Chapter 3: States and Searching

DIT410/TIN172 Artificial Intelligence

Peter Ljunglöf
modifed from slides by Poole & Mackworth
with some help from slides by Russel & Norvig

24, 27 March, 2015

Outline

Outline

Outline

Searching

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

- flat or modular or hierarchical
- explicit states or features or individuals and relations
- static or finite stage or indefinite stage or infinite stage
- fully observable or partially observable
- deterministic or stochastic dynamics
- goals or complex preferences
- single agent or multiple agents
- knowledge is given or knowledge is learned
- perfect rationality or bounded rationality

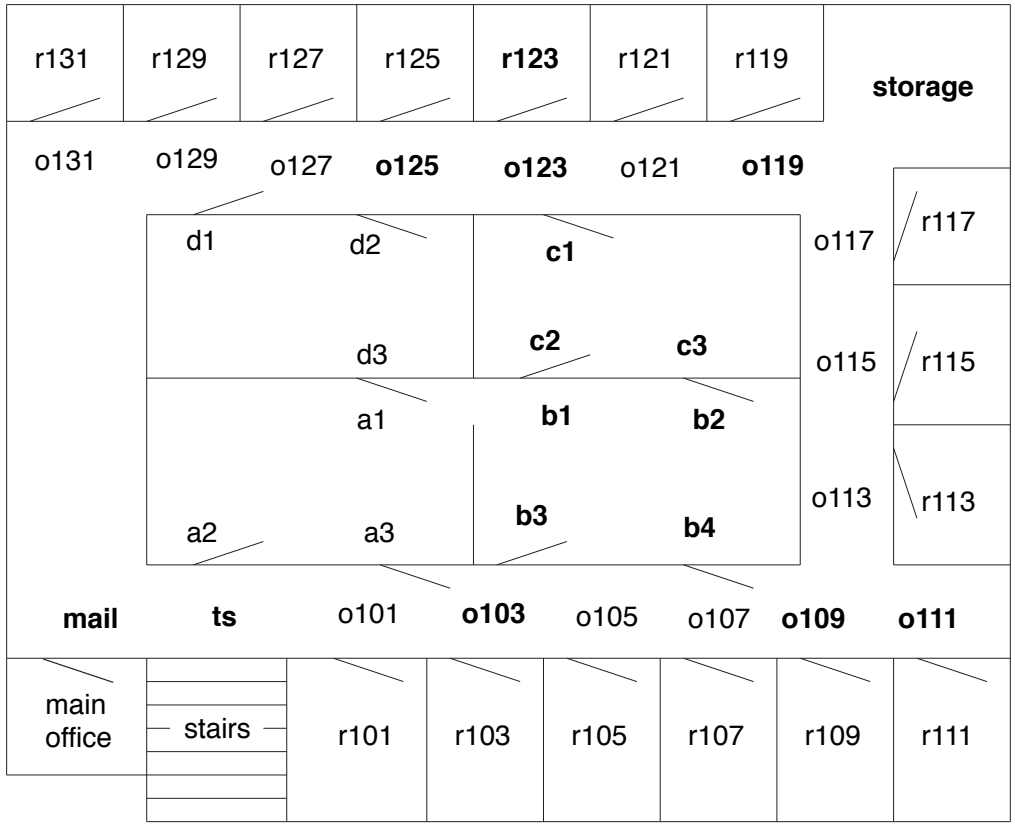
Outline

Directed Graphs

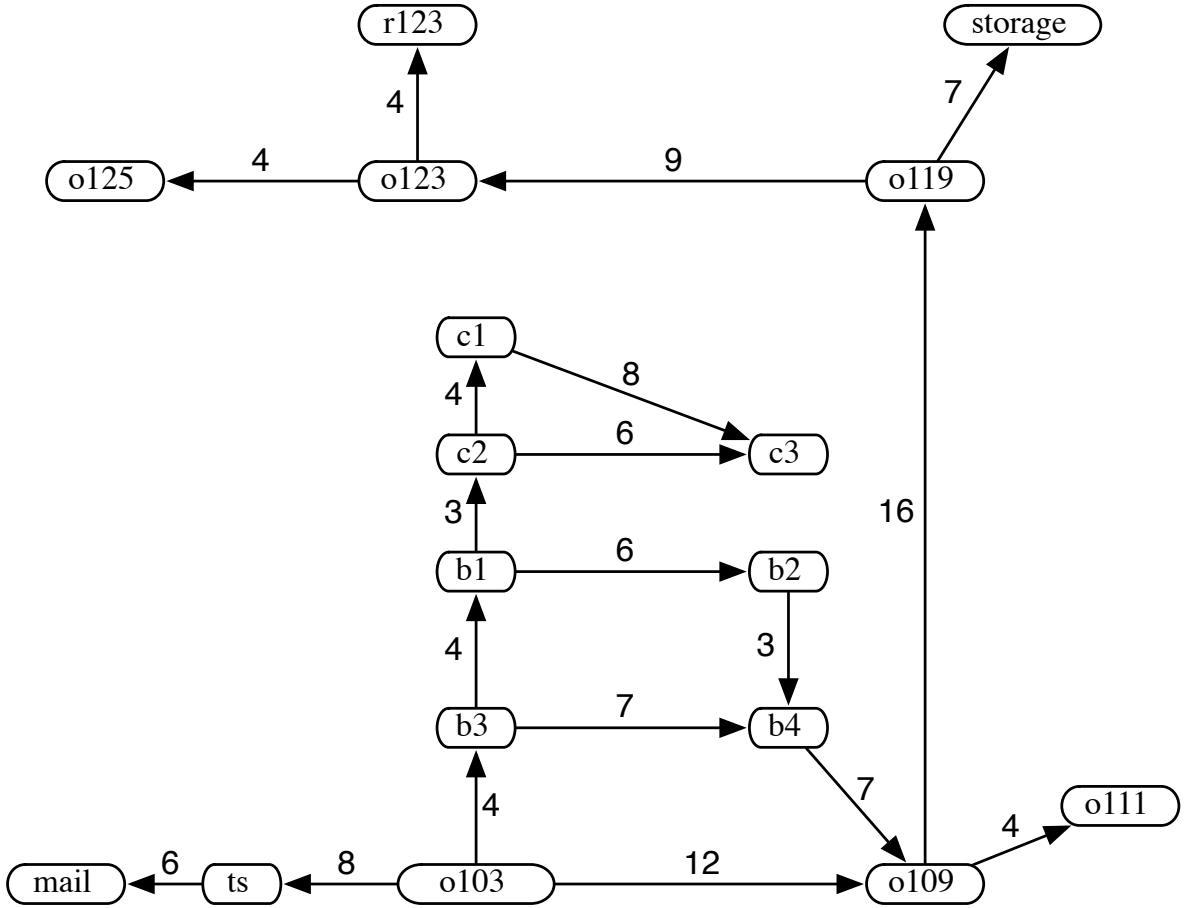
- A graph consists of a set N of nodes and a set A of ordered pairs of nodes, called arcs.
- Node n_2 is a neighbor of n_1 if there is an arc from n_1 to n_2. That is, if $\langle n_1, n_2 \rangle \in A$.
- A path is a sequence of nodes $\langle n_0, n_1, \ldots, n_k \rangle$ such that $\langle n_{i-1}, n_i \rangle \in A$.
- The length of path $\langle n_0, n_1, \ldots, n_k \rangle$ is k.
- A solution is a path from a start node to a goal node, given a set of start nodes and goal nodes.

Example problem: Delivery robot

The robot wants to get from outside room 103 to inside room 123.

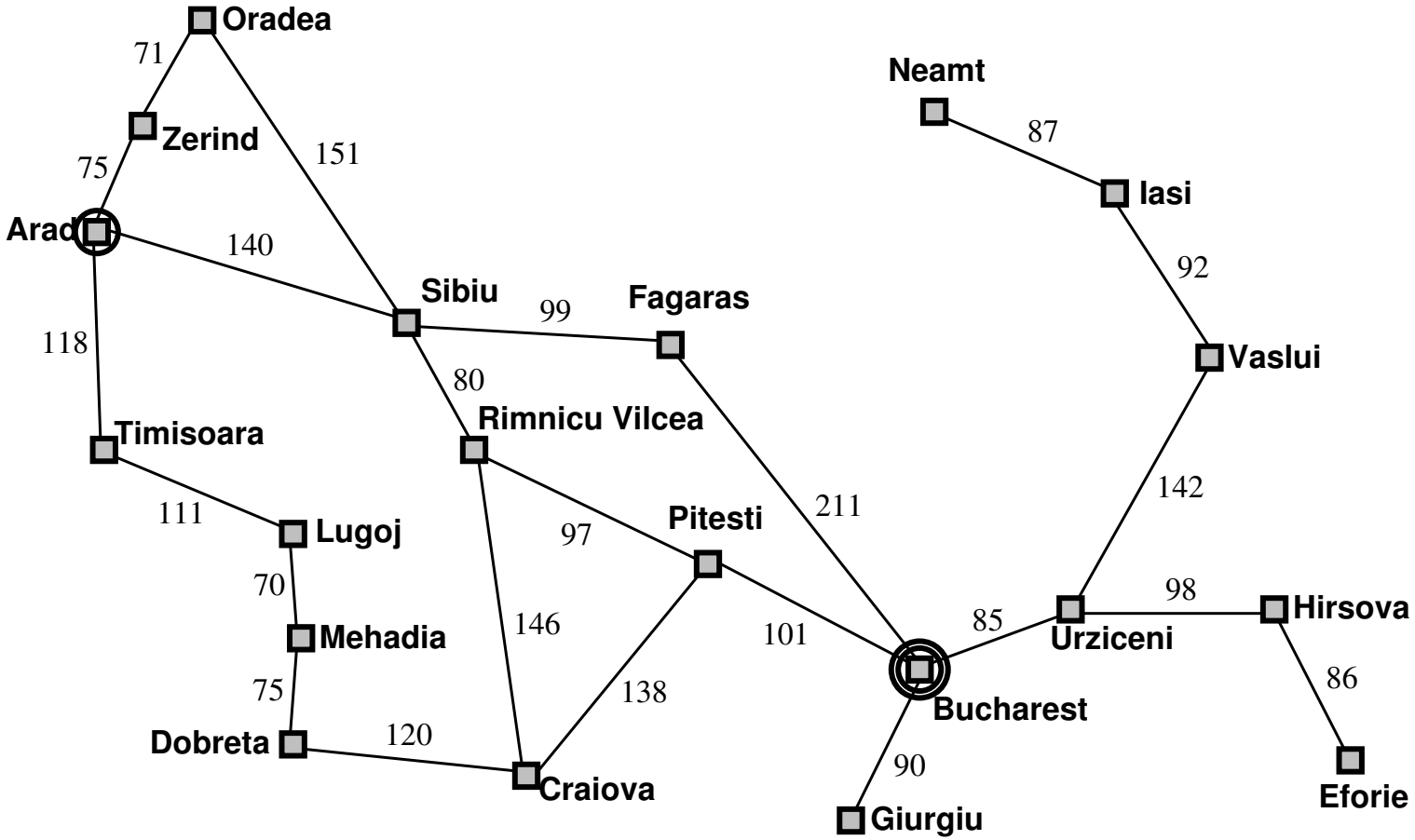


Delivery robot: State space graph



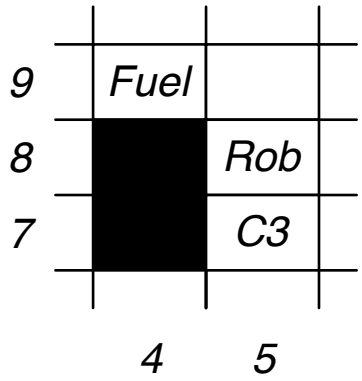
Example problem: Travel in Romania

We want to drive from Arad to Bucharest in Romania.



