CHAPTER 3: CLASSICAL SEARCH ALGORITHMS

DIT410/TIN174, Artificial Intelligence

Peter Ljunglöf

24 March, 2017

TABLE OF CONTENTS

INTRODUCTION (R&N 3.1–3.3)

GRAPHS AND SEARCHING

EXAMPLE PROBLEMS

A GENERIC SEARCHING ALGORITHM

GRAPHS AND SEARCHING

Often we are not given an algorithm to solve a problem, but only a specification of a solution — we have to search for it.

A typical problem is when the agent is in one state, it has a set of deterministic actions it can carry out, and wants to get to a goal state.

Many AI problems can be abstracted into the problem of finding a path in a directed graph.

Often there is more than one way to represent a problem as a graph.

STATE-SPACE SEARCH: COMPLEXITY DIMENSIONS

Observable?	fully
Deterministic?	deterministic
Episodic?	episodic
Static?	static
Discrete?	discrete
N:o of agents	single

Most complex problems (partly observable, stochastic, sequential) usualy have components using state-space search.

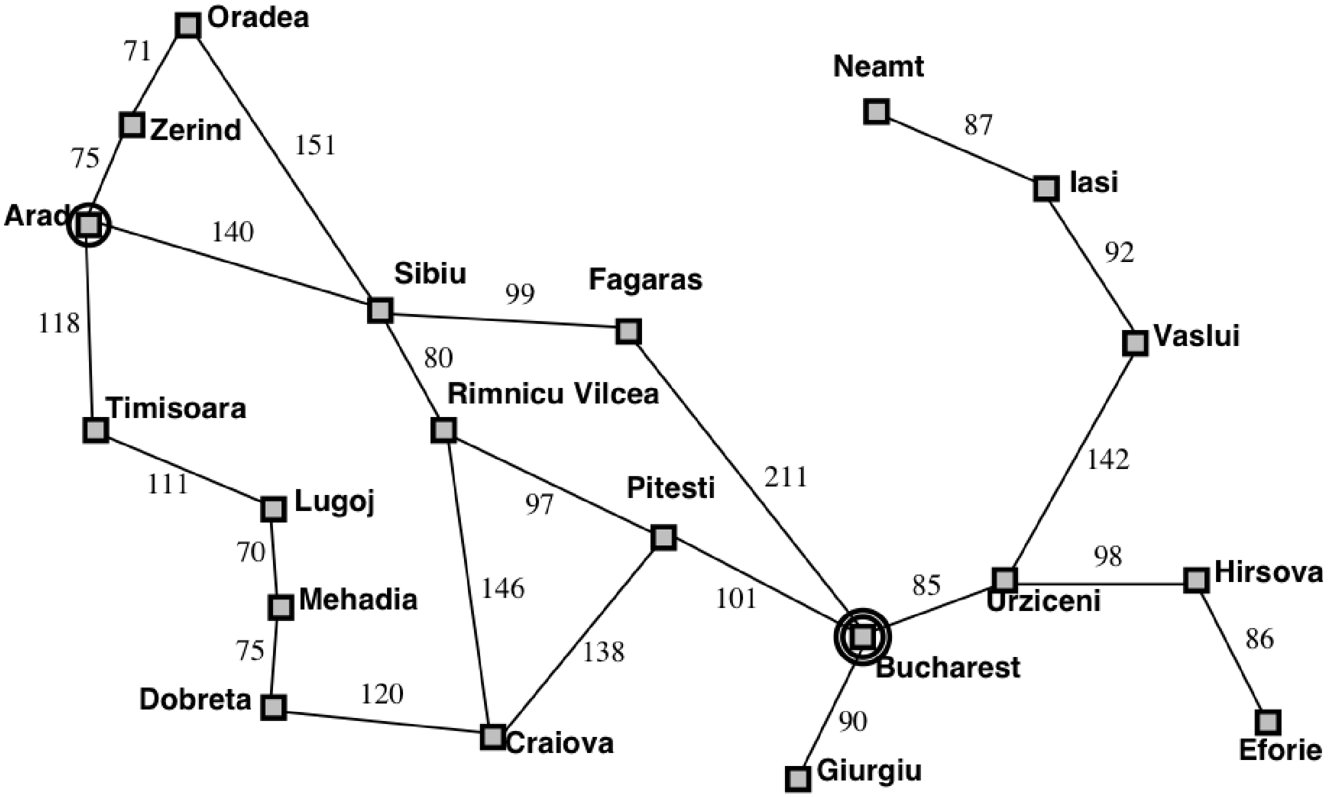
DIRECTED GRAPHS

A *graph* consists of a set N of *nodes* and a set A of ordered pairs of nodes, called *arcs* or *edges*.

- Node n_2 is a *neighbor* of n_1 if there is an arc from n_1 to n_2. That is, if $(n_1, n_2) \in A$.
- A *path* is a sequence of nodes $(n_0, n_1, \ldots, n_k)$ such that $(n_{i-1}, n_i) \in A$.
- The *length* of path $(n_0, n_1, \ldots, n_k)$ is k.
- A *solution* is a path from a start node to a goal node, given a set of *start nodes* and *goal nodes*.
- (Russel & Norvig sometimes call the graph nodes *states*).

