Artificial Intelligence

Dr. Qaiser Abbas
Department of Computer Science & IT
University of Sargodha
qaiser.abbas@uos.edu.pk

3 SOLVING PROBLEMS BY SEARCHING

- *In which we see how an agent can find a sequence of actions that achieves its goals when no single action will do.*

3.1 PROBLEM-SOLVING AGENTS

- Imagine an agent in the city of Arad, Romania, enjoying a touring holiday.
  - **Performance measure** contains many factors: it wants to improve its suntan, improve its Romanian, take in the sights, enjoy the nightlife, avoid hangovers, and so on.
  - **Current State**: Now, suppose the agent has a nonrefundable ticket to fly out from Bucharest the following day.
  - **Goal**: Reaching at Bucharest on tomorrow.

3.1 PROBLEM-SOLVING AGENTS

- **Goal formulation** is the first step in problem solving, based on the current situation and the agent's performance measure.
  - Goals help organize behavior by limiting the objectives that the agent is trying to achieve (performance measures) and hence the actions it needs to consider.

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

3.1 PROBLEM-SOLVING AGENTS

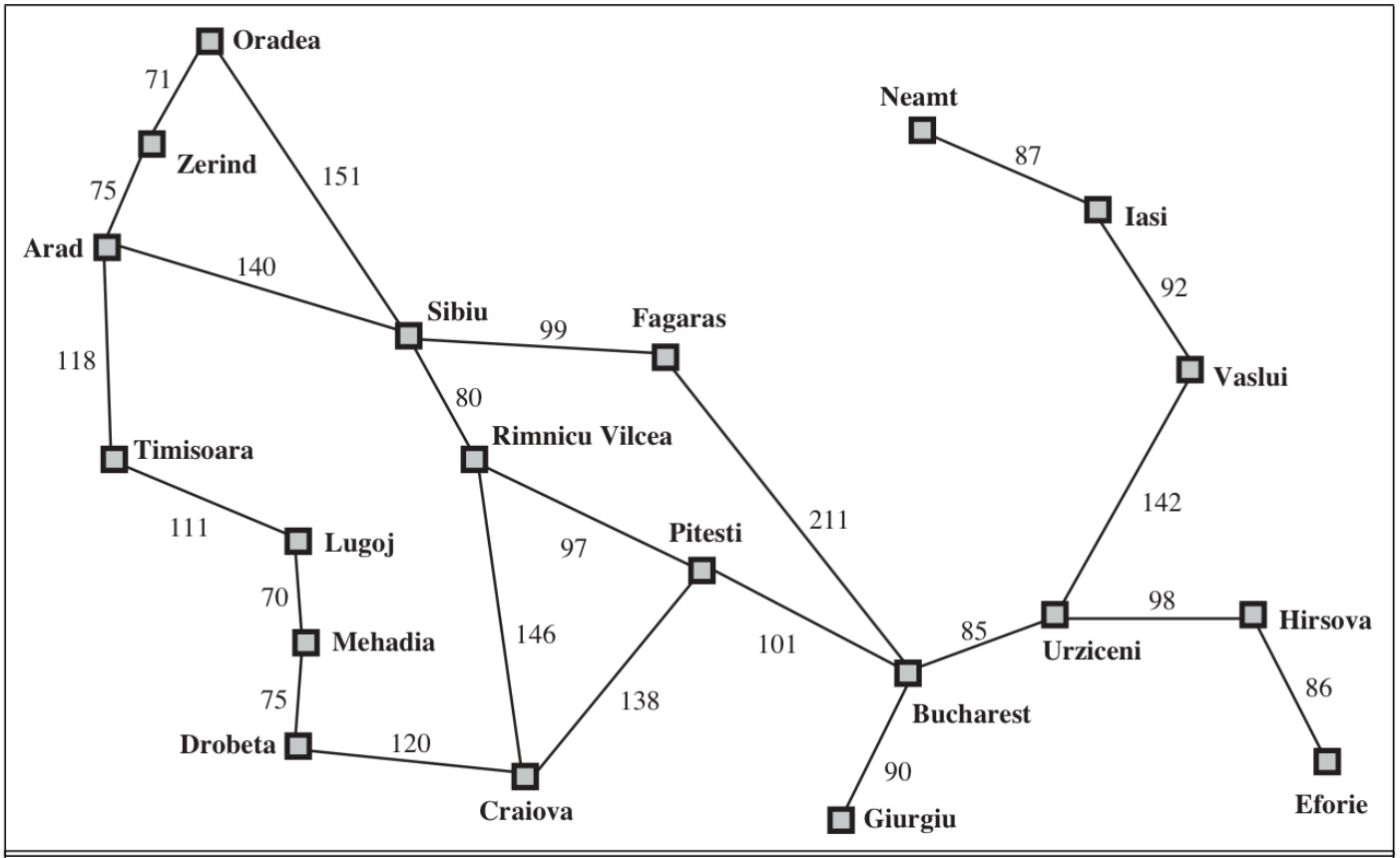


Figure 3.2 A simplified road map of part of Romania.

3.1 PROBLEM-SOLVING AGENTS

- **Discussion at Romanian Example**:
  - Driving from one town to another are the actions and towns are the states.
  - Three roads lead out of Arad, one toward Sibiu, one to Timisoara, and one to Zerind. None of these achieves the goal Bucharest.
  - If the agent has no additional information—i.e., if the environment is **unknown** —then it has no choice but to try one of the actions at random.
  - If the agent has the map of Romania (additional info.) then it can examine future actions that can lead to Bucharest.