Example problem: Grid game

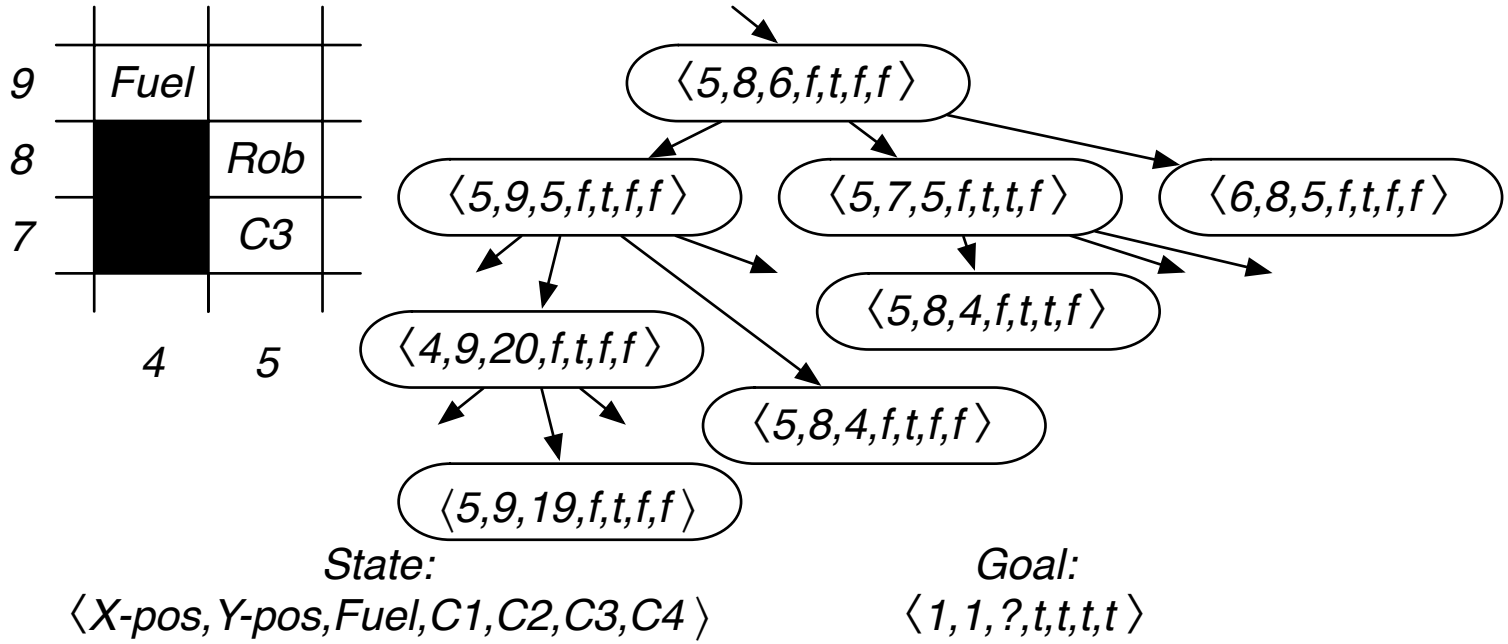
Grid game: Rob needs to collect coins C_1, C_2, C_3, C_4, without running out of fuel, and end up at location $(1, 1)$:



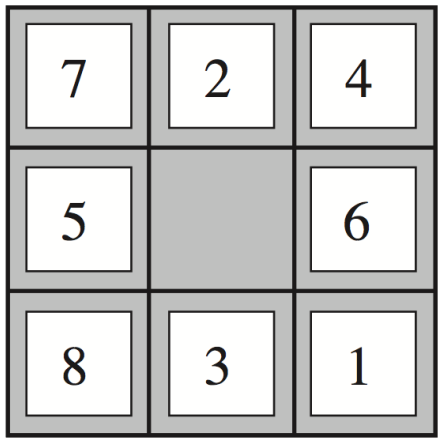
What is a good representation of the search states and the goal?

Grid game: State representation

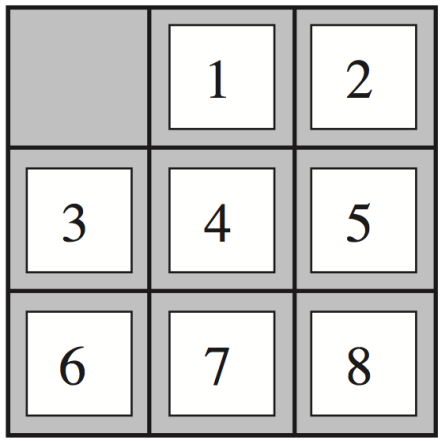
Grid game: Rob needs to collect coins C_1, C_2, C_3, C_4, without running out of fuel, and end up at location $(1, 1)$:



Example: The 8-puzzle



Start State



Goal State

States:

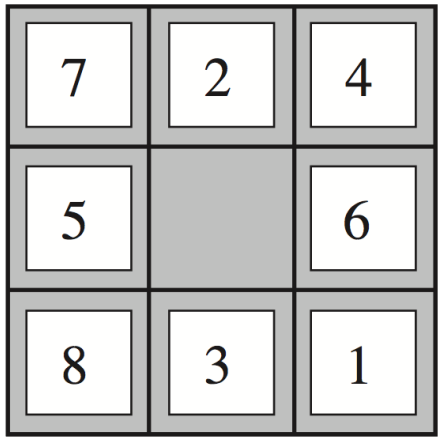
Initial state:

Actions:

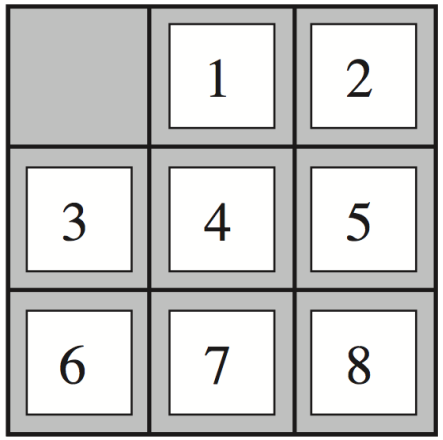
Goal test:

Path cost:

Example: The 8-puzzle



Start State



Goal State

States: a 3×3 matrix of integers $0 \leq n \leq 8$

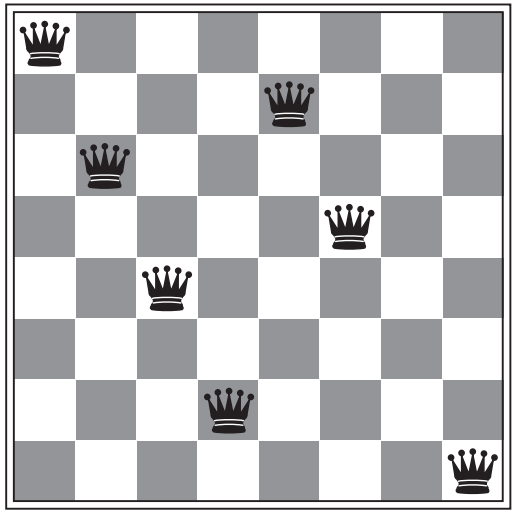
Initial state: any state

Actions: move the blank space: left, right, up, down

Goal test: equal to goal state (given above)

Path cost: 1 per move

Example: The 8-queens problem



States:

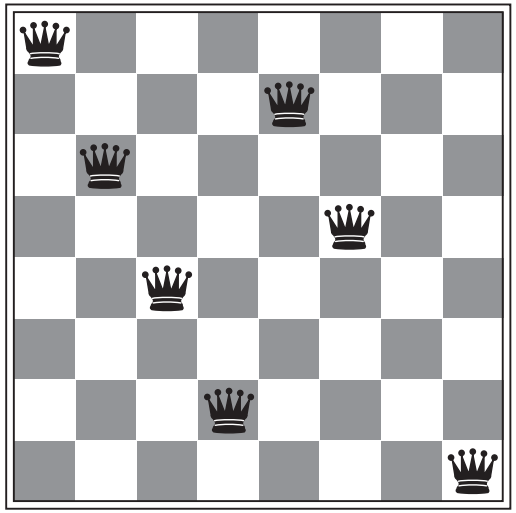
Initial state:

Actions:

Goal test:

Path cost:

Example: The 8-queens problem



States: any arrangement of 0 to 8 queens on the board

Initial state: no queens on the board

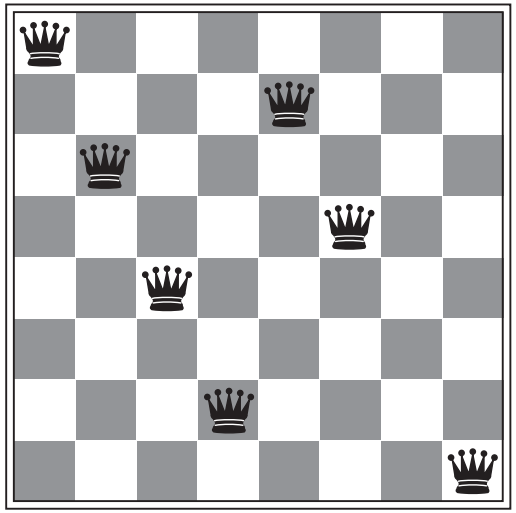
Actions: add a queen to any empty square

Goal test: 8 queens on the board, none attacked

Path cost: 1 per move

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

Example: The 8-queens problem (alternative)



States:

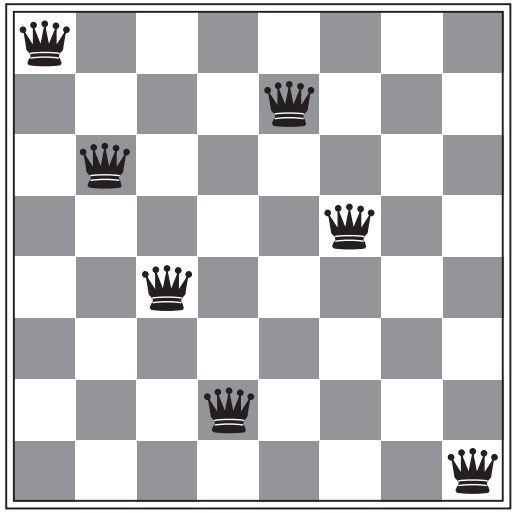
Initial state:

Actions:

Goal test:

Path cost:

Example: The 8-queens problem (alternative)



States: one queen per column in leftmost columns

Initial state: no queens on the board

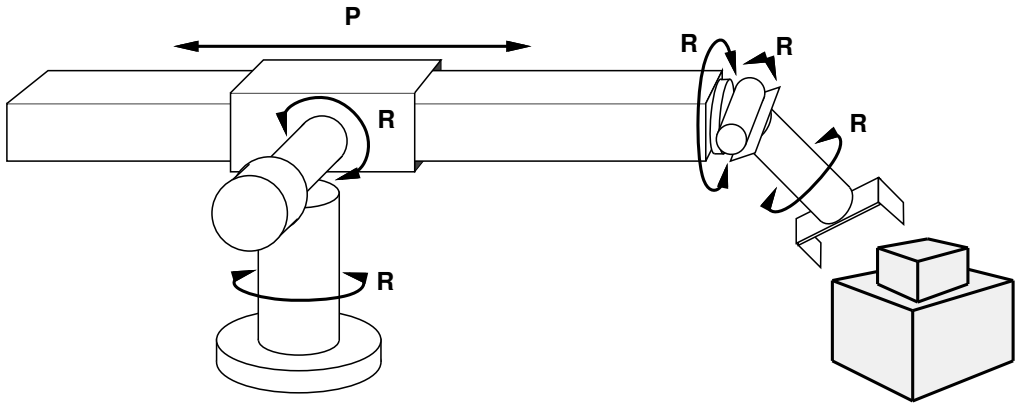
Actions: add a queen to any square in the leftmost empty column, making sure that no queen is attacked

