WPI

Artificial Intelligence
CS 534

Week 2

Specifying the task environment

Task environments, are the "problems" to which rational agents are the "solutions."

Task environment is defined using PEAS descriptors:

- Performance measure
- Environment
- Actuators
- Sensors

Example: Automated taxi

Performance measures: safety, destination, profits, legality, comfort, *etc*.

Environment: US streets/freeways, traffic, pedestrians, customers, weather, *etc*.

Actuators: steering, accelerator, brake, horn, speakers, display, *etc*.

Sensors: cameras, accelerometers, gauges, engine sensors, keyboard, GPS, *etc*.

Environment types

Fully vs. partially observable: Agent's sensors give it access to the complete (vs. incomplete) state of the environment at each time point

Single agent vs. multiagent: An agent operating by itself in an environment (vs. multiple agents in the same environment

Deterministic vs. stochastic: The next state of the environment is completely (vs. partially) determined by the current state and the action executed by the agent. If the environment is deterministic except for the actions of other agents, then the environment is called strategic

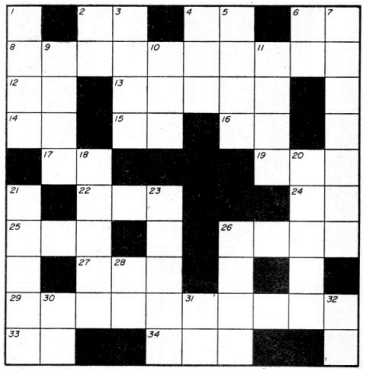
Episodic vs. sequential: Agent's experience is divided (vs. not divided) into atomic 'episodes' (each episode consists of the agent perceiving and then performing a single action). The choice of action in each episode depends only on the episode itself

Environment types

Static vs. dynamic: The environment is unchanged while an agent is deliberating. The environment is semi-dynamic if the environment itself does not change with the passage of time but the agent's performance score does

Discrete vs. continuous: The way the time is handled: *E.g.*, a limited number of distinct, clearly defined percepts and actions

Task environments: Examples

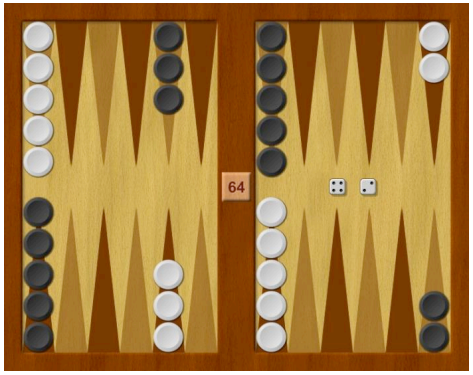


crossword

chess with timer

poker

autonomous taxi



backgammon

medical diagnostics

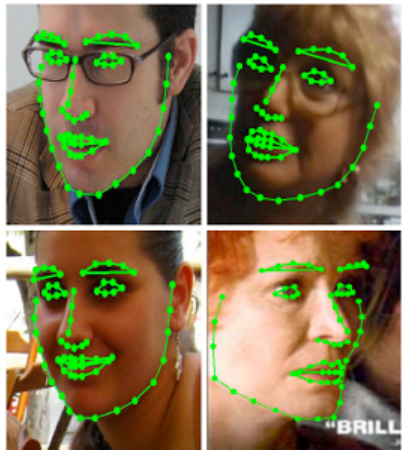


image processing

interactive English tutor

Task environments: Examples

Task Environment	Obser-vable	Agent	Determ.	Episodic	Static	Discrete
Crossword	Fully	S	Determ.	Sequen.	Static	Discrete
Chess w. clock	Fully	M	Determ.	Sequen.	Semi	Discrete
Poker	Partial.	M	Stoch.	Sequen.	Static	Discrete
Backgammon	Fully	M	Stoch.	Sequen.	Static	Discrete
Taxi driving	Partial.	M	Stoch.	Sequen.	Dynamic	Continuous
Diagnostics	Partial.	S	Stoch.	Sequen.	Dynamic	Continuous
Image analys.	Fully	S	Determ.	Episod.	Semi	Continuous
English tutor	Partial.	M	Stoch.	Sequen.	Dynamic	Discrete

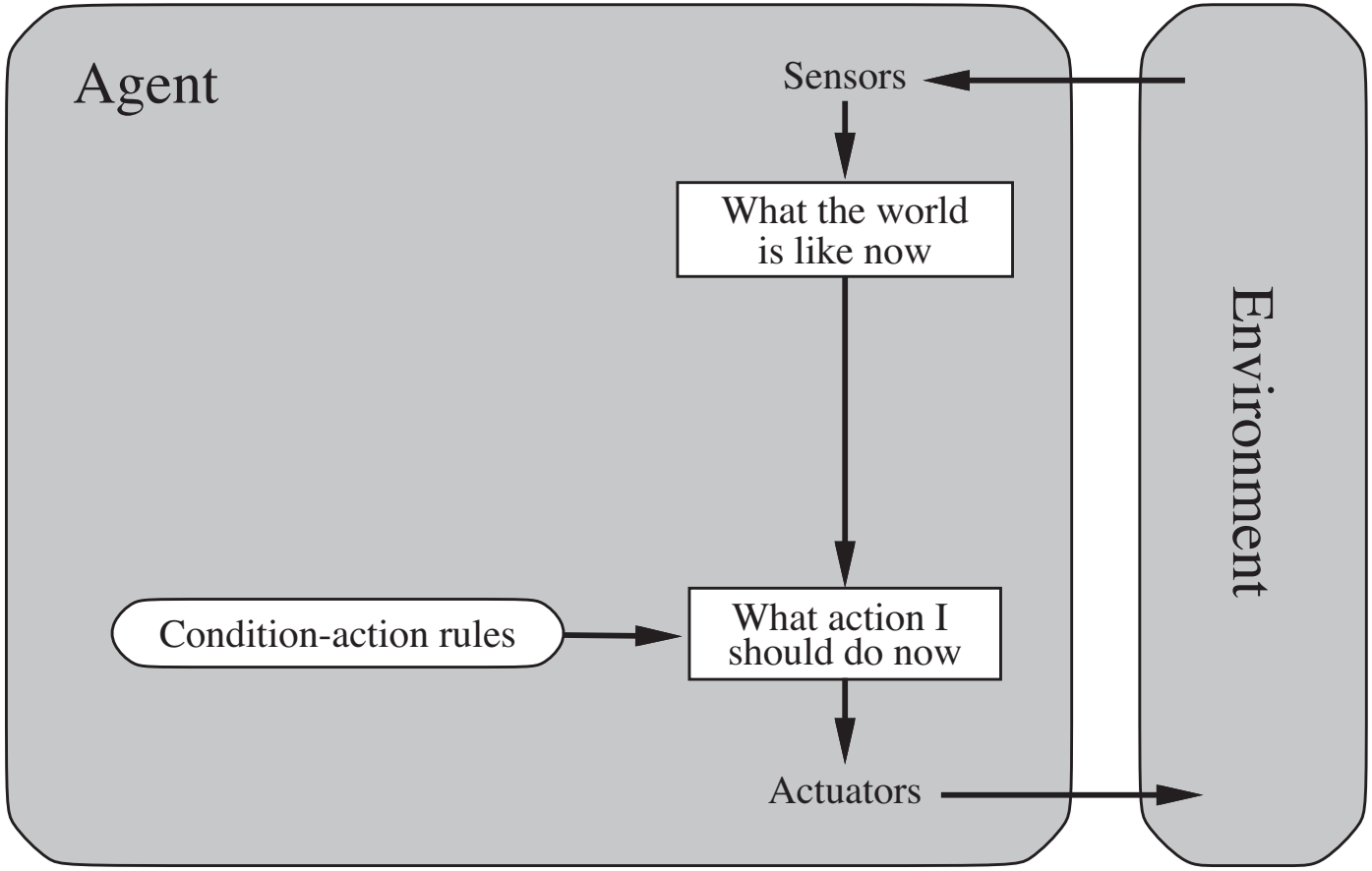
Agent programs

Four basic types of agent programs:

- Simple reflex agents
- Model-based reflex agents
- Goal-based agents
- Utility-based agents

- Note: Each of these can be turned into a learning agent (can improve the performance of their components to generate better actions)

Simple reflex agents



Example

```
function REFLEX-VACUUM-AGENT( [location,status]) returns an action
    if status = Dirty then return Suck
    else if location = A then return Right
    else if location = B then return Left
```

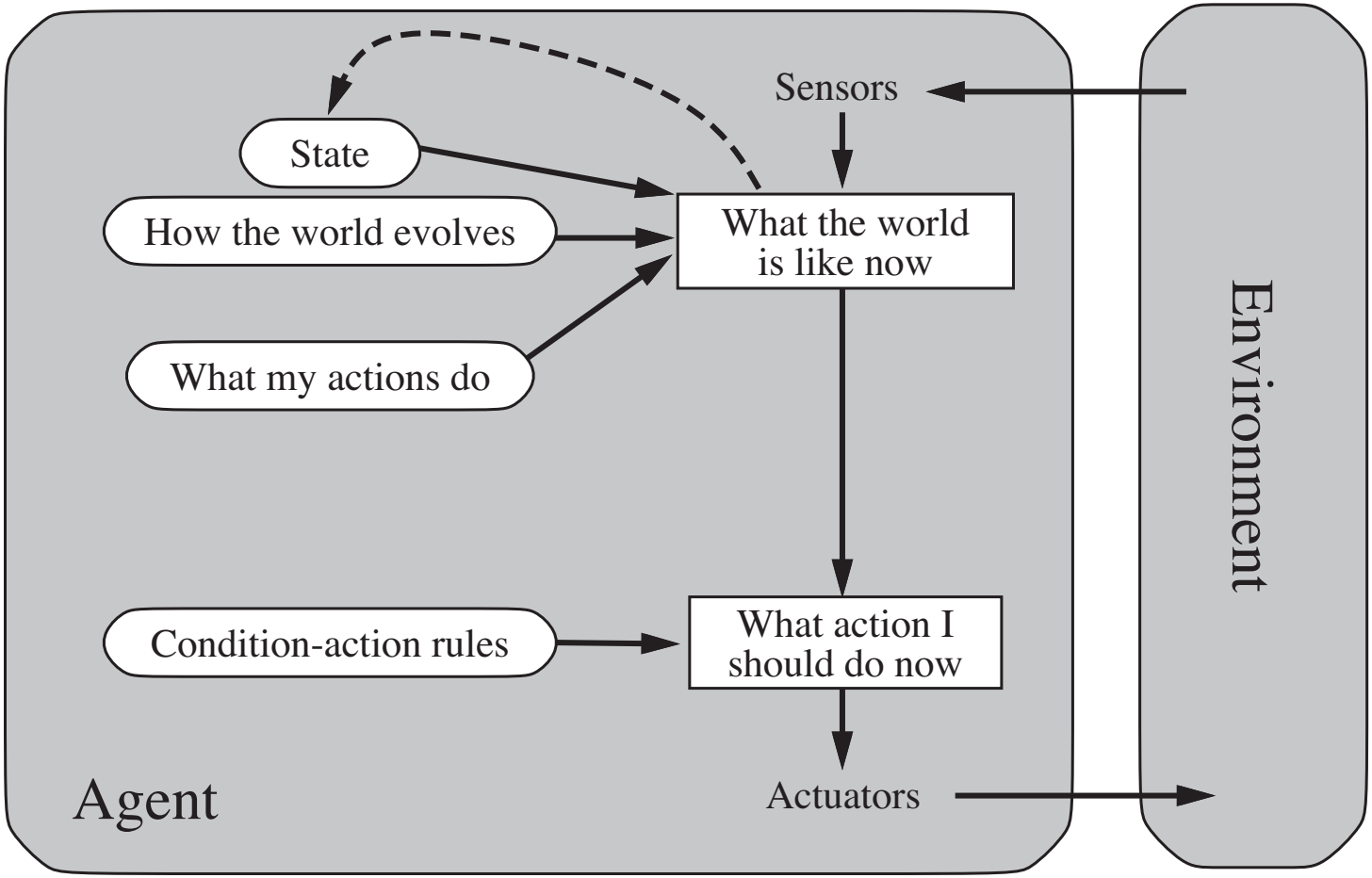
```
function SIMPLE-REFLEX-AGENT(percept) returns action
  static: rules, a set of condition-action rules

  state ← INTERPRET-INPUT(percept)
  rule ← RULE-MATCH(state, rules)
  action ← RULE-ACTION[rule]
  return action
```

- selects rule whose condition matches the current state based on percept
- acts accordingly

Simple reflex agents: Cons and pros

- Very simple
- Will work only if a correct decision can be made on the basis of only the current percept
  - *i.e.* environment is fully observable
- Otherwise — lots of problems, with potential infinite loops
- Infinite loops can be escaped from if the agent can randomize its actions

Model-based reflex agents



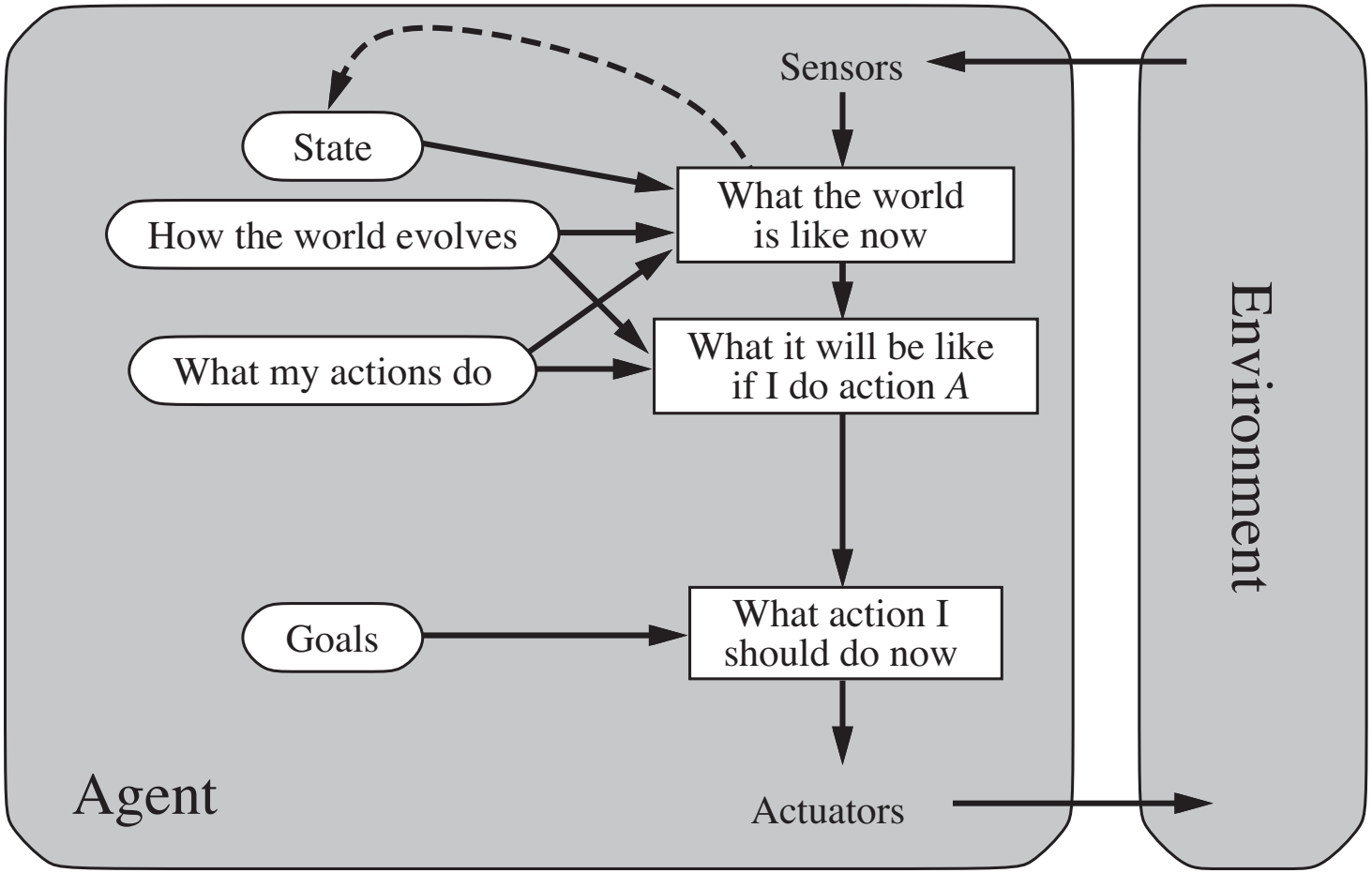
Example

function MODEL-BASED-REFLEX-AGENT(*percept*) **returns** an action
  persistent: *state*, the agent's current conception of the world state
    model, a description of how the next state depends on current state and action
    rules, a set of condition–action rules
    action, the most recent action, initially none

  state ← UPDATE-STATE(*state*, *action*, *percept*, *model*)
  rule ← RULE-MATCH(*state*, *rules*)
  action ← *rule*.ACTION
  return *action*

