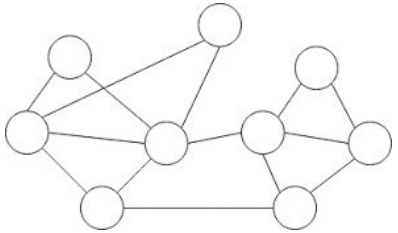
Simple Search

Jacopo Mauro

Slide Based on Slides of the Artificial Intelligence: A Modern Approach book

Problem-solving agents



(Problem-solving) agents need to find sequences of actions that achieve a desired goal.

Ideally they should be:

- Goal-Oriented: Operate with a clear objective to be achieved.
- Search-Based: Explore possible action sequences to find a solution.
- Knowledge-Independent: Can function in environments with little prior knowledge.

What do we need to define their task:

- **State space**: All the possible states the system can be
- **Initial State**: The starting point of the problem.
- **Actions and Transition Model**: Possible steps the agent can take and their consequences
- **Goal**: Criteria to determine if the goal is reached.
- Action **cost** function: A measure to evaluate the efficiency of solutions.

Problem types

Deterministic, fully observable ⇒ single-state problem

- Agent knows exactly which state it will be in; solution is a sequence

Non-observable ⇒ conformant problem

- Agent may have no idea where it is; solution (if any) is a sequence

Nondeterministic and/or partially observable ⇒ contingency problem percepts provide new information about current state

- solution is a contingent plan or a policy often interleave search, execution

Unknown state space ⇒ exploration problem (“online”)

Example: vacuum world

Single-state, start in #5.

Solution → [*Right*, *Suck*]

Conformant, start in {1, 2, 3, 4, 5, 6, 7, 8}

e.g., *Right* goes to {2, 4, 6, 8}.

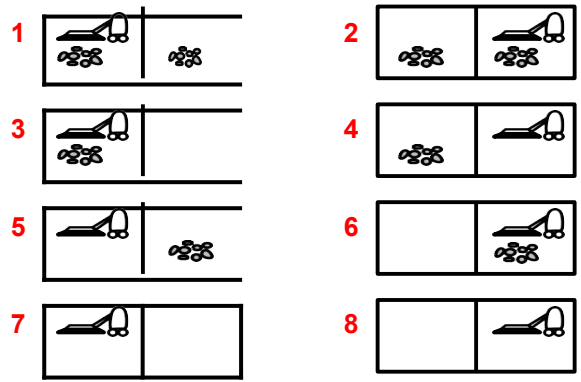
Solution → [*Right*, *Suck*, *Left*, *Suck*]

Contingency, start in #5

Murphy's Law: *Suck* can dirty a clean carpet

Local sensing: dirt, location only.

Solution → [*Right*, if *dirt* then *Suck*]



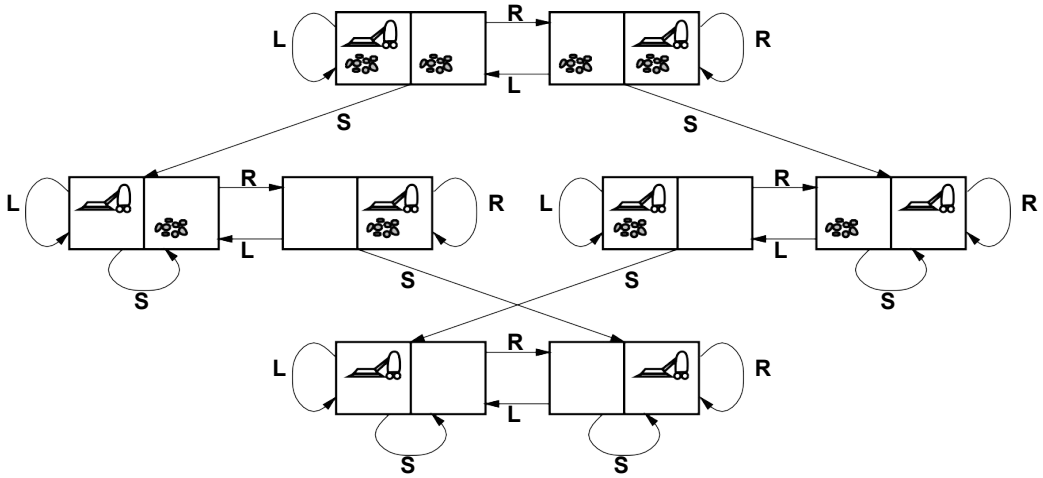
Example: vacuum world state space graph

states: boolean dirty and robot locations (ignore dirt amounts etc.)

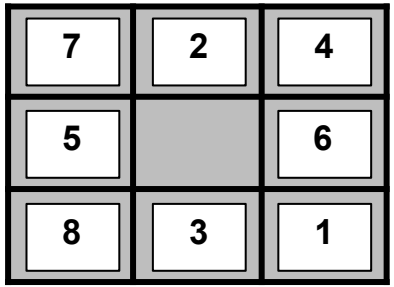
actions: Left, Right, Suck, NoOp

goal test: no dirty

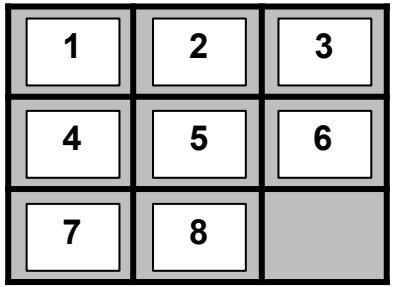
path cost: 1 per action (0 for NoOp)



Example: The 8-puzzle



Start State



Goal State

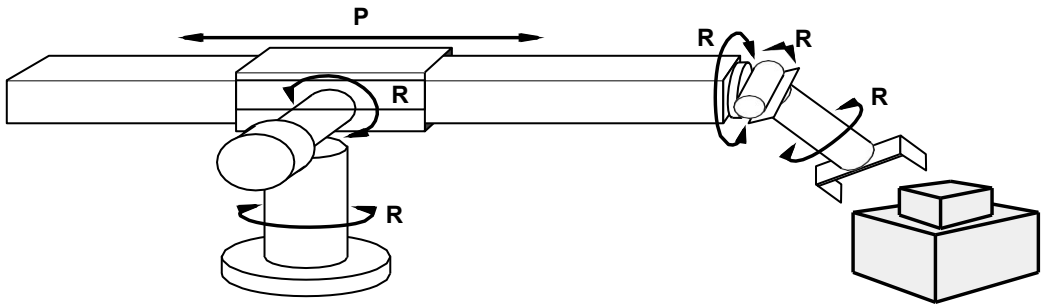
states: integer locations of tiles

actions: move blank left, right, up, down (ignore unjamming etc.)

