ĐẠI HỌC BÁCH KHOA HÀ NỘI
VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

IT3160E
Introduction to Artificial Intelligence

Chapter 3 – Problem solving
Part 1: problem-solving by searching

Lecturer:

Muriel VISANI

Acknowledgements:

Le Thanh Huong

Tran Duc Khanh

Department of Information Systems

School of Information and Communication Technology - HUST

Content of the course

- Chapter 1: Introduction
- Chapter 2: Intelligent agents
- Chapter 3: Problem Solving
  - Search algorithms, adversarial search
  - Constraint Satisfaction Problems
- Chapter 4: Knowledge and Inference
  - Knowledge representation
  - Propositional and first-order logic
- Chapter 5: Uncertain knowledge and reasoning
- Chapter 6: Advanced topics
  - Machine learning
  - Computer Vision

Outline

- Chapter3 - part 1: problem solving by searching
  - Introduction
  - Problem-solving agents
  - Problem formulation
  - Basic search algorithms
    - General tree search
    - Breadth-first search
    - Uniform-cost search
    - Depth-first search
    - Depth-limited search
    - Iterative deepening depth-first search
    - Summary
  - Homework

Goal of this Lecture

Goal	Description of the goal or output requirement	Output division/ Level (I/T/U)
M1	Understand basic concepts and techniques of AI	1.2

Chapter 3 - part1

Introduction

Introduction

- Recall: environment types
  - Fully observable (vs. partially observable): An agent's sensors give it access to the complete state of the environment at each point in time.
    - The environment is effectively and fully observable if the sensors detect all aspects that are relevant to the choice of action
  - Deterministic (vs. stochastic): The next state of the environment is completely determined by:
    - the current state
    - the action executed by the agent
    - If the next state of the environment also depends on other factors/agents that are not totally predictable, then the environment is stochastic

Introduction

- Many different types of problems exist
  - Deterministic, fully observable, static, discrete → single-state problem
    - Fully observable: the agent's sensors give it enough information to tell exactly which state it is in
    - Deterministic: it knows exactly what each of its actions does
    - Agent knows exactly in which state it will be after each sequence
    - Example: vacuum-cleaner world

    → This is called a single-state problem; the solution is an action sequence.
  - Deterministic, partially observable / non-observable, static, discrete → multi-state problem
    - Agent knows what its actions does, but it might not know in which state it is now
    - In this case, the agent must reason about sets of states that it might get to, rather than single states.

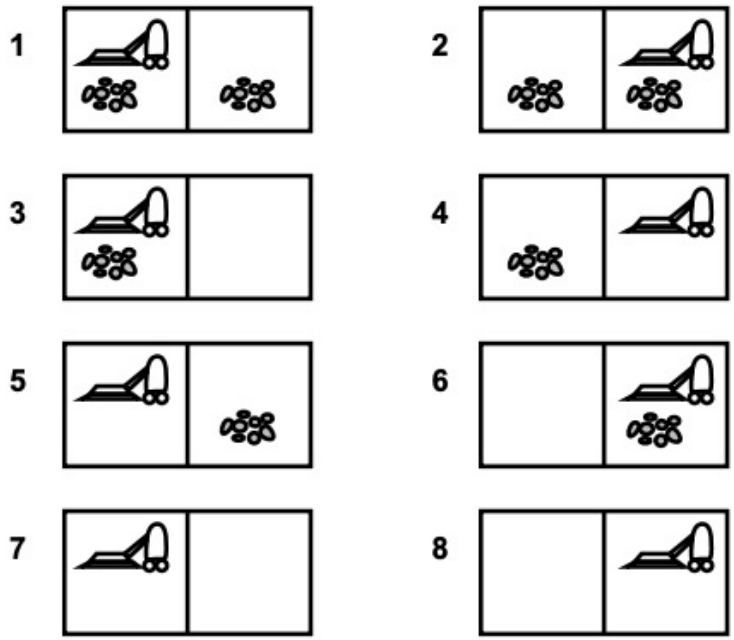
    →This is called a multi-state problem or a sensor-less (conformant) problem if non-observable

    →The solution is an action sequence

Introduction

- Examples: vacuum-cleaner world
  - Possible actions:
    - Left, Right, Suck, NoOp
    - The Agent cannot go Right if it is already in the Right square
      - Trying to go Right then would result in NoOp (the same goes for Left)
  - Goal: having both squares clean
- Example of single-state problem:
    - At each step, the agent knows the environment's state
      - Where it is
      - Where the dirt is
    - The agent knows what are the effects of each of its actions
  - Let us consider that the initial state is 5
  - **Question**: what is the shortest sequence of actions to reach the goal?
    - **Answer**:

Possible states

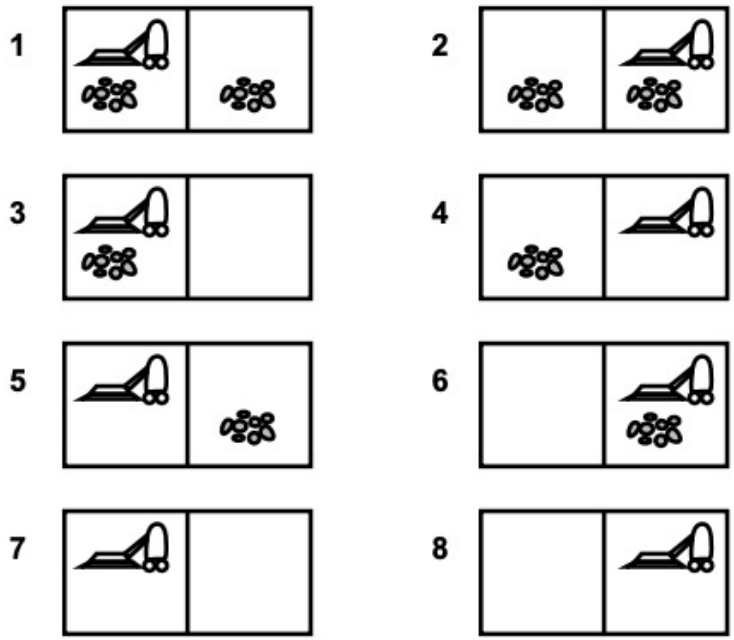


Introduction

- Examples: vacuum-cleaner world
  - Possible actions:
    - Left, Right, Suck, NoOp
    - The Agent cannot go Right if it is already in the Right square
      - Trying to go Right then would result in NoOp (the same goes for Left)
  - Goal: having both squares clean
- Example of multi-state problem:
    - The agent misses some sensor / info
    - The agent know what are the effects of each of its actions
  - Let's say that the agent does not know its initial location

  → Initial state is in {1,2,3,4,5,6,7,8}
  - **Question**: give one among the shortest sequences of actions to reach the goal
    - **Answer**:

Possible states



Introduction

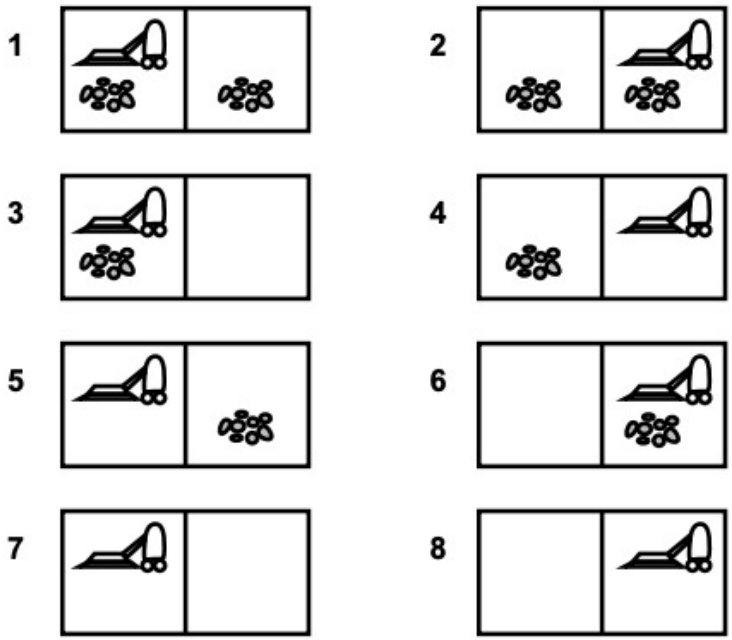
- Many different types of problems exist
  - Nondeterministic and/or partially observable → contingency problem
    - There is no fixed action sequence that guarantees a solution to this problem in general
    - But, the agent can solve the problem if it can sense during execution
    - In that case, percepts provide new information about the current state
    - often interleave with the agent's actions → search, execution
    - In this case, the agent must calculate a whole tree of actions rather than a single action sequence
      - Its plans now have conditionals (ifs), based on the results of sensing
    - N.B. many problems in the real world are contingency problems

Introduction

- Examples: vacuum-cleaner world
  - Possible actions:
    - Left, Right, Suck, NoOp
    - The Agent cannot go Right if it is already in the Right square
      - Trying to go Right then would result in NoOp (the same goes for Left)
  - Goal: having both squares clean
- Example of contingency problem:
  - Let's say that:
    - The agent has a position sensor and a **local** dirt sensor
    - When the agent performs "Suck", then the result might be that it releases dust instead of cleaning it -> non-deterministic
  - Let's say that the agent is in A, with dust

  → Initial state is in {1,3}
  - **Question:** is there any fixed action sequence that guarantees a solution to this problem?
  - **Answer**:

Possible states



Introduction

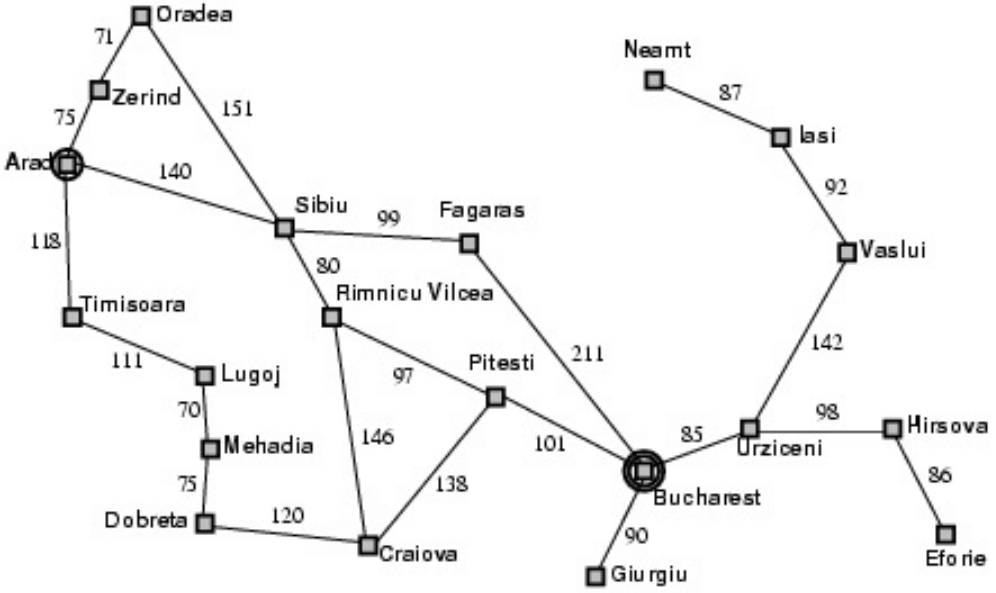
- Many different types of problems exist
  - Unknown state space, stochastic → exploration problem
    - More difficult; to be studied in later chapters.

Introduction

- In this lecture, we focus on the simplest kind of task environment (single-state problem)
  - The environment is known, observable, discrete and deterministic

  →the solution to a problem is always a **fixed sequence** of actions
- The more general cases will be treated later on

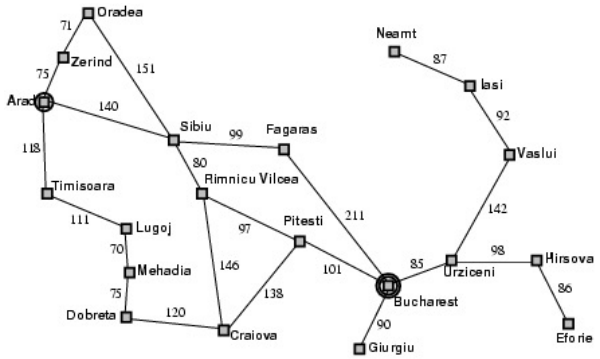
Example: Route Planning



- Goal: Get from Arad to Bucharest; Performance: as little km as possible
- Environment: the map, with cities, roads, and km between cities
- Sensor: a camera with OCR to “see” / “understand” the road signs + all other sensors necessary for the car to drive autonomously
- Actions: Travel a road between adjacent cities
  - For simplicity, we assume that there is no one else on the road and that the agent is able to drive safely from one city to another
  - The agent has access to the map

Exercise: Route Planning

- Exercise: explain why we can assume that the environment is
  - **Known**
  - **Observable**
  - **Discrete**
    - Reminder: an environment is discrete if, at any given state there are only a finite number of actions to choose from
  - **Deterministic**



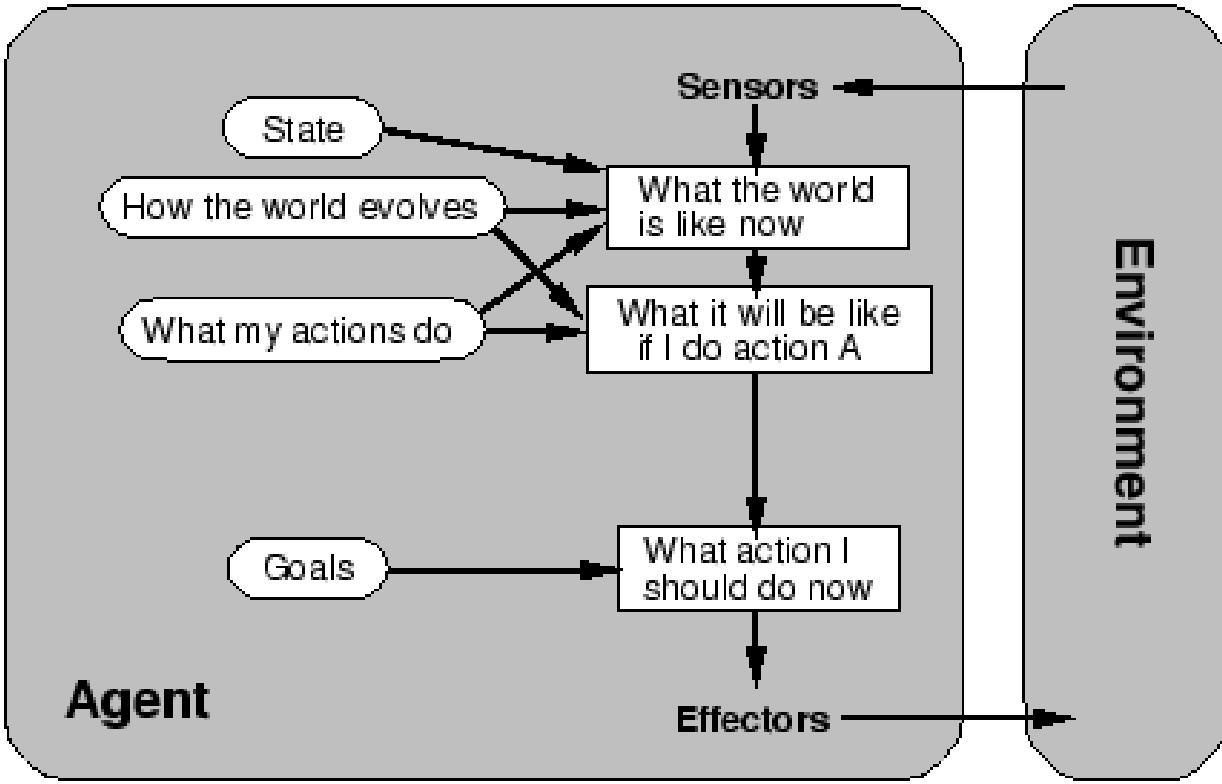
Chapter 3 - part1

Problem-solving agents

Problem-solving agents

- Problem-solving agents are a special kind of goal agents

  - Agents that take actions in the pursuit of a goal (or multiple goals)
  - Goals introduce the need to reason about the future or other hypothetical states
  - → need for searching for the best next action / sequence of actions (given the goal)