- Keeps track of the current state of the world using an internal model

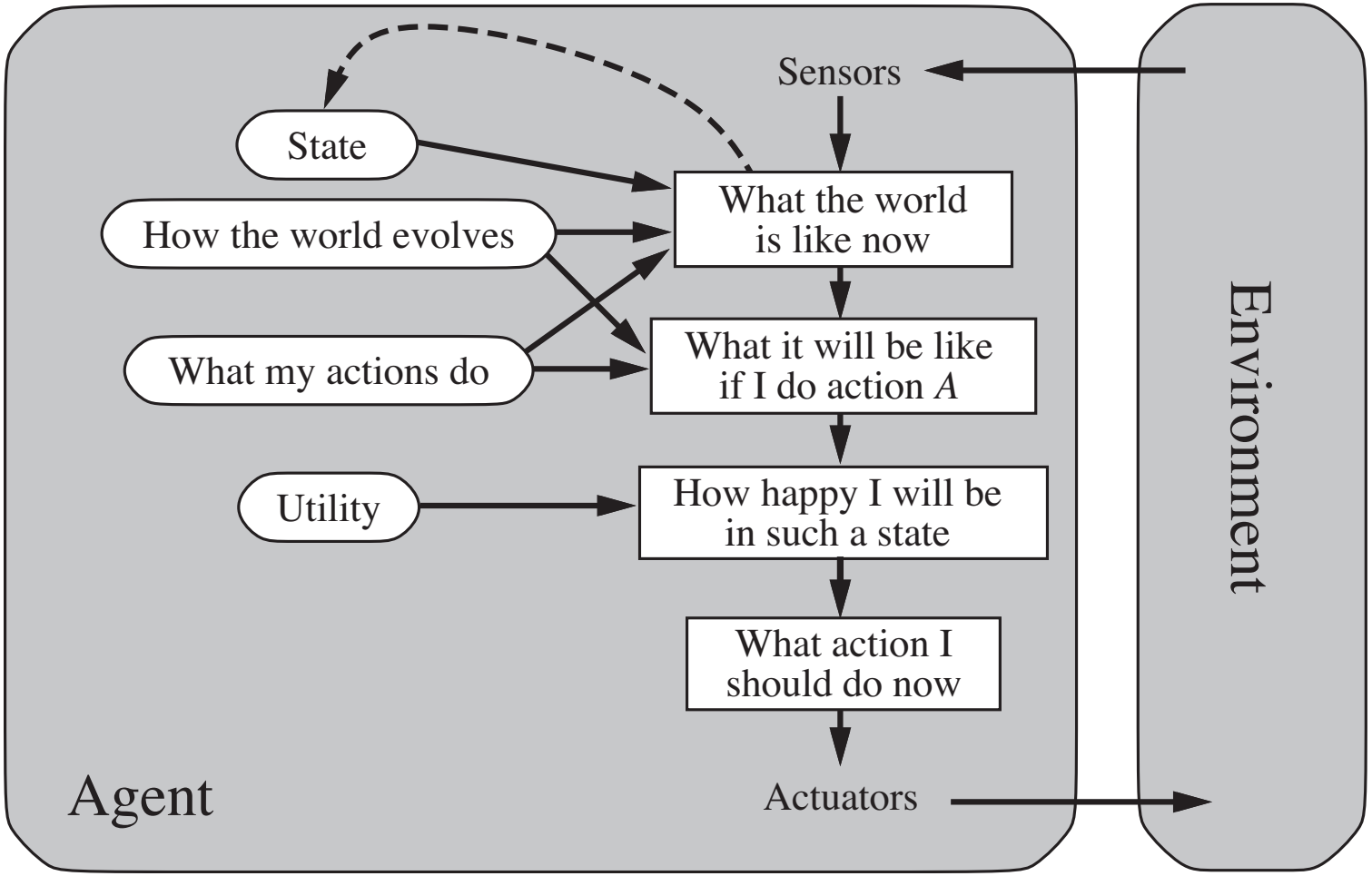
Goal-based agents



Utility-based agents

- Some action sequences are better than others: quicker, safer, more reliable, or cheaper than others, etc..
- Idea: introduce a utility function, an "internalization" of the performance measure. If the internal utility function and the external performance measure are in agreement, then an agent that chooses actions to maximize its utility will be rational according to the external performance measure
- Utility-based agent has many advantages in terms of flexibility and learning
- It allows to resolve the conflicting goals: e.g., speed vs. safety
- When there are several goals, none of which can be achieved with certainty, utility provides a way in which the likelihood of success can be weighed against the importance of the goals

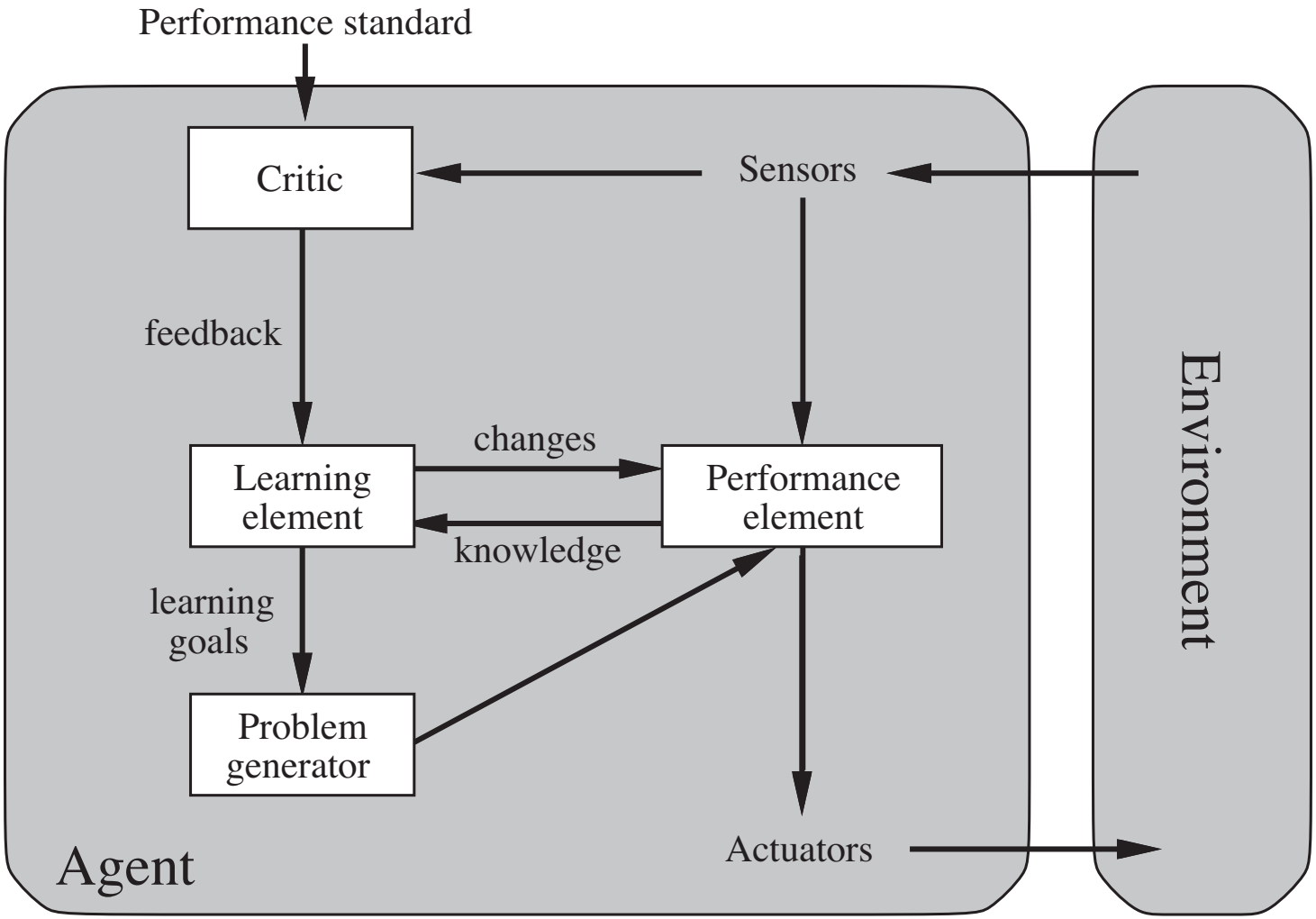
Utility-based agents



Learning agents

- Consists of four basic components: Critic, Learning Element, Performance Element, Problem Generator
- Learning element: is responsible for making improvements
- Performance element: is responsible for selecting external actions (previously, the entire agent)
- Problem generator: is responsible for suggesting actions that will lead to new and informative experiences

Learning agents



Summary

- Agents interact with environments through actuators and sensors
- The agent function describes what the agent does in all circumstances
- The performance measure evaluates the environment sequence
- A perfectly rational agent maximizes expected performance
- Agent programs implement (some) agent functions
- PEAS descriptions define task environments
- Environments are categorized along several dimensions:
  - observable; deterministic; episodic; static; discrete; single-agent
- Several basic agent architectures exist:
  - simple reflex, model-based reflex, goal-based, utility-based

Unit 2. Searching. Problem Solving Agents

Let's define a problem solving agent…

- They use atomic representations: states of the world are considered as wholes, with no internal structure visible to problem solving algorithms

- Search algorithms can be:
  - Uninformed: no information about its problem rather than its definition is given
  - Informed: are provided with some guidance on here to look for solutions

Example: Travelling in Romania

- A touring holiday in the city of Arad, Romania
- Non-refundable ticket to fly out from Bucharest tomorrow
- Must reach Bucharest on time

Note: Goal formulation is therefore extremely simple!

Example: Travelling in Romania

Goal formulation: be in Bucharest

Problem formulation:

- the process of actions and states to consider given a goal
- states: Romanian cities
- actions: drive roads between cities

Solution: sequence of actions—a driving route

Example: Arad, Sibiu, Rimnicu, Pitesti, Bucharest

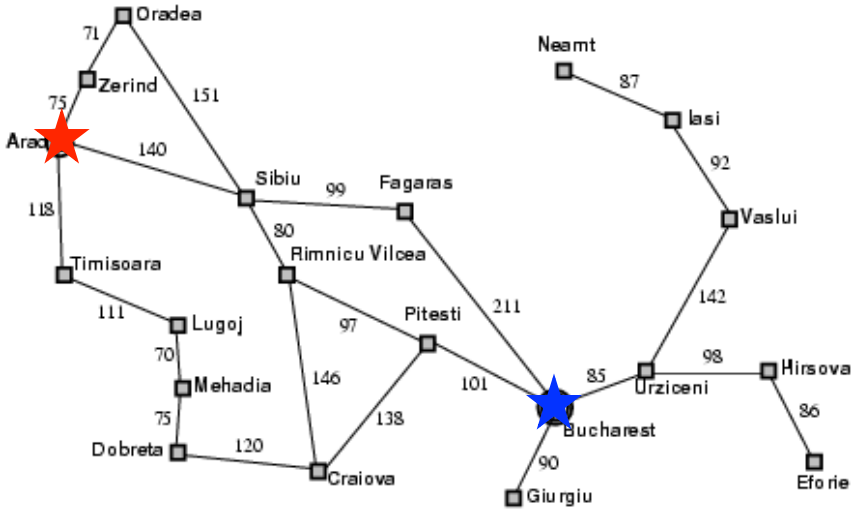
Example: Travelling in Romania

Worst case scenario: Unknown environment, i.e. no additional information

- No choice, but to try one of the actions on random

Better case scenario: Agents has a map of Romania!

- An agent with several immediate options of unknown value can decide what to do by first examining future actions
- Environment is observable
- Environment is discrete
- Environment is deterministic



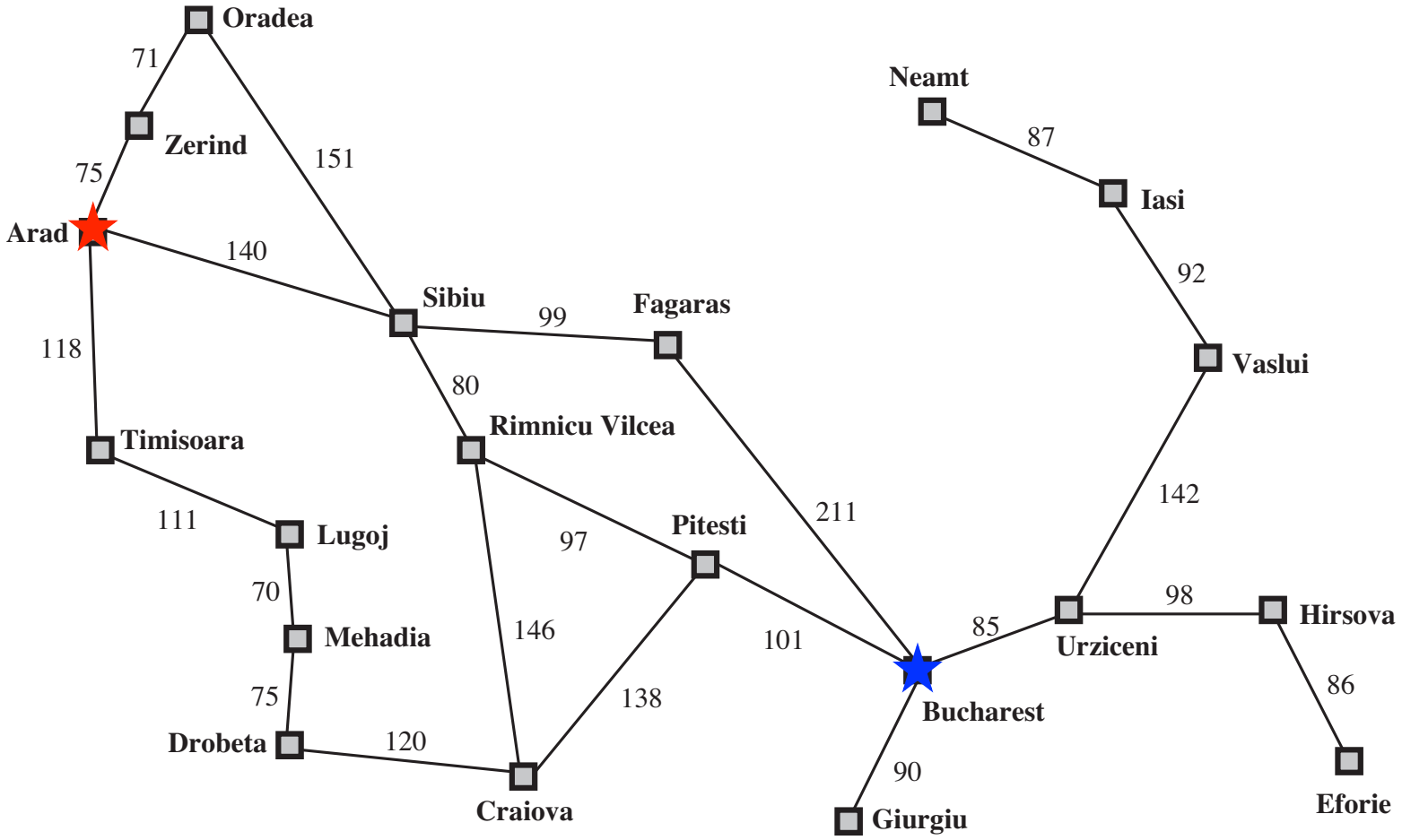
A Simple Problem Solving Agent

Problem can be defined formally by five components:

1. The initial state that the agent starts in
   - *In(Arad)*
2. Description of possible actions. For a state s, Actions(s) returns a set of actions executable from s
   - For s= *In(Arad)*, Actions(s) returns: {*Go(Sibiu), Go(Timisoara), Go(Zerind)*}
3. Description of transition model (what an action does). For a state s and action a, Results(s, a) returns the next state
   - For Results(*In(Arad), Go(Zerind)*) returns: *In(Zerind)*

Definition: Initial state + actions + transition model = State space
State space forms a directed graph, where nodes are states and edges are actions.

Travelling in Romania



A Simple Problem Solving Agent

4. The goal test: determines if a given state is the goal state
   - $\{In(Bucharest)\}$

5. A path cost function. Described as a sum of step costs for individual actions along the path: $c(s, a, s^{l})$
   - For our example the cost the the distance:
     $c(In(Arad, Go(Sibiu), In(Sibiu)) = 140$

Definitions:

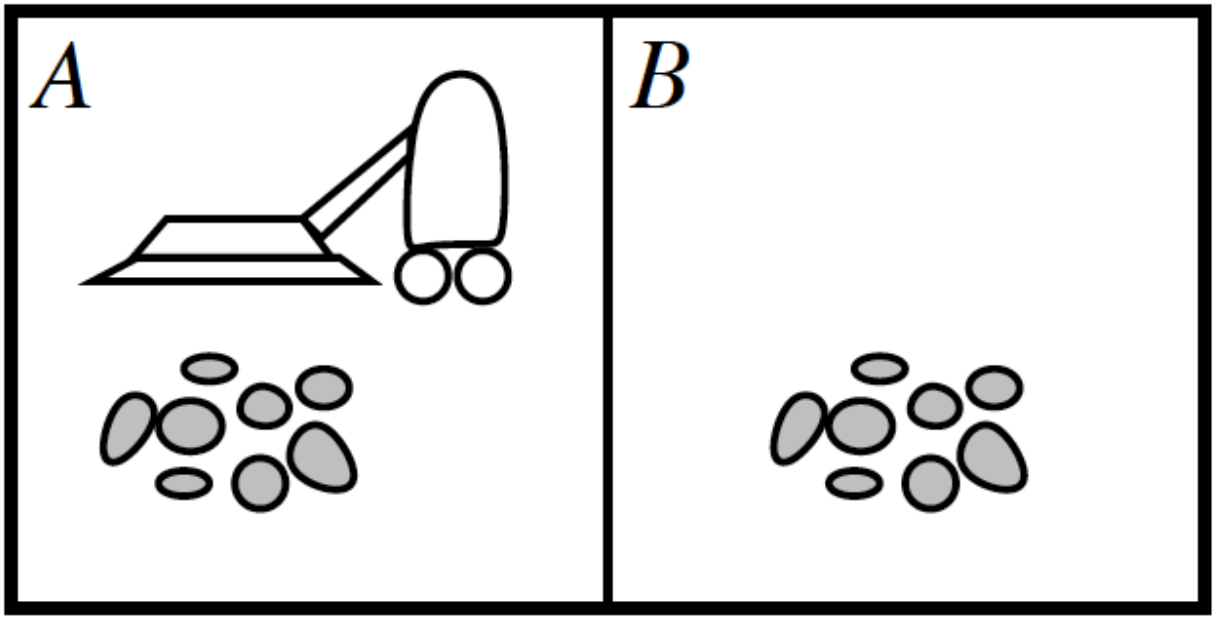
Solution: An action sequence from initial state to goal state

Optimal solution: The one with the lowest path cost among all solutions

A Simple Problem Solving Agent

```
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
        goal ← FORMULATE-GOAL(state)
        problem ← FORMULATE-PROBLEM(state, goal)
        seq ← SEARCH(problem)
    action ← RECOMMENDATION(seq, state)
    seq ← REMAINDER(seq, state)
    return action
```

Recall: Vacuum world



Percepts: location and contents, e.g., $[A, Dirty]$

Actions: $Left$, $Right$, $Suck$, $NoOp$

Example: Vacuum world

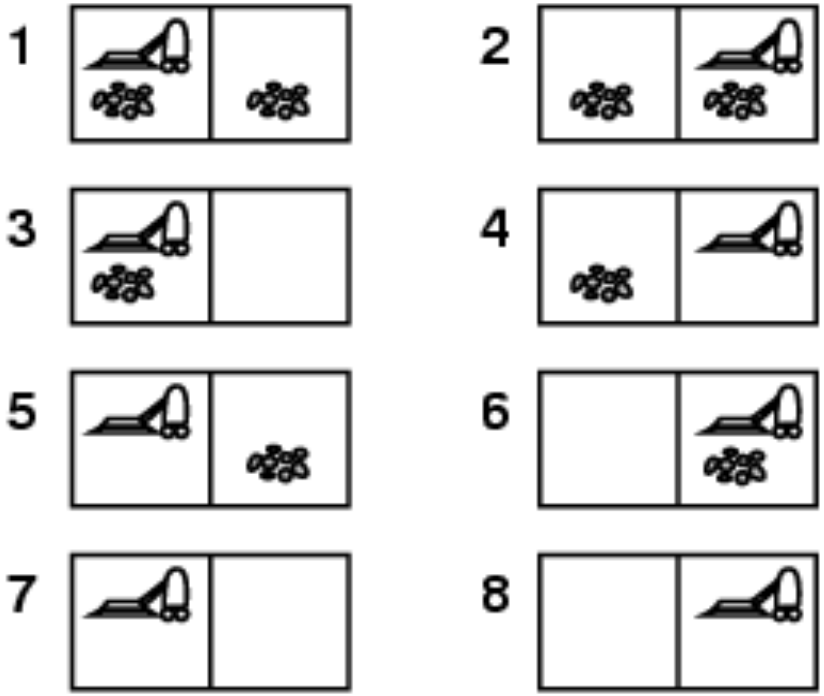
States: robot location, dirt locations → 8 possible states

