

CSCE A405/A605 (Adv) Artificial Intelligence

Uninformed Search

Ref: Artificial Intelligence: A Modern Approach, 4th ed by Stuart Russell and Peter Norvig, chapter 3

Instructor: Masoumeh Heidari (mheidari2@Alaska.edu)

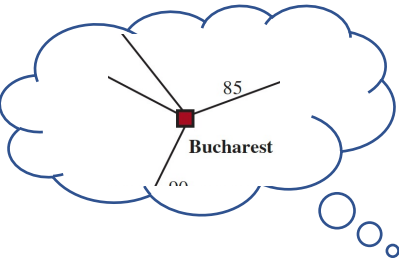
Learning objectives

- How to define a search problem?
- Uninformed search
 - Breadth-first search
 - Uniform cost search
 - Depth-first search
- Search data structures
- How to measure problem-solving performance?

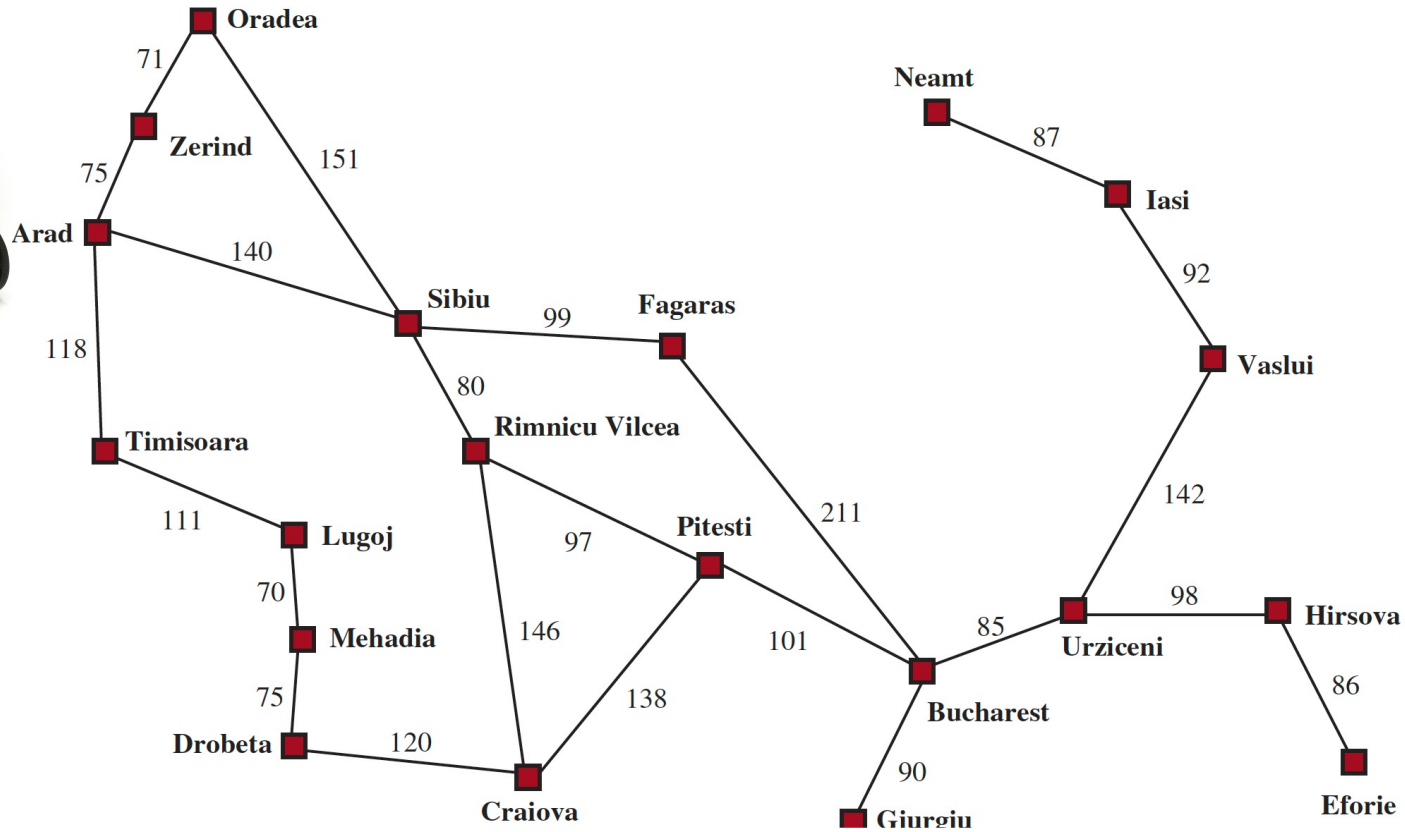
How an agent can look ahead to find a **sequence of actions** that will eventually achieve its **goal**?



Problem-solving agents



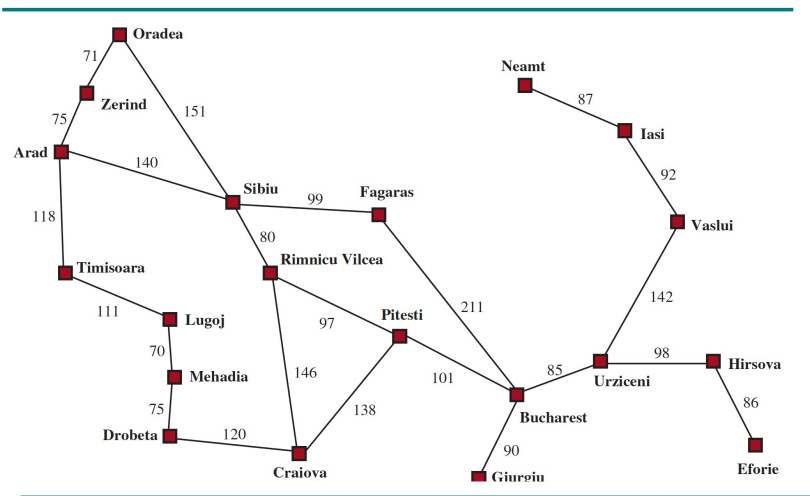
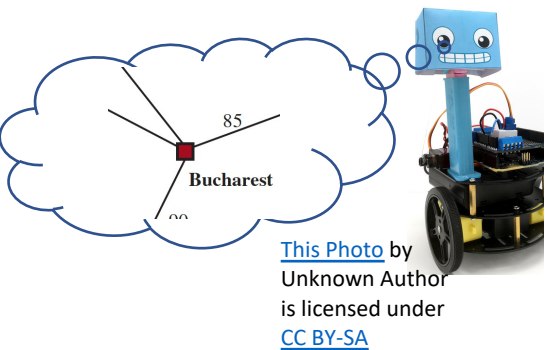
[This Photo](#)
Unknown Author
is licensed under
[CC BY-SA](#)



Search problem

- A search problem can be defined formally as follows:
 1. A set of possible states that the environment can be in. We call this the **state space**.
 2. The **initial state** that the agent starts in.
 3. A set of one or more **goal states**.
 4. The **actions** available to the agent.
 5. A **transition model** which describes what each action does.
 6. An **action cost function**, that gives the numeric cost of applying an action. A problem-solving agent should use a cost function that reflects its own performance measure.

How to define a search problem?



