UNIVERSITÄT DES SAARLANDES

Artificial Intelligence

Systematic (Uninformed) Search

Prof. Dr. habil. Jana Koehler

Dr. Sophia Saller, M. Sc. Annika Engel

Summer 2020

Deep thanks goes to Prof. Jörg Hoffmann for sharing his course material

Agenda

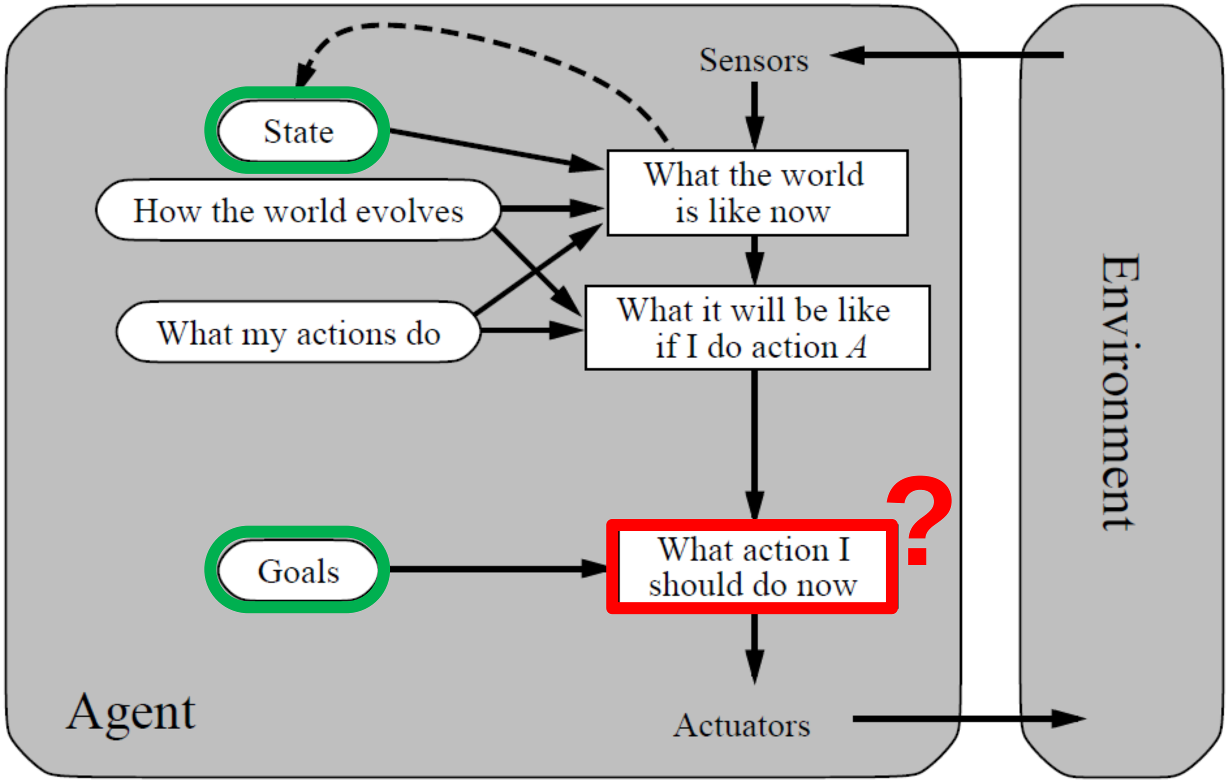
- Basic Terminology and Concepts
- Modeling Search Problems
- Uninformed (Systematic) Search Strategies
  - Breadth-First search
  - Depth-First search
  - Depth-Limited search
  - Iterative Deepening
  - Uniform Cost search

Recommended Reading

- AIMA Chapter 3: Solving Problems by Searching
  - 3.1 Problem Solving Agents
  - 3.2 Example Problems
  - 3.3 Searching for Solutions
  - 3.4 Uninformed Search Strategies, the following subchapters:
    - 3.4.1 Breadth-first search
    - 3.4.2 Uniform-cost search
    - 3.4.3 Depth-first search
    - 3.4.4 Depth-limited search
    - 3.4.5 Iterative deepening depth-first search
    - 3.4.7 Comparing uninformed search strategies

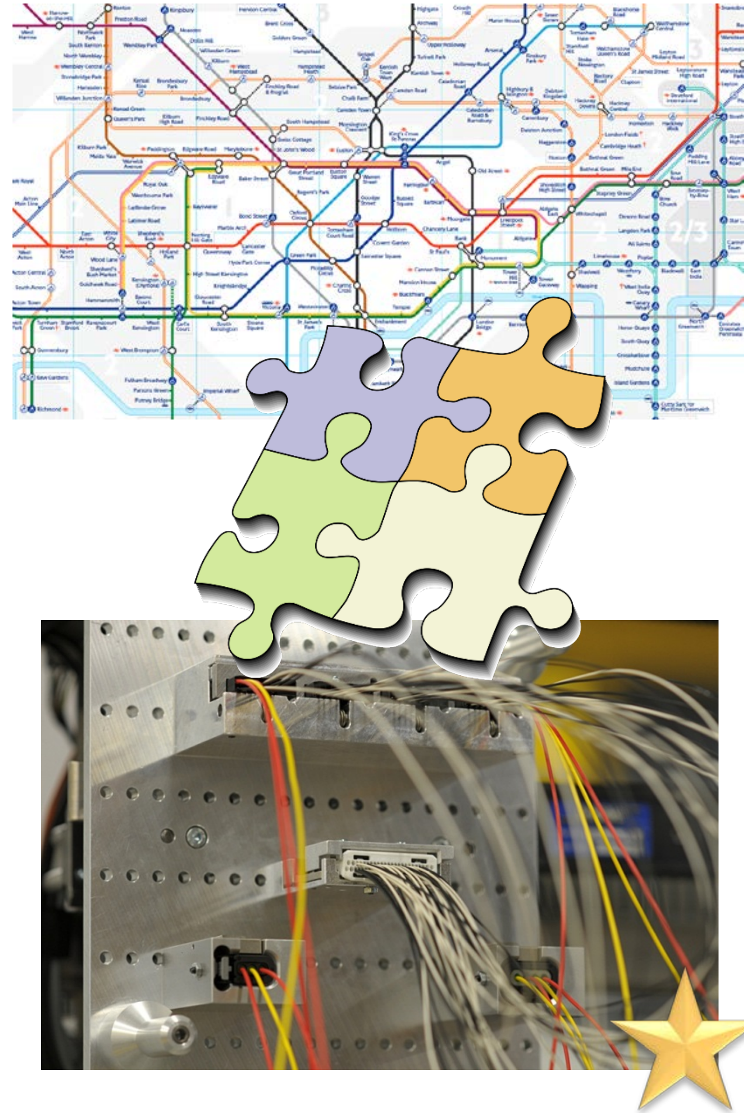
How can a Goal-based Agent reach a Goal?

- Agent perceives the world being in different states
  - Initial state: the current state of the world
  - Goal(s): a future state of the world (desirable for the agent)



Discrete State-Based Search Problems

- **Discrete**
  - Finite number of states & actions
- **Single agent**
  - Do not consider action-based changes by other agents
- **Static**
  - World does not change while agent is deliberating
- **Observable**
  - Agent has access to relevant knowledge
- **Deterministic**
  - Each action has exactly one successor state $s \times a \rightarrow s'$



State Spaces

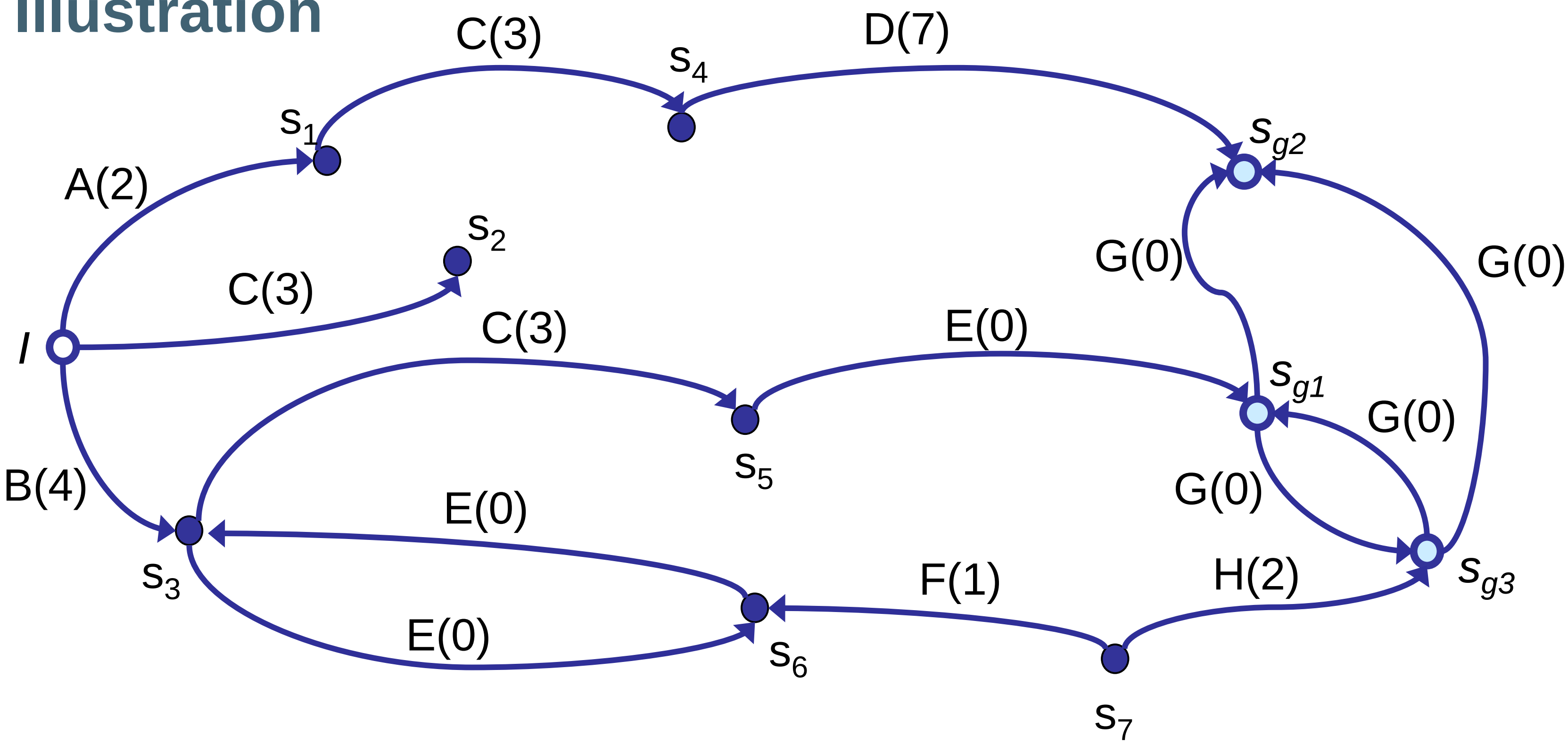
Definition (State Space)