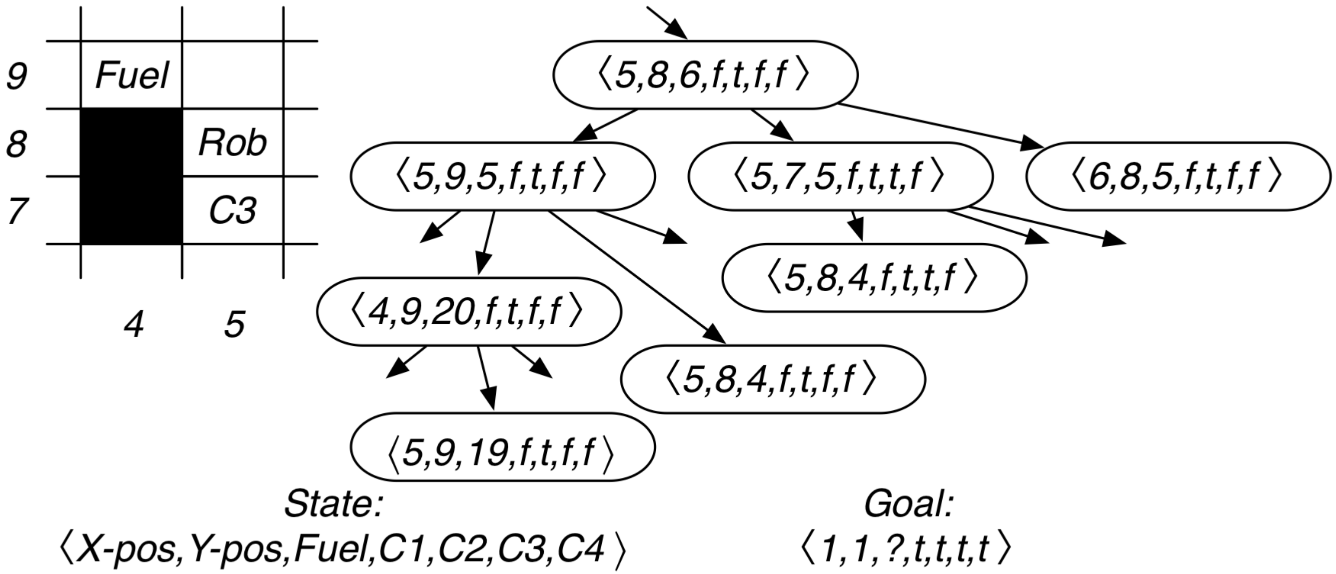
EXAMPLE: TRAVEL IN ROMANIA

We want to drive from Arad to Bucharest in Romania



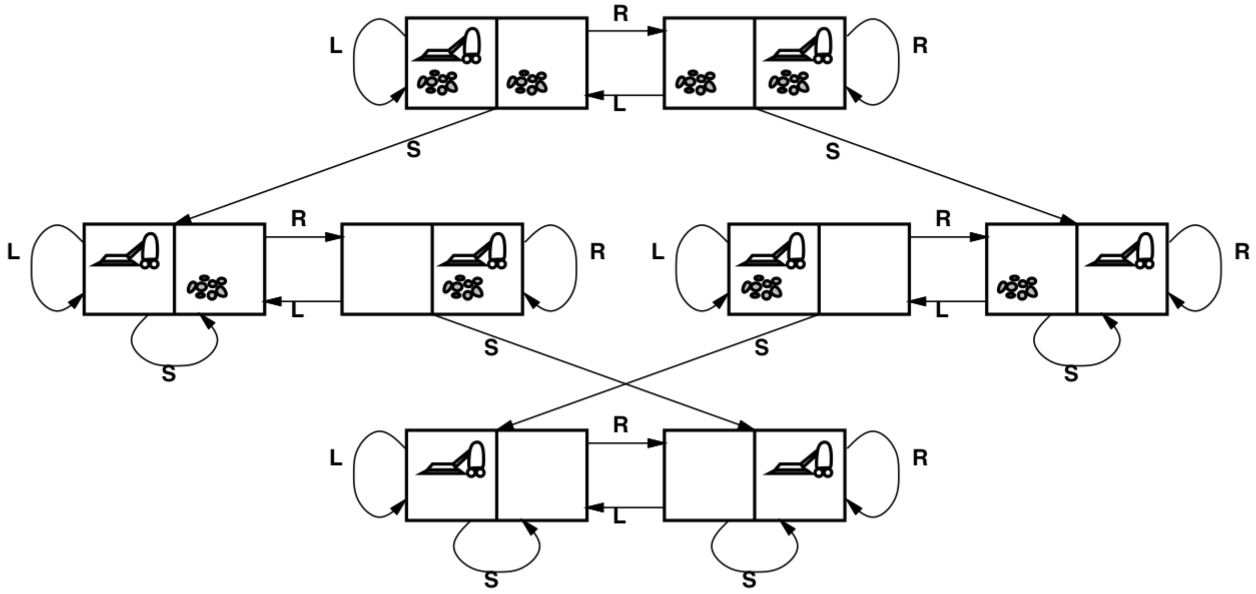
EXAMPLE: GRID GAME

Grid game: Rob needs to collect coins C_1, C_2, C_3, C_4, without running out of fuel, and end up at location (1,1):



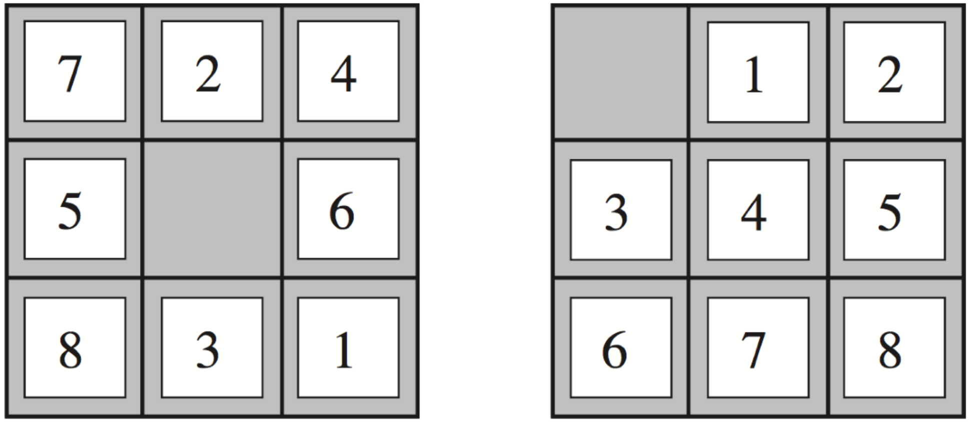
What is a good representation of the *search states* and the *goal*?

EXAMPLE: VACUUM-CLEANING AGENT



States	*[room A dirty?, room B dirty?, robot location]*
Initial state	*any state*
Actions	*left, right, suck, do-nothing*
Goal test	*[false, false, –]*
Path cost	*1 per action (0 for do-nothing)*

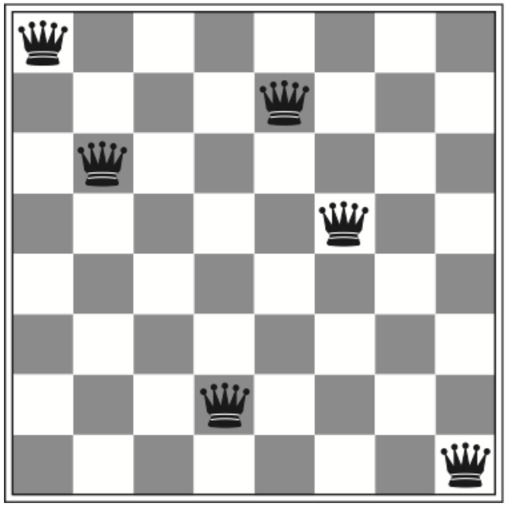
EXAMPLE: THE 8-PUZZLE



Start State　　　Goal State

States	*a 3 x 3 matrix of integers*
Initial state	*any state*
Actions	*move the blank space: left, right, up, down*
Goal test	*equal to the goal state*
Path cost	*1 action (0 for do-nothing)*

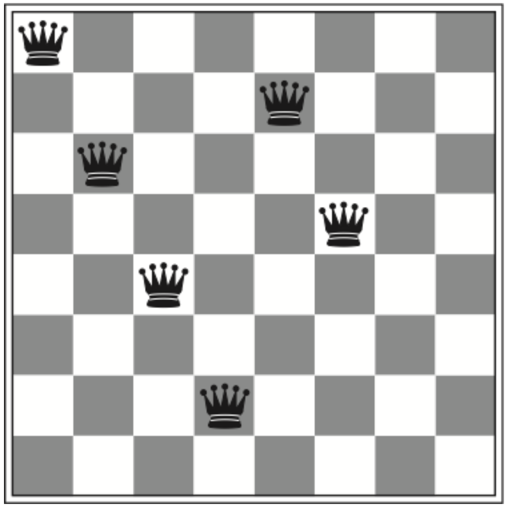
EXAMPLE: THE 8-QUEENS PROBLEM



States	*any arrangement of 0 to 8 queens on the board*
Initial state	*no queens on the board*
Actions	*add a queen to any empty square*
Goal test	*8 queens on the board, none attacked*
Path cost	*1 per move*

This gives us $64 \times 63 \times \cdots \times 57 \approx 1.8 \times 10^{14}$ possible paths to explore!

EXAMPLE: THE 8-QUEENS PROBLEM (ALTERNATIVE)



States	*one queen per column in leftmost columns, none attacked*
Initial state	*no queens on the board*
Actions	*add a queen to a square in the leftmost empty column, make sure that no queen is attacked*
Goal test	*8 queens on the board, none attacked*
Path cost	*1 per move*

Using this formulation, we have only 2,057 paths!

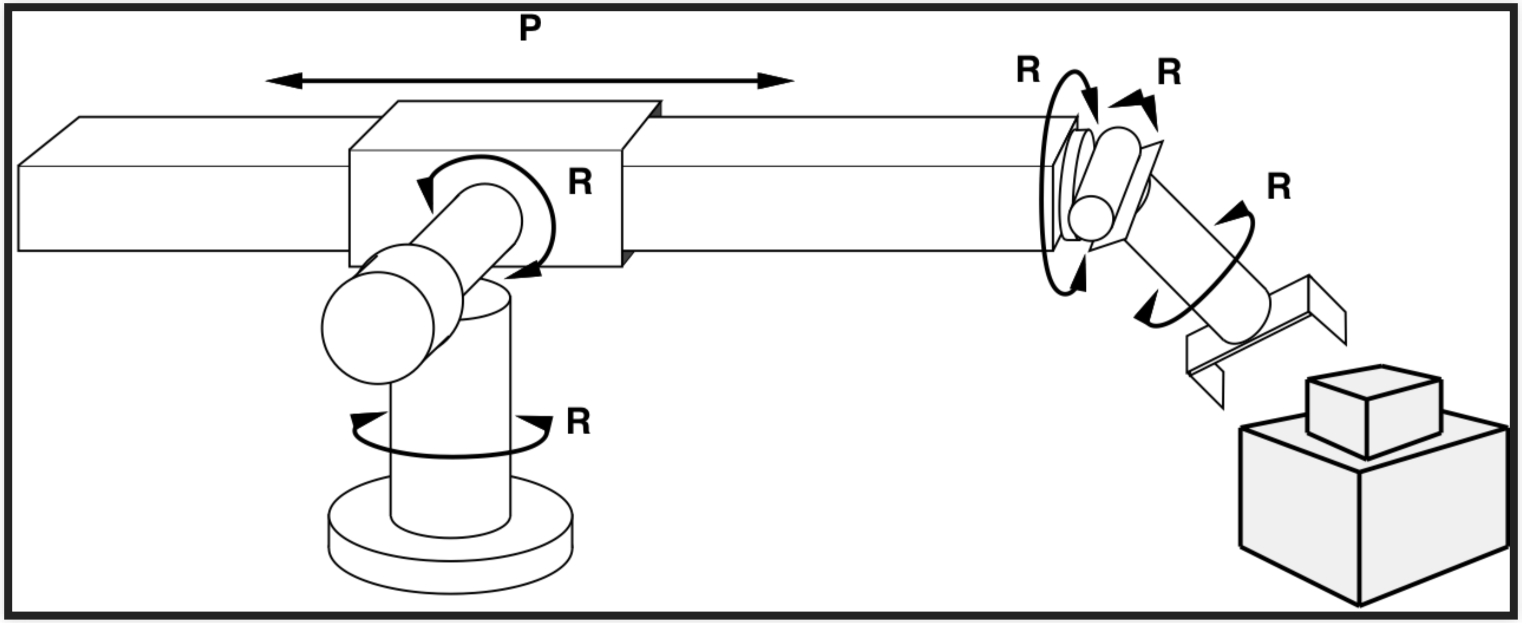
EXAMPLE: KNUTH'S CONJECTURE

Donald Knuth conjectured that all positive integers can be obtained by starting with the number 4 and applying some combination of the factorial, square root, and floor.