A few definitions:

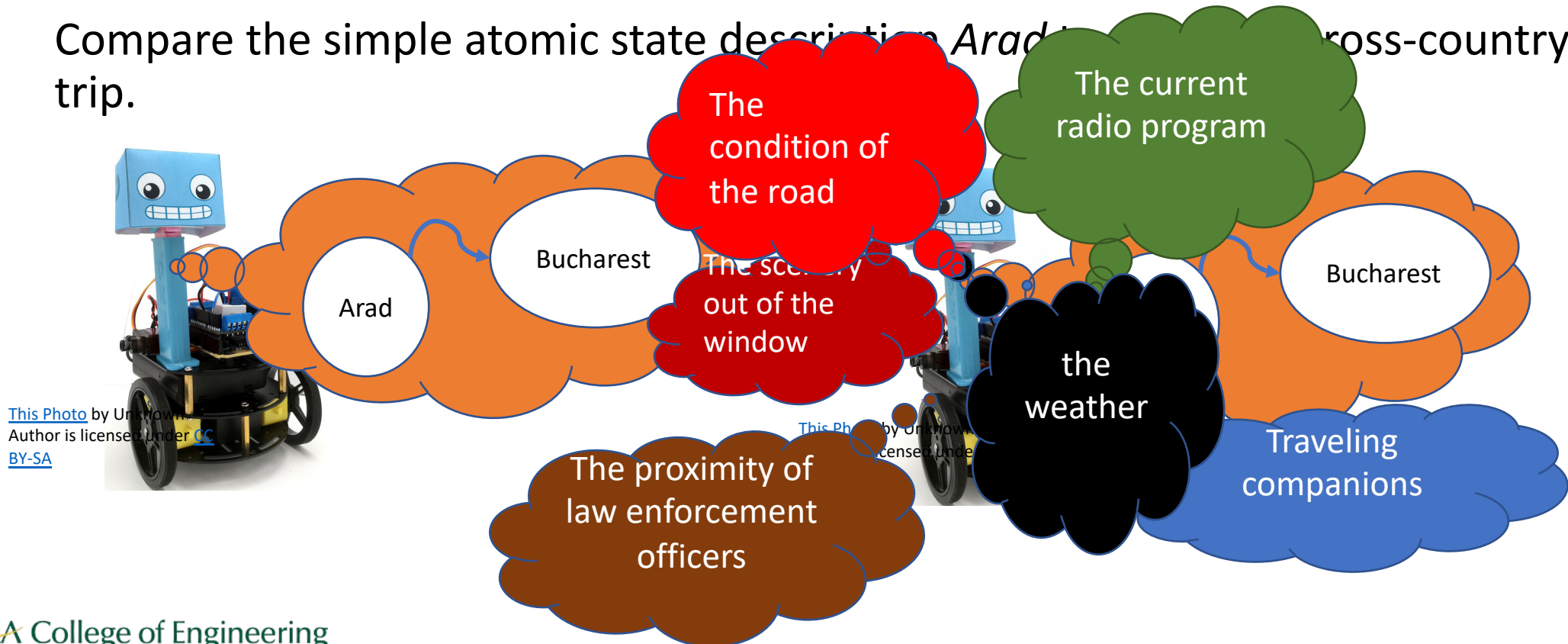
- A sequence of actions forms a **path**.
- A **solution** is a path from the initial state to a goal state.
- An **optimal solution** has the lowest path cost among all solutions.
- The state space can be represented as a graph in which the vertices are states and the directed edges between them are actions.

State space	Cities
Initial state	Arad
Goal state(s)	Bucharest
Actions	Actions(Arad) = {ToSibiu, ToTimisoara, ToZerind}
Transition model	RESULT(Arad, ToZerind) = Zerind
Action cost function	Length in miles / time it takes to complete the action

Formulating problems

- Our formulation of the problem of getting to Bucharest is a model—an abstract mathematical description—and not the real thing.

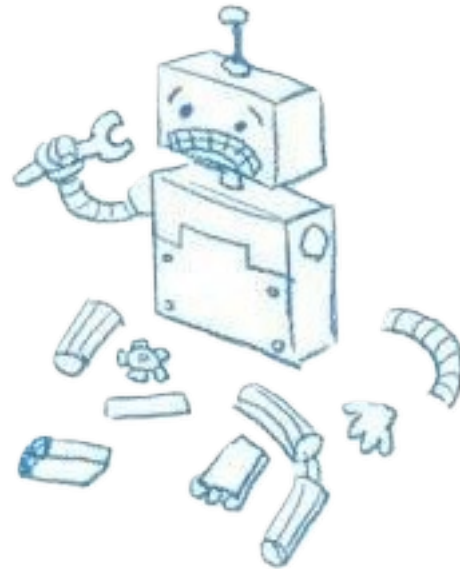
Compare the simple atomic state description *Arad to Bucharest* cross-country trip.



Formulating problems

- Our formulation of the problem of getting to Bucharest is a model—an abstract mathematical description—and not the real thing.

Compare the simple atomic state description *Arad* to an actual cross-country trip.



[This Photo](#) by Unknown Author is licensed under [CC BY-SA](#)

Formulating problems

- The process of removing detail from a representation is called **abstraction**.
- A good problem formulation has the right level of detail.
- The choice of a good abstraction thus involves removing as much detail as possible while retaining validity and ensuring that the abstract actions are easy to carry out.

Example problems

Problems

Standardized

Intended to illustrate or exercise various problem solving methods. It can be given a concise, exact description and hence is suitable as a benchmark for researchers to compare the performance of algorithms.

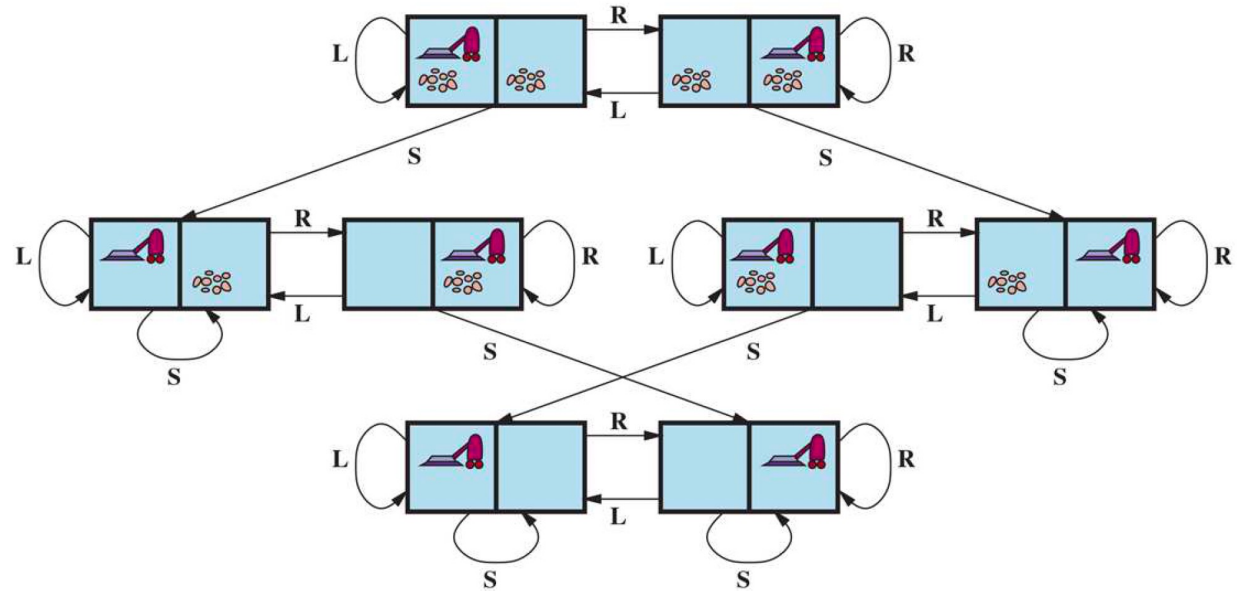
Real world

Such as robot navigation, is one whose solutions people actually use, and whose formulation is idiosyncratic, not standardized, because, for example, each robot has different sensors that produce different data.

Example problems

A *grid world* problem is a two-dimensional rectangular array of square cells in which agents can move from cell to cell.

The simple vacuum world can be formulated as a grid world.



The state-space graph for the two-cell vacuum world. There are 8 states and three actions for each state: L = *Left*, R = *Right*, S = *Suck*.

Example problems

STATES: The agent can be in either of the two cells, and each cell can either contain dirt or not, so there are $2 \times 2 \times 2 = 8$ states.

INITIAL STATE: Any state can be designated as the initial state.

ACTIONS: Suck, move Left, and move Right *or* Suck, Forward, Backward, TurnRight, and TurnLeft

TRANSITION MODEL: Suck removes any dirt from the agent's cell;
Forward moves the agent ahead one cell in the direction it is facing, unless it hits a wall, in which case the action has no effect.

Backward moves the agent in the opposite direction, while
TurnRight and TurnLeft change the direction it is facing by 90° .

GOAL STATES: The states in which every cell is clean.

ACTION COST: Each action costs 1.

Quiz 1 (Extra credit)

Three missionaries and three cannibals must cross a river using a boat which can carry at most two people, under the constraint that, for both banks, if there are missionaries present on the bank, they cannot be outnumbered by cannibals (if they were, the cannibals would eat the missionaries). The boat cannot cross the river by itself with no people on board.

STATES:

INITIAL STATE:

ACTIONS:

TRANSITION MODEL:

GOAL STATES:

ACTION COST:

Missionaries and Cannibals

STATES: Configuration of missionaries, cannibals, and boats on each side of river.

INITIAL STATE: 3 missionaries, 3 cannibals, and the boat are in the near bank (MMMCCCB in the near side).