A state space is a 6-tuple $\Theta = (S, A, c, T, I, S^G)$ where:

- S is a finite set of states.
- A is a finite set of actions.
- $c : S \times A \mapsto \mathbb{R}_0^+$ is the cost function.
- $T \subseteq S \times A \times S$ is the transition relation. We require that T is deterministic, i.e., for all $s \in S$ and $a \in A$, there is at most one state s' such that $(s, a, s') \in T$. If such (s, a, s') exists, then a is applicable to s.
- $I \in S$ is the initial state.
- $S^G \subseteq S$ is the set of goal states.

We say that Θ has the transition (s, a, s') if $(s, a, s') \in T$. We also write $s \xrightarrow{a} s'$, or $s \rightarrow s'$ when not interested in a.
We say that Θ has unit costs if, for all $a \in A$ and all $s \in S$, $c(s, a) = 1$.

Illustration



- Unit costs? No (see numbers in brackets)
- Actions applicable in initial state I: *A, B, C*
- Deterministic T? *No (see action G in state s_{g1})*

UNIVERSITÄT DES SAARLANDES

Terminology

- s' successor of s if $s \rightarrow s'$; s predecessor of s' if $s \rightarrow s'$.
- s' reachable from s if there exists a sequence of transitions:

$$s = s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} \cdots \xrightarrow{a_{n-1}} s_{n-1} \xrightarrow{a_n} s_n = s'$$

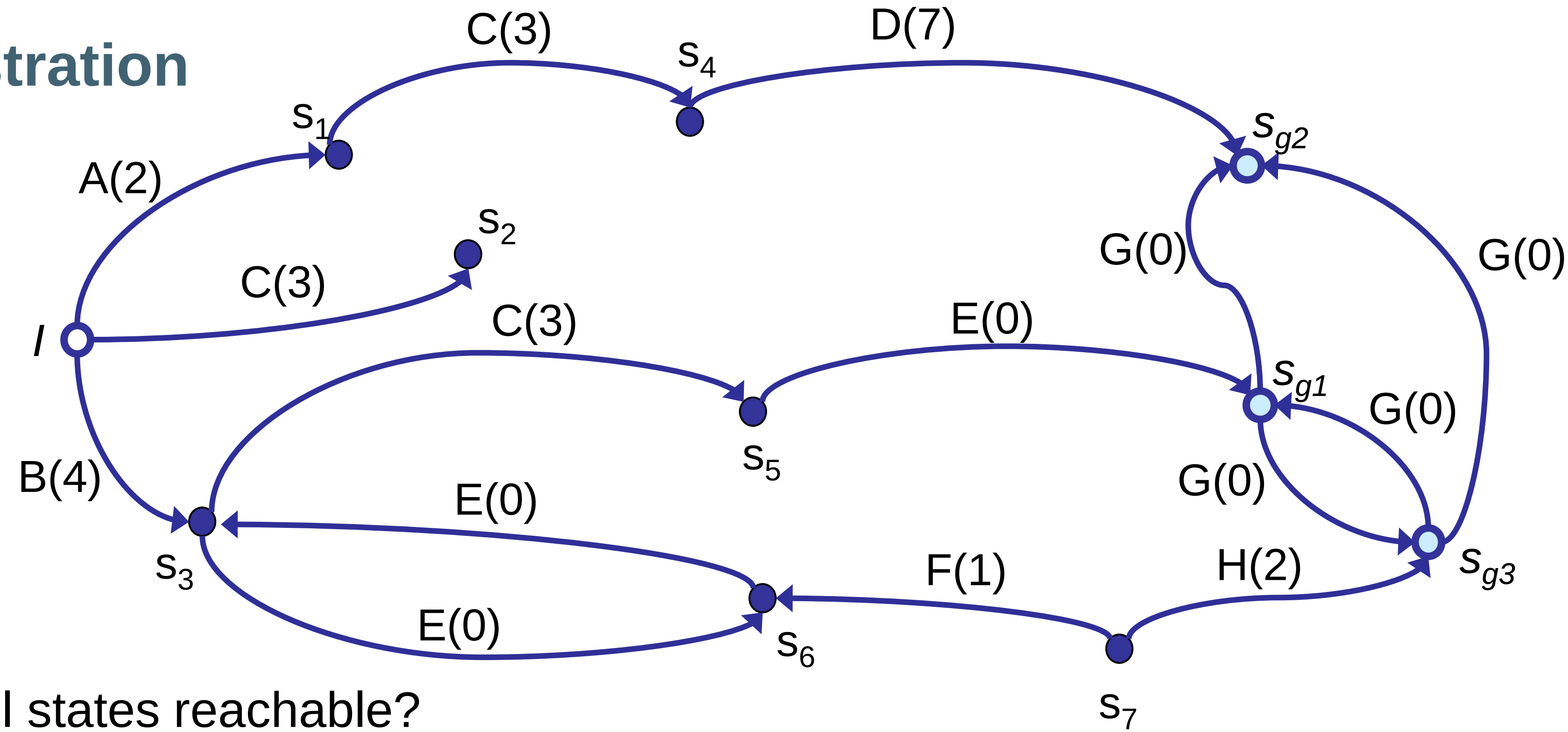
  - $n = 0$ possible; then $s = s'$.
  - $(a_1, \ldots, a_n)$ is called (action) path from s to s'.
  - $(s_0, \ldots, s_n)$ is called (state) path from s to s'.
  - The cost of that path is $\sum_{i=1}^{n} c(s_{i-1}, a_i)$.
- s' is reachable (without reference state) means reachable from I.
- s is solvable if some $s' \in S^G$ is reachable from s; else s is a dead end.

(Optimal) State Space Solution

Let $\Theta = (S, A, c, T, I, S^G)$ be a state space, and let $s \in S$.

- A solution for s is an action path $(a_1, \ldots, a_n)$ from s to some $s' \in S^G$.
- The solution is optimal if its cost is minimal among all solutions for s.
- A solution for I is called a solution for Θ and denoted by ρ.
- The set of all solutions for Θ is denoted by $\mathbb{S}^{\Theta}$.
- If a solution exists, then Θ is solvable, otherwise unsolvable.

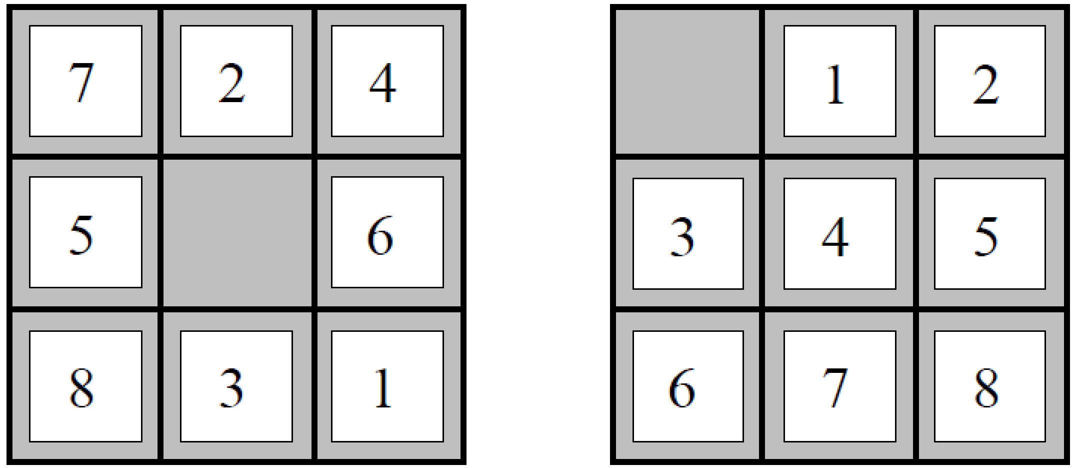
Illustration



- All states reachable?
  - No: s_7 has only outgoing edges
- All states solvable?
  - No: s_2 has no outgoing edges (dead end)
- Optimal solutions?

