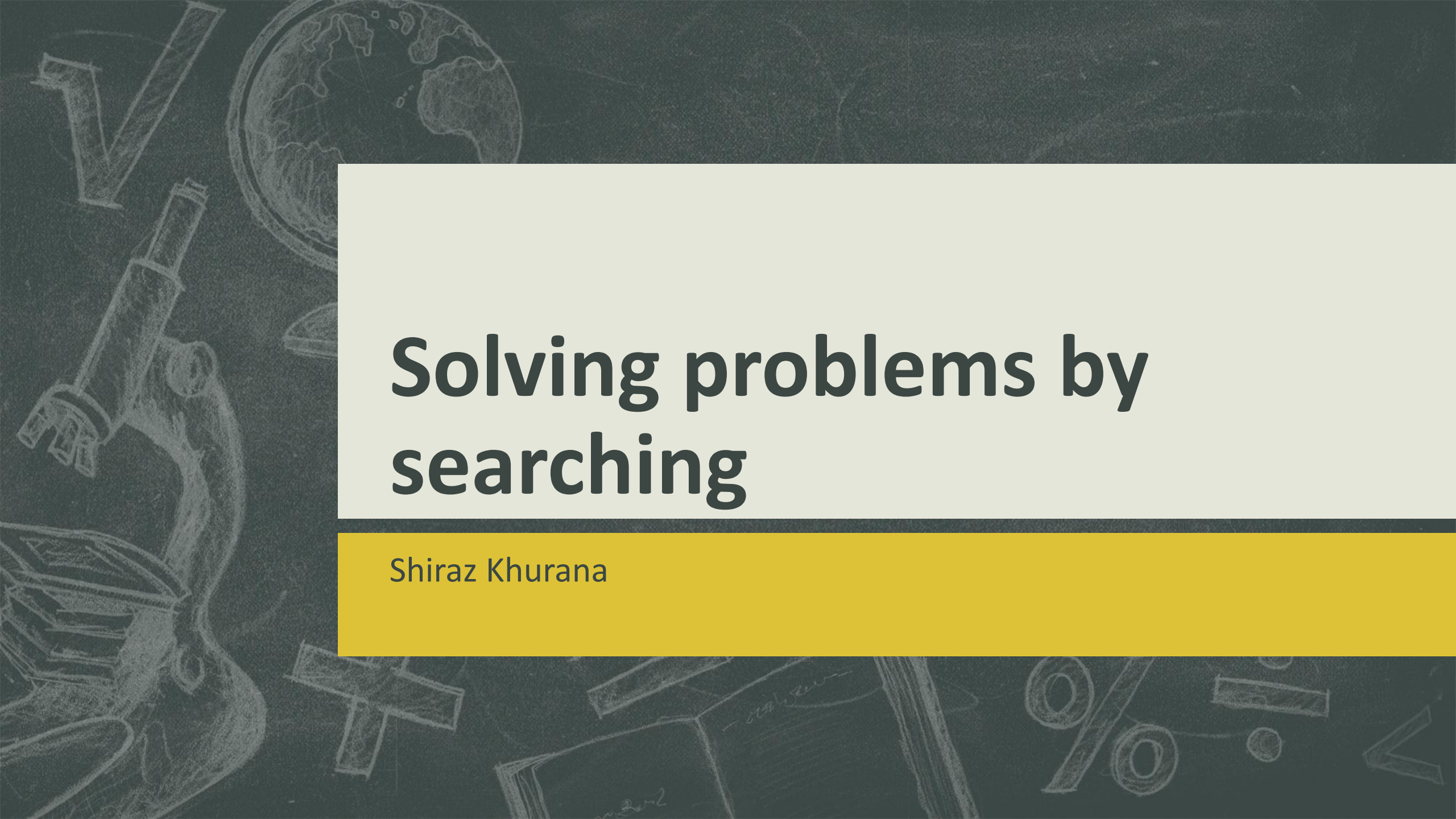


Solving problems by searching

Shiraz Khurana

Problem Statement

- Imagine an agent in the city of Arad, Romania, enjoying a touring holiday.
- The agent's performance measure: it wants to improve its suntan, improve its Romanian, take in the sights, enjoy the nightlife, avoid hangovers.
- The decision problem is a complex one involving many tradeoffs and careful reading of guidebooks.
- Suppose the agent has a nonrefundable ticket to fly out of Bucharest the following day.
- In that case, it makes sense for the agent to adopt the goal of getting to Bucharest.
- Courses of action that don't reach Bucharest on time can be rejected without further consideration and the agent's decision problem is greatly simplified.

Problem Solving Agent

- Goals help organize behavior by limiting the objectives that the agent is trying to achieve.
- Goal formulation : It is the first step in problem solving and based on the current situation and the agent's performance measure.
- A goal to be a set of world states—exactly those states in which the goal is satisfied.
- The agent's task is to find out how to act, now and in the future, so that it reaches a goal state. it needs to decide what sorts of actions and states it should consider.
- Problem formulation is the process of deciding what actions and states to consider, given a goal.

Problem Solving Agent

- 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 environment is **observable**, so the agent always knows the current state. For the agent driving in Romania, it's reasonable to suppose that each city on the map has a sign indicating its presence to arriving drivers.
- The environment is **discrete**, so at any given state there are only finitely many actions to choose from. This is true for navigating in Romania because each city is connected to a small number of other cities.
- We will assume the environment is **known**, so the agent knows which states are reached by each action.
- the environment is **deterministic**, so each action has exactly one outcome.