3.1 PROBLEM-SOLVING AGENTS

- In **observable** environment, agent knows the current state like each city on the map has a sign.
- In **discrete** environment, finite actions are available at a given state like each city is connected to a small number of other cities.
- In **known**, agent knows which states are reached by each action.
  - Having an accurate map fulfills this condition.
- In **deterministic**, each action has exactly one outcome.
  - Under ideal conditions, drive from Arad to Sibiu ends up in Sibiu. Of course, conditions are not always ideal (See Chapter 4)

3.1 PROBLEM-SOLVING AGENTS

- Solution *to any problem is a* fixed sequence of actions.
  - It could be a branching strategy which leads to different actions on different percepts like in driving from Arad to Sibiu and then to Rimnicu Vilcea, it can arrive in Zerind by accident instead of Sibiu.
- If the environment is known and deterministic, then agent knows the states after the first action, second action, and so on, respectively.

3.1 PROBLEM-SOLVING AGENTS

- Looking for a sequence of actions (solution) that reaches the goal is called search.
- Then these actions can be carried out. This is called the execution phase.
- This "**formulate, search, execute**" concludes the design for the agent, as shown in Figure 3.1.
- During execution, it ignores its percepts when choosing an action because it knows in advance what they will be.

- These are called an open-loop system, because ignoring the percepts breaks the loop between agent and environment.

3.1 PROBLEM-SOLVING AGENTS

```
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
```

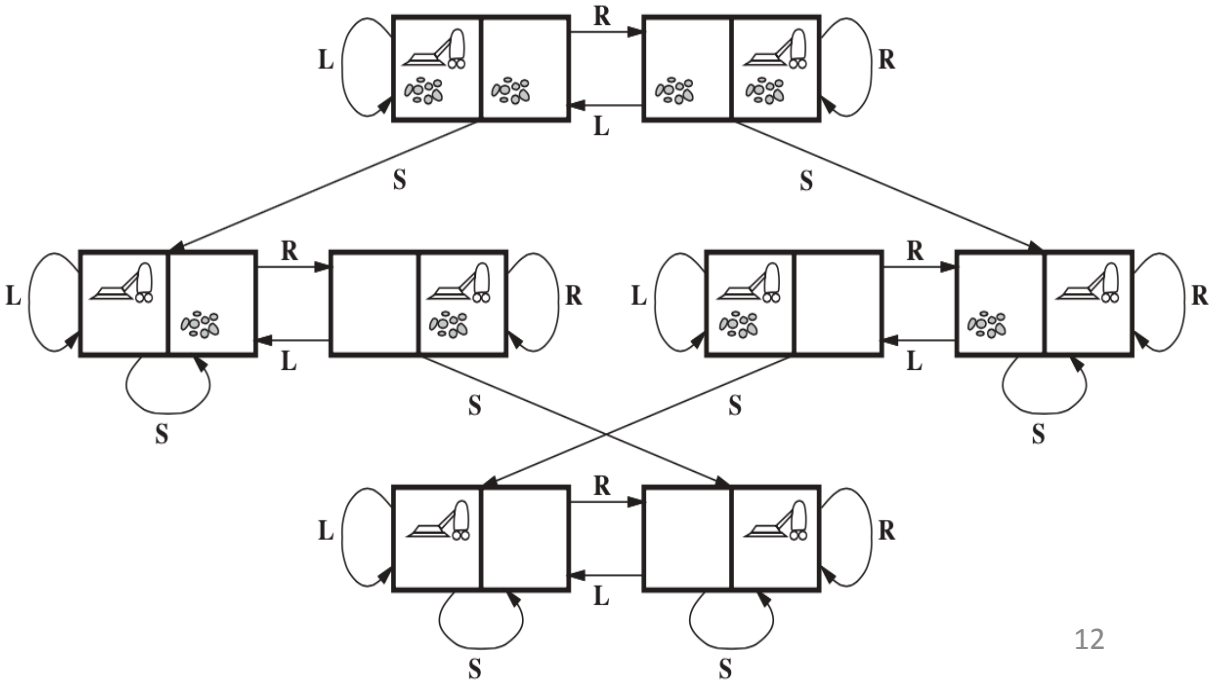
Figure 3.1 A simple problem-solving agent. It first formulates a goal and a problem, searches for a sequence of actions that would solve the problem, and then executes the actions one at a time. When this is complete, it formulates another goal and starts over.

3.1.1 Well Defined Problems

- A problem is defined by five components:
  - **Initial state**: e.g., In(Arad)
  - **Actions:** Actions(s) = set of actions for state s
    e.g., Actions(In(Arad)) = {Go(Sibiu), Go(Timisoara), Go(Zerind)}
  - **Transitions or States:** Result(s, a) = the successor state
    e.g., Result(In(Arad), Go(Zerind)) = In(Zerind)
  - **Goal test:**, can be an explicit set of states, e.g., {In(Bucharest)} or an implicit property, e.g., checkmate in chess
  - **Path cost:** is the sum of the step costs C(s, a, s')
    e.g., sum of distances, number of actions executed, etc.
- Initial state, actions, and transition model implicitly define the state space of the problem.
- Abstracting (removing details) the states (the traveling companions, etc.) and actions' description (turning on the radio, etc.) is necessary.