- For n locations, there are $n \cdot 2^n$ states

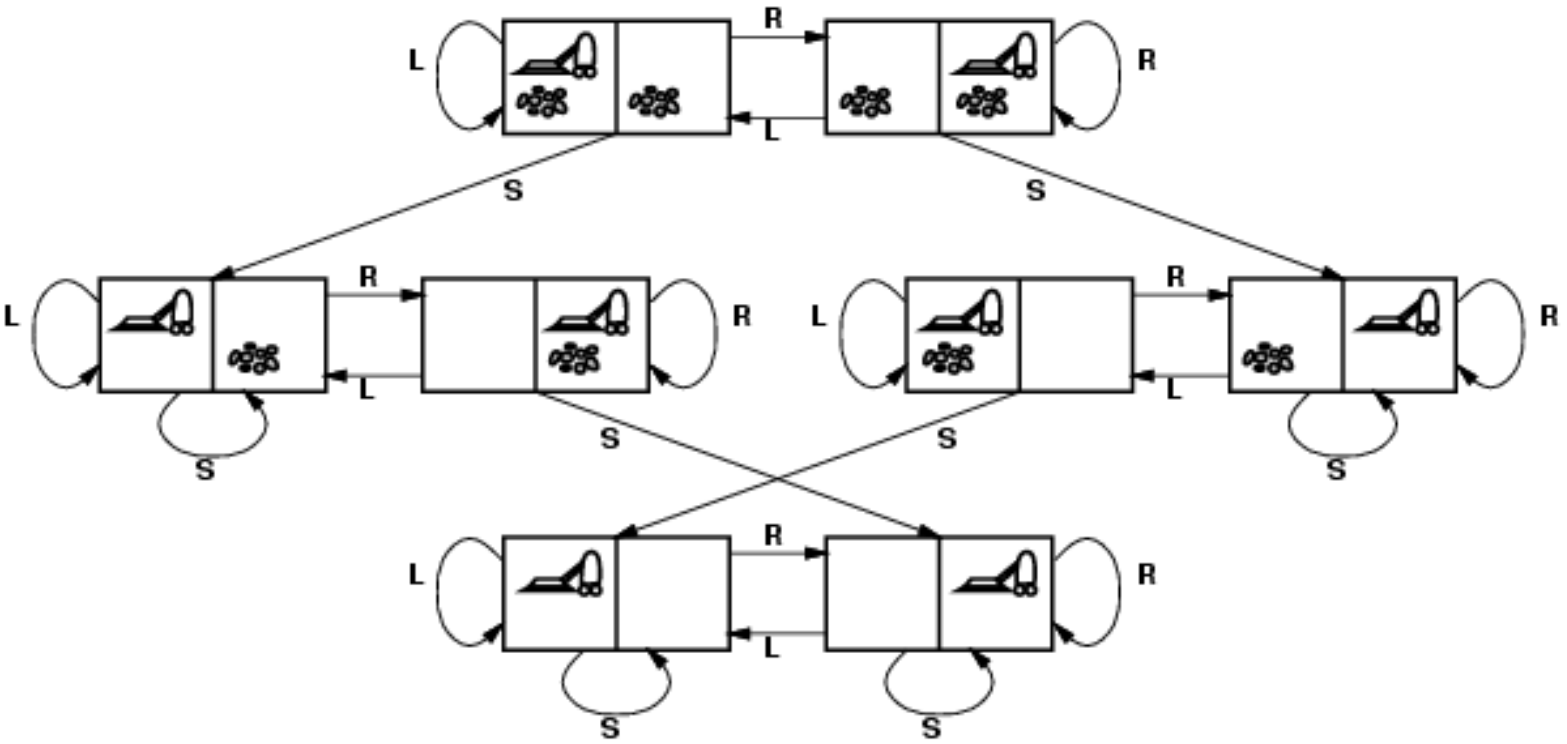
Actions: {Left, Right, Suck}

Goal test: no dirt at all locations

Path cost: $c(s, a, s') = 1$

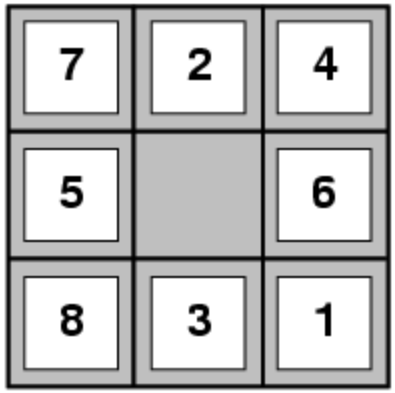


State space of Vacuum world

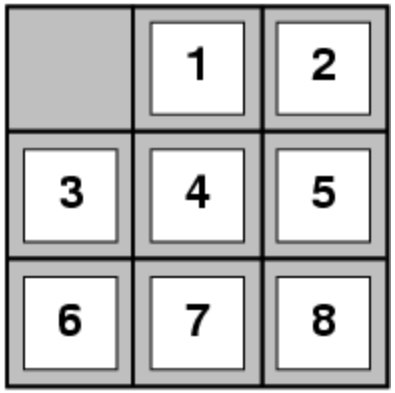


Example: 8-puzzle

- 3x3 board with 8 number titles and an empty space
- A tile adjacent to an empty space can slide into the space
- The objective is to reach the goal state



Start State



Goal State

- 8 puzzle belongs to the family of sliding-block puzzles, that are often used to test new search algorithms
- The problem is known to be NP-complete
- 8 puzzle has 9!/2=181,440 reachable states (easy to solve), while the 15-puzzle (4x4 board) has ~1.3 trillion states, and 24-puzzle (5x5 board) has 10^{25} states

Example: 8-puzzle

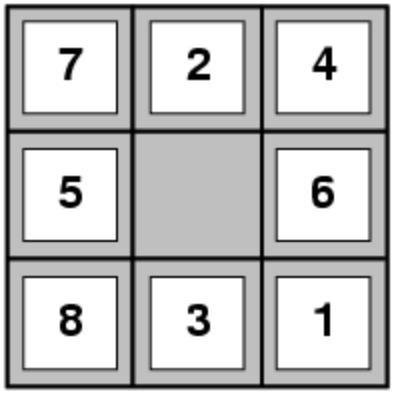
States: Location of each of the tiles

Initial state: Any configuration

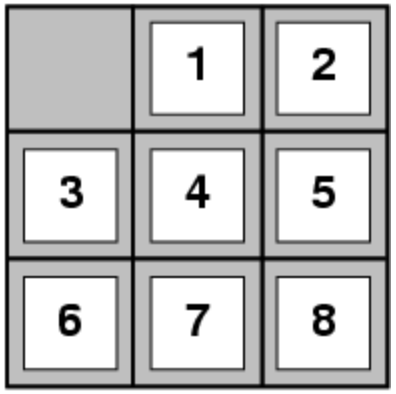
Actions: {Left, Right, Up, Down}

Goal test: check if the state matches the goal test configuration

Path cost: $c(s, a, s') = 1$



Start State

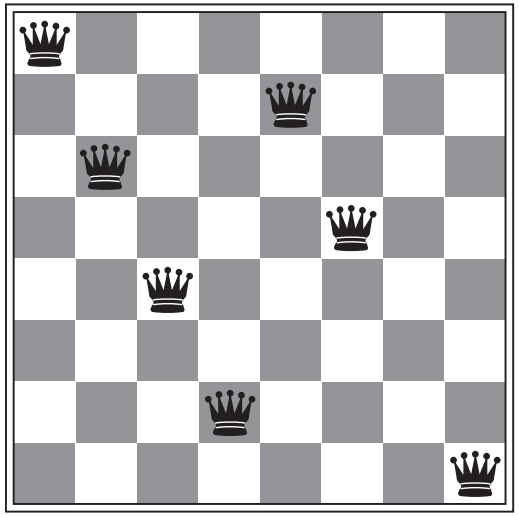


Goal State

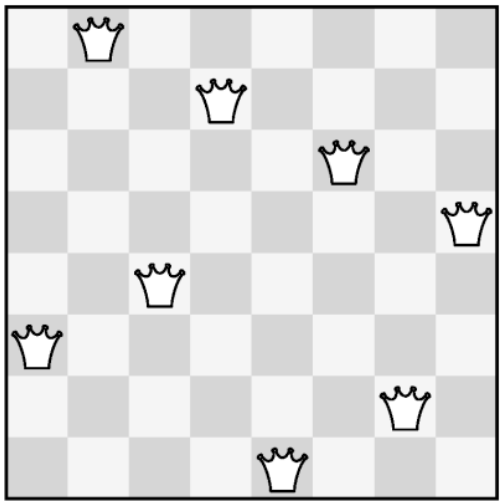
Example: 8-queens problem

- Goal: place eight queens on a chessboard such that no queen attacks any other

- A queen attacks another queen on the same row, column, or diagonal

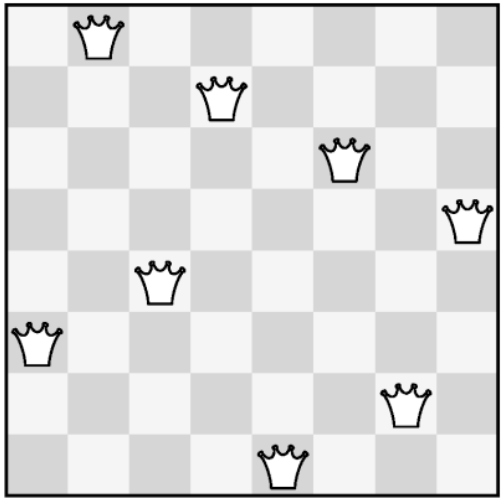


WRONG



CORRECT

8-queens problem: Naïve incremental formulation



States: Any arrangement of 0 to 8 queens

Initial state: No queens on the board

Actions: Add a queen to any empty space

Note: This is an incremental formulation

Transition model: Return the board with a queen added to specified square

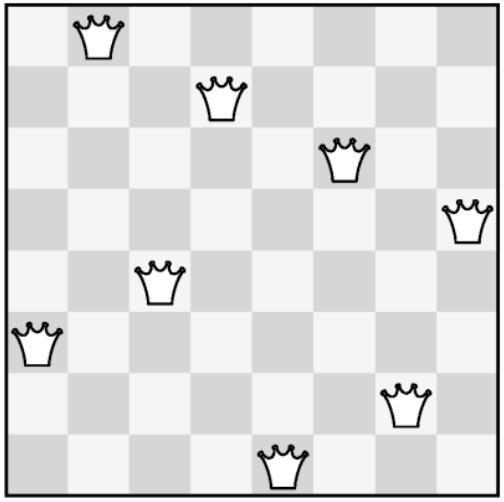
Goal test: 8 queens on the board, none are attacked

Note: Sequences to investigate = 64*63*…57~ 1.8×10^{14}

For 100-queen problem – roughly 10^{400} states !!!

8-queens problem: Advanced incremental formulation

States: Any arrangement of n queens, one per column in the leftmost n columns, with no queen attacking another



Actions: Add a queen to any square in the leftmost empty column such that it is not attacked by a queen

Goal test: 8 queens on the board, none are attacked

Note: Sequences to investigate are reduced from 1.8×10^{14} to 2,057

Toy vs. Real-world problems

1. Route-finding (computer networks, transport, military operation planning, etc.)
2. Touring: travel through all cities
3. VLSI Layout
4. Robot navigation
5. Protein design

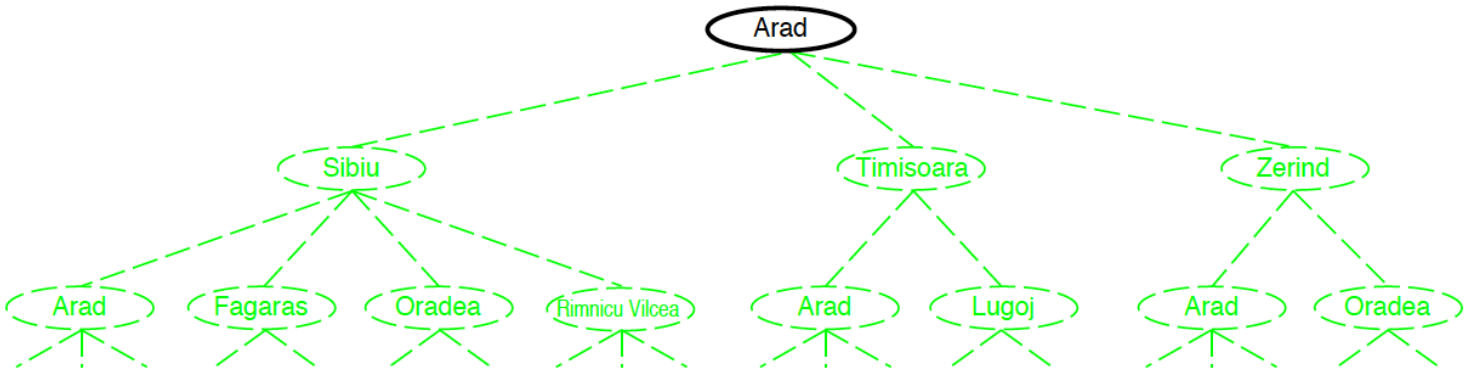
Searching using search tree

Ideas:

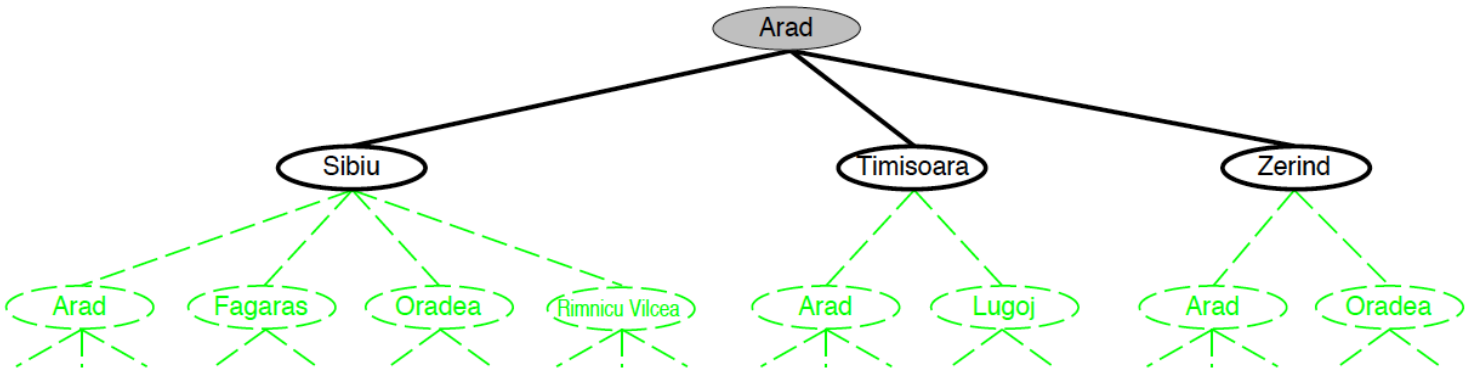
- explore the state space
- expand from the current state
- how to expand the state will depend on the search strategy

```
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
    end
```

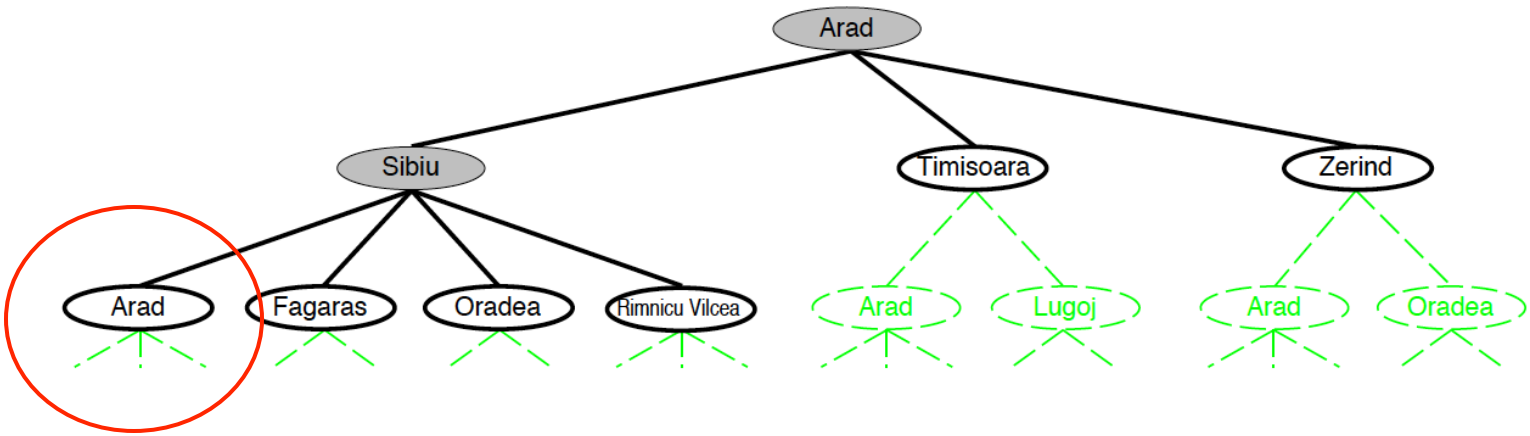
Searching a tree: Example



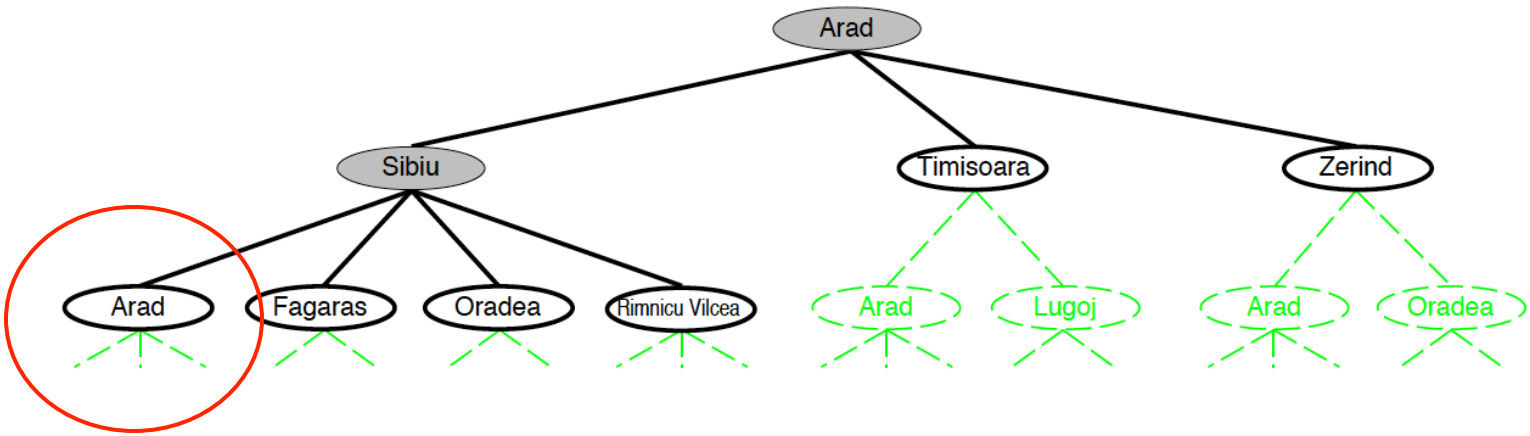
Searching a tree: Example



Searching a tree: Example



Searching a tree: Example



Definition: *In(Arad)* is a repeated state generated by a loopy path

This means that the complete search tree is infinite

Measuring problem-solving performance

- *Completeness*: Is the algorithm guaranteed to find a solution if there is one?
- *Optimality*: Does the strategy find the optimal solution?
- *Time complexity*: How long does it take to find a solution?
- *Space complexity*: How much memory is needed to perform the search?