Problem Solving Agent

- 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.
- “formulate, search, execute” design for the agent
- After formulating a goal and a problem to solve, the agent calls a search procedure to solve it.

Problem Solving Agent

```
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
```

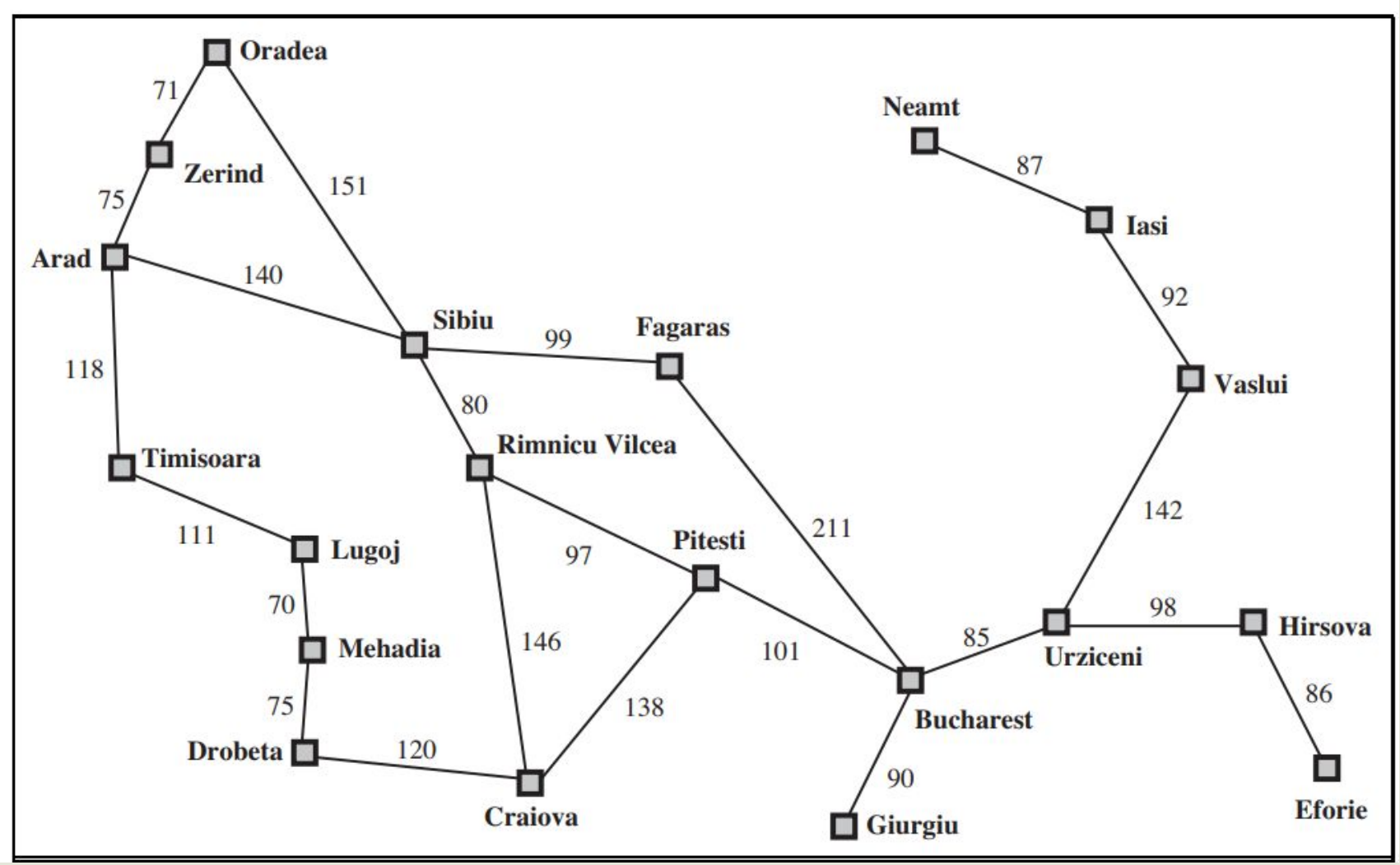
Well-defined problems and solutions

- A problem can be defined formally by five components:
- The initial state that the agent starts in. For example, the initial state for our agent in Romania might be described as In(Arad).
- 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. For example, from the state In(Arad), the applicable actions are {Go(Sibiu), Go(Timisoara), Go(Zerind)}.
- A description of what each action does; RESULT(s, a) that returns the state that results from doing action a in state s. For example, RESULT(In(Arad),Go(Zerind)) = In(Zerind) .
  - 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.

Well-defined problems and solutions

- The goal test, 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)}.
- 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. the cost of a path can be described as the sum of the costs of the individual actions along the path.
- The elements that defines a problem and can be gathered into a single data structure that is given as input to a problem-solving algorithm.
- A solution to a problem is an action sequence that leads 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.

A simplified road map of part of Romania.