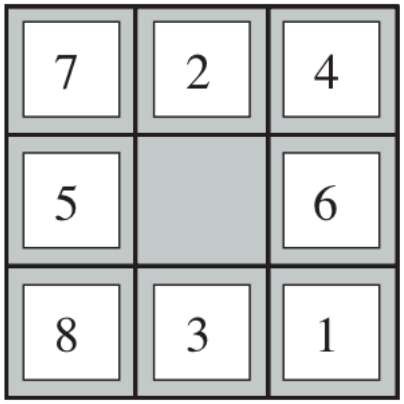
3.2 EXAMPLE PROBLEM DEFINITION
Vacuum Cleaner World

- **States**??: dirt and robot locations (ignore dirt amounts etc.)
- **Initial state**??: any state
- **Actions**??: Left, Right, Suck, NoOp
- **Goal test**??: no dirt in any location
- **Path cost**??: 1 per action (0 for NoOp)

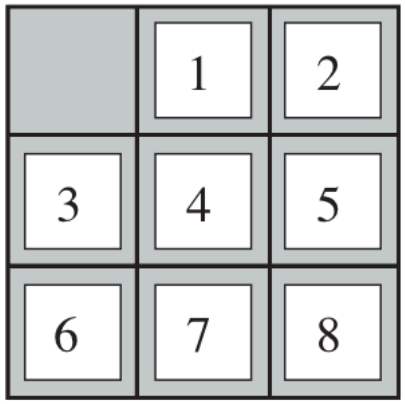


3.2 EXAMPLE PROBLEM DEFINITION
8-Puzzle

- **States**??: a 3×3 matrix of integers
- **Initial state**??: any state
- **Actions**??: move the blank space: left, right, up, and down
- **Goal test**??: equal to given goal state
- **Path cost**??: 1 per move



Start State



Goal State

3.2 EXAMPLE PROBLEMS
8-Queen

- **Complete-state formulation:** Starts with all 8 queens on the board.
  - **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.
  - **Goal test**: 8 queens are on the board, none attacked.
  - **Path Cost**: 1 per move.

✓ We have $64 \cdot 63 \cdots 57 \approx 1.8 \times 10^{14}$ possible sequences to investigate.

- **Incremental formulation:** Starting with an empty state and each action adds a queen to the state.
  - **States**: 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, making sure that no queen is attacked

✓ State space reduced from 1.8×10^{14} to just 2,057, and solutions are easy to find.

3.2 EXAMPLE PROBLEMS
8-Queen

● **Table 5.1** An illustration of how much checking is saved by backtracking in the *n*-Queens problem*

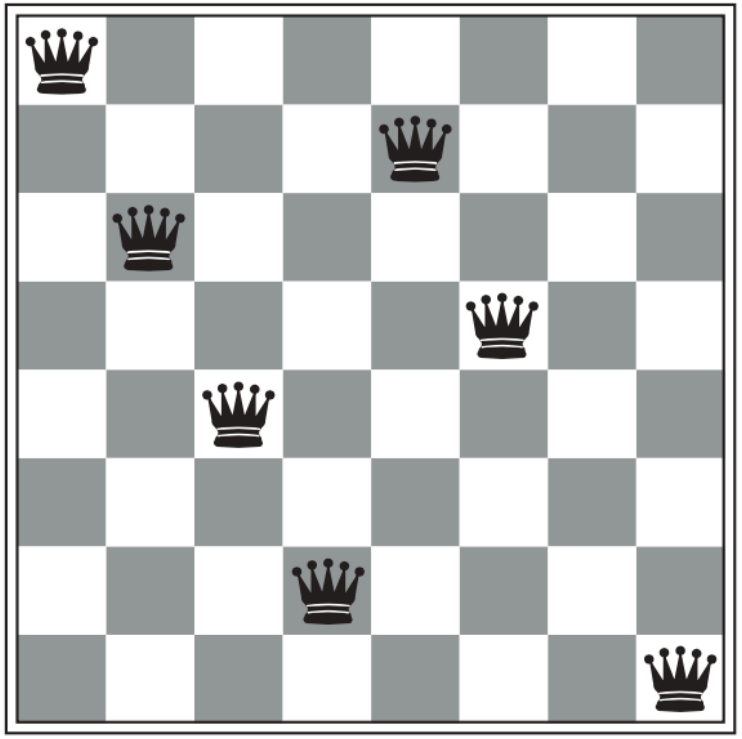
n	Number of Nodes Checked by Algorithm 1†	Number of Candidate Solutions Checked by Algorithm 2‡	Number of Nodes Checked by Backtracking	Number of Nodes Found Promising by Backtracking
4	341	24	61	17
8	19,173,961	40,320	15,721	2057
12	9.73×10^{12}	4.79×10^{8}	1.01×10^{7}	8.56×10^{5}
14	1.20×10^{16}	8.72×10^{10}	3.78×10^{8}	2.74×10^{7}

*Entries indicate numbers of checks required to find all solutions.

†Algorithm 1 does a depth-first search of the state space tree without backtracking.

‡Algorithm 2 generates the $n!$ candidate solutions that place each queen in a different row and column.

Find a Solution for 8-Queen Problem



3.2 EXAMPLE PROBLEMS

- **3.2.2 Real-world problems (**Airline Travel Website**)**
  - **States**: A location (e.g., an airport) and the current time. The state must record extra information about "historical" aspects like fare bases, domestic, international, etc..
  - **Initial state**: 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.
  - **Goal test**: 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.

3.2 EXAMPLE PROBLEMS

(Read it by yourself)

- T**raveling salesperson problem:** Each city must be visited exactly once. The aim is to find the *shortest* tour.
- **VLSI layout:** problem requires positioning millions of components and connections on a chip to minimize area, circuit delays, stray capacitances, and maximize manufacturing yield.
- **Robot navigation:** is a generalization of the route-finding with an infinite set of possible actions and states.
- **Automatic assembly:** Sequencing of complex objects by a robot and so on.