Uninformed search

- Aka the blind search: the strategy has no additional information about the states beyond that one provided in the problem definition
- *Breadth-first search*
- *Uniform-cost search*
- *Depth first search*
  - *Backtracking search*
- *Depth-limited search*
- *Iterative deepening depth-first search*
- *Bidirectional search*

Uninformed search

- Aka the blind search: the strategy has no additional information about the states beyond that one provided in the problem definition
- *Breadth-first search*
- *Uniform-cost search*
- *Depth first search*
  - *Backtracking search*
- *Depth-limited search*
- *Iterative deepening depth-first search*
- *Bidirectional search*

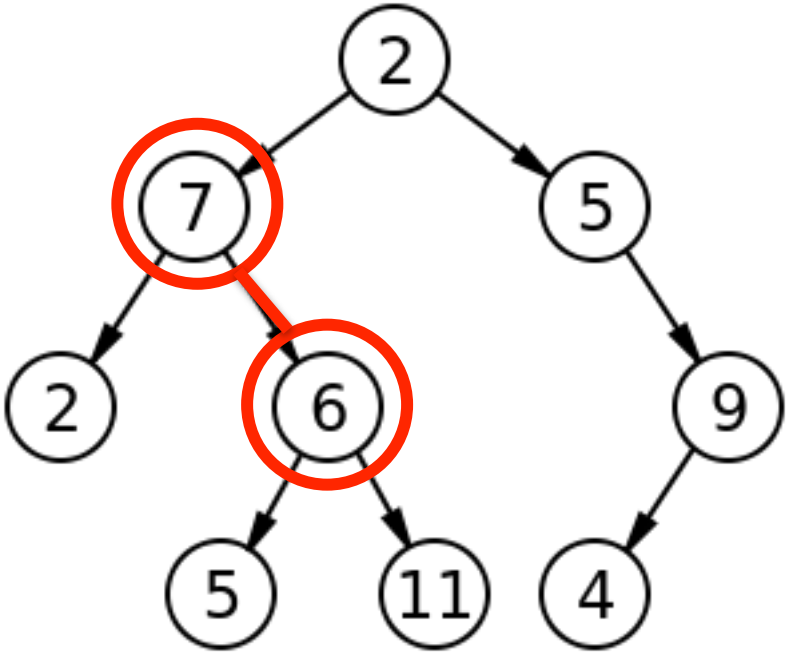
All search strategies are distinguished by the order in which nodes are expanded

Measuring problem-solving performance

- *Completeness*: Is the algorithm guaranteed to find a solution if there is one?
- *Optimality*: Does the strategy find the optimal solution?
- *Time complexity*: How long does it take to find a solution?
- *Space complexity*: How much memory is needed to perform the search?

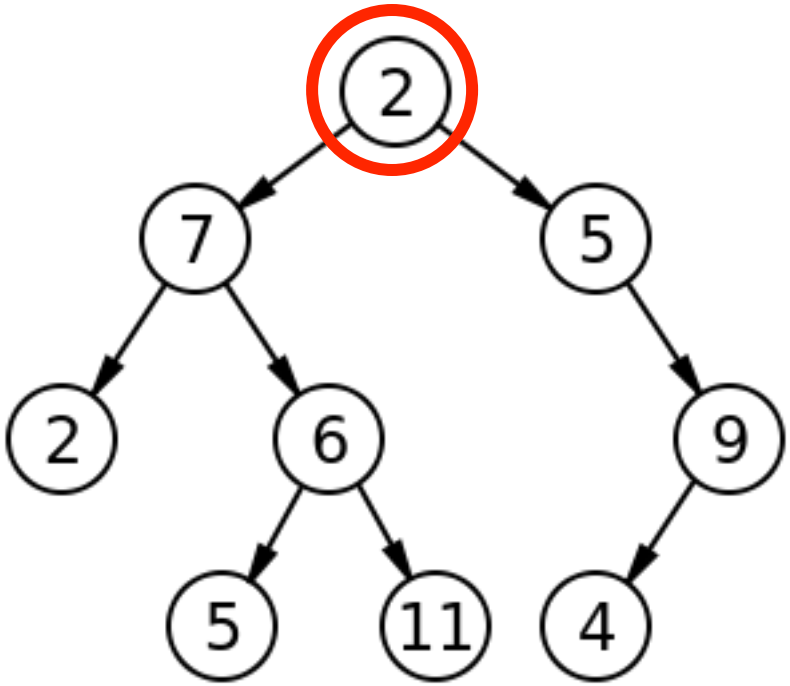
Tree terminology

- *Parent/Child*: Relationship



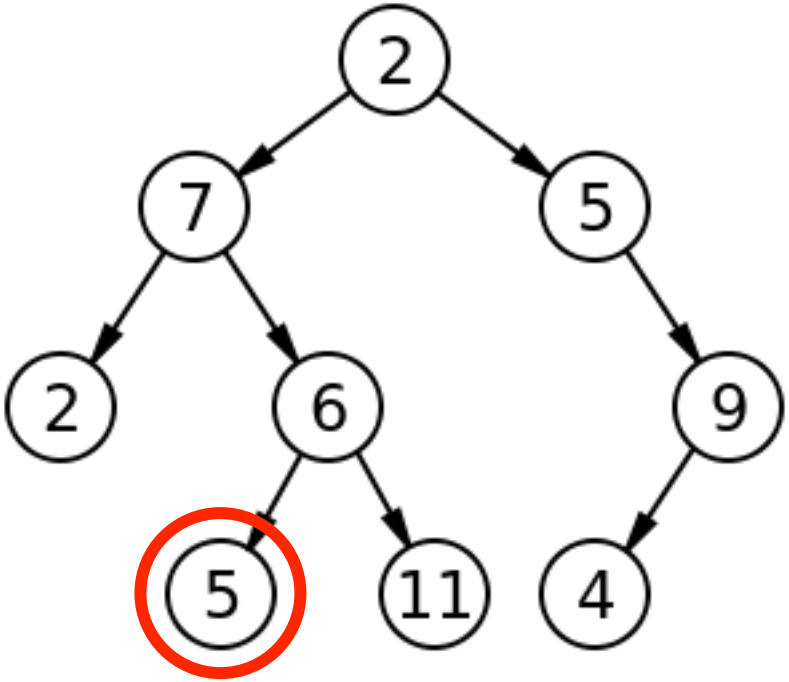
Tree terminology

- *Parent/Child*: Relationship
- *Root*: No parent



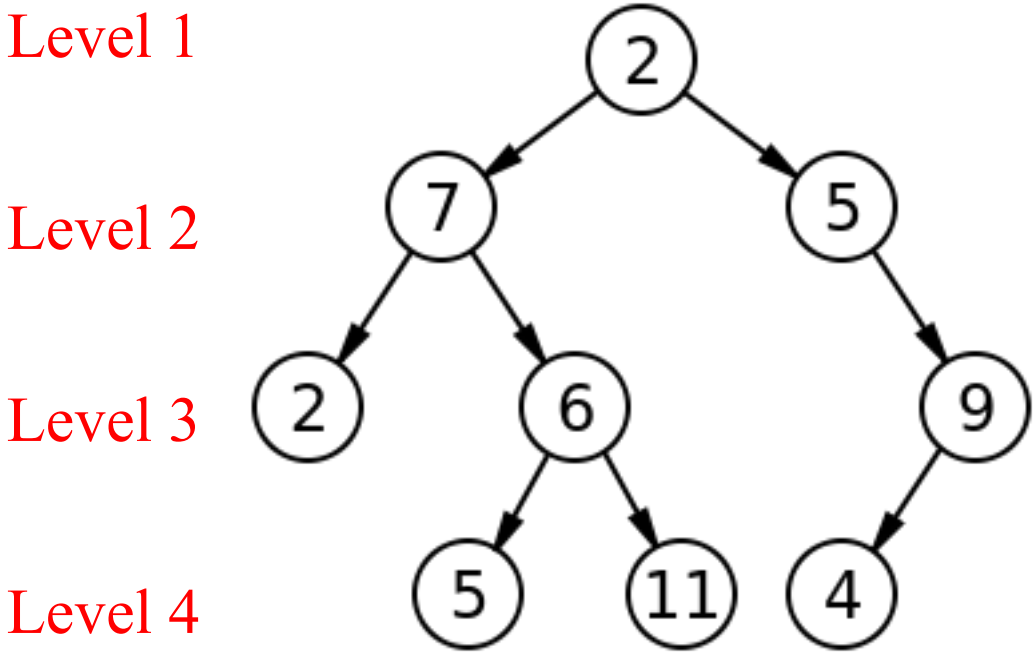
Tree terminology

- *Parent/Child*: Relationship
- *Root*: No parent
- *Leaf*: No child



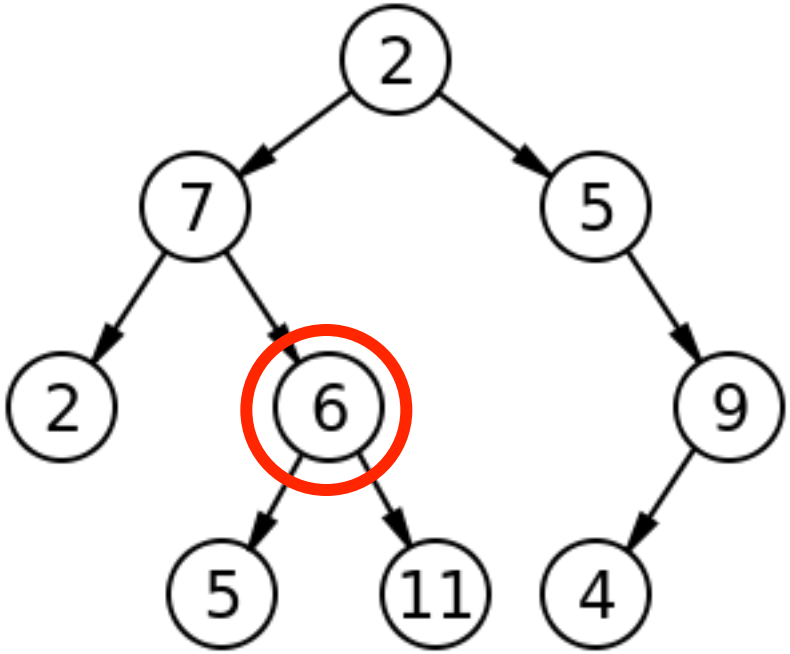
Tree terminology

- *Parent/Child*: Relationship
- *Root*: No parent
- *Leaf*: No child
- *Level*: Depth



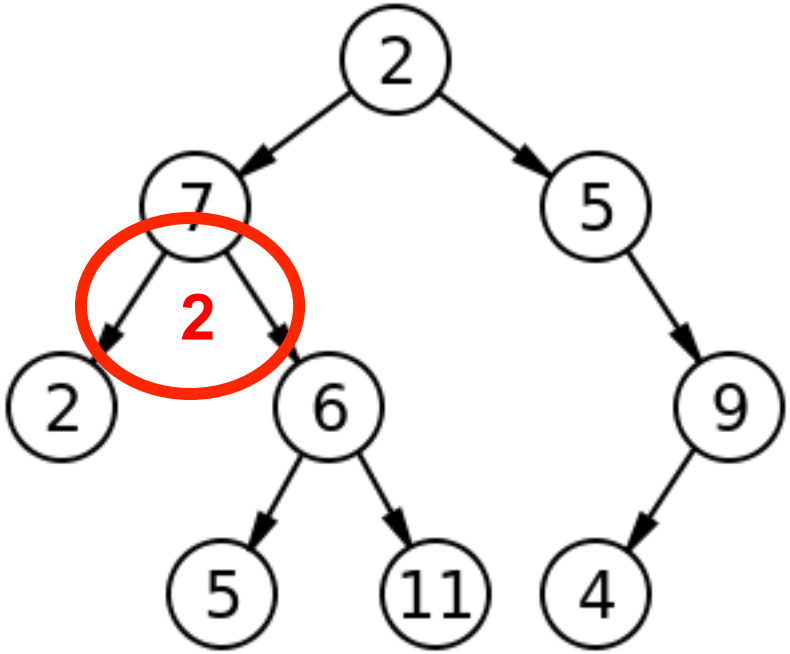
Tree terminology

- *Parent/Child*: Relationship
- *Root*: No parent
- *Leaf*: No child
- *Level*: Depth
- *Internal nodes*: Parents and children



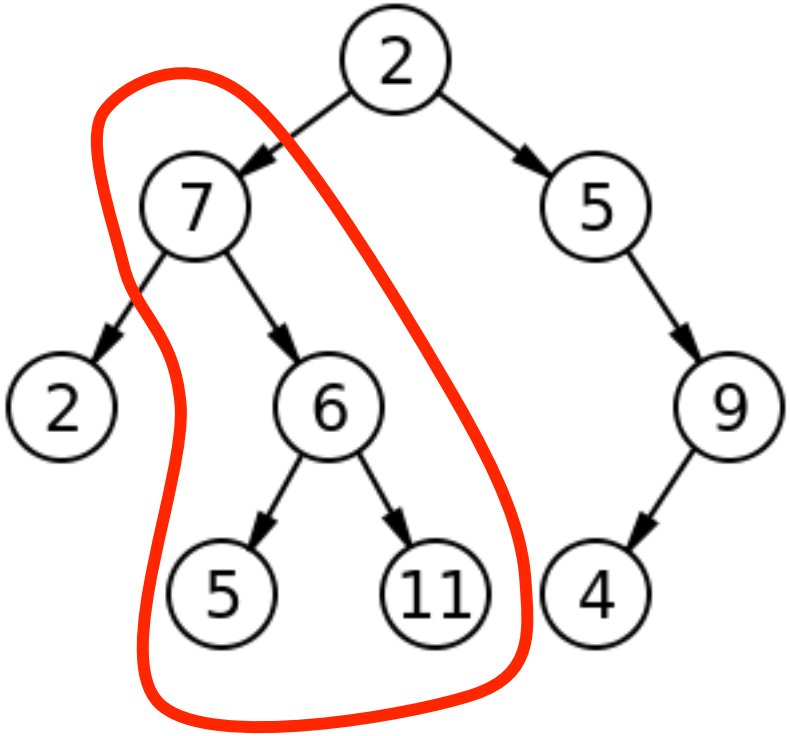
Tree terminology

- *Parent/Child*: Relationship
- *Root*: No parent
- *Leaf*: No child
- *Level*: Depth
- *Internal nodes*: Parents and children
- *Branching factor*: N of children



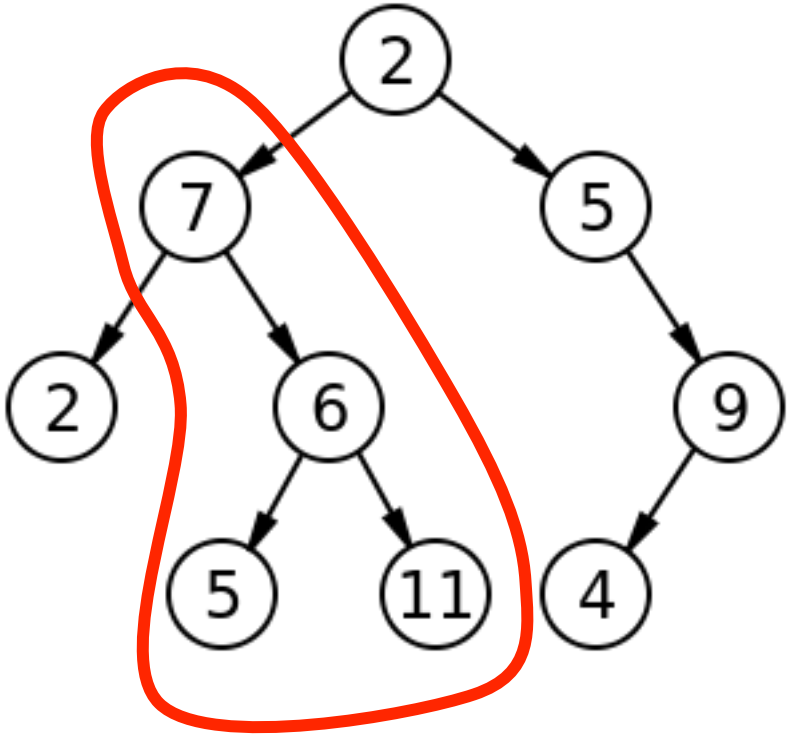
Tree terminology

- *Parent/Child*: Relationship
- *Root*: No parent
- *Leaf*: No child
- *Level*: Depth
- *Internal nodes*: Parents and children
- *Branching factor*: N of children
- *Subtree*: A part of tree (a tree too)



Tree terminology

- *Parent/Child*: Relationship
- *Root*: No parent
- *Leaf*: No child
- *Level*: Depth
- *Internal nodes*: Parents and children
- *Branching factor*: N of children
- *Subtree*: A part of tree (a tree too)



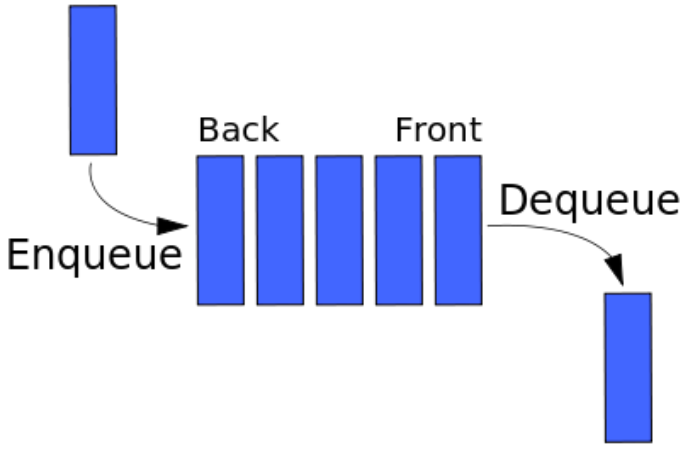
So the complexity of a search algorithm may depend on

- b: maximum branching factor of the search tree
- d: depth of the least-cost (shallowest) solution
- m: maximum depth of the state space (can be infinite)