  B – E* – C – E – G* costs: 4+0+3+0+0 = 7

Example: The 8-Puzzle



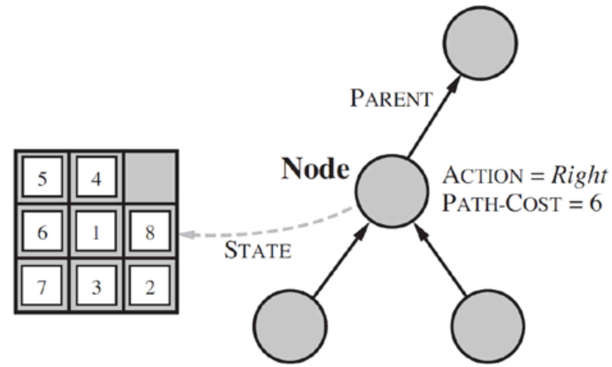
Start State   Goal State

- States: ______________________________
- Initial (Start) State: ______________________________
- Actions: ______________________________
- Goal states: ______________________________
- Path Costs: ______________________________

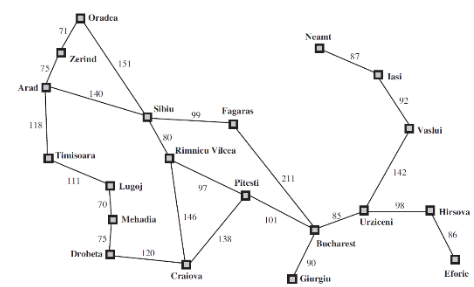
UNIVERSITÄT DES SAARLANDES

Formulating Search Problems

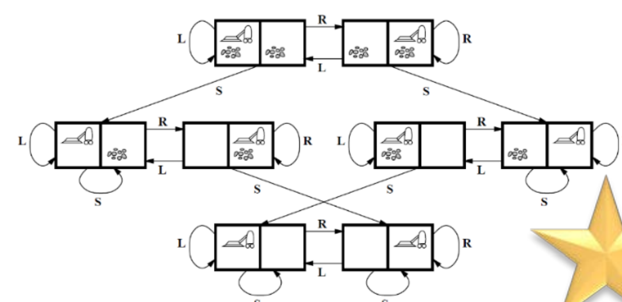
1) Blackbox description
   - Application programming interface (API) to construct the state space



2) Whitebox description
   - Accessible, but compact representation of states, actions, goal test, …

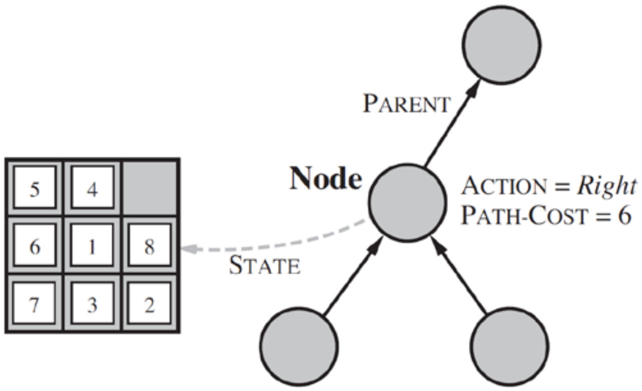


3) Explicit description
   - Explicit representation of all states in the state-space graph



Blackbox Description

- Application programming interface (API) to construct the state space



Blackbox Description of a Problem

- InitialState(): Returns the initial state of the problem.
- GoalTest(s): Returns a Boolean, "true" iff state s is a goal state.
- Cost(a): Returns the cost of action a.
- Actions(s): Returns the set of actions that are applicable to state s.
- ChildState(s, a): Requires that action a is applicable to state s, i.e., there is a transition $s \xrightarrow{a} s'$. Returns the outcome state s'.

Implementation – What is a Search Node?

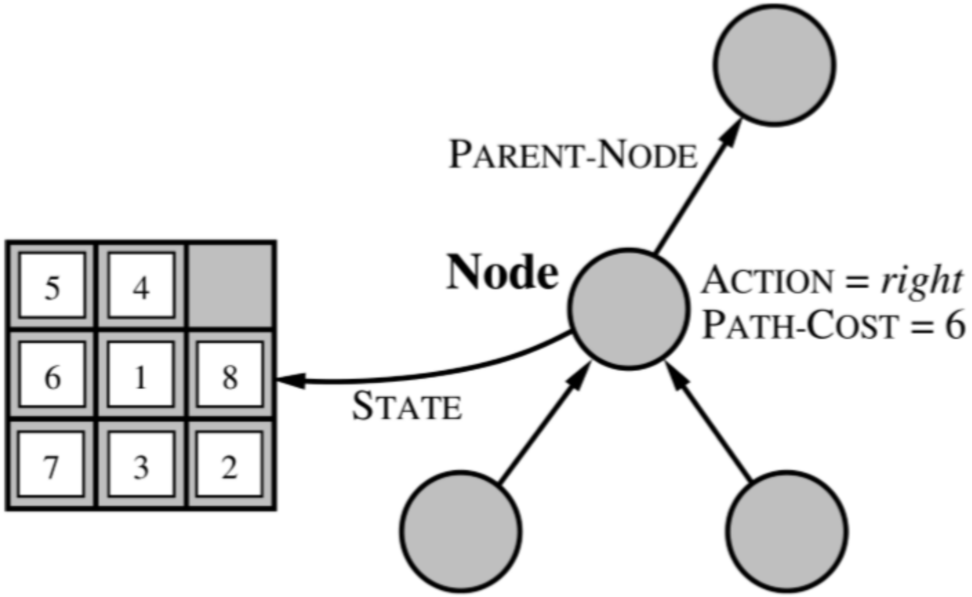
Data Structure for Every Search Node n

n.State: The state (from the state space) which the node contains.

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.PathCost: $g(n)$, the cost of the path from the initial state to the node (as indicated by the parent pointers).



Implementation – Operations on Search Nodes

Operations on Search Nodes

Solution(n): Returns the path to node n. (By backchaining over the n.Parent pointers and collecting n.Action in each step.)

ChildNode(problem,n,a): Generates the node n' corresponding to the application of action a in state n.State. That is:
n'.State:=problem.ChildState(n.State, a);
n'.Parent:= n; n'.Action:= a;
n'.PathCost:= n.PathCost+problem.Cost(a).

Implementation – Operations on the Open List

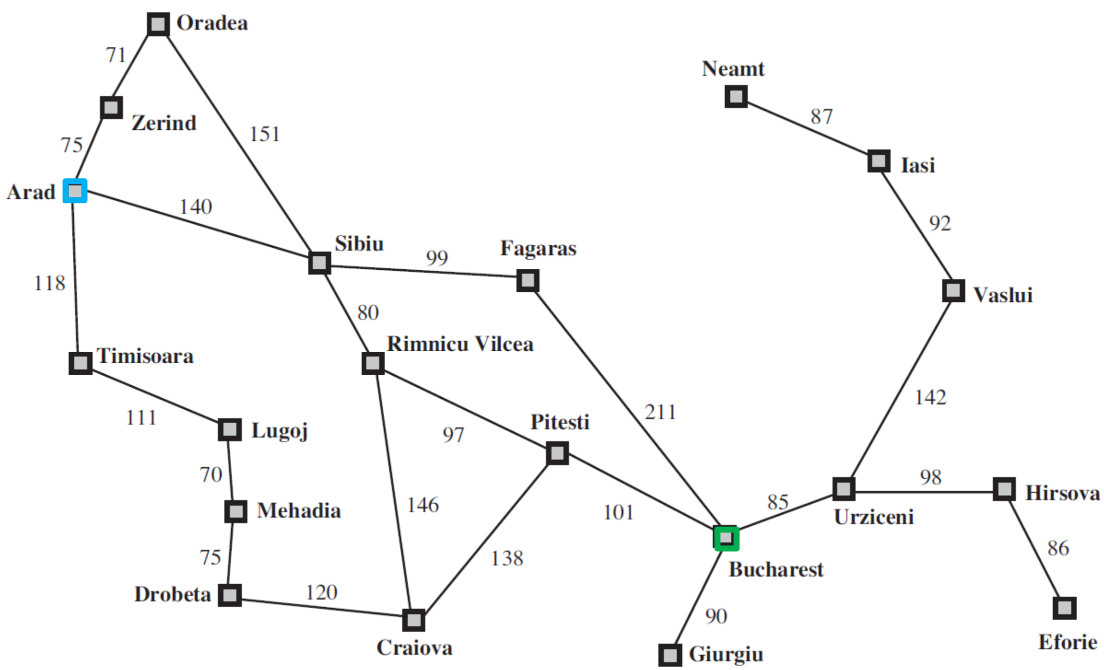
- When being in some node n (with state s), we usually have several options for applying actions that lead us to child nodes (with successor states s')
  - The list of these candidate children nodes is called Open List

Operations for the Open List

Empty?(frontier):	Returns true iff there are no more elements in the open list.
Pop(frontier):	Returns the first element of the open list, and removes that element from the list.
Insert(element, frontier):	Inserts an element into the open list.