Problem-solving agents

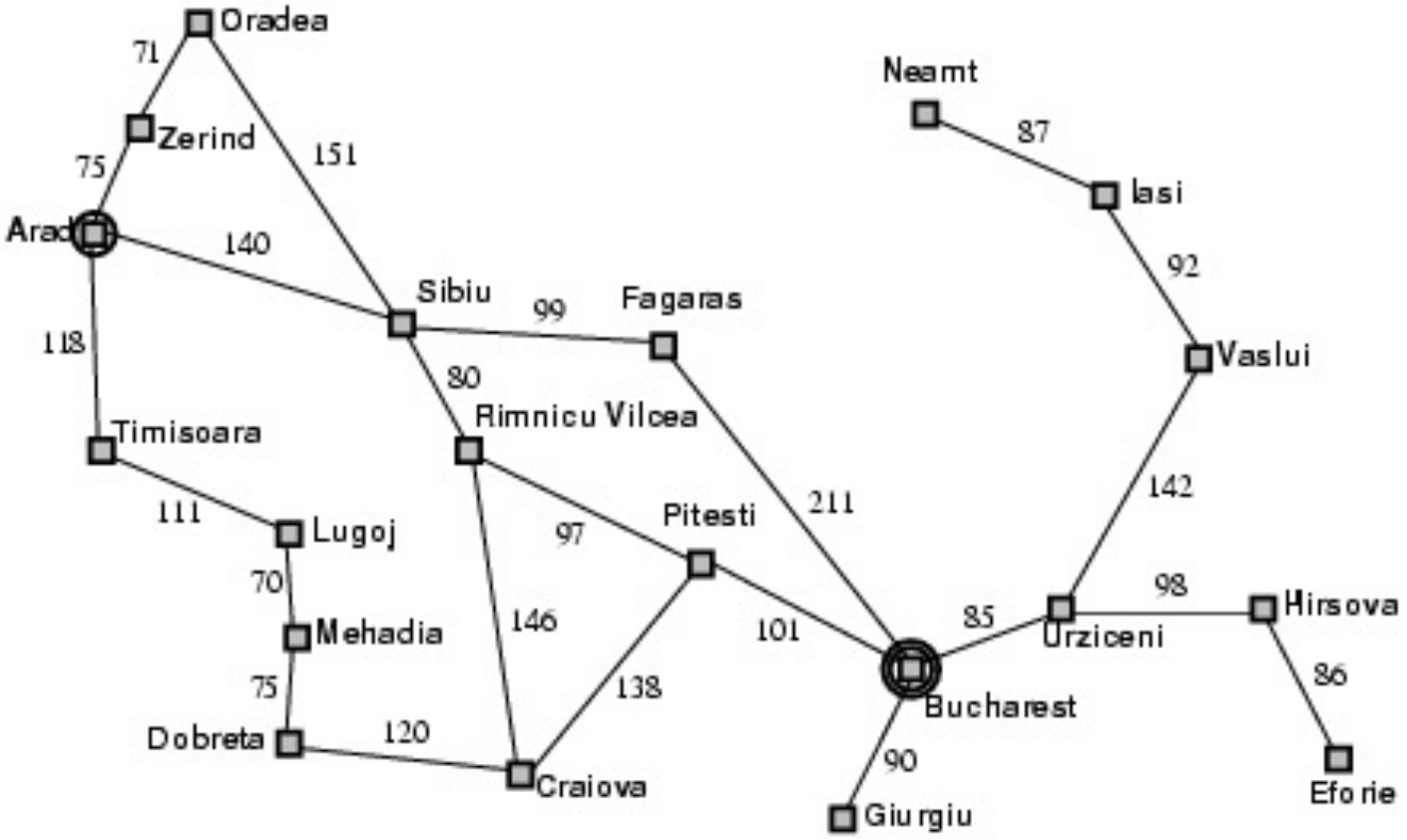
- Problem-solving agents are a special kind of goal agents
- A goal is the set of world states where the final objective is satisfied
- Goal formulation is the 1st step in problem solving
  - based on the current situation and the agent's performance measure
  - the agent's task is to search for a sequence of actions to reach the goal
  - Before choosing its next action, the agent needs to examine to which states each of the possible action will lead it to
- Problem formulation is deciding what actions and states to consider, given a goal

Problem-solving agents

```
function SIMPLE-PROBLEM-SOLVING-AGENT(percept) returns an action
    static: 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 do
        goal ← FORMULATE-GOAL(state)
        problem ← FORMULATE-PROBLEM(state, goal)
        seq ← SEARCH(problem)
    action ← FIRST(seq)
    seq ← REST(seq)
    return action
```

Example: Route Planning



- Goal: Get from Arad to Bucharest; Performance: as little km as possible
- Environment: the map, with cities, roads, and km between cities
- Actions: Travel a road between adjacent cities

Formulation of actions: notion of **abstraction**

Chapter 3 - part1

Problem formulation

Search Problem Formulation

- A problem is defined by four items:

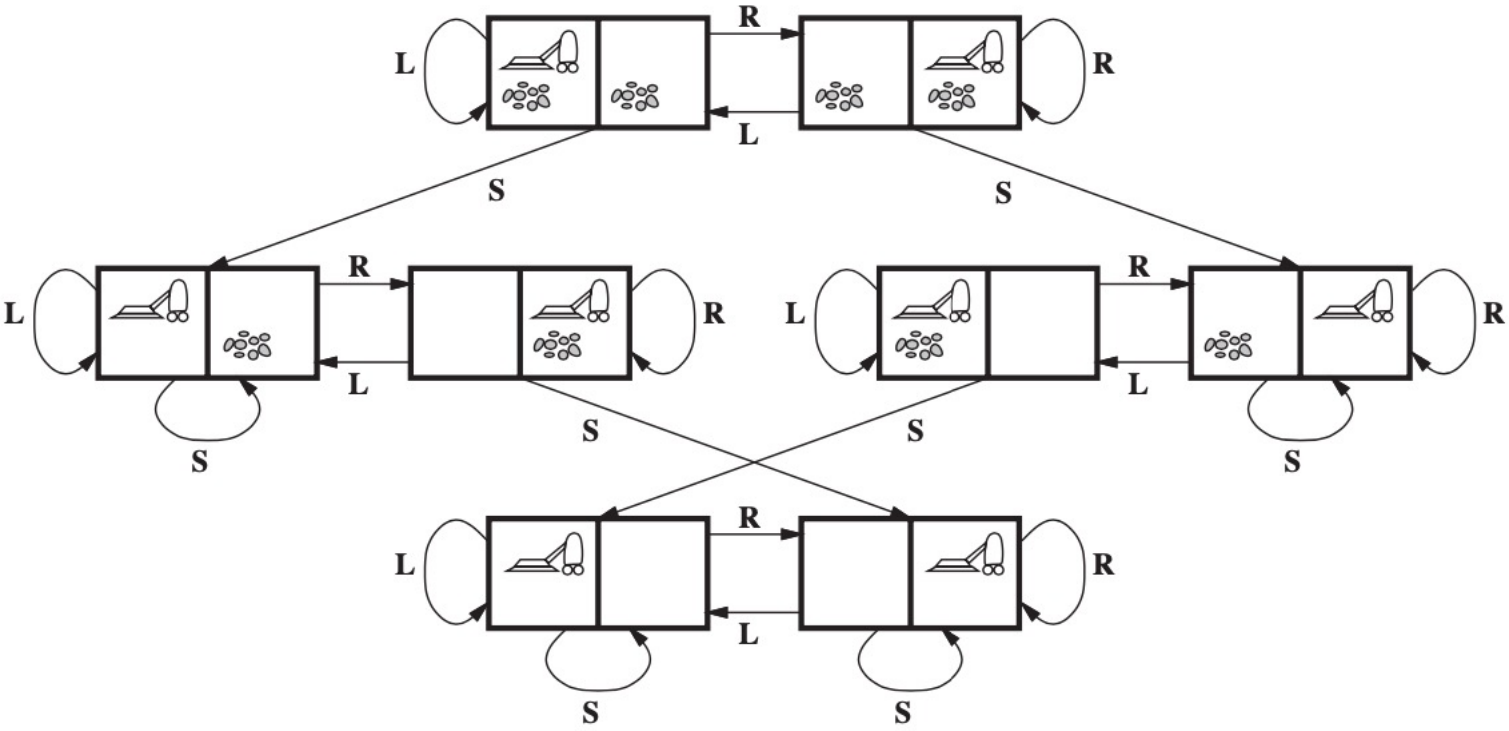
state space of the problem (graph)

1. initial state: e.g., *Arad*
2. Actions/transition model or successor function *S(x)* = set of action-state pairs
   - e.g., *S(Arad)* = {<*Arad → Sibiu, Sibiu*>, … }
3. goal test, can be
   - explicit, e.g., *x* = Bucharest
   - implicit, e.g., *Checkmate(x)*
4. path cost (additive)
   - *e.g.* sum of distances, number of actions executed, etc.
   - *c(s,a,s')* is the step cost, assumed to be ≥ 0
   - Question: how much is *c(Arad, Arad → Sibiu, Sibiu)* ?
     - Answer:

- A solution is a sequence of actions leading from the initial state to a goal state

Example: vacuum-cleaner

- State-space for the vacuum cleaner problem



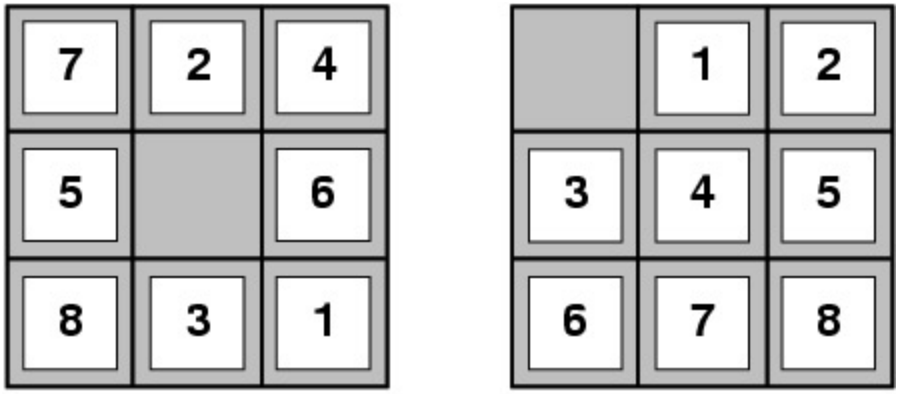
Not every action in the state space makes sense!!!

The idea is to search, from the state space, the sequence of actions that makes the most sense…

Example: vacuum-cleaner

- Example problem formulation for the original vacuum cleaner problem
  - **States**: agent location + dirt locations. 2 locations, 2 status (dirty / clean) -> 2 x 2^2 states (a 2-state environment with n locations has n x 2^n states)
  - **Initial state**: Any state can be designated as the initial state
  - **Actions**: *Left*, *Right*, *Suck*
  - **Transition model**: Expected effects (except moving *Left* in the leftmost square, moving *Right* in the rightmost square, and *Suck*ing in a clean square)
  - **Goal test**: This checks whether all the squares are clean.
  - **Path cost**: Each step costs 1, so path cost = number of steps in path

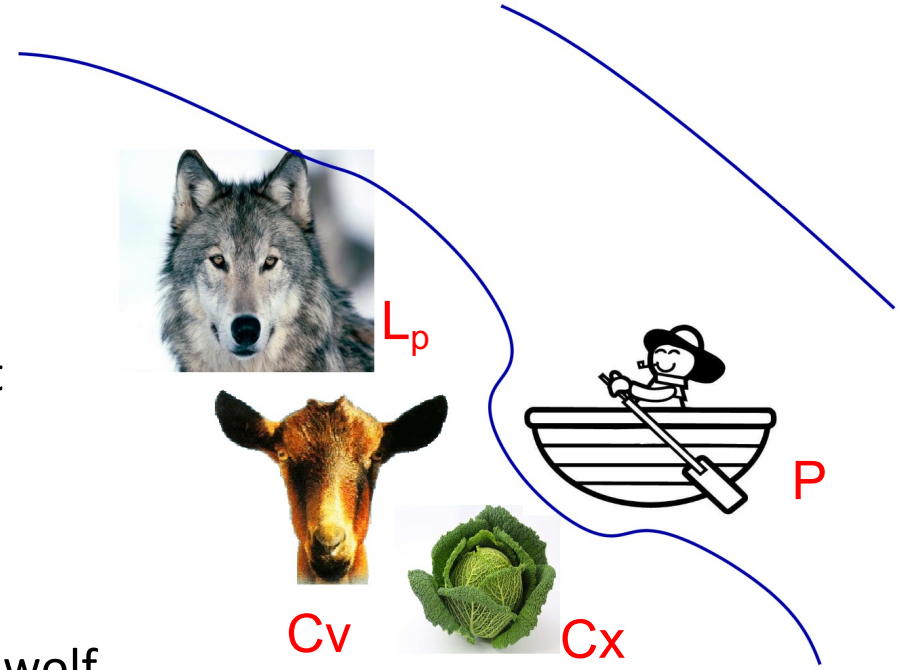
Exercise: The 8-puzzle



- What are the states?
- What are the actions?
- What is the goal test?
- What is the path cost?