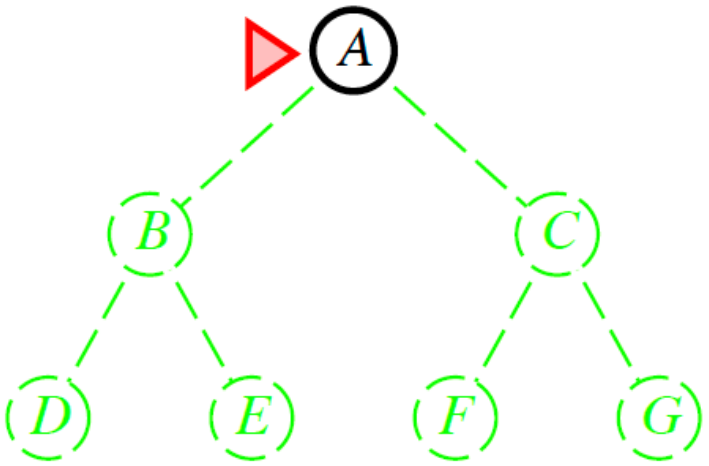
Algorithm 1

Breadth-first search (BFS): Ideas

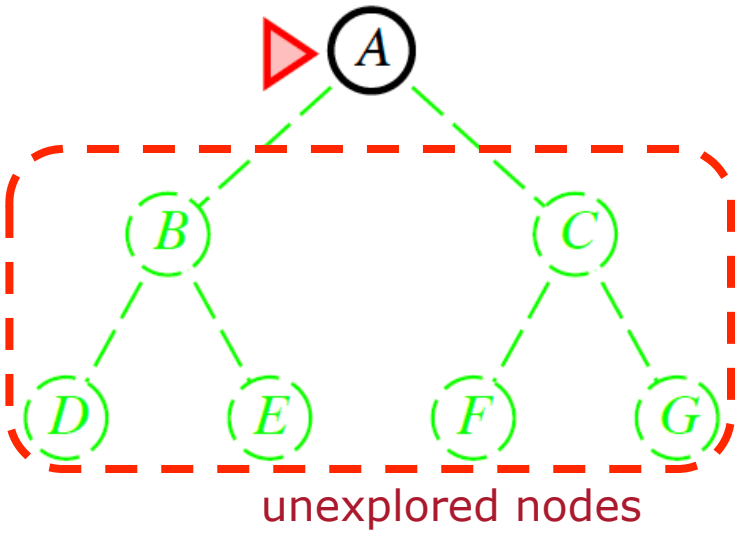
- A simple strategy in which the root node is expanded first, then all successors are expanded, then successors of the successors, *etc*
- All the nodes are expanded at a given depth in the search tree before the any nodes at the next level are expanded
- Expand the shallowest unexpanded node
- Implementation as a First In First Out (*FIFO*) queue: new successors go at end



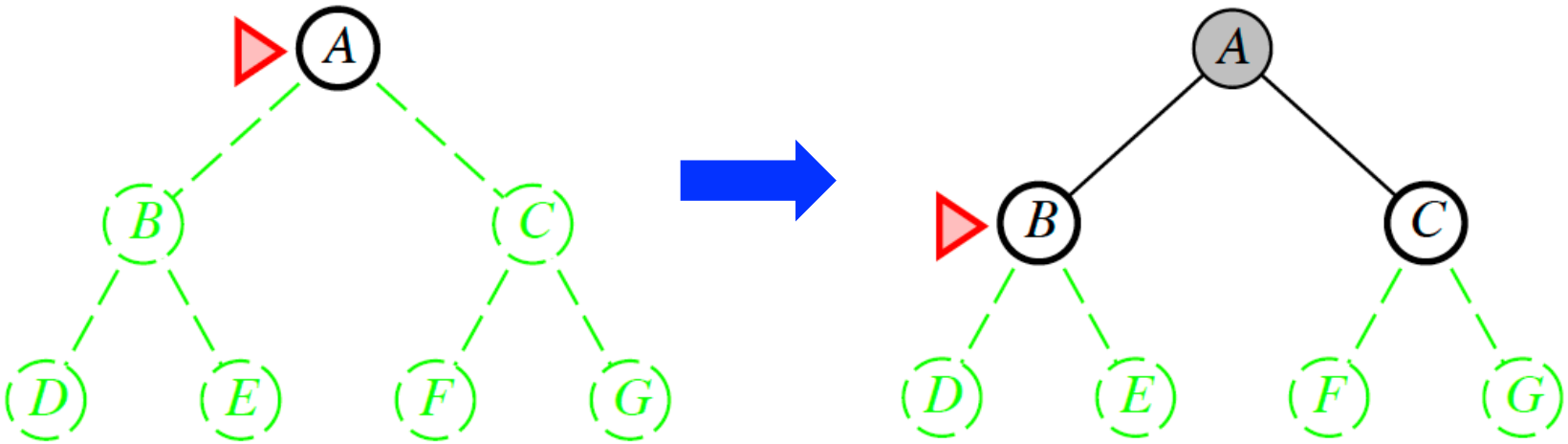
Breadth-first search: Method



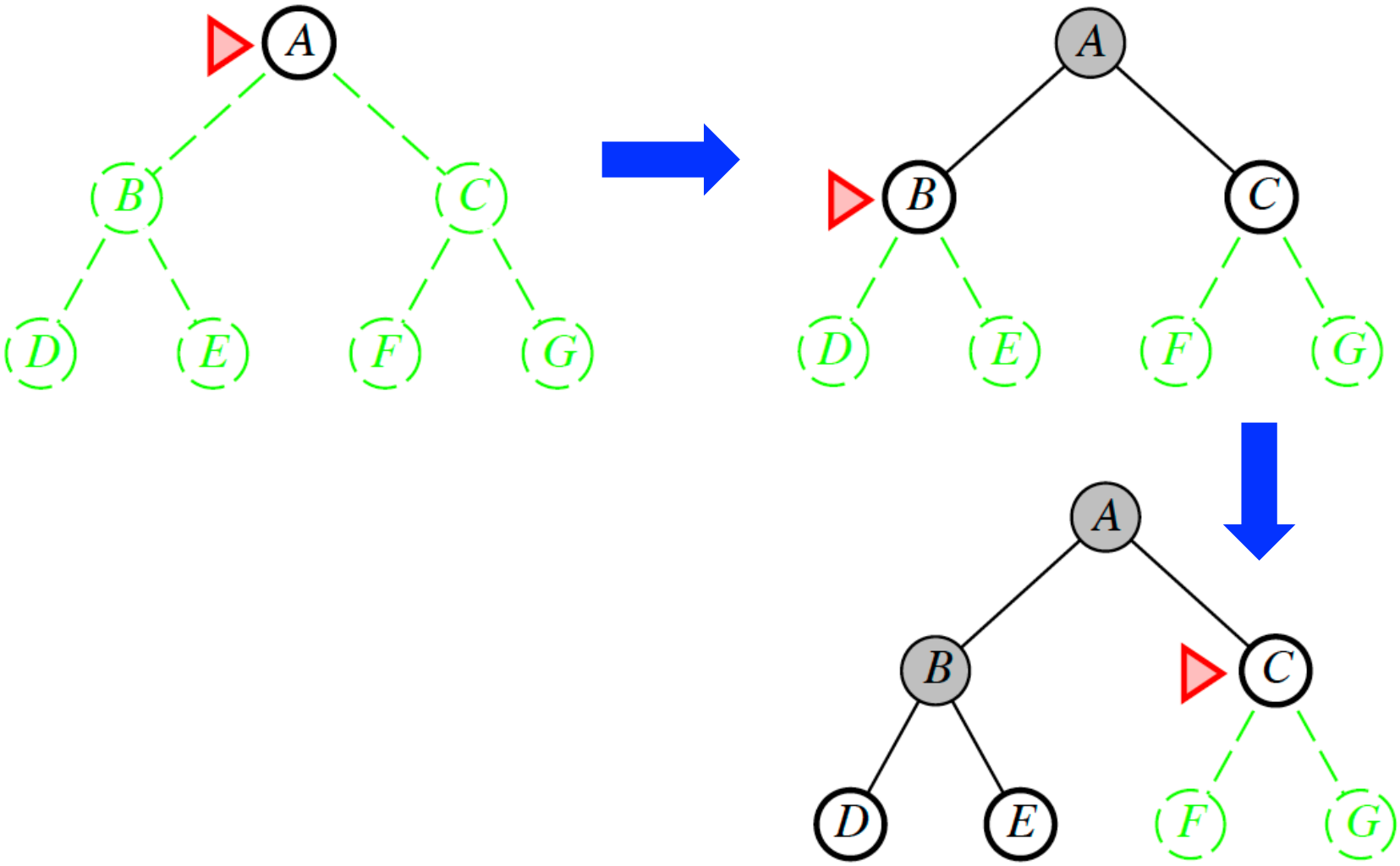
Breadth-first search: Method



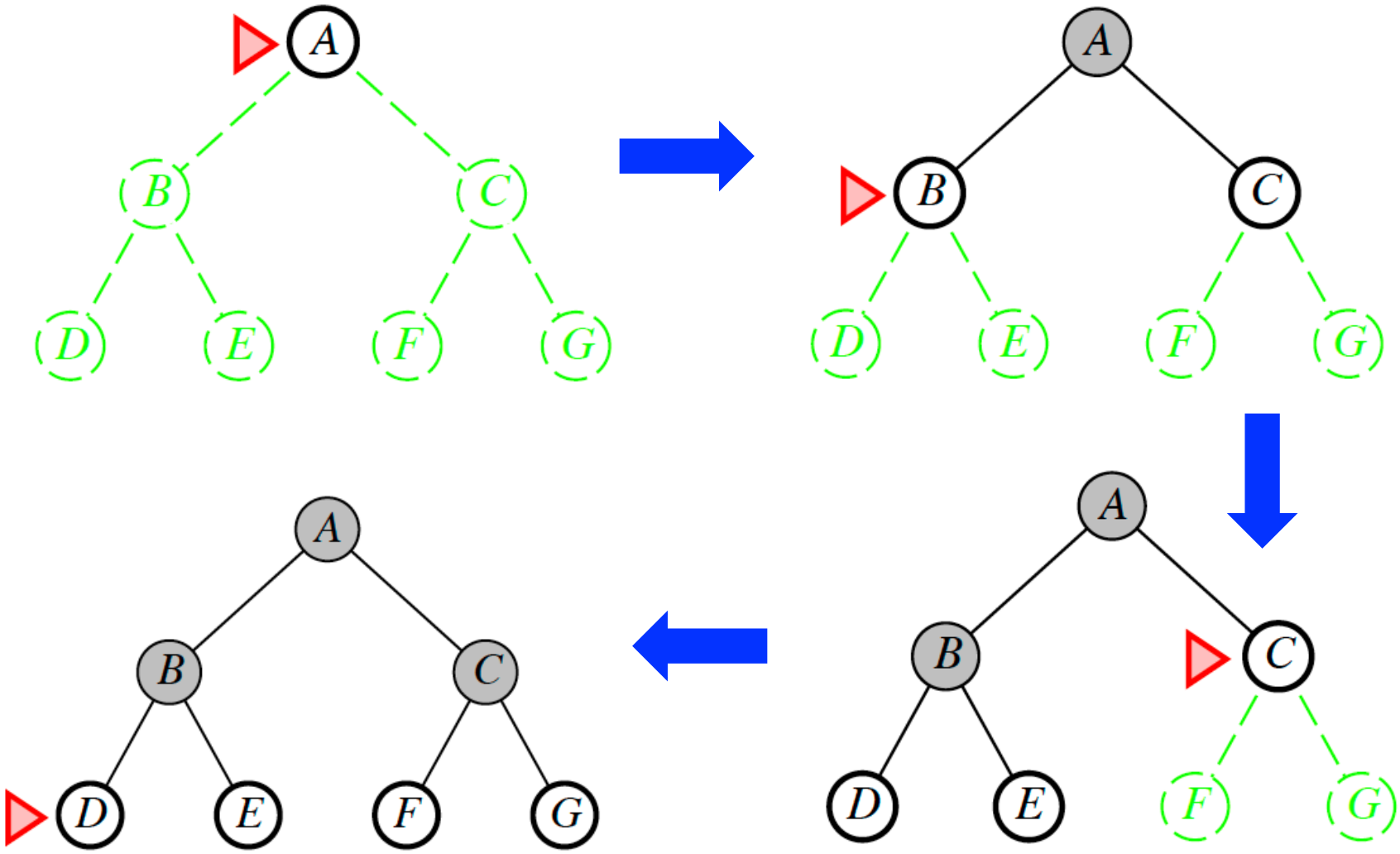
Breadth-first search: Method



Breadth-first search: Method



Breadth-first search: Method



Measuring performance of breadth-first

- *Completeness*: Complete if b is finite

Measuring performance of breadth-first

- *Completeness*: Complete if b *is finite*
- *Optimality*: Optimal if cost is universally c (*e.g.* 1) per each step

Measuring performance of breadth-first search

- *Completeness*: Complete if b *is finite*
- *Optimality*: Optimal if cost is universally c (*e.g.* 1) per each step
- *Time complexity*: Goal test is applied to each node when it is selected for expansion

$$b+b^2+b^3+\ldots+b^d+b(b^d\text{-}1) = \mathrm{O}(b^{d+1})$$

Measuring performance of breadth-first search

- *Completeness*: Complete if b *is finite*
- *Optimality*: Optimal if cost is universally c (*e.g.* 1) per each step
- *Time complexity*: (1) Goal test is applied to each node when it is selected for expansion

$$b+b^2+b^3+\ldots+b^d+b(b^d\text{-}1)=\mathrm{O}(b^{d+1})$$

(2) Goal test is applied to each node when it is generated rather than when it is selected for expansion

$$b+b^2+b^3+\ldots+b^d=\mathrm{O}(b^d)$$

- *Space complexity*: $\mathrm{O}(b^{d+1})$—keeps every node in memory
  - A much bigger problem than time

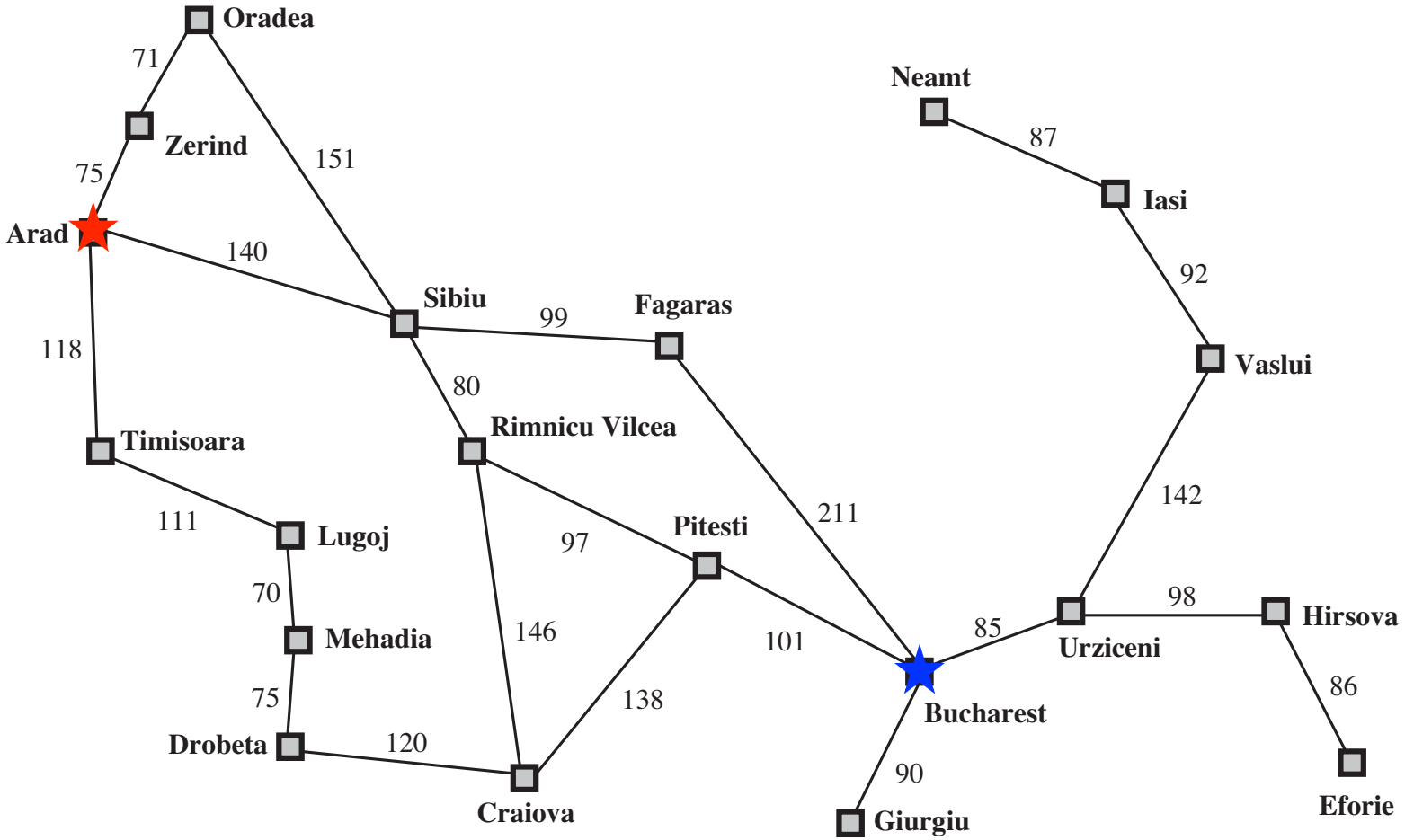
Time *vs.* memory requirements for BFS

Depth	Nodes	Time	Memory
2	1100	.11 seconds	1 megabyte
4	111,100	11 seconds	106 megabytes
6	10^7	19 minutes	10 gigabytes
8	10^9	31 hours	1 terabytes
10	10^{11}	129 days	101 terabytes
12	10^{13}	35 years	10 petabytes
14	10^{15}	3,523 years	1 exabyte

Figure 3.11 Time and memory requirements for breadth-first search. The numbers shown assume branching factor $b = 10$; 10,000 nodes/second; 1000 bytes/node.

