CSE3013 Artificial Intelligence

Module-2: Problem solving by search

Dr Sunil Kumar P V

SCOPE

Overview

Introduction

Introduction

- Upto now, we discussed Reflex agents $\longrightarrow$ base their actions on a direct mapping from states to actions.
- Cannot operate well in environments for which this mapping would be too large to store
- Also, would take too long to learn
- Goal-based agents, on the other hand, consider future actions and the desirability of their outcomes
- Problem-solving agent $\longrightarrow$ a kind of goal based agent
- Intelligent agents are supposed to maximize their performance measure
- For this, the agent can adopt a goal and aim at satisfying it.

Goals

- Goals help organize behavior by limiting the objectives that the agent is trying to achieve and hence the actions it needs to consider.
- Goal formulation is the first step in problem solving.
- It is based on the current situation and the agent's performance measure
- Goal $\longrightarrow$ a set of states – exactly those states in which the goal is satisfied
- The agent has to find out how to act so that it reaches a goal state.

Problem Formulation

Problem formulation

- Detailing should be avoided
- If the problem is traveling from *Arad* to *Bucharest* then it should exclude minute details like "move the left foot forward an inch" or "turn the steering wheel one degree left"
- Because at that level of detail there is too much uncertainty and there would be too many steps in a solution
- Each state therefore corresponds to being in a particular town/junction in between
- Problem formulation is the process of deciding what actions and states to consider, given a goal

Assumptions on environments

- If the environment is **unknown** in —then it is has no choice but to try one of the actions at random.
  - If map is not known, when the agent is in a junction
- If map is known, it can foresee the best route and then achieve its goal by carrying out the driving actions of the journey
- If the environment is observable, the agent always knows the current state.
- If the environment is discrete, at any given state there are only finitely many actions to choose from
- If the environment is deterministic, each action has exactly one outcome
- We assume that the environment is **known, observable, discrete and deterministic**