Example Problem

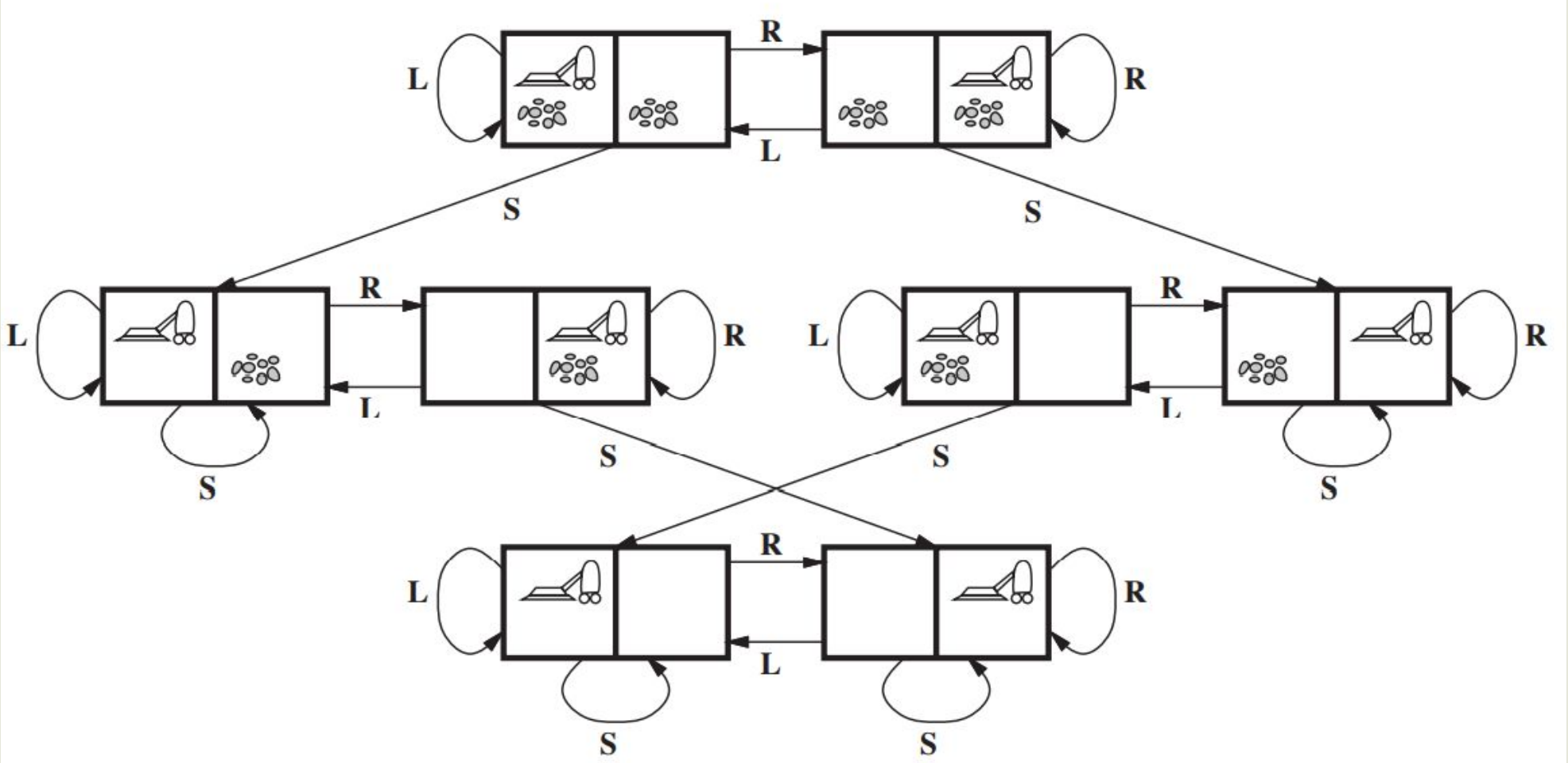
A toy problem is intended to illustrate or exercise various problem-solving methods. It can be given a concise, exact description and hence is usable by different researchers to compare the performance of algorithms.

A real-world problem is one whose solutions people care about. Such problems tend not to have a single agreed-upon description.