Exercise: shepherd, goat, wolf, cabbage

- Problem
  - The shepherd must bring a wolf, a goat and a cabbage on the other side of the river
  - Initial state: All 4 are on the left side of the river
  - The boat can only transport:
    - 1 object or animal + the shepherd
  - The shepherd cannot leave the wolf and goat alone on any river bank
  - The shepherd cannot leave the goat and cabbage alone on any river bank
  - States: list of objects on each bank; action: crossing the river



Exercise: shepherd, goat, wolf, cabbage

- Exercise: give the state space of this problem
  - Tips: take as example the graph on slide 23
    - You just need to show the « contents » of the two sides of the river after each action (river crossing)
    - Initial state: *Cx Cv Lp P*

The state space might contain conflicting states

- **Answer: state space**

Exercise: shepherd, goat, wolf, cabbage

- Now, from the state space, remove conflicting states
  - **Answer:**

Exercise: shepherd, goat, wolf, cabbage

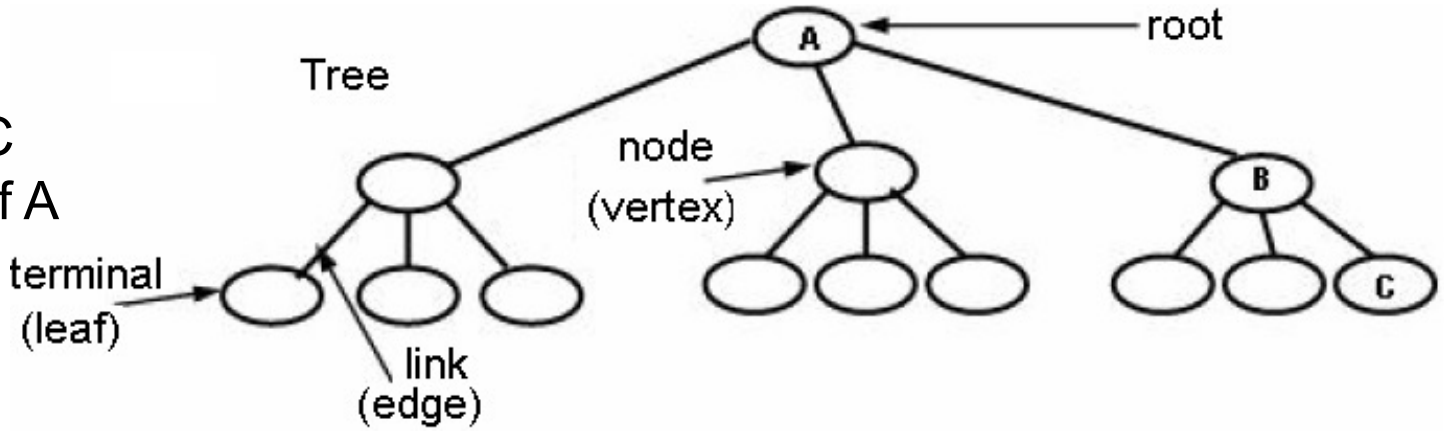
- Simplified state space

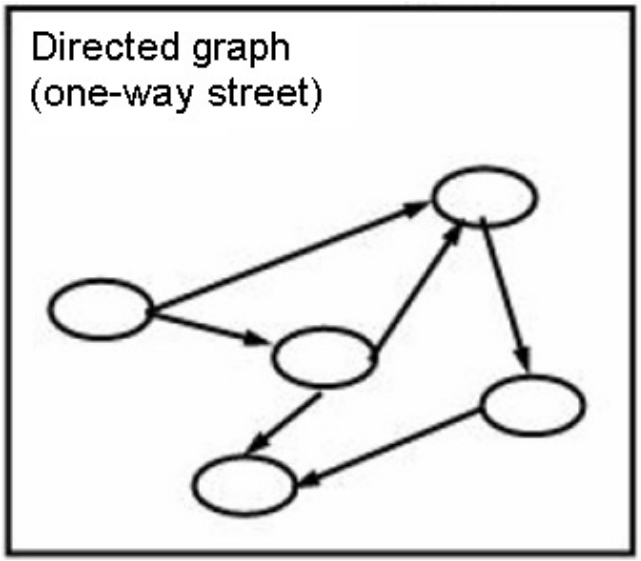
Chapter 3 - part1

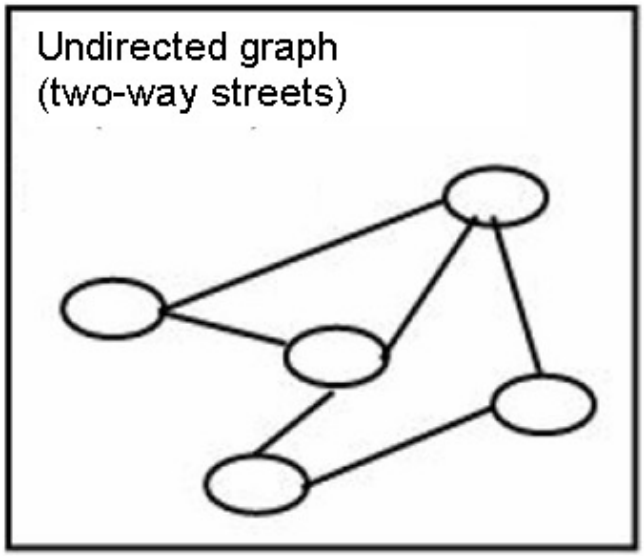
Basic search algorithms

Definitions: tree and graph

B is parent of C
C is child of B
A is ancestor of C
C is decendant of A

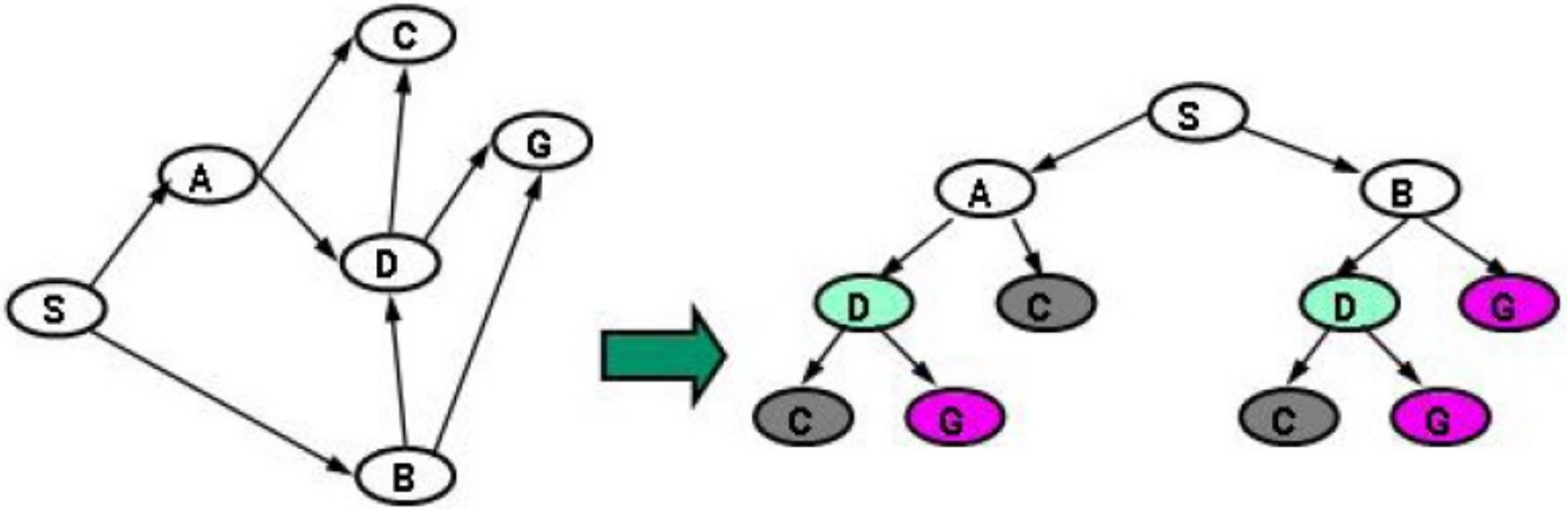




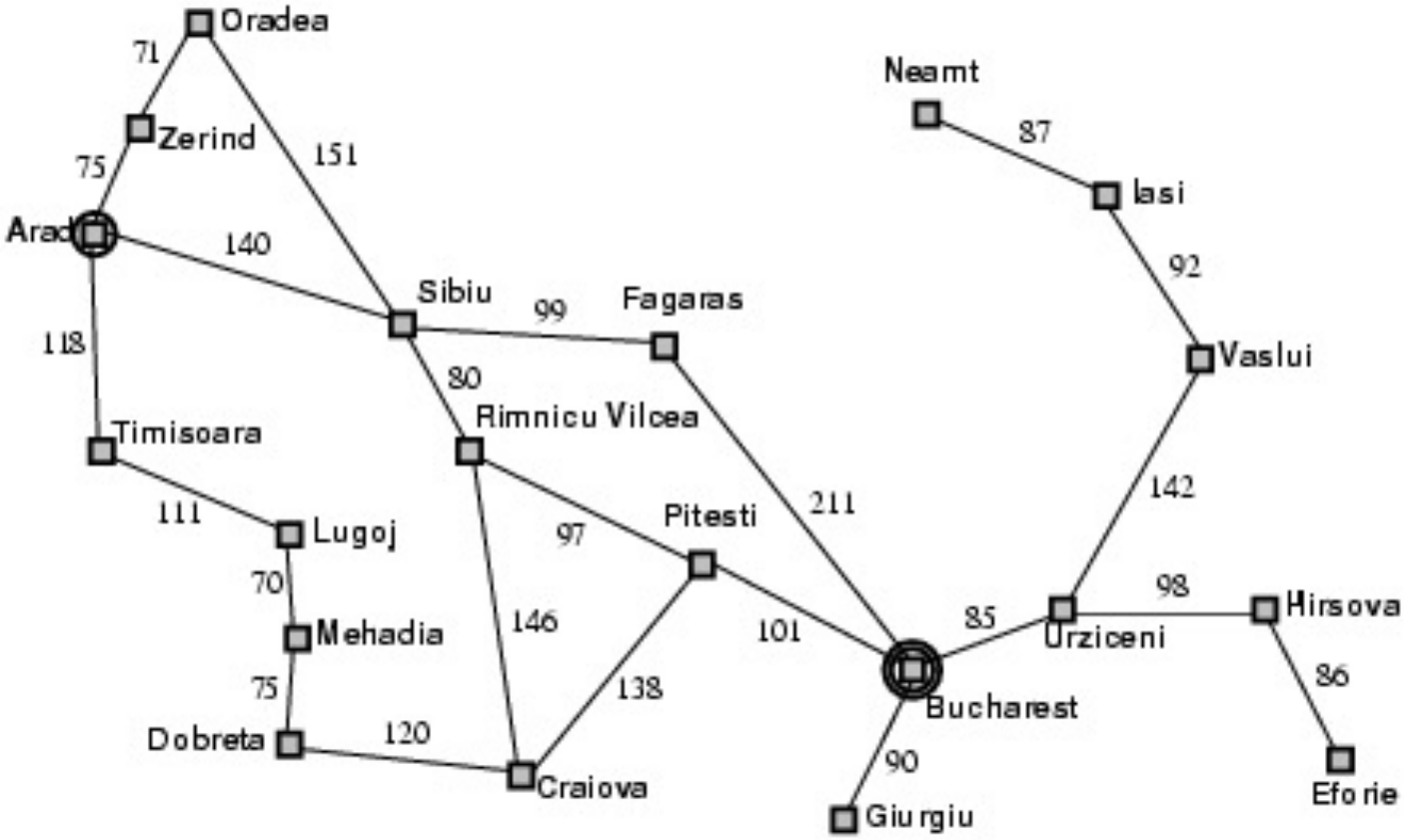


Convert from search graph to search tree

- One can easily turn graph search problems into tree search problems
  - Possibly with repeated states in the tree

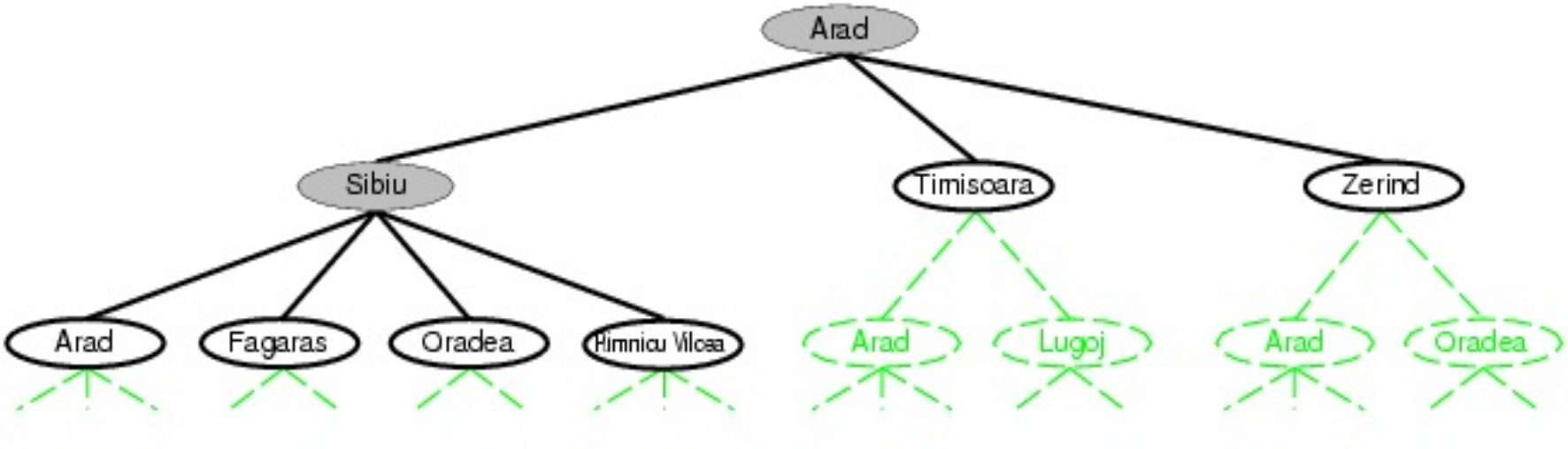


Example: Route Planning



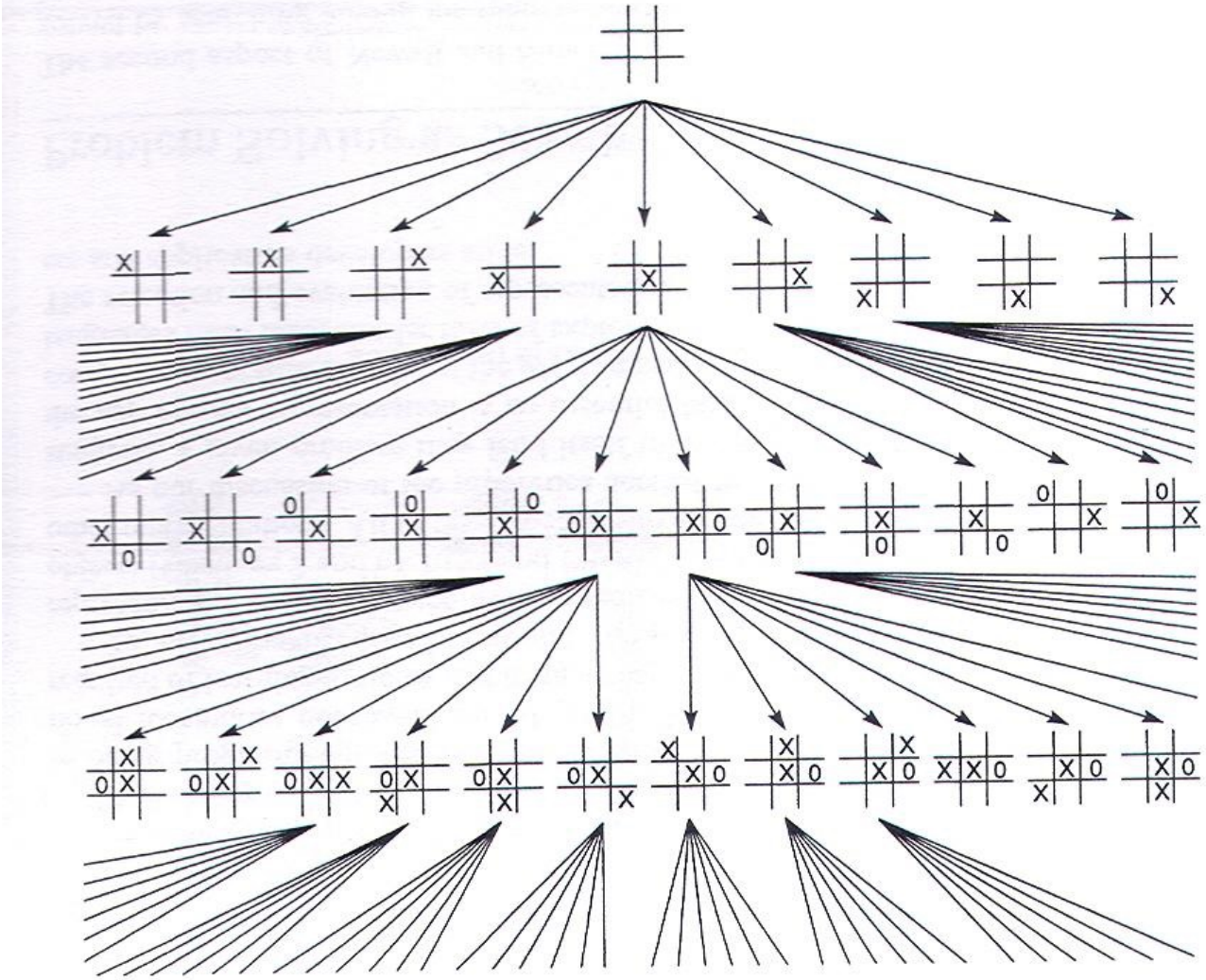
- Goal: Get from Arad to Bucharest

Search tree: route planning



- Search trees:
  - The root is the initial state, and the tree is un-directed
    - The car can go back to where it comes from
  - Represent the branching paths through a state graph
  - At each node, the goal test is checked
  - Usually **much** larger than the state graph
- Question: Can a finite state graph give an infinite search tree?
  - Answer:

Search tree of the game Tic-Tac-Toe

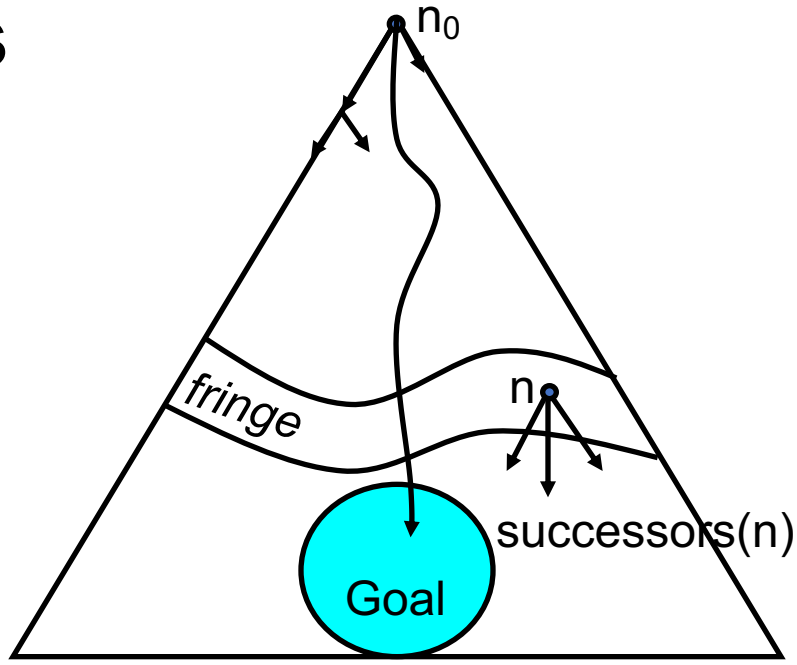


Question: Is this search tree finite or infinite?

Answer:

Tree search algorithms

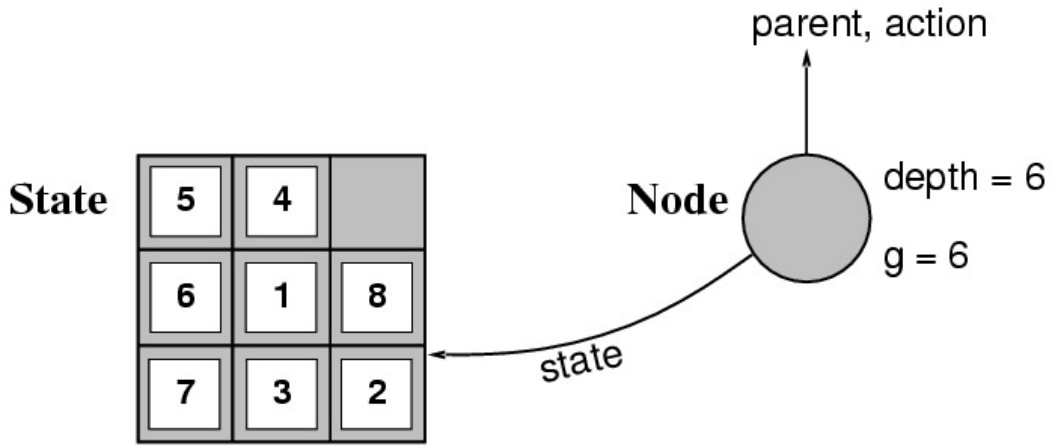
- Basic idea:
  - offline, simulated exploration of state space by generating successors of already-explored states



function TREE-SEARCH(*problem, strategy*) **returns** a solution, or failure
  initialize the search tree using the initial state of *problem*
  loop do
    if there are no candidates for expansion **then return** failure
    choose a leaf node for expansion according to *strategy*
    if the node contains a goal state **then return** the corresponding solution
    else expand the node and add the resulting nodes to the search tree

Implementation: states vs. nodes

- A state is a (representation of) a physical configuration
- A node is a data structure constituting part of a search tree including state, parent node, action, path cost $g(x)$, depth



- We'll use an `Expand` function to create new nodes (with their attributes)
- We will use a `SuccessorFn` of the problem to create the corresponding states

Implementation: general tree search

```
function TREE-SEARCH( problem, fringe) returns a solution, or failure
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe)
        if GOAL-TEST[problem](STATE[node]) then return SOLUTION(node)
        fringe ← INSERTALL(EXPAND(node, problem), fringe)

function EXPAND( node, problem) returns a set of nodes
    successors ← the empty set
    for each action, result in SUCCESSOR-FN[problem](STATE[node]) do
        s ← a new NODE
        PARENT-NODE[s] ← node;  ACTION[s] ← action;  STATE[s] ← result
        PATH-COST[s] ← PATH-COST[node] + STEP-COST(node, action, s)
        DEPTH[s] ← DEPTH[node] + 1
        add s to successors
    return successors
```

Search strategies

- A search 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?
  - time complexity: number of nodes generated
  - space complexity: maximum number of nodes in memory
  - optimality: does it always find a least-cost solution?

- Time and space complexity are usually measured in terms of
  - *b:* maximum branching factor of the search tree
  - *d:* depth of the least-cost solution (shallowest node achieving goal)
    - The root is at depth 0
  - *m*: maximum depth of the state space (may be ∞)

Queues

- Different search strategies differ according to the order in which the nodes in the fringe are expanded
  - FIFO: First in, first out
  - LIFO: Last in, first out
  - Priority queue: data structure in which you can insert and retrieve (key, value) pairs with the following operations:

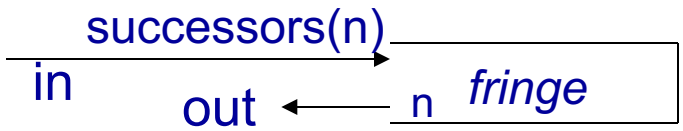
pq.setPriority(key, value)	inserts *(key, value)* into the queue.
pq.dequeue()	returns the key with the lowest value and removes it

- You can promote or demote keys by resetting their priorities
- Unlike a regular queue, insertions into a priority queue are not constant time, usually O(log *n*)
- We'll need priority queues for most methods based on costs

Uninformed search strategies

- Uninformed search strategies use only the information available in the problem definition; they include the following
- Breadth-first search
  - Expand shallowest unexpanded node
  - *fringe* = queue (FIFO)

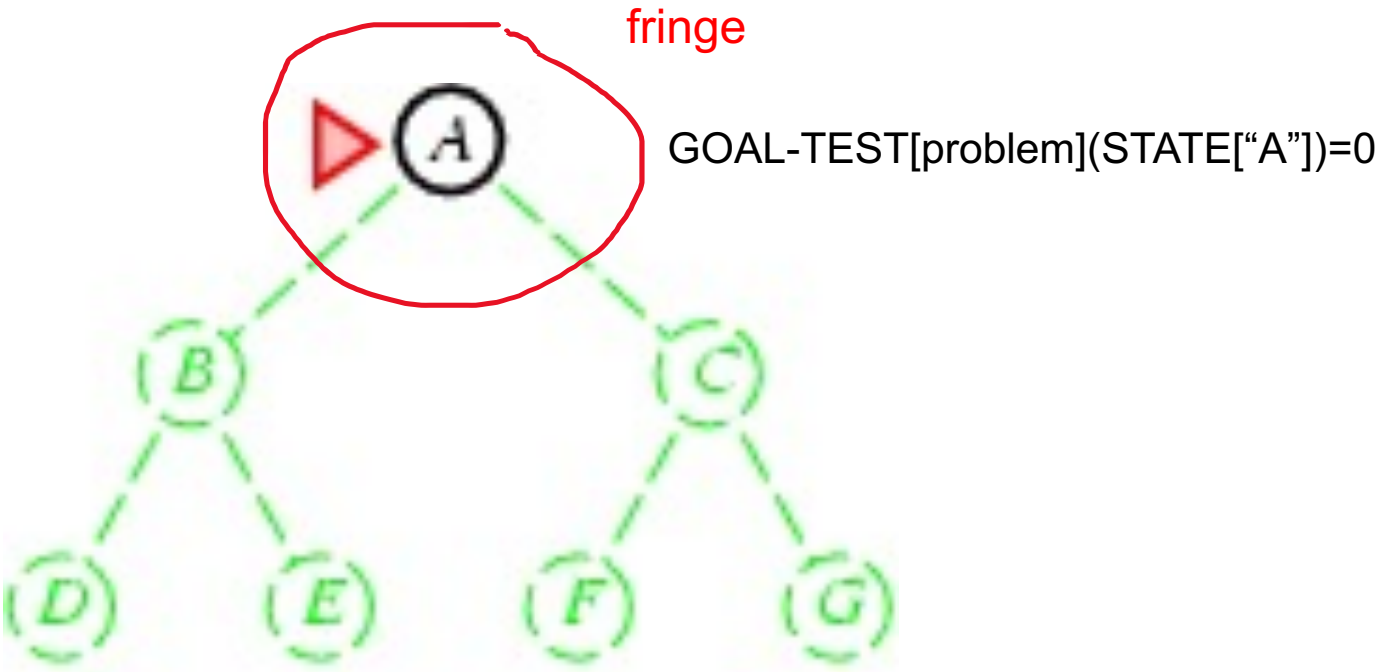
- Uniform-cost search
  - Expand cheapest unexpanded node
  - *fringe* = priority queue ordered by path cost
- Depth-first search
  - Expand deepest unexpanded node
  - *fringe* = stack (LIFO)



- Depth-limited search: depth-first search with depth limit
- Iterative deepening search