Whitebox Description: Romania Travel Example

- Find a route from Arad to Bucharest
- States: map with cities, initial state city, current city, and goal state city
- Actions: edges (trips) between cities
- Action costs: distance information on the edges

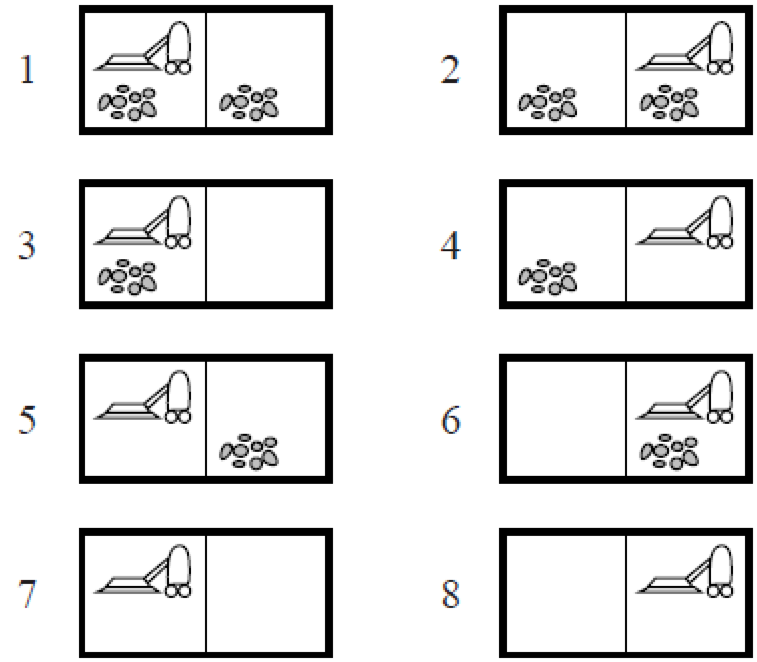


solution
a path from Arad to Bucharest

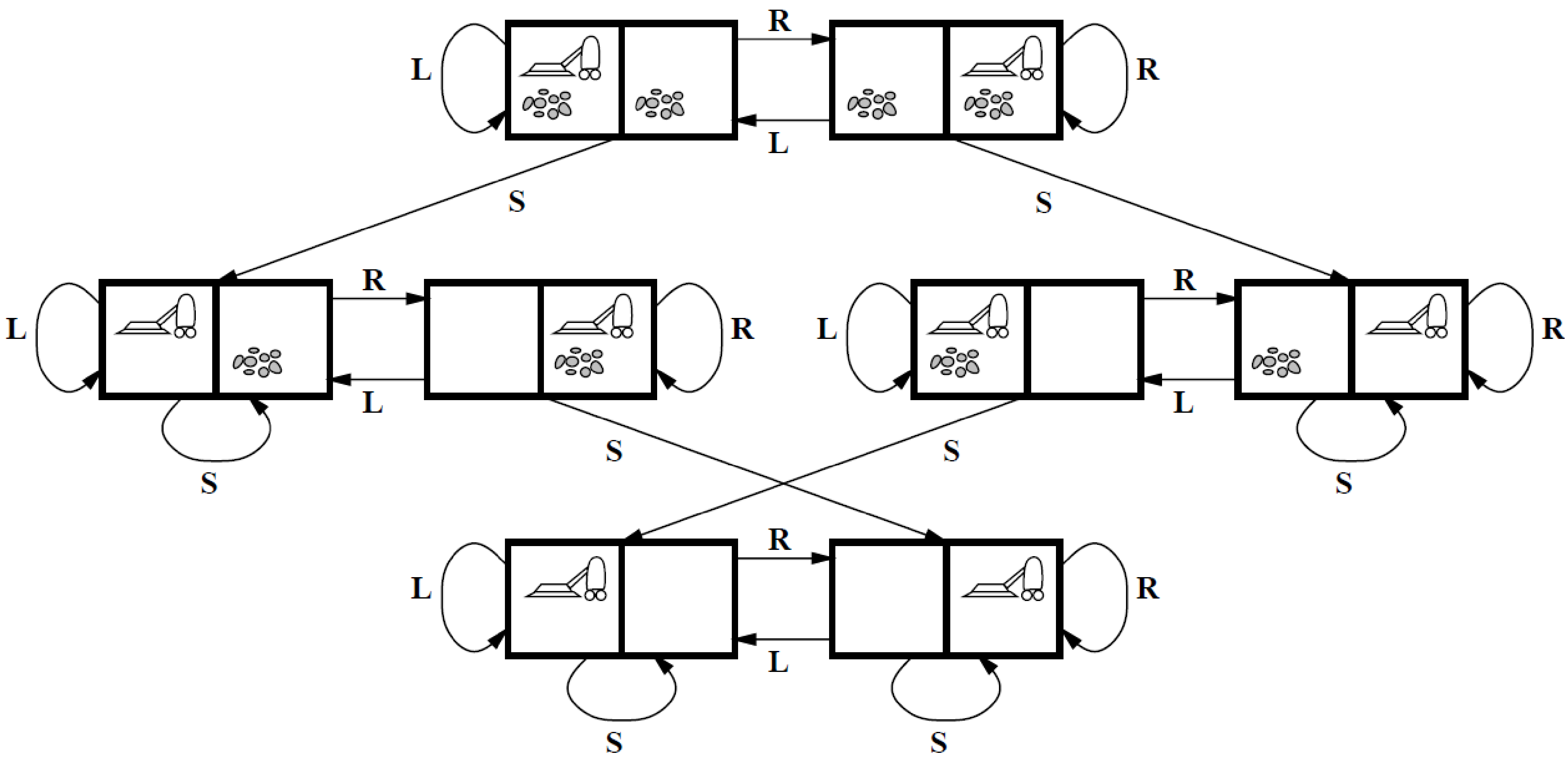
optimal solution
the path from Arad to Bucharest with shortest path costs

Whitebox Description: Vacuum Cleaner Agent

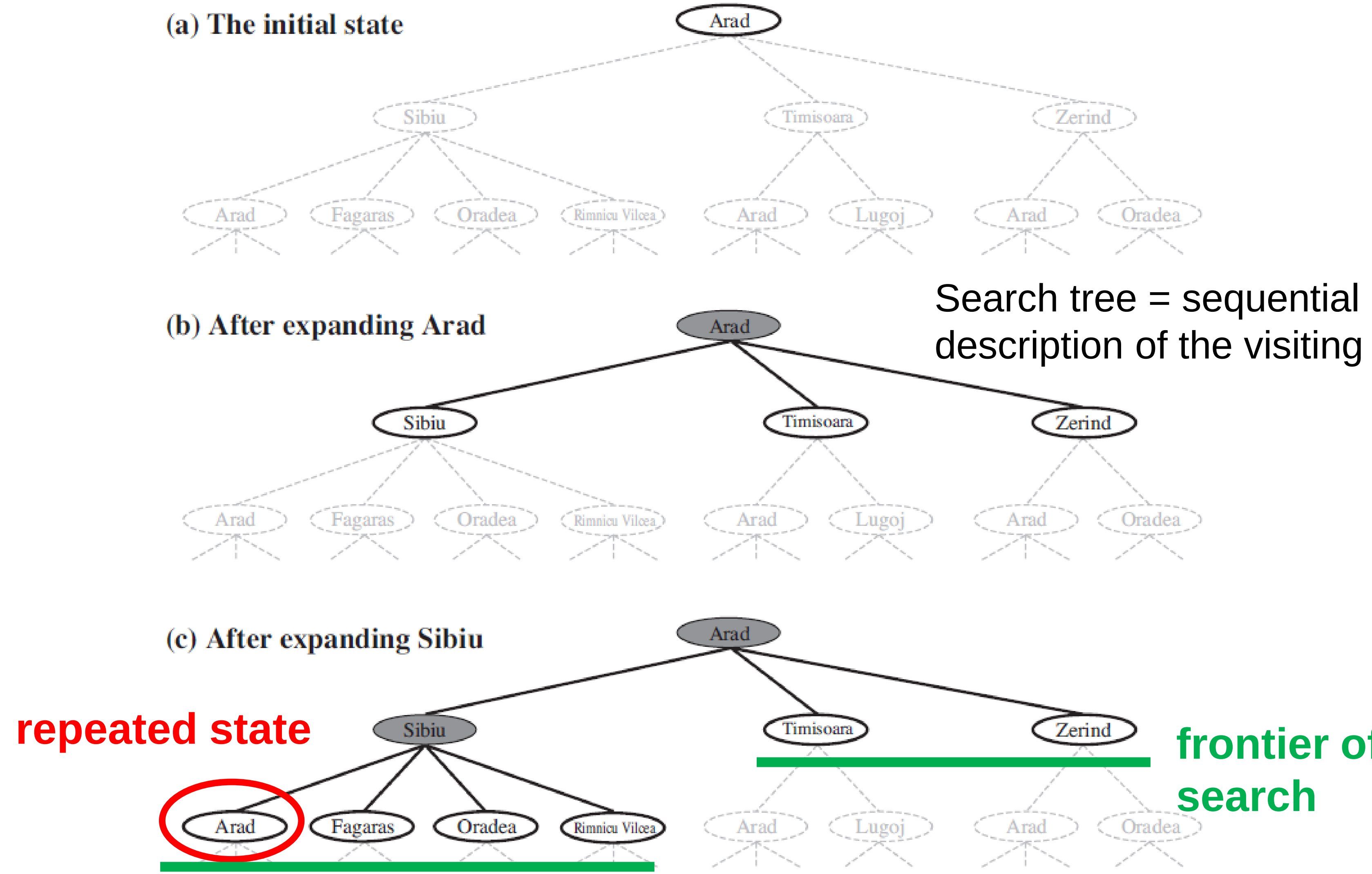
- World state space:
  2 positions, dirt or no dirt
  $\rightarrow$ 8 world states
- Actions:
  Left (L), *Right* (R), or *Suck* (S)
- Goal:
  no dirt in the rooms
- Path costs:
  one unit per action



Explicit Description: State Space of Vacuum Cleaning Agent



Search builds a Search Tree when exploring the State Space Graph



Search tree = sequential description of the visiting order

Terminology to discuss Search Algorithms

Search node n: Contains a *state* reached by the search, plus information about how it was reached.