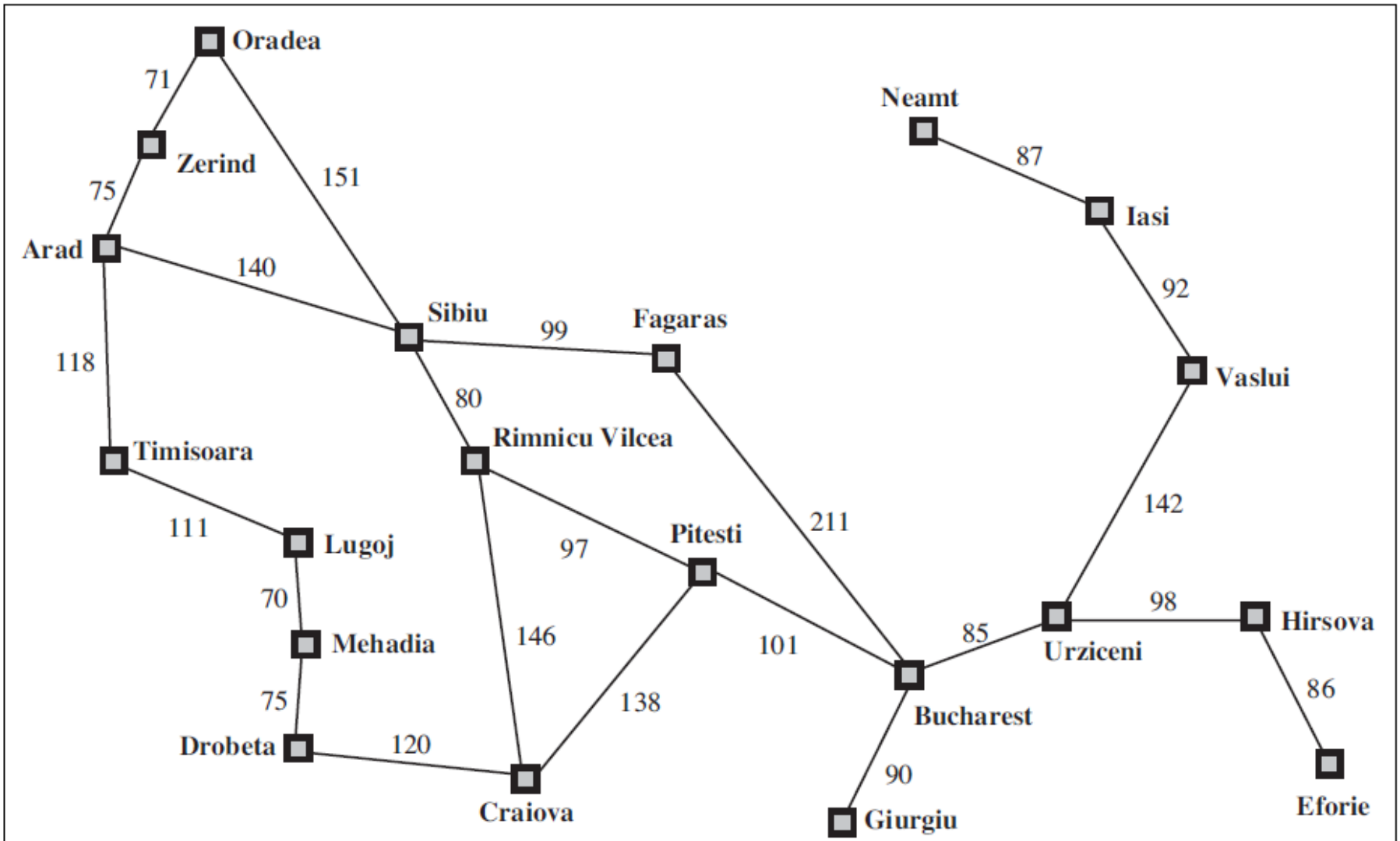


Figure 3.2 A simplified road map of part of Romania.

Working of agent

- *Under these assumptions, the solution to any problem is a fixed sequence of actions*
- Search $\longrightarrow$ The process of looking for a sequence of actions that reaches the goal
- Solution $\longrightarrow$ Output in the form of an **action sequence,** returned by a search algorithm which takes a **problem** as input
- Execution $\longrightarrow$ The process of carrying out the actions recommended by a solution
- The "formulate, search, execute" design for an agent

Problem space and state space

Problem Space(1/2)

- A problem can be defined formally by five components:
  - Initial state $\longrightarrow$ The state that the agent starts in, eg:- $In(Arad)$
  - 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.
    - Each of these actions is applicable in s.
    - Eg: from the state $In(Arad)$, the applicable actions are $\{Go(Sibiu), Go(Timisoara), Go(Zerind)\}$.
  - Transition model $\longrightarrow$ A description of what each action does
    - specified by a function $RESULT(s, a)$ that returns the state that results from doing action a in state s.
    - Successor $\longrightarrow$ Refers to any state reachable from a given state by a single action
    - Eg, we have $RESULT(In(Arad), Go(Zerind)) = In(Zerind)$

Problem Space(2/2)

- Goal test $\longrightarrow$ Determines whether a given state is a goal state
  - If there is an explicit set of possible goal states, and the test simply checks whether the given state is one of them
  - Eg: In our problem: $\{In(Bucharest)\}$
  - Sometimes the goal is specified by an abstract property rather than an explicitly enumerated set of states
  - Eg. in chess, the goal is to reach a state called "checkmate," where the opponent's king is under attack and can't escape
- Path Cost $\longrightarrow$ The function that assigns a numeric cost to each path
  - Length of path
  - The sum of the costs of the individual actions along the path
  - The step cost of taking action a in state s to reach state s' is denoted by $c(s, a, s'), c \geq 0$

State Space

- Is the set of all states reachable from the initial state by any sequence of actions.
- Together, the **initial state, actions, and transition model** implicitly define the state space of the problem
- Forms a directed network or graph in which the **nodes are states** and the **links** between nodes are **actions**
- A path in the state space is a sequence of states connected by a sequence of actions