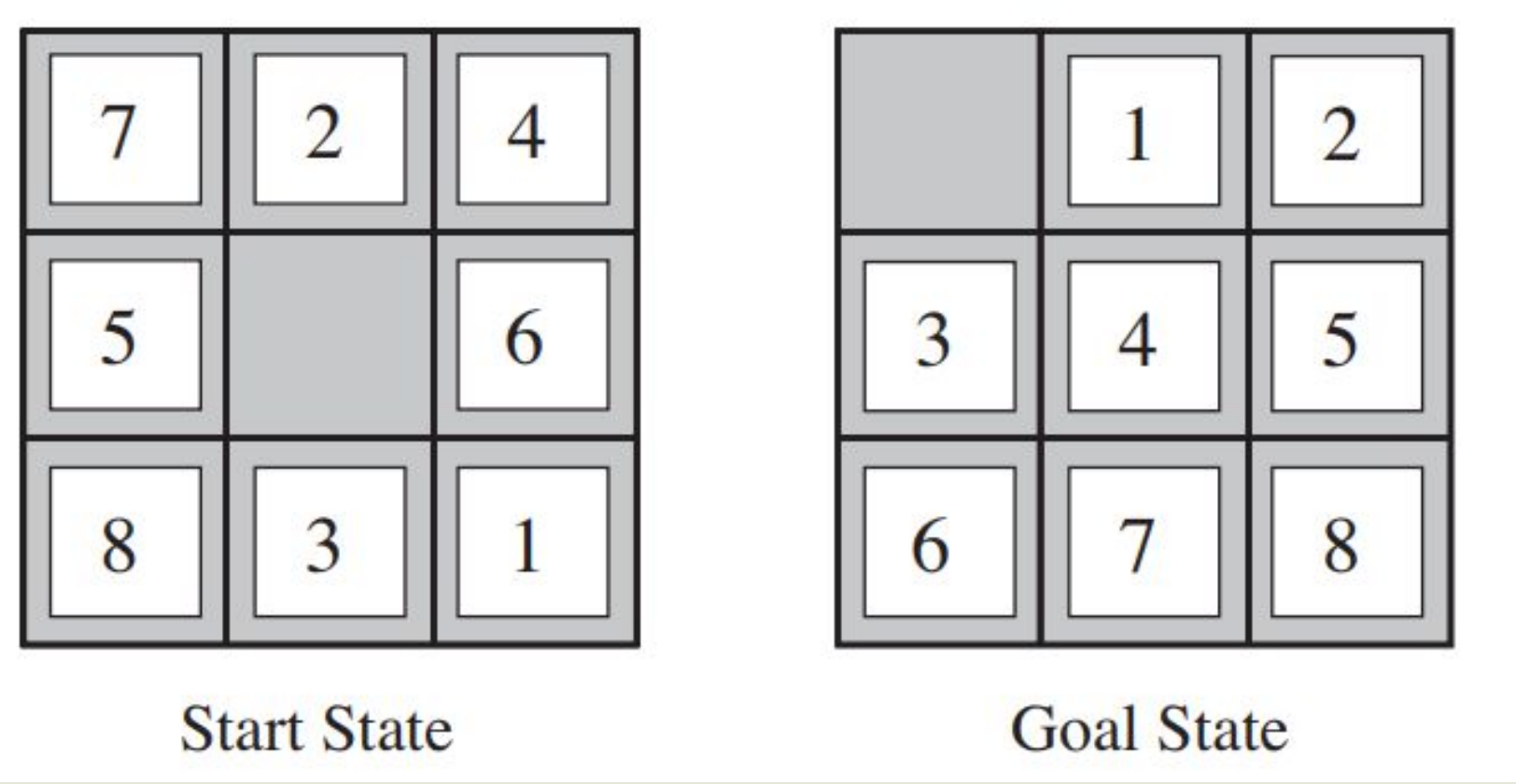
Toy Problem 1 : Vacuum Cleaner

- 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 $n \cdot 2n$ 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.
- Transition model: The actions have their expected effects, except that moving Left in the leftmost square, moving Right in the rightmost square, and Sucking in a clean square have no effect.
- 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.

Toy Problem 1 : Vacuum Cleaner



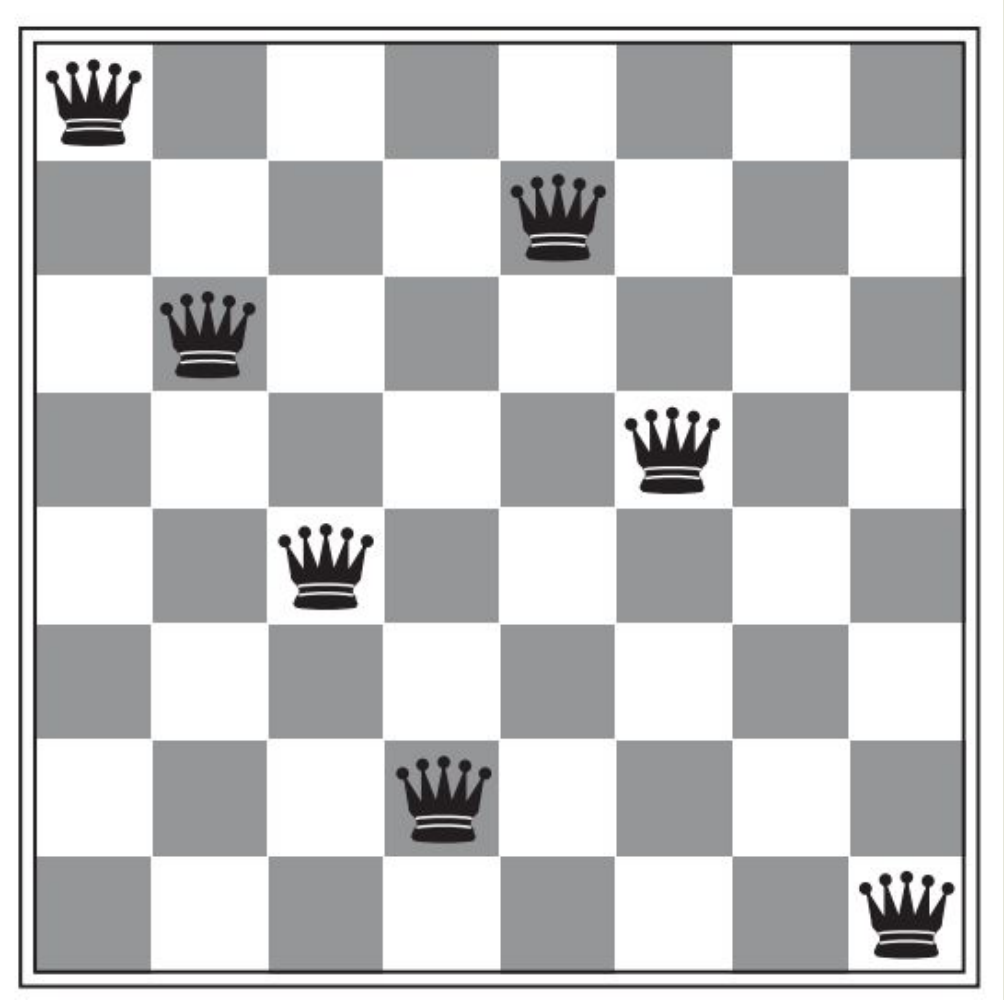
Toy Problem 2: 8-puzzle



Toy Problem 2: 8-puzzle

- 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.
- Actions: The simplest formulation defines the actions as movements of the blank space Left, Right, Up, or Down.
- Transition model: Given a state and action, this returns the resulting state.
- 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.

