UNIVERSITÄT DES SAARLANDES

Artificial Intelligence

Systematic (Uninformed) Search

Prof. Dr. habil. Jana Koehler

Dr. Sophia Saller, M. Sc. Annika Engel

Summer 2021

Agenda

- Basic Terminology and Concepts
- Modeling a Search Problem
- Systematic (Uninformed, Blind) Search Strategies
  - Tree Search vs. Graph Search
  - Breadth-First search
  - Depth-First search
  - Depth-Limited search
  - Iterative Deepening search
  - Uniform Cost search

Recommended Reading

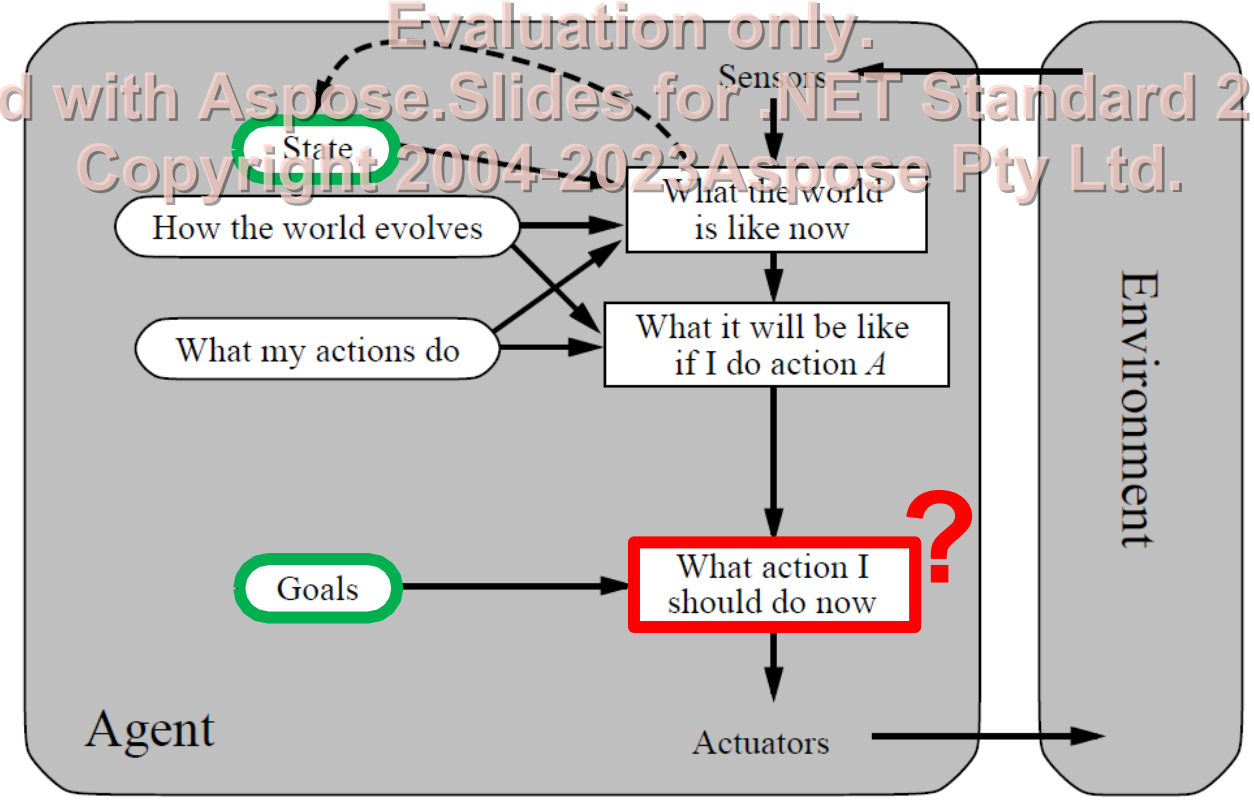
- AIMA Chapter 3: Solving Problems by Searching
  - 3.1 Problem Solving Agents
  - 3.2 Example Problems
  - 3.3 Searching for Solutions
  - 3.4 Uninformed Search Strategies, the following subchapters:
    - 3.4.1 Breadth-first search
    - 3.4.2 Uniform-cost search
    - 3.4.3 Depth-first search
   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

UNIVERSITÄT DES SAARLANDES

Basic Terminology and Concepts

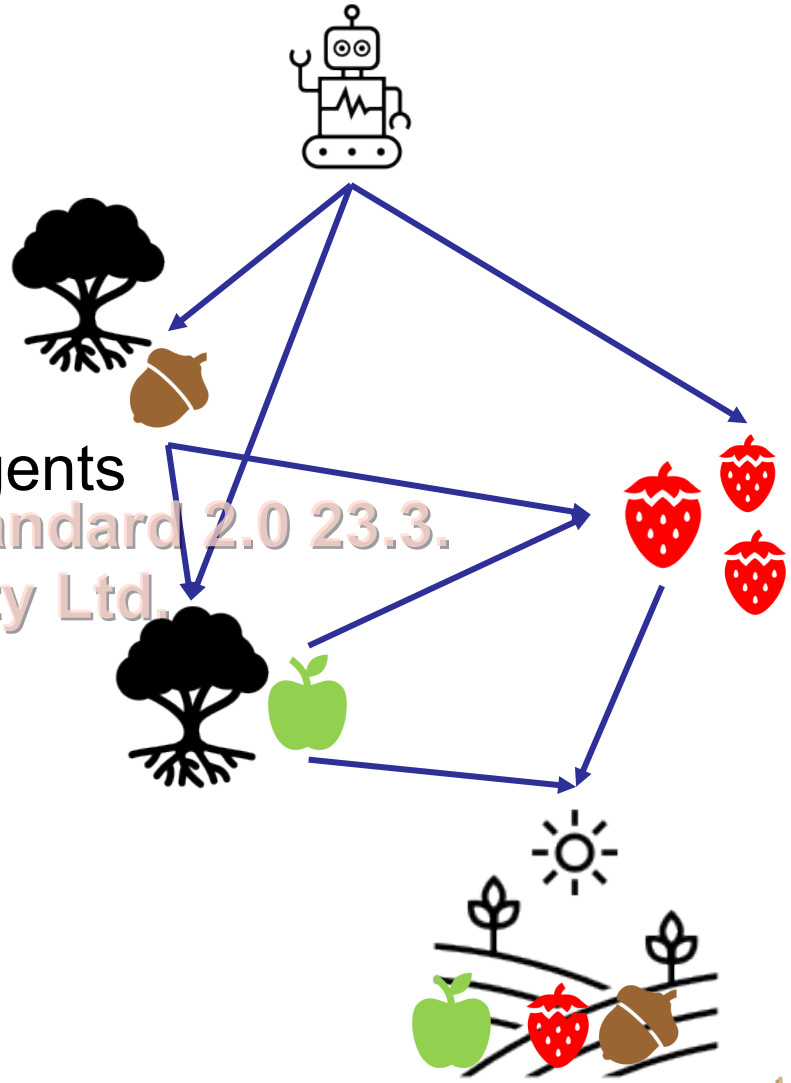
How can a Goal-based Agent reach a Goal?

- 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



Definition: State Space

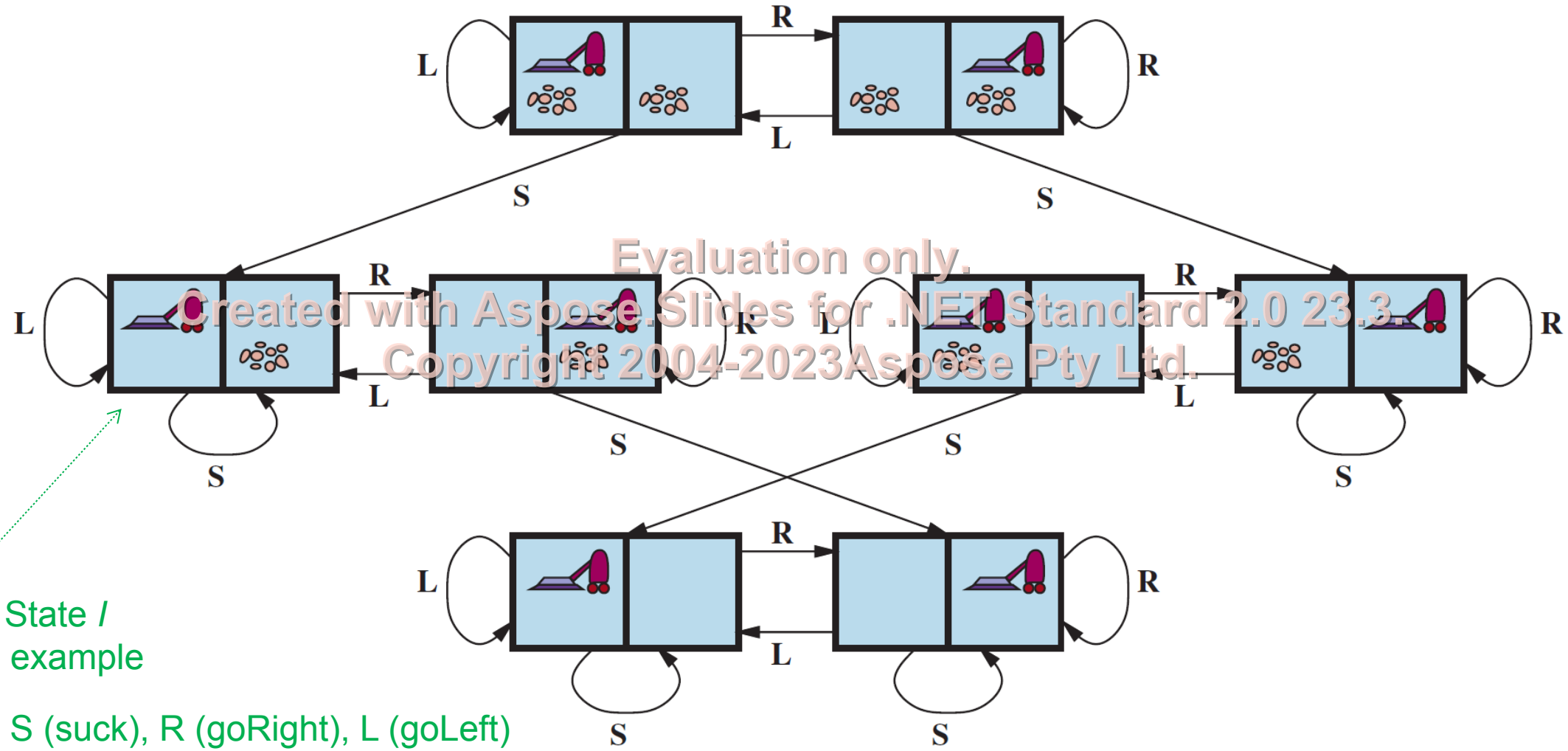
A state space is a 4-tuple $\Theta = (S, A, T, I)$ where:

- S is a finite set of states
- A is a finite set of actions
- $T \subseteq S \times A \times S$ is the transition relation
- $I \in S$ is the initial state

Note:

- 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
- 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
- The state space forms a directed graph in which the nodes are states and the links between nodes are actions

Vacuum Cleaner World: Example of a State Space Graph



Initial State *I*
of the example

Actions S (suck), R (goRight), L (goLeft)
8 possible states
Arrows annotated with actions show possible state transitions in the state space graph

Definition: Search Problem

A *search problem* is a 6-tuple $\Pi = (S, A, c, T, I, S^G)$ where:

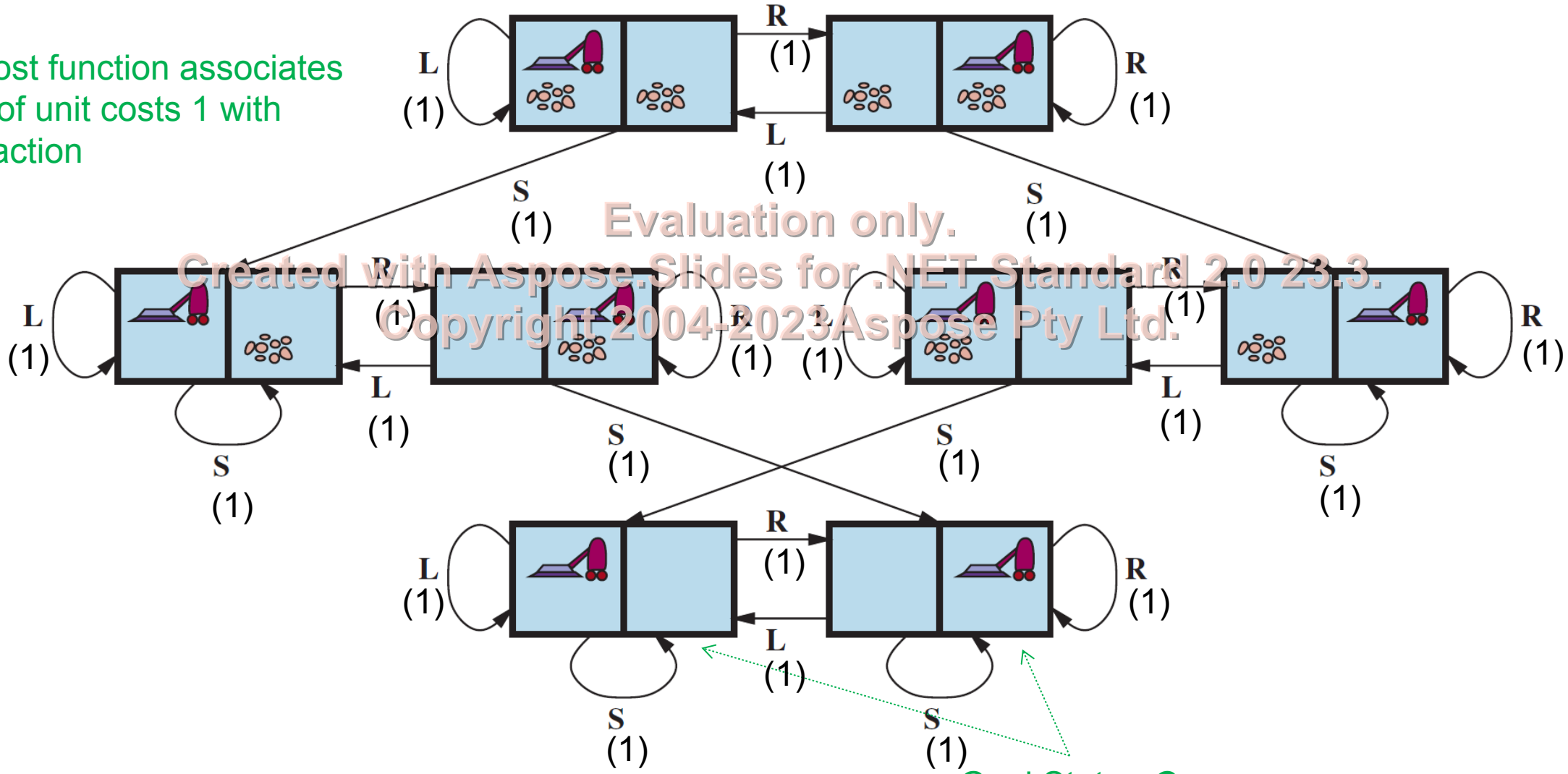
- $S, \mathrm{A}, \mathrm{T}, \mathrm{I}$ are given through the state space
- $c : S \times A \mapsto \mathbb{R}_0^+$ is the cost function
- $S^G \subseteq S$ is the set of goal states

We say that the search *problem* has unit costs if, for all $a \in A$ and all $s \in S$,

$c(s, a) = 1.$

Vacuum Cleaner World: Example of Cost Function and Goal State(s)

The cost function associates costs of unit costs 1 with each action



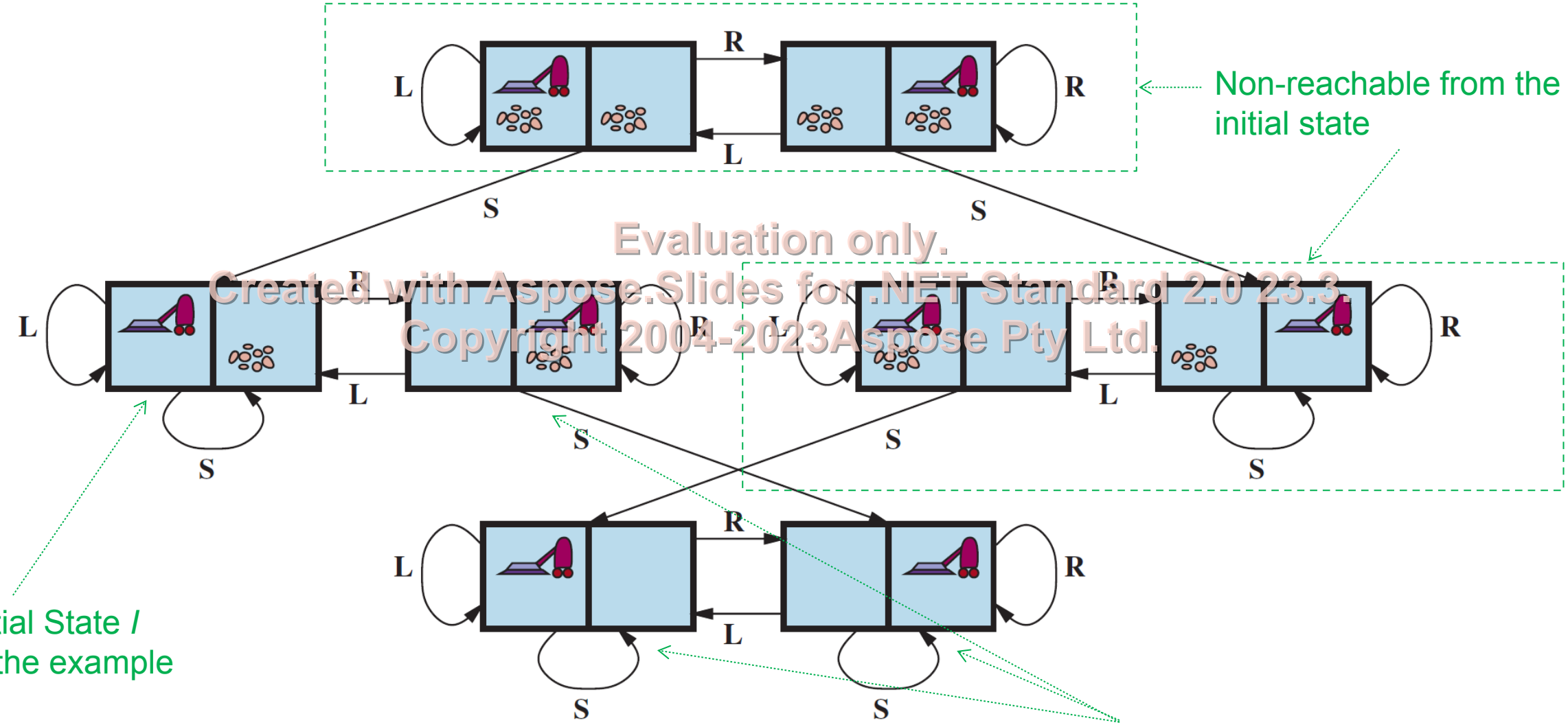
Definition: Reachability of States

- s' successor of s if $s \to s'$; s predecessor of s' if $s \to s'$
- 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 this 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 (trap state)

Vacuum Cleaner World: Example of Reachable and Non-Reachable States



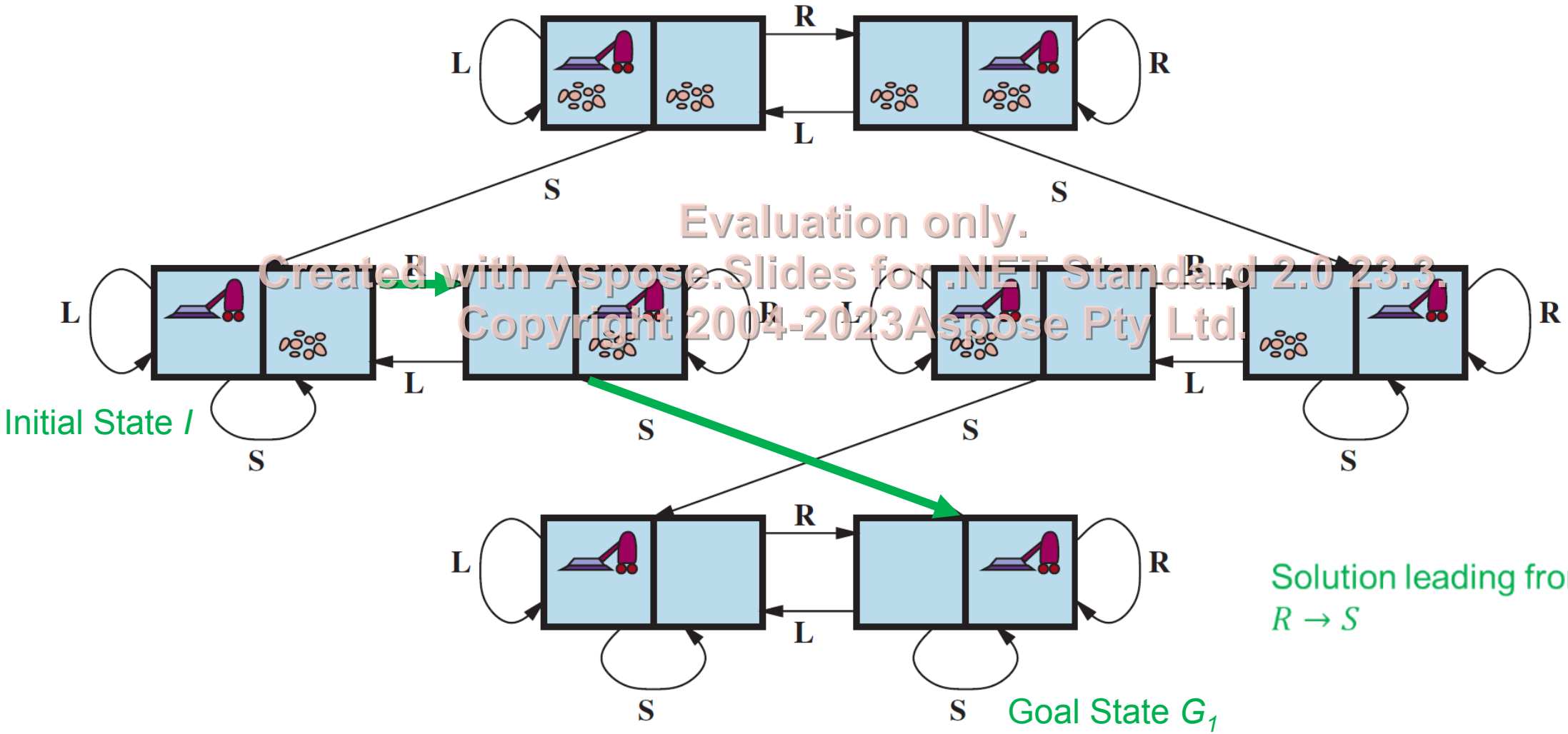
Definition: (Optimal) Problem Solution

Let $\Pi = (S, A, c, T, I, S^G)$ be a search problem, and let $s \in S$

- A solution for s is an action path $(a_1, \ldots, a_n)$ from I to s
- A solution for $s \in S^G$ 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 Θ is unsolvable

- A solution is optimal if its cost is minimal among all solutions for s. For a state s, the optimal cost $c^*(s)$ is the cost of a cheapest path reaching s

Vacuum Cleaner World: Example of a Solution



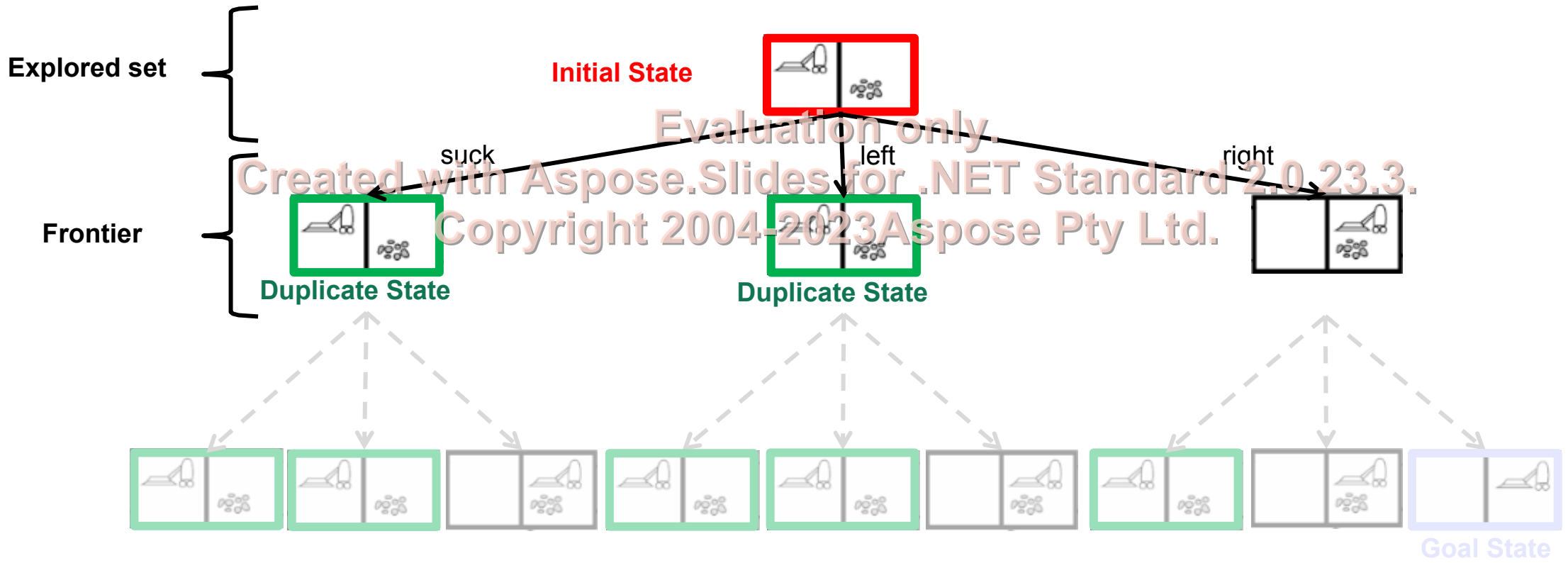
Solution leading from I to G_1:
$R \rightarrow S$