Goal test: 8 queens on the board, none attacked

Path cost: 1 per move

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

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

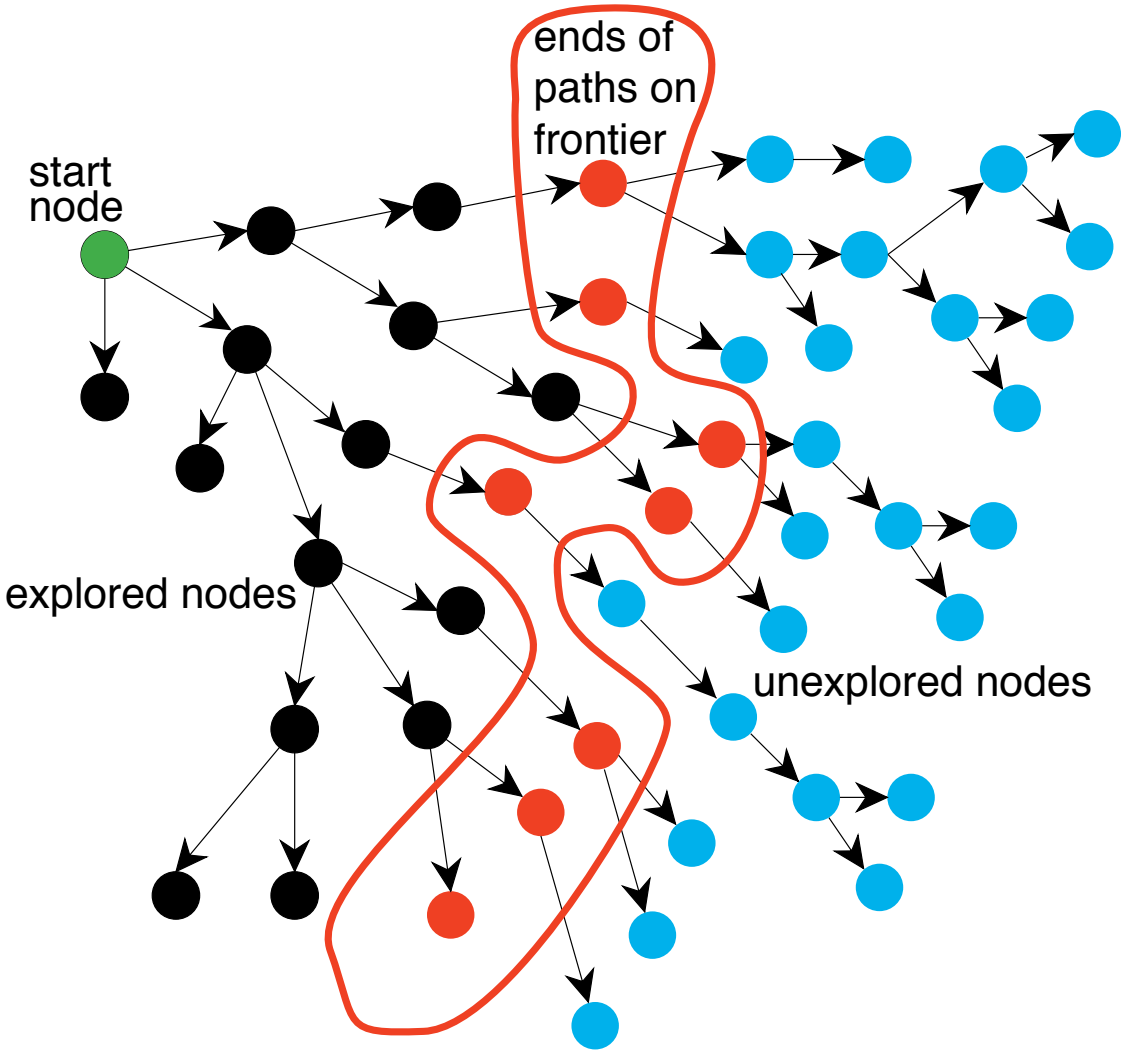
Path cost: time to execute

Outline

How do we search in a graph?

- Generic search algorithm: given a graph, start nodes, and goal nodes, incrementally explore paths from the start nodes.
- Maintain a *frontier* of paths from the start node that have been 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 graph searching



A generic graph search algorithm

procedure GenericSearch(*graph*, *startnodes*, *goal*):
    frontier := $\{\langle s \rangle : s \in \mathit{startnodes}\}$
    while *frontier* is not empty:
        select and remove path $\langle n_0, \ldots, n_k \rangle$ from *frontier*
        if $\mathit{goal}(n_k)$:
            return $\langle n_0, \ldots, n_k \rangle$
        for every *graph* neighbor n of n_k:
            add $\langle n_0, \ldots, n_k, n \rangle$ to *frontier*
    end while
end procedure GenericSearch

A generic graph search algorithm

- The search strategy is defined by which value is selected from the frontier at each stage.
- The graph is defined by its neighbors.
- The solution is defined by *goal*.
- If more than one answer is required, the search can continue from the return.

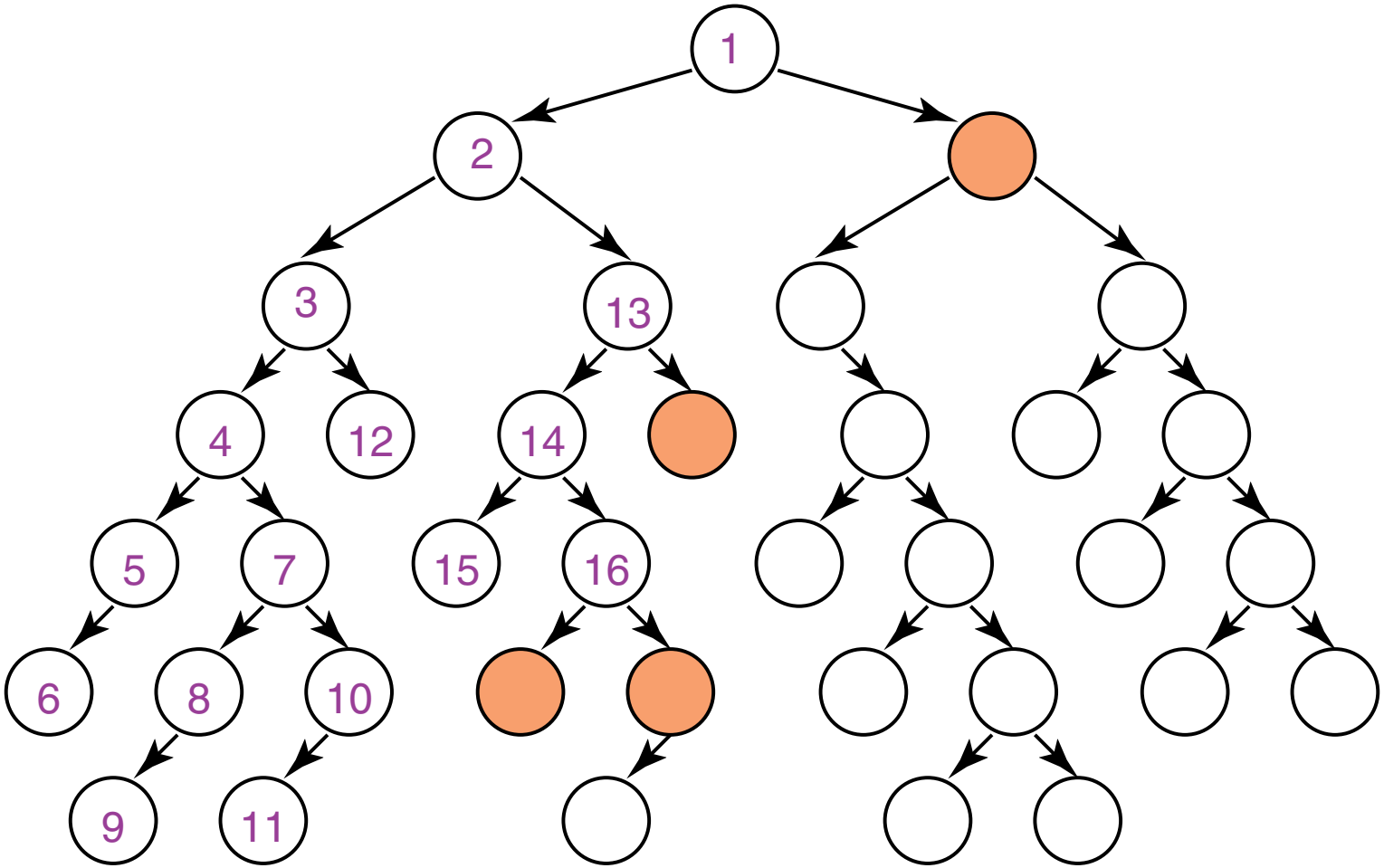
Outline

Outline

Depth-first search

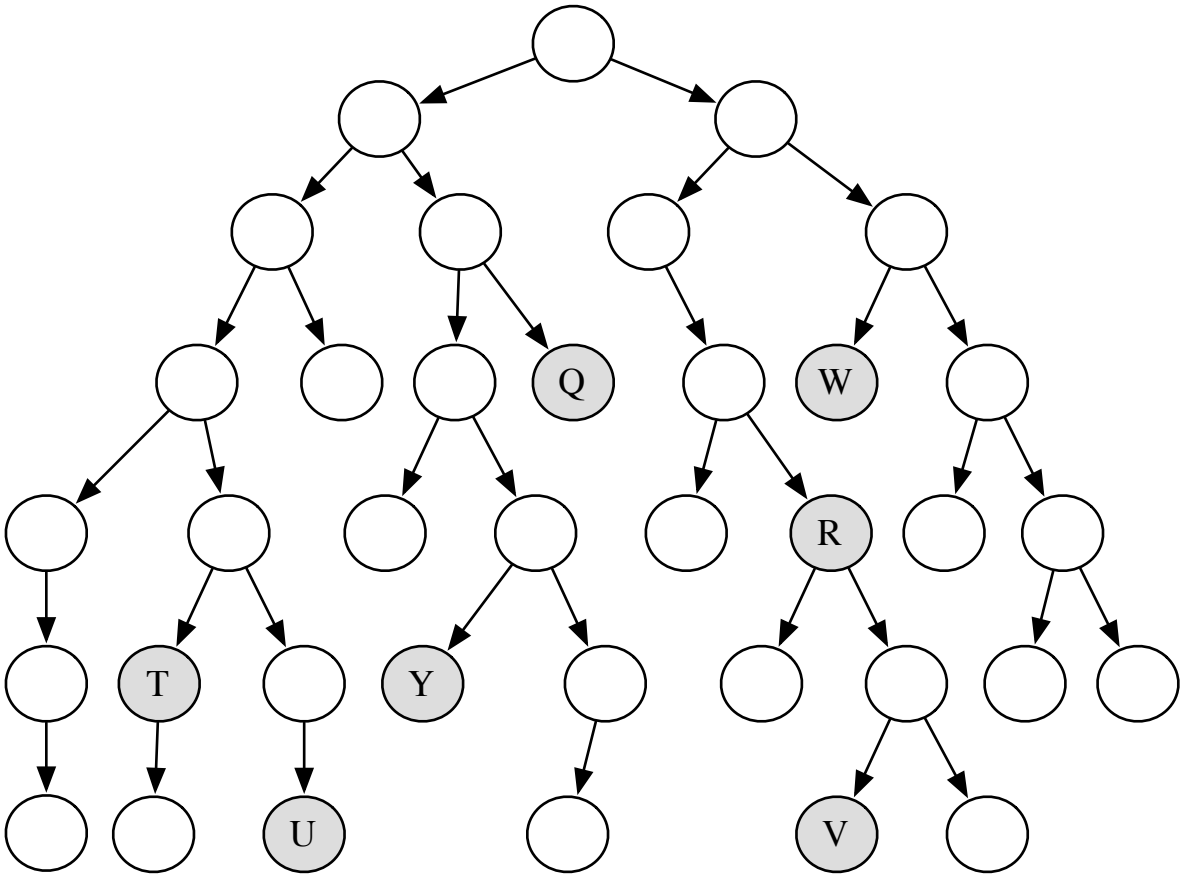
- Depth-first search treats the frontier as a stack
- It always selects one of the last elements added to the frontier.
- If the list of paths on the frontier is $[p_1, p_2, \ldots]$
  - p_1 is selected. Paths that extend p_1 are added to the front of the stack (in front of p_2).
  - p_2 is only selected when all paths from p_1 have been explored.