Abstractions and actual problems

- In the example we used abstractions by excluding the details
- Abstract state descriptions and actions
- The state can include details such as
  - 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
- The abstract solution of path from Arad to Sibiu to Rimnicu Vilcea to Pitesti to Bucharest corresponds to a large number of more detailed paths
- Eg, we could drive with the radio on between Sibiu and Rimnicu Vilcea, and then switch it off for the rest of the trip

Actual problem example (Toy problem)

- Toy problems are is intended to illustrate or exercise various problem-solving methods
- 8 queens problem $\longrightarrow$ place eight queens on a chessboard such that no queen attacks any other
  - 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.

Actual problem example (Real world problem)

- 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?

Searching for solutions

Search Trees

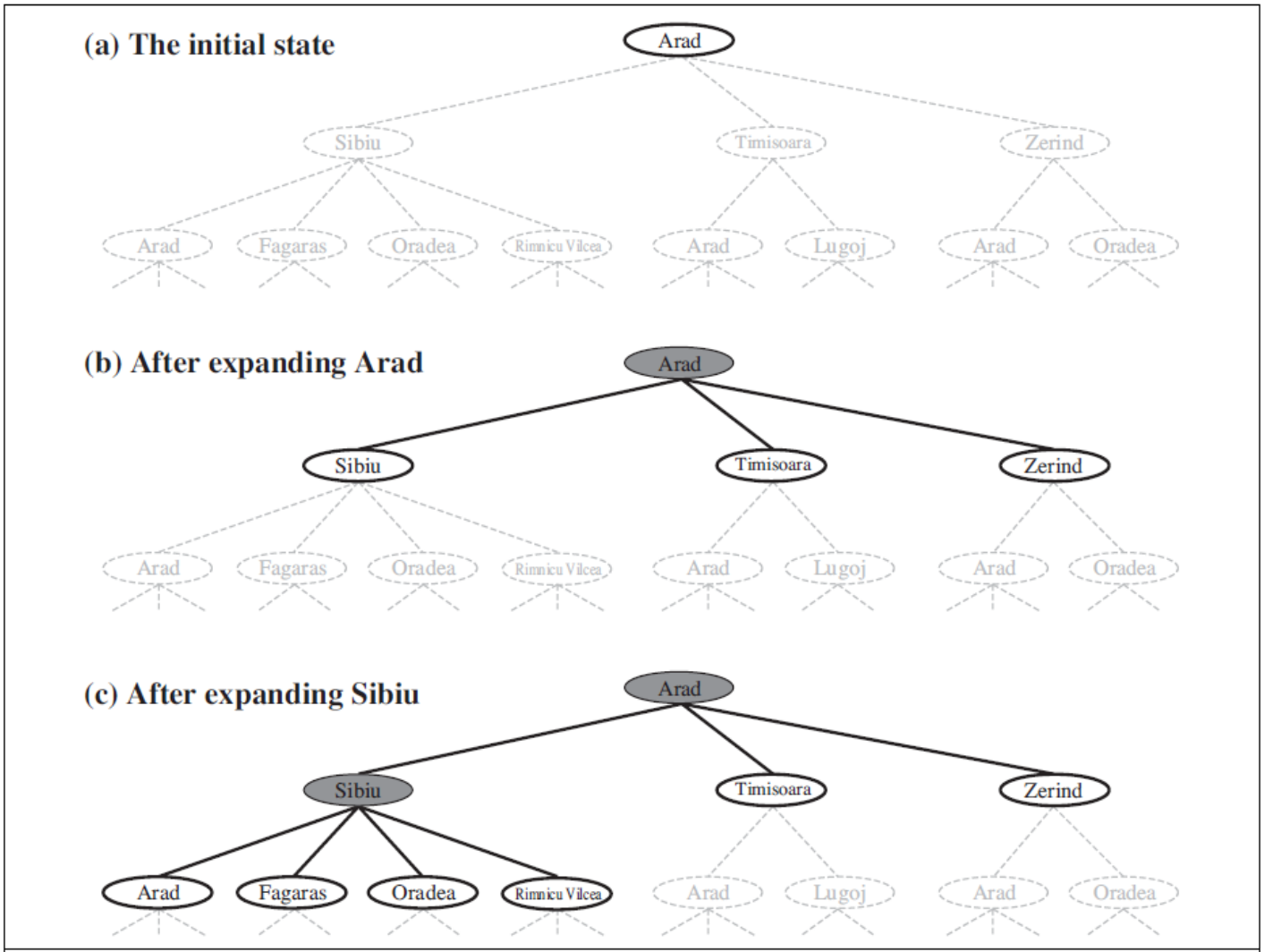


Figure 3.6 Partial search trees for finding a route from Arad to Bucharest. Nodes that have been expanded are shaded; nodes that have been generated but not yet expanded are outlined in bold; nodes that have not yet been generated are shown in faint dashed lines.