ACTIONS: Move boat containing some set of occupants.

TRANSITION MODEL: for example (MMMCCCB) and action move(MC) leads to (MMCC in the near side) and (MCB in the far side).

GOAL STATES: (MMMCCCB in the far side).

ACTION COST: If we want minimum number of movement, we can consider that each movement cost 1.

Missionaries and Cannibals

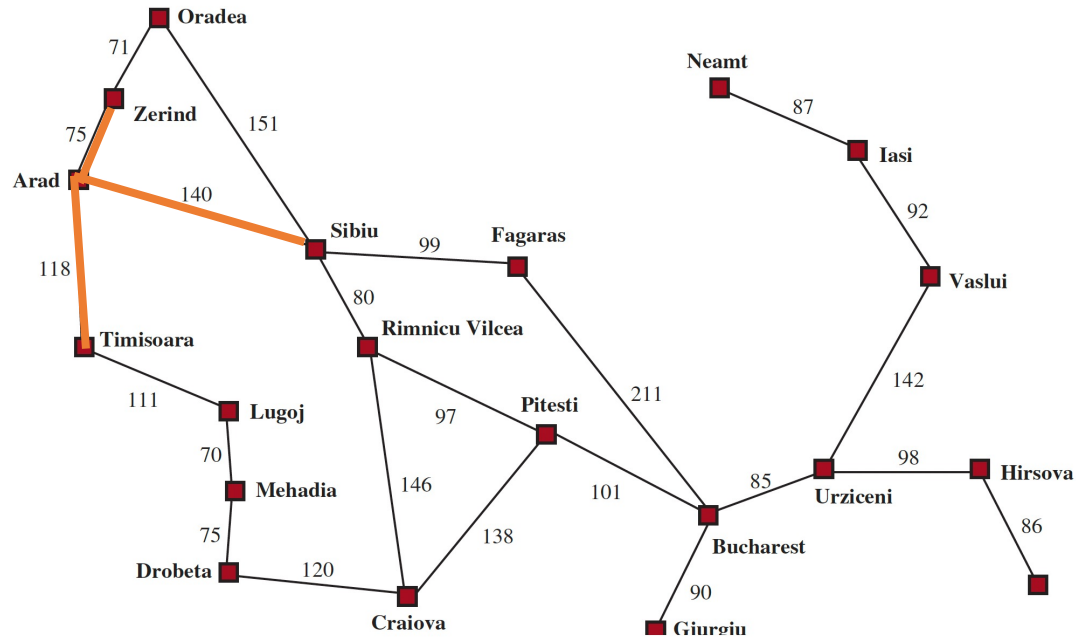
Near Side		Far Side
MMMCCCB		---
MMMC		CCB
MMMCCB		C
MMM		CCCB
MMMCB		CC
MC		MMCCB
MMCCB		MC
CC		MMMCB
CCCB		MMM
C		MMMCCB
CCB		MMMC
---		MMMCCCB



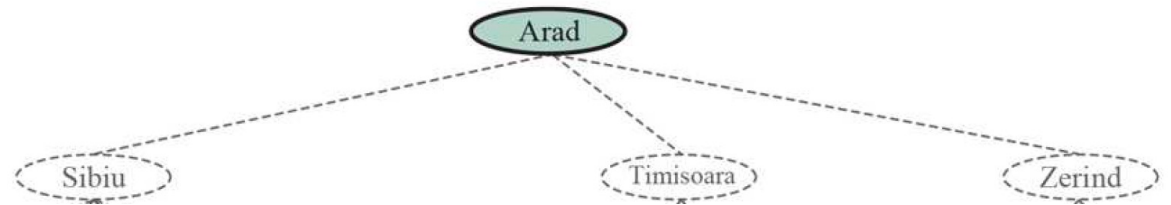
Search algorithms

- A search algorithm takes a search problem as input and returns a solution, or an indication of failure.
- **Search tree:** Each **node** in the search tree corresponds to a state in the state space and the **edges** in the search tree correspond to actions. The **root** of the tree corresponds to the initial state of the problem.
- The search tree describes paths between states, reaching towards the goal.
- The search tree may have multiple paths to (and thus multiple nodes for) any given state, but each node in the tree has a unique path back to the root (as in all trees).

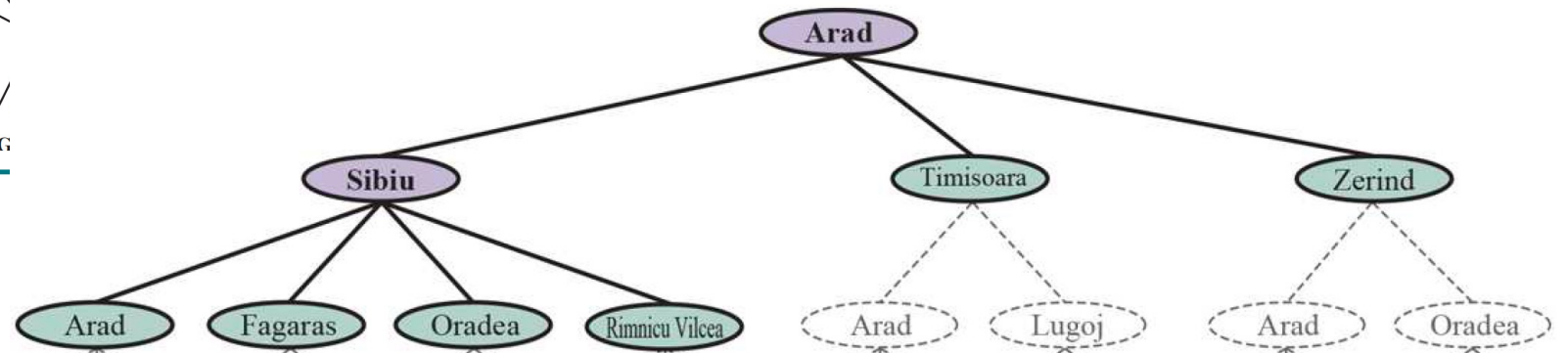
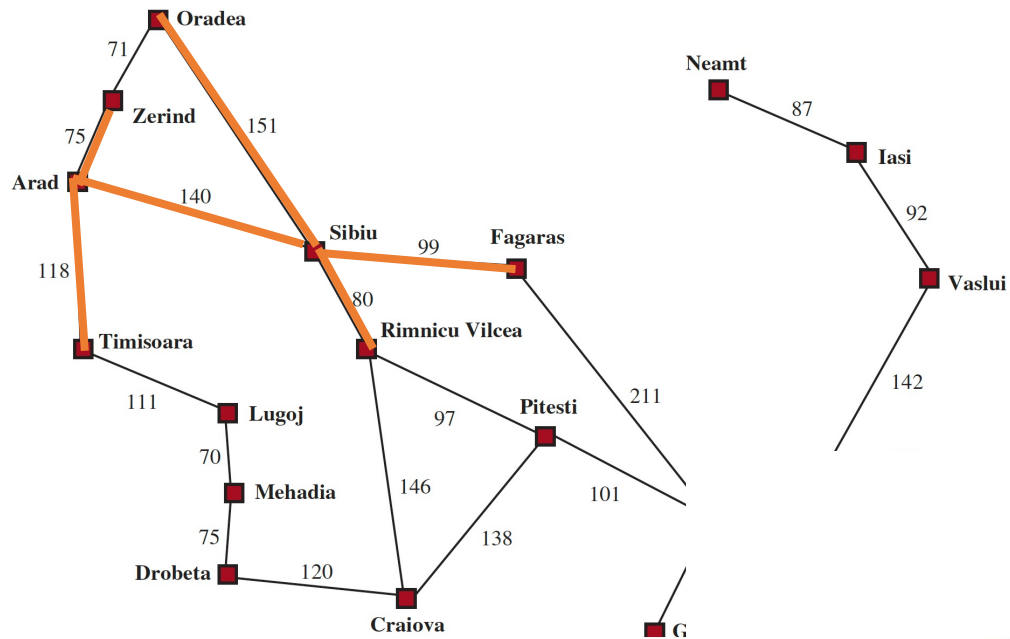
Search tree