$$\left\lfloor \sqrt{\sqrt{\sqrt{\sqrt{\sqrt{(4!)!}}}}} \right\rfloor = 5$$

States	*algebraic numbers* $(1, 2.5, 9, \sqrt{2}, 1.23 \cdot 10^{456}, \sqrt{\sqrt{2}}, \ldots)$
Initial state	*4*
Actions	*apply factorial, square root, or floor operation*
Goal test	*any positive integer (e.g., 5)*
Path cost	*1 per move*

EXAMPLE: ROBOTIC ASSEMBLY



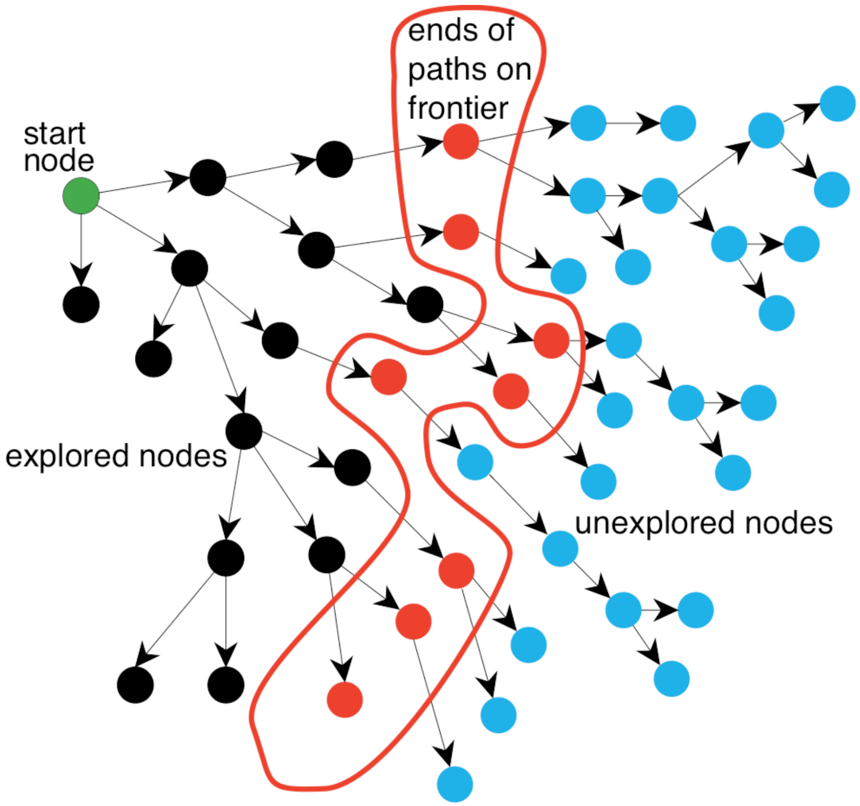
States	*real-valued coordinates of robot joint angles parts of the object to be assembled*
Actions	*continuous motions of robot joints*
Goal test	*complete assembly of the object*
Path cost	*time to execute*

HOW DO WE SEARCH IN A GRAPH?

A generic search algorithm:

- Given a graph, start nodes, and a goal description, incrementally explore paths from the start nodes.
- Maintain a *frontier* of nodes that are to be explored.
- As search proceeds, the frontier expands into the unexplored nodes until a goal node is encountered.
- The way in which the frontier is expanded defines the search strategy.

ILLUSTRATION OF GENERIC SEARCH



A GENERIC TREE SEARCH ALGORITHM

Tree search: Don't check if nodes are visited multiple times

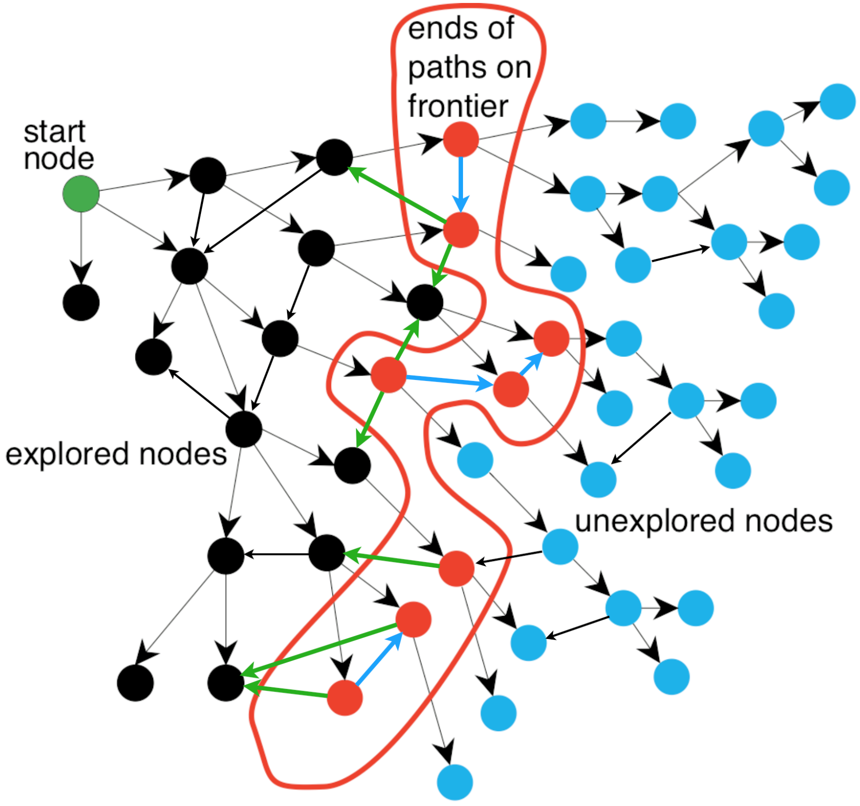
```
function Search(graph, initialState, goalState):
    initialise frontier using the initialState

    while frontier is not empty:
        select and remove node from frontier
        if node.state is a goalState then return node

        for each child in ExpandChildNodes(node, graph):
            add child to frontier
    return failure
```

TURNING TREE SEARCH INTO GRAPH SEARCH

Graph search: Keep track of visited nodes

```
function Search(graph, initialState, goalState):
    initialise frontier using the initialState
    initialise exploredSet to the empty set
    while frontier is not empty:
        select and remove node from frontier
        if node.state is a goalState then return node
        add node to exploredSet
        for each child in ExpandChildNodes(node, graph):
            add child to frontier if child is not in frontier or exploredSet
    return failure
```

USING TREE SEARCH ON A GRAPH



- explored nodes might be revisited
- frontier nodes might be duplicated

TREE SEARCH VS. GRAPH SEARCH

Tree search

- **Pro**: uses less memory
- **Con**: might visit the same node several times

Graph search

- **Pro**: only visits nodes at most once
- **Con**: uses more memory

Note: The pseudocode in these slides (and the course book) is not the only possible! E.g., Wikipedia uses a different variant.

GRAPH NODES VS. SEARCH NODES

Search nodes are not the same as graph nodes!