Modeling a Search Problem

Search Tree

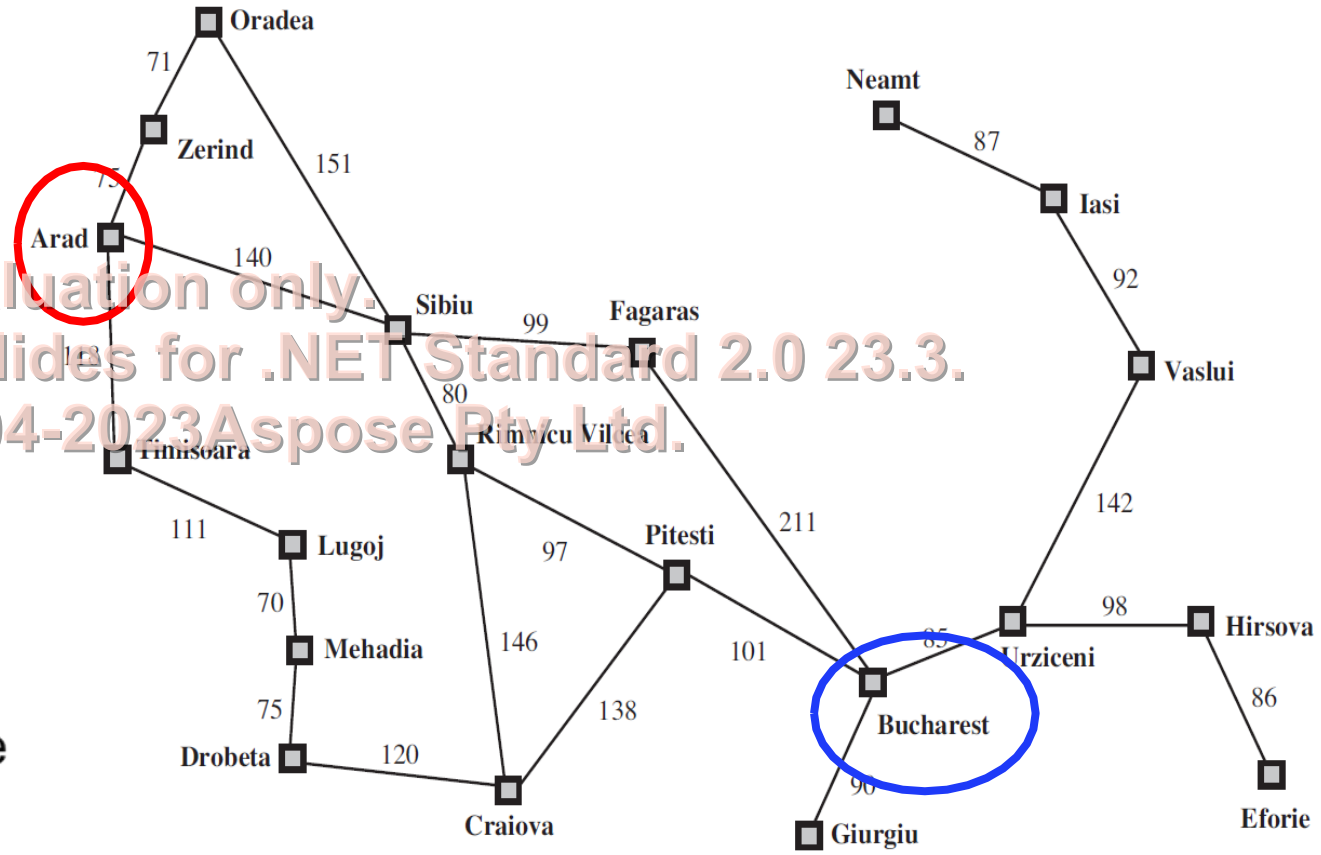
- The search node n contains a state reached by the search, plus information about how it was reached.
- Generating all successors of a node, by applying all actions applicable to the node's state s is called node expansion. We also say the state s is expanded.
- The search strategy is the method for deciding which node is expanded next.
- The frontier is a set of nodes that are candidates for expansion, also called open list.
- The explored set is the set of all states which were already expanded (visited), also called closed list.
- A duplicate (repeated) state is a state, which is generated during expansion and which is already contained in the explored set. Duplicate states never contribute to an optimal solution.

Vacuum Cleaner World: Example Search Tree



Romania Travel Example: State Space

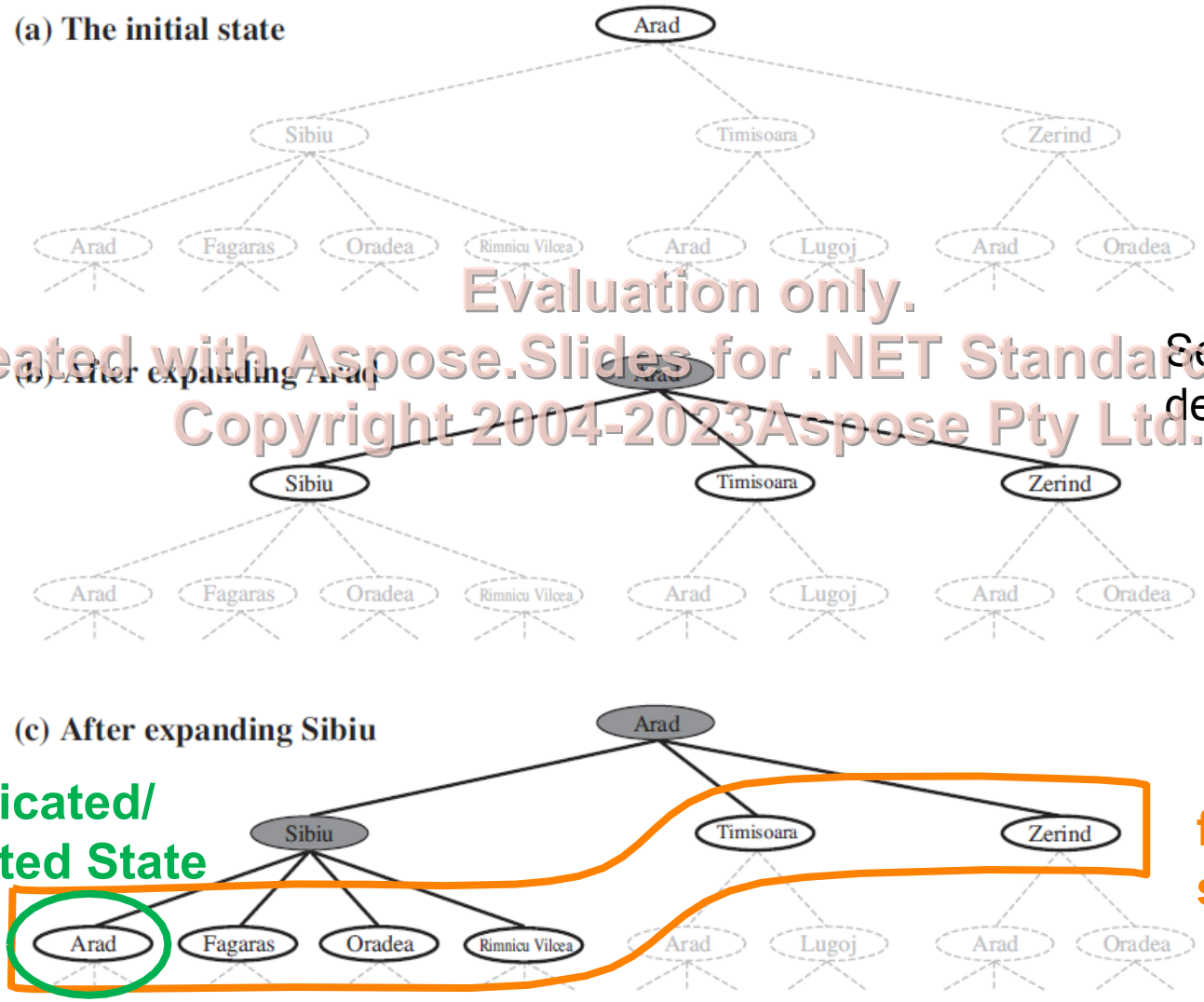
- States: $\{In(Arad), \ In(Oradea), In(Zerind), In(Sibiu), \dots\}$
- Initial state: $In(Arad)$
- Goal state: $\{In(Bucharest)\}$
- Example actions: applicable actions in the state $In(Arad)$ are $\{Go(Sibiu), Go(Timisoara), Go(Zerind)\}$
- Example transition from Arad to Zerind is given by $(In(Arad), Go(Zerind), In(Zerind))$
- The cost of each action describes the time which is needed to travel from one city to another



Russell, S., & Norvig, P. (2010). Figure 3.2

UNIVERSITÄT DES SAARLANDES

Romania Travel Example: Search builds a Search Tree when expanding the State Space Graph



Search tree = sequential description of the visiting order

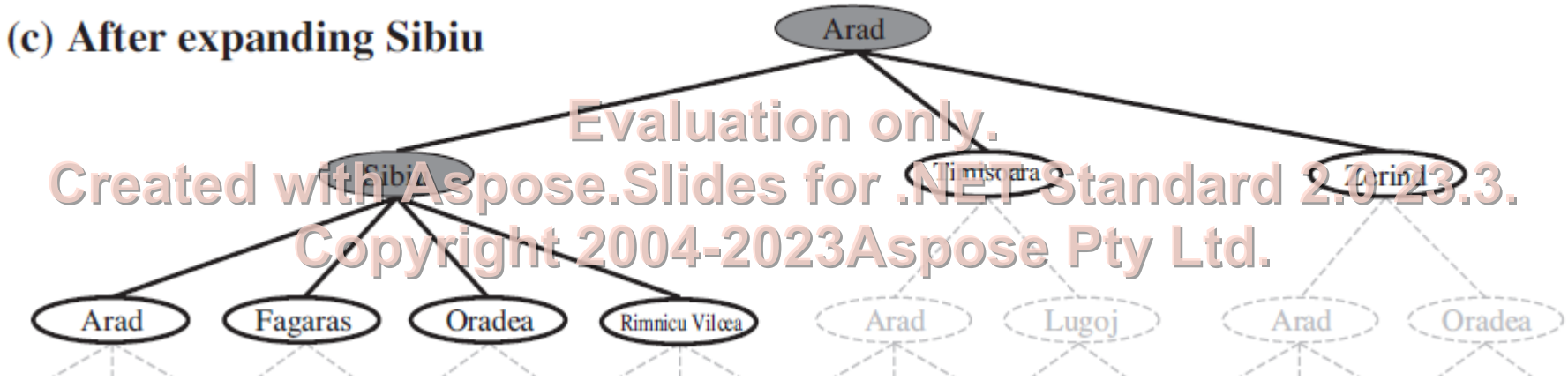
Duplicated/ Repeated State

frontier of the search

UNIVERSITÄT DES SAARLANDES

Romania Travel Example: Duplicate / Repeated States

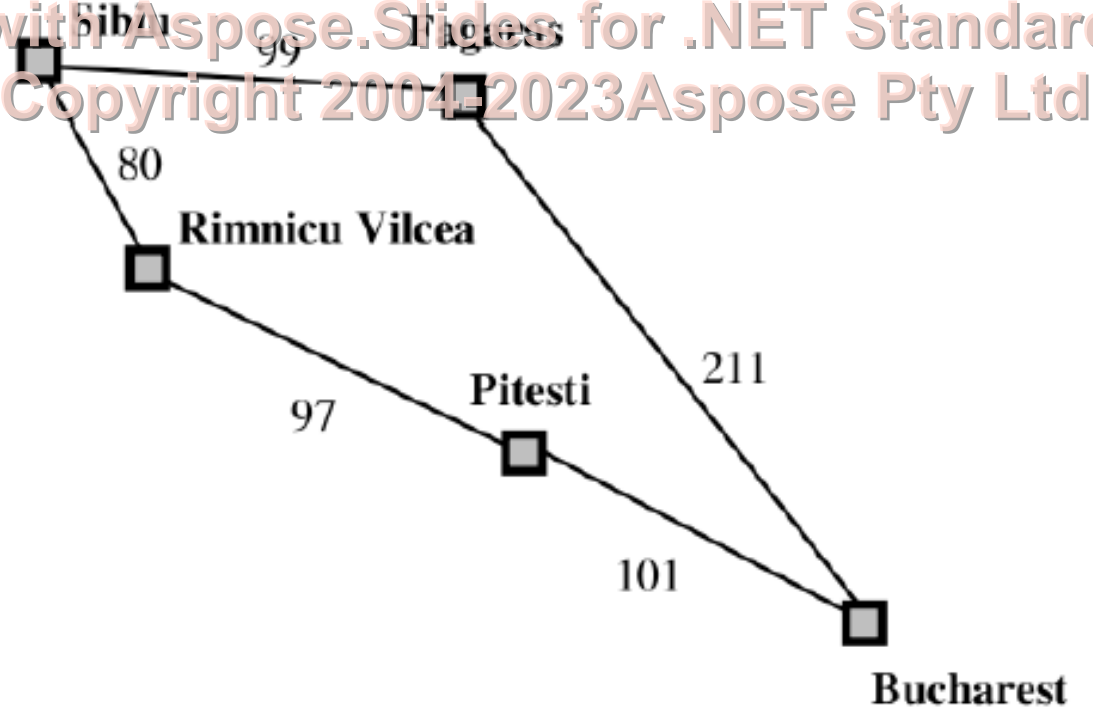
§ … created by loopy paths in the state space graph