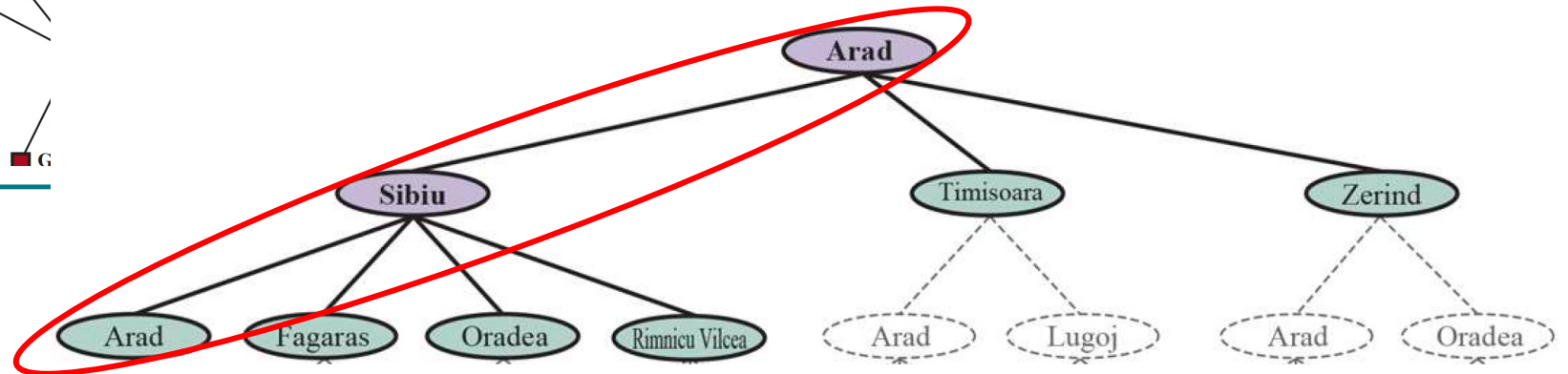
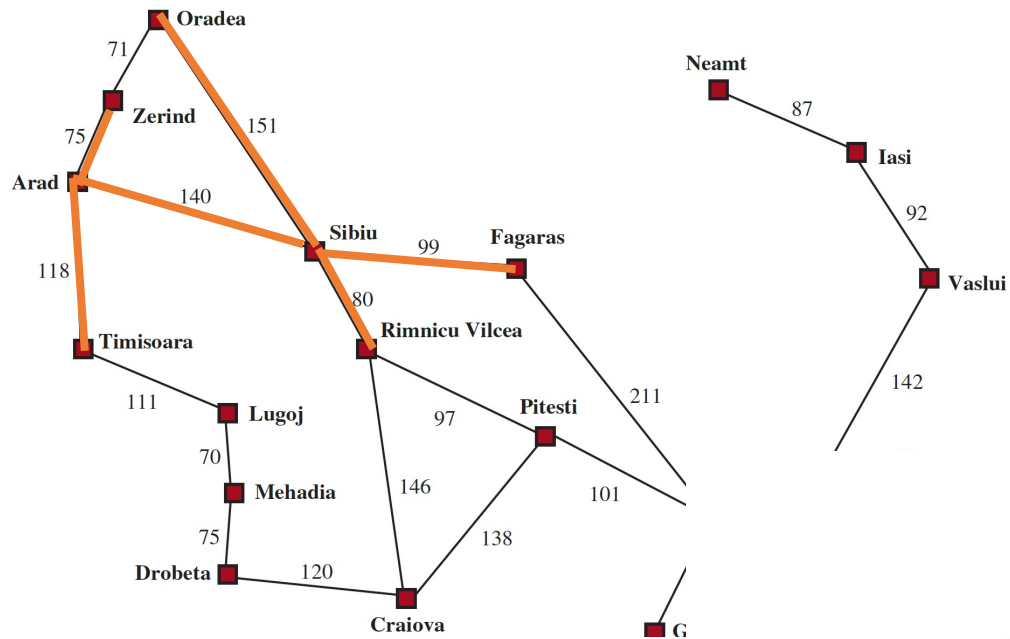
Children/Successors



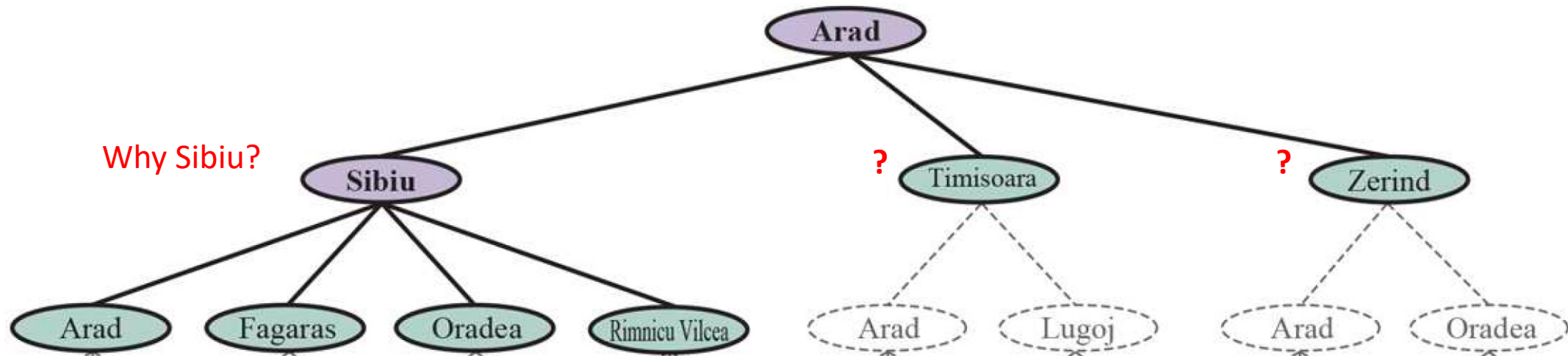
Search tree



Search tree

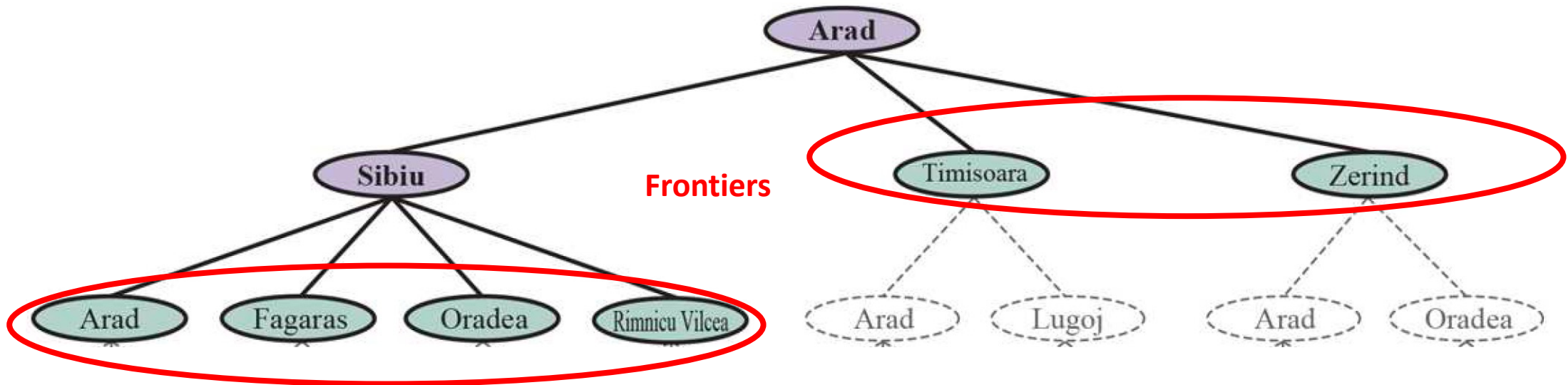


Search tree



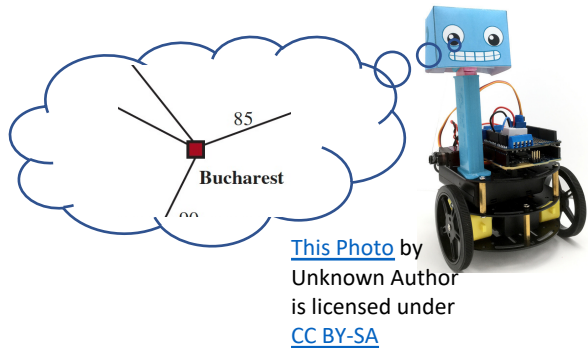
Search tree

Frontier separates two regions of the state-space graph: an interior region where every state has been expanded, and an exterior region of states that have not yet been reached.