3.1 PROBLEM-SOLVING AGENTS

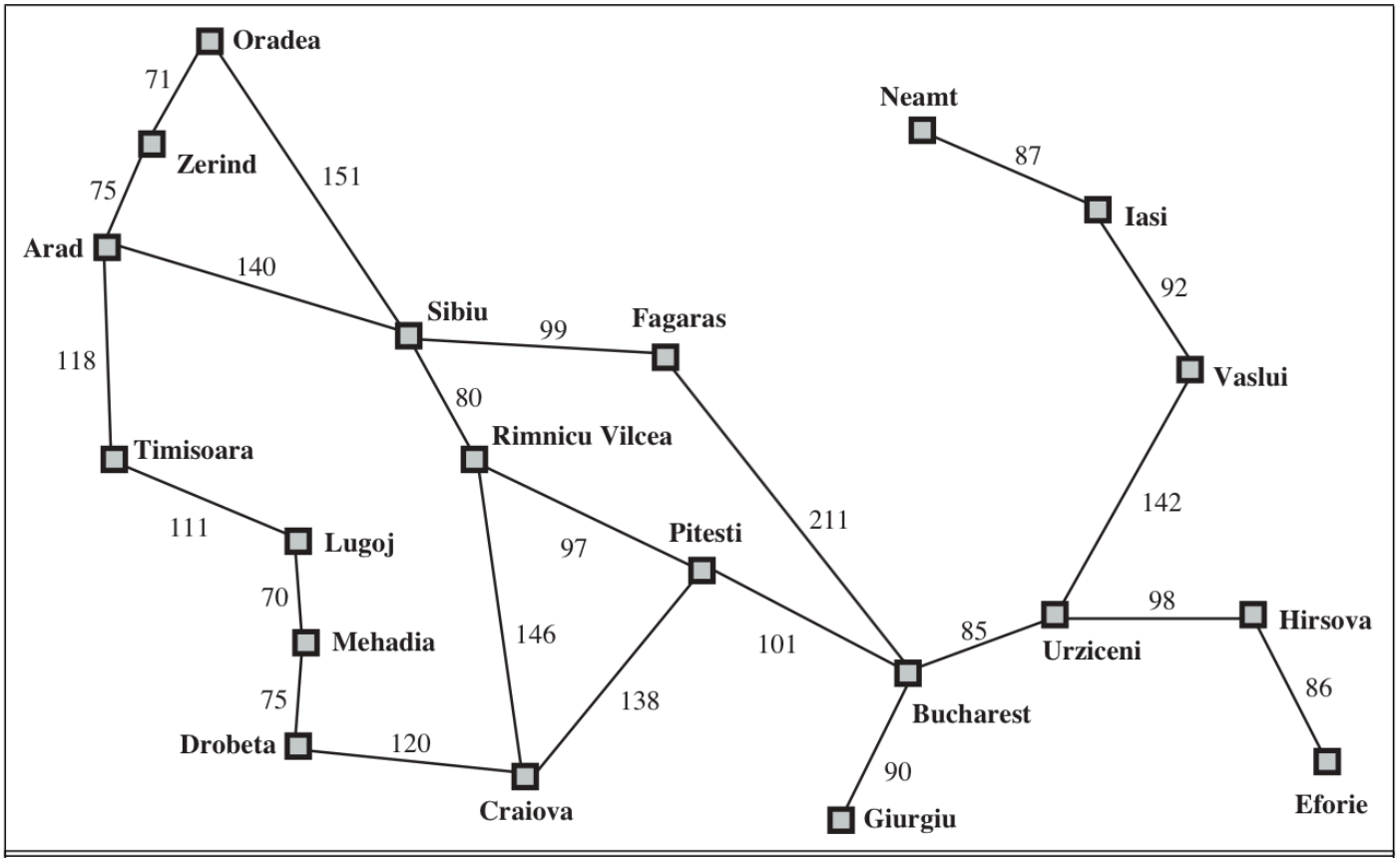


Figure 3.2 A simplified road map of part of Romania.