Illustrative graph: Depth-first search



Question time: Depth-first search

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



Complexity of depth-first search

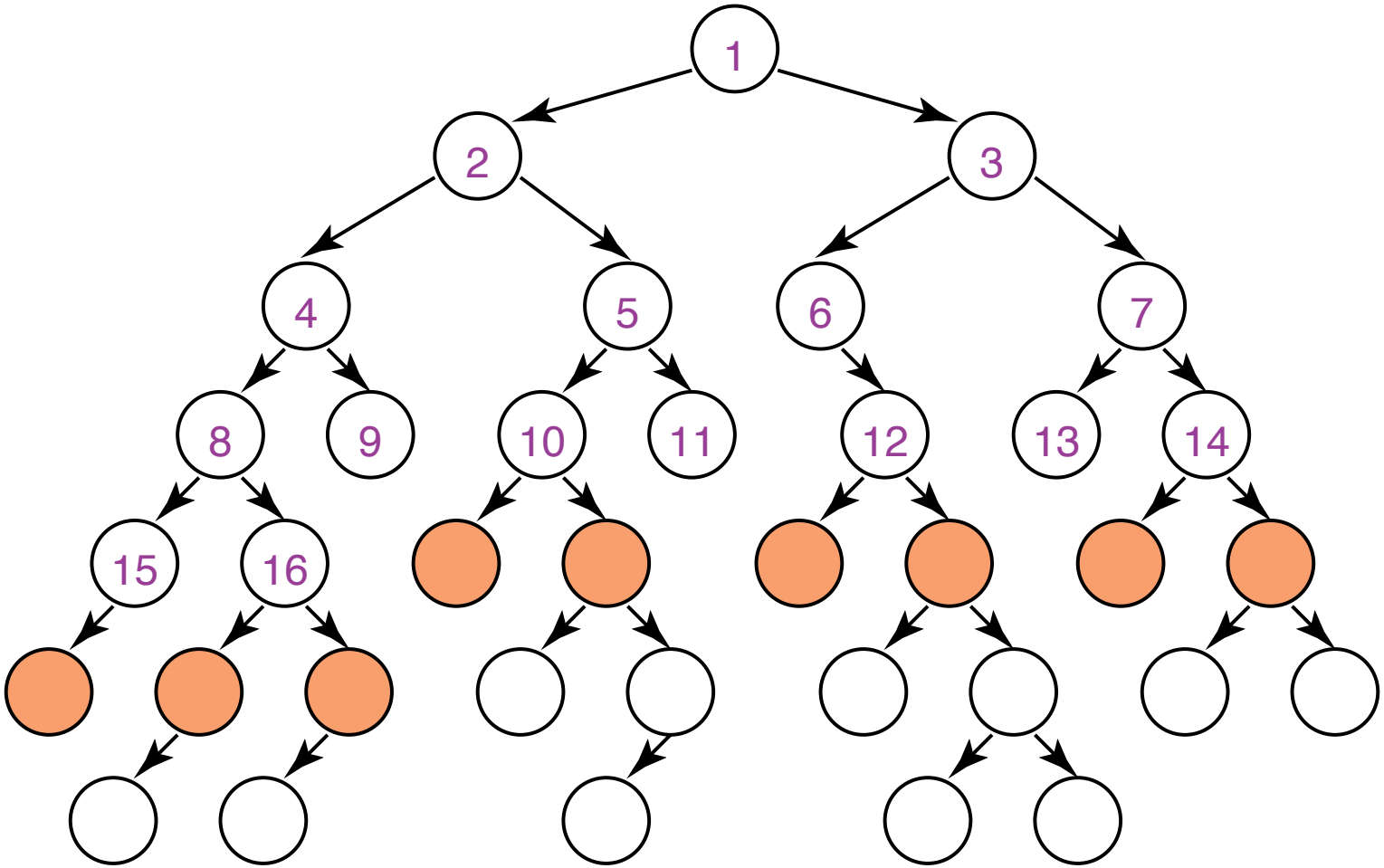
- Does depth-first search guarantee to find the path with fewest arcs?
- What happens on infinite graphs or on graphs with cycles if there is a solution?
- What is the time complexity as a function of length of the path selected?
- What is the space complexity as a function of length of the path selected?
- How does the goal affect the search?

Outline

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 $[p_1, p_2, \ldots, p_r]$:
  - p_1 is selected. Its neighbors are added to the end of the queue, after p_r.
  - p_2 is selected next.

Illustrative graph: breadth-first search

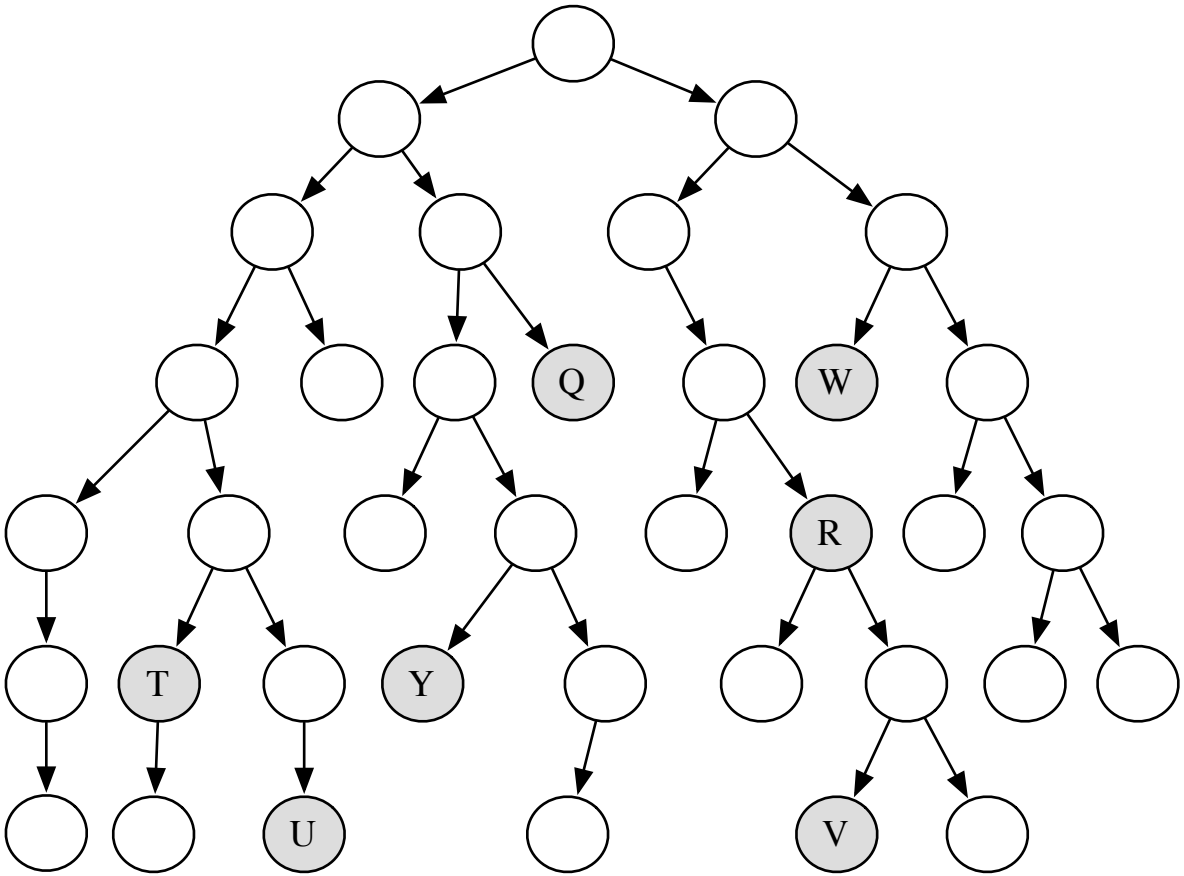


Complexity of breadth-first search

- Does breadth-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 length of the path selected?
- What is the space complexity as a function of the length of the path selected?
- How does the goal affect the search?

Question time: Breadth-first search

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



Outline

Lowest-cost-first search

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

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

  An optimal solution is one with minimum cost.
- At each stage, lowest-cost-first search selects a path on the frontier with lowest cost.
- The frontier is a priority queue ordered by path cost.
- It finds a least-cost path to a goal node.
- When arc costs are equal $\Longrightarrow$ breadth-first search.

Outline

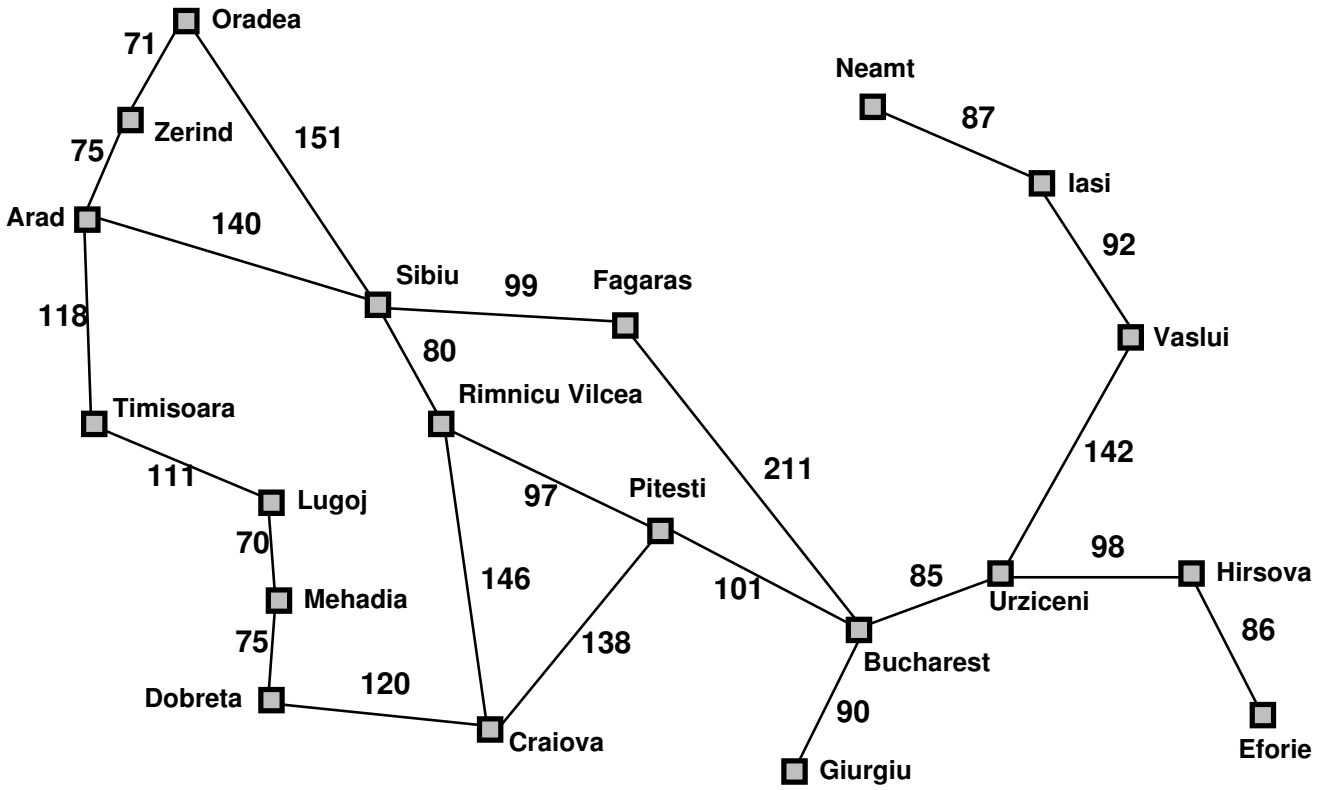
Heuristic search

- Idea: don't ignore the goal when selecting paths.
- Often there is extra knowledge that can be used to 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 can be extended to paths: $h(\langle n_0, \ldots, n_k \rangle) = h(n_k)$.
- $h(n)$ 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 heuristic function that is an underestimate of the actual cost of a path to a goal.