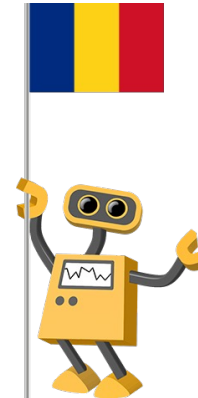


Uninformed search

- An uninformed search algorithm is given no clue about how close a state is to the goal(s).



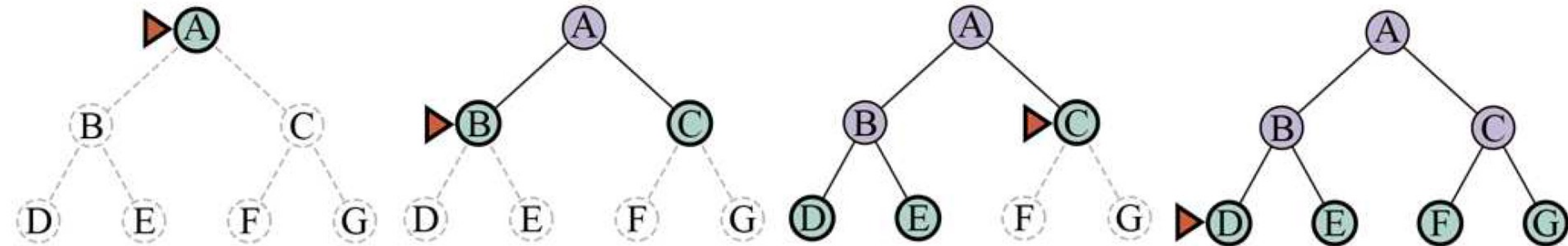
Has no clue whether going to Zerind or Sibiu is a better first step.



An informed agent knows the location of each city. It knows that Sibiu is much closer to Bucharest and thus more likely to be on the shortest path.

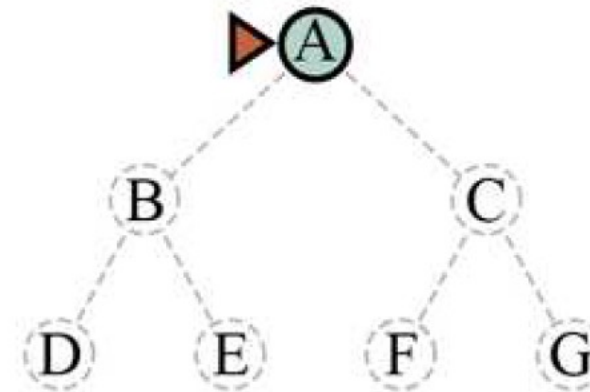
Breadth-first search

- When all actions have the same cost, an appropriate strategy is **breadth-first search**.
 - The root node is expanded first.
 - Then all the successors of the root node are expanded next.
 - Then their successors, and so on.



Breadth-first search

```
function BREADTH-FIRST-SEARCH(problem) returns a solution node or failure  
  node  $\leftarrow$  NODE(problem.INITIAL)  
  if problem.IS-GOAL(node.STATE) then return node  
  frontier  $\leftarrow$  a FIFO queue, with node as an element  
  reached  $\leftarrow$  {problem.INITIAL}  
  while not IS-EMPTY(frontier) do  
    node  $\leftarrow$  POP(frontier)  
    for each child in EXPAND(problem, node) do  
      s  $\leftarrow$  child.STATE  
      if problem.IS-GOAL(s) then return child  
      if s is not in reached then  
        add s to reached  
        add child to frontier  
  return failure
```



Breadth-first search

function BREADTH-FIRST-SEARCH(*problem*) **returns** a solution node or *failure*

node \leftarrow NODE(*problem*.INITIAL)

if *problem*.IS-GOAL(*node*.STATE) **then return** *node*

frontier \leftarrow a FIFO queue, with *node* as an element

reached \leftarrow {*problem*.INITIAL}

while not IS-EMPTY(*frontier*) **do**

node \leftarrow POP(*frontier*)

for each *child* **in** EXPAND(*problem*, *node*) **do**

s \leftarrow *child*.STATE

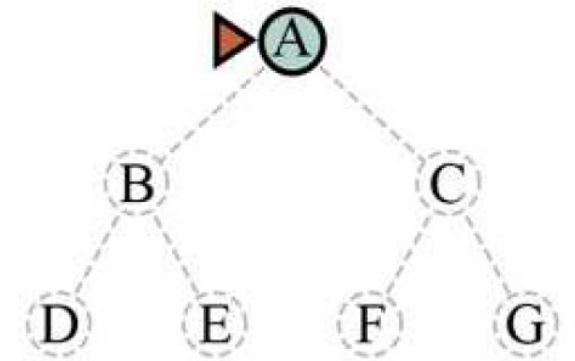
if *problem*.IS-GOAL(*s*) **then return** *child*

if *s* is not in *reached* **then**

add *s* to *reached*

add *child* to *frontier*

return *failure*



A



Breadth-first search

function BREADTH-FIRST-SEARCH(*problem*) **returns** a solution node or *failure*

node \leftarrow NODE(*problem*.INITIAL)

if *problem*.IS-GOAL(*node*.STATE) **then return** *node*

frontier \leftarrow a FIFO queue, with *node* as an element

reached \leftarrow {*problem*.INITIAL}

while not IS-EMPTY(*frontier*) **do**

node \leftarrow POP(*frontier*)

for each *child* **in** EXPAND(*problem*, *node*) **do**

s \leftarrow *child*.STATE

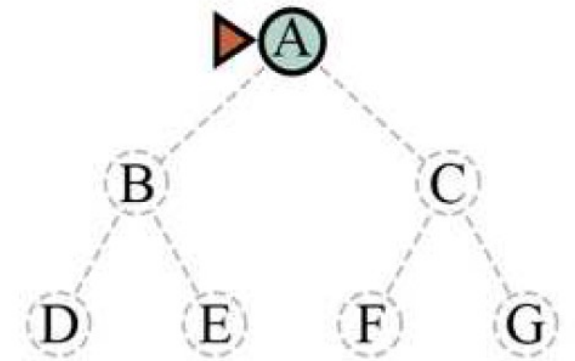
if *problem*.IS-GOAL(*s*) **then return** *child*

if *s* is not in *reached* **then**

add *s* to *reached*

add *child* to *frontier*

return *failure*



A B



Breadth-first search

function BREADTH-FIRST-SEARCH(*problem*) **returns** a solution node or *failure*

node \leftarrow NODE(*problem*.INITIAL)

if *problem*.IS-GOAL(*node*.STATE) **then return** *node*

frontier \leftarrow a FIFO queue, with *node* as an element

reached \leftarrow {*problem*.INITIAL}

while not IS-EMPTY(*frontier*) **do**

node \leftarrow POP(*frontier*)

for each *child* **in** EXPAND(*problem*, *node*) **do**

s \leftarrow *child*.STATE

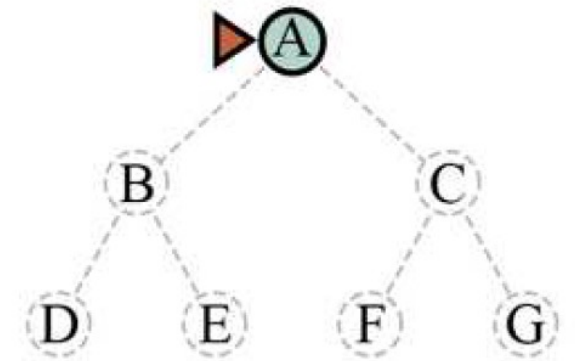
if *problem*.IS-GOAL(*s*) **then return** *child*

if *s* is not in *reached* **then**

add *s* to *reached*

add *child* to *frontier*

return *failure*



A B C



Breadth-first search

function BREADTH-FIRST-SEARCH(*problem*) **returns** a solution node or *failure*

node \leftarrow NODE(*problem*.INITIAL)

if *problem*.IS-GOAL(*node*.STATE) **then return** *node*

frontier \leftarrow a FIFO queue, with *node* as an element

reached \leftarrow {*problem*.INITIAL}

while not IS-EMPTY(*frontier*) **do**

node \leftarrow POP(*frontier*)

for each *child* **in** EXPAND(*problem*, *node*) **do**

s \leftarrow *child*.STATE

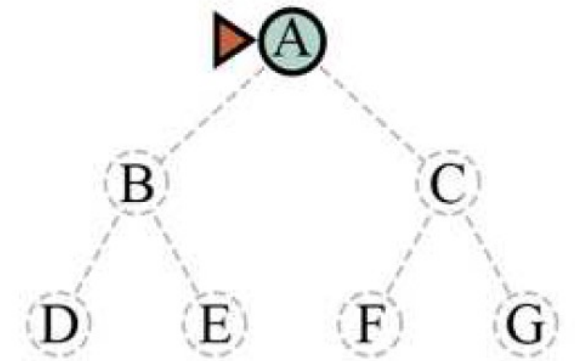
if *problem*.IS-GOAL(*s*) **then return** *child*