goal test = numbers order from top to bottom, left to right

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

path cost: time to execute

Tree search algorithms

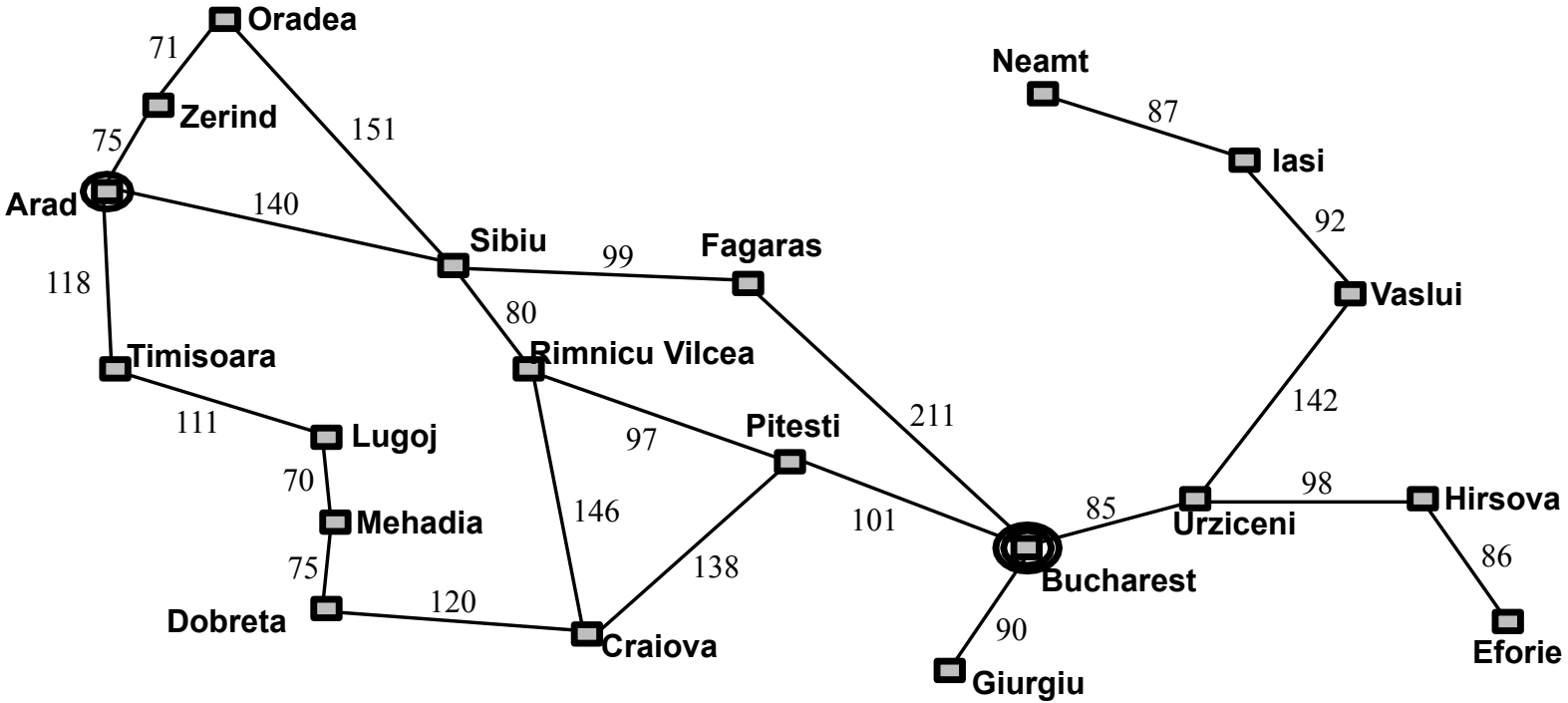
Basic idea: see search as exploring a tree offline

State = Nodes of the tree

Root initial state.

Generate successors of already-explored states (a.k.a. expanding states) until goal is reached.

Example: Travel in Romania



Example: Travel in Romania

Currently in Arad. Flight leaves tomorrow from Bucharest

Formulate goal:

- be in Bucharest

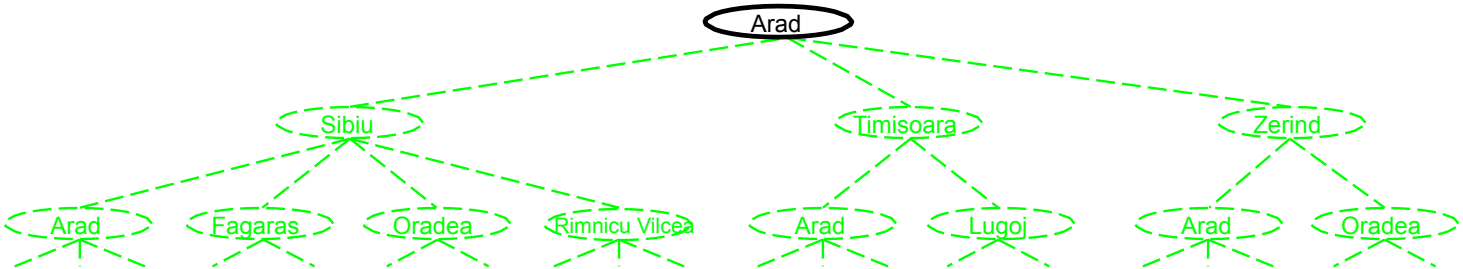
Formulate problem:

- states: various cities
- actions: drive between cities

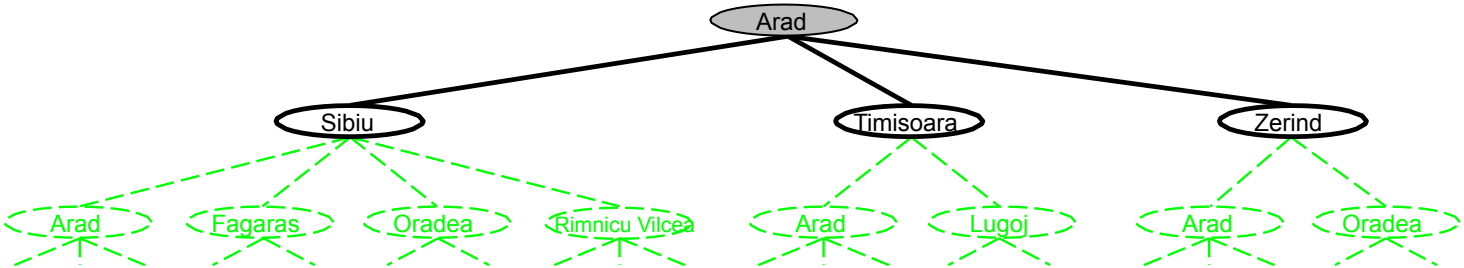
Find solution:

- sequence of cities, e.g., Arad, Sibiu, Fagaras, Bucharest

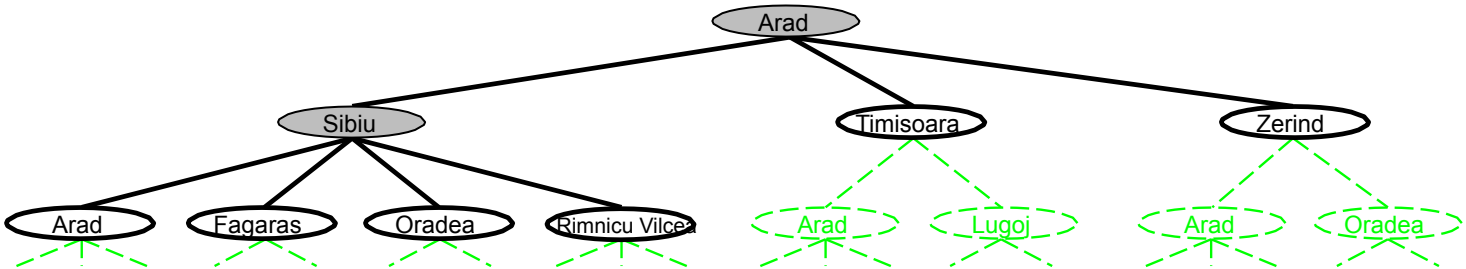
Tree search example



Tree search example



Tree search example



Search strategies

A strategy is defined by picking the order of node expansion

Strategies are evaluated along the following dimensions:

- Completeness: does it always find a solution if one exists?
- Time complexity: number of nodes generated/expanded
- Space complexity: maximum number of nodes in memory
- Optimality: does it always find a least-cost solution?

Uninformed search strategies

Uninformed strategies use only the information available in the problem definition

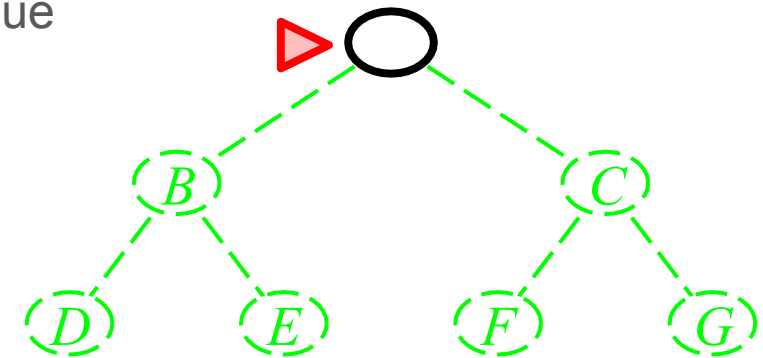
- Breadth-first search
- Depth-first search
- Depth-limited search
- Iterative deepening search

Note: you will see some of these algorithms again in Algorithms and Data Structure in far more details

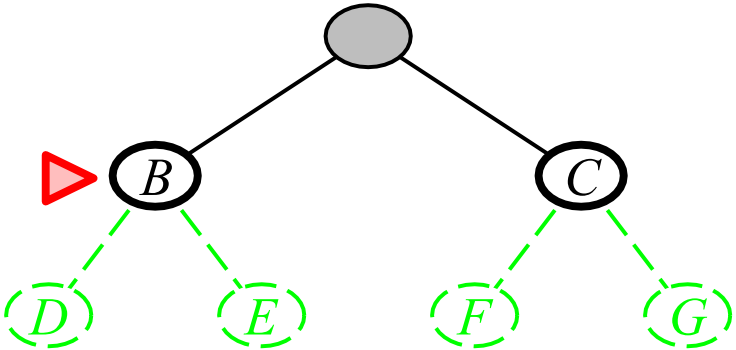
Breadth-first search

Expand shallowest unexpanded node

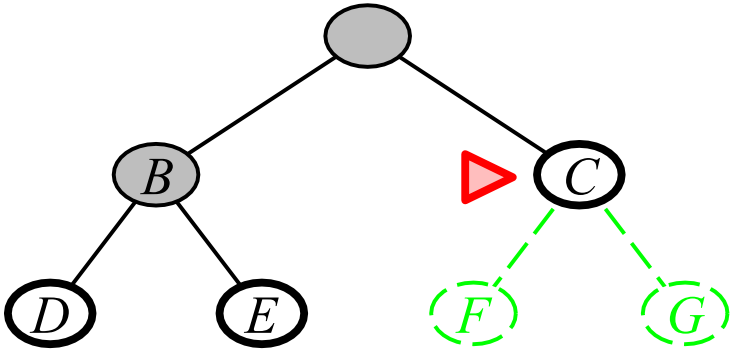
Fringe: FIFO queue



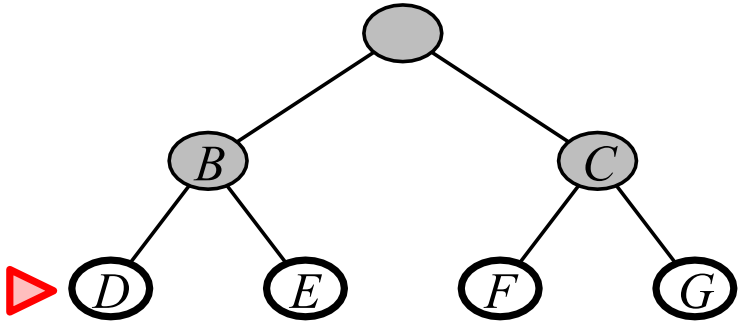
Breadth-first search



Breadth-first search



Breadth-first search



Properties of breadth-first search

Complete (if branching factor b is finite)

Time: exp in depth of the tree

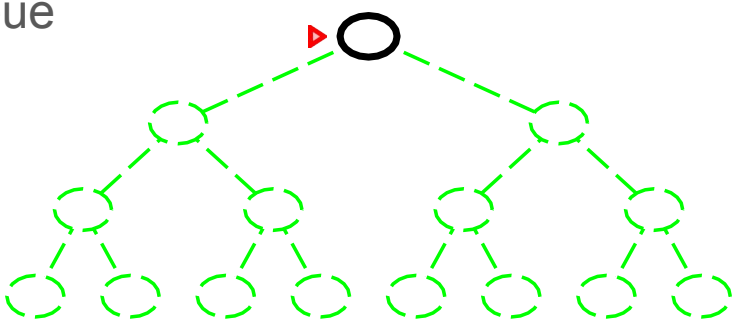
Space: $O(b^d)$ (keeps every node in memory) → can become a big problem