§ Loopy paths can never contribute to the optimal solution

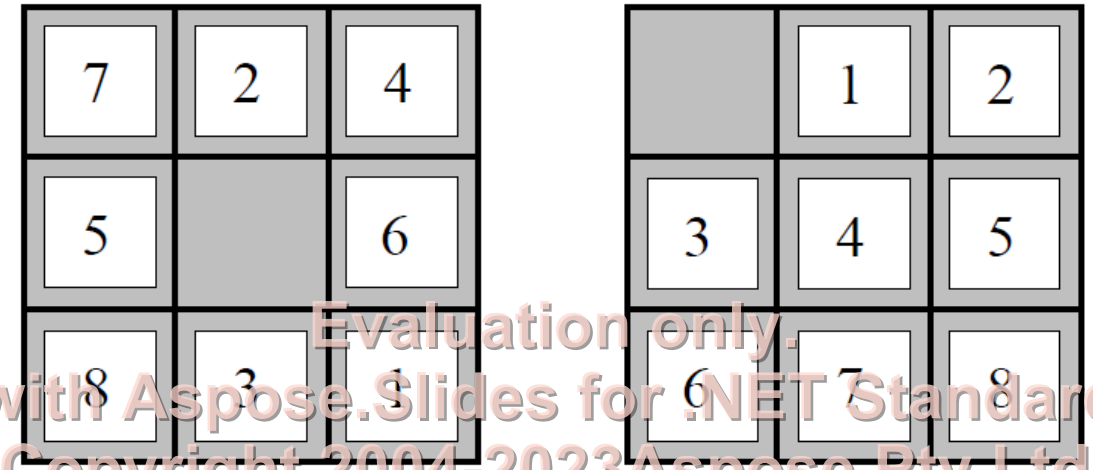
Romania Travel Example: Redundant Paths

- 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**



UNIVERSITÄT DES SAARLANDES

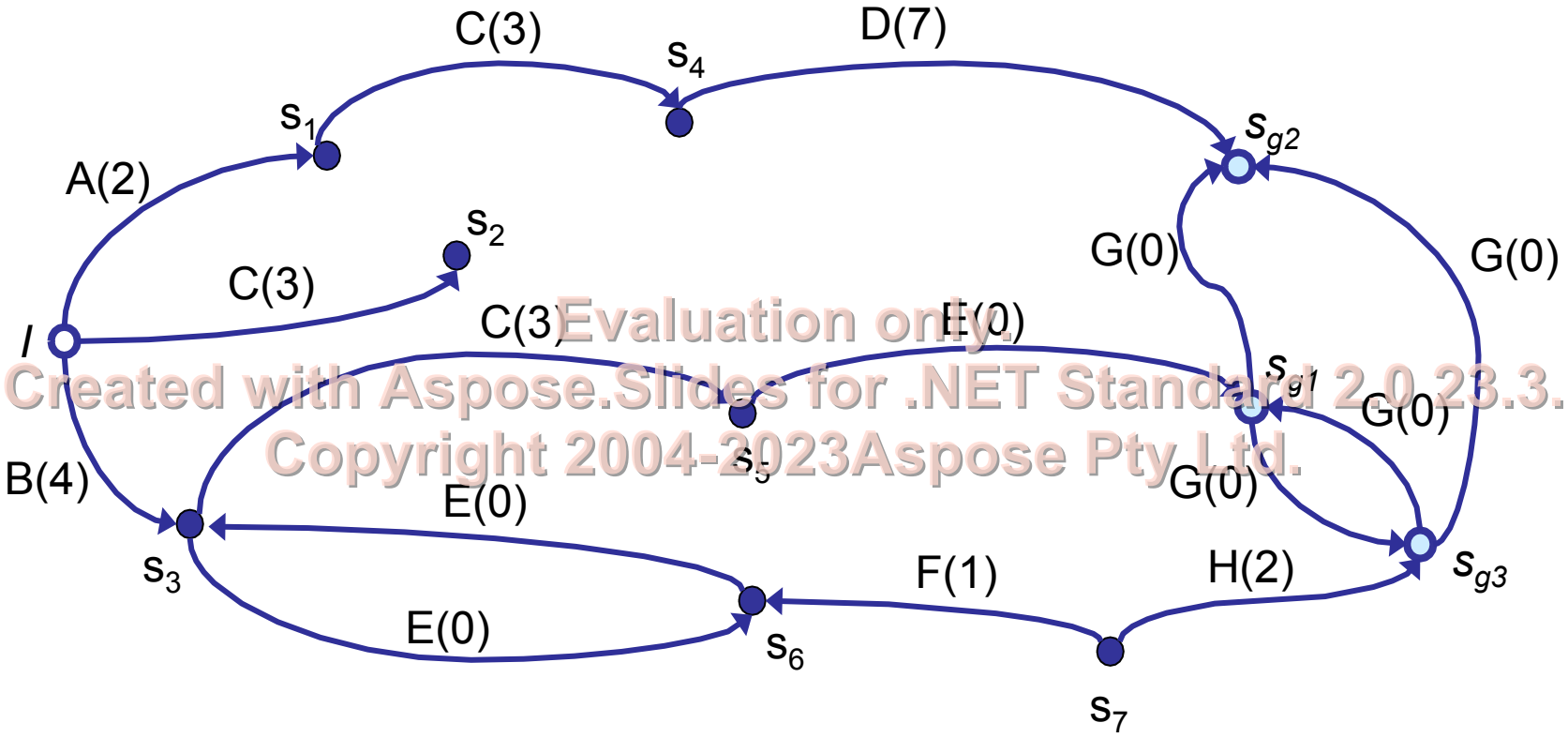
The 8-Puzzle Example



Start State Goal State

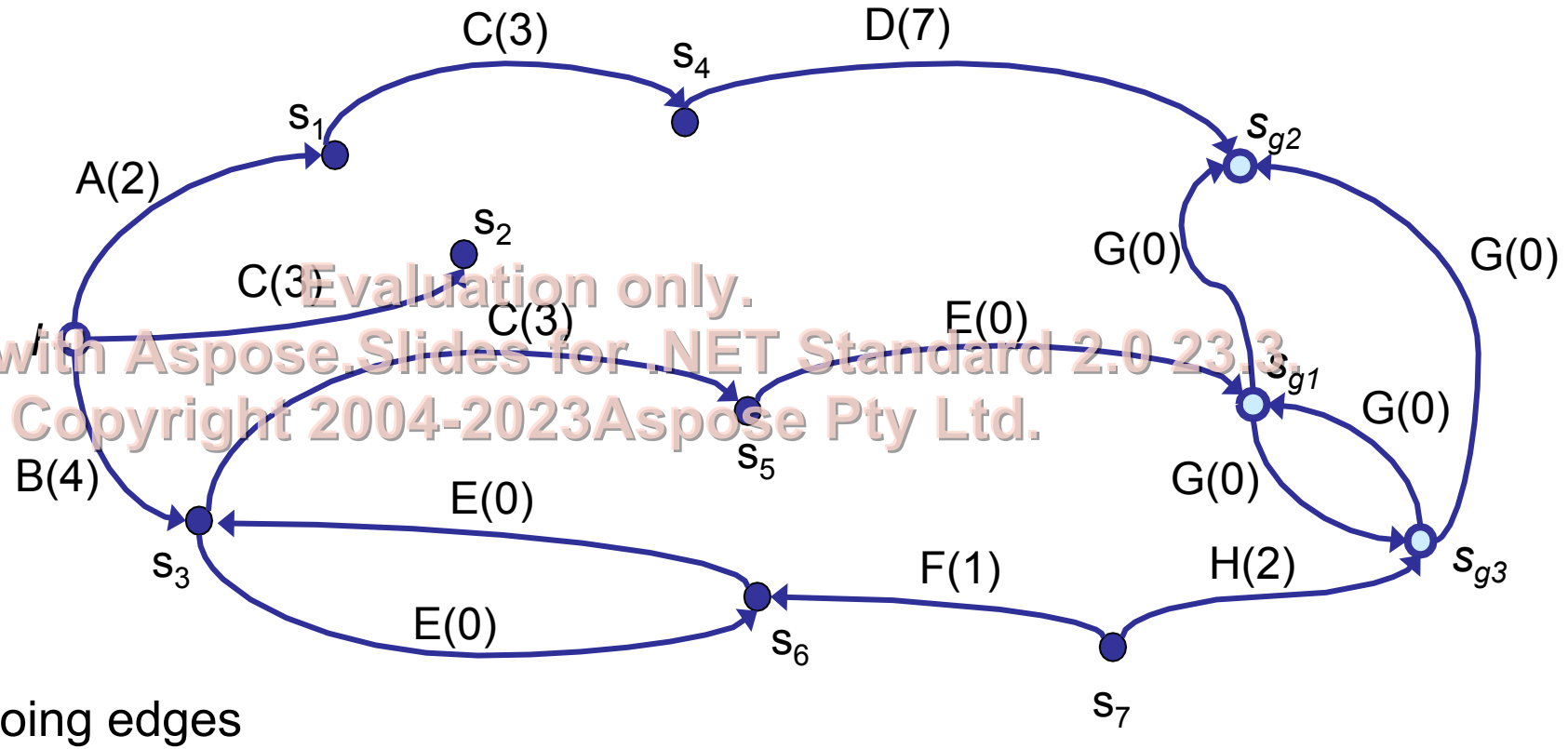
- States: _____
- Initial (Start) State: _____
- Actions: _____
- Goal states: _____
- Path Costs: _____

A State Space Graph with Costs



- 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})*

Reachable and Solvable States



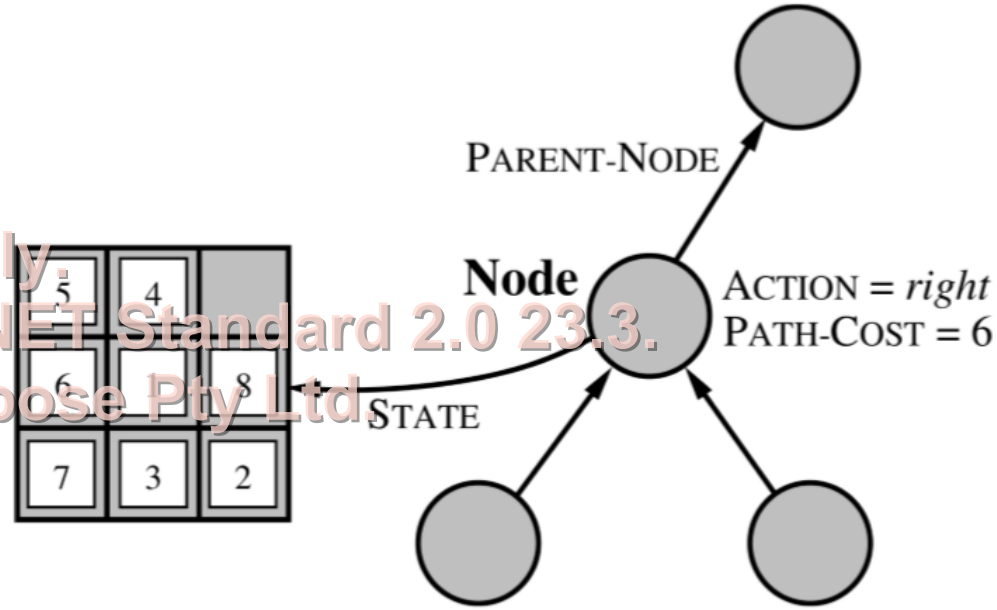
- All states reachable?
  - No: s_7 has only outgoing edges
- All states solvable?
  - No: s_2 has no outgoing edges (dead end)
- Optimal solutions?