Search nodes should contain more information:

- the corresponding graph node (called state in R&N)
- the total path cost from the start node
- the estimated (heuristic) cost to the goal
- enough information to be able to calculate the final path

```
procedure ExpandChildNodes(parent, graph):
    for each (action, child, edgecost) in graph.successors(parent.state):
        yield new SearchNode(child,
                             …total cost so far…,
                             …estimated cost to goal…,
                             …information for calculating final path…)
```

UNINFORMED SEARCH (R&N 3.4)

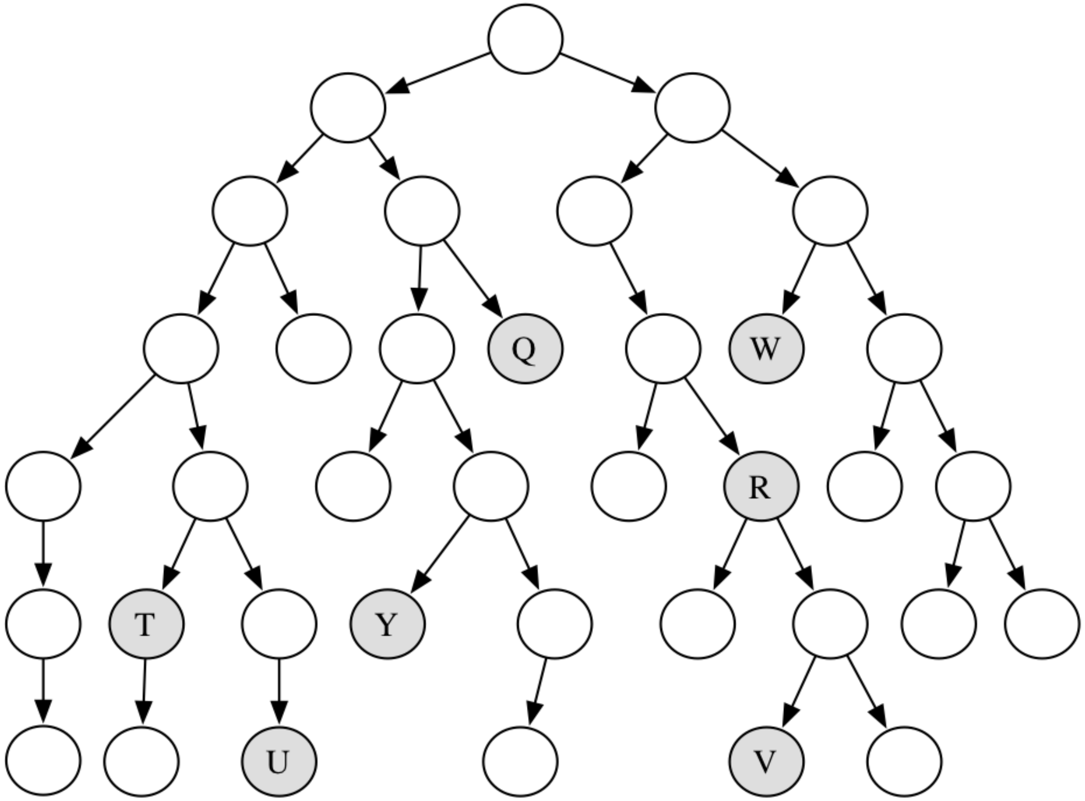
DEPTH-FIRST SEARCH

BREADTH-FIRST SEARCH

UNIFORM-COST SEARCH

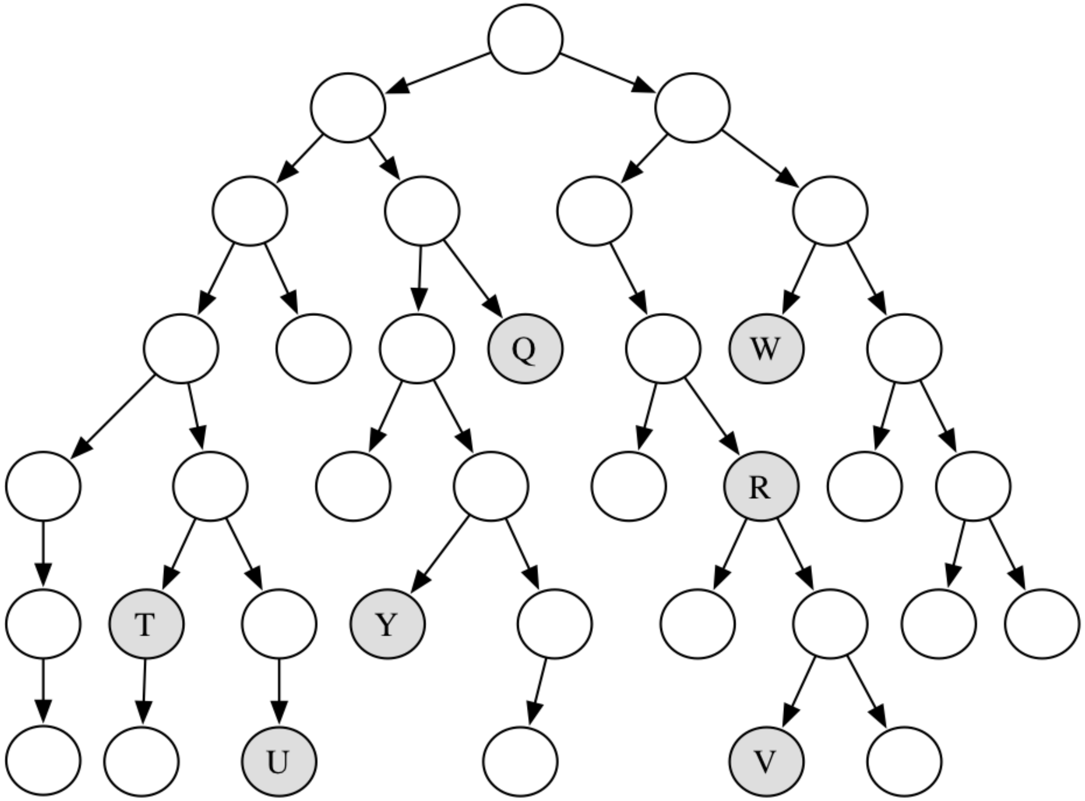
QUESTION TIME: DEPTH-FIRST SEARCH

Which shaded goal will a depth-first search find first?



QUESTION TIME: BREADTH-FIRST SEARCH

Which shaded goal will a breadth-first search find first?



DEPTH-FIRST SEARCH

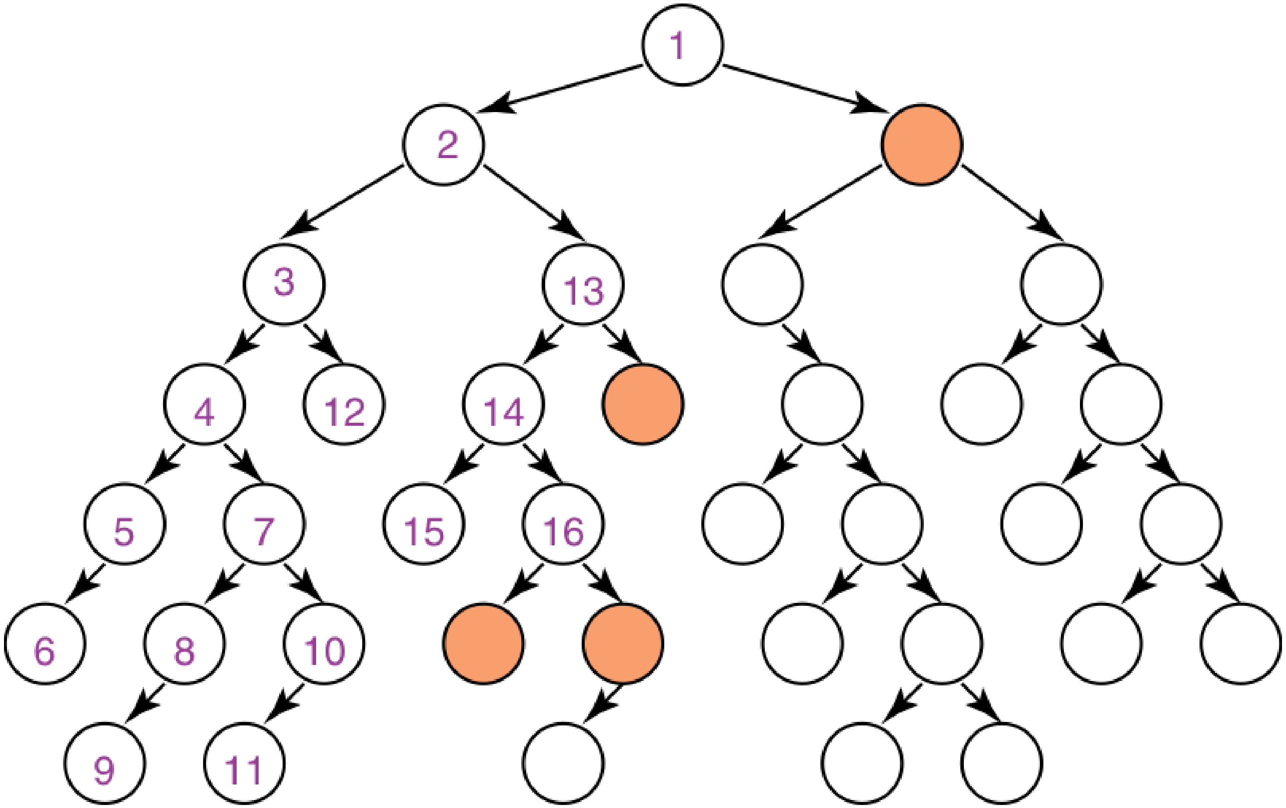
Depth-first search treats the frontier as a stack.