Optimal: Yes if costs are all the same, otherwise no

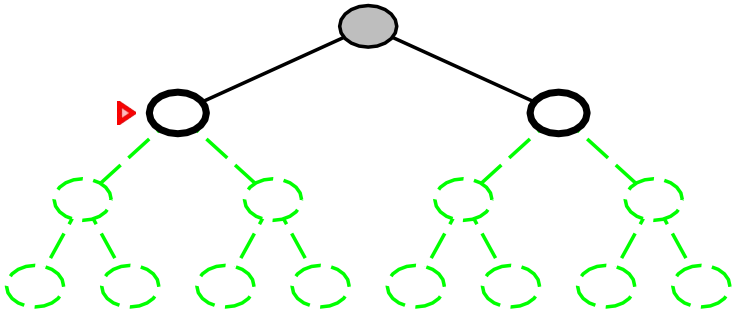
Depth-first search

Expand deepest unexpanded node

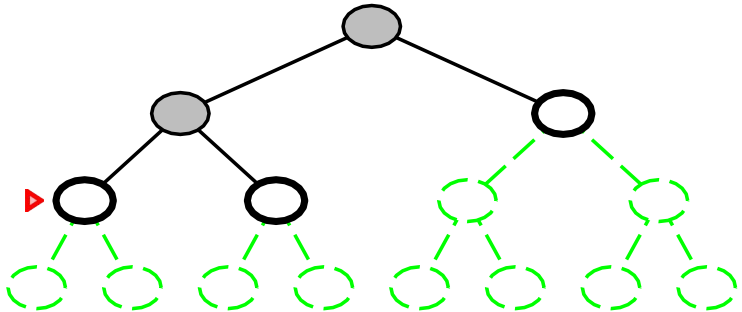
Fringe = LIFO queue



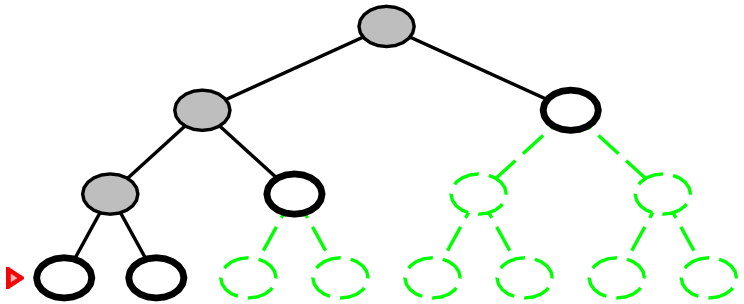
Depth-first search



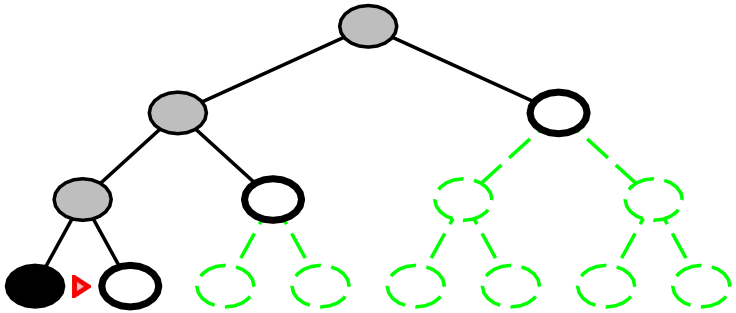
Depth-first search



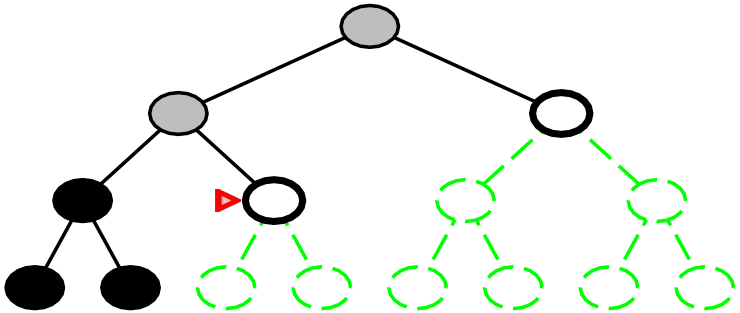
Depth-first search



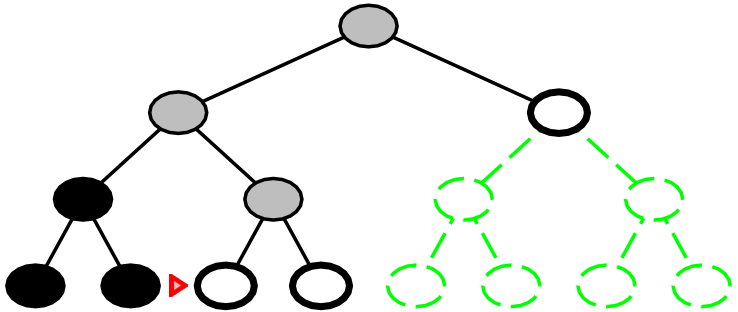
Depth-first search



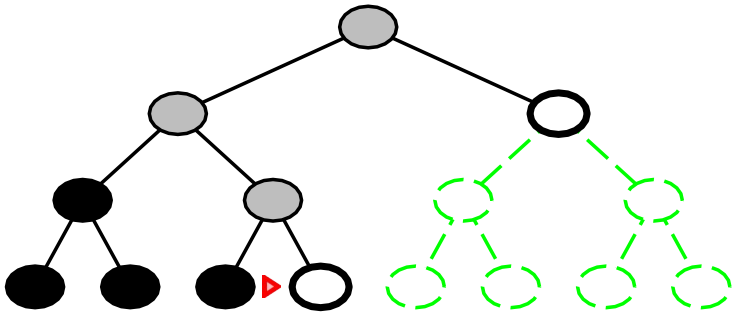
Depth-first search



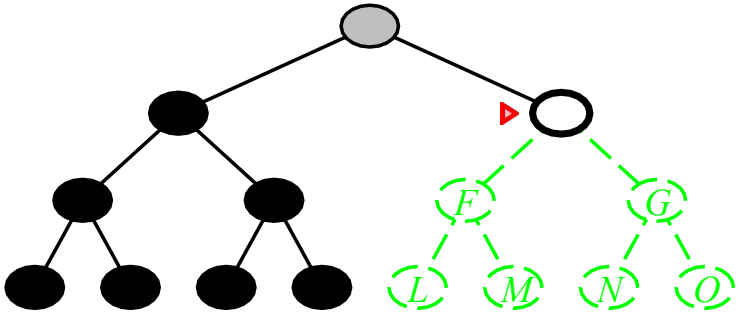
Depth-first search



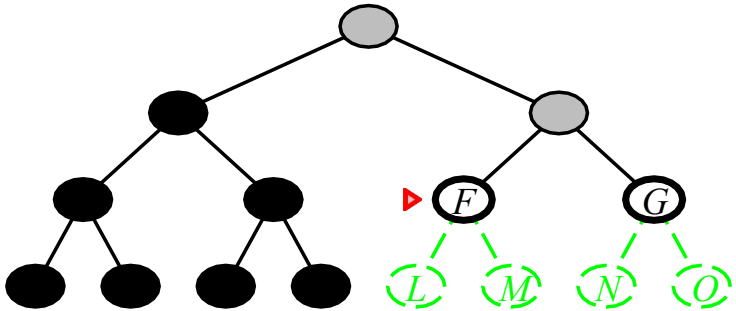
Depth-first search



Depth-first search



Depth-first search



Properties of depth-first search

Complete: No → fails in infinite-depth spaces, spaces with loops