Path cost $g(n)$: The cost of the path reaching n.

Optimal cost g^*: The cost of an optimal solution path. For a state s, $g^*(s)$ is the cost of a cheapest path reaching s.

Node expansion: Generating all successors of a node, by applying all actions applicable to the node's state s. Afterwards, the *state* s itself is also said to be expanded.

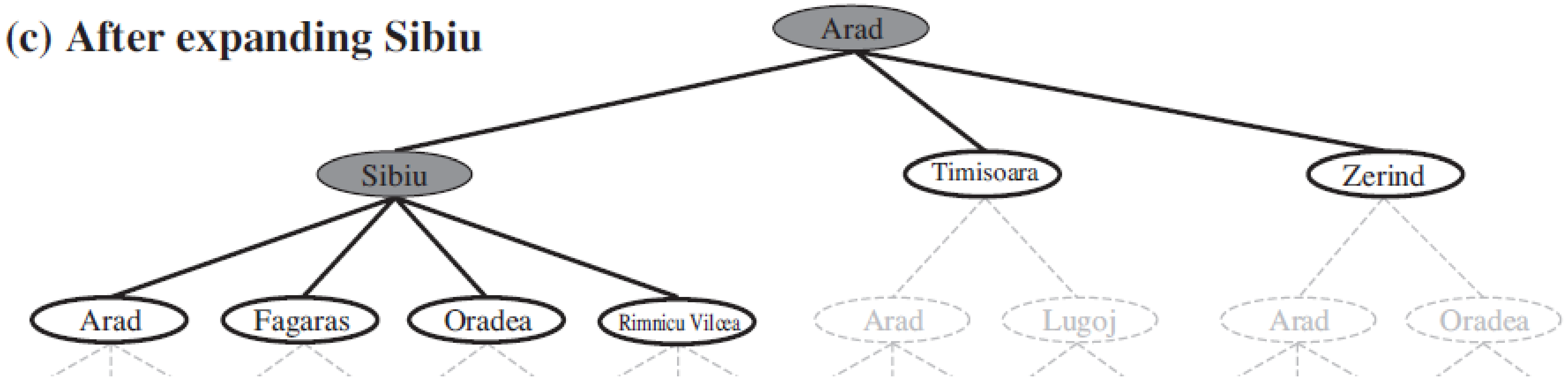
Search strategy: Method for deciding which node is expanded next.

Open list: Set of all *nodes* that currently are candidates for expansion. Also called frontier.

Closed list: Set of all *states* that were already expanded. Used only in graph search, not in tree search (up next). Also called explored set.

Repeated States

- … lead to loopy path

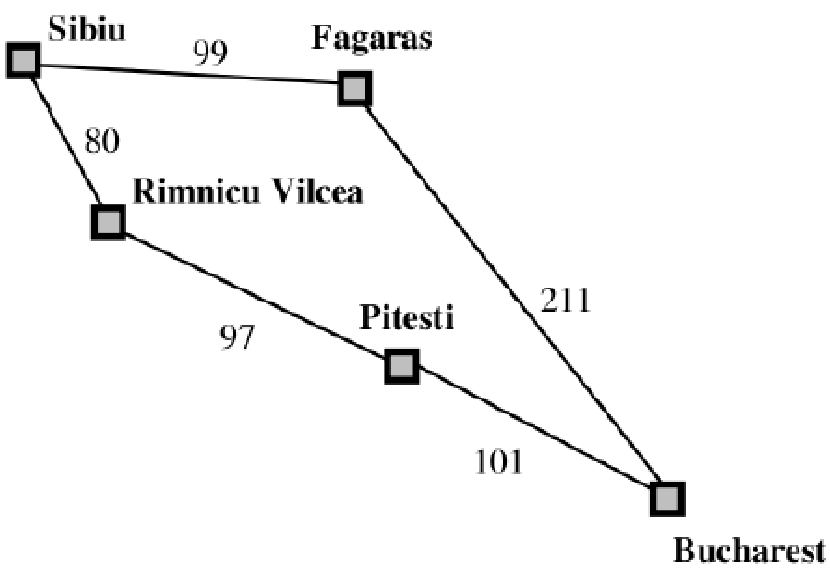


- Loopy paths can never contribute to the optimal solution

Redundant Paths

- Two possible paths from Sibiu to Bucharest
- The route via Fagaras is a more costly way to get to Bucharest

  99 + 211 = **310** vs. 80 + 97 + 101 = **278**



Tree Search vs. Graph Search

- Tree search
  - We assume that the search space has tree structure
  - When performing tree search, we do not remember visited nodes, because one node can only be visited via exactly one path from the root of the tree (which represents the initial state)
  - However, with tree search on a graph we will not know whether we generate repeated states
- Graph search
  - Remember visited nodes (keep a closed list)
  - Use *duplicate elimination*: If a generated state is in the closed list, skip it, otherwise explore it

Comparing Tree Search with Graph Search

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

** a leaf in the expanded region (the frontier) of the search graph/tree*

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

Criteria for Evaluating Search Strategies

Guarantees:

Completeness: Is the strategy guaranteed to find a solution when there is one?

Optimality: Are the returned solutions guaranteed to be optimal?

Complexity:

Time Complexity: How long does it take to find a solution? (Measured in generated states.)

Space Complexity: How much memory does the search require? (Measured in states.)

Typical state space features governing complexity:

Branching factor b: How many successors does each state have?

Goal depth d: The number of actions required to reach the shallowest goal state.

Systematic (Uninformed, Blind) Search Strategies

- **No** information on the length or cost of a path to the solution

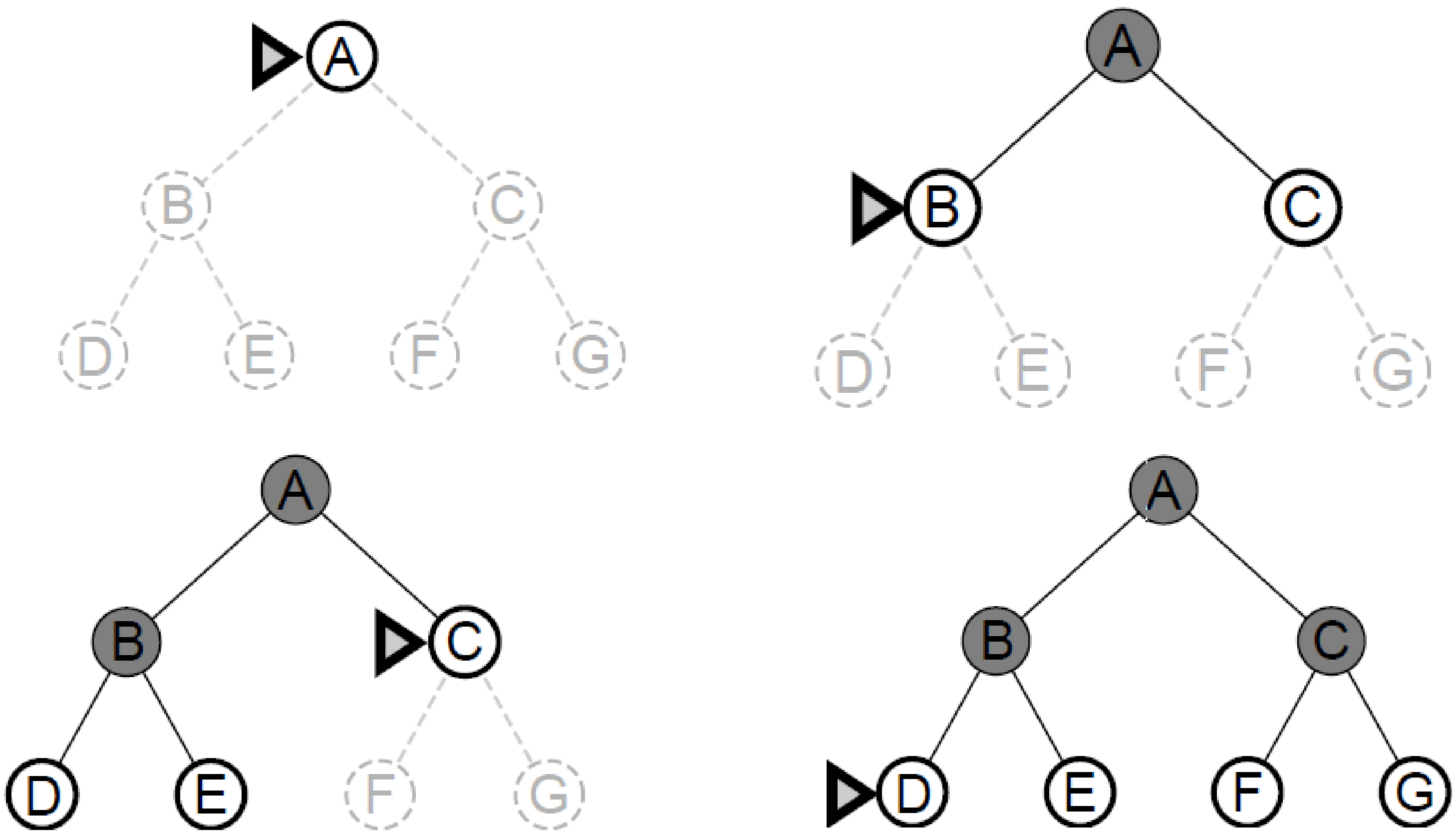
1) Breadth-first search
2) Depth-first search
3) Depth-limited search
4) Iterative deepening search