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

Implementation – Data Structure of a Search Node

Let *Node* be a search node

- *Node*.STATE is a state from the state space which the node *Node* contains
- *Node*.PARENT-NODE is the node in the search tree whose node expansion generates the node *Node*
- *Node*.ACTION is the action which was applied to the parent node to generate the node *Node*
- *Node*.PATH-COST is the cost of the path from the initial state to the node *Node* (as indicated by the parent pointers)



Implementation – Operations on the State Space

- *problem*.InitialState returns the initial state of a problem
- GoalTest(*Node*.STATE) returns a Boolean, “true” iff a state is a goal state, otherwise “false”

- Cost(*Node*.ACTION) returns the cost function value of the action
- Actions(*Node*.STATE) returns the set of actions that are applicable to the state
- ChildState(*Node*.STATE,*Node*.ACTION) returns the outcome state *childState* if the action is applicable to the state *state*

Implementation – Operations on Search Nodes

Let *Node* be a search node

- Solution(*Node*) returns the path to node *Node*. By back chaining over *Node*.PARENT and collecting *Node*.ACTION in each step

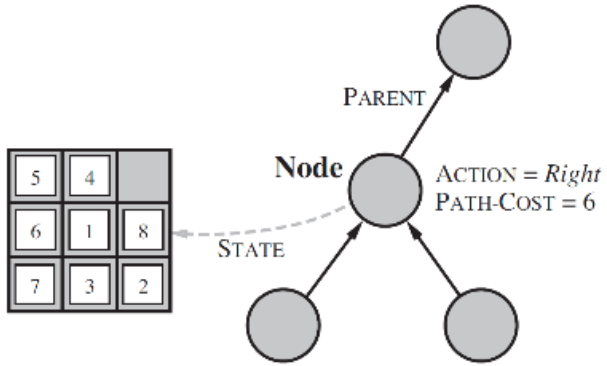
- ChildNode(*Node.STATE,Node.ACTION*) generates the outcome *childNode* corresponding to the application of the action in the state

Implementation – Operations on the Frontier

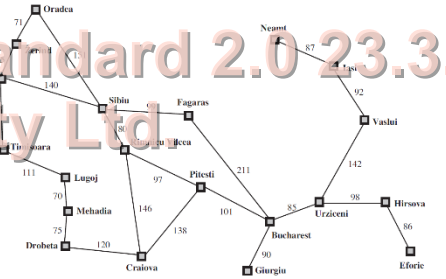
- Empty?(*frontier*) returns true iff there are no more elements in the frontier
- Pop(*frontier*) returns the first node in the frontier and removes it from the list
- Insert(*Node, frontier*) adds a node to the frontier

Formulating Search Problems

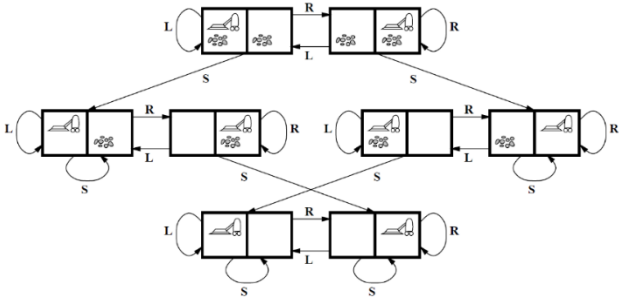
1) **Blackbox description**
   - Application programming interface (API) to construct the state space



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

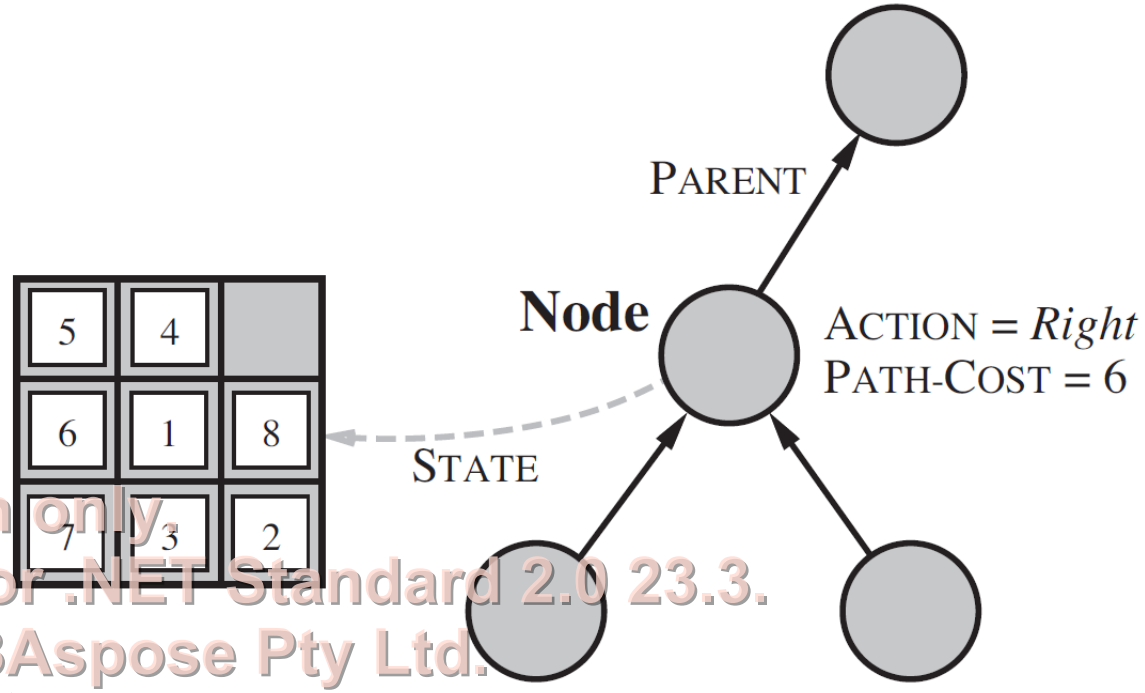


3) **Explicit description**
   - Explicit representation of all states in the state-space graph



Blackbox Description

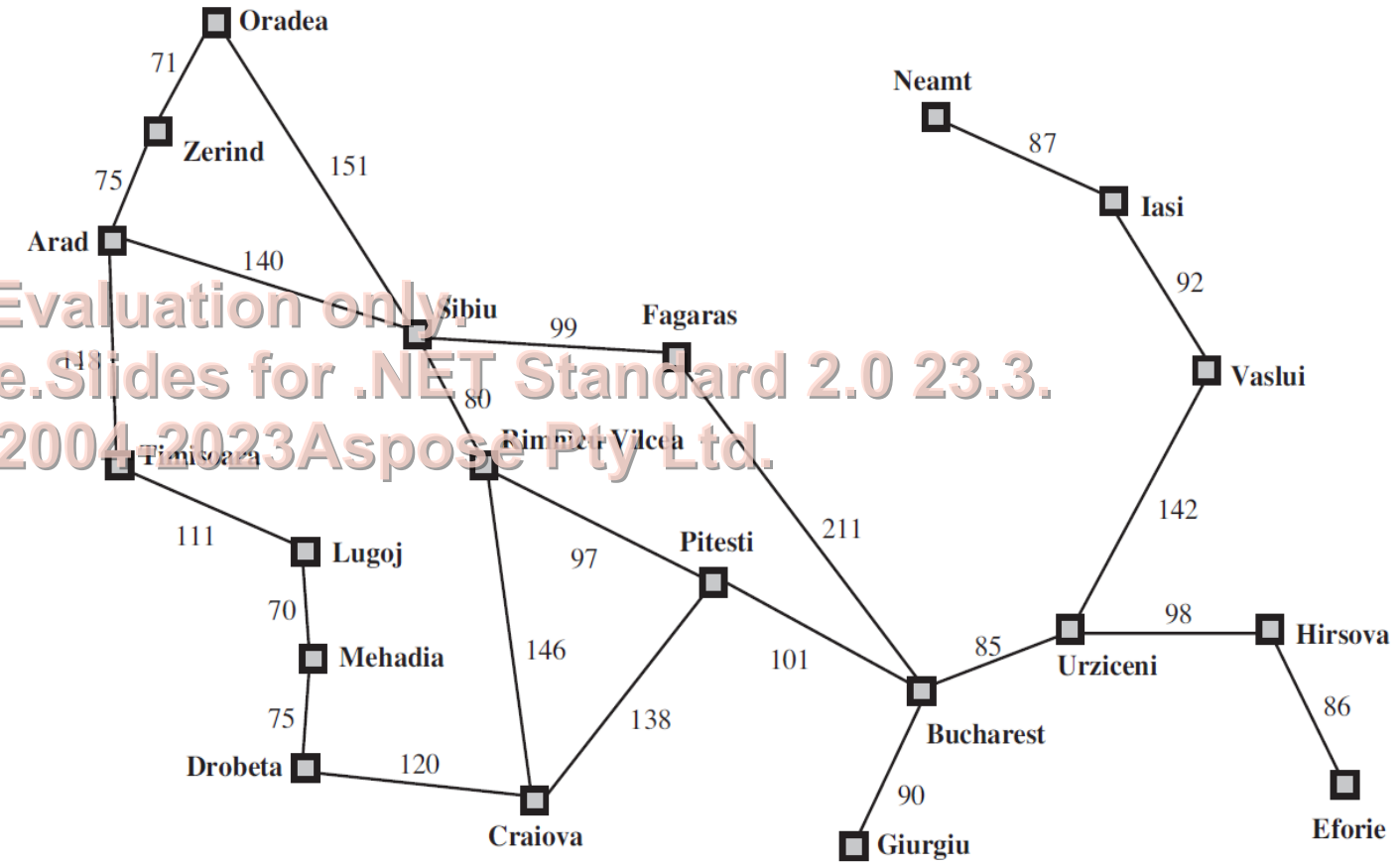
Application programming interface (API) to construct the state space



- *problem*.InitialState is given
- GoalTest(*Node*.STATE) is given for all *states*
- Cost(*Node*.*Action*) is given for all *actions*
- Actions(*Node*.STATE) is given for all *states*
- ChildState(*Node*.STATE,*Node*.ACTION) is given for all *states* and actions

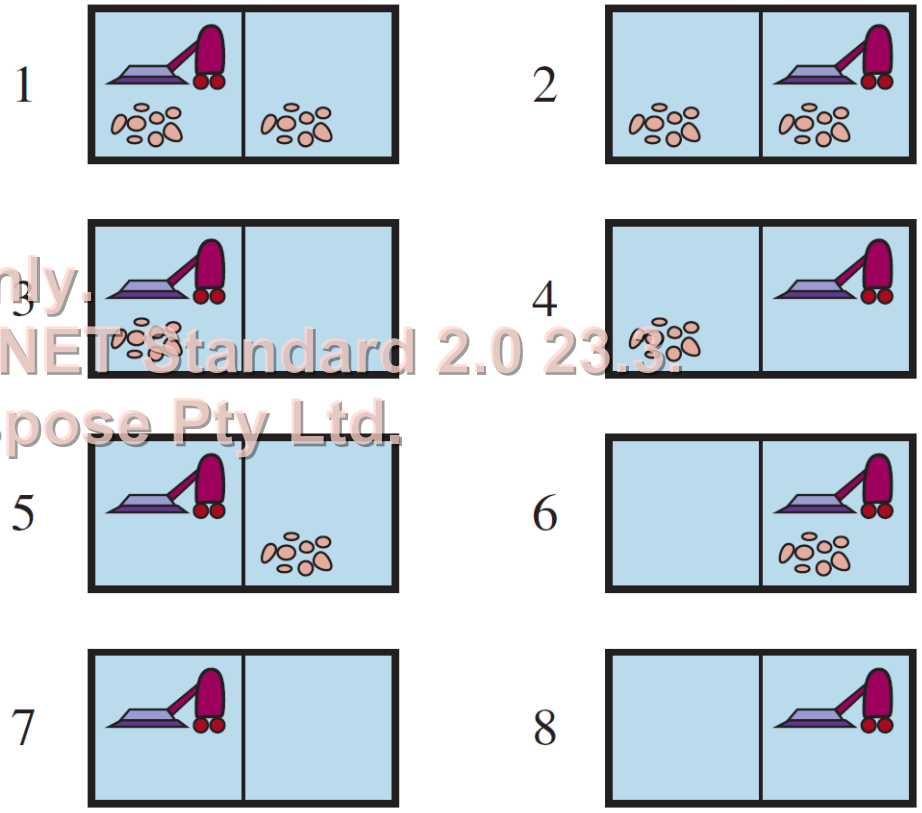
Whitebox Description: Romania Travel Example

