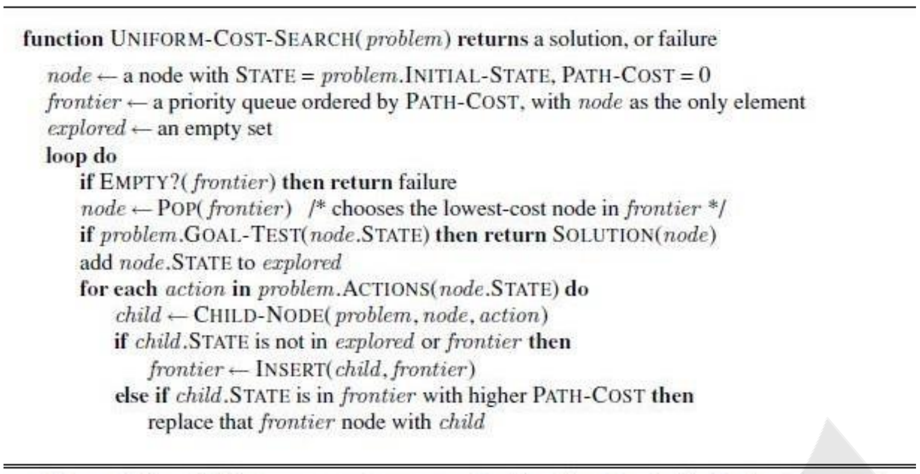


Figure 3.15 Part of the Romania state space, selected to illustrate uniform-cost search.



```
function UNIFORM-COST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  frontier ← a priority queue ordered by PATH-COST, with node as the only element
  explored ← an empty set
  loop do
      if EMPTY?(frontier) then return failure
      node ← POP(frontier)   /* chooses the lowest-cost node in frontier */
      if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
      add node.STATE to explored
      for each action in problem.ACTIONS(node.STATE) do
          child ← CHILD-NODE(problem, node, action)
          if child.STATE is not in explored or frontier then
              frontier ← INSERT(child, frontier)
          else if child.STATE is in frontier with higher PATH-COST then
              replace that frontier node with child
```

Figure 3.14 Uniform-cost search on a graph. The algorithm is identical to the general graph search algorithm in Figure 3.7, except for the use of a priority queue and the addition of an extra check in case a shorter path to a frontier state is discovered. The data structure for *frontier* needs to support efficient membership testing, so it should combine the capabilities of a priority queue and a hash table.

Both of these modifications are given in the example shown in Figure 3.15, where the problem is to get from Sibiu to Bucharest.

- The successors of Sibiu are Rimnicu Vilcea and Fagaras, with costs 80 and 99, respectively.
- The least-cost node, Rimnicu Vilcea, is expanded next, adding Pitesti with cost 80 + 97 =177.
- The least-cost node is now Fagaras, so it is expanded, adding Bucharest with cost 99 + 211 = 310.
- Now a goal node has been generated; but uniform-cost search keeps going, choosing Pitesti for expansion and adding a second path to Bucharest with cost 80+97+101= 278.
- Now the algorithm checks to see if this new path is better than the old one: it is, so the old one is discarded. Bucharest, now with g-cost 278, is selected for expansion and the solution is returned.

4.6 Bidirectional search

- Bidirectional search algorithm runs two simultaneous searches, one form initial state called as forward-search and other from goal node called as backward-search, to find the goal node.
- Bidirectional search replaces one single search graph with two small subgraphs in which one starts the search from an initial vertex and other starts from goal vertex. The search stops when these two graphs intersect each other.

Properties: Whether the algorithm is complete/optimal depends on the search strategies in both searches.

1. **Time complexity:** O(b d/2) (Assume BFS is used) Checking node for membership in the other search tree can be done in constant time.
2. **Space complexity**: O(b d/2) (Assume BFS is used) At least one of the search tree must be kept in memory for membership checking.
3. **Optimal:** Bidirectional search is Optimal.

Advantages:

- Bidirectional search is fast.
- Bidirectional search requires less memory

Disadvantages:

- Implementation of the bidirectional search tree is difficult.
- In bidirectional search, one should know the goal state in advance.