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 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.
- 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 rest of 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



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

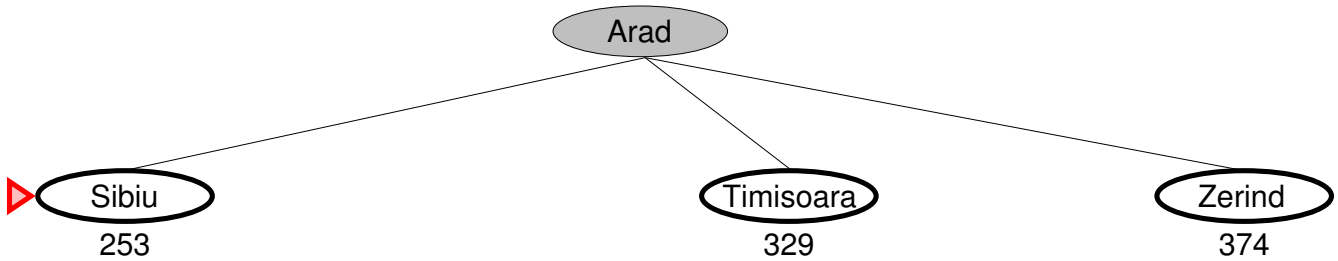
Outline

Best-first search

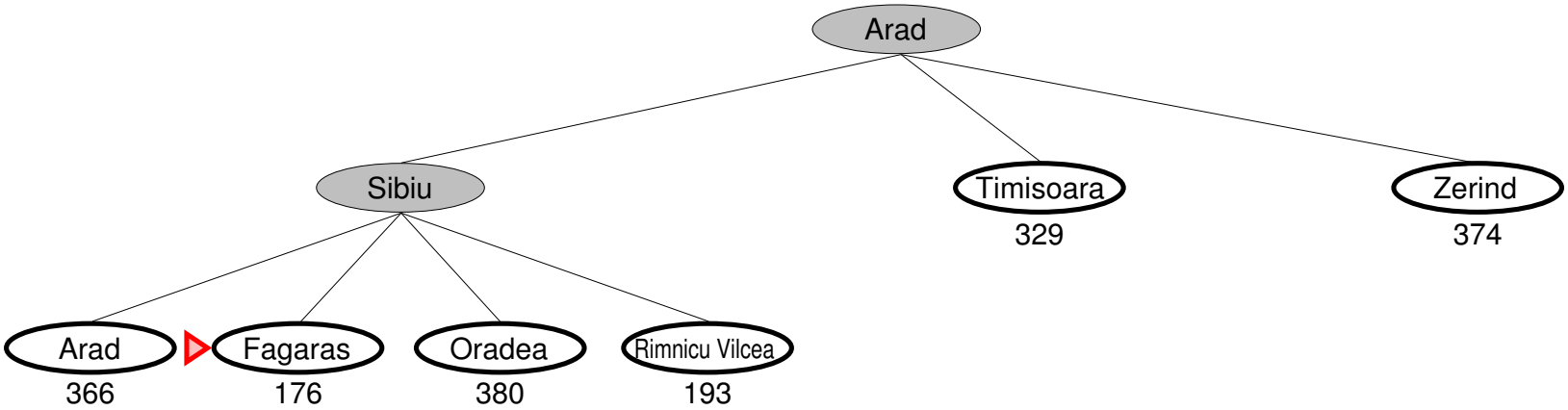
- Idea: select the path whose end is closest to a goal according to the heuristic function.
- Best-first search selects a path on the frontier with minimal h-value.
- It treats the frontier as a priority queue ordered by h.

Example: Romania

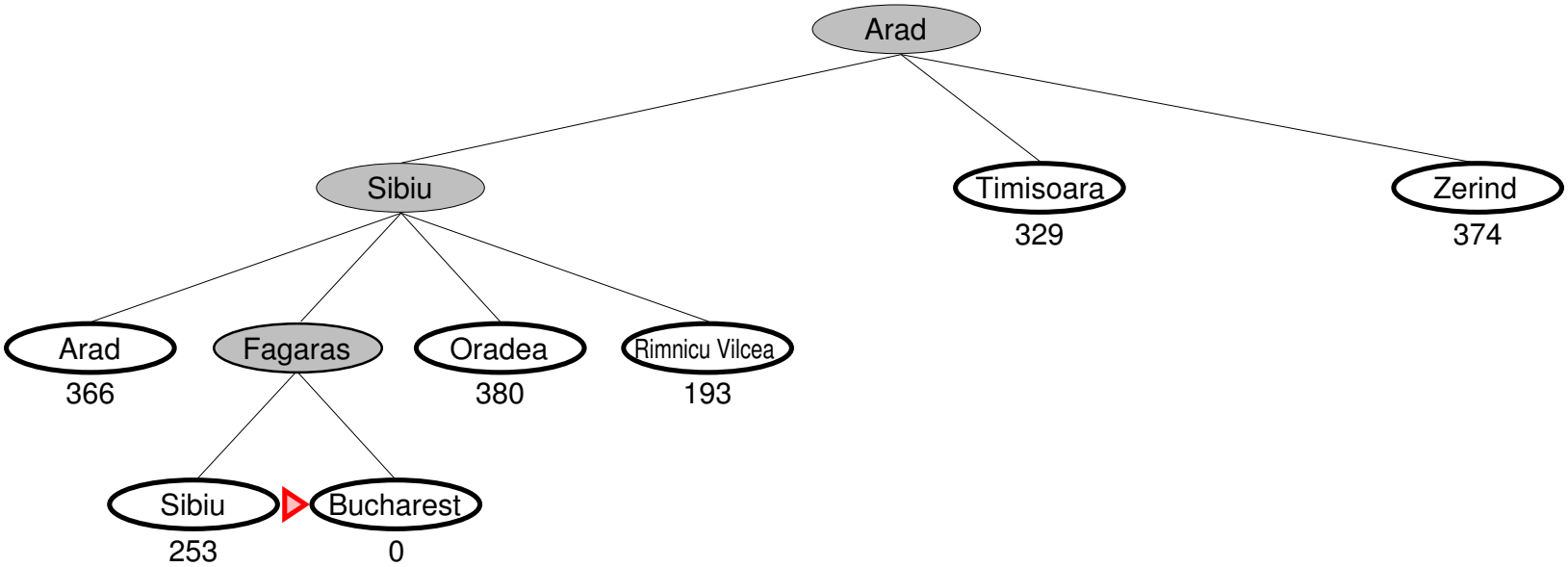
Example: Romania



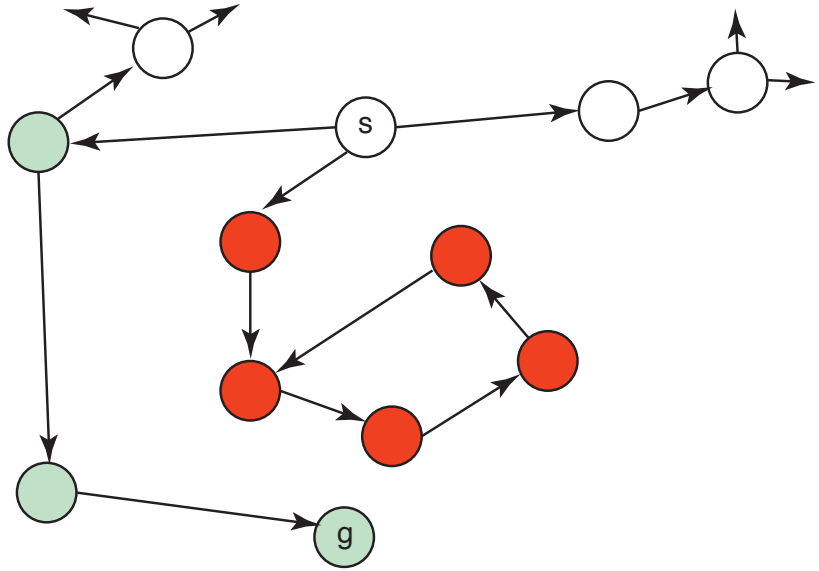
Example: Romania



Example: Romania



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 shortest path or 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 length of the path selected?
- What is the space complexity as a function of length of the path selected?
- How does the goal affect the search?

Outline

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

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

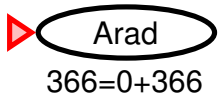
A^* search

- 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.

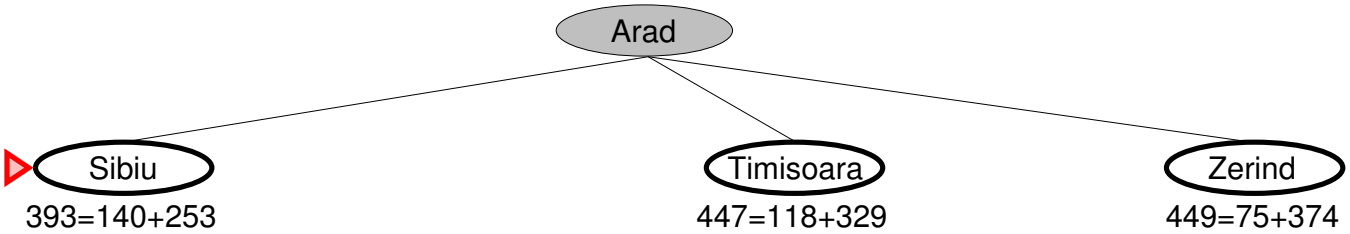
Complexity of A^* search

- Does A^* search guarantee to find the shortest path or 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 length of the path selected?
- What is the space complexity as a function of length of the path selected?
- How does the goal affect the search?