3.3 SEARCHING FOR SOLUTIONS

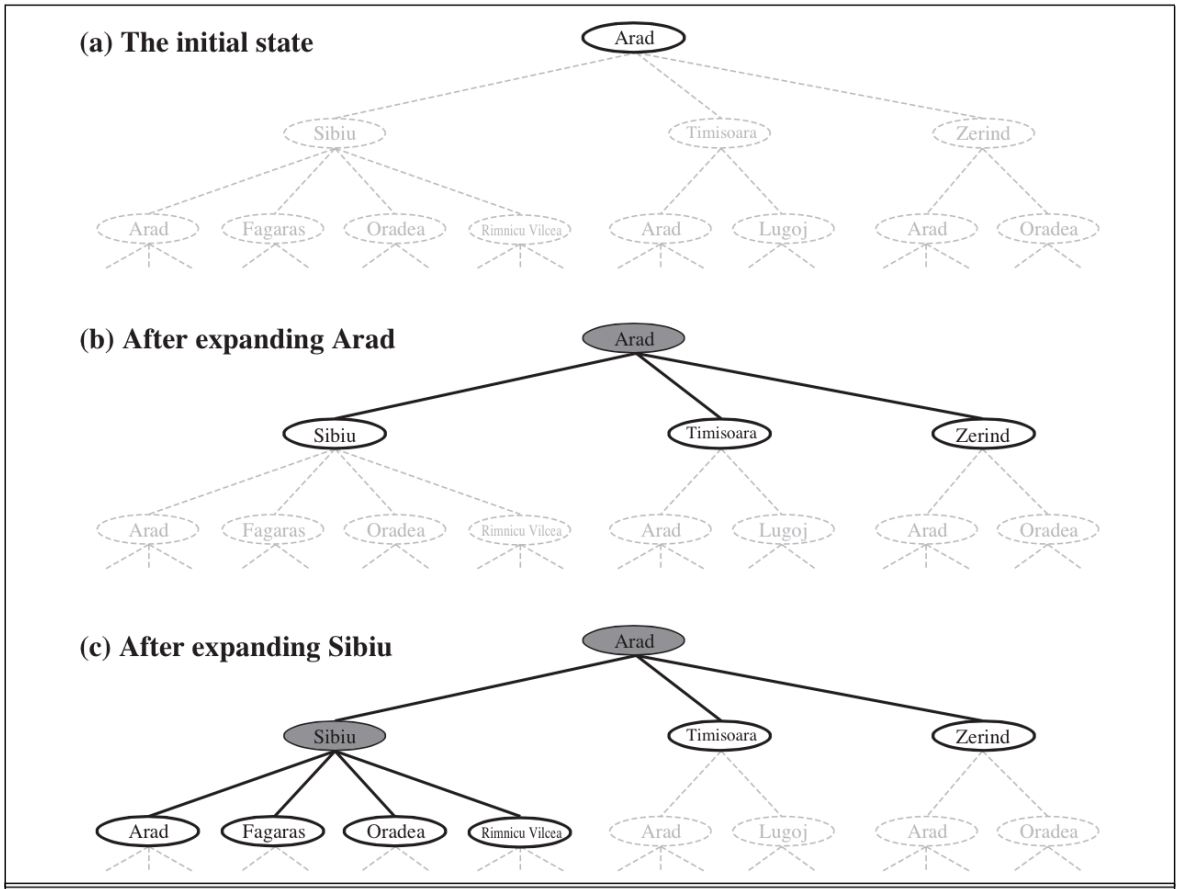


Figure 3.6 Partial search trees for finding a route from Arad to Bucharest. Nodes that have been expanded are shaded; nodes that have been generated but not yet expanded are outlined in bold; nodes that have not yet been generated are shown in faint dashed lines.

3.3 SEARCHING FOR SOLUTIONS

- Root node corresponds to the initial state, *In(Arad)*.
- 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.")
- Expanding the current state; thereby generating a new set of states. Add three branches from the parent node *In(Arad)* leading to three new child nodes: *In(Sibiu), In(Timisoara),* and *In(Zerind)*.

Which of these three possibilities to consider further.

3.3 SEARCHING FOR SOLUTIONS

- This is the essence of search—following up one option now and putting the others aside for later, in case the first choice does not lead to a solution.

- 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.

- All leaf nodes are kept in frontier.

3.3 SEARCHING FOR SOLUTIONS

```
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
```

- Repeated states can happen in tree search **through loopy path** like from Sibiu to Arad again.
- Considering loopy paths (redundant paths) means the search tree is *infinite* and in other words no solution.
- Reaching the goal is possible after removing these redundant paths.

3.3 SEARCHING FOR SOLUTIONS

```
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
```

- *Algorithms that forget their history are doomed to repeat it.*
- Explored set to remember every expanded node.
- Newly generated nodes that match with ones in the explored set or the frontier can be discarded.

3.3 SEARCHING FOR SOLUTIONS

- Route- finding on a **rectangular grid** in Figure 3.9 is a particularly important example in computer games.

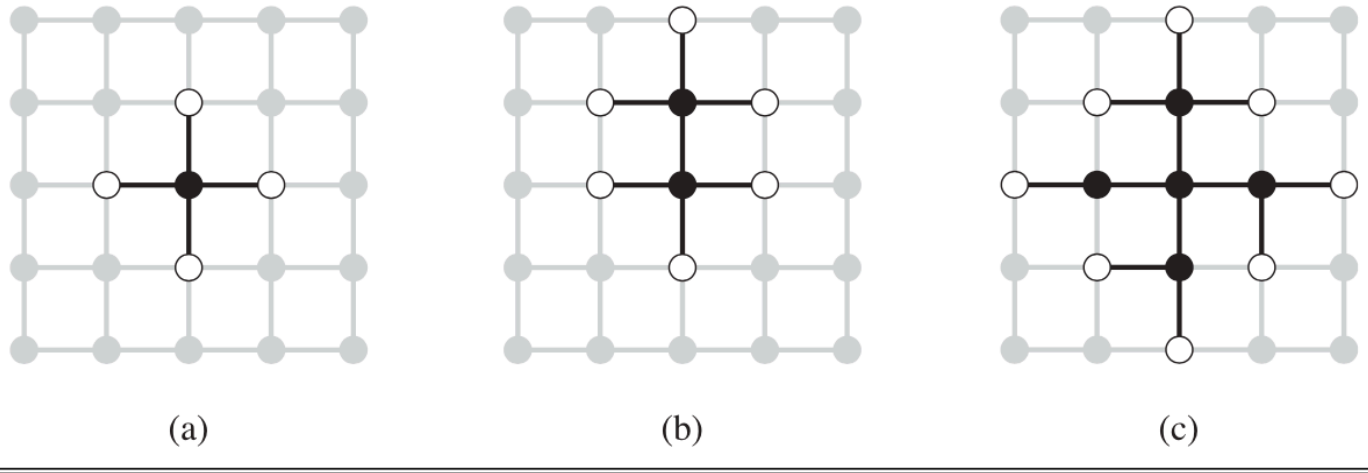


Figure 3.9 The separation property of GRAPH-SEARCH, illustrated on a rectangular-grid problem. The frontier (white nodes) always separates the explored region of the state space (black nodes) from the unexplored region (gray nodes). In (a), just the root has been expanded. In (b), one leaf node has been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