Toy Problem 3: 8-Queen



Toy Problem 3: 8-Queen

- 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.

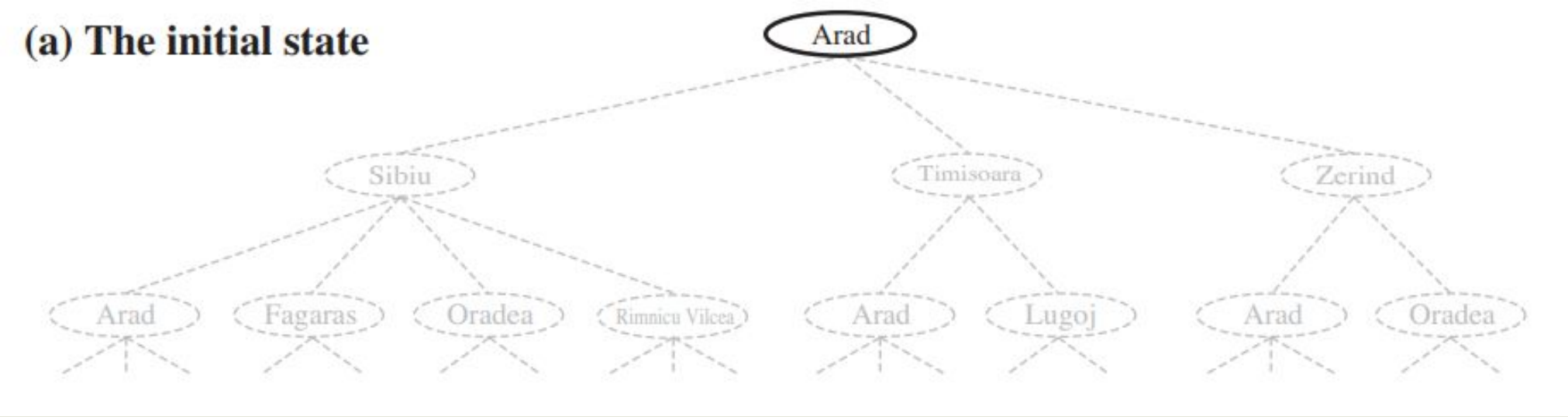
Real World Problem : Airline Travel Problems

- States: Each state obviously includes a location (e.g., an airport) and the current time.
- 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.

Searching for solutions

- 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.
- 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.

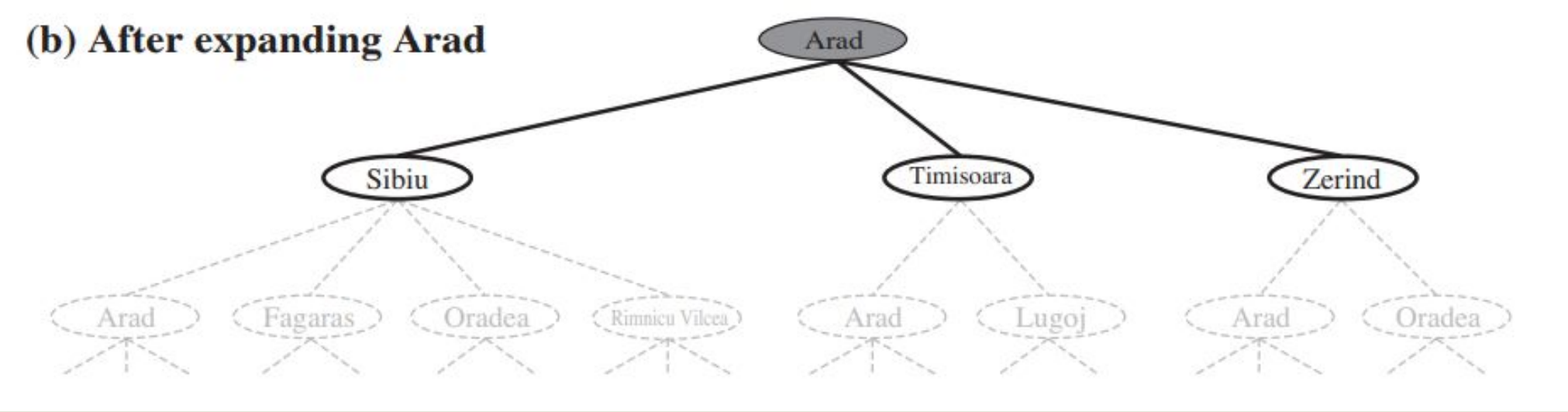
Tree search example



Searching for solutions

- We do this by expanding the current state; that is, applying each legal action to the current state, thereby generating a new set of states.
- we add three branches from the parent node In(Arad) leading to three new child nodes: In(Sibiu), In(Timisoara), and In(Zerind).