if *s* is not in *reached* **then**

add *s* to *reached*

add *child* to *frontier*

return *failure*



A B C D

G F E



Breadth-first search

function BREADTH-FIRST-SEARCH(*problem*) **returns** a solution node or *failure*

$$node \leftarrow \text{NODE}(\text{problem.INITIAL})$$

if *problem*.IS-GOAL(*node*.STATE) **then return** *node*

frontier \leftarrow a FIFO queue, with *node* as an element

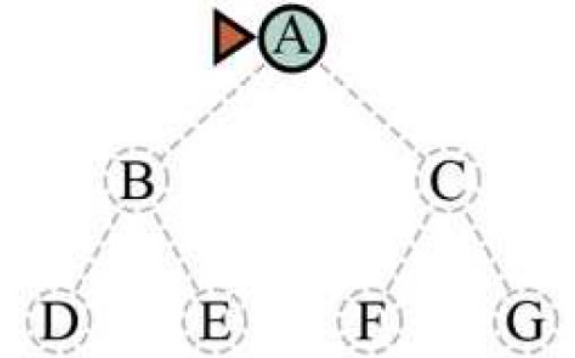
$$reached \leftarrow \{problem.INITIAL\}$$
while not IS-EMPTY(*frontier*) do
$$node \leftarrow \text{POP}(frontier)$$
for each *child* **in** EXPAND(*problem*, *node*) **do** $s \leftarrow child.STATE$

if *problem*.IS-GOAL(*s*) **then return** *child*

if s is not in *reached* **then**

add s to *reached*

add *child* to *frontier*

return *failure*

A B C D E



Breadth-first search

function BREADTH-FIRST-SEARCH(*problem*) **returns** a solution node or *failure*

$$node \leftarrow \text{NODE}(\text{problem.INITIAL})$$

if *problem.IS-GOAL*(*node.STATE*) **then return** *node*

frontier \leftarrow a FIFO queue, with *node* as an element

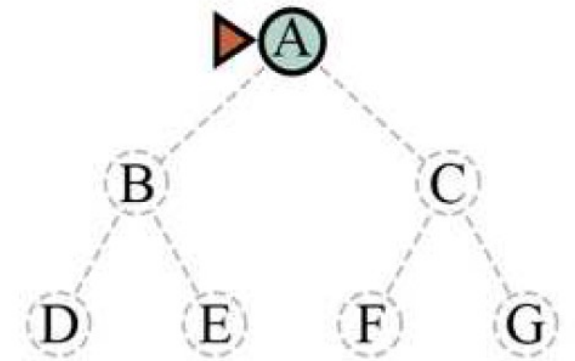
$$reached \leftarrow \{problem.INITIAL\}$$
while not IS-EMPTY(*frontier*) do
$$node \leftarrow \text{POP}(frontier)$$
for each *child* **in** EXPAND(*problem*, *node*) **do** $s \leftarrow child.STATE$

if *problem*.IS-GOAL(*s*) **then return** *child*

if s is not in *reached* **then**

add s to *reached*

add *child* to *frontier*

return *failure*

A B C D E F



Breadth-first search

function BREADTH-FIRST-SEARCH(*problem*) **returns** a solution node or *failure*

node \leftarrow NODE(*problem*.INITIAL)

if *problem*.IS-GOAL(*node*.STATE) **then return** *node*

frontier \leftarrow a FIFO queue, with *node* as an element

reached \leftarrow {*problem*.INITIAL}

while not IS-EMPTY(*frontier*) **do**

node \leftarrow POP(*frontier*)

for each *child* **in** EXPAND(*problem*, *node*) **do**

s \leftarrow *child*.STATE

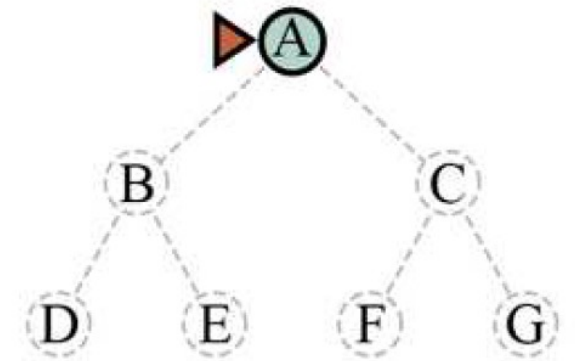
if *problem*.IS-GOAL(*s*) **then return** *child*

if *s* is not in *reached* **then**

add *s* to *reached*

add *child* to *frontier*

return *failure*



A B C D E F G



Breadth-first search

- Breadth-first search always finds a solution with a minimal number of actions because when it is generating nodes at depth d , it has already generated all the nodes at depth $d-1$, so if one of them were a solution, it would have been found.
- Is it optimal?
 - It is cost optimal if all actions have the same cost.

How to measure problem-solving performance?

- We can evaluate an algorithm's performance in four ways:
 1. **Completeness:** Is the algorithm guaranteed to find a solution when there is one, and to correctly report failure when there is not?
 2. **Cost optimality:** Does it find a solution with the lowest path cost of all solutions?
 3. **Time complexity:** How long does it take to find a solution? This can be measured in seconds, or more abstractly by the number of states and actions considered.
 4. **Space complexity:** How much memory is needed to perform the search?

Breadth-first search performance



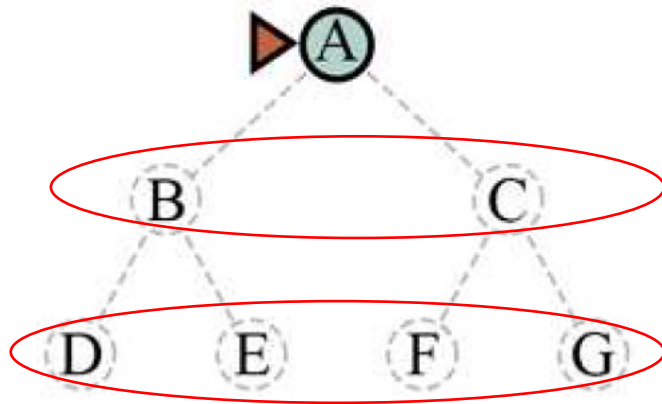
1. **Completeness:**

2. **Cost optimality:** Only if all actions have the same cost.

3. **Time complexity:**

4. **Space complexity:**

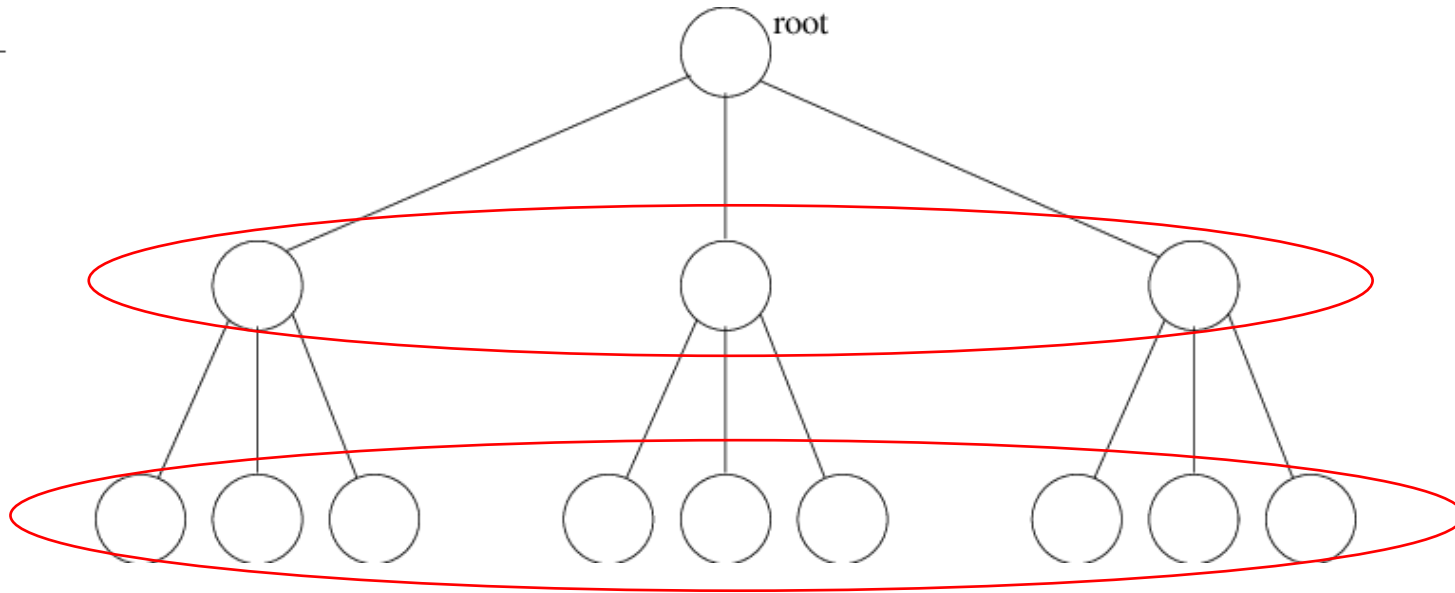
Breadth-first search performance



2 nodes

2^2 nodes

Breadth-first search performance



3 nodes

3^2 nodes

$$b + b^2 + b^3 + \dots b^d = O(b^d)$$

Exponential bounds are scary!