Algorithm 2

BFS did not use the immediately available road length information



Uniform-cost search: Ideas

- Branches of the tree (steps) have different weights (step costs)
- Expand least-cost unexpanded node
- Equivalent to BFS if step costs all equal
- *Implementation*: queue ordered by path cost

Measuring performance of uniform-cost search

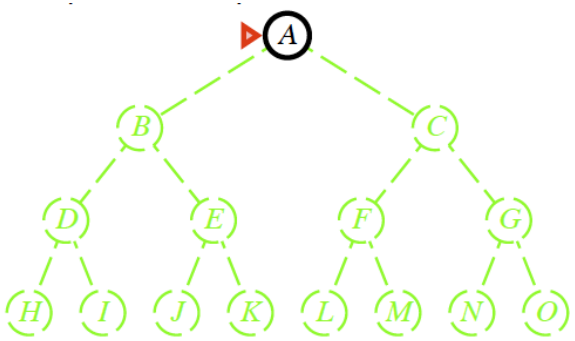
- *Completeness*: Complete if step cost is $\geq \varepsilon$
- *Optimality*: Yes: nodes are expanded in increasing order of $g(n)$
- *Time complexity*: Number of nodes with $g \leq \text{cost}$ of optimal solution, $O(b^{1+ceiling(C^*/\varepsilon)})$ where C^* is the cost of the optimal solution
- *Space complexity*: Number of nodes with $g \leq \text{cost}$ of optimal solution, $O(b^{1+ceiling(C^*/\varepsilon)})$

Algorithm 3

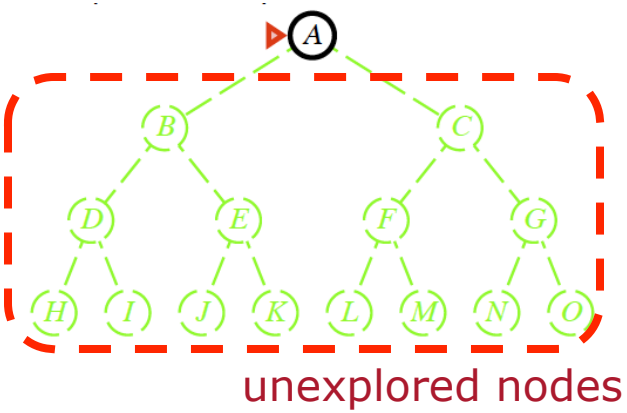
Depth first search (DFS): Ideas

- Always expand the deepest node in the current frontier
- As those nodes are expanded, they are dropped from the frontier, so the search backups to the next deepest unexplored node.
- *Implementation*: Last-in-first-out (LIFO) queue—put successors in front (most recently generated node is chosen for expansion)
- It is common to implement DFS with a recursive function that calls itself on each child

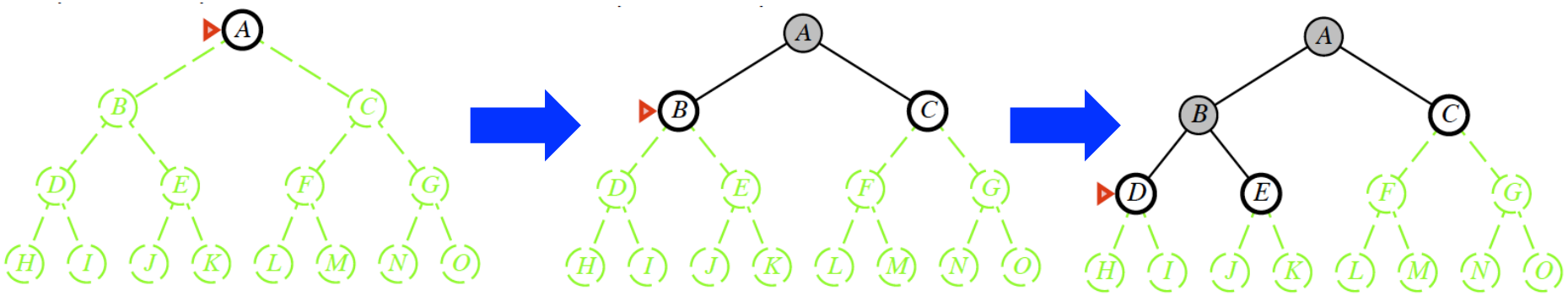
Depth first search (DFS)



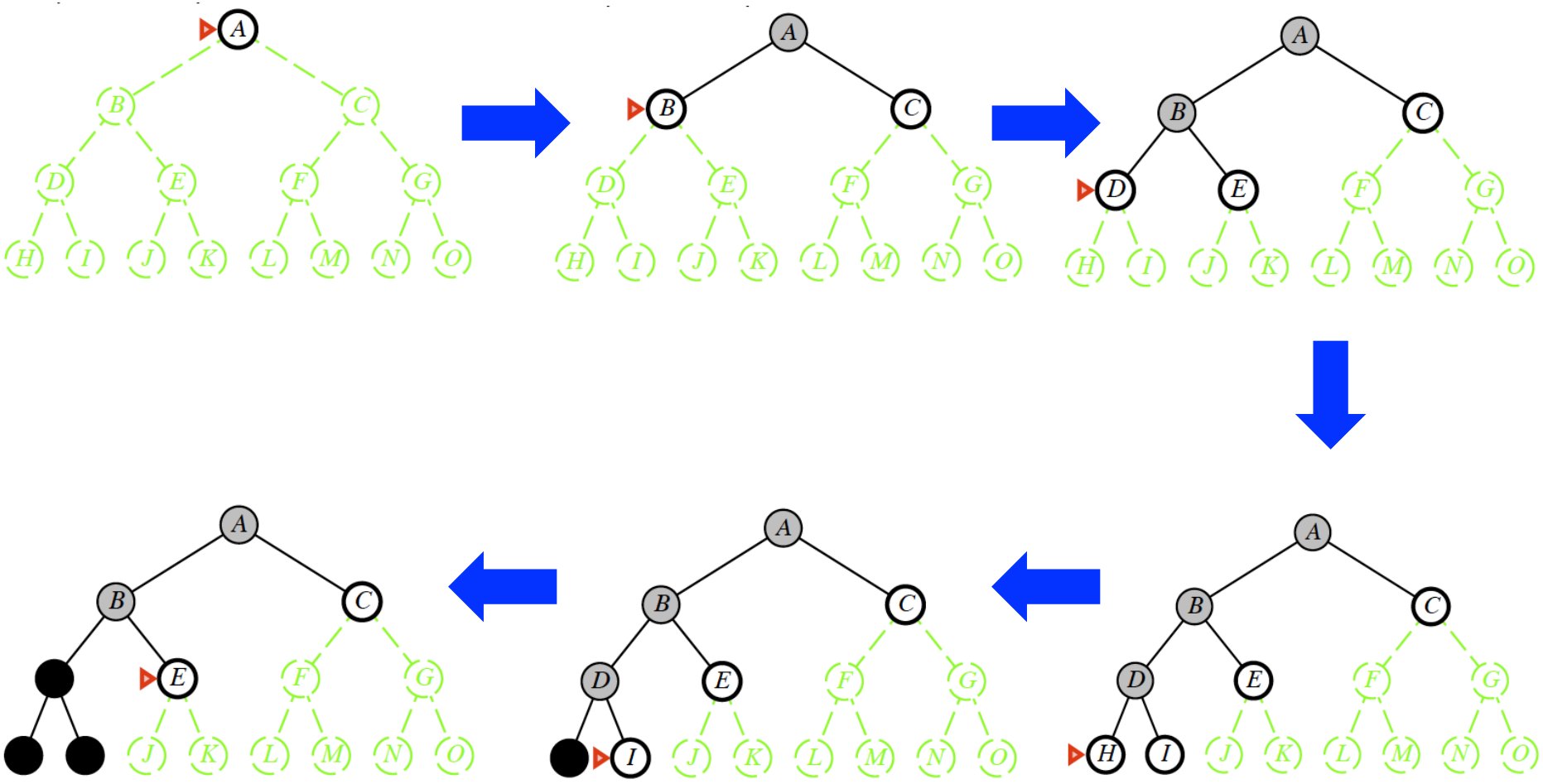
Depth first search (DFS)



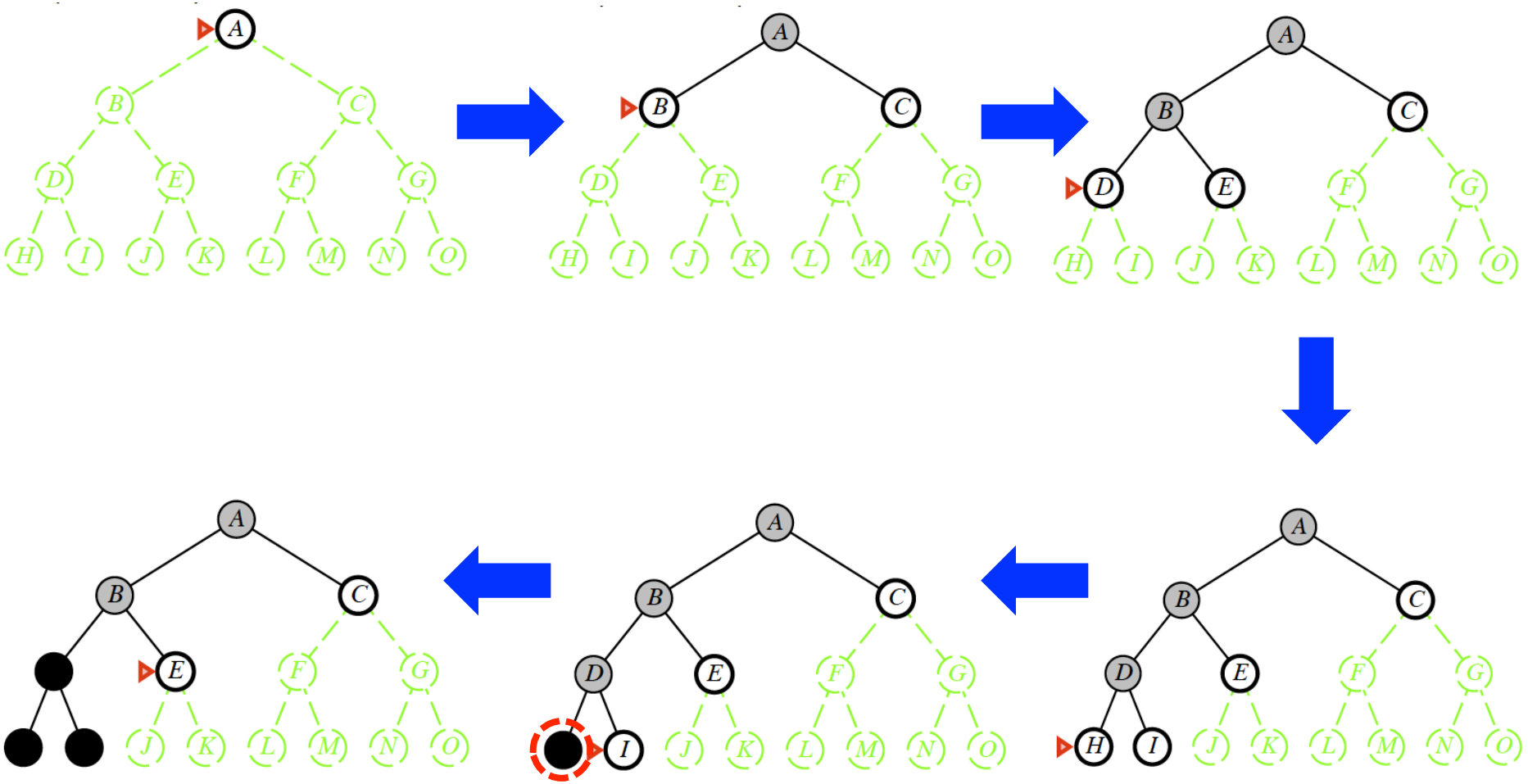
Depth first search (DFS)



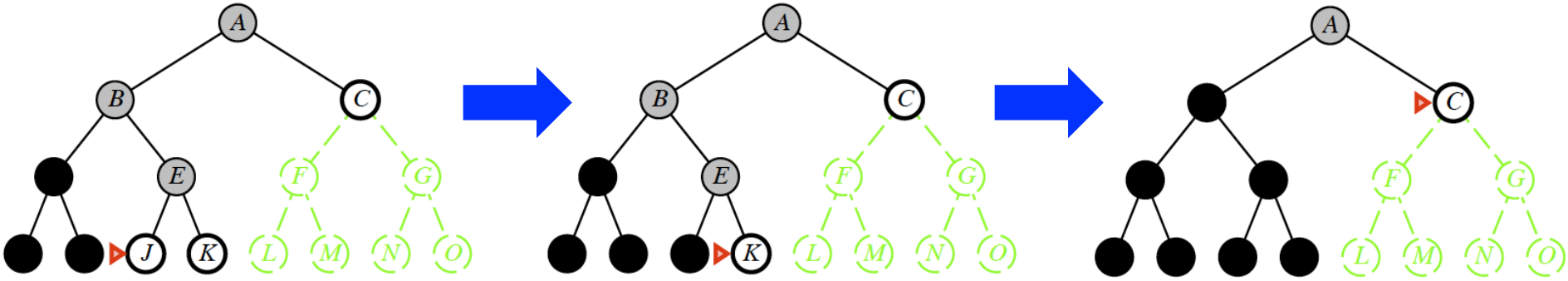
Depth first search (DFS)



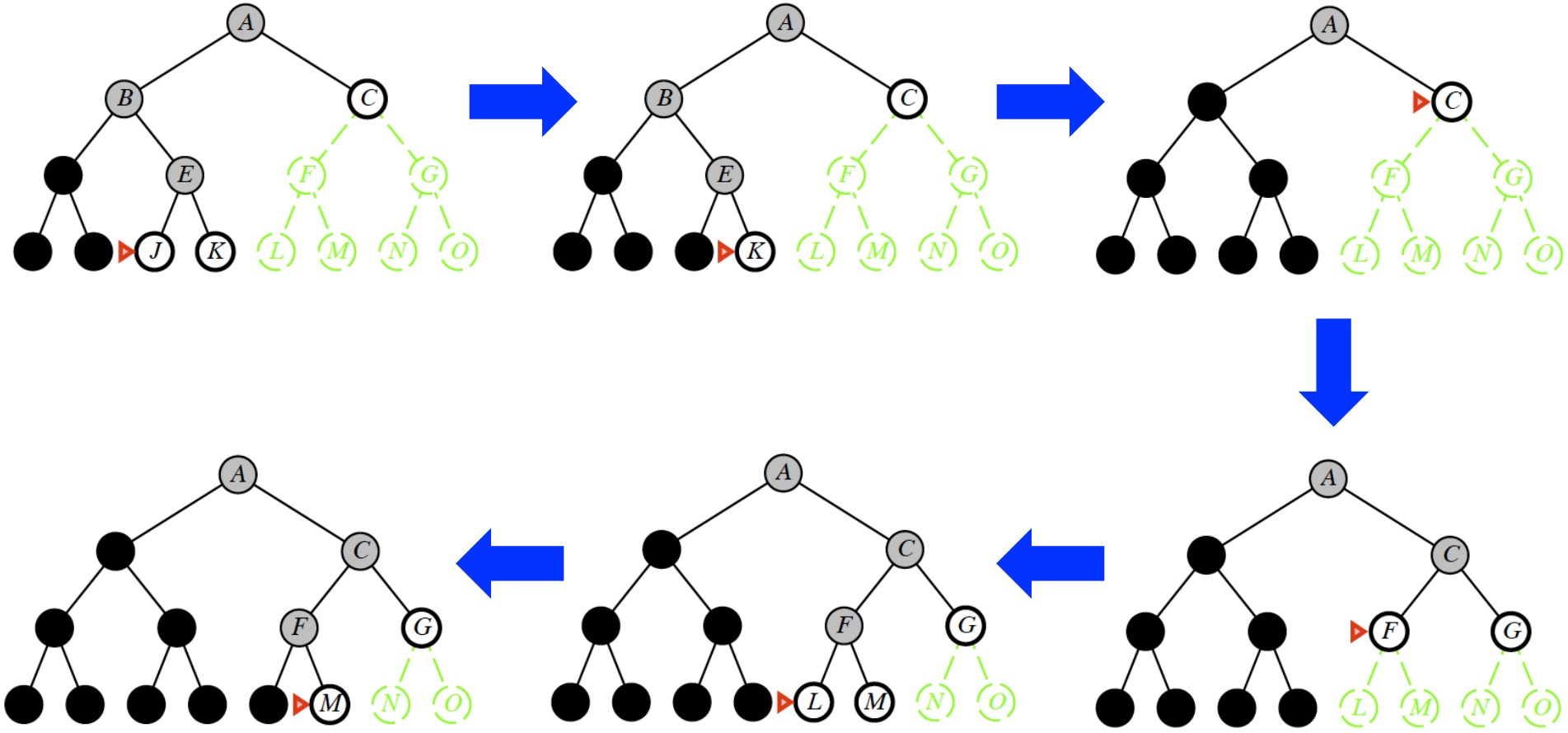
Depth first search (DFS)



Depth first search (DFS)



Depth first search (DFS)

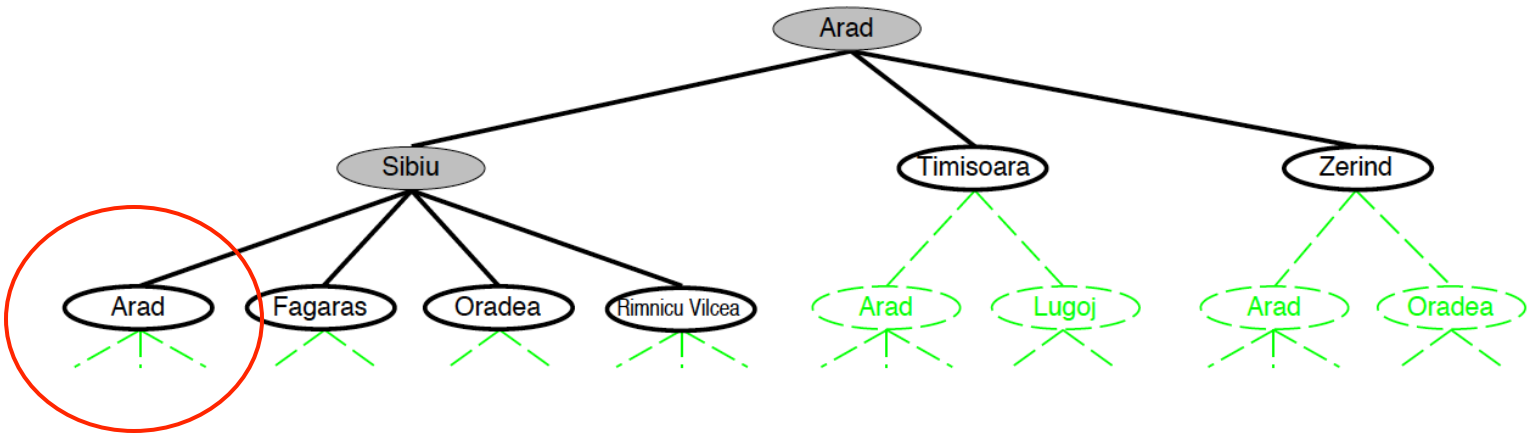


- b: maximum branching factor of the search tree
- d: depth of the least-cost (shallowest) solution
- m: maximum depth of the state space (can be infinite)

Measuring performance of DFS

- *Completeness*: No, fails in infinite-depth spaces (spaces with loops)
  - Needs to be modified to avoid repeated states along path
  - As a result: complete in finite spaces
- *Optimality*: No
- *Time complexity*: $O(b^m)$ only useful if m is much smaller than d
  - If solutions are dense, may be much faster than BFS
- *Space complexity*: $O(bm)$ — linear!!!

Searching a tree: Example



Definition: *In(Arad)* is a repeated state generated by a loopy path

This means that the complete search tree is infinite

Algorithm 4

Backtracking search: Ideas

- Only one successor is generated at a time rather than all successors
- Each partially expanded node remembers which successor to generate next
- Generate a successor by modifying the current state description directly rather than copying it first
- However, we must able to “undo” each modification when we go back to generate the next successor

Measuring performance of backtracking

- *Completeness*: No, fails in infinite-depth spaces (spaces with loops)
  - Needs to be modified to avoid repeated states along path
  - As a result: complete in finite spaces
- *Optimality*: No
- *Time complexity*: $O(b^m)$
- *Space complexity*: $O(m)$ — linear, independent of b !!!