Graph Search

- Redundant paths $\longrightarrow$ Exists when more than one way to get from one state to another
- Eg: Arad–Sibiu (140 km long) and Arad–Zerind–Oradea–Sibiu (297 km long).
- To reach goal, keeping more than one path is not required
- Search tree algorithm has the disadvantage of redundant paths
- Graph search incorporates this improvement

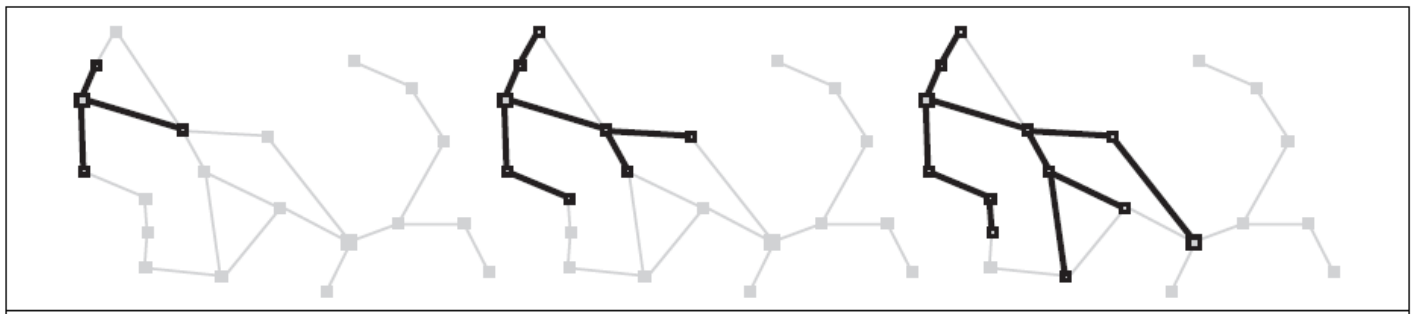


Figure 3.8 A sequence of search trees generated by a graph search on the Romania problem of Figure 3.2. At each stage, we have extended each path by one step. Notice that at the third stage, the northernmost city (Oradea) has become a dead end: both of its successors are already explored via other paths.

Blind search

Blind Search/Uninformed search

- Remember, Search $\longrightarrow$ The process of looking for a sequence of actions that reaches the goal
- Blind: 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
- Search strategies discussed here are *distinguished by the order* in which nodes are expanded
- Strategies that know whether one non-goal state is "more promising" than another are called **informed search** or **heuristic search** strategies (will be discussed later)