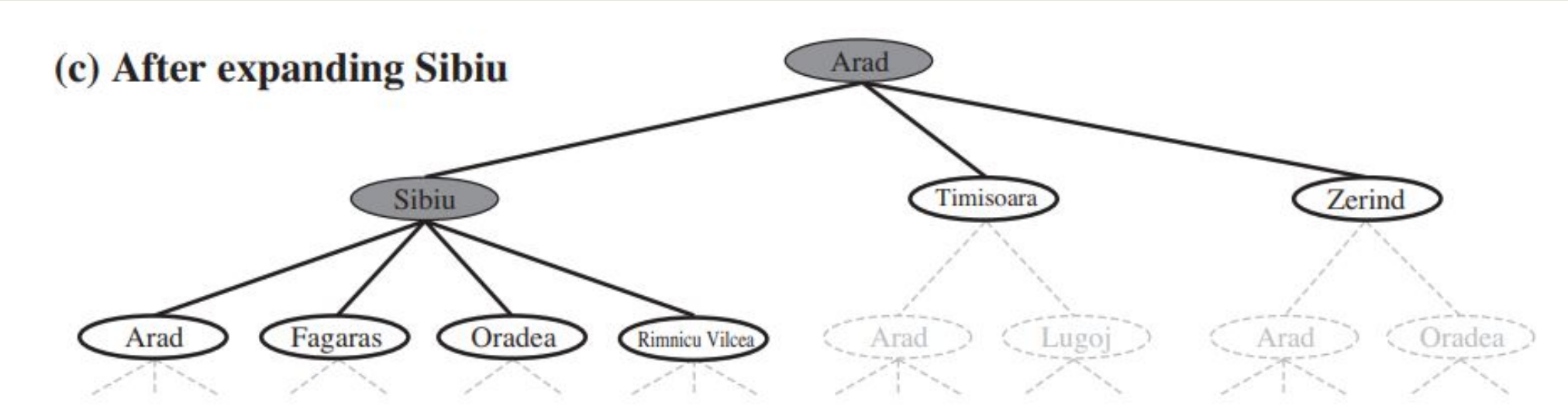
Tree search example



Searching for solutions

- 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(RimnicuVilcea). We can then choose any of these four or go back and choose Timisoara or Zerind.
- Each of these six nodes is a leaf node, that is, a node with no children in the tree. The set of all leaf nodes available for expansion at any given point is called the frontier.

Tree search example



Tree search example

```
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
```

How to Avoid These Loops

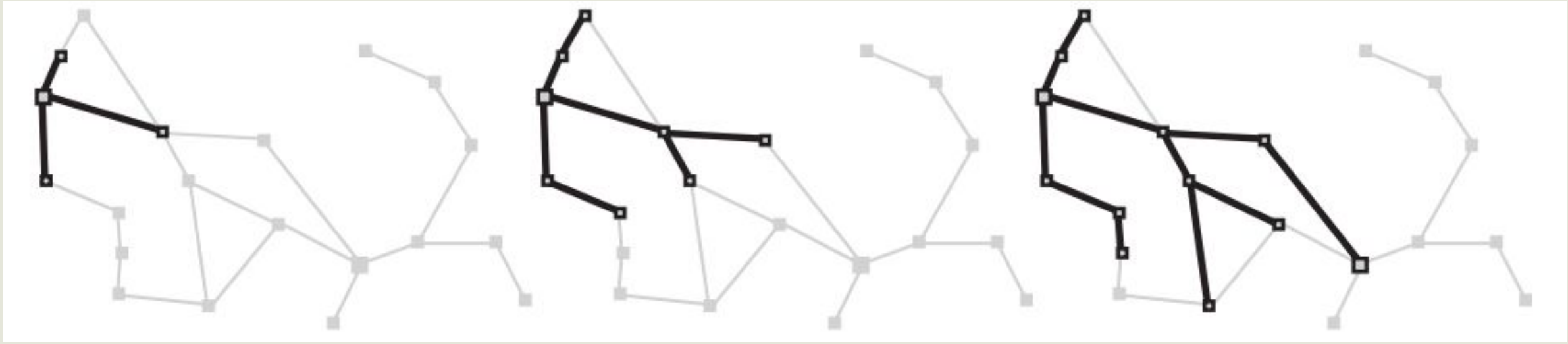
- we augment the TREE-SEARCH algorithm with a data structure called the explored set which remembers every expanded node.
- Newly generated nodes that match previously generated nodes—ones in the explored set or the frontier—can be discarded instead of being added to the frontier.

Graph Search

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

Graph Search Tree

- the search tree constructed by the GRAPH-SEARCH algorithm contains at most one copy of each state, so it is growing a tree directly on the state-space graph.



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.