Algorithm 5

Depth limited search: Ideas

- Depth-first search with depth limit l, i.e., nodes at depth l have no successors
- Allows to avoid the failure of DFS in infinite state spaces
- Sometimes the depth limit can be based on the knowledge od the problem

Algorithm

```
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 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
```

Measuring performance of depth-limited search

- *Completeness*: No—Introduces an additional source of incompleteness, when $l<d$: the shallowest goal is beyond the depth limit

- *Optimality*: No

- *Time complexity*: $O(b^l)$ only useful if l is much smaller than d
  - If solutions are dense, may be much faster than BFS

- *Space complexity*: $O(bl)$ — linear!!!

Algorithm 6

Iterative deepening DFS: Ideas

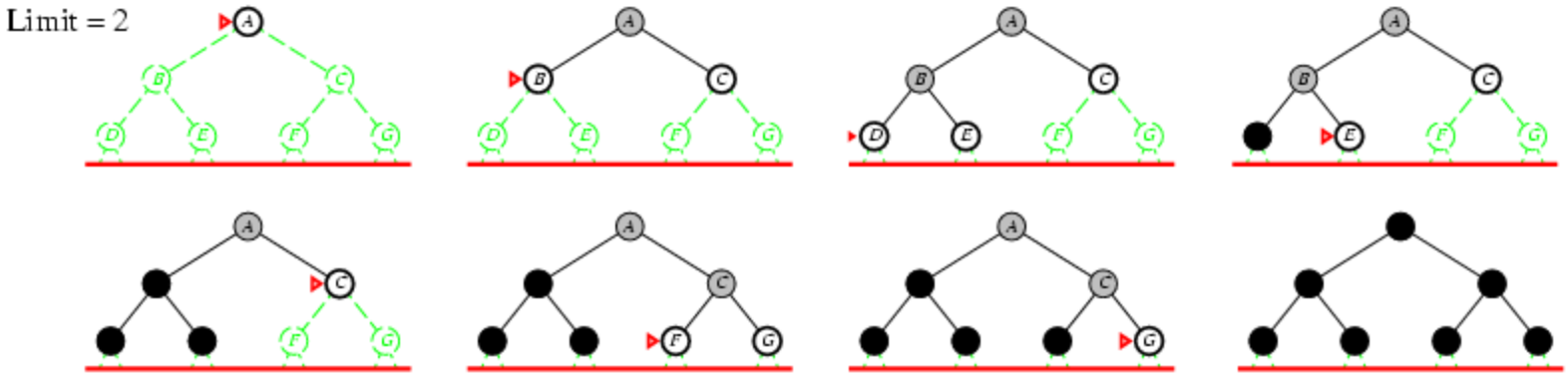
- Strategy that finds the best depth limit
- It gradually increases the limit from 0 to 1, to 2, to 3, until a goal is found
- While it may seem to be wasteful because states are generated multiple time, it is actually not too costly
- Indeed, in a search tree with the same (nearly the same) branching factor at each level, most of the nodes are in the bottom level, but the bottom level nodes are generated once