Search strategies

- A strategy is defined by picking the order of node expansion
- Strategies are evaluated along the following dimensions:
  - Completeness—does it always find a solution if one exists
  - Optimality—does it always find a least-cost solution?
  - Time complexity—number of nodes generated/expanded
  - Space complexity—maximum number of nodes in memory
- 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 (may be ∞)

3.4 UNINFORMED SEARCH STRATEGIES

- **Uninformed search** strategies use only the information available in the problem definition and have no additional information.
  - ♦ Breadth-first search
  - ♦ Uniform-cost search
  - ♦ Depth-first search
  - ♦ Depth-limited search
  - ♦ Iterative deepening search

3.4 UNINFORMED SEARCH STRATEGIES

3.4.1 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.
- Achieved simply by using FIFO queue for the frontier.
- *Slight tweak on the general graph-search algorithm (Figure 3.7)--Goal test is applied to each node when it is generated.*

3.4.1 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)
```

Properties of breadth-first search

- **Complete**?? Yes if b is finite
- **Time**?? $1+b+b^2+b^3+\ldots+b^d = O(b^d)$, i.e., exponential in d
- **Optimal**?? Yes, if step cost = identical, Not optimal in general
- **Space**?? There will be $O(b^{d-1})$ nodes in the explored set and $O(b^d)$ nodes in the frontier, so the **space** complexity is $O(b^d)$.
  - Problem: it can easily generate 1M nodes/second, so after 24hrs it has used 86,000GB (and then it has only reached depth 9 in the search tree)

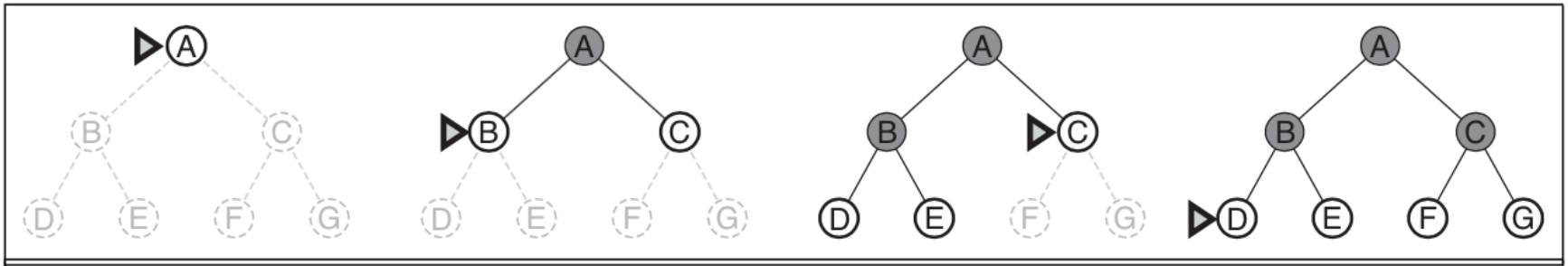


Figure 3.12 Breadth-first search on a simple binary tree. At each stage, the node to be expanded next is indicated by a marker.

3.4.2 Uniform-cost search (UCS)