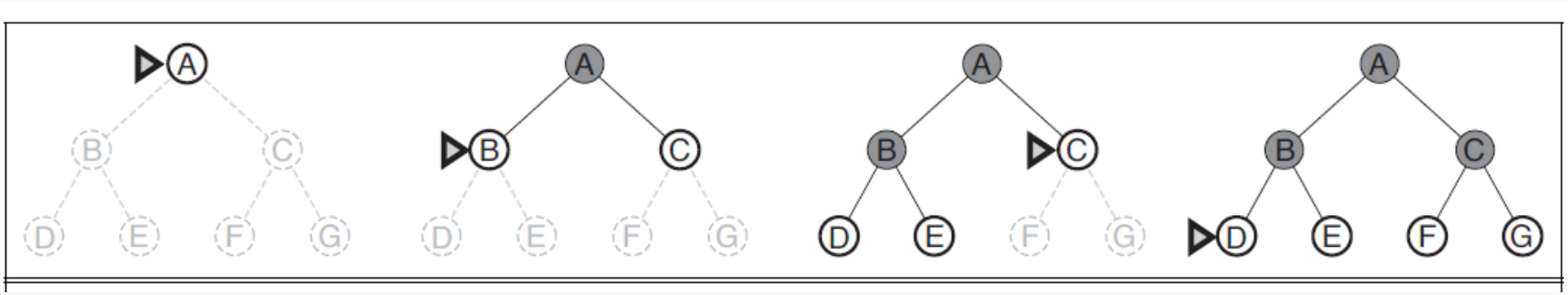
BFS

- Simple strategy in which the root node is expanded first, then all the SEARCH 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
- Implemented using a FIFO queue
- Frontier: - The set of all nodes available for expansion at any given point is called the frontier

BFS - Pseudocode

```
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)
```

BFS - Demo



Performance of BFS

- Imagine searching a uniform tree where every state has b successors
- 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 $b + b^2 + b^3 + \cdots + b^d = O(b^d)$
- Space complexity $\longrightarrow$ for any kind of graph search, which stores every expanded node in the explored set
- So the space complexity is $O(b^d)$
- Both are very high, exponential

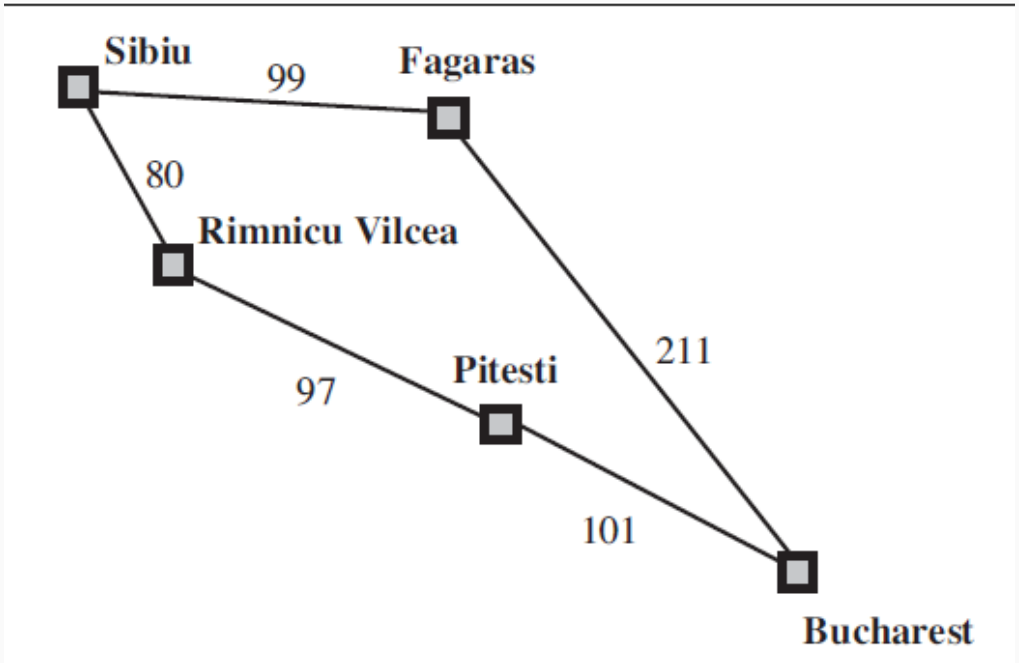
Uniform cost search

- When all step costs are equal, breadth-first search is optimal
- **Uniform-cost search** expands the node n with the lowest path cost $g(n)$
- Store the frontier as a priority queue ordered by g