Breadth-first search

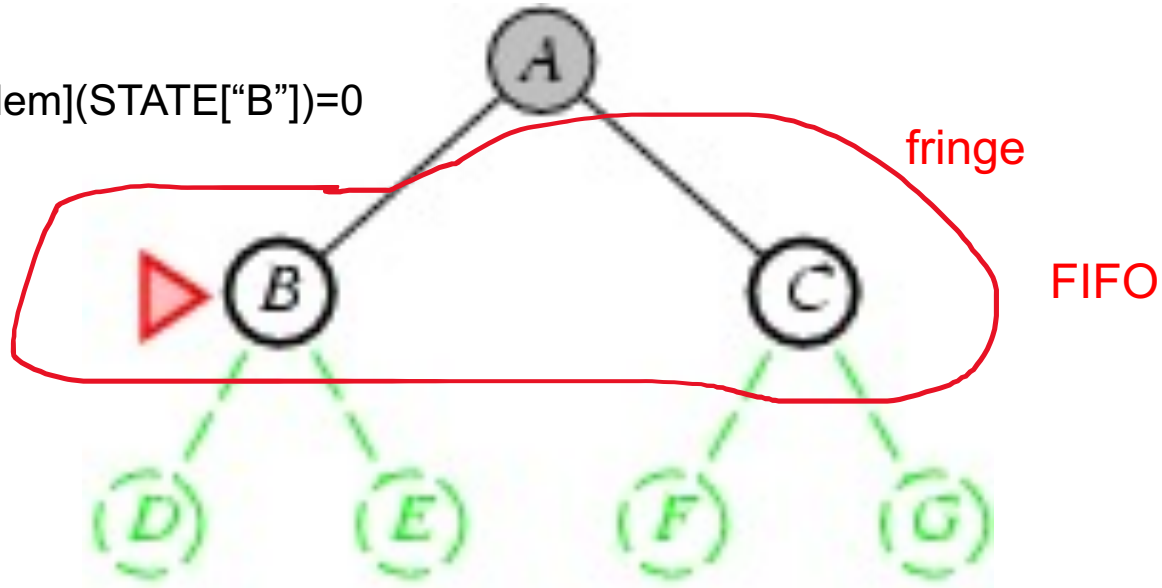
- Expand shallowest unexpanded node



Breadth-first search

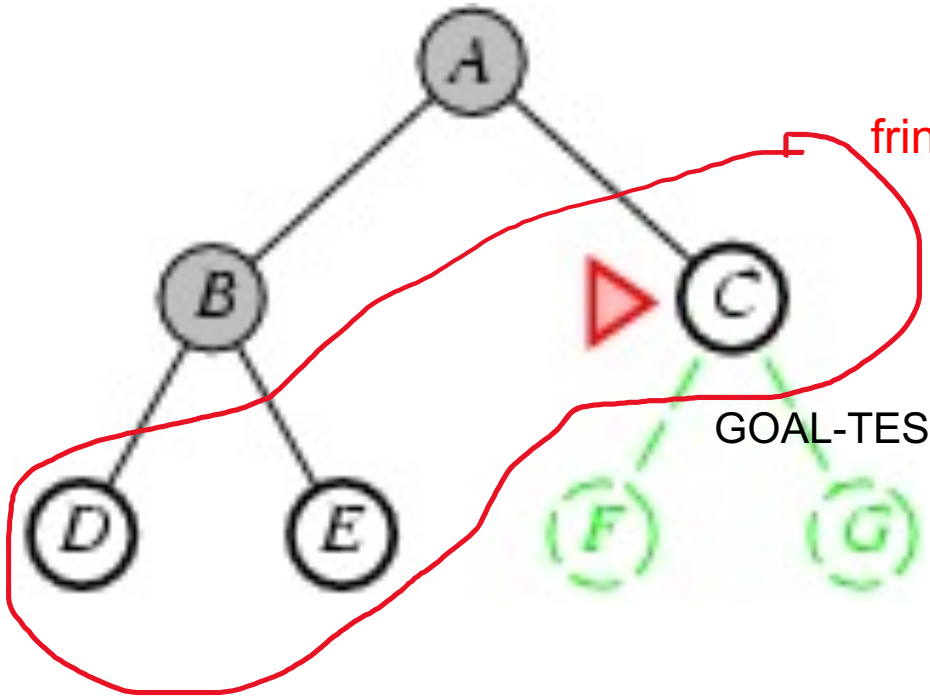
- Expand shallowest unexpanded node

GOAL-TEST[problem](STATE[“B”])=0



Breadth-first search

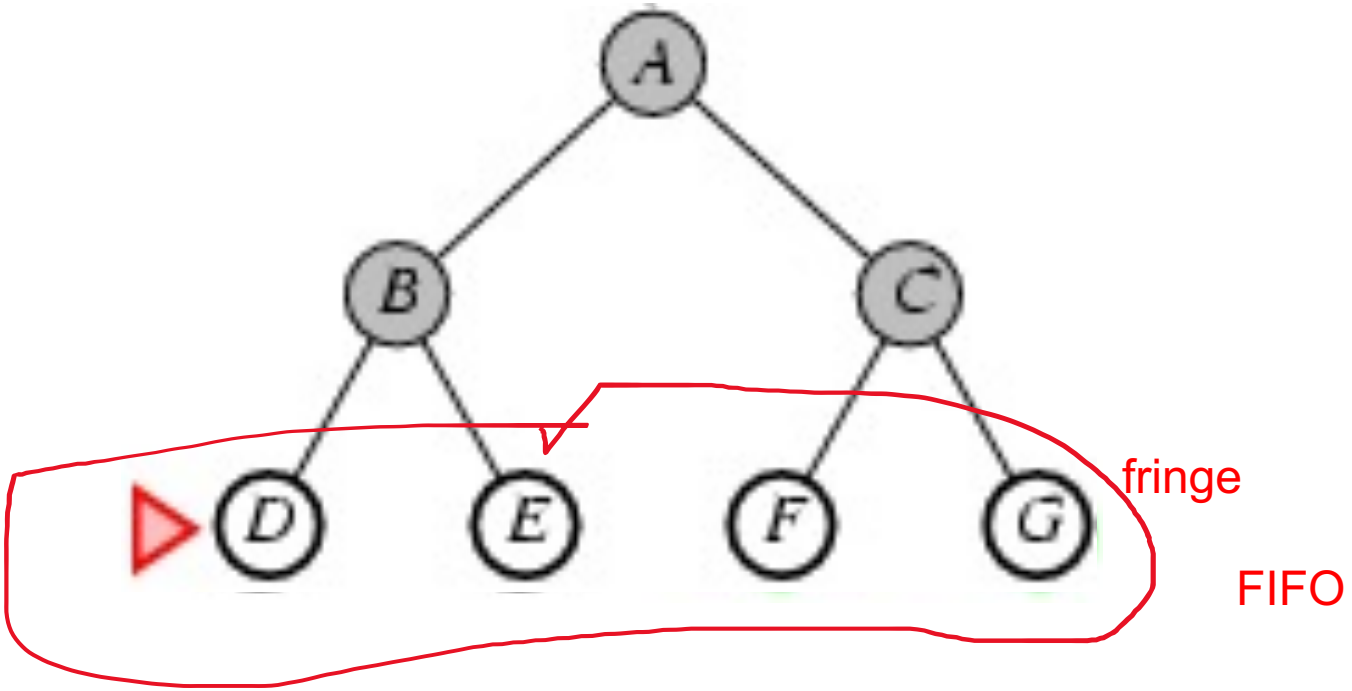
- Expand shallowest unexpanded node



$fringe \leftarrow \text{INSERTALL}(\text{EXPAND}(node, problem), fringe)$

Breadth-first search

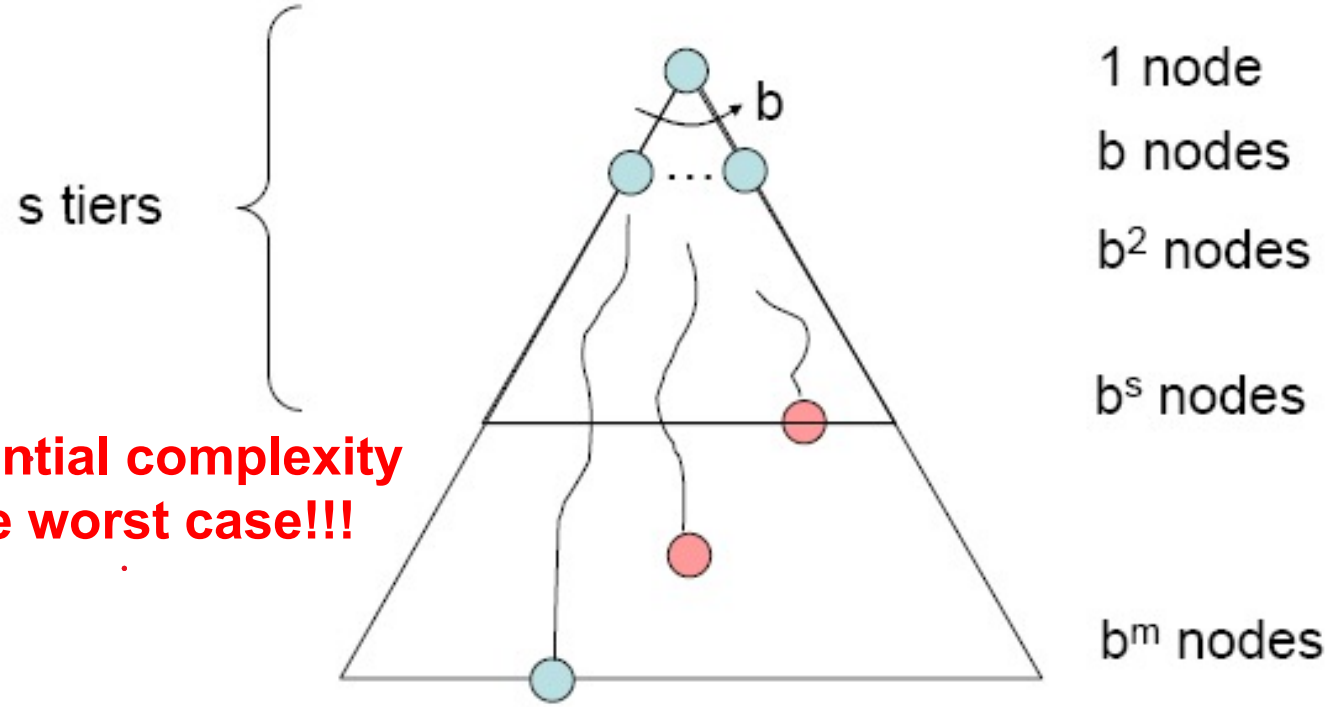
- Expand shallowest unexpanded node



Now we consider GOAL-TEST[problem](STATE[“D”]), etc

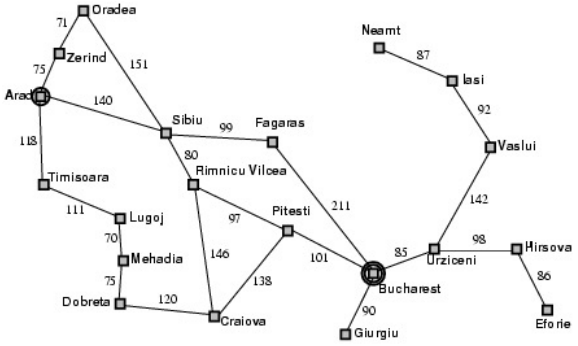
Breadth-first search (con't)

- Complete? Yes (if *b* is finite)
- Time? $1+b+b^2+b^3+\ldots+b^d = O(b^d)$ or $O(b^{d+1})$
- Space? $O(b^d)$ (keeps every node in memory)
- Optimal? Yes (if cost = 1 per step)



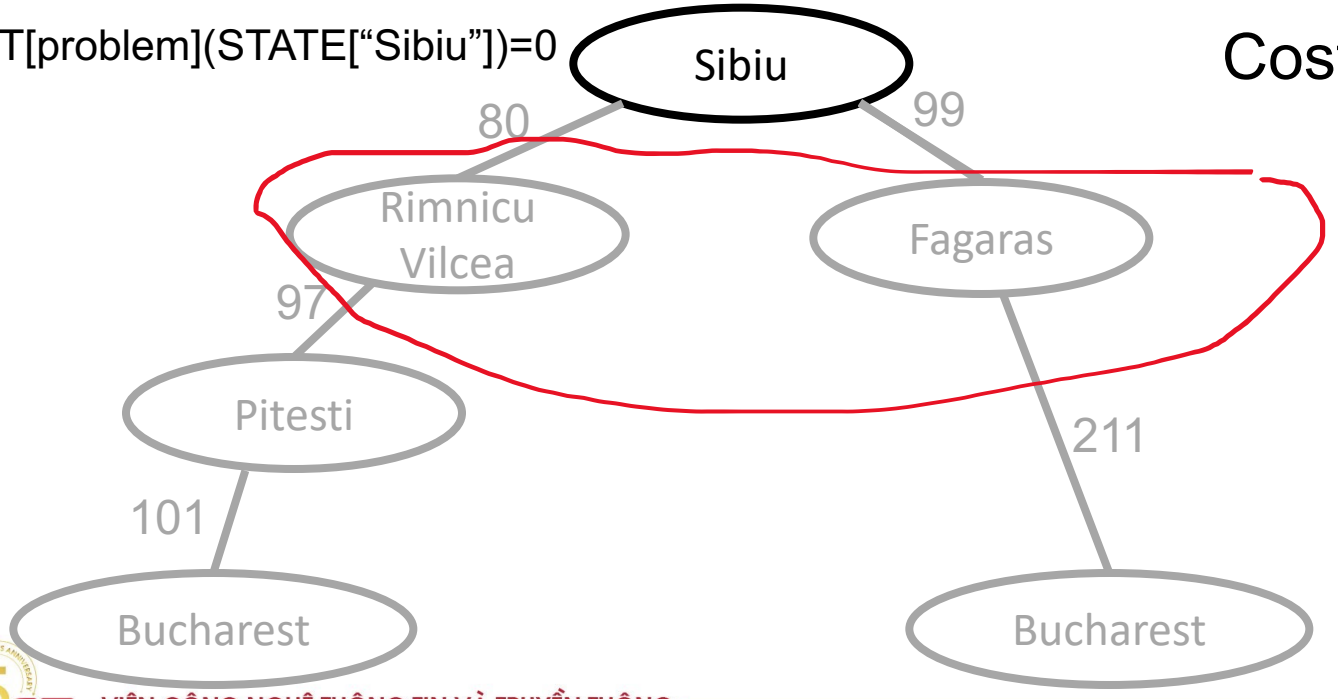
Exponential complexity in the worst case!!!

Uniform-cost search



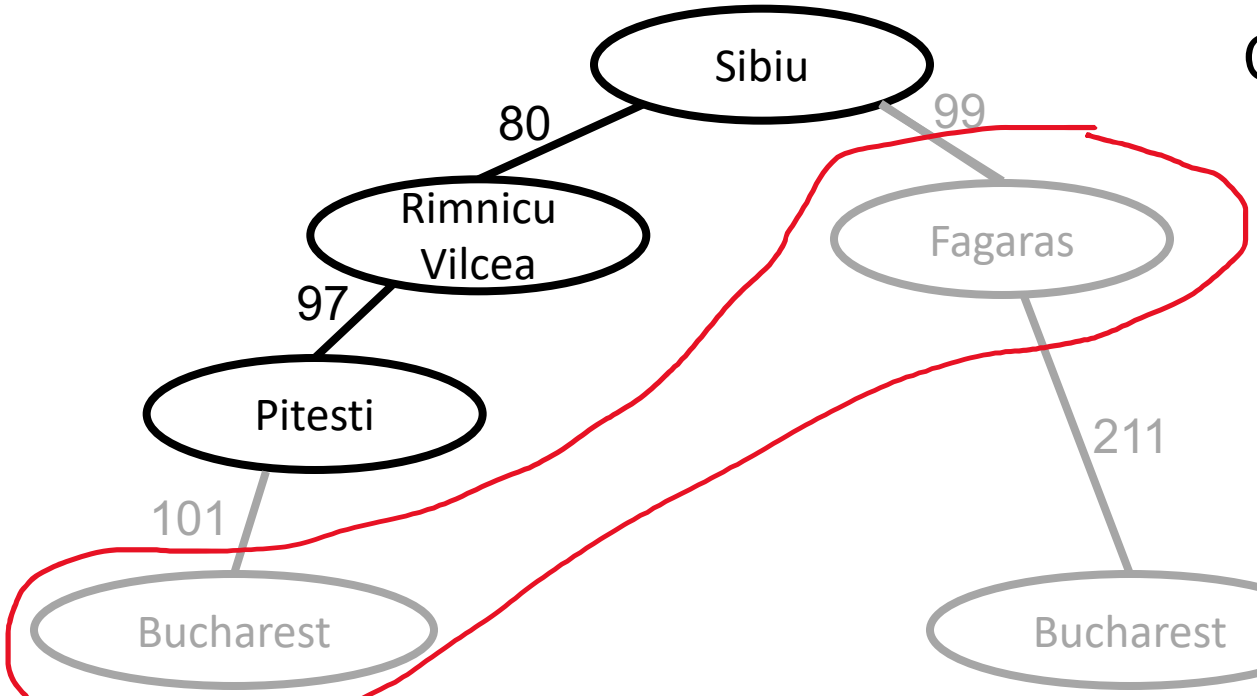
- May be seen as a breadth-first that takes into account step costs
- Expand cheapest unexpanded node n (cost so far to reach n)
  - Here cost=number of km **TO REACH THE NODE IN THE FRINGE!**
- *fringe* = queue ordered by path cost
- Example: go from Sibiu to Bucharest with no U-turns, and uniform-cost search

GOAL-TEST[problem](STATE[“Sibiu”])=0



Uniform-cost search

- Example: go from Sibiu to Bucharest with no U-turns, and uniform-cost search
  - Here cost=number of km **TO REACH THE NODE IN THE FRINGE**!
  - We have expanded the least-cost node from the fringe (Rimnicu Vilcea)

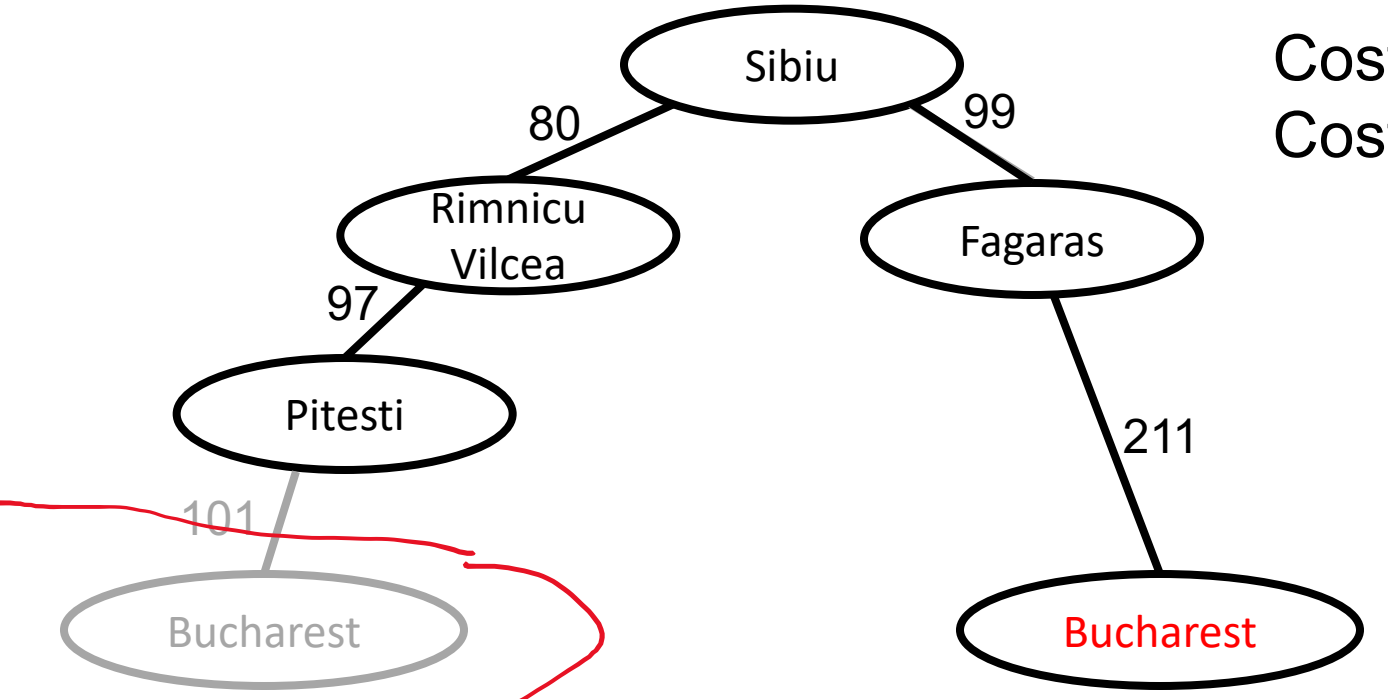


Cost=80+97=177

GOAL-TEST[problem](STATE[“Pitesti”])=0

Uniform-cost search

- Example: go from Sibiu to Bucharest with no U-turns, and uniform-cost search
  - Here cost=number of km **TO REACH THE NODE IN THE FRINGE**!
  - We have expanded the least-cost node from the fringe (Fagaras)