- modify to avoid repeated states along path ⇒ complete in finite spaces

Time: very bad if the goal is in the last part of the search space considered

Space: maximal depth of the tree

Optimal: No

Iterative deepening search $l = 0$

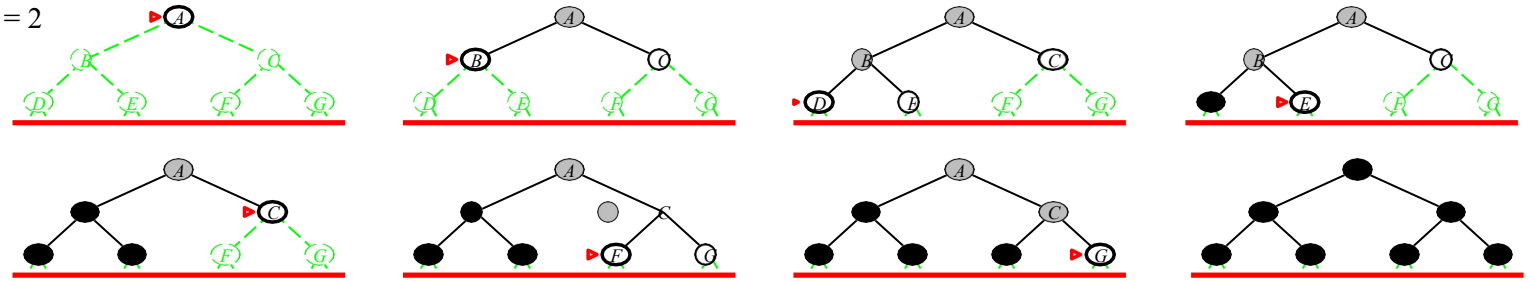
Limit = 0

Iterative deepening search $l = 1$

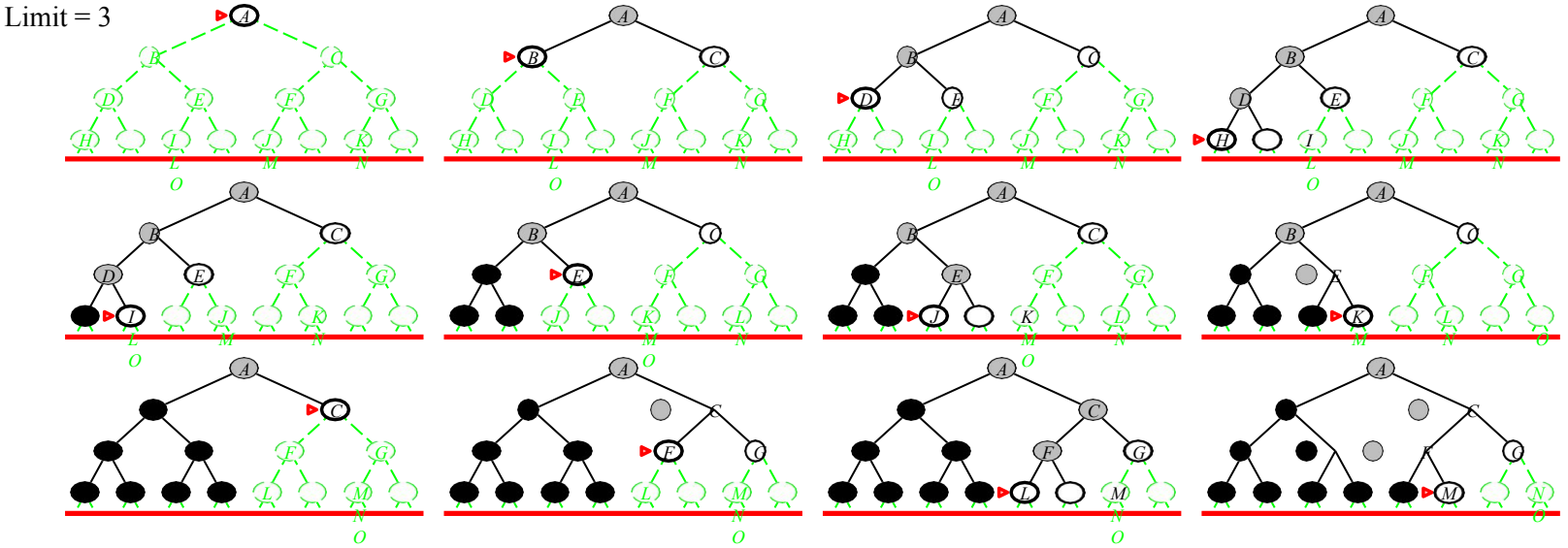
Limit = 1

Iterative deepening search $l = 2$

Limit = 2



Iterative deepening search $l = 3$



Properties of iterative deepening search

Complete: Yes

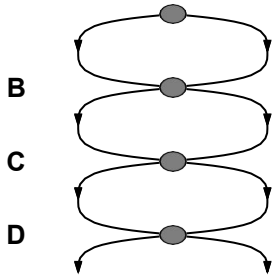
Time: max $O(b^d)$

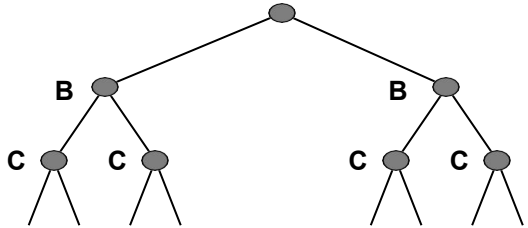
Space: max $O(bd)$

Optimal: Yes, if step cost = 1

Repeated states

Failure to detect repeated states can turn a linear problem into an exponential one!