In general, exponential complexity search problems cannot be solved by uninformed search for any but the smallest instances.

Dijkstra's algorithm or uniform-cost search

- What if actions have different costs?
- While breadth-first search spreads out in waves of uniform depth—first depth 1, then depth 2, and so on—uniform-cost search spreads out in waves of uniform path-cost.

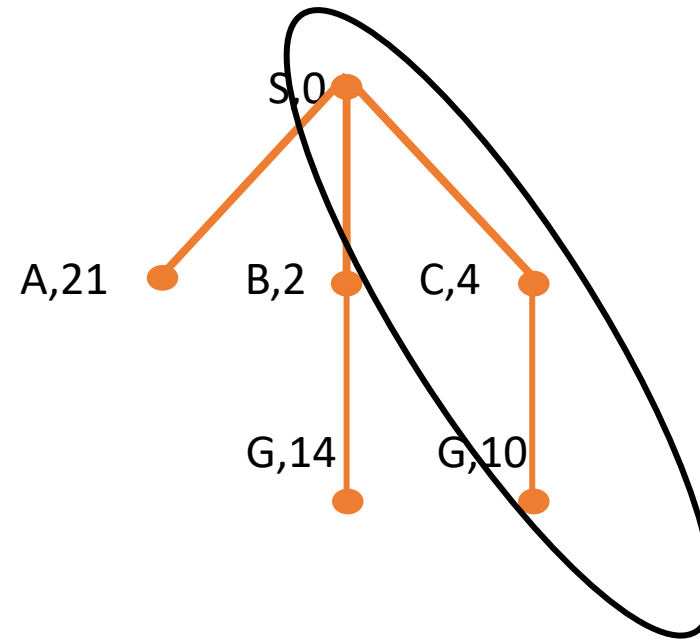
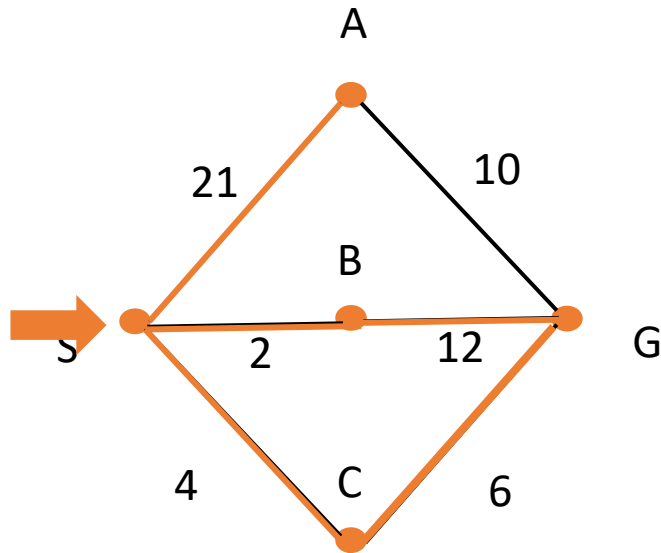
function UNIFORM-COST-SEARCH(*problem*) **returns** a solution node, or *failure*
return BEST-FIRST-SEARCH(*problem*, PATH-COST)

Best-first search

Best-first search is a very general approach in which we choose a node n , with minimum value of some evaluation function $f(n)$.

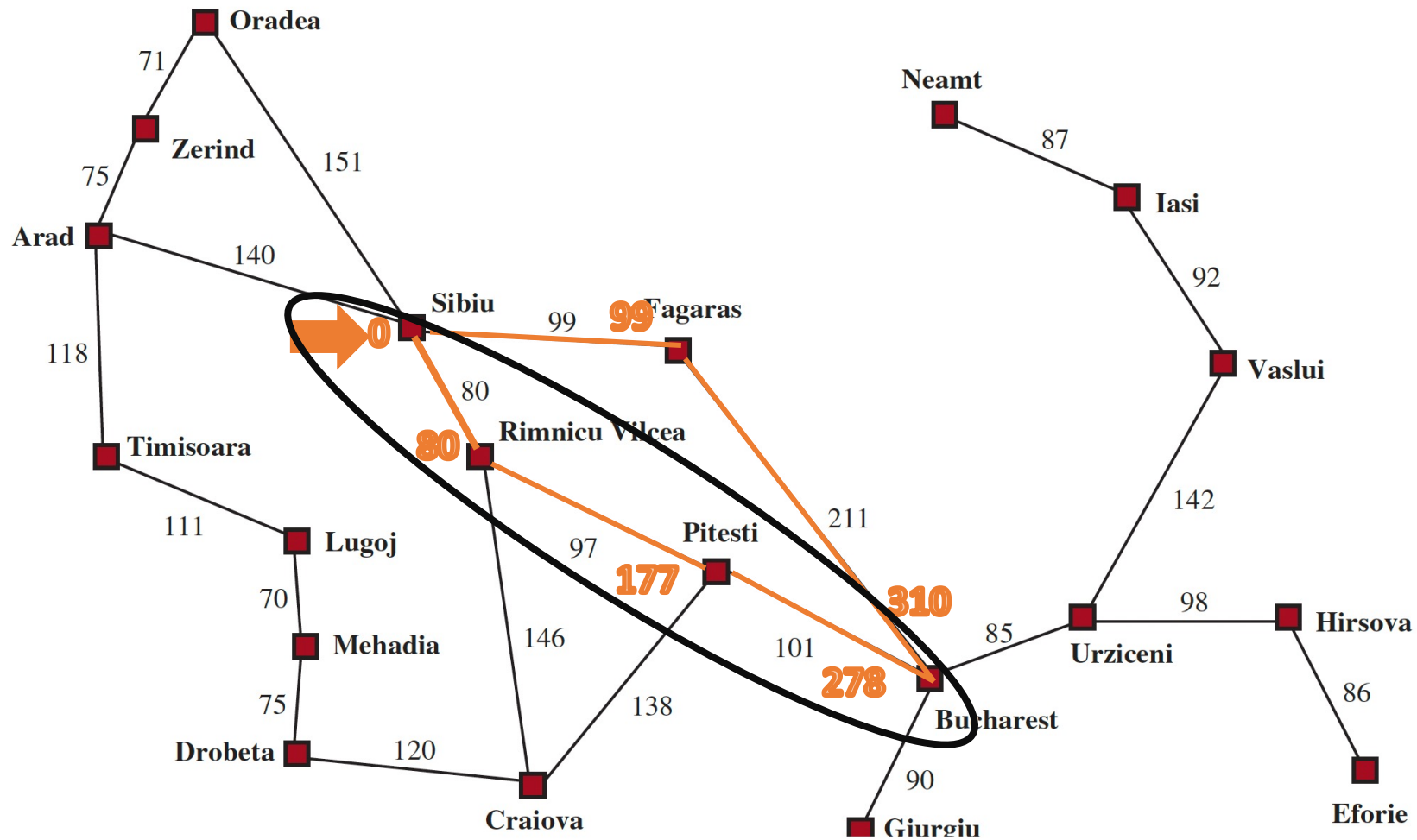
- On each iteration, we choose a node on the frontier with minimum $f(n)$ value.
- Return it if its state is a goal state. Otherwise,
- Apply *Expand* to generate child nodes.
- Each child node is added to the frontier if it has not been reached before, or is re-added if it is now being reached with a path that has a lower path cost than any previous path.
- The algorithm returns either an indication of failure, or a node that represents a path to a goal.