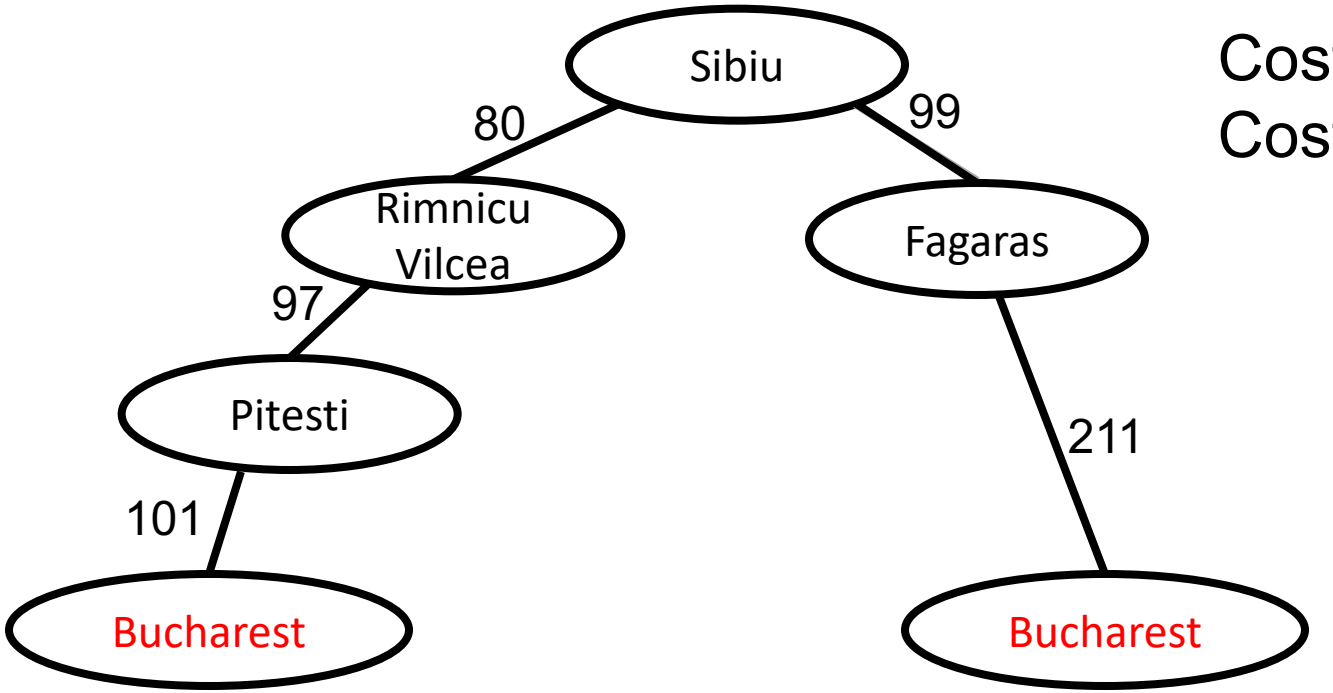
Cost1=177
Cost2=99+211=310

But, we're still going to expand the other nodes in the same level!

GOAL-TEST[problem](STATE["Bucharest"])=1

Uniform-cost search

- Example: go from Sibiu to Bucharest with no U-turns, and uniform-cost search
  - Here cost=number of km **TO REACH THE NODE IN THE FRINGE**!

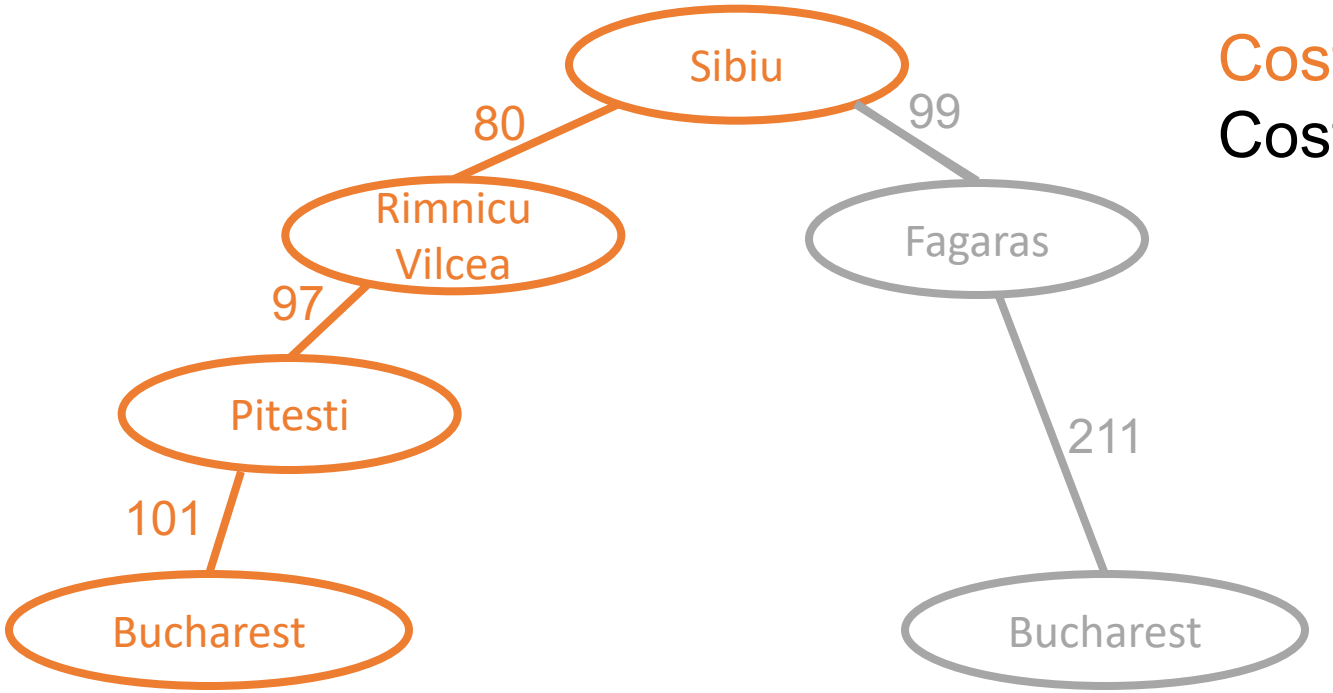


Cost1=177+101=278
Cost2=99+211=310

GOAL-TEST[problem](STATE[“Bucharest”])=1

Uniform-cost search

- Example: go from Sibiu to Bucharest with no U-turns, and uniform-cost search
  - Here cost=number of km **TO REACH THE NODE IN THE FRINGE**!



Cost1=177+101=278

Cost2=99+211=310

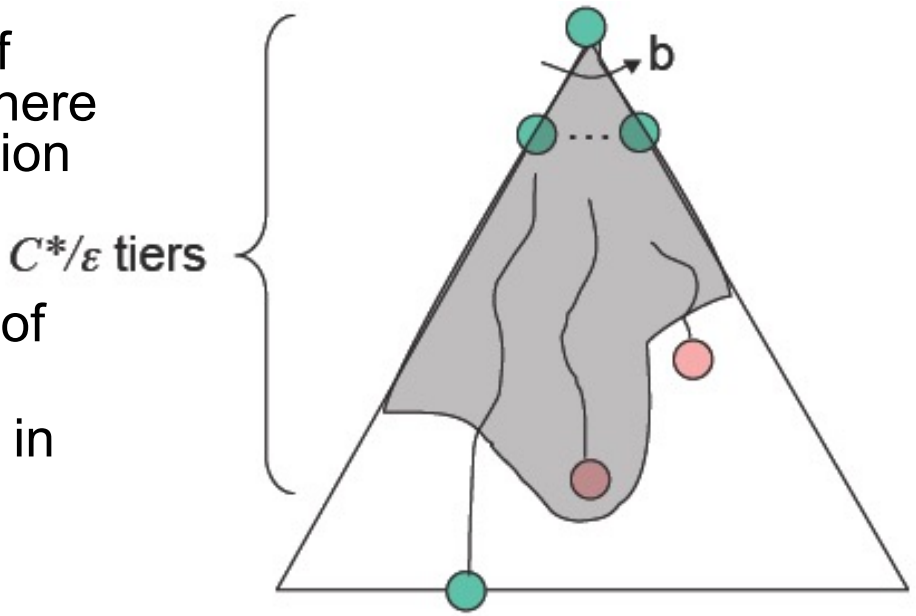
Both paths lead to the goal. So now, we choose the cheapest one!

Uniform-cost search

- May be seen as a breadth-first that takes into account step costs
- Expand cheapest unexpanded node n (cost so far to reach n)
- *fringe* = queue ordered by path cost

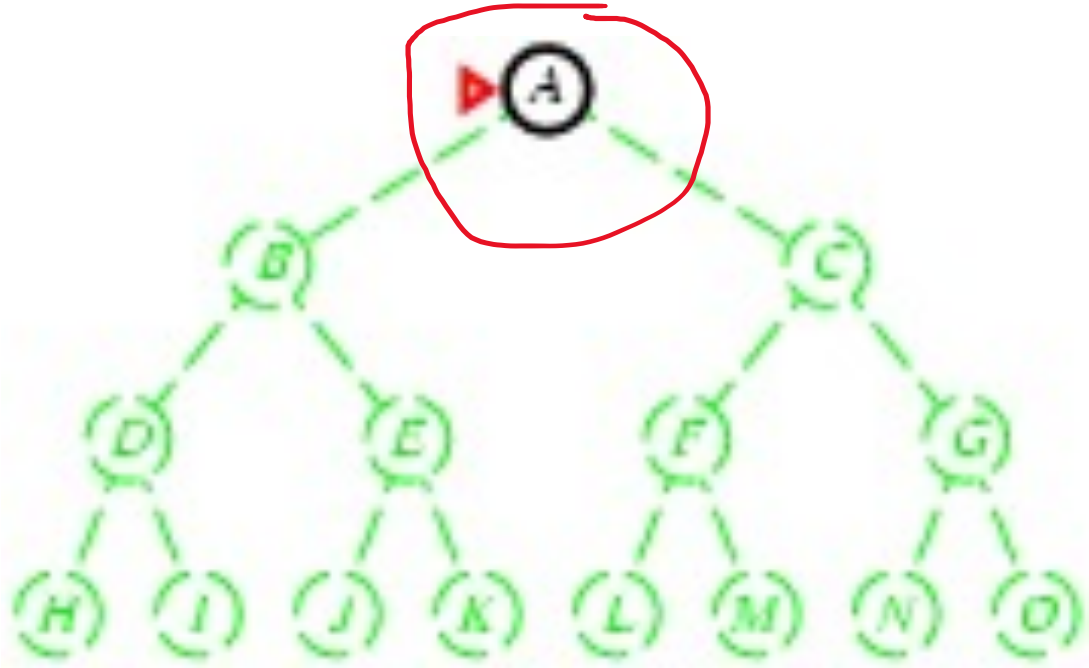
- Complete? Yes, if step cost ≥ ε
- Time? # of nodes with $g \leq$ cost of optimal solution, $O(b^{ceiling(C^*/\varepsilon)})$ where C^* is the cost of the optimal solution

- Space? # of nodes with $g \leq$ cost of optimal solution, $O(b^{ceiling(C^*/\varepsilon)})$
- Optimal? Yes – nodes expanded in increasing order of $g(n)$