PseudoCode

```
function BEST-FIRST-SEARCH(problem, f) returns a solution node or failure
  node ← NODE(STATE=problem.INITIAL)
  frontier ← a priority queue ordered by f, with node as an element
  reached ← a lookup table, with one entry with key problem.INITIAL and value no
  while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    if problem.IS-GOAL(node.STATE) then return node
    for each child in EXPAND(problem, node) do
      s ← child.STATE
      if s is not in reached or child.PATH-COST < reached[s].PATH-COST then
        reached[s] ← child
        add child to frontier
  return failure

function EXPAND(problem, node) yields nodes
  s ← node.STATE
  for each action in problem.ACTIONS(s) do
    s' ← problem.RESULT(s, action)
    cost ← node.PATH-COST + problem.ACTION-COST(s, action, s')
    yield NODE(STATE=s', PARENT=node, ACTION=action, PATH-COST=cost)
```

Forward, Backward, Bidirectional

- Forward Search
  - Starts from the initial state.
  - Explores possible actions to reach the goal state.
- Backward Search
  - Starts from the goal state.
  - Works backwards to find a path to the initial state.
  - Good when easier to determine predecessors of the state than successors.
- Bidirectional Search
  - Searches simultaneously from the initial state and the goal.
  - Meets in the middle, reducing search depth.
  - Each search (forward and backward) could explores ~roughly half the depth of the solution space → reduce the number of nodes visited

Informed search strategies

Use problem-specific knowledge to guide exploration and improve efficiency

Key Concepts:

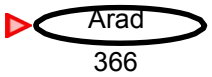
- Heuristic Function (h(n)): Estimates the cost to reach the goal from a given state.
- Evaluation Function (f(n)): Combines heuristic and path cost for decision-making (e.g., f(n) = g(n) + h(n)).

Greedy search

E.g. of heuristics: hSLD(n) = straight-line distance from n to Bucharest

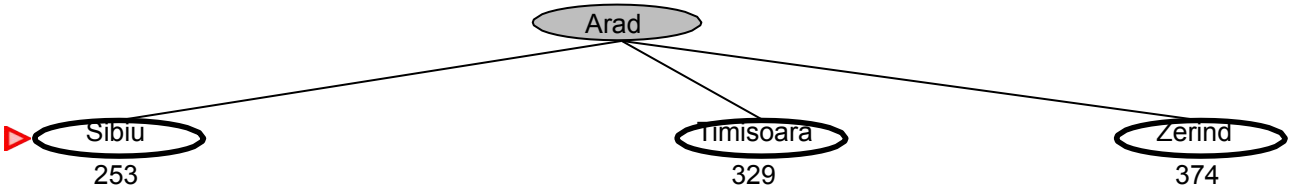
Greedy search expands the node that appears to be closest to goal

Greedy search example



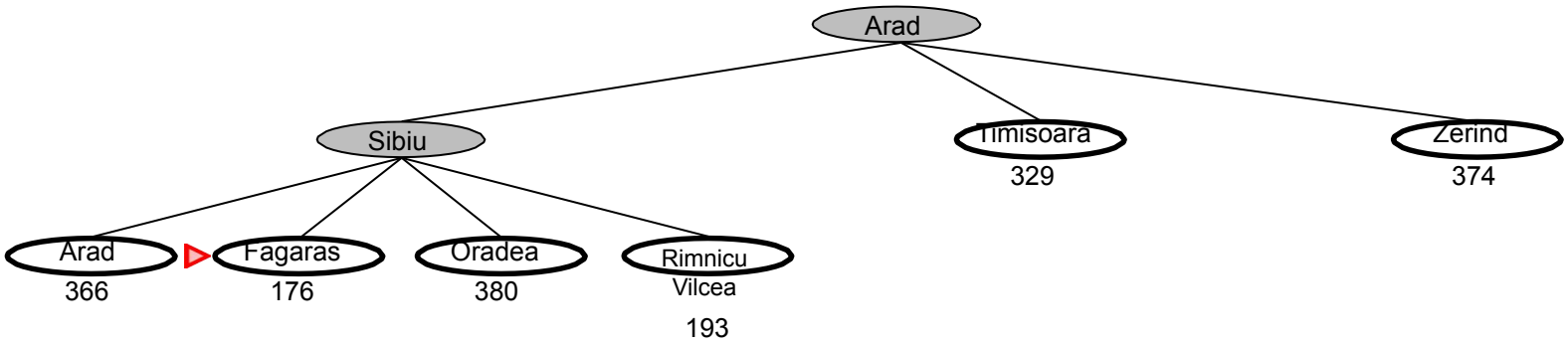
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Greedy search example

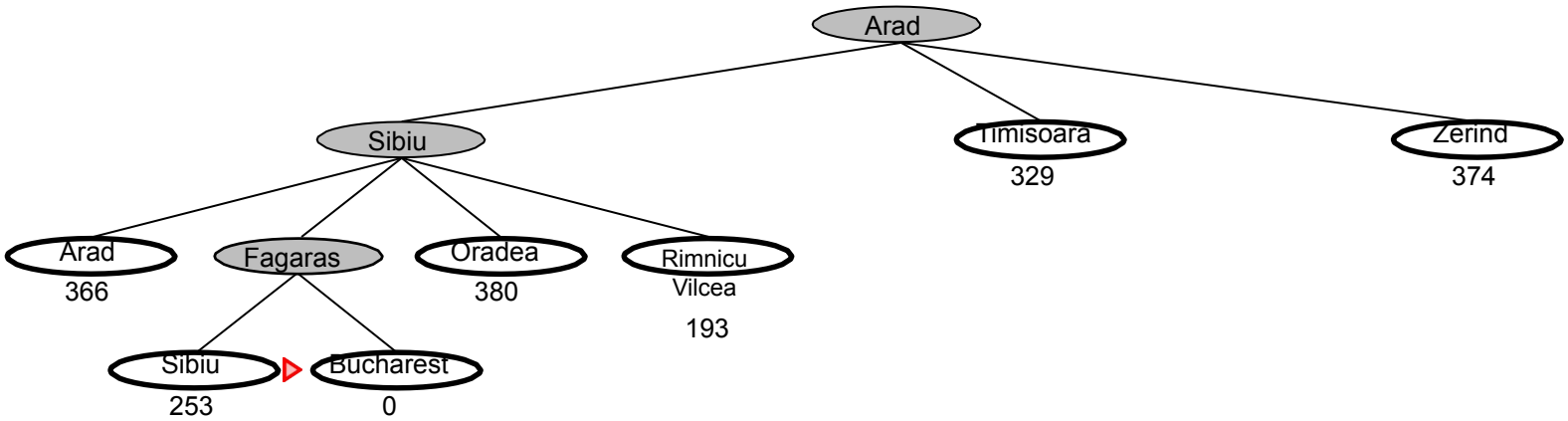


Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Greedy search example



Greedy search example



Properties of greedy search

Complete: No. Can get stuck in loops (e.g., Iasi → Neamt → Iasi → Neamt)