- Given:
  - State space (in some compact encoding, not as a state space graph)
  - Goal test
  - Cost function
- The only difference to the blackbox description is that all states in the state space are directly accessible and not only via the *childstate* function

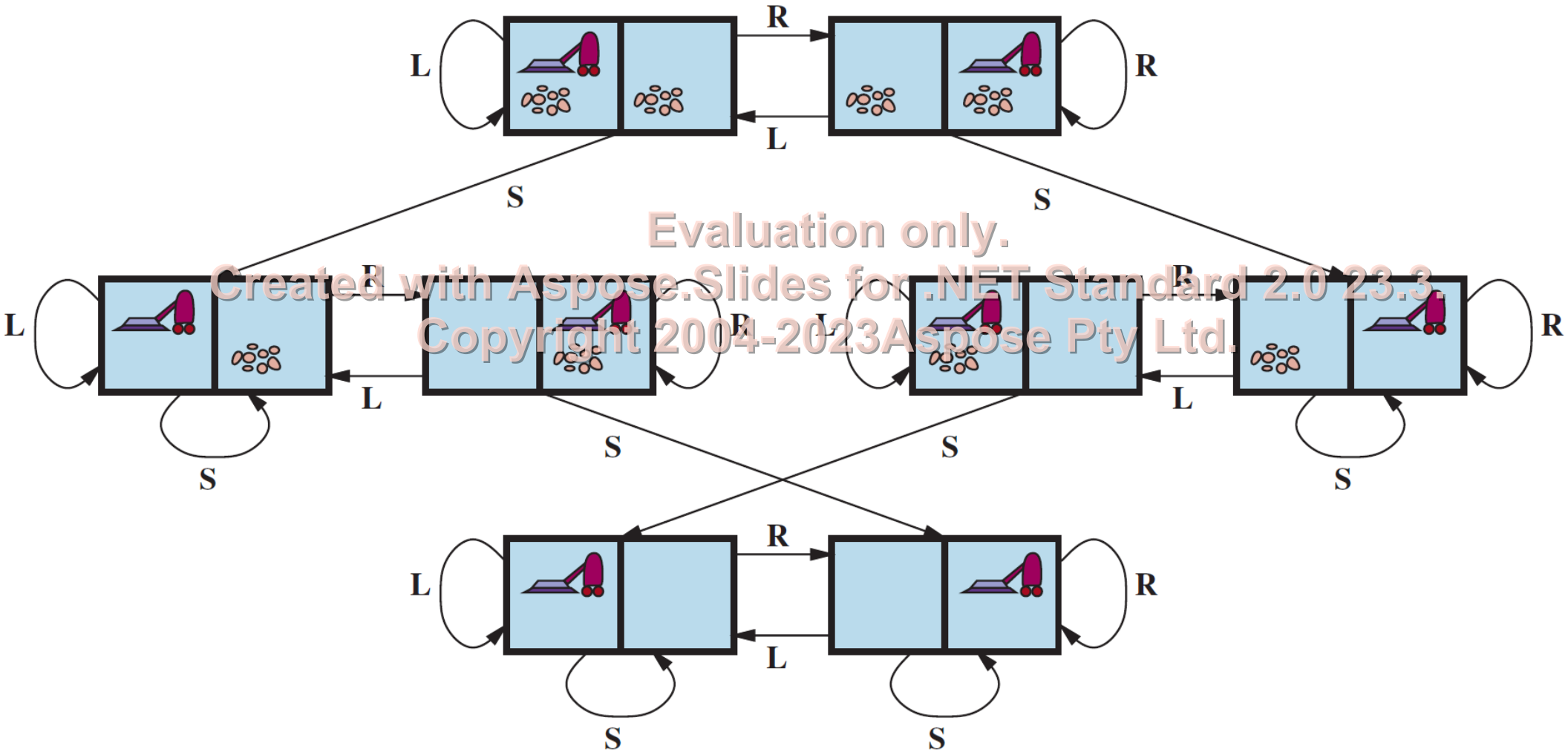


Whitebox Description: Vacuum Cleaner Agent

- State space:
  - States: 2 positions, dirt or no dirt
    → 8 states
  - Actions: Left (L),
    Right (R),
    Suck (S)
- Cost function: Unit costs
- Goal states: no dirt in the rooms



Explicit Description: State Space Graph of Vacuum Cleaning Agent



UNIVERSITÄT DES SAARLANDES

Systematic (Uninformed, Blind) Search Strategies

Tree Search vs. Graph Search

- Tree search
  - We assume that the search space has tree structure
  - When performing tree search, we do not need to remember visited states, because a state can only be visited via exactly one path from the initial state
  - Consequently, when performing tree search on a graph, we will not know whether we generate repeated states
- Graph search
  - Remember visited states (keep an explored set)
  - Use *duplicate elimination*: If a generated state is in the explored set, skip it, otherwise explore it

UNIVERSITÄT DES SAARLANDES

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 *node* from the frontier and remove it from the frontier
if the *node* contains a goal state **then return** the corresponding solution
expand the chosen *node*, adding the resulting nodes to the frontier

function GRAPH-SEARCH(*problem*) **returns** a solution or failure

initialize the frontier using the initial state of *problem*
initialize the explored set to be empty
loop do
if the frontier is empty **then return** failure
choose a *node* from the frontier 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 Algorithms

Guarantees:

- Soundness: Is an action path returned a solution to the problem?
- 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 node expansions)
- Space Complexity: How much memory does the search require? (Measured in nodes stored in memory)

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.

Overview on Discussed Search Strategies

§ **No** information on the length or cost of a path to the solution, only state space and goal states are given

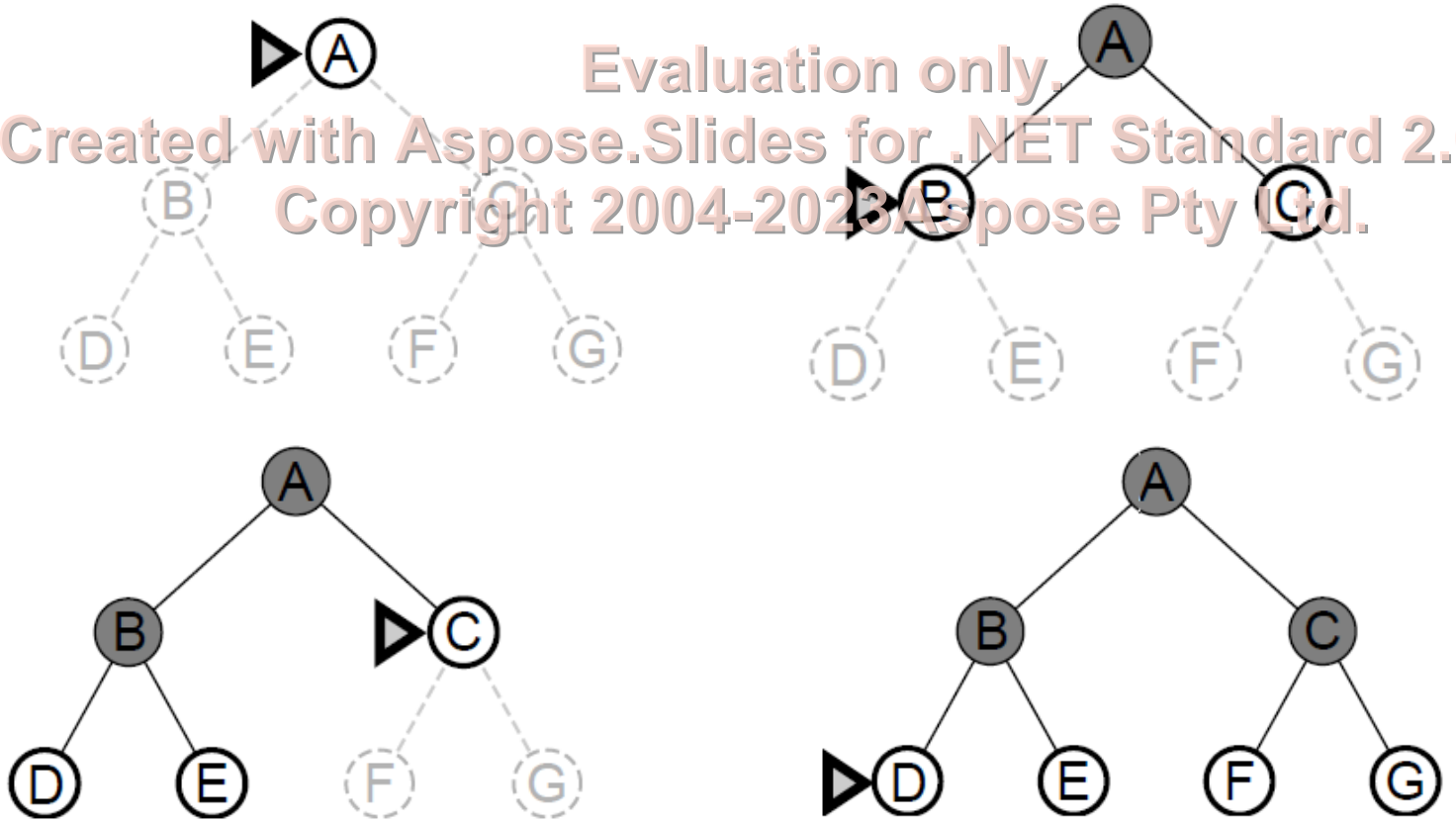
1) Breadth-first search
2) Depth-first search
3) Depth-limited search
4) Iterative deepening search

§ **Only current path costs** influence search

5) Uniform cost search

(1) Breadth-First Search (BFS)

§ Nodes are expanded in the order they are produced
  – Frontier is a **FIFO** queue