It always selects one of the last elements added to the frontier.

If the list of nodes on the frontier is $[p_1, p_2, p_3, \ldots]$, then:

- p_1 is selected (and removed).
- Nodes that extend p_1 are added to the front of the stack (in front of p_2).
- p_2 is only selected when all nodes from p_1 have been explored.

ILLUSTRATIVE GRAPH: DEPTH-FIRST SEARCH



COMPLEXITY OF DEPTH-FIRST SEARCH

Does DFS guarantee to find the path with fewest arcs?

What happens on infinite graphs or on graphs with cycles if there is a solution?

What is the time complexity as a function of the path length?

What is the space complexity as a function of the path length?

How does the goal affect the search?

BREADTH-FIRST SEARCH

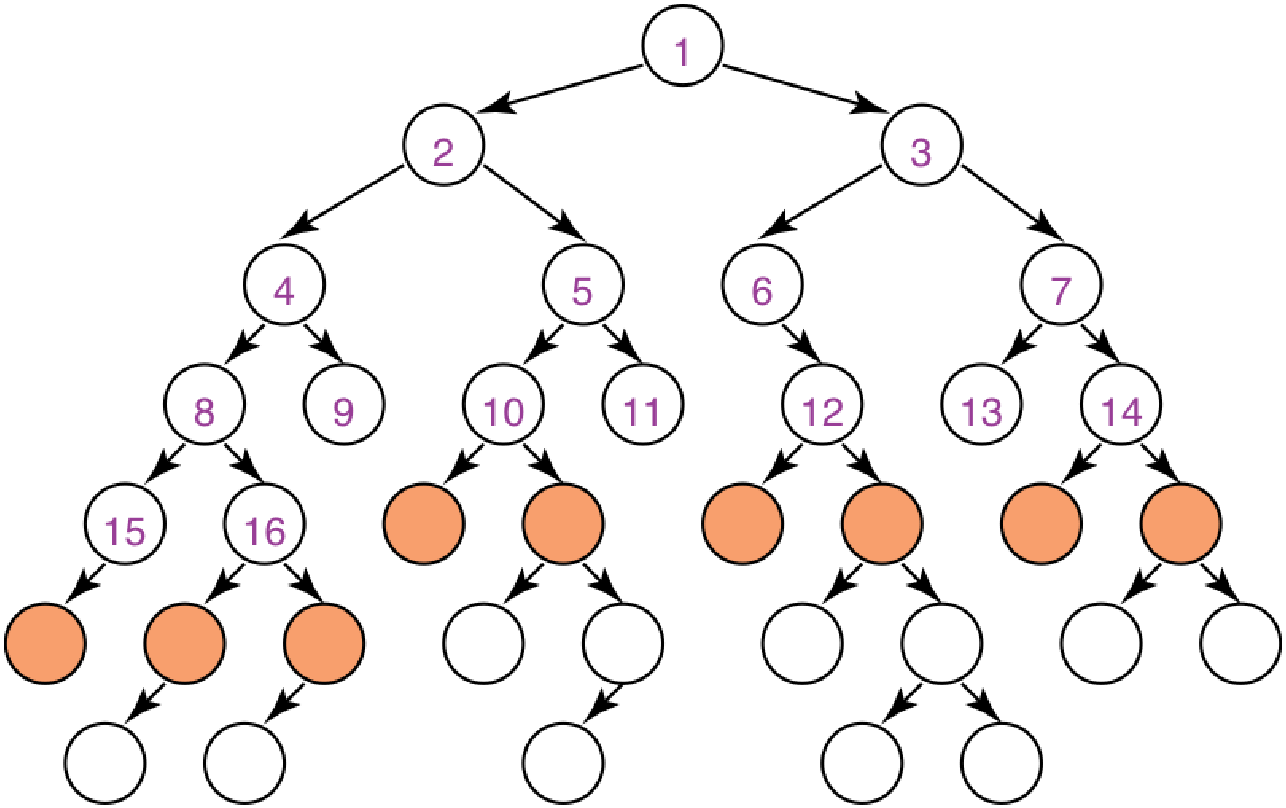
Breadth-first search treats the frontier as a queue.

It always selects one of the earliest elements added to the frontier.

If the list of paths on the frontier is $[p_1, p_2, \ldots, p_r]$, then:

- p_1 is selected (and removed).
- Its neighbors are added to the end of the queue, after p_r.
- p_2 is selected next.

ILLUSTRATIVE GRAPH: BREADTH-FIRST SEARCH



COMPLEXITY OF BREADTH-FIRST SEARCH

Does BFS guarantee to find the path with fewest arcs?

What happens on infinite graphs or on graphs with cycles if there is a solution?

What is the time complexity as a function of the path length?

What is the space complexity as a function of the path length?

How does the goal affect the search?

UNIFORM-COST SEARCH

Weighted graphs:

- Sometimes there are *costs* associated with arcs.
  The cost of a path is the sum of the costs of its arcs.

  $$cost(n_0, \ldots, n_k) = \sum_{i=1}^{k} |(n_{i-1}, n_i)|$$

  An *optimal solution* is one with minimum cost.

Uniform-cost search:

- Uniform-cost search selects a path on the frontier with the lowest cost.
- The frontier is a *priority queue* ordered by path cost.
- It finds a least-cost path to a goal node — i.e., uniform-cost search is optimal
- When arc costs are equal $\Rightarrow$ breadth-first search.

HEURISTIC SEARCH (R&N 3.5–3.6)

GREEDY BEST-FIRST SEARCH

A* SEARCH

ADMISSIBLE AND CONSISTENT HEURISTICS

HEURISTIC SEARCH

Previous methods don't use the goal to select a path to explore.

Main idea: don't ignore the goal when selecting paths.

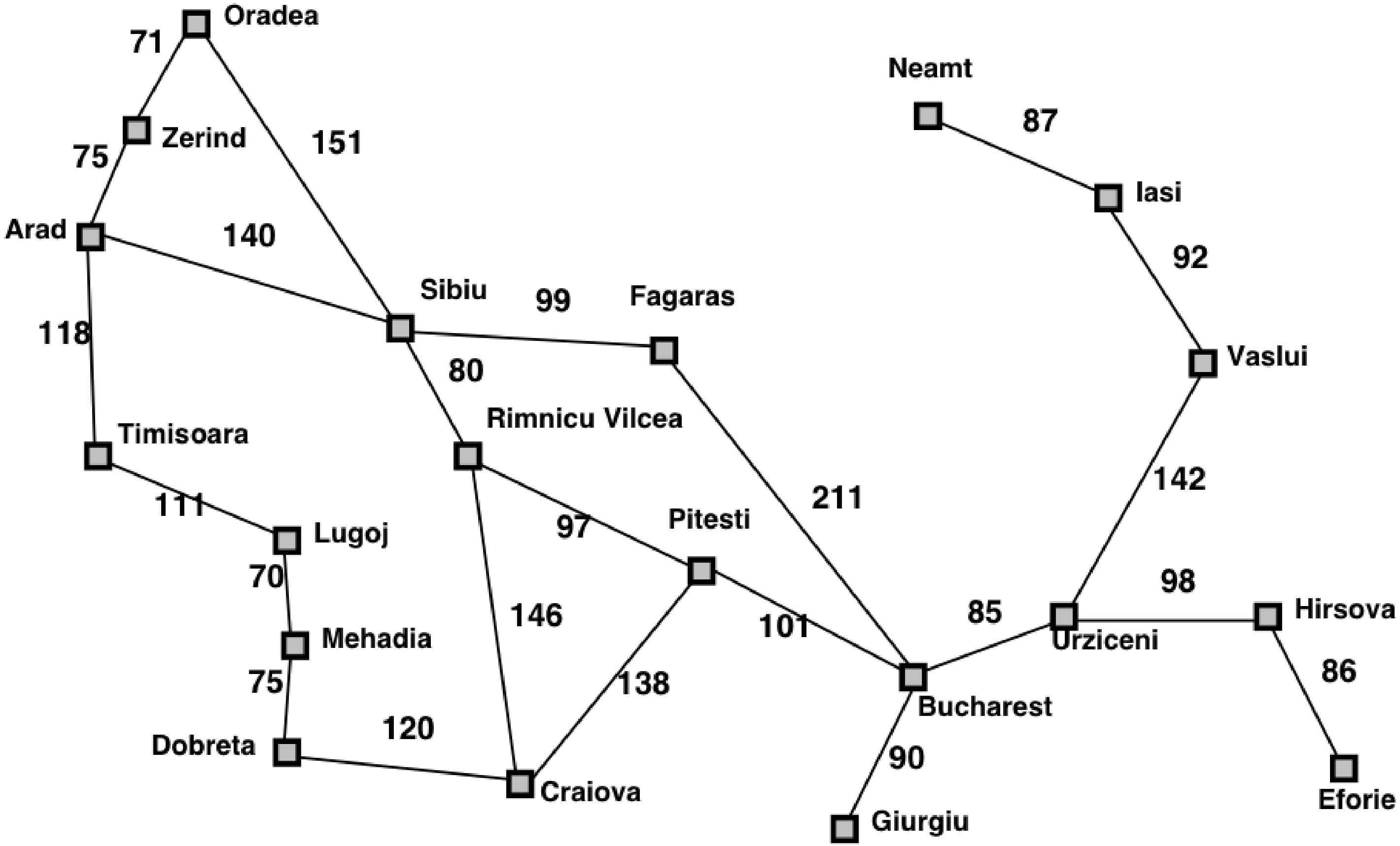
- Often there is extra knowledge that can guide the search: ***heuristics***.
- $h(n)$ is an estimate of the cost of the shortest path from node n to a goal node.
- $h(n)$ needs to be efficient to compute.
- $h(n)$ is an *underestimate* if there is no path from n to a goal with cost less than $h(n)$.
- An *admissible heuristic* is a nonnegative underestimating heuristic function: $0 \leq h(n) \leq cost(n, goal)$