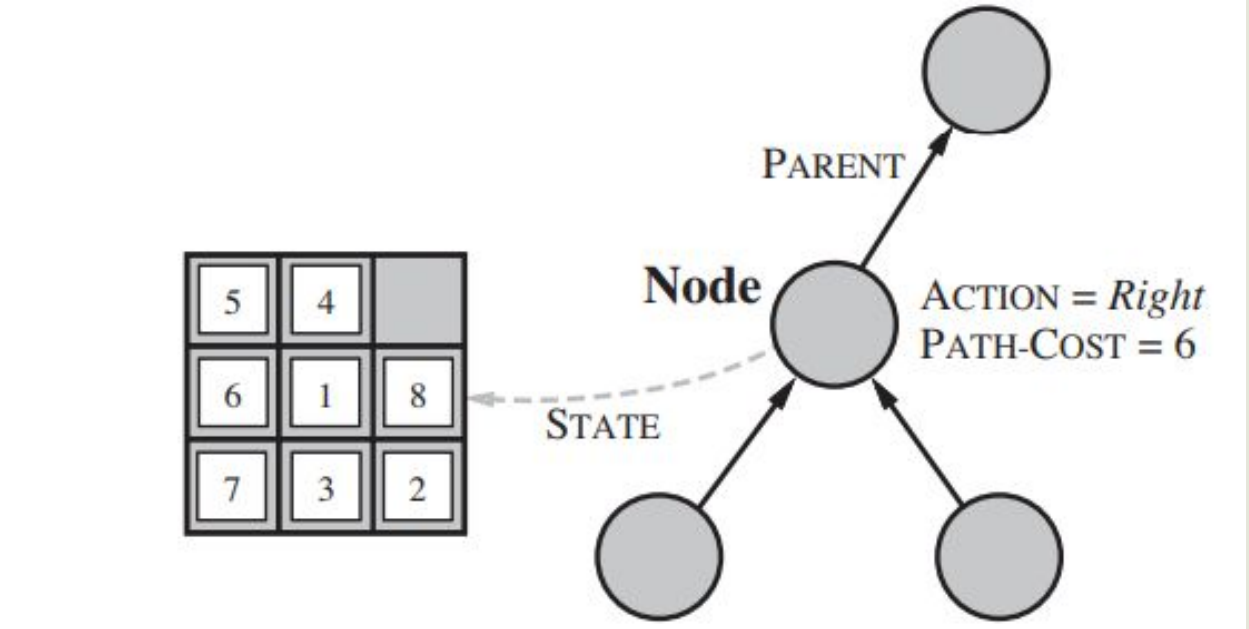
Infrastructure for search algorithms

- Given the components for a parent node, it is easy to compute the necessary components for a 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)
```

Infrastructure for search algorithms

- 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.
- **A node is a bookkeeping data structure used to represent the search tree.**
- **A state corresponds to a configuration of the world.**
- **Nodes are on particular paths, as defined by PARENT pointers, whereas states are not.**
- **Two different nodes can contain the same world state if that state is generated via two different search paths.**



Frontier Style

- 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.
- Queues are characterized by the order in which they store the inserted nodes. Three common variants are the first-in, first-out or FIFO queue, which pops the oldest element of the queue;
- the last-in, first-out or LIFO queue (also known as a stack), which pops the newest element of the queue.
- The priority queue, which pops the element of the queue with the highest priority according to some ordering function.

Measuring problem-solving performance

- an algorithm's performance can be evaluated 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 & Space Complexity

- The 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);
- 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

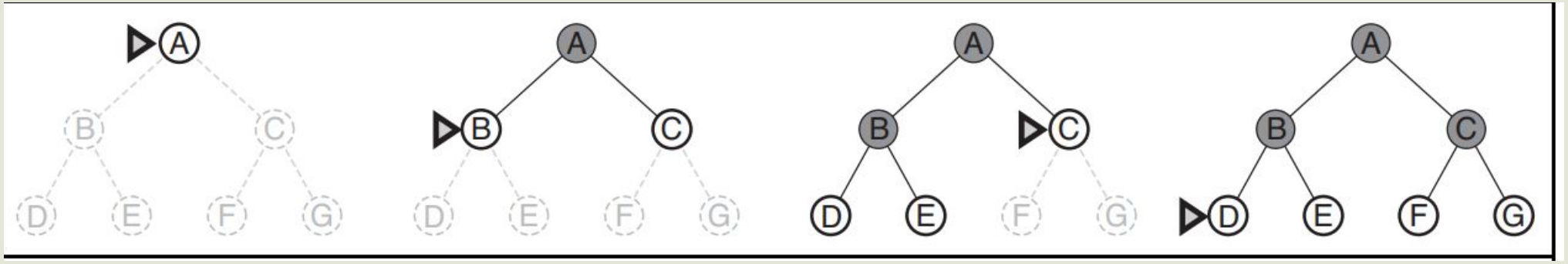
Uninformed Search Strategies

- The term 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.
- 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.
- All the nodes are expanded at a given depth in the search tree before any nodes at the next level are expanded.
- the frontier is utilized in FIFO fashion. The new nodes (which are always deeper than their parents) go to the back of the queue, and old nodes, which are shallower than the new nodes, get expanded first.
- **Breadth-first search always has the shallowest path to every node on the frontier**

Breadth-first search



Breadth-first search