Uniform-cost search



Note that if we had checked for a goal upon generating a node rather than when expanding the lowest-cost node, then we would have returned a higher-cost path.

Uniform-cost search



Uniform-cost search performance

1. **Completeness:**



2. **Cost optimality:**



3. **Time complexity:**

4. **Space complexity:**



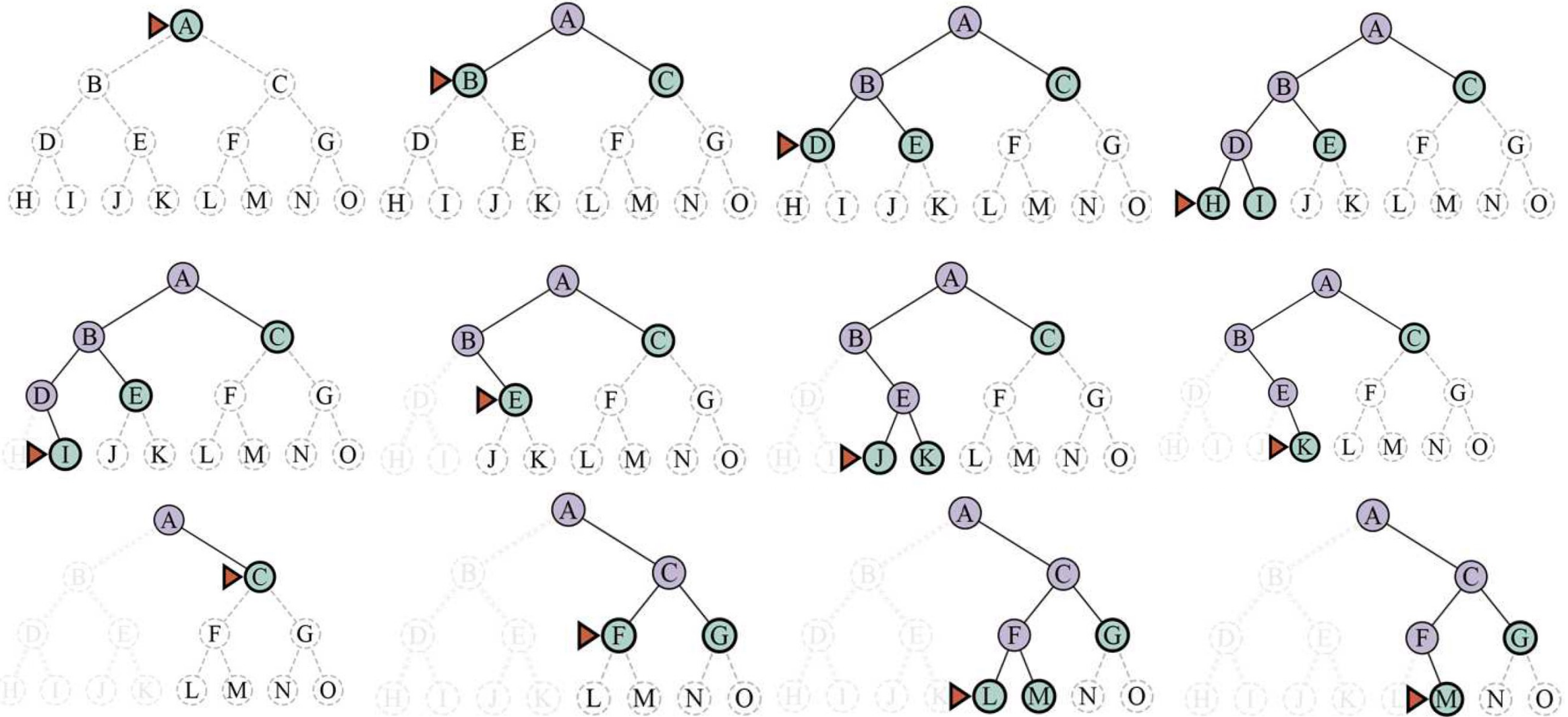
BFS: $O(b^d)$

- C^* : the cost of the optimal solution
- ϵ : lower bound on the cost of each action, with $\epsilon > 0$
- Effective depth : $1 + \left\lceil \frac{C^*}{\epsilon} \right\rceil$
- Worst case time and space complexity: $O(b^{1 + \left\lceil \frac{C^*}{\epsilon} \right\rceil})$

Depth-first search

- Depth-first search always expands the deepest node in the frontier first.
 - Search proceeds immediately to the deepest level of the search tree where the nodes have no successors.
 - The search then “backs up” to the next deepest node that still has unexpanded successors.

Progress of a depth-first search on a binary tree from start state A to goal M



Depth-first search performance

1. **Completeness:** Only for finite and acyclic state spaces

2. **Cost optimality:**

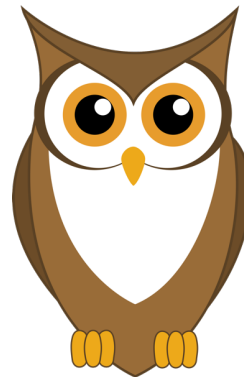


3. **Time complexity:**

4. **Space complexity:**




Why would anyone consider using depth-first search, then?



Much smaller needs for memory.

Depth-first search performance

- 1. Completeness:** Only for finite and acyclic state spaces.
- 2. Cost optimality:** 
- 3. Time complexity:** $O(b^m)$. In the worst-case scenario, DFS creates a search tree whose depth is m (maximum depth of the tree).
- 4. Space complexity:** Only has siblings on path to root, so $O(bm)$.

Some other uniformed search algorithms

- **Depth-limited search:** adds a depth bound.
- **Iterative deepening search:** calls depth-first search with increasing depth limits until a goal is found. It is **complete** when full cycle checking is done, **optimal** for unit action costs, has time complexity comparable to breadth-first search, and has **linear space complexity**.
- **Bidirectional search:** expands two frontiers, one around the initial state and one around the goal, stopping when the two frontiers meet.

Three kinds of queues used in search algorithms

- **Priority queue:** first pops the node with the minimum cost according to some evaluation function, f .
- **FIFO queue:** first pops the node that was added to the queue first.
- **LIFO queue (stack):** pops first the most recently added node;
- The reached states can be stored as a lookup table (e.g. a hash table) where each key is a state and each value is the node for that state.

Recap

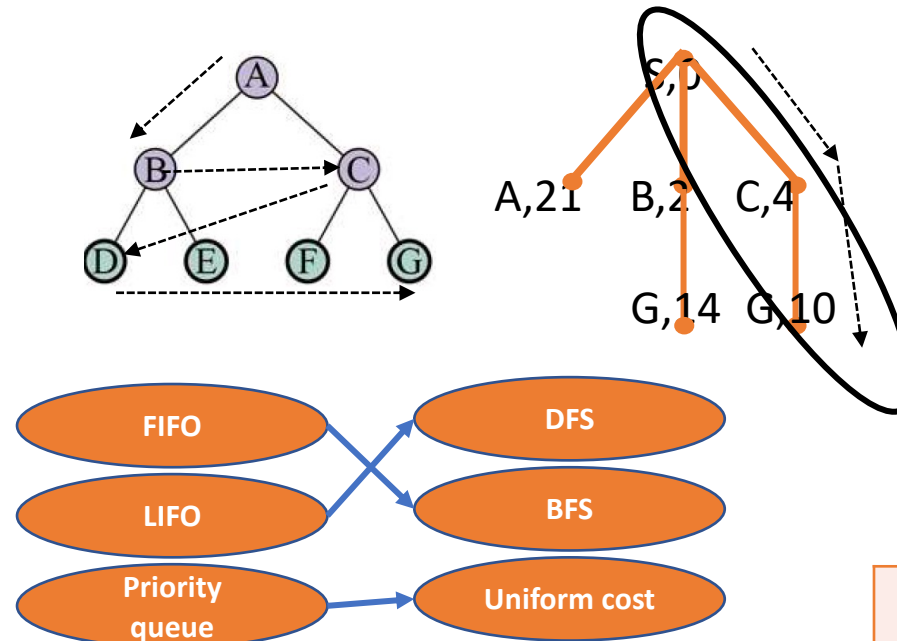
- How to define a search problem?

State space
Initial state
Goal state(s)
Actions
Transition model
Action cost function

- Uninformed search

- Breadth-first search
- Uniform cost search
- Depth-first search

- Search data structures



- How to measure problem-solving performance?

Completeness
Cost optimality
Time complexity
Space complexity