Iterative deepening DFS

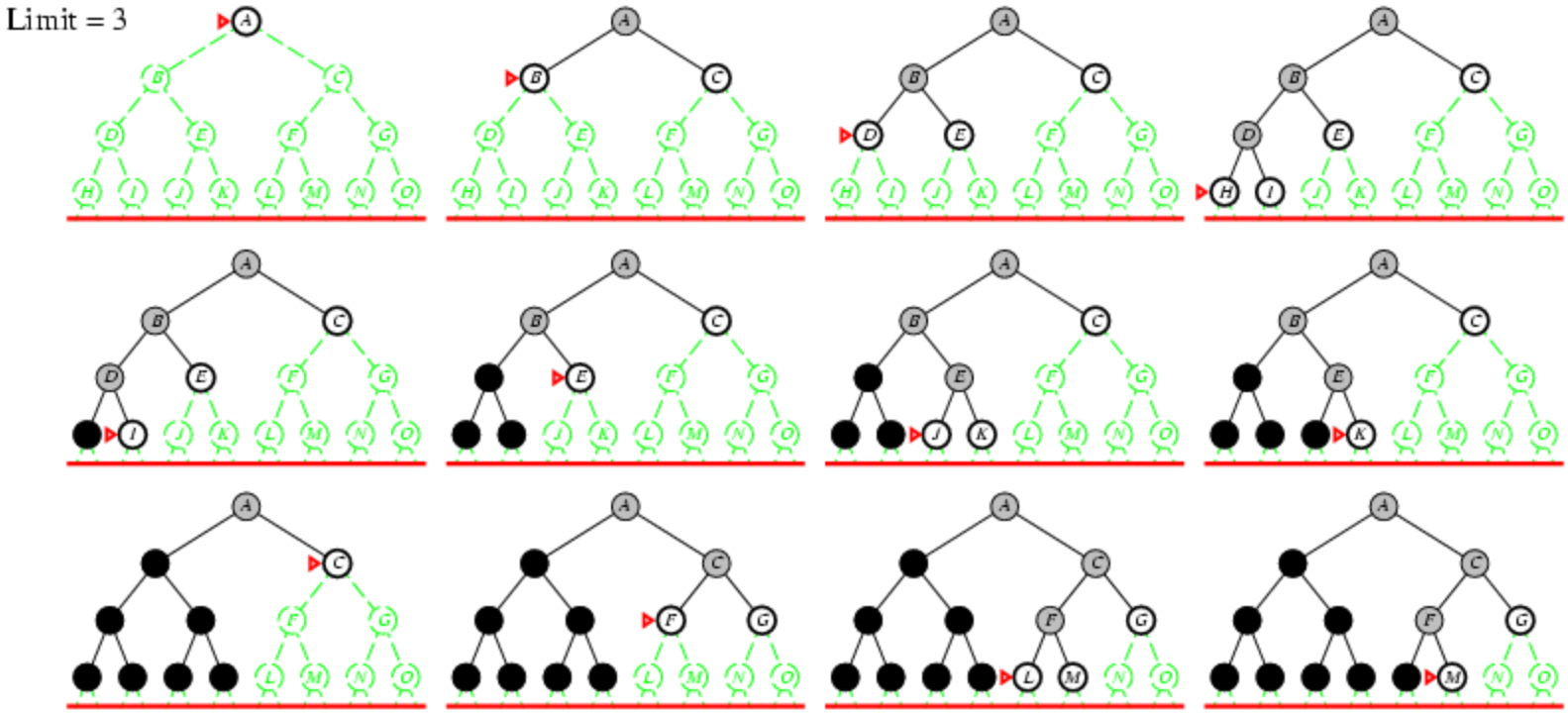
Limit = 0

Iterative deepening DFS

Iterative deepening DFS



Iterative deepening DFS



Algorithm

```
function ITERATIVE-DEEPENING-SEARCH( problem) returns a solution
    inputs: problem, a problem

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

Time complexity analysis

- 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*:

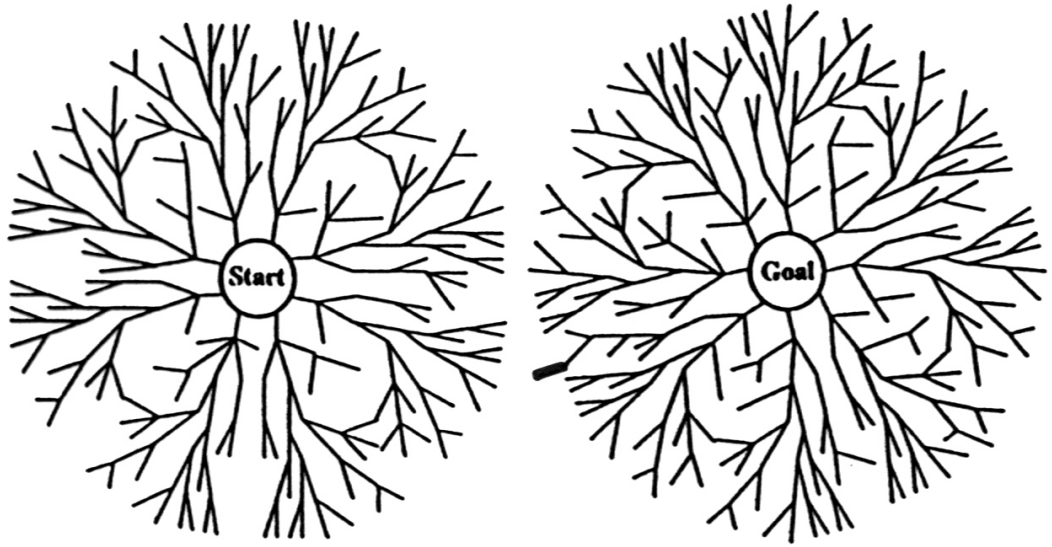
$$N_{IDS} = (d+1)b^0 + d\,b^1 + (d-1)b^2 + \ldots + 3b^{d-2} + 2b^{d-1} + 1b^d$$

- 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%

Measuring performance of iterative deepening DFS

- *Completeness*: Yes
- *Optimality*: *Yes if step cost = 1*
- *Time complexity*: $(d+1)b^0 + d\,b^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$
- *Space complexity*: $O(bd)$ — linear!!!

Bidirectional search



A schematic view of a bidirectional breadth-first search that is about to succeed, when a branch from the start node meets a branch from the goal node.

Motivation: $b^{d/2} + b^{d/2} << b^{d}$

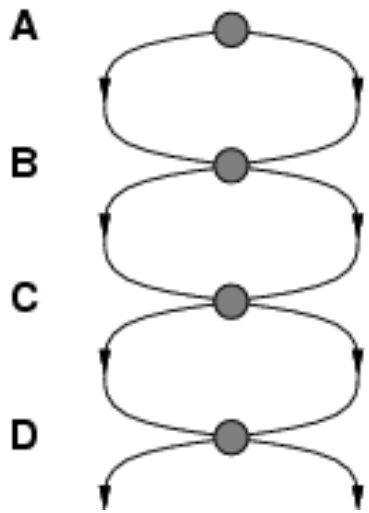
Summary of algorithms

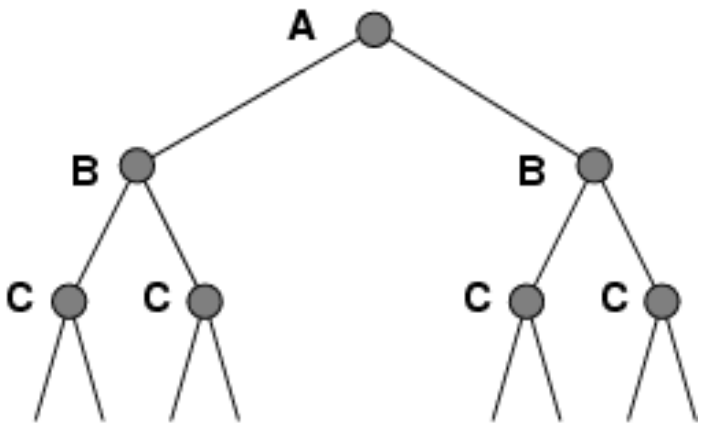
Criterion	BFS	Uniform-Cost	DFS	Depth-Limited	Iterative Deepening DLS	Bidirectional (if applicable)
Complete?	Yes[a]	Yes[a,b]	No	No	Yes[a]	Yes[a,d]
Time	$O(b^d)$	$O(b^{1+C^*/\varepsilon})$	$O(b^m)$	$O(b^l)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^d)$	$O(b^{1+C^*/\varepsilon})$	$O(bm)$	$O(bl)$	$O(bd)$	$O(b^{d/2})$
Optimal?	Yes[c]	Yes	No	No	Yes[c]	Yes[c,d]

Generally the preferred uninformed search strategy

[a] complete if b is finite
[b] complete if step costs $\geq \varepsilon$ for positive
[c] optimal if step costs are all identical
(also if path cost non-decreasing function of depth only)
[d] if both directions use breadth-first search

Repeated states





Need to be able to detect repeated states in order to avoid exponential time complexity:

- Do not return to the state you just came from. Have the expand function (or the operator set) refuse to generate any successor that is the same state as the node's parent.
- Do not create paths with cycles in them. Have the expand function (or the operator set) refuse to generate any successor of a node that is the same as any of the node's ancestors.
- Do not generate any state that was ever generated before. This requires every state that is generated to be kept in memory, resulting in a space complexity of $O(b^d)$, potentially. It is better to think of this as $O(s)$, where s is the number of states in the entire state space.

Let’s continue: Informed search

Material

- Reading Chapter 3 (Section 3.5), Chapter 4

Informed algorithms

- Best-first search
- Greedy best-first search
- A* search
- RBFS
- SMA
- Heuristics
- Local search algorithms
- Hill-climbing search
- Simulated annealing search
- Local beam search
- Genetic algorithms
- AND-OR Search trees
- Searching with no observations
- Online search

Best-first search

- Idea: use an evaluation function *f(n)* for each node
  - Estimate of "desirability"
  - Expand most desirable unexpanded node

- Implementation:

  Order the nodes in frontier in decreasing order of desirability

- Special cases:
  - Greedy best-first search
  - A* search

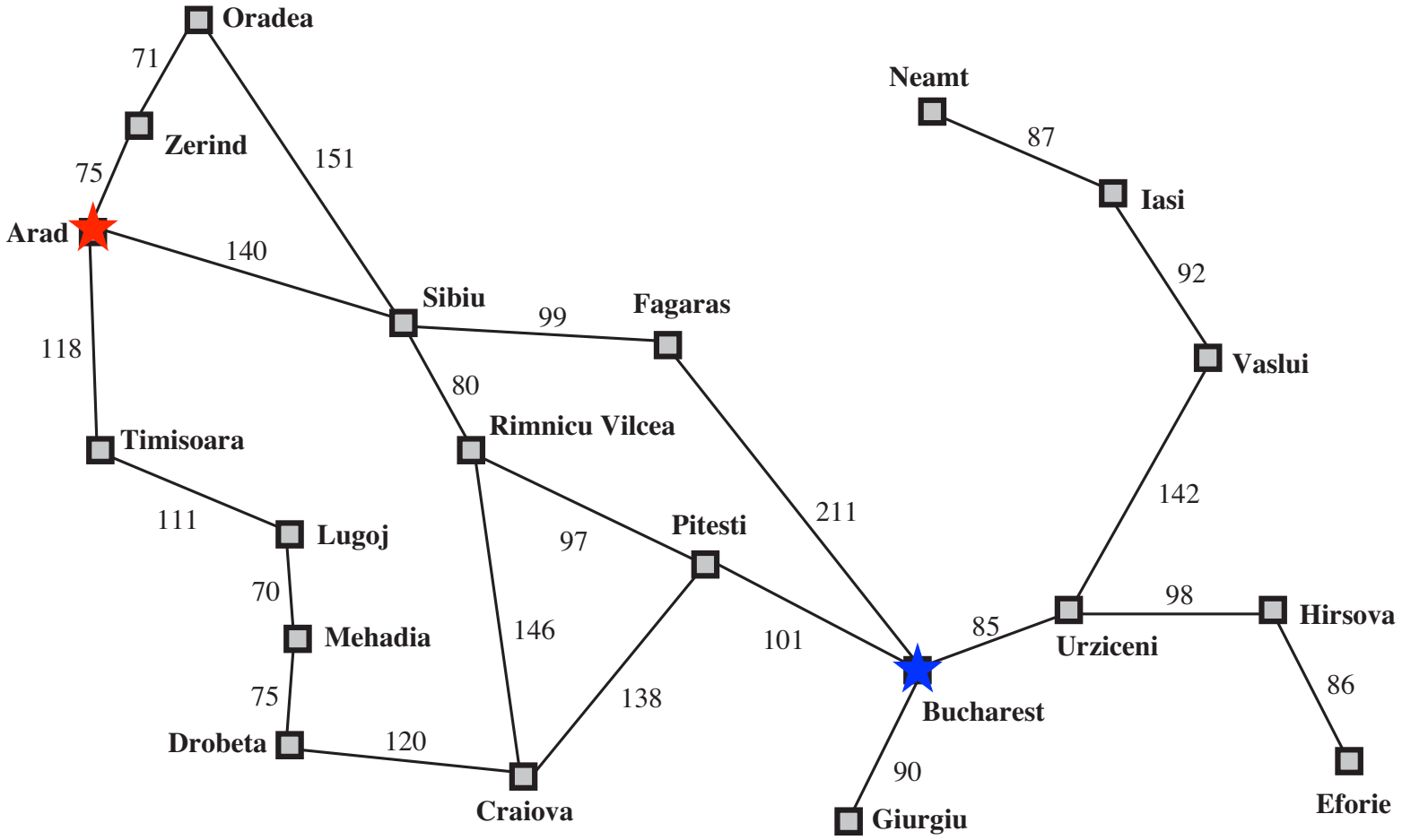
Greedy best-first search: Ideas

- Tries to expand the node that appears closest to the goal first, assuming that it leads to solution quickly

- Evaluation function $f(n) = h(n)$ (heuristic)
  = estimate of cost from n to *goal*

- Example:
  $h_{SLD}(n)$ = straight-line distance from n to Bucharest

Traveling in Hungary



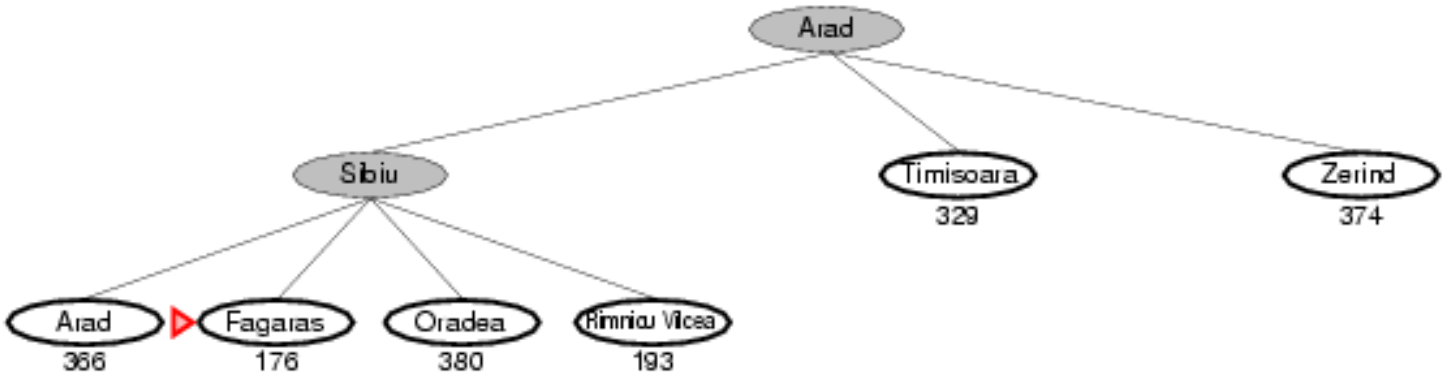
Romania with step costs in km: Straight line distance to Bucharest

Arad	366	**Mehadia**	241
Bucharest	0	**Neamt**	234
Craiova	160	**Oradea**	380
Drobeta	242	**Pitesti**	100
Eforie	161	**Rimnicu Vilcea**	193
Fagaras	176	**Sibiu**	253
Giurgiu	77	**Timisoara**	329
Hirsova	151	**Urziceni**	80
Iasi	226	**Vaslui**	199
Lugoj	244	**Zerind**	374

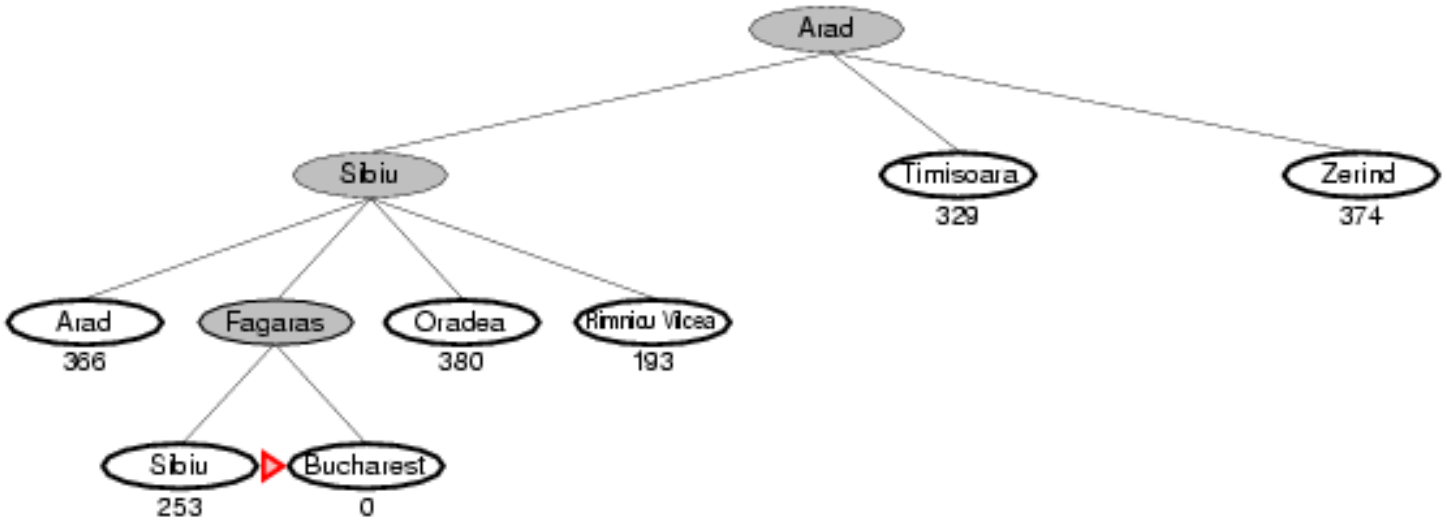
Greedy best-first search example: The initial state

Greedy best-first search example: After expanding Arad

Greedy best-first search example: After expanding Sibiu



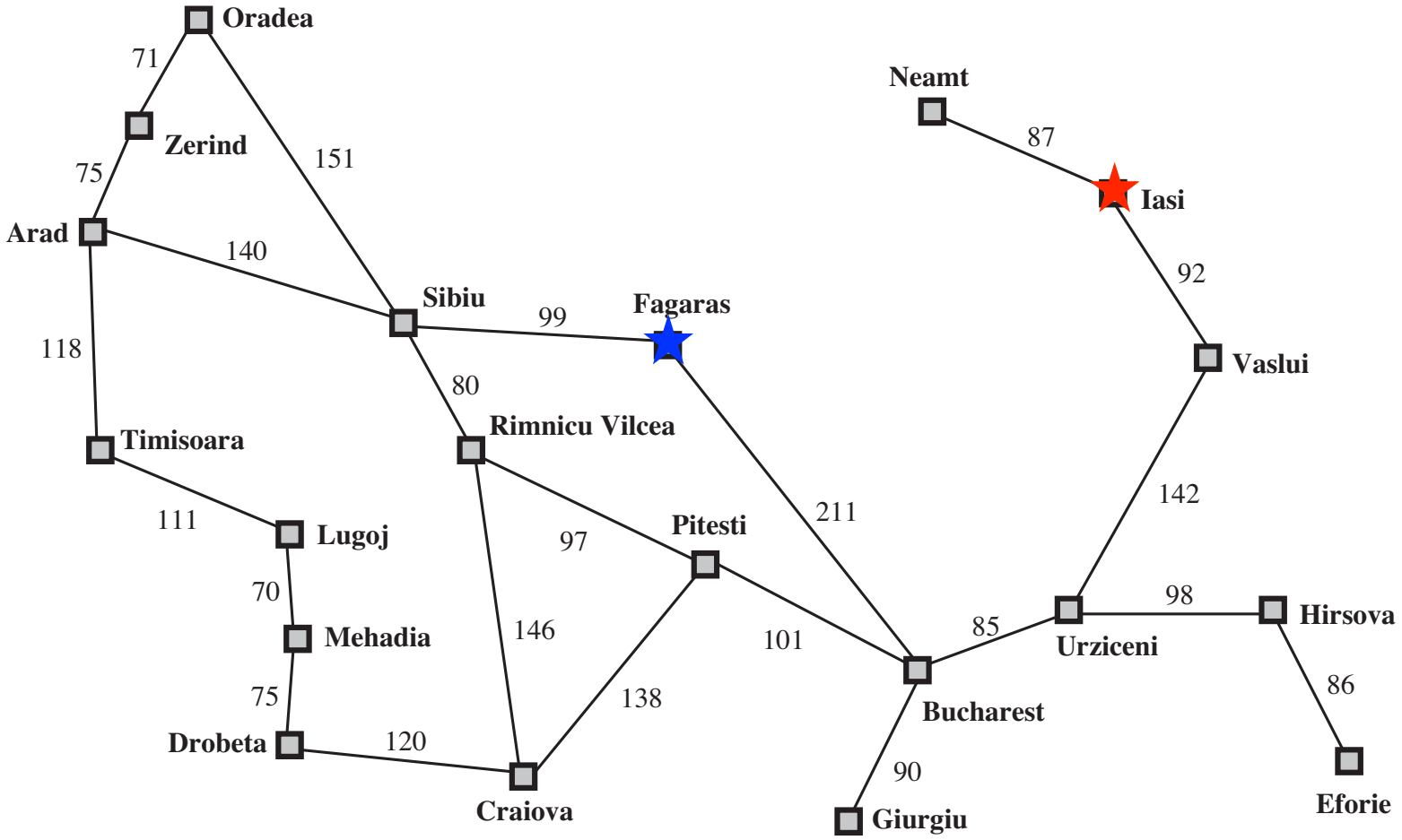
Greedy best-first search example: After expanding Fagaras



Measuring performance of greedy best-first search

- *Completeness*: No – can get stuck in loops, *e.g.*
  Iasi → Neamt → Iasi → Neamt →
- *Optimality*: No, since there might be a shorter path
- *Time complexity*:
- *Space complexity*:

Traveling in Hungary



Measuring performance of greedy best-first search

- *Completeness*: No – can get stuck in loops, *e.g.*
  Iasi → Neamt → Iasi → Neamt →
- *Optimality*: No, since there might be a shorter path
  Arad → Sibiu→ Rimnicu Vilscea→ Pitesti→ Bucharest is 32 km shorter
- *Time complexity*:
- *Space complexity*:

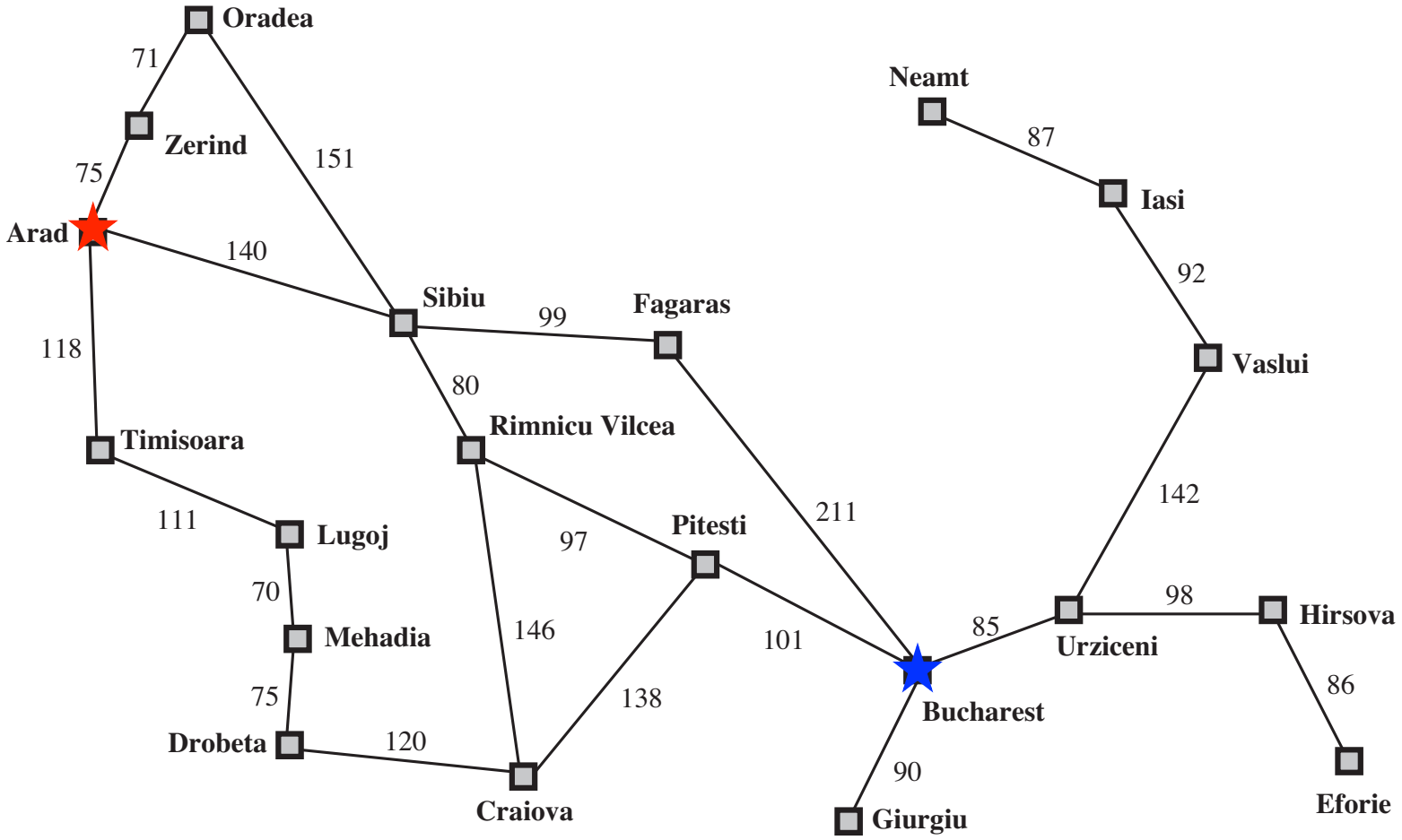
Measuring performance of greedy best-first search

- *Completeness*: No – can get stuck in loops, *e.g.*
  Iasi → Neamt → Iasi → Neamt →
- *Optimality*: No, since there might be a shorter path
  Arad → Sibiu→ Rimnicu Vilscea→ Pitesti→ Bucharest is 32 km shorter
- *Time complexity*: $O(b^m)$ — *might be big, but a good heuristic can give dramatic improvement*
- *Space complexity*: $O(b^m)$ — *keeps all nodes in memory*

A* search: Ideas

- Most widely known form of informed search
- A* tries to avoid expanding paths that are already too expensive
- Evaluation function $f(n) = g(n) + h(n)$
  - $g(n)$ = path cost from the start node to node n
  - $h(n)$ = estimated cost of the cheapest path from n to goal
  - $f(n)$ = estimated total cost of path through n to goal
- Algorithm is identical to Uniform-Cost-Search, except using $g(n) + h(n)$ instead of $g(n)$

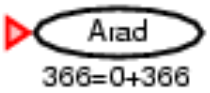
Traveling in Hungary



Romania with step costs in km: Straight line distance to Bucharest

Arad	366	**Mehadia**	241
Bucharest	0	**Neamt**	234
Craiova	160	**Oradea**	380
Drobeta	242	**Pitesti**	100
Eforie	161	**Rimnicu Vilcea**	193
Fagaras	176	**Sibiu**	253
Giurgiu	77	**Timisoara**	329
Hirsova	151	**Urziceni**	80
Iasi	226	**Vaslui**	199
Lugoj	244	**Zerind**	374

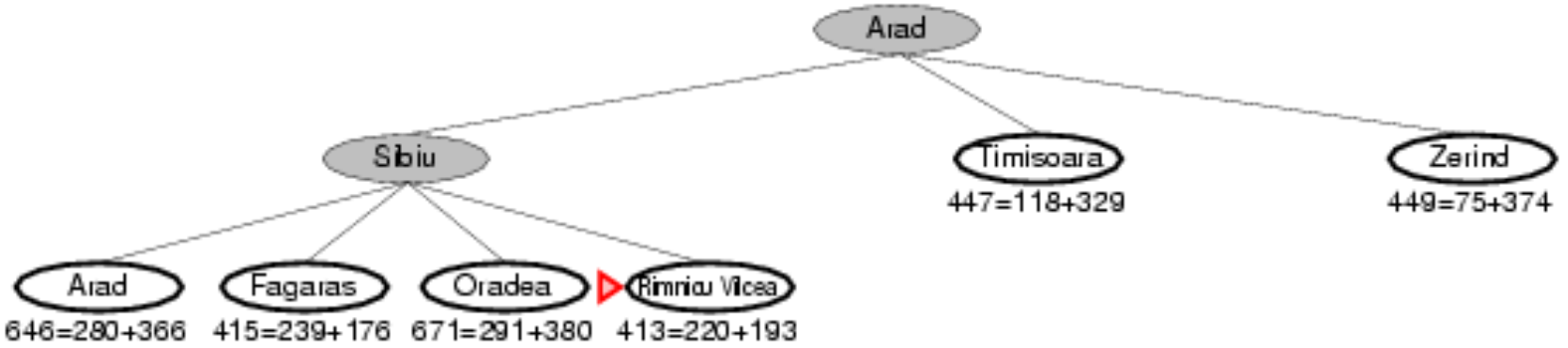
A* search example: Traveling in Hungary



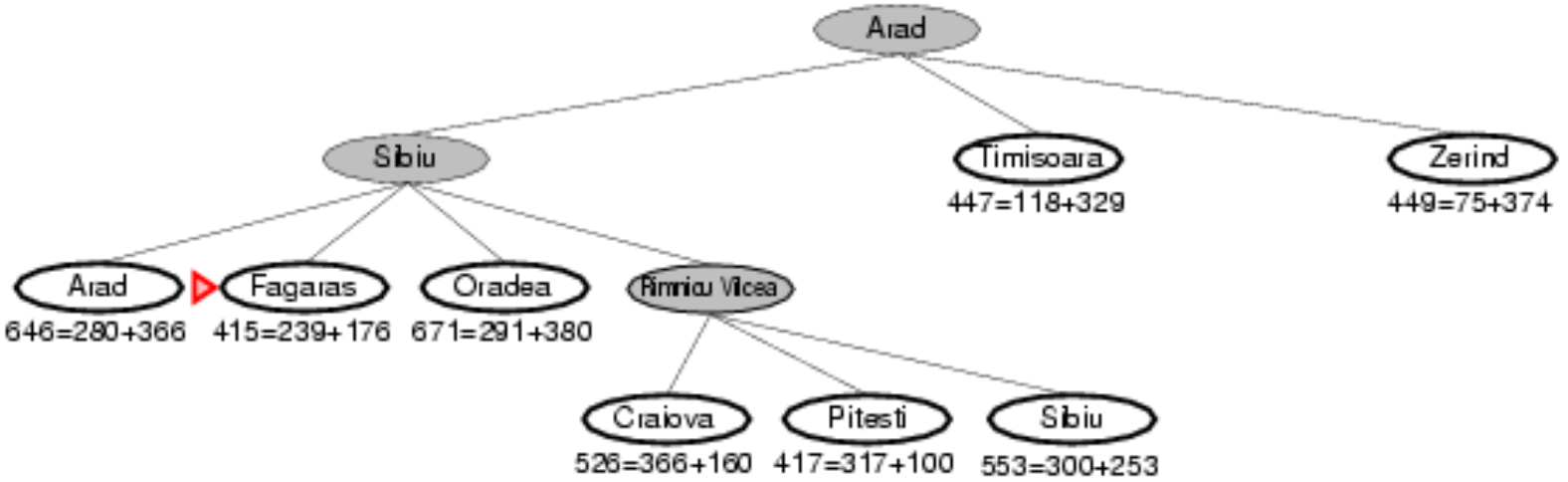
A* search example: Traveling in Hungary

A* search example: Expanding Arand

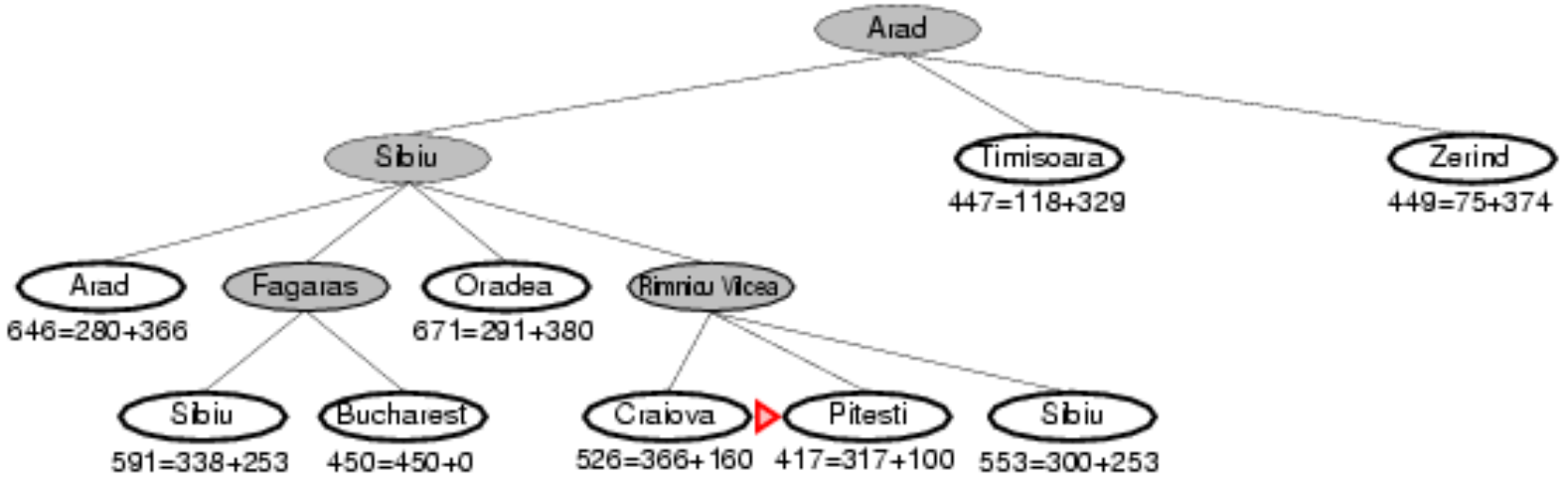
A* search example: Expanding Sibiu



A* search example: Expanding Rimnicu Vilcea



A* search example: Expanding Fagaras



A* search example: Expanding Pitesti