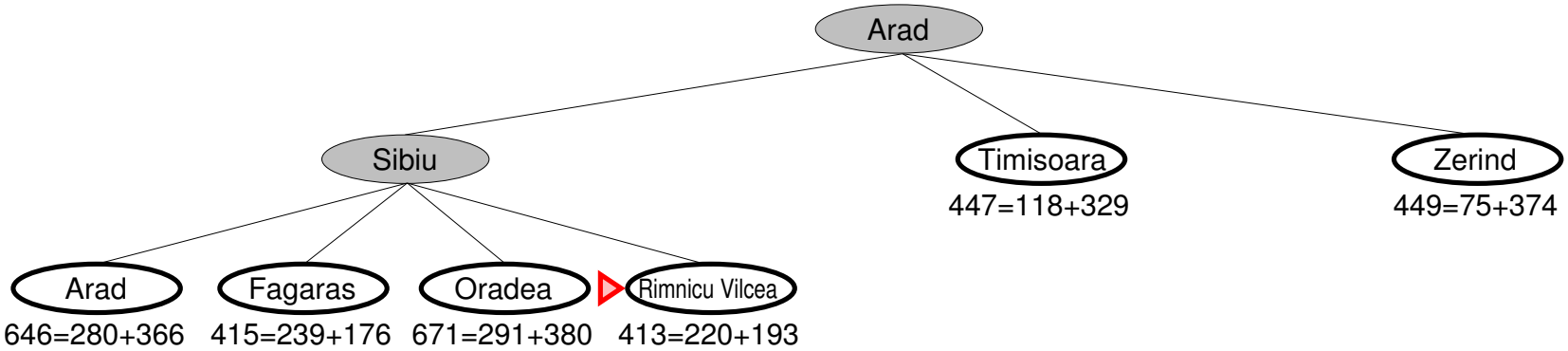
Example: Romania



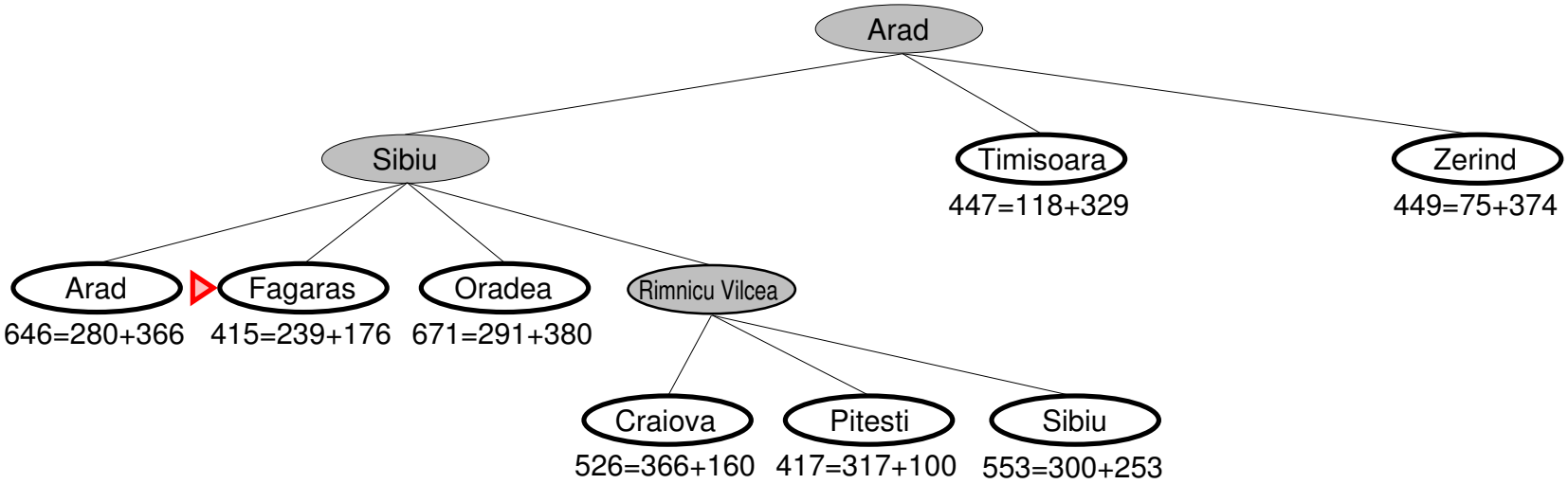
Example: Romania



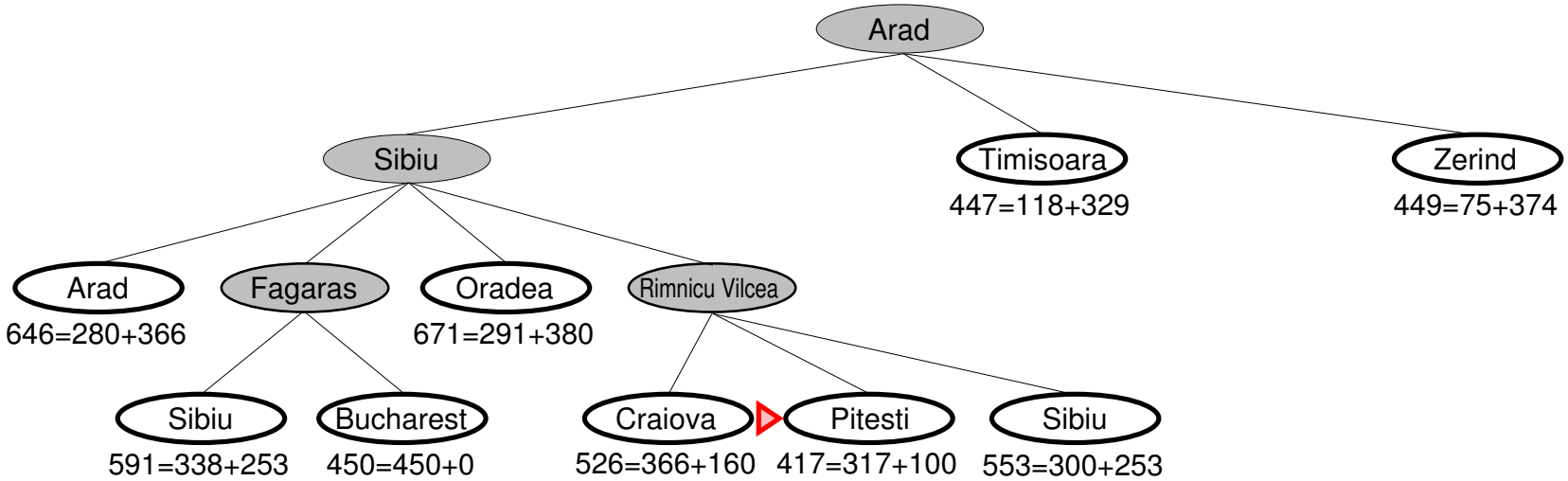
Example: Romania



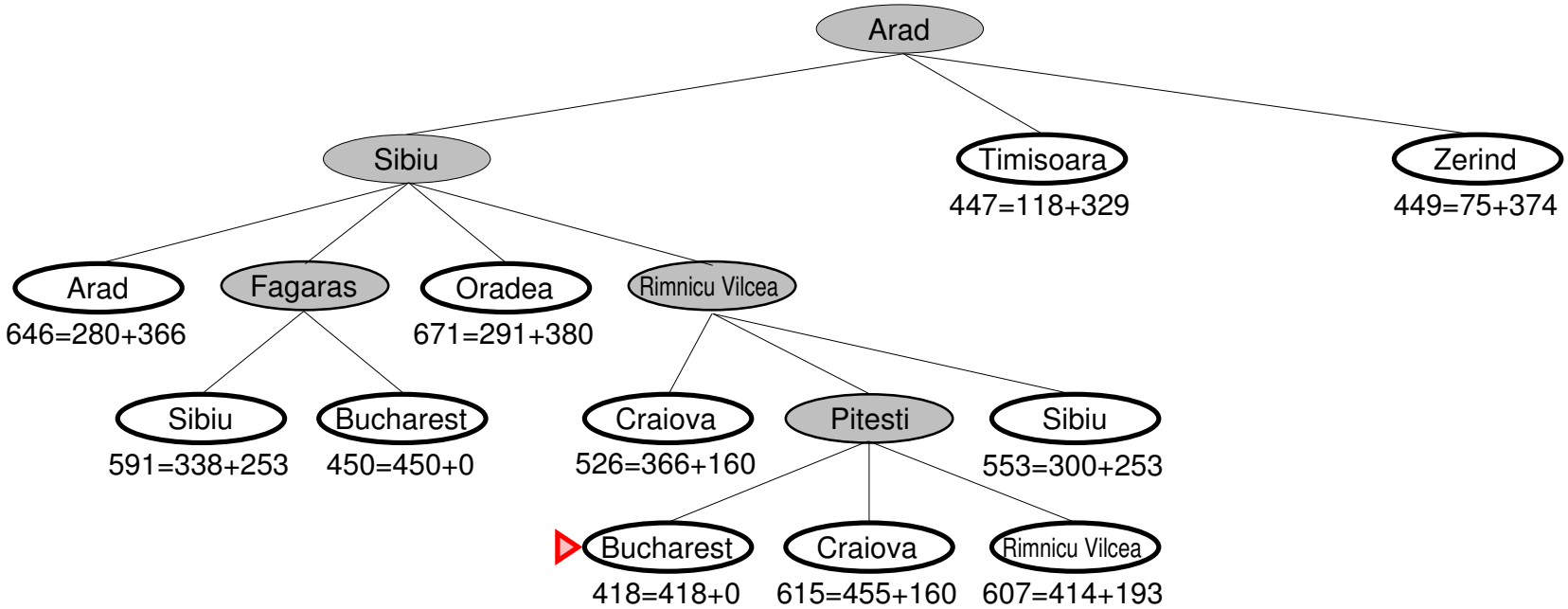
Example: Romania



Example: Romania



Example: Romania



Admissibility (optimality) of A^*

If there is a solution, A^* always finds an optimal one first, if:

- the branching factor is finite,
- arc costs are bounded above zero (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.

A^* always finds a solution

A^* can 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.

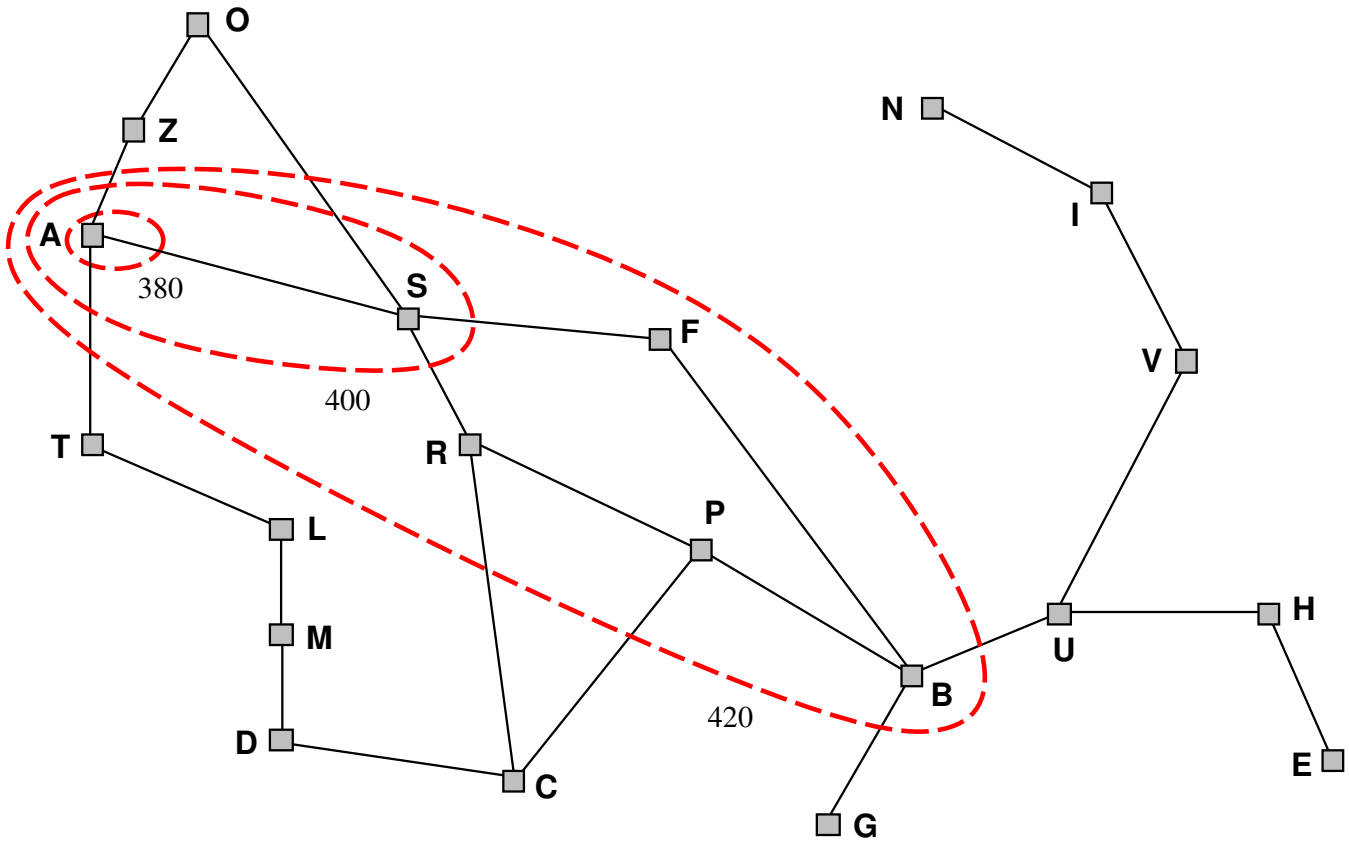
A^* finds an optimal solution first

The first path to a goal selected is an optimal path, because:

- The f-value for any node on an optimal solution path is less than or equal to the f-value of an optimal solution.
  - this is because h is an *underestimate* of the actual cost
- Thus, the f-value of a node on an optimal solution path is less than the f-value for any non-optimal solution.
- Thus, a non-optimal solution can never be chosen while a node exists on the frontier that leads to an optimal solution.
  - because an element with minimum f-value is chosen at each step
- So, before it can select a non-optimal solution, it will have to pick all of the nodes on an optimal path, including each of the optimal solutions.