EXAMPLE HEURISTIC FUNCTIONS

Here are some example heuristic functions:

- If the nodes are points on a Euclidean plane and the cost is the distance, $h(n)$ can be the straight-line distance (SLD) from n to the closest goal.
- If the nodes are locations and cost is time, we can use the distance to a goal divided by the maximum speed, $h(n) = d(n)/v_{\max}$ (or the average speed, $h(n) = d(n)/v_{\text{avg}}$, which makes it non-admissible).
- If the goal is to collect all of the coins and not run out of fuel, we can use an estimate of how many steps it will take to collect the coins and return to goal position, without caring about the fuel consumption.
- A heuristic function can be found by solving a simpler (less constrained) version of the problem.

EXAMPLE HEURISTIC: ROMANIA DISTANCES



Straight−line distance
to Bucharest

Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

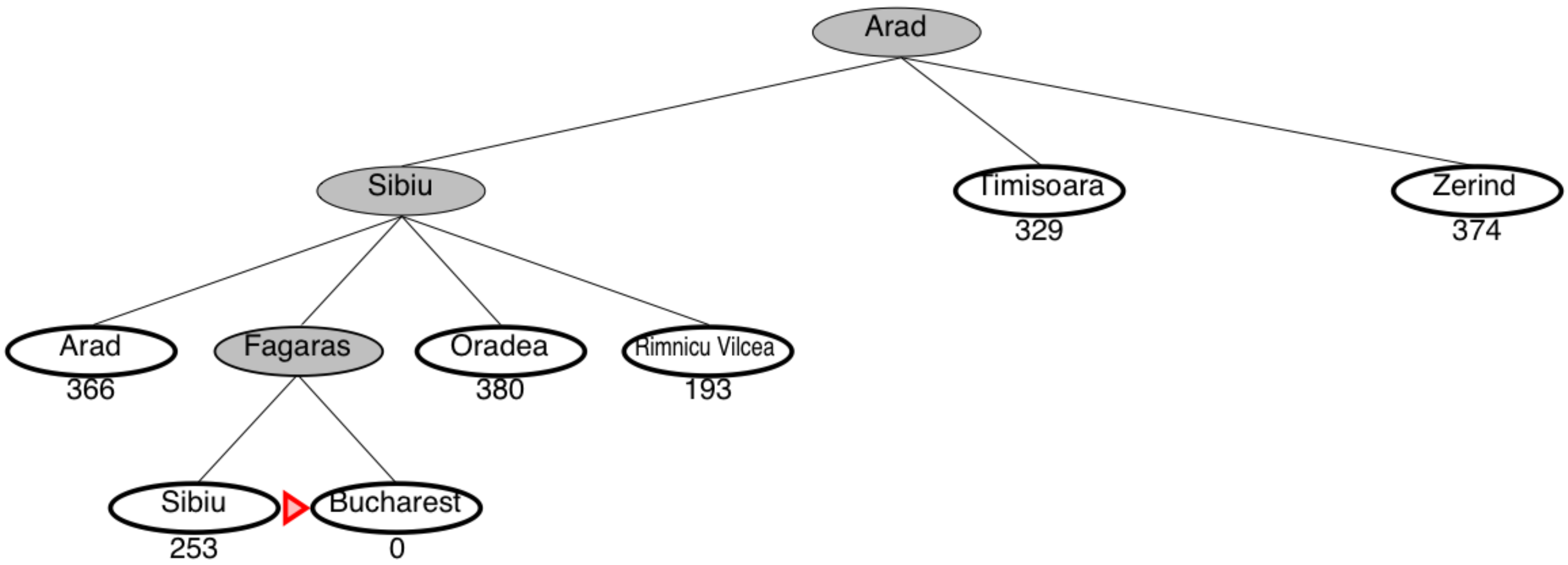
GREEDY BEST-FIRST SEARCH

Main idea: select the path whose end is closest to a goal according to the heuristic function.

Best-first search selects a path on the frontier with minimal h-value.

It treats the frontier as a priority queue ordered by h.