- **Only current path costs** influence search

5) Uniform cost search

(1) Breadth-First Search (BFS)

- Nodes are expanded in the order they are produced
  - the frontier is a FIFO queue



BFS Algorithm

```
function BREADTH-FIRST-SEARCH(problem) returns a solution or failure
    node ← a node with node.State = problem.InitialState
    PathCost = 0
    if GoalTest(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 ← ChildNode(problem, node, action)
            if child.State is not in explored or frontier then
                if GoalTest(child.State) then return Solution(child)
                frontier ← Insert(child, frontier)
```

- Duplicate check against explored set and frontier: No need to re-generate a state already in the (current) last layer
- Goal test at node-generation time (as opposed to node-expansion time): We already know this is a shortest path so can just stop

Properties of BFS

- Always finds the shallowest goal state first
- Completeness is obvious
  - Incomplete for search spaces with infinite branching (non-finite action space)
- The solution is optimal, provided every action has identical, non-negative (unit) costs

➢ The Romania travel example has <u>non-unit action costs</u>

➢ The solution found is sub-optimal

Time and Space Complexity of BFS

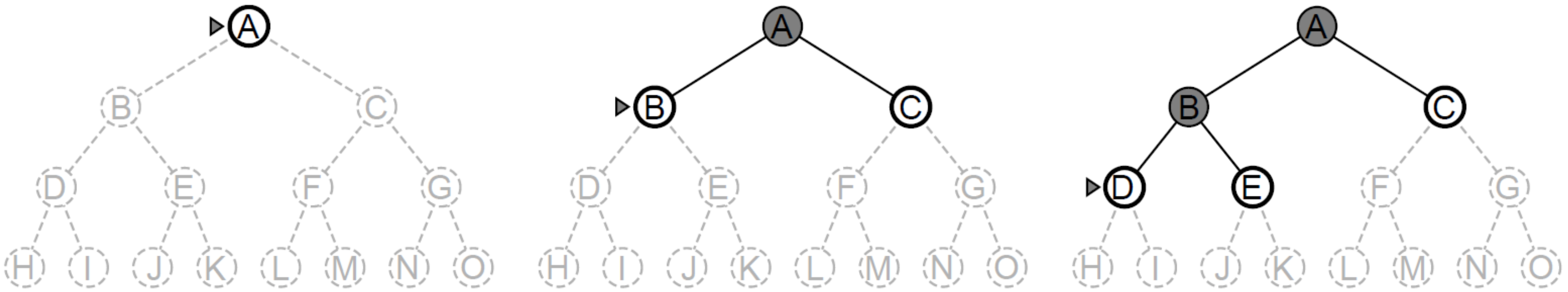
- Time Complexity
  - let *b* be the maximum branching factor and *d* the depth of a solution path
  - Maximum number of nodes expanded is

$$b + b^2 + b^3 + \ldots + b^d = \sum_{n=1}^{d} b^n \in O\left(b^d\right)$$

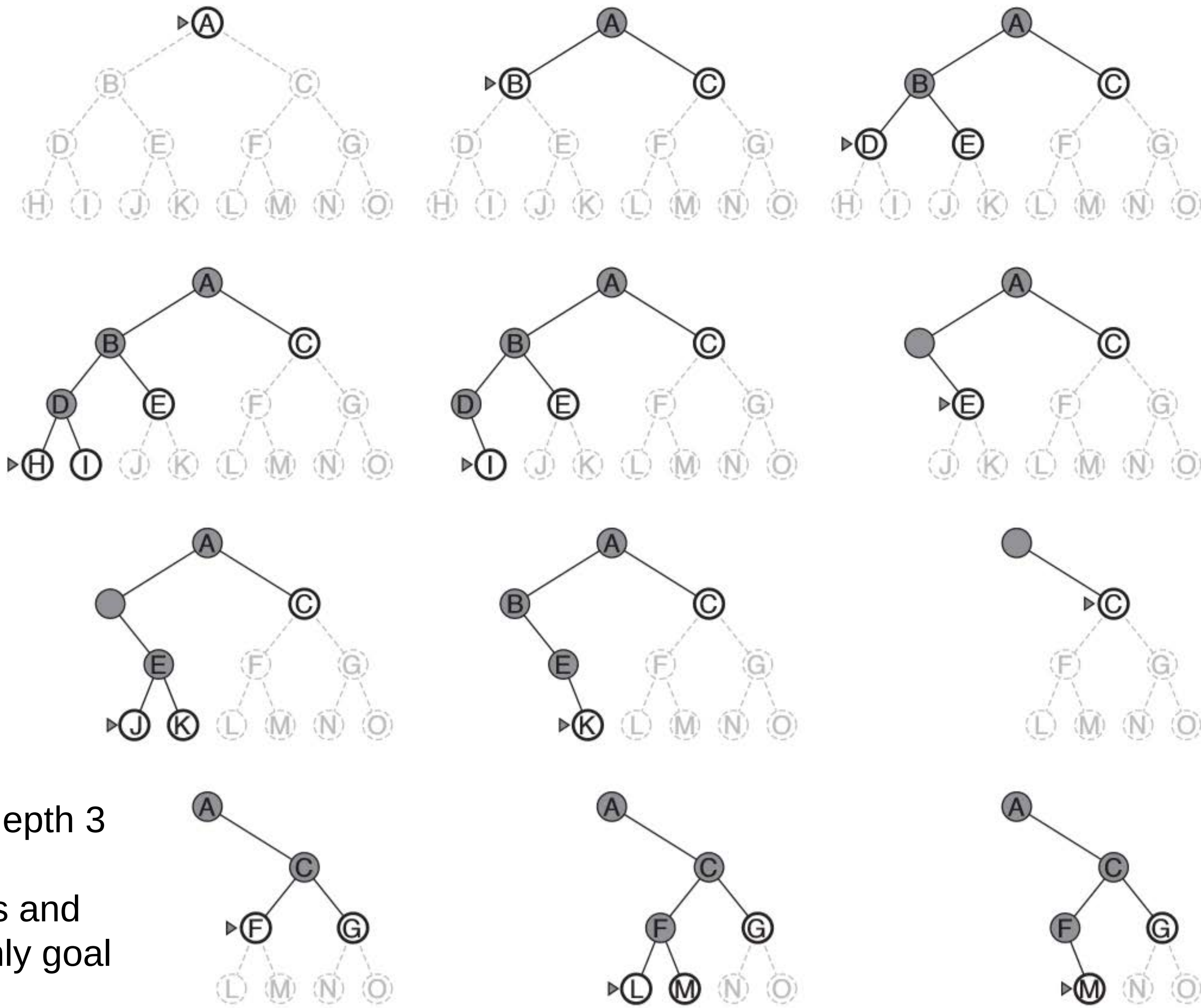
- Space Complexity
  - every node generated is kept in memory: $\sum_{n=1}^{d} b^n$
  - space needed for the frontier is: $O(b^d)$
  - space needed for the explored set: $O(b^{d-1})$

(2) Depth-First Search (DFS)

- Always expand the deepest (most recent) node in the frontier
  - the frontier is a a LIFO queue
  - when a node has no children, search backs up to the next deepest node that has unexplored children



Example of DFS



Nodes at depth 3 have no successors and M is the only goal node

DFS Algorithm

function RECURSIVE DEPTH-FIRST SEARCH(n, problem) **returns** a solution or failure
    if problem.GoalTest(n.State) **then return** the empty action sequence
    for each action a **in** problem.Actions(n.State) **do**
        $n' \leftarrow$ ChildNode(problem, n, a)
        result $\leftarrow$ RECURSIVE DEPTH-FIRST SEARCH(n', problem)
        if result $\neq$ failure **then return** $a \circ$ result
    return failure

Properties of DFS

- In general, solution found is not optimal

- Incomplete!
- Completeness can be guaranteed only for graph search (we need to remember the visited nodes) or acyclic finite state spaces
  – In infinite state spaces, descends forever on infinite paths
  – Tree search may loop forever in repeated states