Illustration: Why is A^* admissible?

A^* gradually adds "f-contours" of nodes (cf. BFS adds layers).
Contour i has all nodes with $f = f_i$, where $f_i < f_{i+1}$.



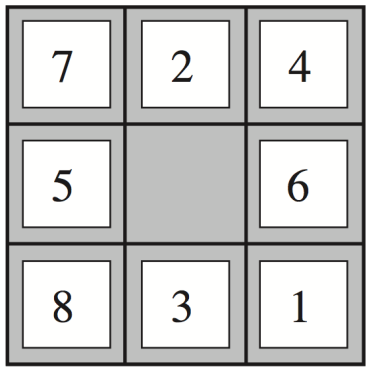
Example: Admissible heuristics

For the 8-puzzle:

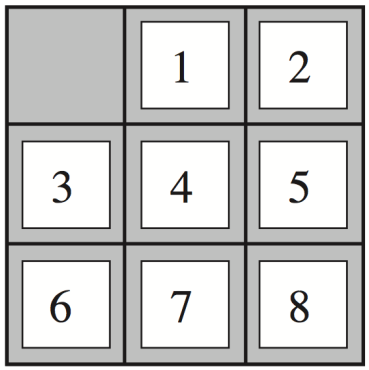
$h_1(n)$ = number of misplaced tiles

$h_2(n)$ = total *Manhattan distance*

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



Start State



Goal State

$h_1(S)$ = ??

$h_2(S)$ = ??

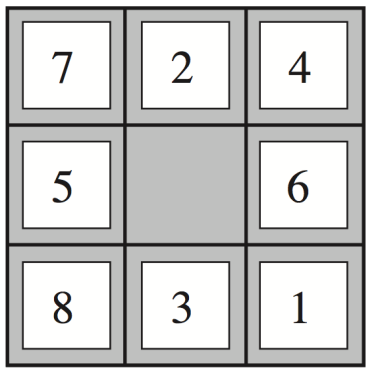
Example: Admissible heuristics

For the 8-puzzle:

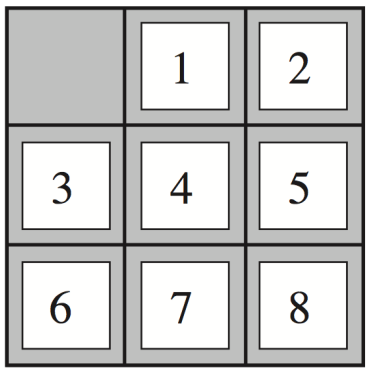
$h_1(n)$ = number of misplaced tiles

$h_2(n)$ = total *Manhattan distance*

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



Start State



Goal State

$h_1(S) = 8$

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

Dominating heuristics

If $h_2(n) \geq h_1(n)$ for all n (both admissible), then h_2 *dominates* h_1 and is better for search. Typical search costs (for 8-puzzle):

$depth = 14$	DFS $\approx$ 3,000,000 nodes
	$A^*(h_1) = 539$ nodes
	$A^*(h_2) = 113$ nodes
$depth = 24$	DFS $\approx$ 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,

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

is also admissible and dominates h_a, h_b.

Summary of search strategies

Strategy	**Frontier selection**	**Halts if solution?**	**Halts if no solution?**	**Space**
Depth-first	Last node added			
Breadth-first	First node added			
Best-first	Global min $h(p)$			
Lowest-cost-first	Minimal $cost(p)$			
A^*	Minimal $f(p)$			

Halts if: If there 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* (perhaps with cycles).

Space: Space complexity as a function of the length of the current path.

Summary of search strategies

Strategy	**Frontier selection**	**Halts if solution?**	**Halts if no solution?**	**Space**
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 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* (perhaps 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^*. Furthermore you can play with different heuristics:

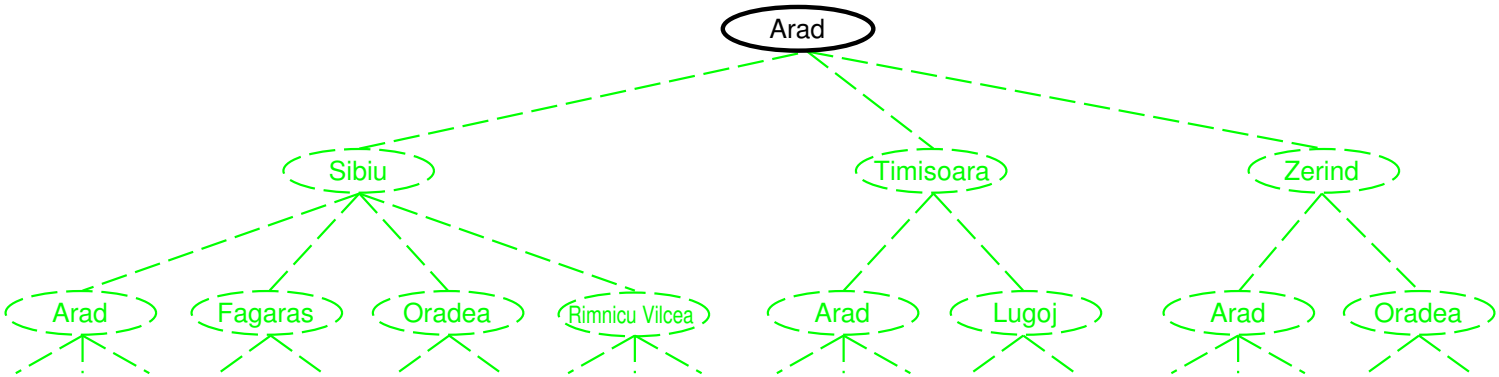
`http://qiao.github.io/PathFinding.js/visual/`

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

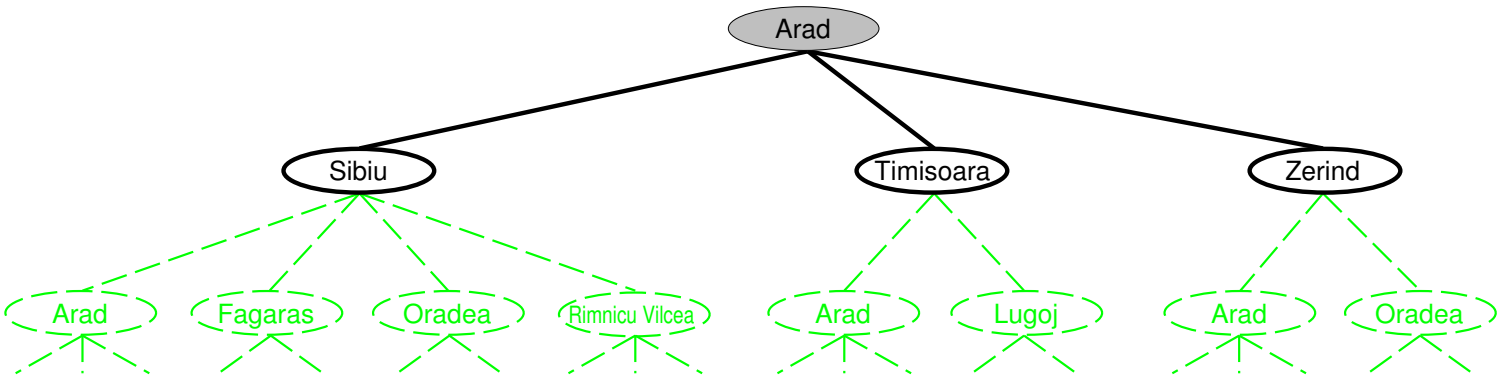
Outline

Outline

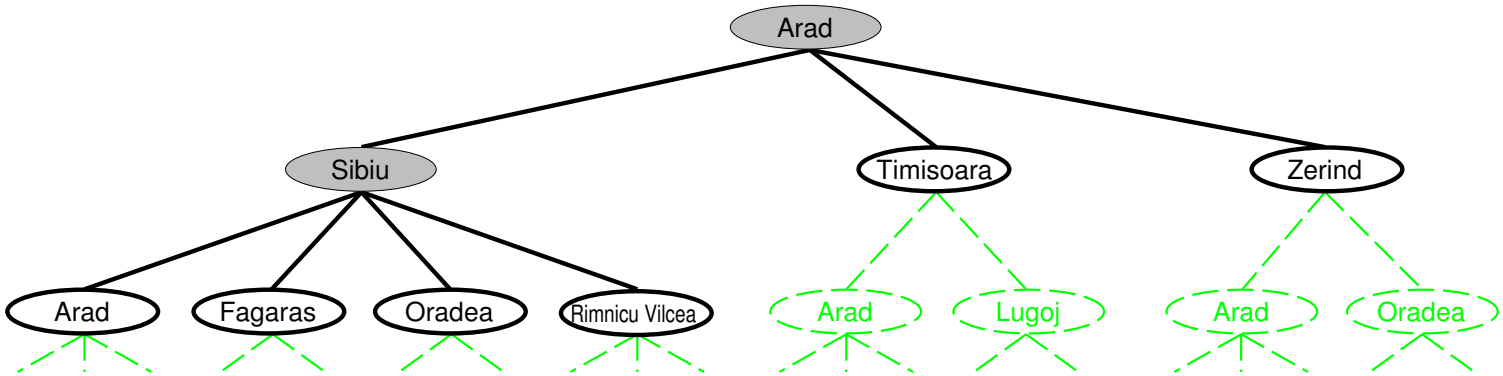
Example: Driving to Bucharest



Example: Driving to Bucharest



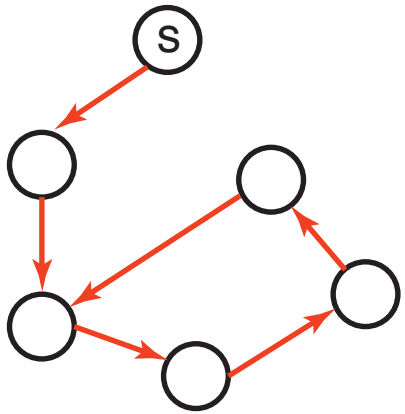
Example: Driving to Bucharest



Note: Arad is one of the expanded nodes!

This corresponds to going to Sibiu and then returning to Arad.