Depth-first search

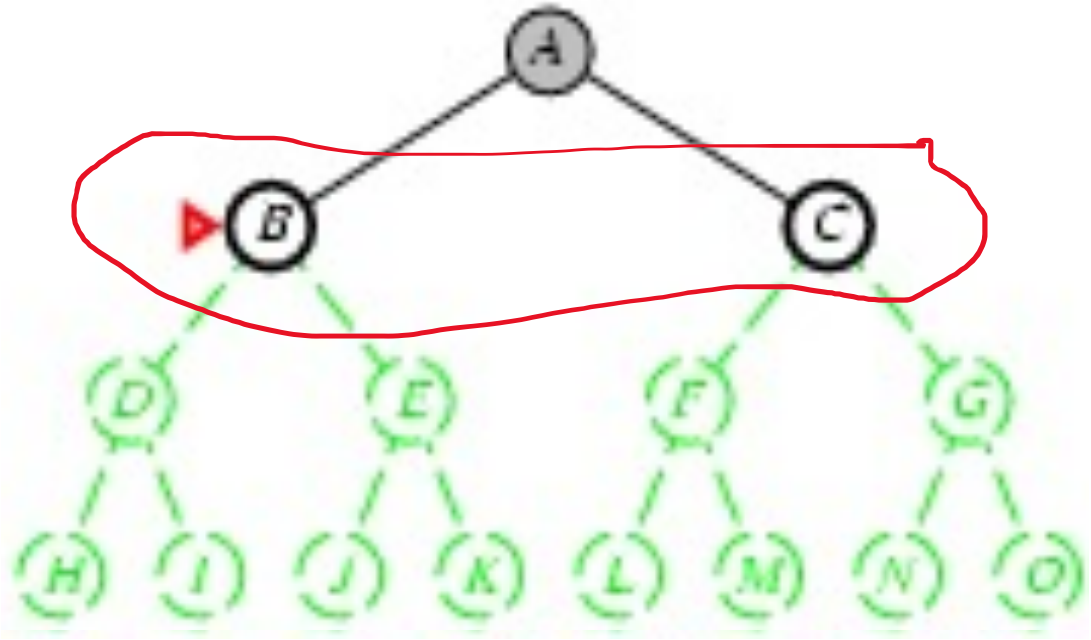
- Expand deepest unexpanded node



fringe

Depth-first search

- Expand deepest unexpanded node

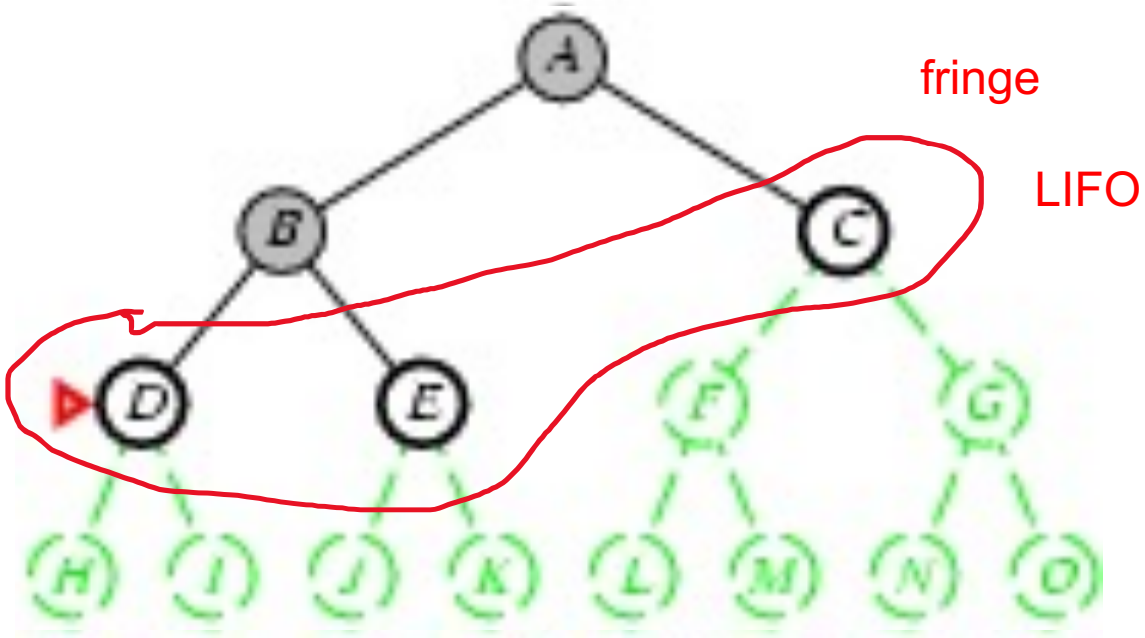


fringe

LIFO

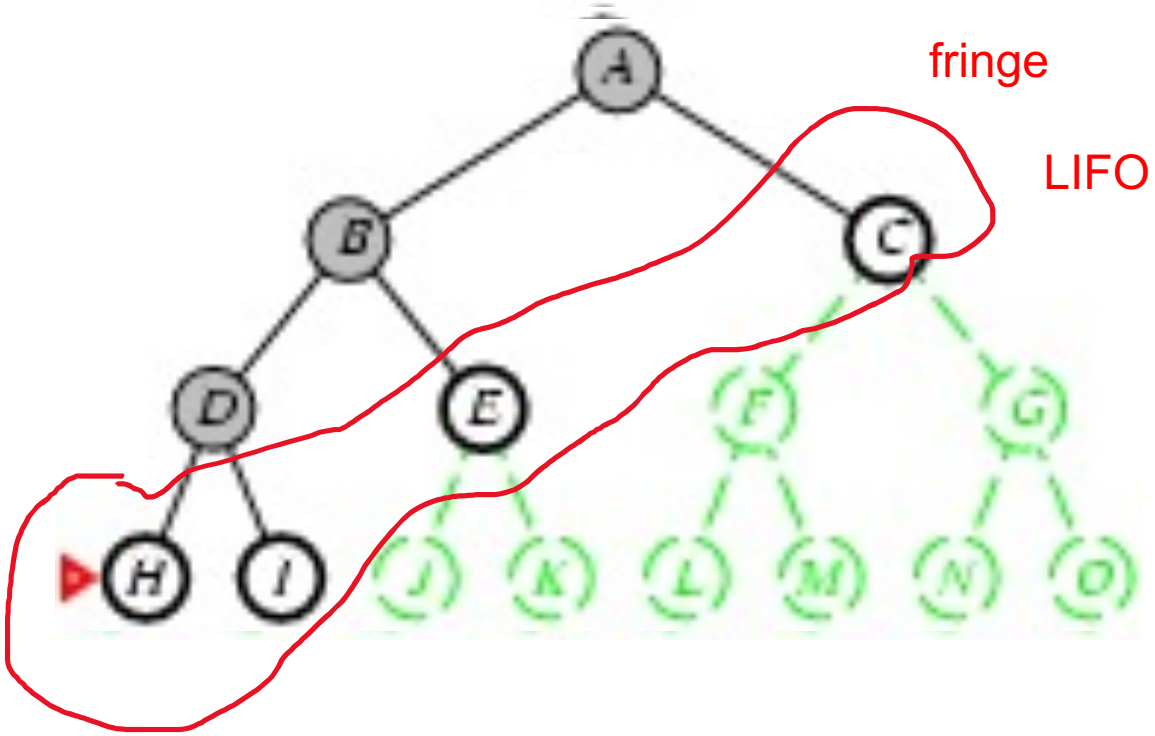
Depth-first search

- Expand deepest unexpanded node



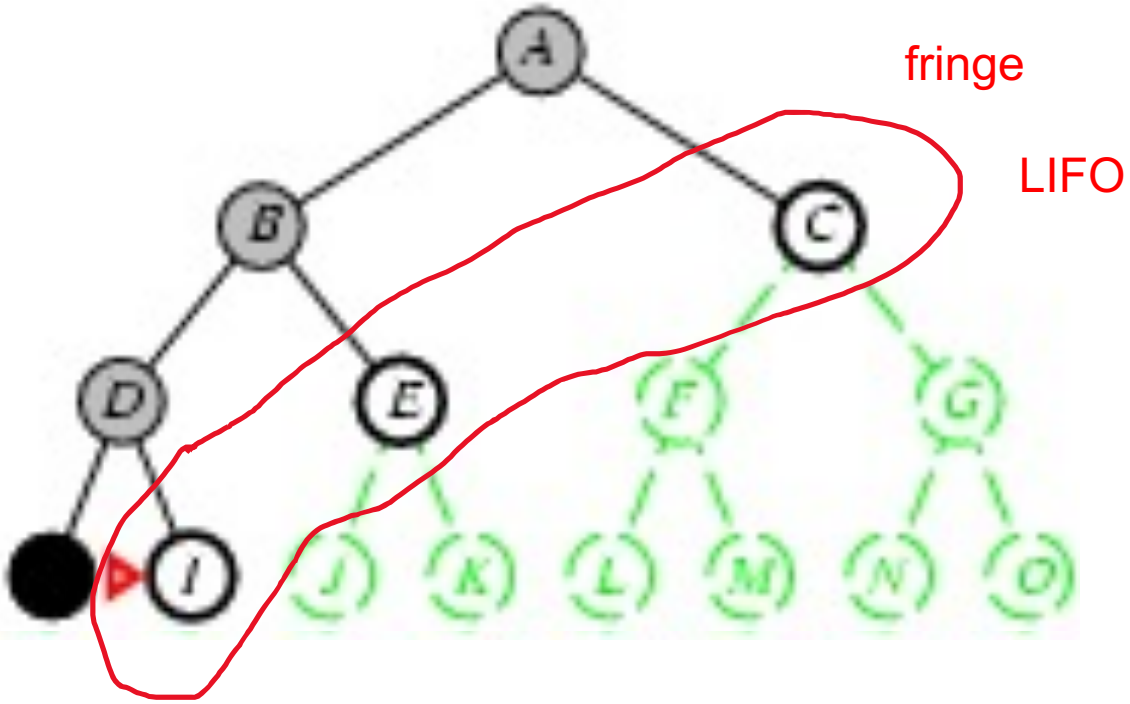
Depth-first search

- Expand deepest unexpanded node



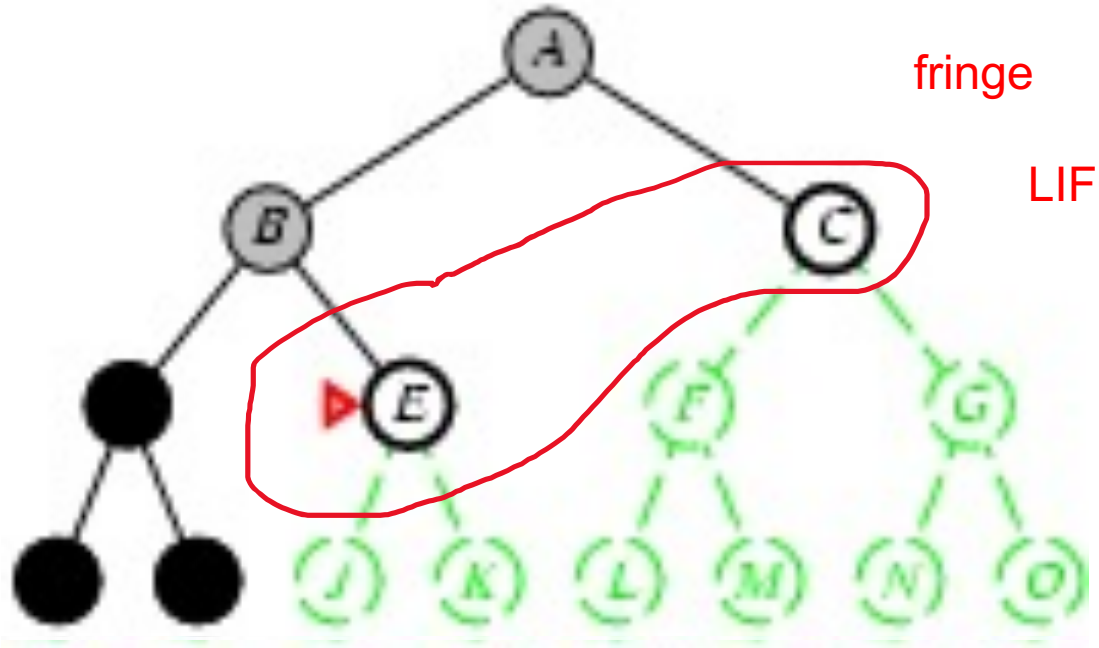
Depth-first search

- Expand deepest unexpanded node



Depth-first search

- Expand deepest unexpanded node



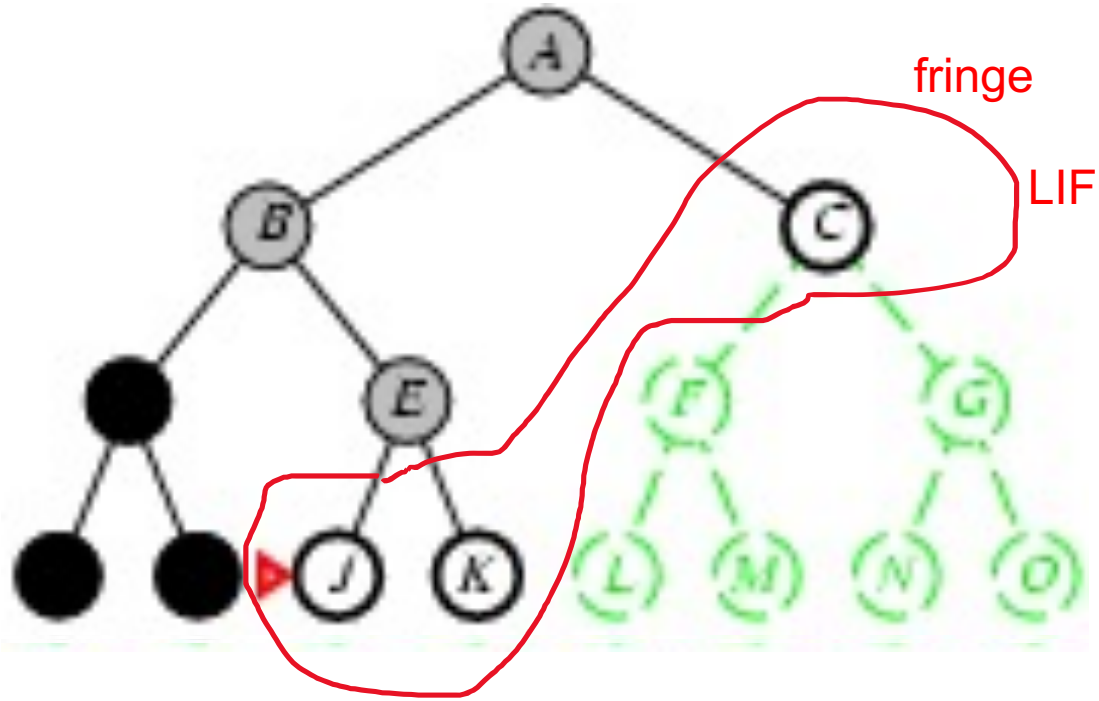
fringe

LIFO

Explored final nodes / paths are removed from the fringe

Depth-first search

- Expand deepest unexpanded node

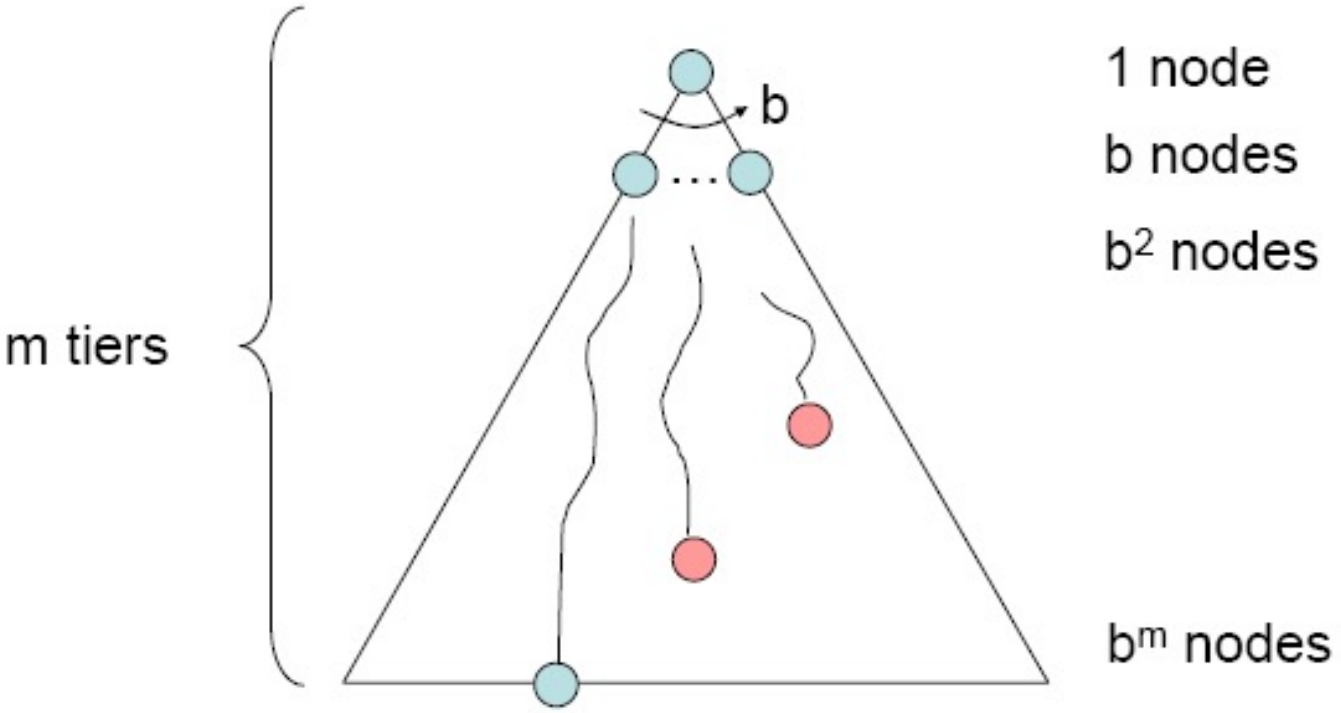


Explored final nodes / paths are removed from the fringe

We stop as soon as we found a node that verifies the goal

Depth-first search (con't)

- Complete? No: fails in infinite-depth spaces, spaces with loops
  - Modify to avoid repeated states along path → complete in finite spaces
- Time? $O(b^m)$: terrible if m is much larger than d
  - but if solutions are dense, may be much faster than breadth-first
- Space? $O(bm)$, *i.e.* linear space!
- Optimal? No