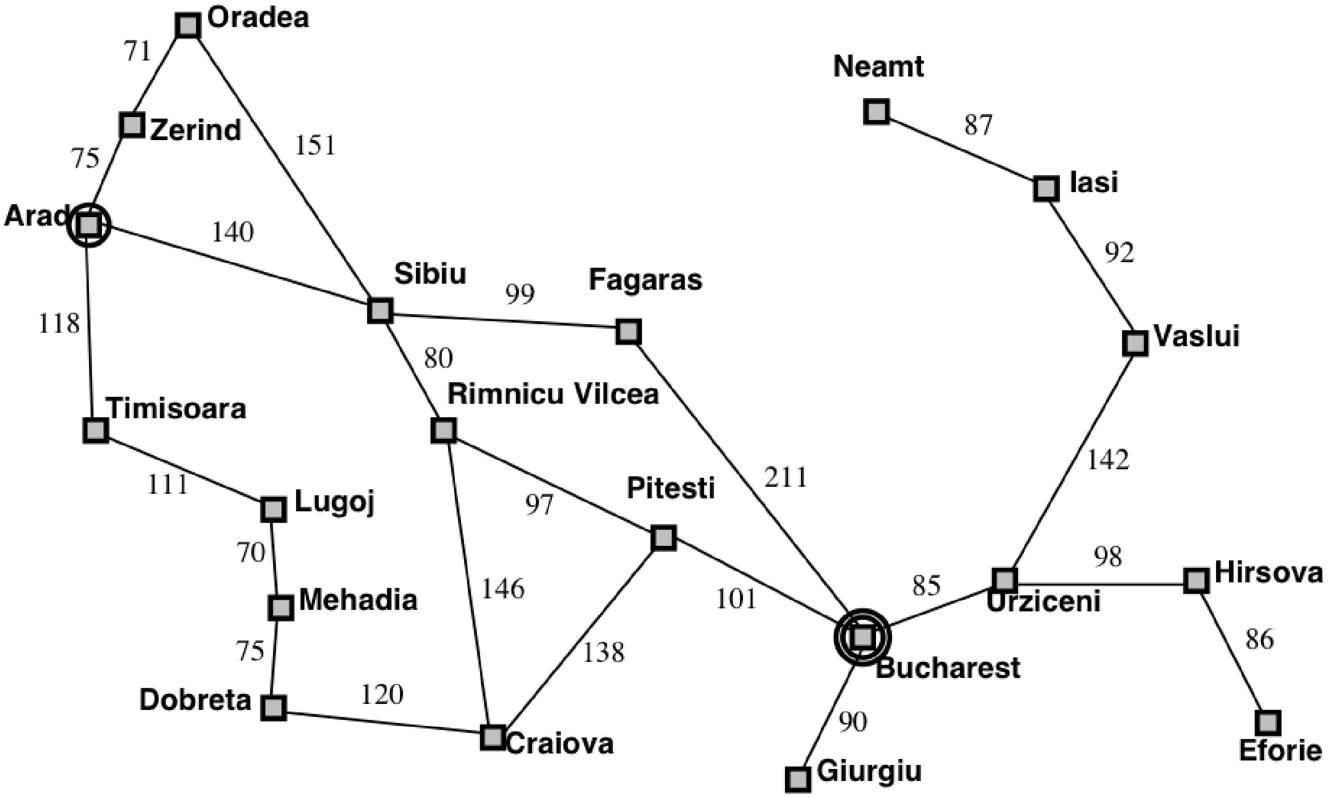
GREEDY SEARCH EXAMPLE: ROMANIA



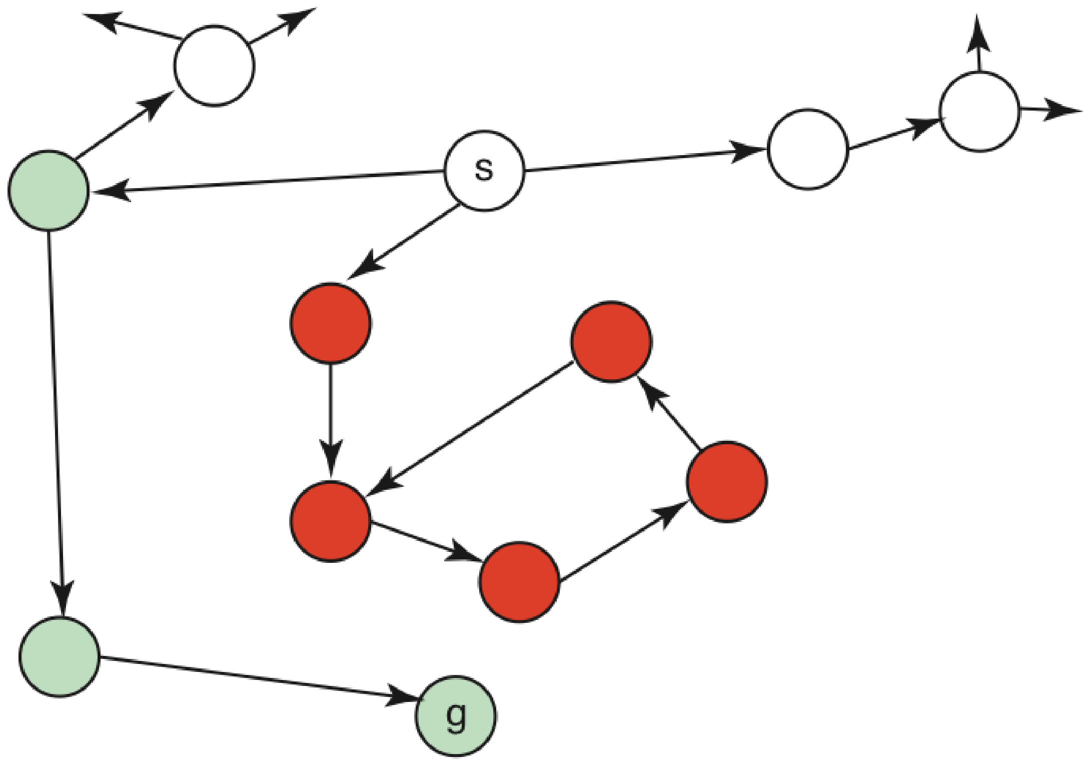
This is not the shortest path!

GREEDY SEARCH IS NOT OPTIMAL

Greedy search returns the path: *Arad–Sibiu–Fagaras–Bucharest* (450km)
The optimal path is: *Arad–Sibiu–Rimnicu–Pitesti–Bucharest* (418km)



BEST-FIRST SEARCH AND INFINITE LOOPS



Best-first search might fall into an infinite loop!

COMPLEXITY OF BEST-FIRST SEARCH

Does best-first search guarantee to find the path with fewest arcs?

What happens on infinite graphs or on graphs with cycles if there is a solution?

What is the time complexity as a function of the path length?

What is the space complexity as a function of the path length?

How does the goal affect the search?

A* SEARCH

A* search uses both path cost and heuristic values.

$cost(p)$ is the cost of path p.

$h(p)$ estimates the cost from the end node of p to a goal.

$f(p) = cost(p) + h(p)$, estimates the total path cost of going from the start node, via path p to a goal:

$$\underbrace{\underbrace{start \xrightarrow{\text{path } p} n}_{cost(p)} \underbrace{\xrightarrow{\text{estimate}} goal}_{h(p)}}_{f(p)}$$

A* SEARCH

A* is a mix of lowest-cost-first and best-first search.

It treats the frontier as a priority queue ordered by $f(p)$.

It always selects the node on the frontier with the lowest estimated distance from the start to a goal node constrained to go via that node.

COMPLEXITY OF A* SEARCH

Does A* search guarantee to find the path with fewest arcs?

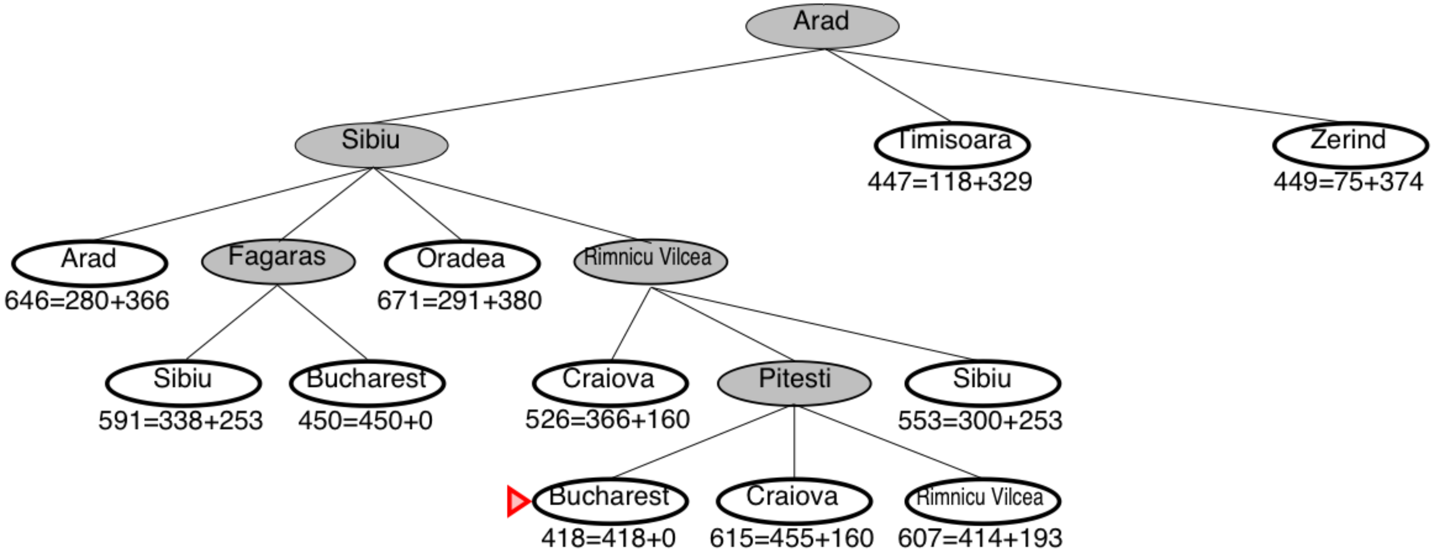
What happens on infinite graphs or on graphs with cycles if there is a solution?

What is the time complexity as a function of the path length?

What is the space complexity as a function of the path length?

How does the goal affect the search?