```
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)
```

Breadth First Search Performance

it is **complete**—if the shallowest goal node is at some finite depth d, breadth-first search will eventually find it after generating all shallower nodes (provided the branching factor b is finite).

The shallowest goal node need not compulsorily be the optimal goal node. BFS is **optimal** if the path cost is a non-decreasing function of d(depth) of the node. The most common use of BFS when all actions have the same cost.

Breadth First Search Time Complexity

- In searching a uniform tree where every state has b successors. The root of the search tree generates b nodes at the first level, each of which generates b more nodes, for a total of b^2 at the second level. Each of these generates b more nodes, yielding b^3 nodes at the third level, and so on. Now suppose that the solution is at depth d. In the worst case, it is the last node generated at that level. Then the total number of nodes generated is
- $b + b^2 + b^3 + \cdots + b^d = O(b^d)$.
- the goal test to nodes when selected for expansion, rather than when generated.

Breadth First Search Space Complexity

- Graph search stores every expanded node in the explored set, the space complexity is always within a factor of b of the time complexity. For breadth-first graph search in particular, every node generated remains in memory. There will be $O(bd-1)$ nodes in the explored set and $O(bd)$ nodes in the frontier,

Issues with BFS

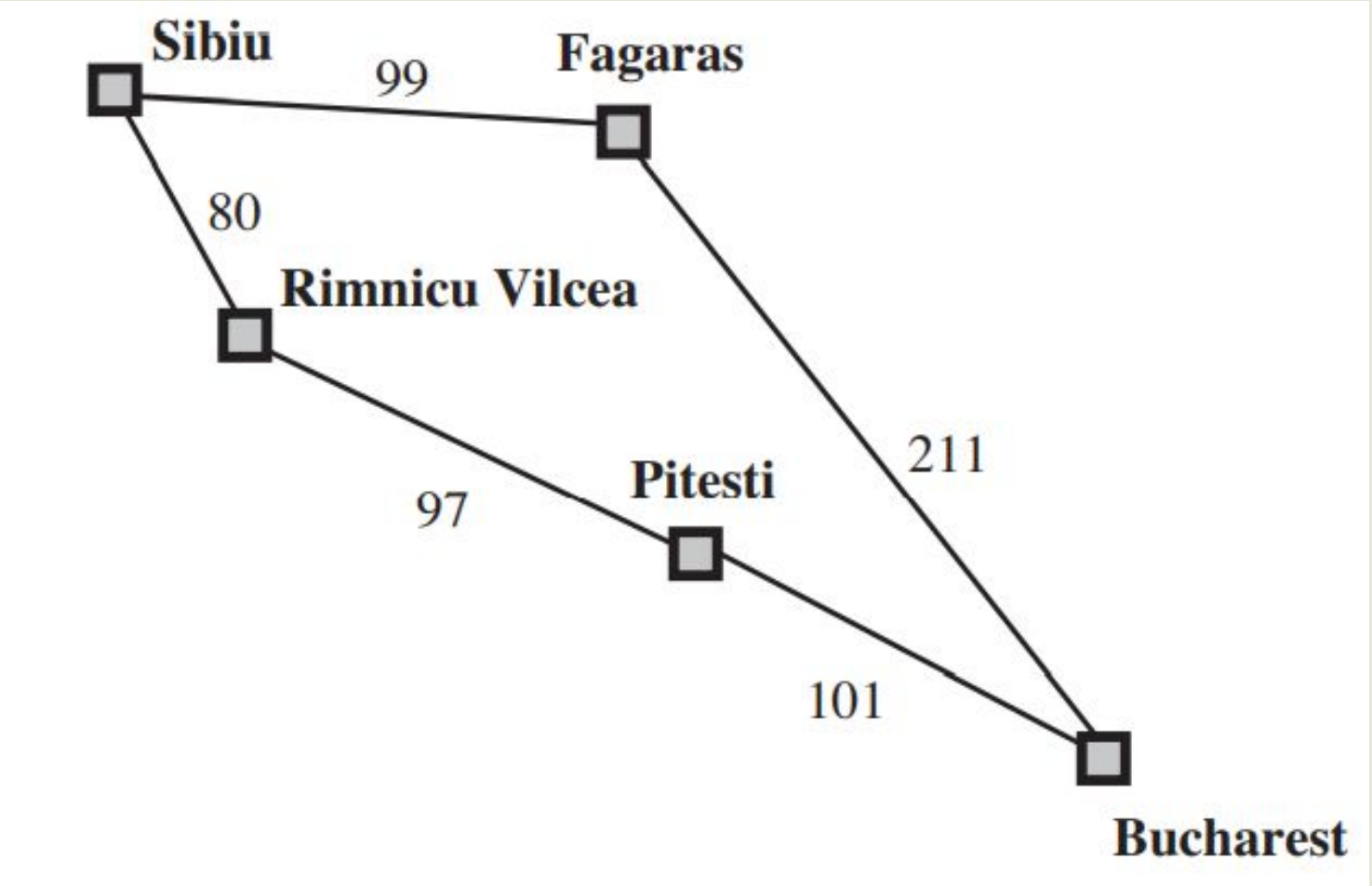
Depth	Nodes	Time	Memory
2	110	.11 milliseconds	107 kilobytes
4	11,110	11 milliseconds	10.6 megabytes
6	10^6	1.1 seconds	1 gigabyte
8	10^8	2 minutes	103 gigabytes
10	10^{10}	3 hours	10 terabytes
12	10^{12}	13 days	1 petabyte
14	10^{14}	3.5 years	99 petabytes
16	10^{16}	350 years	10 exabytes

Figure 3.13 Time and memory requirements for breadth-first search. The numbers shown assume branching factor $b = 10$; 1 million nodes/second; 1000 bytes/node.

Uniform-Cost Search

- UCS algorithm 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 g(n). This is done by storing the frontier as a priority queue ordered by g.
- In addition to the ordering of the queue by path cost, there are two other significant differences from BFS.
- 1) 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.
- 2) a test is added in case a better path is found to a node currently on the frontier

Uniform Cost Search



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
```

Uniform Cost Search Performance

- **Uniform-cost search is optimal in general.**
- uniform-cost search selects a node n for expansion, the optimal path to that node has been found.
- Because step costs are nonnegative, paths never get shorter as nodes are added.
- These two facts together imply that uniform-cost search expands nodes in order of their optimal path cost. Hence, the first goal node selected for expansion must be the optimal solution.

Uniform Cost Search Performance

- 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 there is a path with an infinite sequence of zero-cost actions. for example, a sequence of NoOp actions.
- **Completeness is guaranteed provided the cost of every step exceeds some small positive constant .**

Uniform Cost Search Performance

- let C^* be the cost of the optimal solution and assume that every action costs at least . Then the algorithm's worst-case time and space complexity is $O(b^{1+C*/\epsilon})$, which can be much greater than b^d.
- This is because uniform cost search can explore large trees of small steps before exploring paths involving large and perhaps useful steps.
- When all step costs are equal, $O(b^{1+C*/\epsilon})$, is just b^{d+1}. When all step costs are the same, uniform-cost search is like BFS,
- BFS stops as soon as it generates a goal, whereas UCS examines all the nodes at the goal's depth to see if one has a lower cost;
- **uniform-cost search does strictly more work by expanding nodes at depth d unnecessarily.**