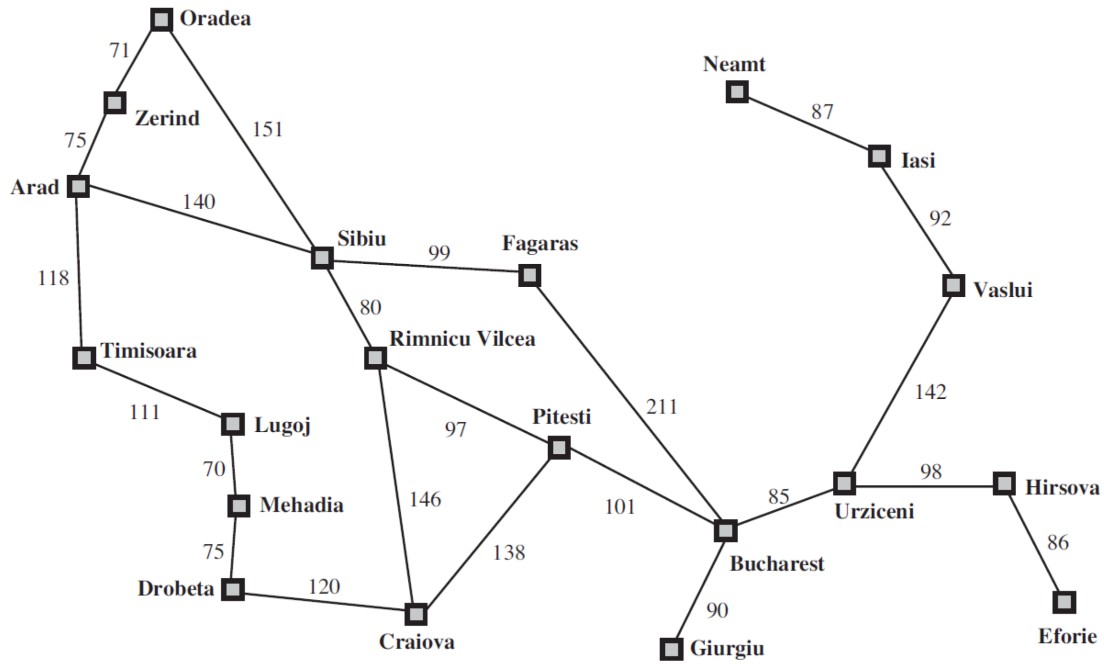
Time and Space Complexity of DFS

- **Time complexity** is: $O(b^m)$
  - where m is the maximum depth of the graph (longest path)
  - in the worst case all nodes have to be visited until a solution is found
  - this can be even larger than the state space if we do not remember already visited nodes (on graphs)

- **Space complexity** is: $O(bm)$ or $O(m)$
  - we need $O(m)$ to store the nodes along the current path and $O(b)$ to store all neighbours (open list at each level)
  - with clever indexing (backtracking search), we can save $O(b)$ and compute the neighbors dynamically in an efficient way

(3) Depth-Limited Search (DLS)

- Depth-first search with an imposed cutoff on the maximum depth of a path
  - e.g., route planning: with n cities, the maximum depth is n -1
  - in the example a depth of 9 is sufficient - every city can be reached in at most 9 steps



Depth-Limited Search Algorithm

function DEPTH-LIMITED-SEARCH(*problem*, *limit*) **returns** a solution or failure/cutoff
  return RECURSIVE-DLS(MakeNode(*problem*.InitialState, *problem*, *limit*))

function RECURSIVE-DLS(*node*, *problem*, *limit*) **returns** a solution or failure/cutoff
  if GoalTest(*node*.State) **then return** *Solution*(*node*)
  else if $limit = 0$ **then return** *cutoff*
  else
    cutoffOccurred ← false
    for each *action* **in** Actions(*node*.State) **do**
      child ← ChildNode(*problem*, *node*, *action*)
      result ← RECURSIVE-DLS(*child*, *problem*, *limit*-1)
      if $result = cutoff$ **then** *cutoffOccurred* ← true
      else if $result \neq failure$ **then return** *result*
    if *cutoffOccurred* **then return** *cutoff*
    else return *failure*

Limit must not be smaller than the depth of the shallowest goal state, otherwise DLS is incomplete

Properties and Complexity of DLS

- Complete if the depth limit is larger than length of shortest solution
- First solution found may not be optimal

- Time and space complexity as with DFS, but $m = l$ (the depth-limit)

- **Time complexity**: $O(b^l)$
- **Space complexiy**: $O(bl)$ or $O(l)$ with backtracking search

(4) Iterative Deepening Search (IDS)

- Use depth-limited search and in every iteration increase search depth by 1

function ITERATIVE-DEEPENING-SEARCH(*problem*) **returns** a solution or failure
    for $depth = 0$ **to** ∞ **do**
        $result \leftarrow$ DEPTH-LIMITED-SEARCH(*problem*, *depth*)
        if $result \neq cutoff$ **then return** *result*

Illustration of IDS

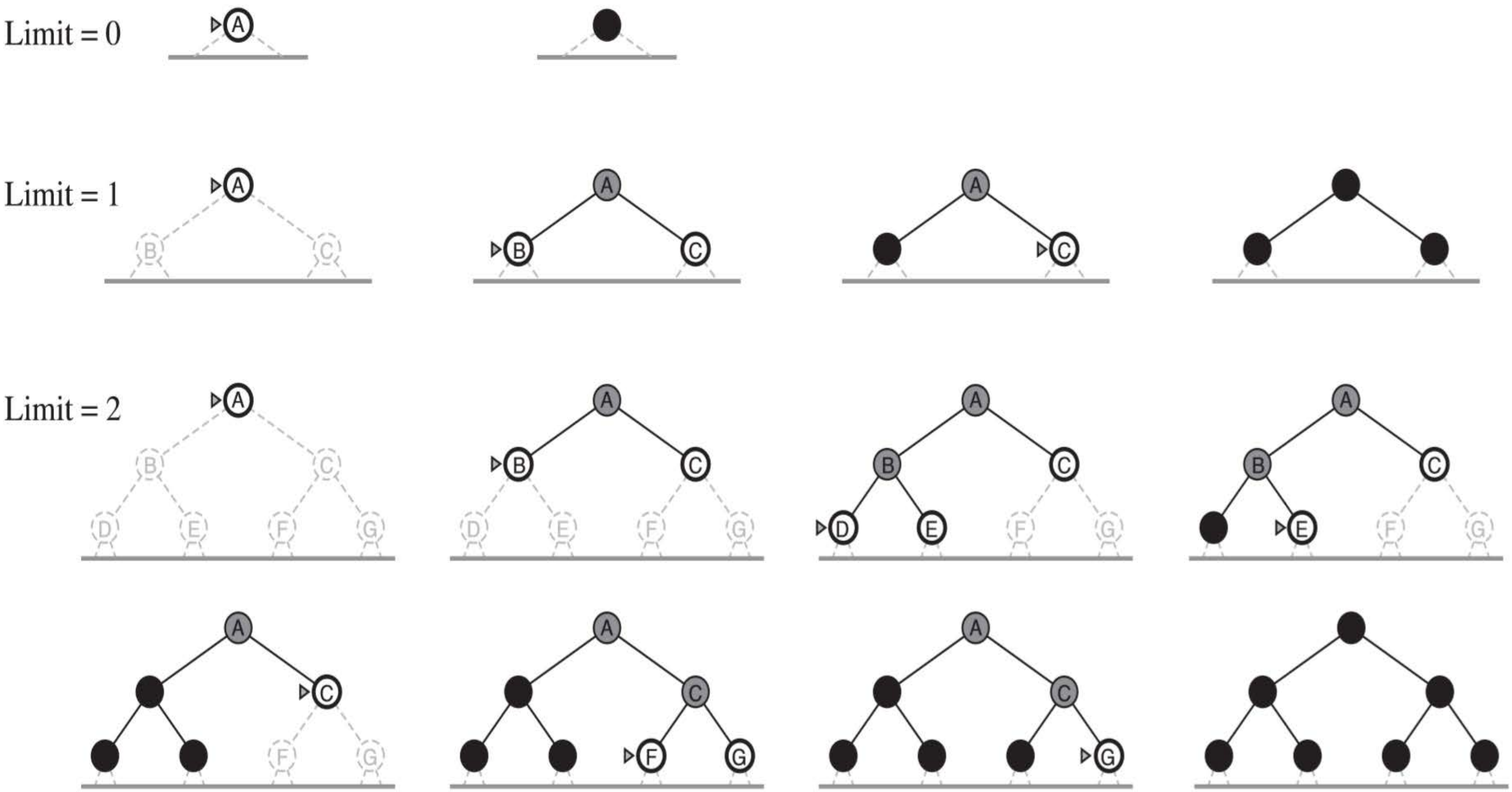
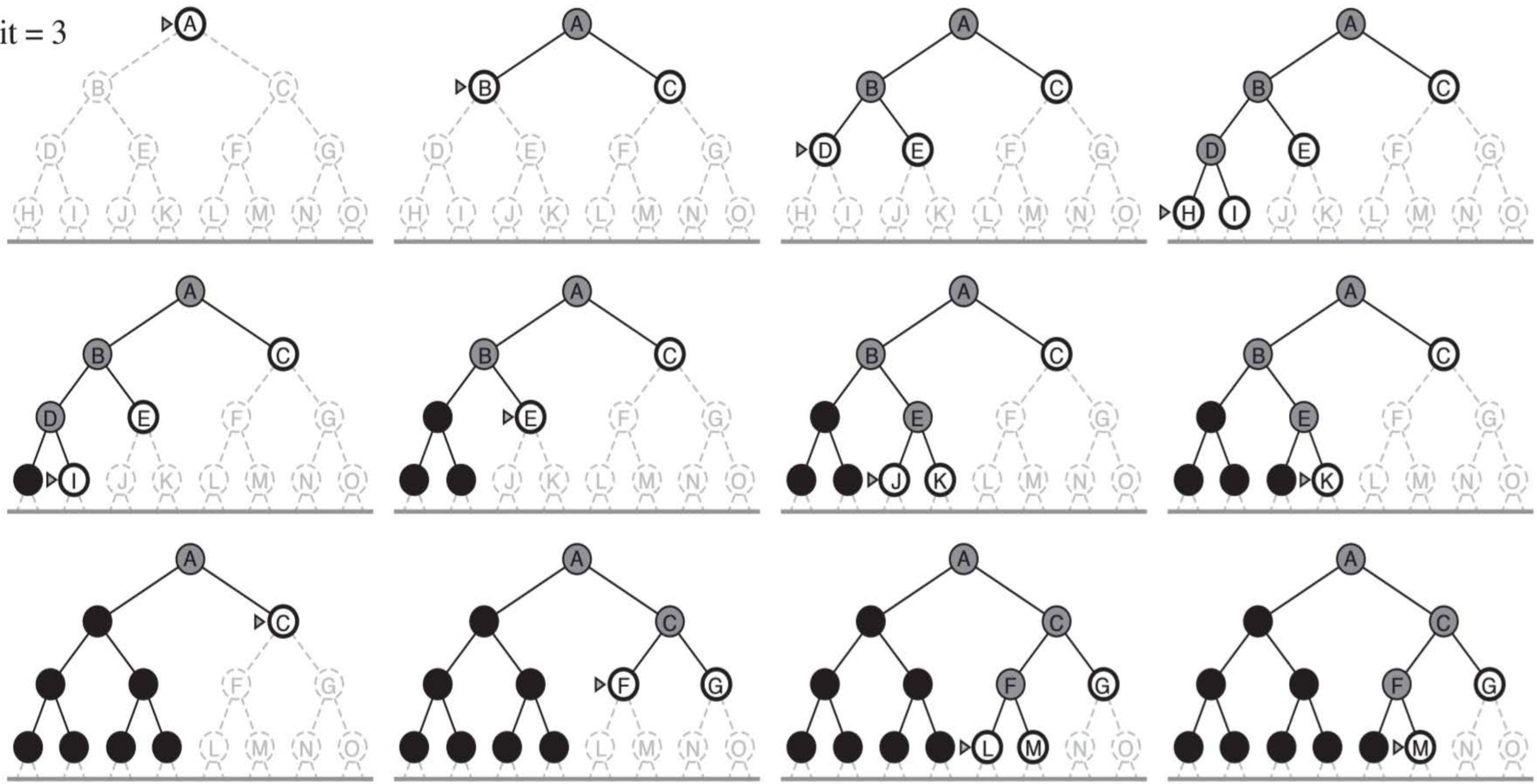