A* SEARCH EXAMPLE: ROMANIA

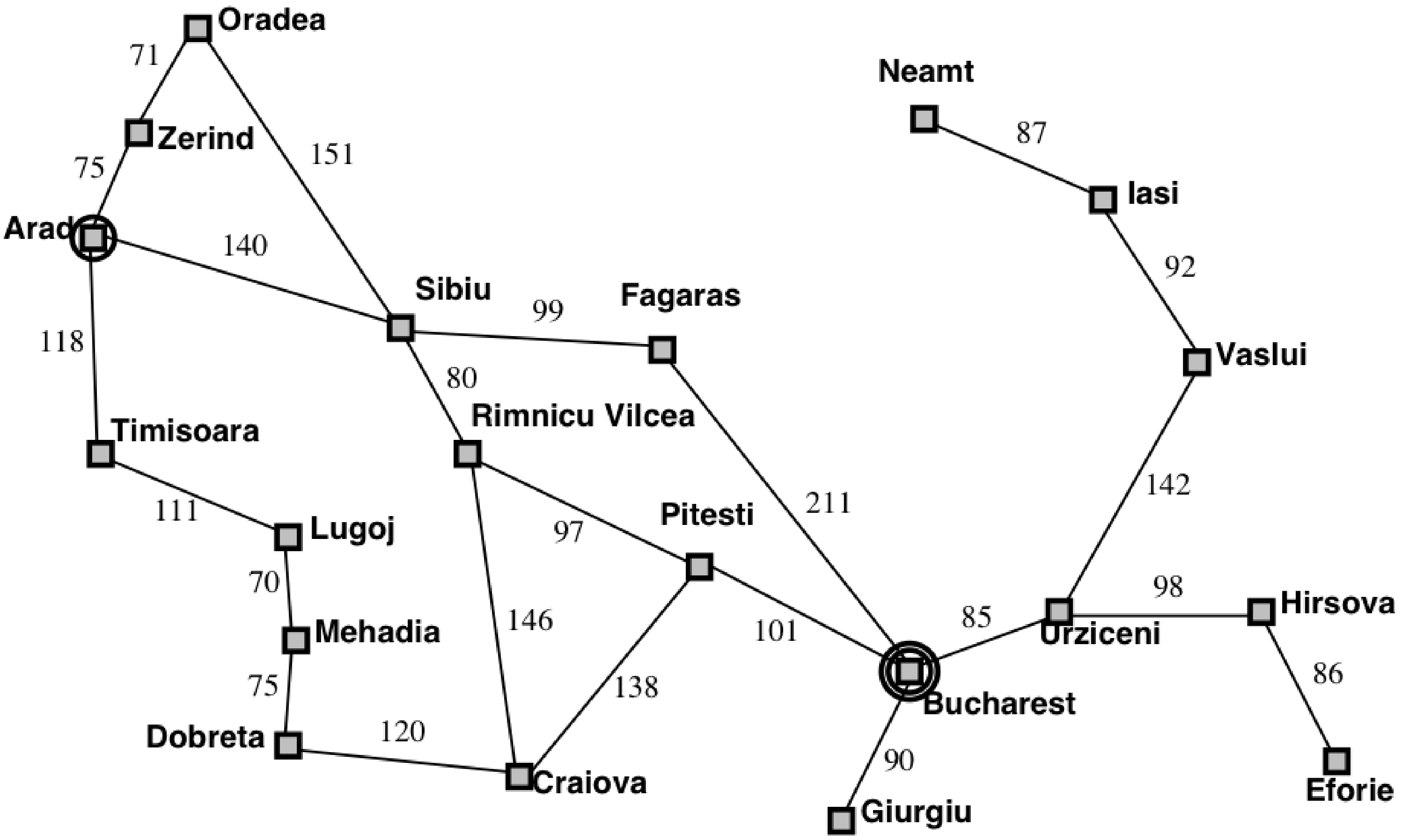


A guarantees that this is the shortest path!*

Note that we didn't select the first occurrence of Bucharest

A* SEARCH IS OPTIMAL

The optimal path is: *Arad–Sibiu–Rimnicu–Pitesti–Bucharest* (418km)



A* ALWAYS FINDS A SOLUTION

A* will always find a solution if there is one, because:

- The frontier always contains the initial part of a path to a goal, before that goal is selected.
- A* halts, because the costs of the paths on the frontier keeps increasing, and will eventually exceed any finite number.

ADMISSIBILITY (OPTIMALITY) OF A*

If there is a solution, A* always finds an optimal one first, provided that:

- the branching factor is finite,
- arc costs are bounded above zero
  (i.e., there is some $\epsilon > 0$ such that all
  of the arc costs are greater than ϵ), and
- $h(n)$ is nonnegative and an underestimate of
  the cost of the shortest path from n to a goal node.

WHY IS A* OPTIMAL?

The f values in A* are nondecreasing, therefore:

first	A* expands all nodes with $f(n) < C$
then	A* expands all nodes with $f(n) = C$
finally	A* expands all nodes with $f(n) > C$

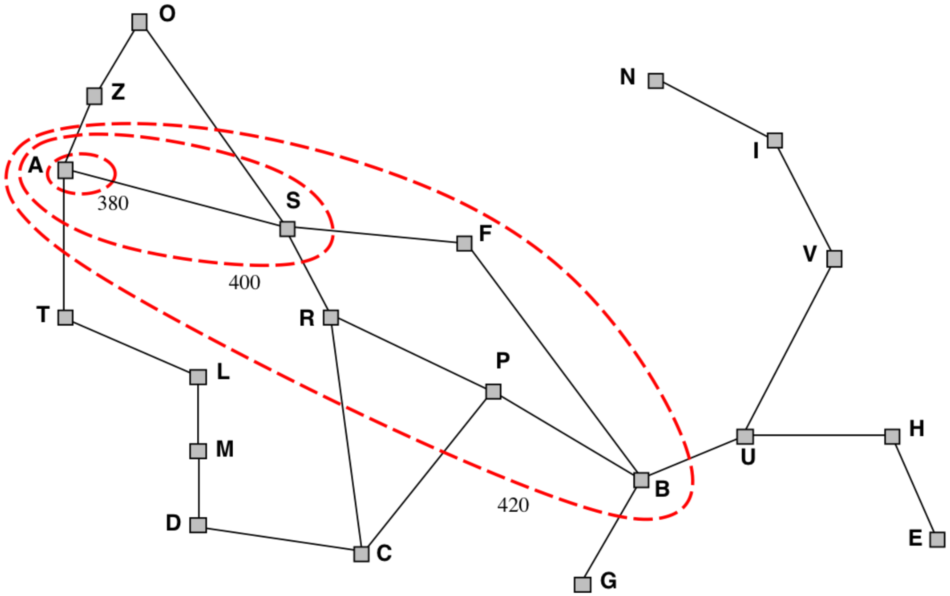
A* will not expand any nodes with $f(n) > C*$,
where $C*$ is the cost of an optimal solution.

(***Note***: all this assumes that the heuristics is admissible)

ILLUSTRATION: WHY IS A* OPTIMAL?

A* gradually adds "f-contours" of nodes (cf. BFS adds layers)

Contour i has all nodes with $f = f_i$, where $f_i < f_{i+1}$

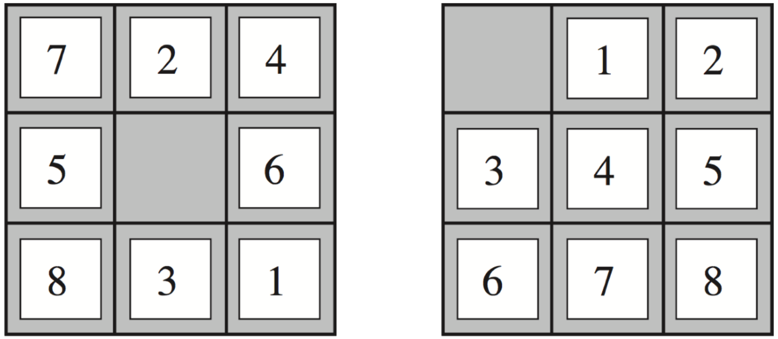


QUESTION TIME: HEURISTICS FOR THE 8 PUZZLE

$h_1(n)$ = number of misplaced tiles

$h_2(n)$ = total Manhattan distance

(i.e., no. of squares from desired location of each tile)



Start State   Goal State

$h_1(StartState)$ = 8

$h_2(StartState)$ = $3+1+2+2+2+3+3+2 = 18$

DOMINATING HEURISTICS

If (admissible) $h_2(n) \geq h_1(n)$ for all n,
then h_2 **dominates** h_1 and is better for search.

Typical search costs (for 8-puzzle):

depth = 14	DFS ≈ 3,000,000 nodes
	A*(h_1) = 539 nodes
	A*(h_2) = 113 nodes
depth = 24	DFS ≈ 54,000,000,000 nodes
	A*(h_1) = 39,135 nodes
	A*(h_2) = 1,641 nodes

Given any admissible heuristics h_a, h_b, the **maximum** heuristics $h(n)$ is also admissible and dominates both:

$$h(n) = \max(h_a(n), h_b(n))$$

HEURISTICS FROM A RELAXED PROBLEM

Admissible heuristics can be derived from the exact solution cost of a relaxed problem:

- If the rules of the 8-puzzle are relaxed so that a tile can move anywhere, then $h_1(n)$ gives the shortest solution
- If the rules are relaxed so that a tile can move to any adjacent square, then $h_2(n)$ gives the shortest solution

Key point: the optimal solution cost of a relaxed problem is never greater than the optimal solution cost of the real problem

SUMMARY OF TREE SEARCH STRATEGIES

Search strategy	**Frontier selection**	**Halts if solution?**	**Halts if no solution?**	**Space usage**
Depth first	Last node added	*No*	*No*	*Linear*
Breadth first	First node added	*Yes*	*No*	*Exp*
Best first	Global min $h(p)$	*No*	*No*	*Exp*
Lowest cost first	Minimal $cost(p)$	*Yes*	*No*	*Exp*
A*	Minimal $f(p)$	*Yes*	*No*	*Exp*

Halts if: If there is a path to a goal, it can find one, even on infinite graphs.
Halts if no: Even if there is no solution, it will halt on a finite graph (with cycles).
Space: Space complexity as a function of the length of the current path.

EXAMPLE DEMO

Here is an example demo of several different search algorithms, including A*. And you can play with different heuristics:

- http://qiao.github.io/PathFinding.js/visual/

Note that this demo is tailor-made for planar grids, which is a special case of all possible search graphs.

- (e.g., the Shrdlite graph will not be a planar grid)