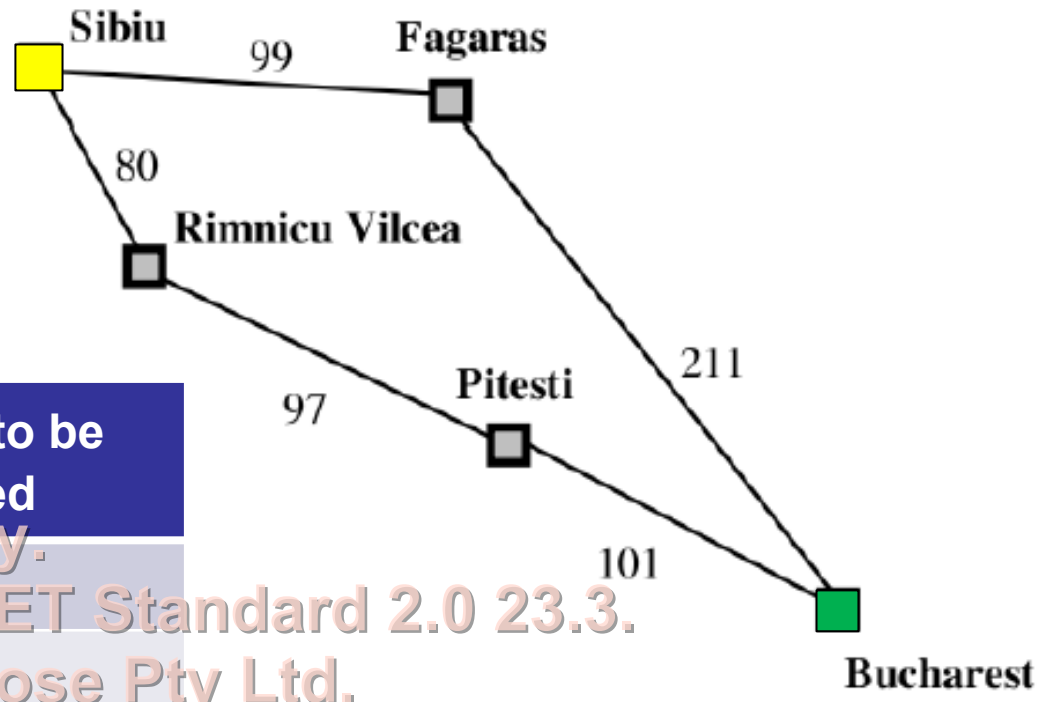


Breadth-First Search (BFS)

```
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 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): The search strategy already knows that this is a shortest path so it can just stop

Breadth-First Search (BFS) in Romania Travel Example



Search Step	Frontier	Explored set	Next node to be expanded
0	Sibiu		Sibiu
1	Rimnicu Vilcea, Fagaras	Sibiu	Rimnicu Vilcea
2	Fagaras, Pitesti	Sibiu, Rimnicu Vilcea	Fagaras

When expanding Fagaras, the goal test on Bucharest returns True, so the solution returned is:

Sibiu → Fagaras → Bucharest

Properties of Breadth-First Search (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 non-unit action costs
- The solution found is sub-optimal

Time and Space Complexity of Breadth-First Search (BFS)

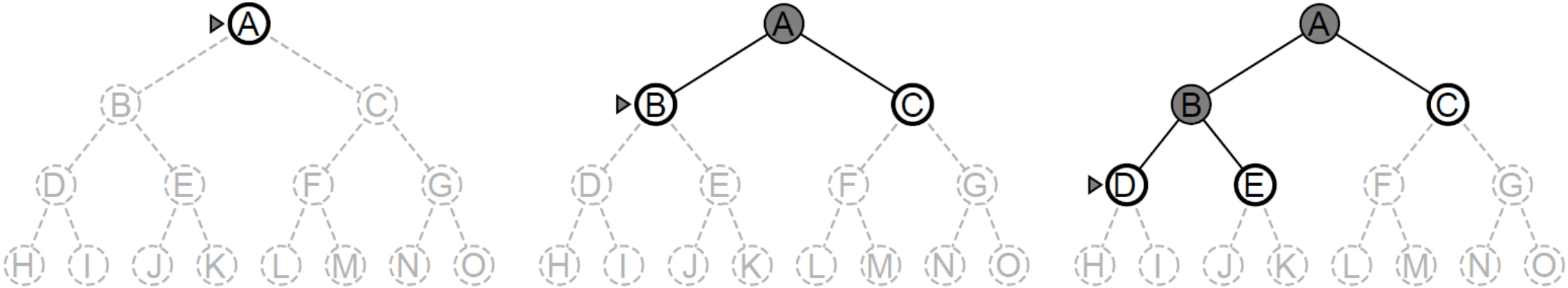
- Exponential time complexity
  - Let b be the maximum branching factor and d the depth of a solution path
  - Maximum number of node generations is

$$1 + b + b^2 + b^3 + \cdots + b^d = \sum_{n=0}^{d} b^n \in O(b^d)$$

- Exponential space complexity
  - Every node generated is kept in memory: $\sum_{n=0}^{d} b^n$
  - Space needed for the frontier is: $O(b^d)$
  - Space needed for the explored set: $O(b^{d-1})$

(2) Depth-First Search (DFS)

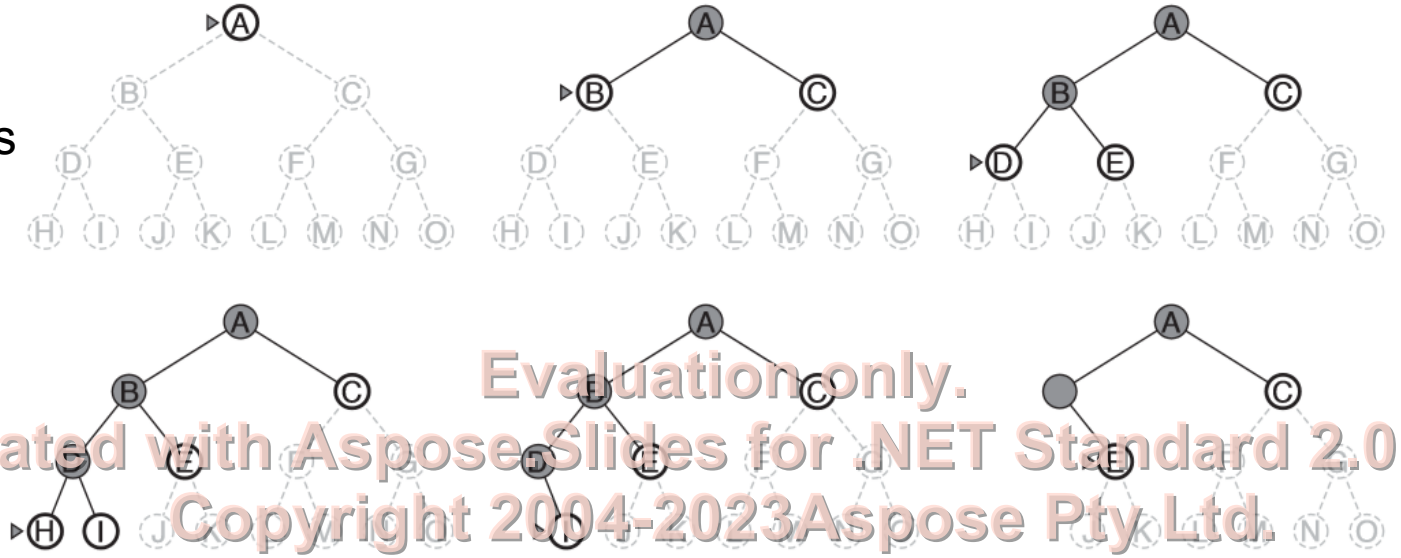
§ Always expand the deepest (most recent) node in the frontier
  – Frontier is a **LIFO** queue
  – If a node has no children, search backs up to the next deepest node that has unexplored children



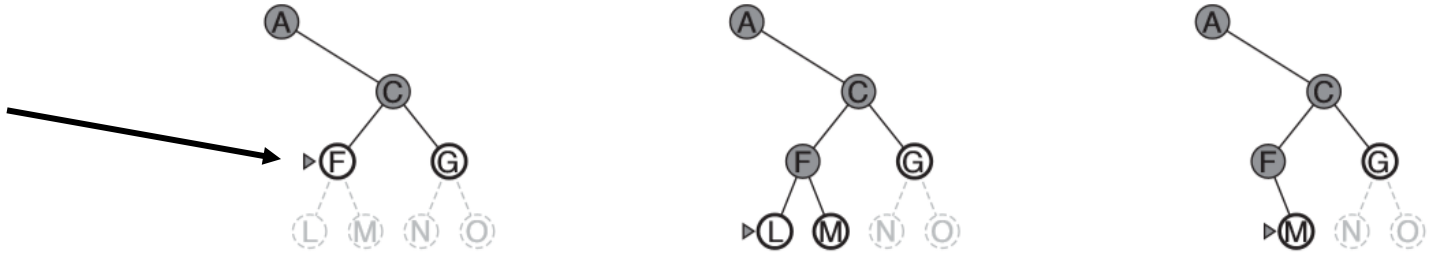
UNIVERSITÄT DES SAARLANDES

Example of Depth-First Search (DFS)

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



The pseudo code of the algorithm executes the goal test when expanding F and can thus terminate with a solution 2 steps earlier



Russell & Norvig, Fig. 3.16.

Depth-First Search (DFS) Algorithm

```
function DEPTH-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 LIFO queue with node as the only element
    explored ← an empty set
    loop do
        if EMPTY?(frontier) then return failure
        node ← POP(frontier) /*chooses the deepest node in frontier*/
        add node.State to explored
        for each action in 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)
```


Properties of Depth-First Search (DFS)

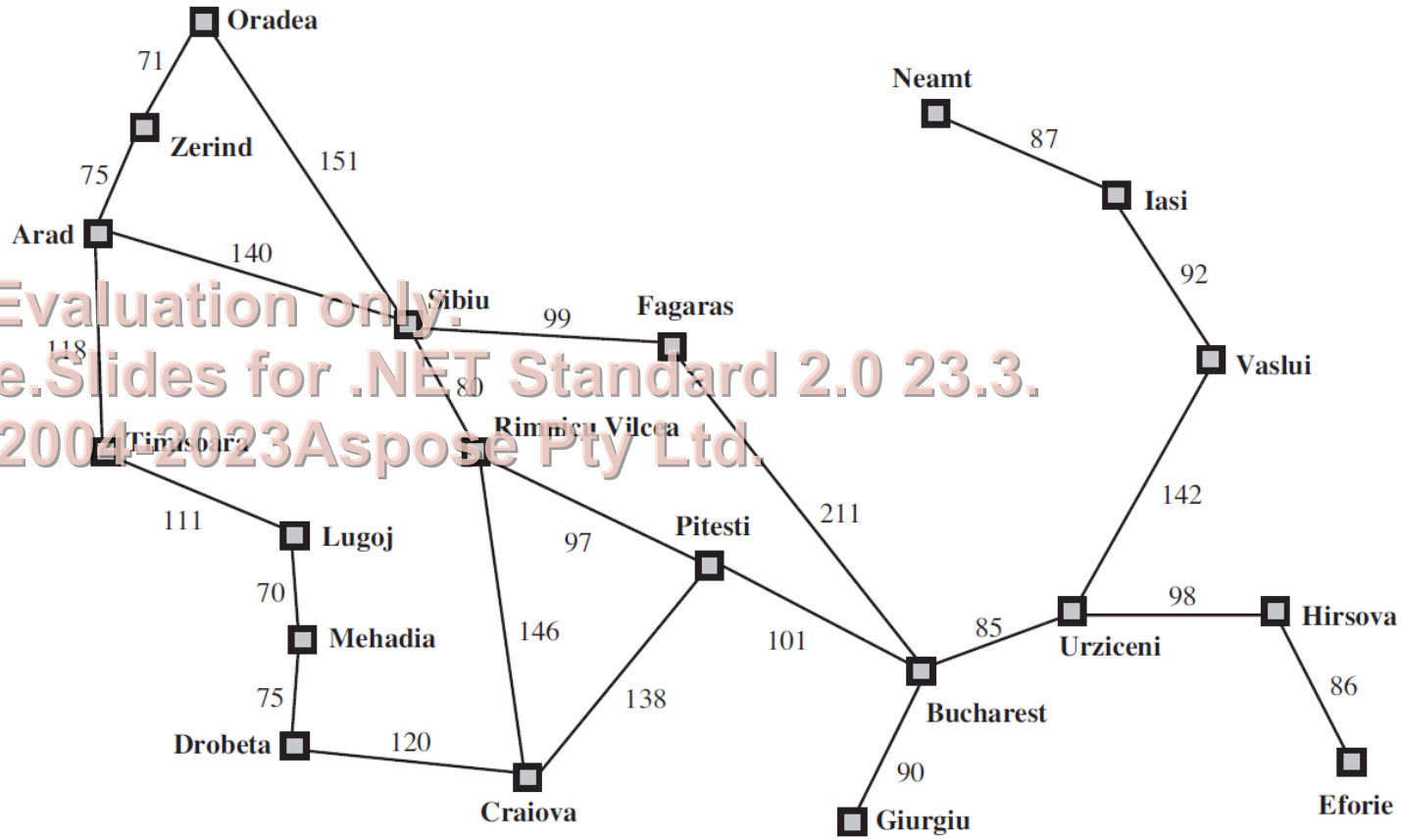
- In general, solution found is not optimal
- Incomplete!
- Completeness can be guaranteed only for graph search (we need to remember the visited nodes) or acyclic finite state spaces
  - In infinite state spaces, descends forever on infinite paths
  - Tree search may loop forever in repeated states

Time and Space Complexity of Depth-First Search (DFS)

- **Exponential time complexity**: $O(b^m)$
  - 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 tree search on graphs, i.e., do not remember already visited nodes
- **Linear space complexity**: $O(bm)$ or $O(m)$
  - We need $O(m)$ to store the nodes along the current path and $O(b)$ to store all neighbours (frontier at each level)
  - With clever indexing (backtracking search), we can eliminate $O(b)$ space and compute the neighbors dynamically in an efficient way using only $O(m)$

(3) Depth-Limited Search (DLS)

- Depth-first search with an imposed cutoff on the maximum depth of a path
- For example, for route planning with n cities: the maximum depth is n-1 to visit each city once
- In the Romania travel example, a depth of 9 is sufficient - every city can be reached in at most 9 steps



Depth-Limited Search (DLS) Algorithm

```
function DEPTH-LIMITED-SEARCH(problem, limit) returns a solution or failure/cutoff
    startNode ← a node with currentNode.State = problem.InitialState
    return RECURSIVE-DLS(startNode, 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 ≠ 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 Depth-Limited Search (DLS)

- Complete if the depth limit is not smaller than the depth of the shallowest goal state
- First solution found may not be optimal

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

- **Exponential time complexity**: $O(b^l)$
- **Linear space complexiy**: $O(bl)$ or $O(l)$ with backtracking search