- Complete in finite space with repeated-state checking

Time: similar to deep search but a good heuristic can give dramatic improvement

Space: similar to deep search

Optimal: No

A* Search

Idea: avoid expanding paths that are already expensive

Evaluation function f(n) = g(n) + h(n)

- g(n) = cost so far to reach n
- h(n) = estimated cost to goal from n

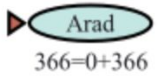
A∗ search uses an **admissible heuristic**

- h(n) ≤ h^*(n) where h^*(n) is the true cost from n (i.e., never overestimates)
- h(n) ≥ 0 + h(G) = 0 for any goal G

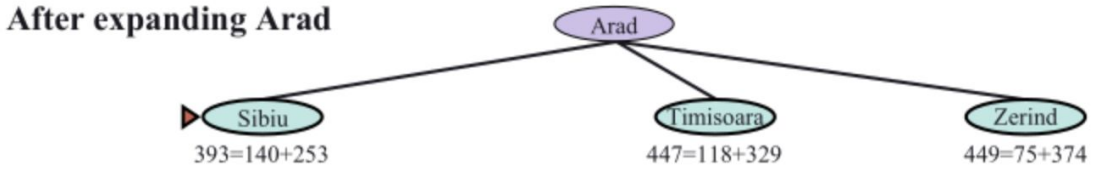
Theorem: A∗ search is optimal

A* Search

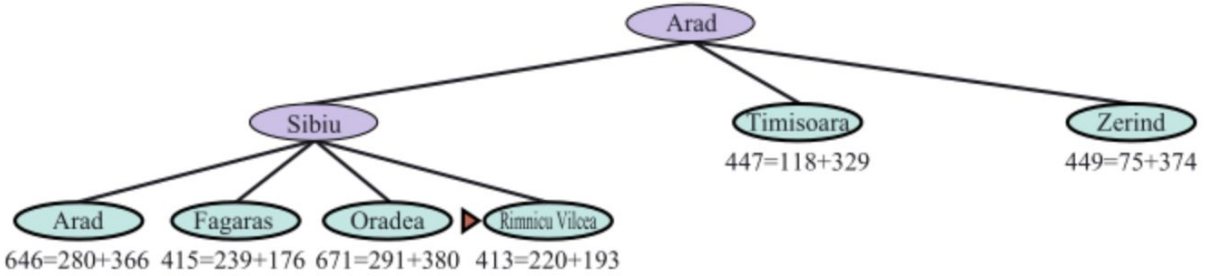
The initial state



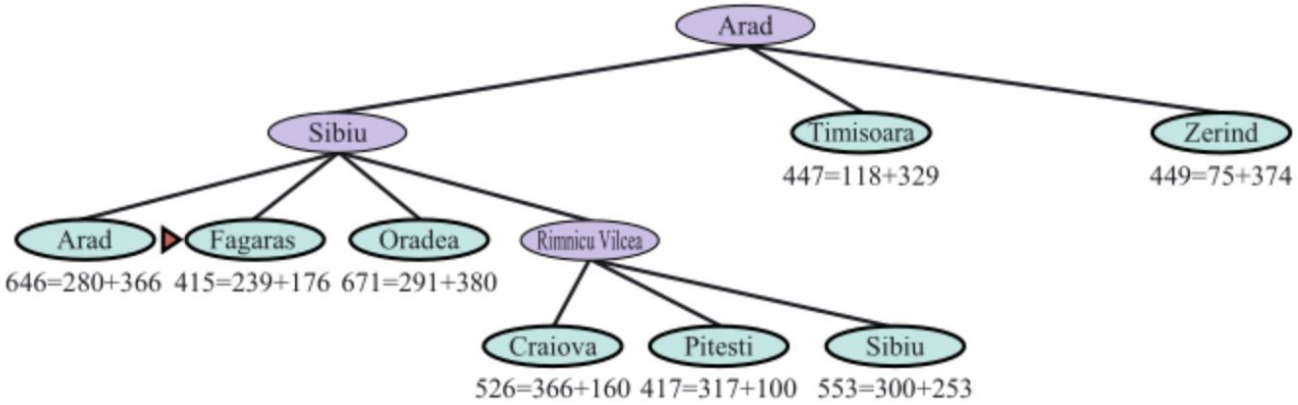
A* Search


After expanding Arad
Arad
Sibiu
393=140+253
Timisoara
447=118+329
Zerind
449=75+374

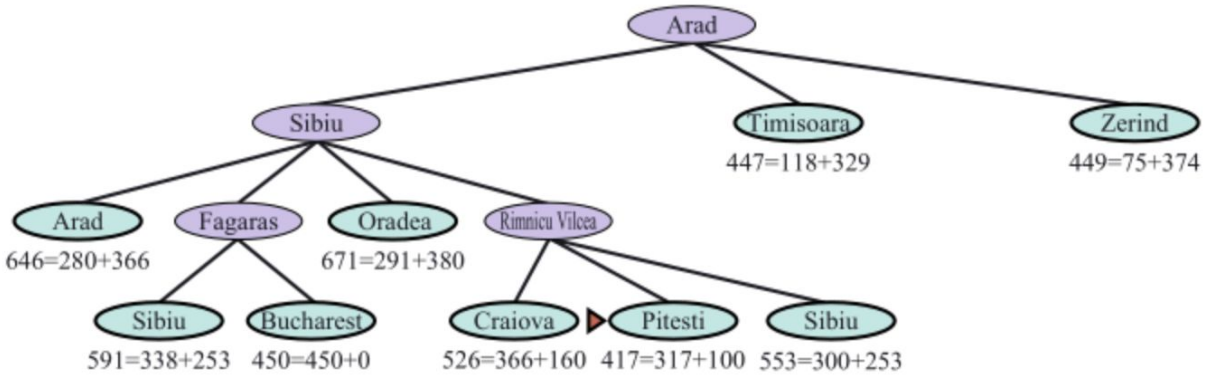
A* Search


Arad
Sibiu
Timisoara
Zerind
447=118+329
449=75+374
Arad
Fagaras
Oradea
Rimnicu Vilcea
646=280+366
415=239+176
671=291+380
413=220+193