Uniform cost search - Algorithm

```
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: Demo



Uniform cost search performance

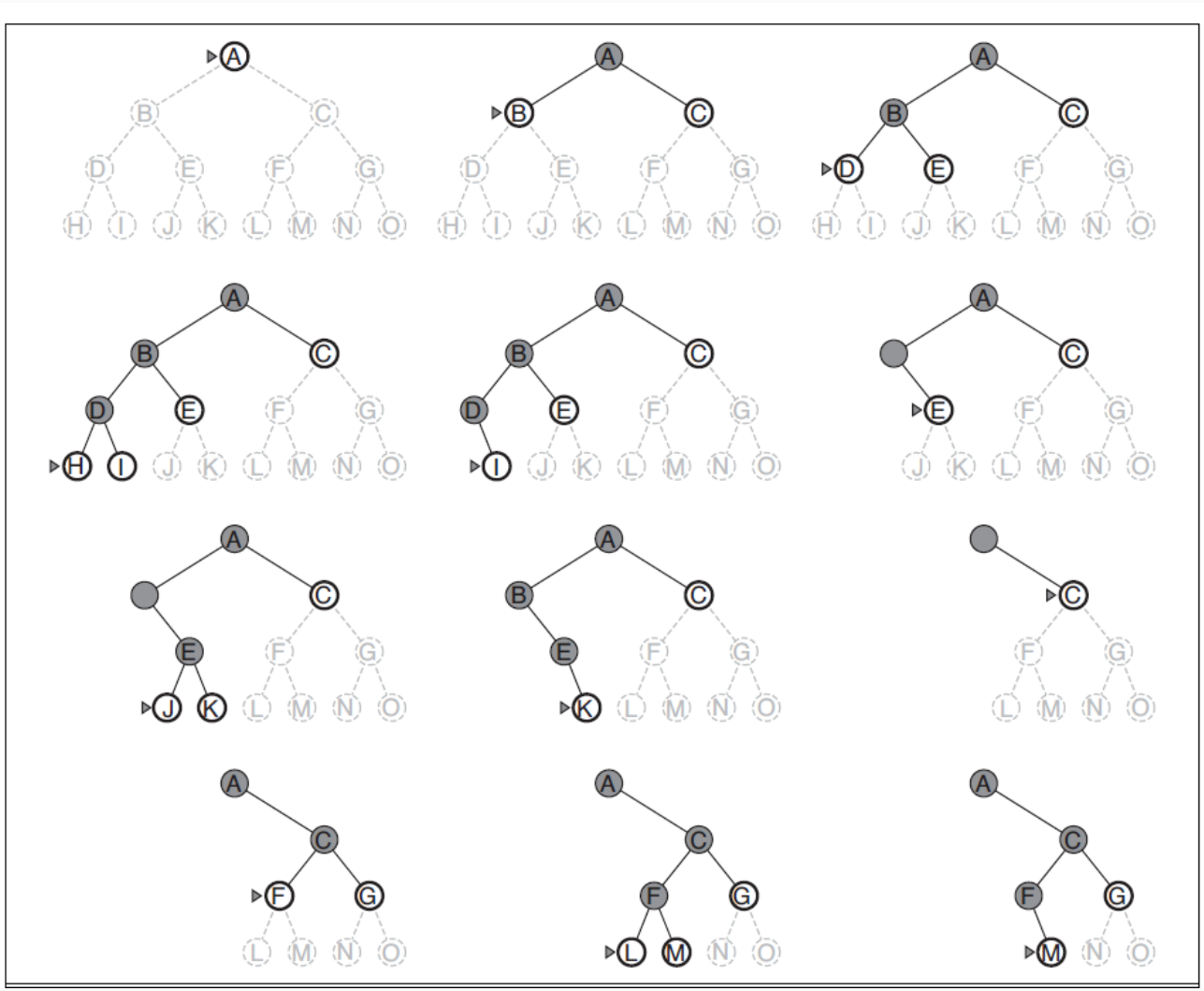
- Let
  - C^* be the cost of the optimal solution
  - Every action costs at least ϵ, otherwise the algorithm will go to infinite loop
- Then the algorithm's worst-case time and space complexity is $O(b^{1+\lfloor C^*/\epsilon \rfloor})$, which can be much greater than b^d

Depth first search (DFS)

- Depth-first search always expands the deepest node in the current frontier of the search tree
- It will use a LIFO queue (stack)

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

DFS - Demo



Performance of DFS (1/2)

- The properties of depth-first search depend strongly on whether the graph-search or tree-search version is used.
- The graph-search version avoids repeated states and redundant paths
- So it is complete in finite state spaces because it will eventually expand every node
- The tree-search version is not complete, in case of redundant paths (Example on **Slide-14**)
- Will loop *Arad–Sibiu–Arad–Sibiu* forever
- Non-optimal
- DFS will explore the entire left subtree even if node C is a goal node
- If node J were also a goal node, it will be returned as a solution instead of C, which would be a better solution

Performance of DFS (2/2)

- The time complexity of DFS graph search is bounded by the size of the state space (which may be infinite)
- A DFS tree search may generate all of the $O(b^m)$ nodes in the search tree ($m \longrightarrow$ maximum depth of any node)
- DFS has no clear advantage over BFS
- Yet why do we include it? The reason is the space complexity
- For a state space with branching factor b and maximum depth m, DFS needs to store only $O(bm)$ nodes
- A variant of DFS called backtracking search uses still less memory
  - In backtracking, only one successor is generated at a time rather than all successors
  - Each partially expanded node remembers which successor to generate next
  - In this way, only $O(m)$ memory is needed rather than $O(bm)$

Depth Limited Search

- DFS disadvantage $\longrightarrow$ It is bounded by the size of the state space (can be infinity)
- Overcome by using a DFS with a predetermined depth limit, l
- Nodes at depth l are treated as if they have no successors
- The depth limit solves the infinite-path problem
- Its time complexity is $O(b^l)$ and its space complexity is $O(bl)$
- DFS is depth limited search with $l = \infty$

Depth limited search algorithm