```
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) /*with the cumulative costs as priority*/
      else if child.STATE is in frontier with higher PATH-COST then
        replace that frontier node with child
```

Output:

Iteration 1: ES=S, [S→RV, 80], [S→F, 99]

Iteration 2: ES=S+RV, [S→RV→P, 177], [S→F, 99]

Iteration 3: ES=S+RV+F, [S→RV→P, 177], [S→F→B, 310]

Iteration 4: ES=S+RV+F+P, [S→RV→P→B, 278], ~~[S→F→B, 310]~~

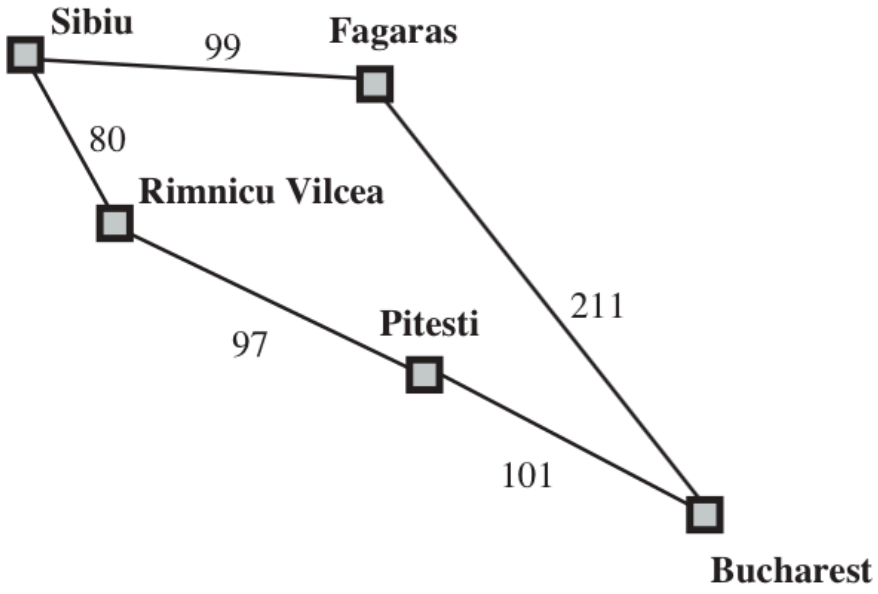


Figure 3.15 Part of the Romania state space, selected to illustrate uniform-cost search.

Properties of UCS

- **Complete**?? Yes, if step cost ≥ **ε** > 0
- **Time**?? # of nodes with path cost g(n) ≤ C^*,
  - i.e., $O(b^{1+\lfloor C^*/\varepsilon \rfloor})$ where C^* is the cost of the optimal solution and ε is the least cost of every action(step cost).
- **Space**?? Same as time
- **Optimal**?? Yes—nodes are expanded in increasing order of g(n)

- Uniform-cost search is guided by path costs rather than depths, so its complexity is not easily characterized in terms of b and d.
- Algorithm's worst-case time and space complexity is $O(b^{1+\lfloor C^*/\varepsilon \rfloor})$, which can be much greater than b^d.

3.4.3 Depth-first search

- **Depth-first search** always expands the *deepest* node in the current frontier of the search tree.
- **Complete**?? No: it fails in infinite-depth spaces. It also fails in finite spaces with loops but if we modify the search to avoid repeated states
  - complete in finite spaces
- **Time**?? $O(b^m)$: terrible if m is much larger than d but if solutions are dense, it may be much faster than breadth-first.
- **Space**?? O(bm): i.e., linear space!
- **Optimal**?? No

3.4.3 Depth-first search

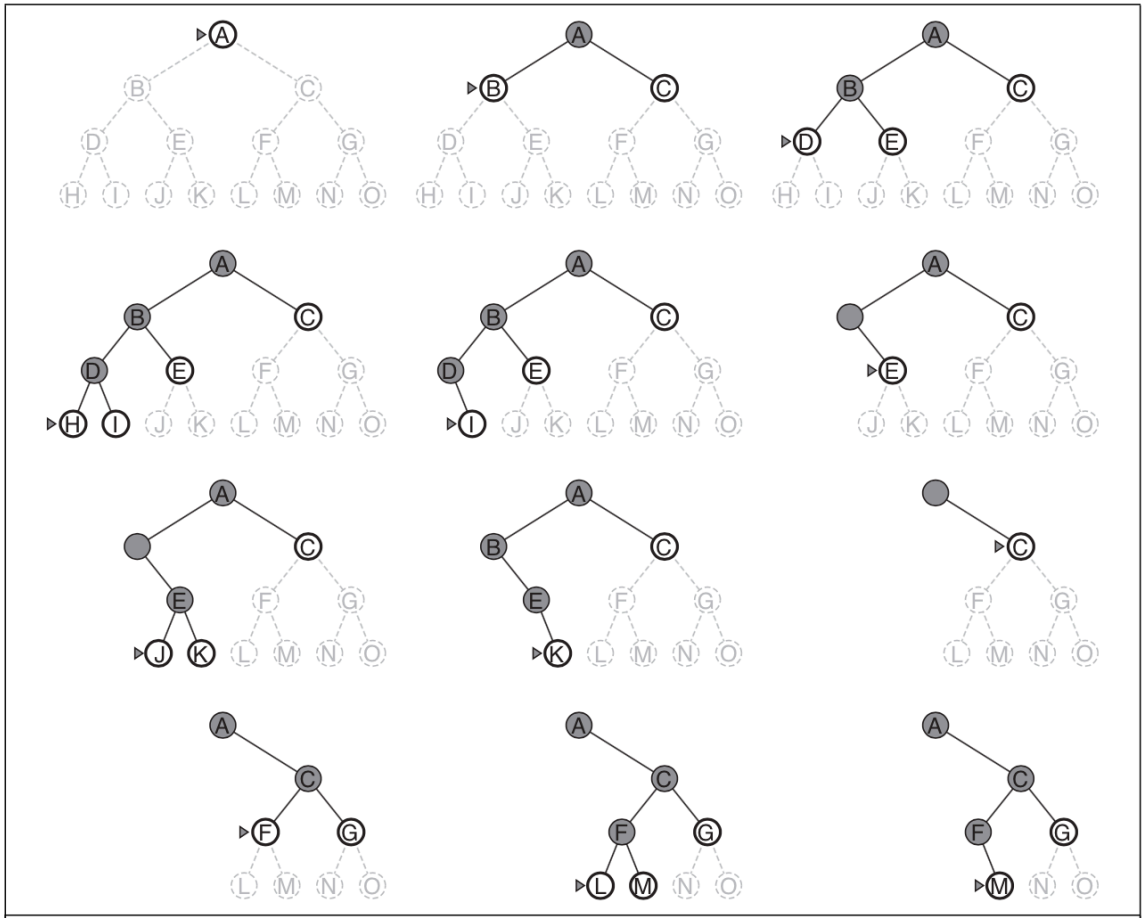


Figure 3.16 Depth-first search on a binary tree. The unexplored region is shown in light gray. Explored nodes with no descendants in the frontier are removed from memory. Nodes at depth 3 have no successors and M is the only goal node.

3.4.4 Depth-limited search

- Failure of depth-first search in infinite state spaces can be alleviated by supplying depth limit l. This approach is called **depth-limited search**.

Read it yourself

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

function RECURSIVE-DLS(node, problem, limit) returns soln/failure/cutoff
   if GOAL-TEST(problem, STATE[node]) then return node
   else if limit = 0 then return cutoff
   else
        cutoff-occurred? ← false
        for each action in ACTIONS(STATE[node], problem) 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
```

3.4.4 Depth-limited search

```
depth limit = max depth to search to;
agenda = initial state;
if initial state is goal state then
    return solution
else
    while agenda not empty do
        take node from front of agenda;
        if depth(node) < depth limit then
        {
            new nodes = apply operations to node;
            add new nodes to front of agenda;
            if goal state in new nodes then
            return solution;
        }
```

Read it yourself

Iterative Version of DLS

3.4.5 Iterative deepening depth-first search

- **Iterative deepening search** or iterative deepening depth-first search gradually increases the limit—first 0, then 1, then 2, and so on—until a goal is found.
- So is the combination of depth first search and depth limited search.