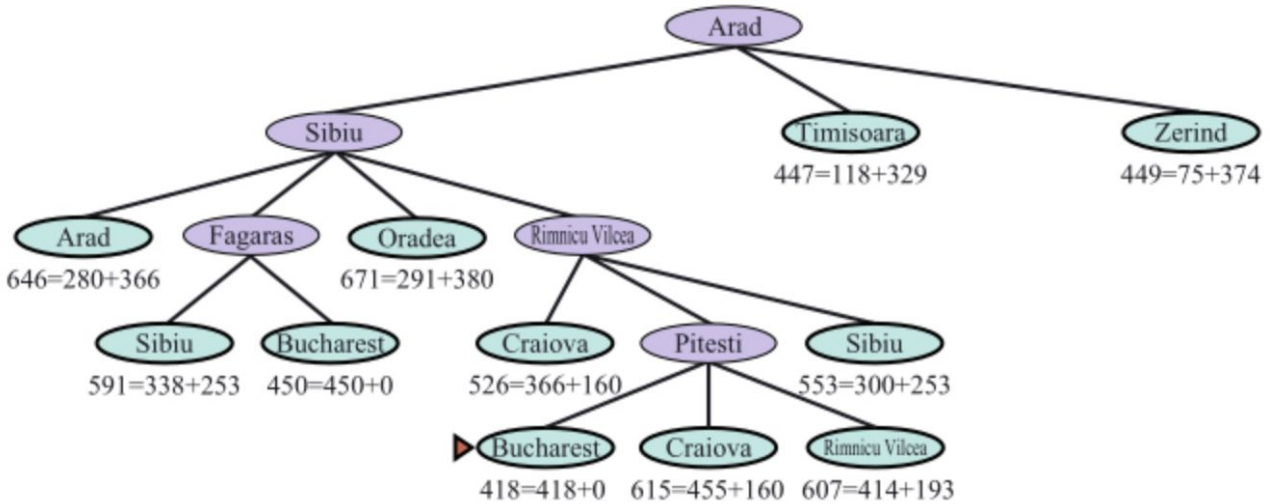
A* Search


Arad
Sibiu
Timisoara
Zerind
447=118+329
449=75+374
Arad
Fagaras
Oradea
Rimnicu Vilcea
646=280+366
415=239+176
671=291+380
Craiova
Pitesti
Sibiu
526=366+160
417=317+100
553=300+253

A* Search


Arad
Sibiu
Timisoara
447=118+329
Zerind
449=75+374
Arad
646=280+366
Fagaras
Oradea
671=291+380
Rimnicu Vilcea
Sibiu
591=338+253
Bucharest
450=450+0
Craiova
526=366+160
Pitesti
417=317+100
Sibiu
553=300+253

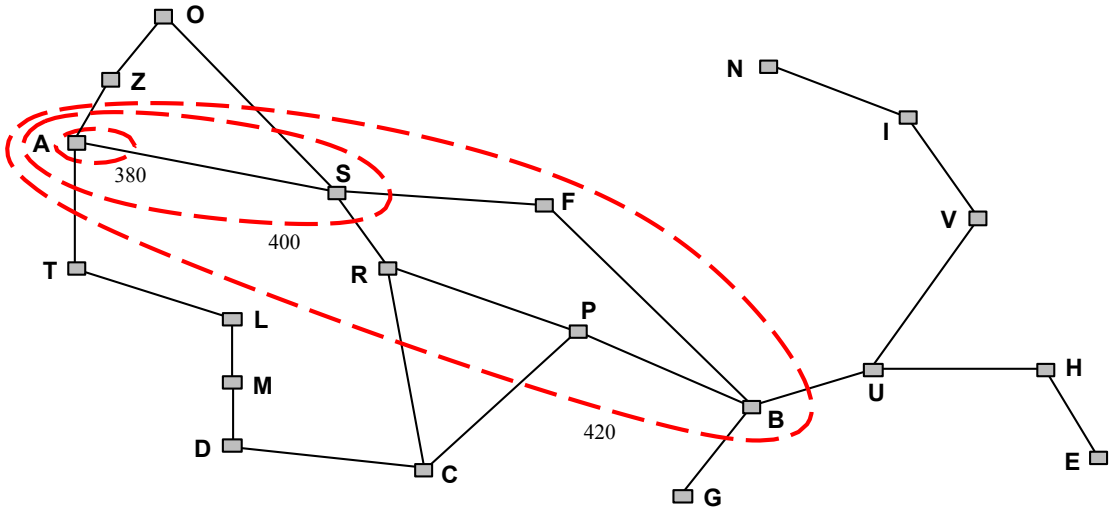
A* Search


Arad
Sibiu
Timisoara
447=118+329
Zerind
449=75+374
Arad
646=280+366
Fagaras
Oradea
671=291+380
Rimnicu Vilcea
Sibiu
591=338+253
Bucharest
450=450+0
Craiova
526=366+160
Pitesti
Sibiu
553=300+253
Bucharest
418=418+0
Craiova
615=455+160
Rimnicu Vilcea
607=414+193

Idea behind Optimality of A*

A* expands nodes in order of increasing f value

Gradually adds “f-contours” of nodes (cf. breadth-first adds layers)



Properties of A*

Complete: Yes

Time: exponential in [relative error in h × length of soln.]

Space: keeps all nodes in memory

Optimal: Yes

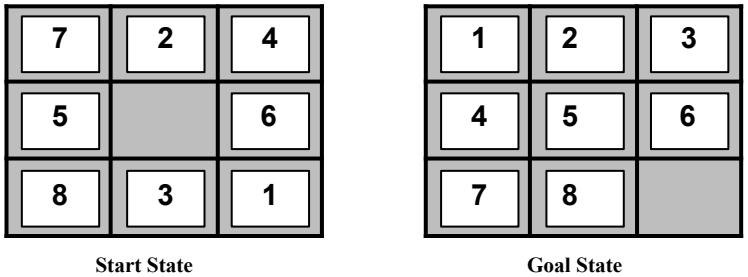
- It does not expand any nodes that exceed the optimal cost

Admissible heuristics

E.g., for the 8-puzzle:

h1(n) = number of misplaced tiles

h2(n) = total Manhattan distance (i.e., no. of squares from desired location of each tile)



$h_1(S)$ = 6
$h_2(S)$ = 4+0+3+3+1+0+2+1 = 14

Dominance

If h2(n) ≥ h1(n) for all n (both admissible), then h2 dominates h1 and is better for search

Given any admissible heuristics ha, hb we have that:

- h(n) = max(ha(n), hb(n)) is also admissible and dominates ha, hb

Relaxed problems

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

If the rules of the 8-puzzle are relaxed so that a tile can move anywhere, then h1(n) gives the shortest solution

If the rules are relaxed so that a tile can move to any adjacent square, then h2(n) gives the shortest solution

Key point: the optimal solution cost of a relaxed problem is no greater than the optimal solution cost of the real problem

Homework

Read Chapter 3

Note that a lot of discussion on the complexity will be clearer after seeing this concepts in Algorithms and Data Structures. For the time being do not pay too much attention to complexity results (but revise the chapter later on before the exam to fully grasp all the details)