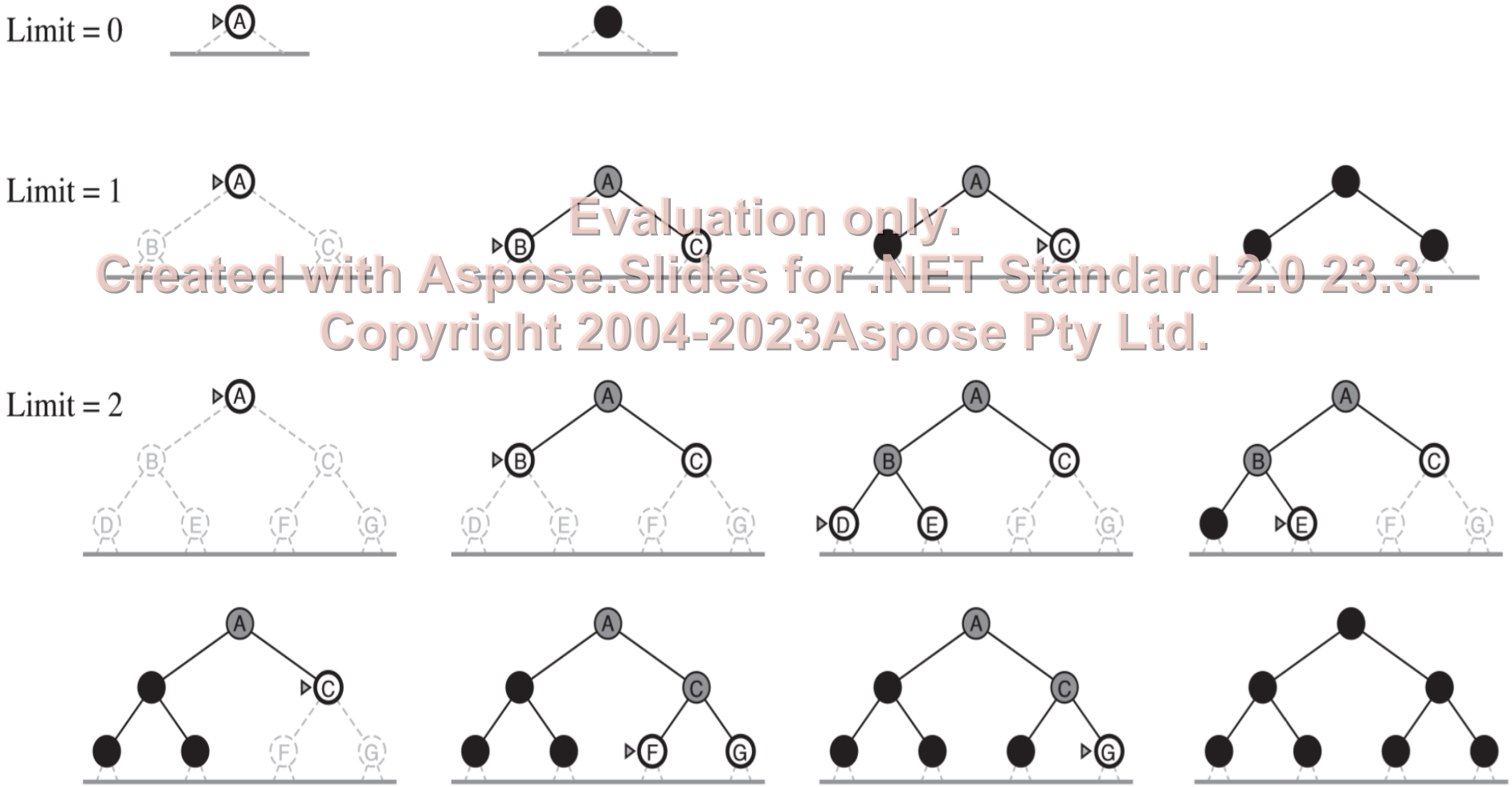
UNIVERSITÄT DES SAARLANDES

(4) Iterative Deepening Search (IDS)

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

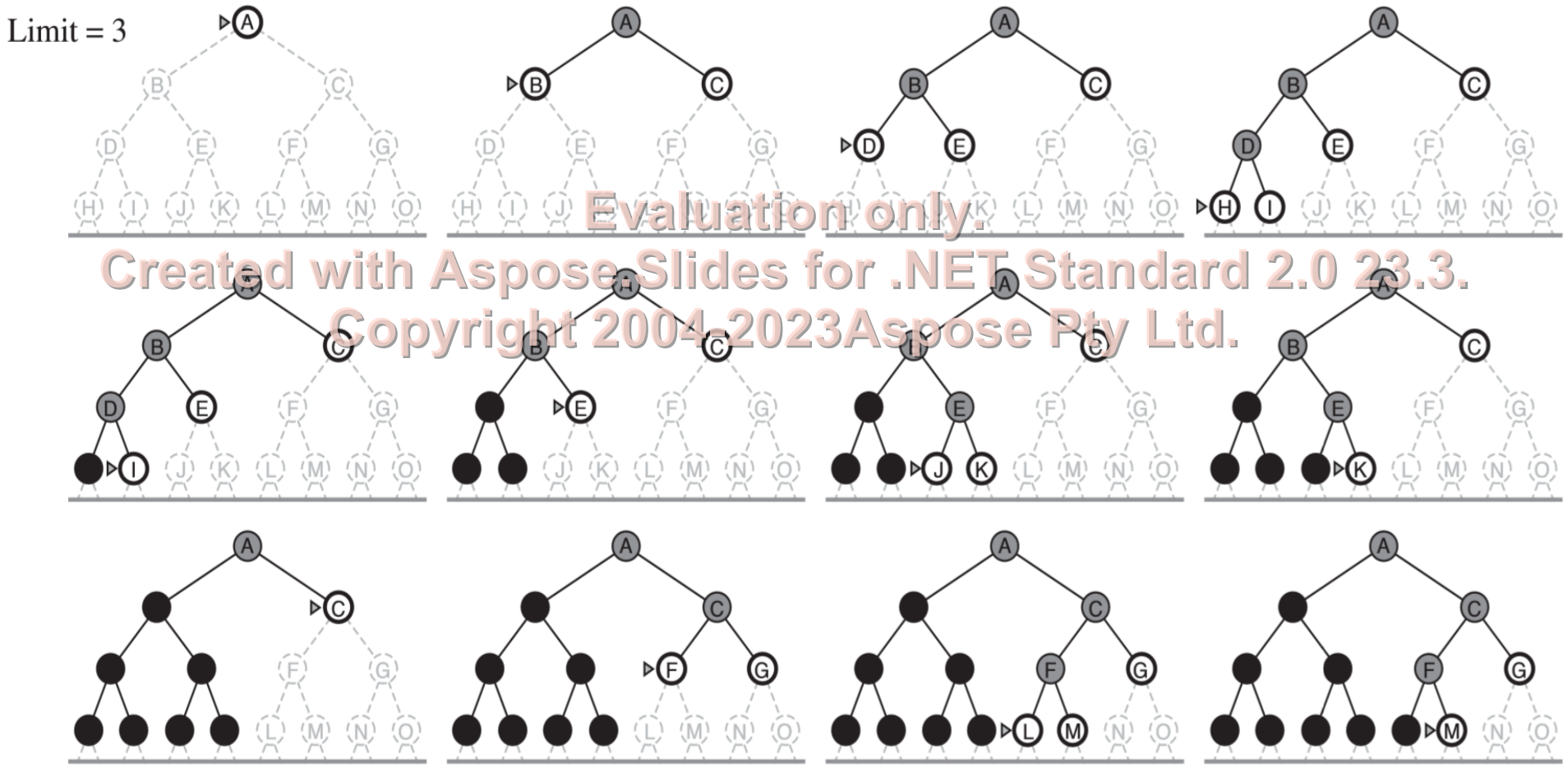
function ITERATIVE-DEEPENING-SEARCH(*problem*) **returns** a solution or failure
  for *depth* = 0 **to** ∞ **do**
    result $\leftarrow$ DEPTH-LIMITED-SEARCH(*problem*, *depth*)
    if *result* $\neq$ *cutoff* **then return** *result*

Illustration of Iterative Deepening Search (IDS)



UNIVERSITÄT DES SAARLANDES

Illustration Iterative Deepening Search (IDS) Continued



Properties and Complexity of Iterative Deepening Search (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: exponential time $O(b^l)$ with depth d set to limit l and linear 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. In this case:

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

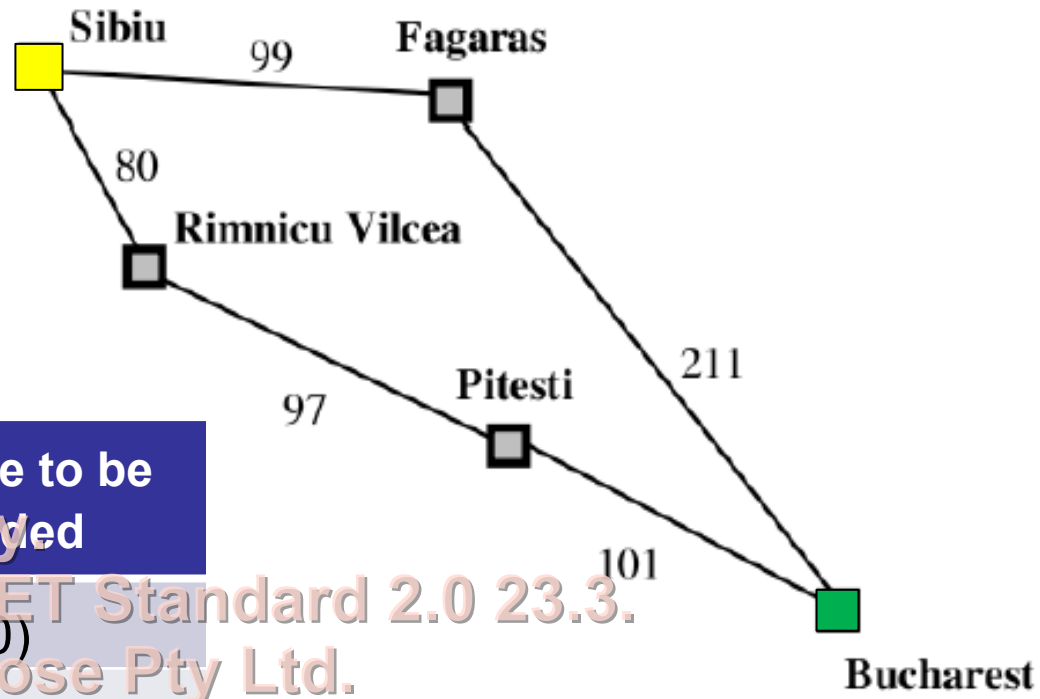
must hold for all n

Uniform-Cost Search (UCS) Algorithm

function UNIFORM-COST SEARCH(problem) **returns** a solution or failure
  node ← a node with *node*.State = problem.InitialState
  frontier ← a priority queue ordered by ascending PathCost, with *node* as the only element
  explored ← empty set of states
  loop
    if Empty?(*frontier*) **then return** failure
    node ← Pop(*frontier*)
    if GoalTest(*node*.State) **then return** Solution(*node*)
    explored ← explored ∪ *node*.State
    for *action* **in** Actions(*node*.State) **do**
      child ← ChildNode(problem, *node*, *action*)
      if *child*.State ∉ [explored ∪ States(*frontier*)] **then** Insert(*child*, *child*.PathCost, *frontier*)
      else if there exists *otherNode* ∈ *frontier* s.t. *otherNode*.State = *child*.State and
          child.PathCost < *otherNode*.PathCost **then** replace *otherNode* in *frontier* with *child*

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

Example of Uniform-Cost Search (UCS)



Search Step	Frontier	Explored set	Next node to be expanded
0	S (0)		S (0)
1	RV (80), F (99)	S	RV (80)
2	F (99), P (177)	S, RV	F (99)
3	P (177), B (310)	S, RV, F	P (177)
4	B (278)	S, RV, F, P	**B (278)**

The solution is returned with cost 278:

Sibiu → Rimnicu Vilcea → Pitesti → Bucharest

Properties and Complexity of Uniform-Cost Search (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 cost actions
- Complete, if all action costs > 0

- Exponential time and space complexity depend on length of optimal solution: $O\left(b^{1+\lfloor C^*/\varepsilon \rfloor}\right)$
  - C* path cost of optimal solution, action costs $\geq \varepsilon$
  - If all action costs are equal and optimal solution has length d, 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)

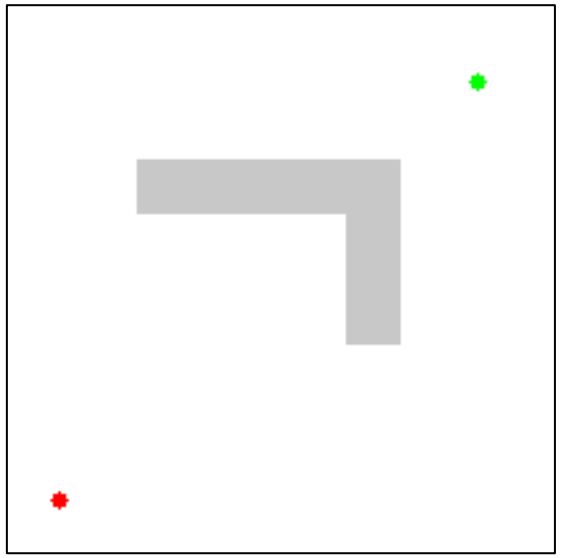
The only differences are:

(a) We generate only a part of the 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:

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



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

Summary of Algorithm Properties and Complexity

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

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 action costs not less than ε
- c if action costs are all identical
- d Yes for finite search spaces
- e $O(b)$ can be eliminated by backtracking search

Summary

- 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 due to its exponential space complexity
  - 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
- IDS is the preferred uninformed search method when there is a large search space and the depth (length) of the solution is not known
- UCS finds optimal solutions when all actions have non-negative action costs (and thus is optimal for problems with non-unit action costs)

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.