```
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
```

Iterative Deepening Depth First Search (IDS)

- Combines the benefits of DFS and BFS
- Finds the best depth limit iteratively
- Gradually increases the limit—first 0, then 1, then 2, and so on—until a goal is found
- Memory requirements are modest: $O(bd)$

```
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
```

IDS - Demo

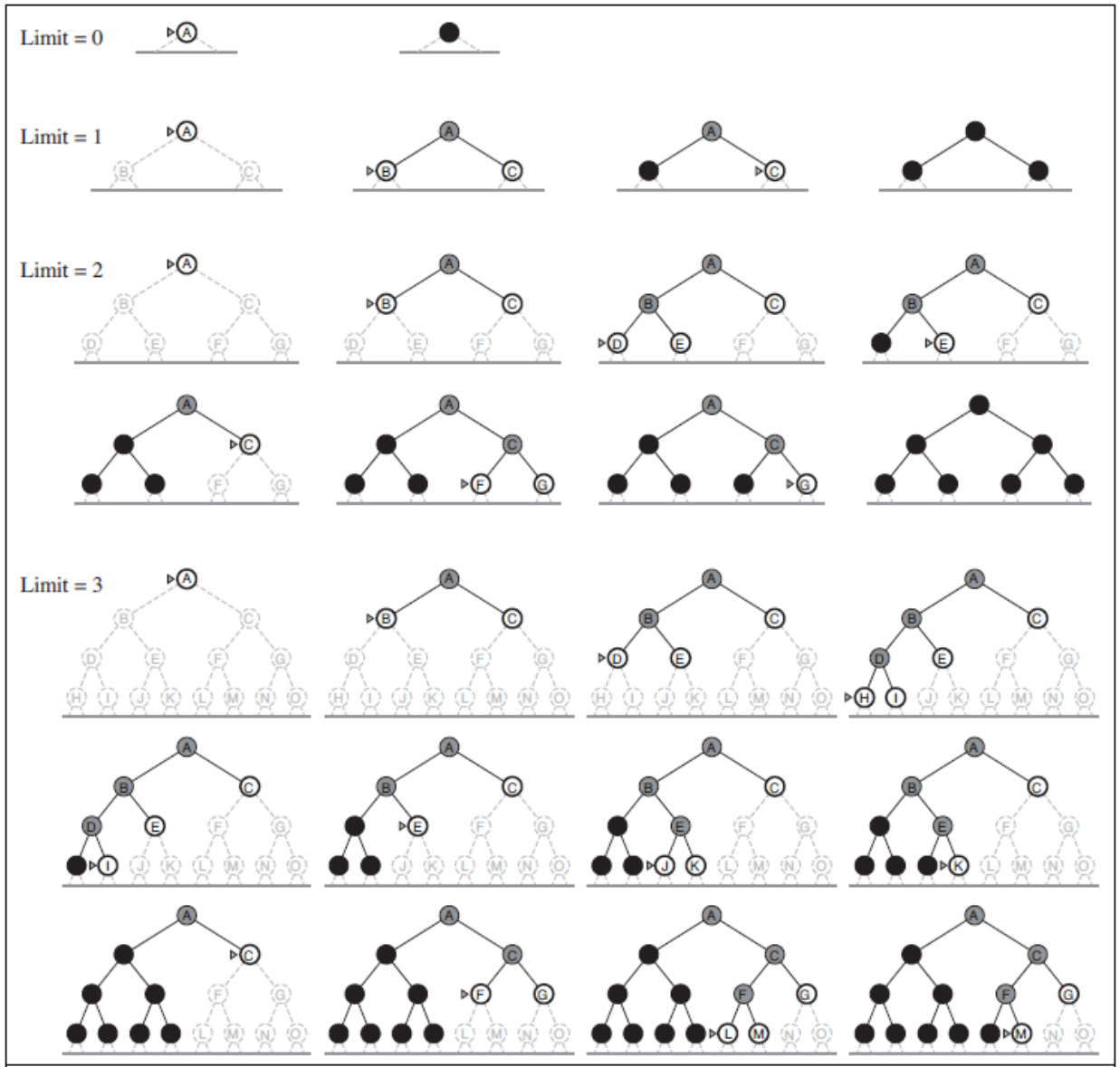


Figure 3.19 Four iterations of iterative deepening search on a binary tree.

Bidirectional search

- Idea $\longrightarrow$ to 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
- Motivation is that $b^{d/2} + b^{d/2}$ is much less than b^d
- Implemented by replacing the goal test with a check to see whether the frontiers of the two searches intersect
- If they do, a solution has been found

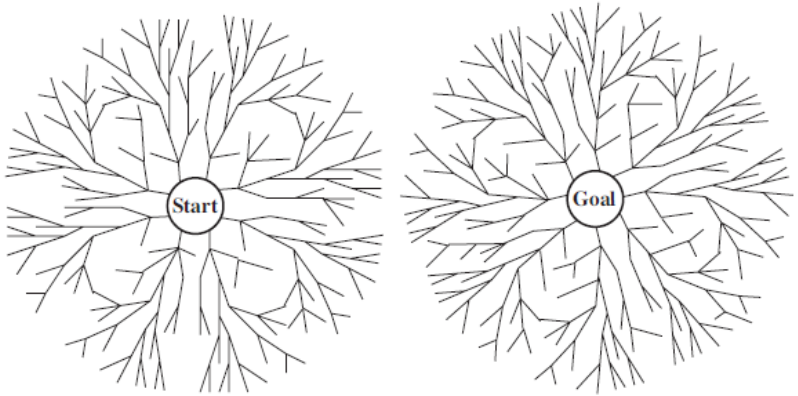


Figure 3.20 A schematic view of a bidirectional search that is about to succeed when a branch from the start node meets a branch from the goal node.

Performance Measurement

Performance parameters - Search techniques

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?

Optimal Solution:

- Solution quality is measured by the path cost function
- Optimal solution has the lowest path cost among all solutions

Performance of blind search techniques

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.

Thank you...