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

3.4.5 Iterative deepening depth-first search

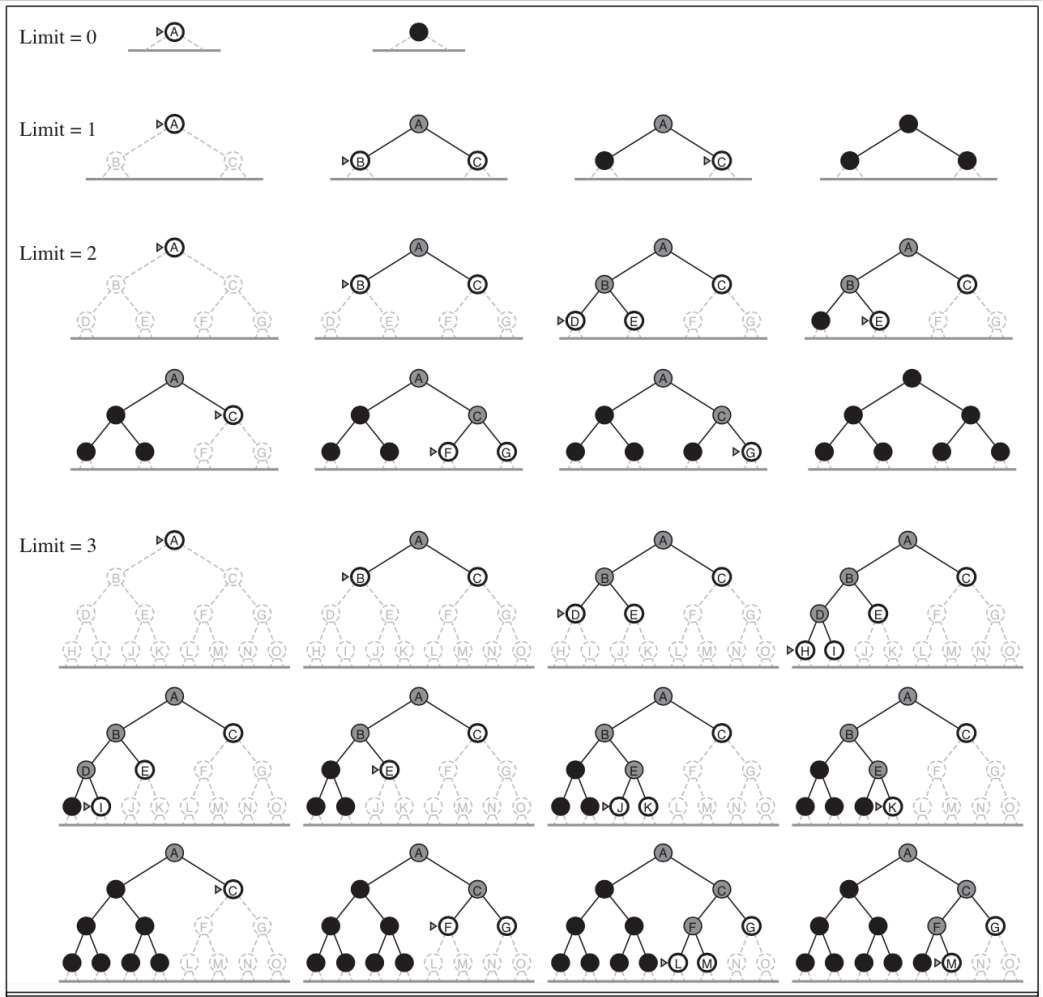
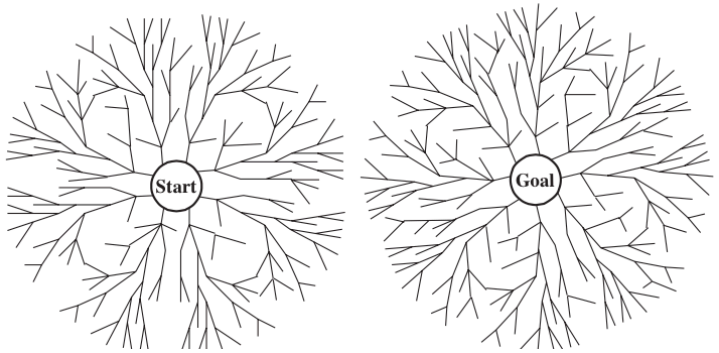


Figure 3.19 Four iterations of iterative deepening search on a binary tree.

3.4.6 Bidirectional search

- Run two simultaneous searches—one forward from the initial state and the other backward from the goal—hoping that the two searches meet in the middle is known as **Bidirectional Search**.
- The motivation is that $b^{d/2} + b^{d/2}$ is much less than b^{d}.
- (Read it yourself)



Comparing Uninformed Search Strategies

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	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(bm)$	$O(b\ell)$	$O(bd)$	$O(b^{d/2})$
Optimal?	Yes[c]	Yes	No	No	Yes[c]	Yes[c,d]

Figure 3.21 Evaluation of tree-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.

Lab Assignment No.2

- Implement the Romanian Example using the Depth First Search.

- For reference look at the following URL: https://github.com/TheCodingGent/UninformedSearches