Illustration Continued

Limit = 3



Properties of IDS

- Combines advantages of BFS and DFS
- Optimal for unit action costs
  - extension to general action costs possible
- Complete (for finite branching)
- Complexity as for DLS
  - Time: $O(b^l)$
  - Space: $O(bl)$ or $O(l)$ with backtracking search

Time complexity:

Breadth-First-Search	$b + b^2 + \cdots + b^{d-1} + b^d \in O(b^d)$
Iterative Deepening Search	$(d)b + (d-1)b^2 + \cdots + 3b^{d-2} + 2b^{d-1} + 1b^d \in O(b^d)$

Example: $b = 10, d = 5$

Breadth-First Search	$10 + 100 + 1,000 + 10,000 + 100,000 = 111,110$
Iterative Deepening Search	$50 + 400 + 3,000 + 20,000 + 100,000 = 123,450$

$= 5 \times 10^1 + 4 \times 10^2 + 3 \times 10^3 + 2 \times 10^4 + 1 \times 10^5$

(5) Uniform-Cost Search (UCS)

- Consider the **path costs** for each node $g(n)$
- Organize the frontier as a priority queue and expand the node with the lowest path costs first
- Finds an optimal solution if all actions have non-negative costs and if

$$g(successor(n)) \geq g(n)$$

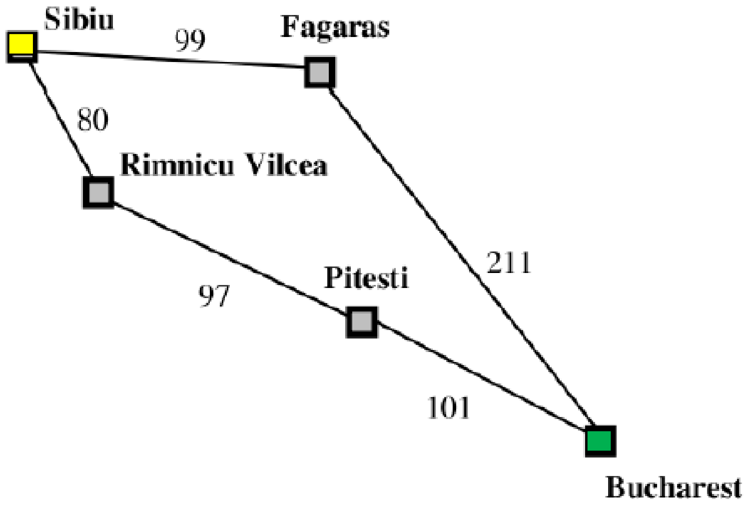
for all n.

UCS Algorithm

function UNIFORM-COST SEARCH(problem) **returns** a solution or failure
  node $\leftarrow$ a node n with n.State = problem.InitialState
  frontier $\leftarrow$ a priority queue ordered by ascending g, only element n
  explored $\leftarrow$ empty set of states
  loop
    if Is.Empty(*frontier*) **then** **return** failure
    $n \leftarrow$ Pop(*frontier*)
    if problem.GoalTest(n.State) **then return** Solution(n)
    explored $\leftarrow$ explored $\cup$ n.State
    for action a **in** problem.Actions(n.State) **do**
      $n' \leftarrow$ ChildNode(problem, n, a)
      if n'.State $\notin$ [explored $\cup$ States(*frontier*)] **then** Insert(n', $g(n')$, *frontier*)
      else if ex. $n'' \in$ *frontier* s.t. n''.State = n'.State and $g(n') < g(n'')$
      then replace n'' in *frontier* with n'

- Goal test at node-expansion time
- Duplicates in frontier replaced in case of cheaper path

Example of UCS



1) S
2) RV (80), F (99)
3) F (99), P (177), S is pruned
4) P(177), B (via F 99 + 211) = 310
5) B (via P 177 + 101) = 278
6) Replace B(310) with B(278)
7) Expand B (278), all pruned

Properties and Complexity of UCS

- Optimal for non-negative action costs
  - whenever a node is selected for expansion, the optimal path to this node has been found
  - does not care about the number of actions on a path, but only about the total path costs
    - will get stuck on infinite paths with zero-action costs
- Complete if all action costs > 0

- Time and space complexity: $O\left(b^{1+\lfloor C^*/\varepsilon \rfloor}\right)$
  - C* path cost of optimal solution, action costs $\geq \varepsilon$
  - if all action costs are equal then b^{d+1}

UCS and Dijkstra´s Algorithm

Lemma. Uniform-cost search is equivalent to Dijkstra‘s algorithm on the state space graph. (Obvious from the definition of the two algorithms)

The only differences are:

(a) We generate only a part of that graph incrementally, whereas Dijkstra inputs and processes the whole graph

(b) We stop when we reach any goal state (rather than a fixed target state given the input)

Dijkstra‘s algorithm:

https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm

Initialise the cost of each node to ∞ and the cost of the source to 0

While there are unknown nodes left in the graph

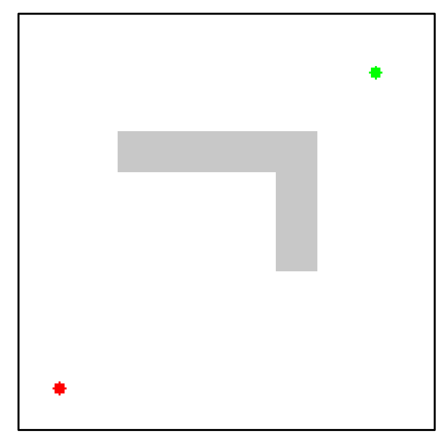
    Select an unknown node with the lowest cost and mark as known

    For each node b adjacent to a

        If $cost(a) + cost(a,b) < cost(b)$ do

            $cost(b) = cost(a) + cost(a,b)$

            $parent(b) = a$



Overview on Algorithm Properties

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

Where:

- b branching factor
- d depth of solution
- m maximum depth of the search tree
- l depth limit
- C^* cost of the optimal solution
- ε minimal cost of an action

Superscripts:

- a b is finite
- b if step costs not less than ε
- c if step costs are all identical
- d if both directions use breadth-first search
- e Yes for finite search spaces
- f $O(b)$ can be eliminated by backtracking search

Summary

- IDS is the preferred uninformed search method when there is a large search space and the depth (length) of the solution is not known
- DFS is often used because of its minimal memory requirements
  - compact encodings of exponential-size explored node set exists
- BFS is rarely found in practice
  - this does not mean that there are no applications for which this would be the search methods of choice!
- DLS prevents infinite descends on infinite paths

Working Questions

1. Which concepts are used to describe search problems?
2. Which concepts are used to describe search algorithms and search spaces?
3. What is the difference between tree and graph search?
4. What is the set of explored nodes used for?
5. Why don’t we need a set of explored nodes when the search space is a tree?
6. Can you explain how BFS, DFS, DLS, IDS, UCS work?
7. What properties are used to characterize search algorithms?
8. Compare uninformed search methods based on time complexity, space complexity, optimality, completeness.