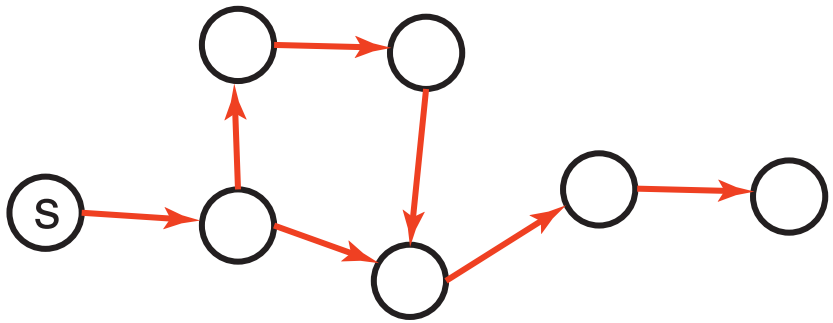
Cycle checking



- A searcher can prune a path that ends in a node already on the path, without removing an optimal solution.
- Checking for cycles can be done in linear time (in path length)
  - except for depth-first, where it can be done in constant time
- Does cycle checking mean the algorithms halt on finite graphs?

Outline

Multiple-path pruning



- Multiple path pruning: prune a path to node n that the searcher has already found a path to.
- What needs to be stored?
- How does multiple-path pruning compare to cycle checking?
- Do search algorithms with multiple-path pruning always halt on finite graphs?
- What is the space & time overhead of multiple-path pruning?
- Can multiple-path pruning prevent an optimal solution being found?

Multiple-path pruning & Optimal solutions

What if a subsequent path to n is shorter than the first path to n?

Possible solutions:

- Remove all paths from the frontier that use the longer path.
- Change the initial segment of the paths on the frontier to use the shorter path.
- Ensure this doesn't happen. Make sure that the shortest path to a node is found first.

*Multiple-path pruning & A**

- Suppose path p to n was selected, but there is a shorter path to n. Suppose this shorter path is via path p' on the frontier.
- Suppose path p' ends at node n'.
- p was selected before p', so: $cost(p) + h(n) \leq cost(p') + h(n')$.
- Suppose $cost(n', n)$ is the actual cost of a path from n' to n. The path to n via p' is shorter that p so: $cost(p') + cost(n', n) < cost(p)$.
- Combining the two:

$$cost(n', n) < cost(p) - cost(p') \leq h(n') - h(n)$$

- We can ensure this doesn't occur if $|h(n') - h(n)| \leq cost(n', n)$.

Monotone restriction

- Heuristic function h satisfies the monotone restriction if $|h(m) - h(n)| \leq cost(m, n)$ for every arc $\langle m, n \rangle$.
- If h satisfies the monotone restriction, A^* with multiple path pruning always finds the shortest path to a goal.
- This is a strengthening of the admissibility criterion.

Outline

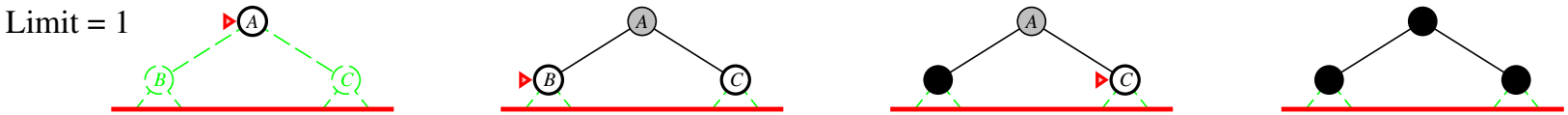
Iterative deepening

- So far all search strategies that are guaranteed to halt use exponential space.
- Idea: recompute elements of the frontier rather than saving them.
- Look for paths of depth 0, then 1, then 2, then 3, etc.
- Depth-bounded depth-first search can do this in linear space.
- If a path cannot be found at *depth bound*, look for a path at *depth bound* + 1. Increase *depth bound* when the search fails unnaturally (*depth bound* was reached).

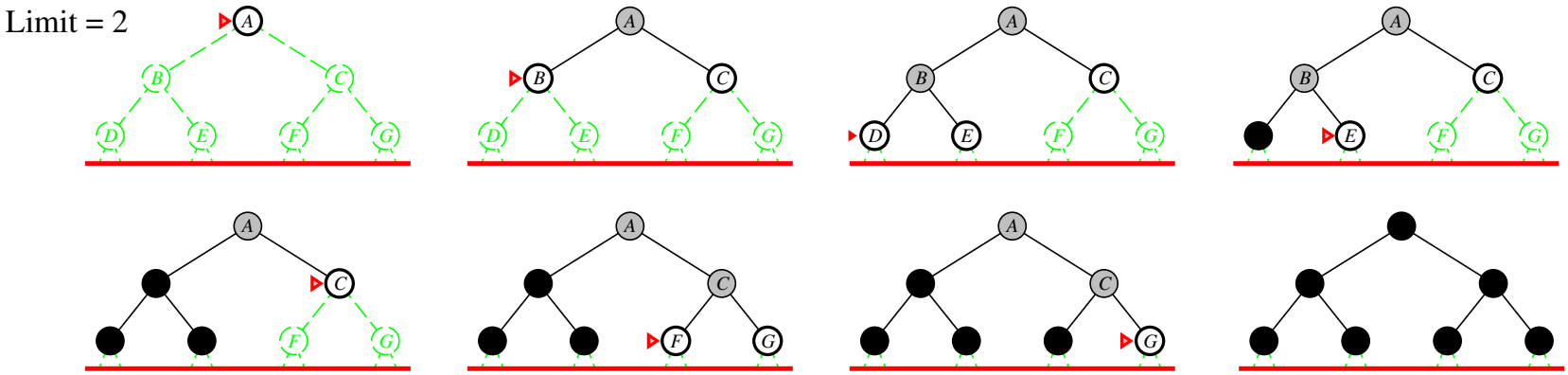
Iterative deepening search, bound = 0

Limit = 0

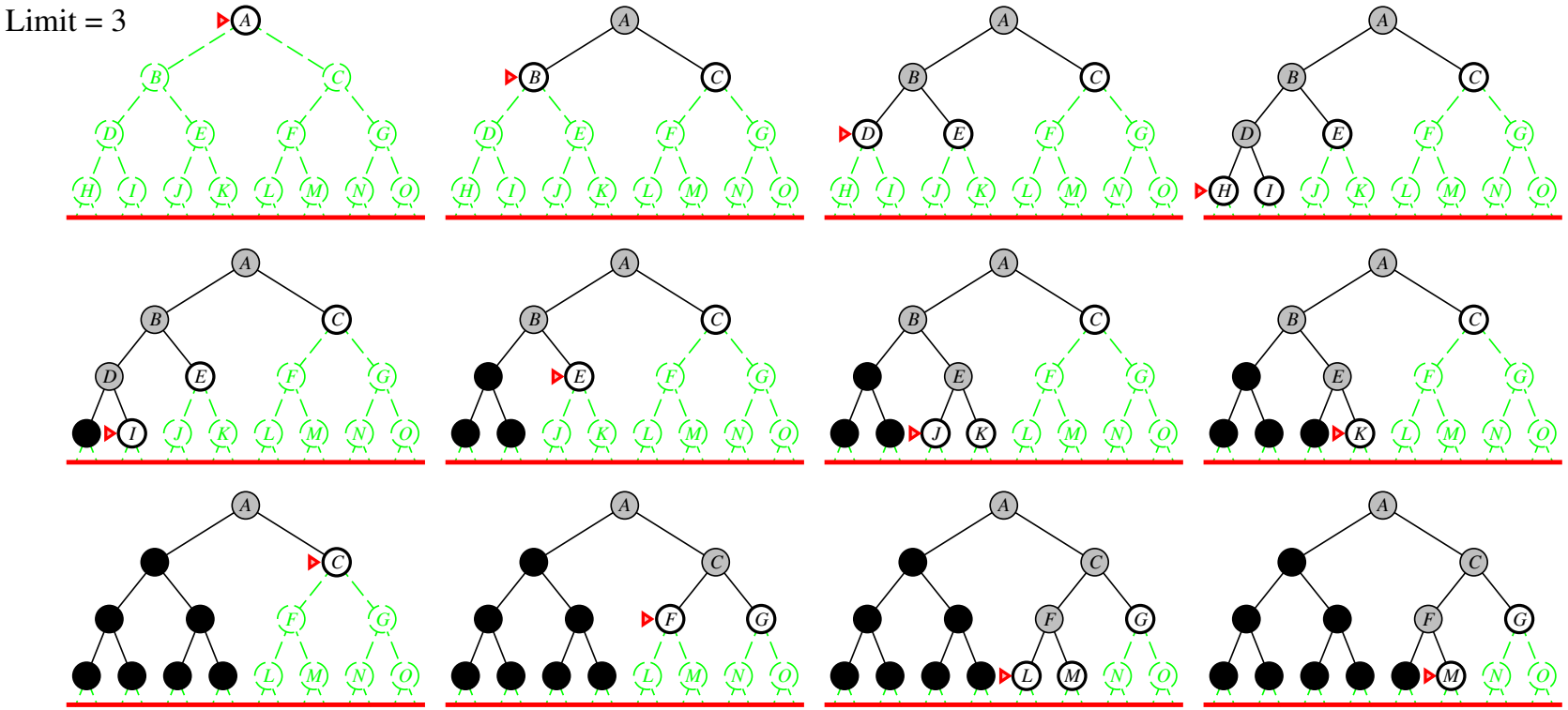
Iterative deepening search, bound = 1



Iterative deepening search, bound = 2



Iterative deepening search, bound = 3



Iterative-deepening search

procedure IDSearch(*graph*, *start*, *goal*):
    for *bound* **in** $0, 1, 2, \ldots$:
        result := BoundedDFSearch($\langle s \rangle$, *bound*)
        if *result* is a path **or** *result* is failure:
            return *result*
end procedure IDSearch

procedure BoundedDFSearch($\langle n_0, \ldots, n_k \rangle$, *bound*):
    if $bound > 0$:
        for every neighbor n of n_k:
            BoundedDFSearch($\langle n_0, \ldots, n_k, n \rangle$, $bound - 1$)
    else if $goal(n_k)$**: return** path $\langle n_0, \ldots, n_k \rangle$
    else: return failure
end procedure BoundedDFSearch

Iterative deepening complexity

Complexity with solution at depth k and branching factor b:

level	breadth-first	iterative deepening	# nodes
1	1	k	b
2	1	$k-1$	b^2
$\ldots$	$\ldots$	$\ldots$	$\ldots$
$k-1$	1	2	b^{k-1}
k	1	1	b^k
total	$\geq b^k$	$\leq b^k \left(\frac{b}{b-1}\right)^2$	

Numerical comparison for $k = 5$ and $b = 10$:

$$\#(\text{BFS}) = 10 + 100 + 1,000 + 10,000 + 100,000 = 111,110$$
$$\#(\text{IDS}) = 50 + 400 + 3,000 + 20,000 + 100,000 = 123,450$$

Note: IDS recalculates shallow nodes several times, but this doesn't have a big effect compared to BFS!

Outline

Direction of search

- The definition of searching is symmetric: find path from start nodes to goal node or from goal node to start nodes.
- Forward branching factor: number of arcs out of a node.
- Backward branching factor: number of arcs into a node.
- Search complexity is b^n. Should use forward search if forward branching factor is less than backward branching factor, and vice versa.
- Note: when graph is dynamically constructed, the backwards graph may not be available.

Bidirectional search

- Idea: search backward from the goal and forward from the start simultaneously.
- This wins as $2b^{k/2} \ll b^k$. This can result in an exponential saving in time and space.
- The main problem is making sure the frontiers meet.
- This is often used with one breadth-first method that builds a set of locations that can lead to the goal. In the other direction another method can be used to find a path to these interesting locations.

Island driven search

- Idea: find a set of islands between s and g.

$$s \longrightarrow i_1 \longrightarrow i_2 \longrightarrow \ldots \longrightarrow i_{m-1} \longrightarrow g$$

  This gives us m smaller problems rather than 1 big problem.
- This can win as $mb^{k/m} \ll b^k$.
- The problem is to identify the islands that the path must pass through. It is difficult to guarantee optimality.
- The subproblems can be solved using islands $\Longrightarrow$ hierarchy of abstractions.