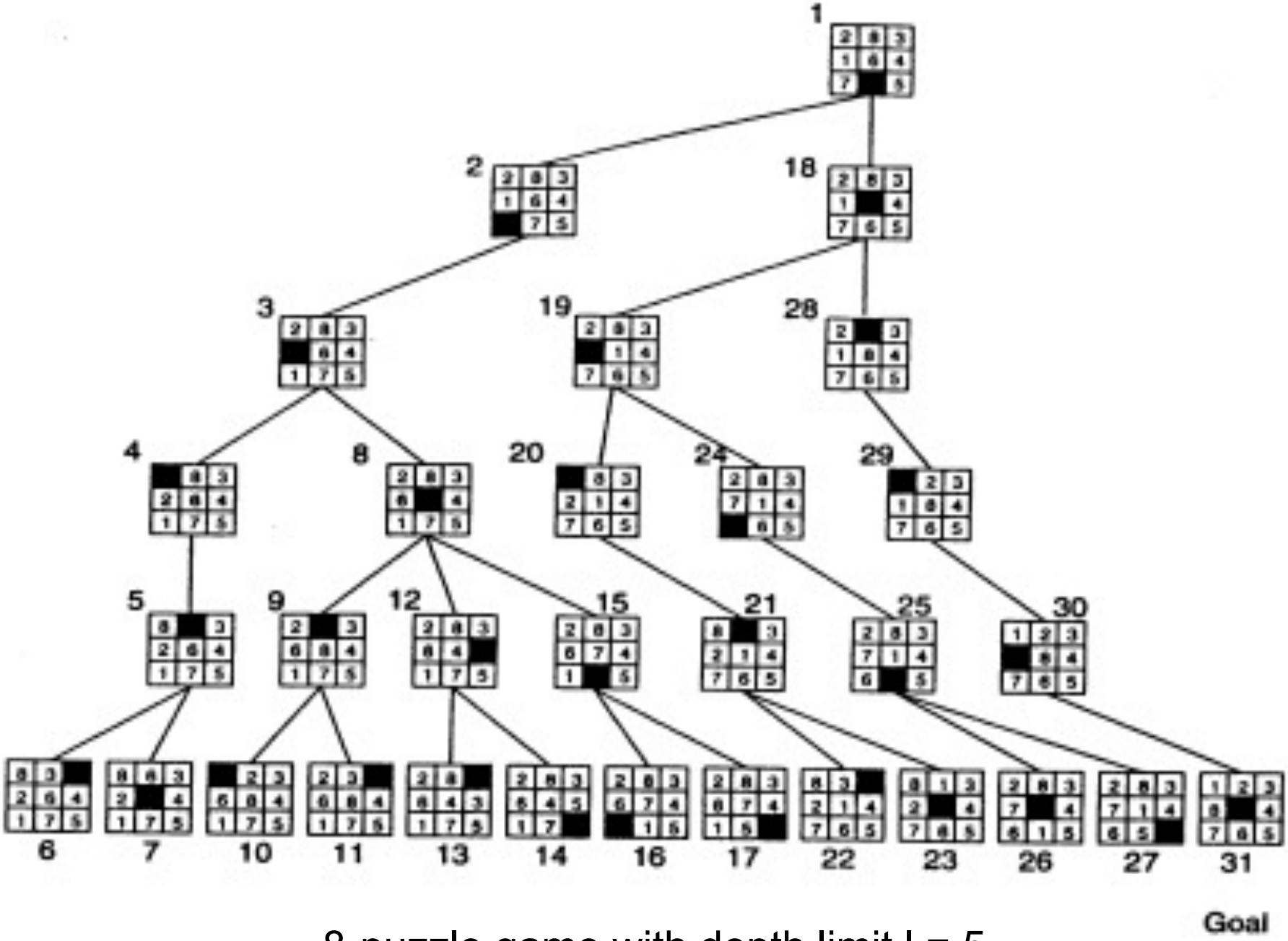
Depth-limited search

- Depth-first search can get stuck on infinite path when a different choice would lead to a solution

$\Rightarrow$ Depth-limited search = depth-first search with depth limit *l,* i.e., nodes at depth *l* have no successors

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

function RECURSIVE-DLS(node, problem, limit) returns soln/fail/cutoff
    cutoff-occurred? ← false
    if GOAL-TEST[problem](STATE[node]) then return SOLUTION(node)
    else if DEPTH[node] = limit then return cutoff
    else for each successor in EXPAND(node, problem) do
        result ← RECURSIVE-DLS(successor, problem, limit)
        if result = cutoff then cutoff-occurred? ← true
        else if result ≠ failure then return result
    if cutoff-occurred? then return cutoff else return failure
```



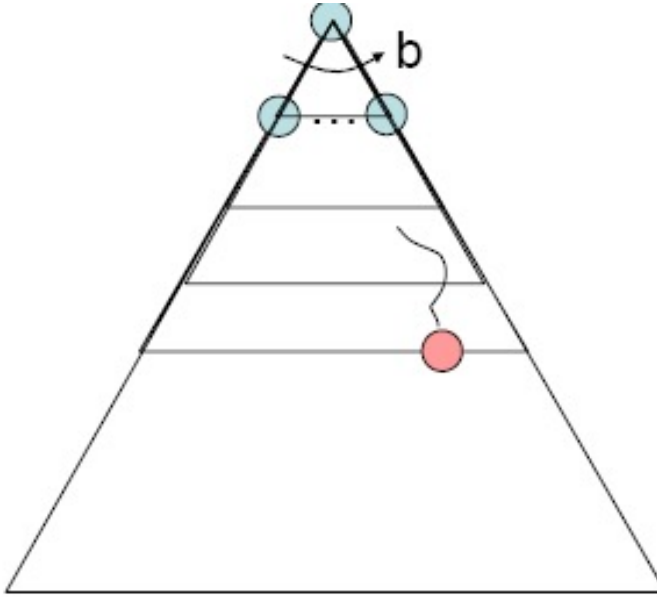
8-puzzle game with depth limit l = 5

Iterative deepening search

- Problem with depth-limited search: if the shallowest goal is beyond the depth limit, no solution is found.

⇒ Iterative deepening search:

1. Do a DFS which only searches for paths of length 1 or less. (DFS gives up on any path of length 2)
2. If "1" failed, do a DFS which only searches paths of length 2 or less.
3. If "2" failed, do a DFS which only searches paths of length 3 or less.
4. ….and so on.



function ITERATIVE-DEEPENING-SEARCH(*problem*) **returns** a solution, or failure

inputs: *problem*, a problem

for *depth* ← 0 **to** ∞ **do**
  result ← DEPTH-LIMITED-SEARCH(*problem*, *depth*)
  if *result* ≠ cutoff **then return** *result*

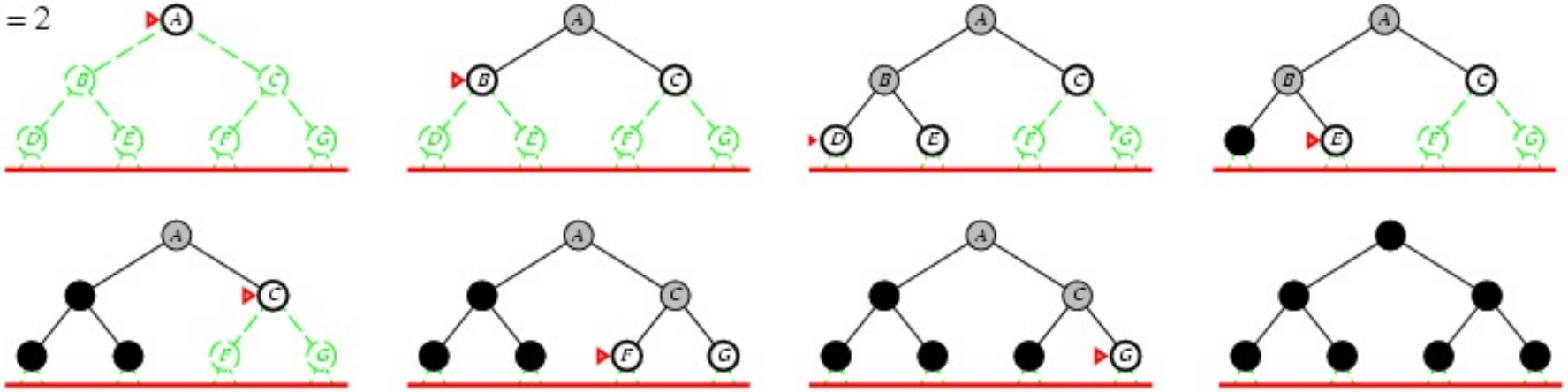
63

Iterative deepening search (con't)

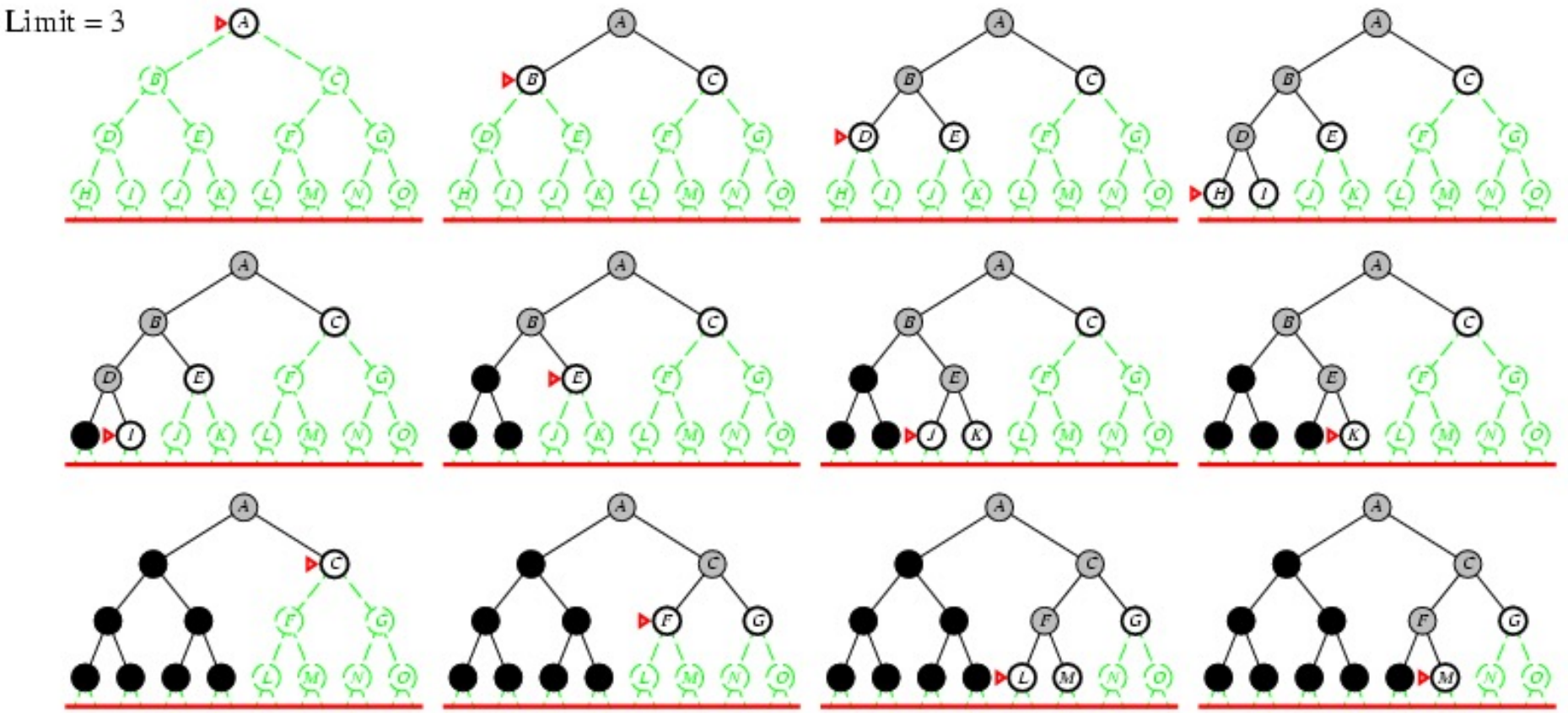
Limit = 0

Limit = 1

Limit = 2



Iterative deepening search (con't)



Iterative deepening search (con't)

- Number of nodes generated in a **depth-limited search** to depth *d* with branching factor *b*:

$$N_{DLS} = b^0 + b^1 + b^2 + \ldots + b^{d-2} + b^{d-1} + b^d$$

- Number of nodes generated in an **iterative deepening search** to depth *d* with branching factor *b*:

$$N_{IDS} = (d+1)b^0 + d\,b\text{^}1 + (d-1)b\text{^}2 + \ldots + 3b^{d-2} + 2b^{d-1} + 1b^d$$

- **N.B.** In most cases, the waste of computational time is not so much

- Example: for *b = 10*, *d = 5*,
  - $N_{DLS} = 1 + 10 + 100 + 1{,}000 + 10{,}000 + 100{,}000 = 111{,}111$
  - $N_{IDS} = 6 + 50 + 400 + 3{,}000 + 20{,}000 + 100{,}000 = 123{,}456$
  - Overhead = (123,456 - 111,111)/111,111 = 11%

Properties of iterative deepening search

- Complete? Yes
- Time? $(d+1)b^0 + d\,b^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$
- Space? $O(bd)$
- Optimal? Yes, if step cost = 1
  - Or at least when the path cost is a non-decreasing function of depth

Chapter 3 - part1

Summary

Summary of algorithms

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening
Complete?	Yes	Yes	No	No	Yes
Time	$O(b^{d+1})$	$O(b^{\lceil C^*/\epsilon \rceil})$	$O(b^m)$	$O(b^l)$	$O(b^d)$
Space	$O(b^{d+1})$	$O(b^{\lceil C^*/\epsilon \rceil})$	$O(bm)$	$O(bl)$	$O(bd)$
Optimal?	Yes	Yes	No	No	Yes

Iterative deepening combines the advantages of BFS and DFS:

- Like DFS, it consumes less memory
- Like BFS, it is complete when *b* is finite, and is optimal when the path cost is a non-decreasing function of depth.

In general, iterative deepening is the preferred search method **when there is a large search space and the depth of the solution is not known**

Chapter 3 - part1

Homework

Homework (assignment on Teams)

- Exercise 1:
  - Consider the shepherd, goat, wolf, cabbage problem
  - 1.1. Represent the simplified state space using a tree
    - See slide 27 for the graph
  - 1.2. Specify the values of b, d and m
  - 1.3. Display (on paper or using software) all the steps of breadth-first search
    - Specify the space complexity in practice, and in the worst case
    - Specify the time complexity in practice, and in the worst case
  - 1.4. Display (on paper or using software) all the steps of depth-first search
    - Specify the space complexity in practice, and in the worst case
    - Specify the time complexity in practice, and in the worst case

Homework (assignment on Teams)

- Exercise 2:
  - Let us now consider that:
    - It is dangerous for the shepherd to transport the wolf: cost=3
    - Because of the goat's weight, it is also (but less) dangerous to transport the goat: cost=2
    - Transporting the shepherd only, or the shepherd+cabbage costs 1

  2.1. Display (on paper or using software) all the steps of uniform-cost search
    - Specify C* and epsilon
    - Specify the space complexity in practice, and in the worst case
    - Specify the time complexity in practice, and in the worst case

Chapter 3 – part 1

Questions

ĐẠI HỌC BÁCH KHOA

25 YEARS ANNIVERSARY SOICT

VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

Thank you for your attention!

soict.hust.edu.vn/ fb.com/groups/soict