Artificial Intelligence

3. Search

Shashi Prabh

School of Engineering and Applied Science
Ahmedabad University

Contents

Goal: use search to solve problems

Topics

- Problem-solving agents
  - Steps
- Searching
  - Uninformed Search
  - Informed Serach

Problem-Solving Agents

- Finds a *sequence* of actions that form a path to the goal state(s)
- Steps
  - Goal Formulation: limits the action choices
  - Problem formulation: a description of the states and actions to reach the goal
  - Search: simulates sequences of actions in its model, produces solution
    - In partially observable or nondeterministic environments, the solution would be a branching strategy
  - Execution
    - In fully observable, deterministic and known environment, the agent can ignore the percepts – open loop system
    - Otherwise, percepts need to be monitored – closed loop system

Navigation example

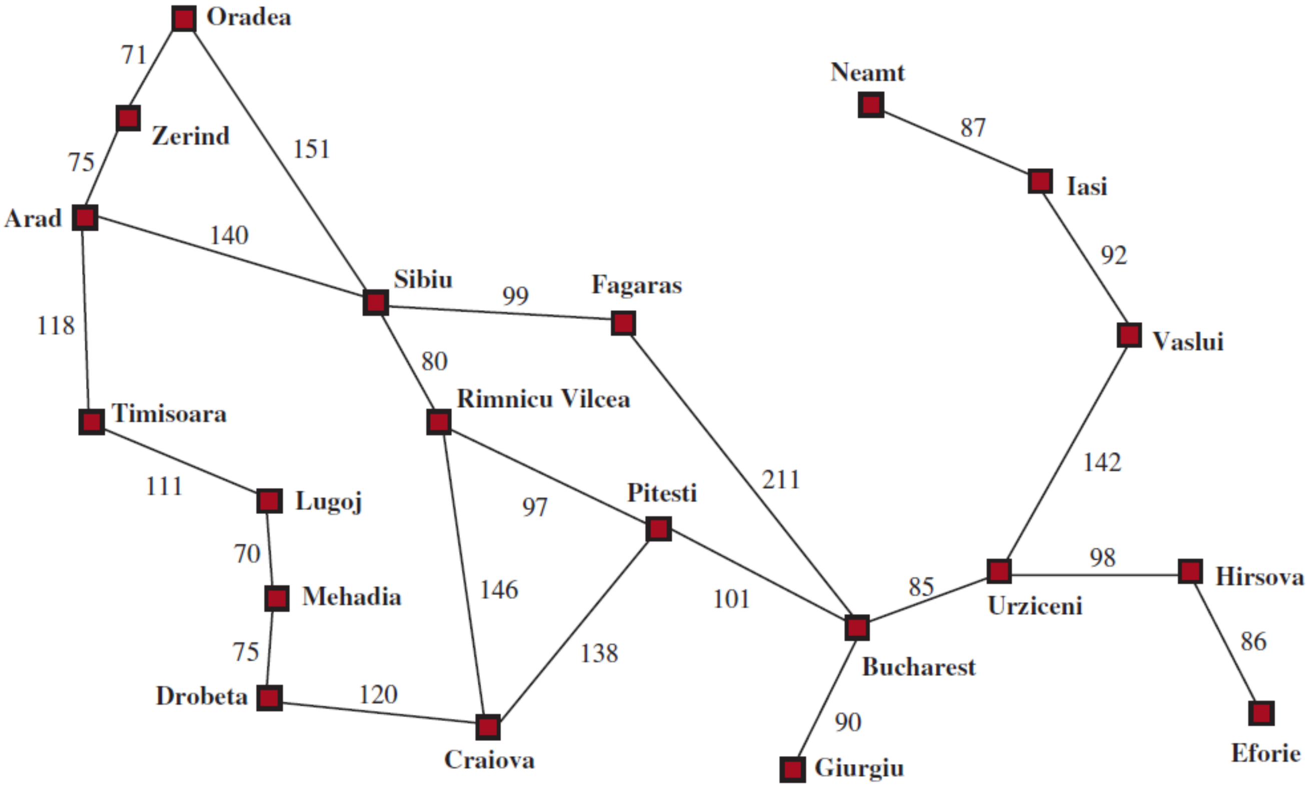


Figure 3.1 A simplified road map of part of Romania, with road distances in miles.

Defining a search problem

- A set of possible states that the environment can be in. We call this the state space.
  - The initial state that the agent starts in. For example: *Arad*. State space
- A set of one or more goal states
- The actions available to the agent. Given a state *s*, ACTIONS(*s*) returns a finite set of actions that can be executed in *s*
  - We say that each of these actions is applicable in *s*
  - ACTIONS (*Arad*) = {*ToSibiu*,*ToTimisoara*,*ToZerind*}
- Transition model, which describes what each action does
  - RESULT(*s*, *a*) returns the state that results from doing action *a* in state *s*
  - RESULT(*Arad*, *ToZerind*) = *Zerind* .

Defining a search problem

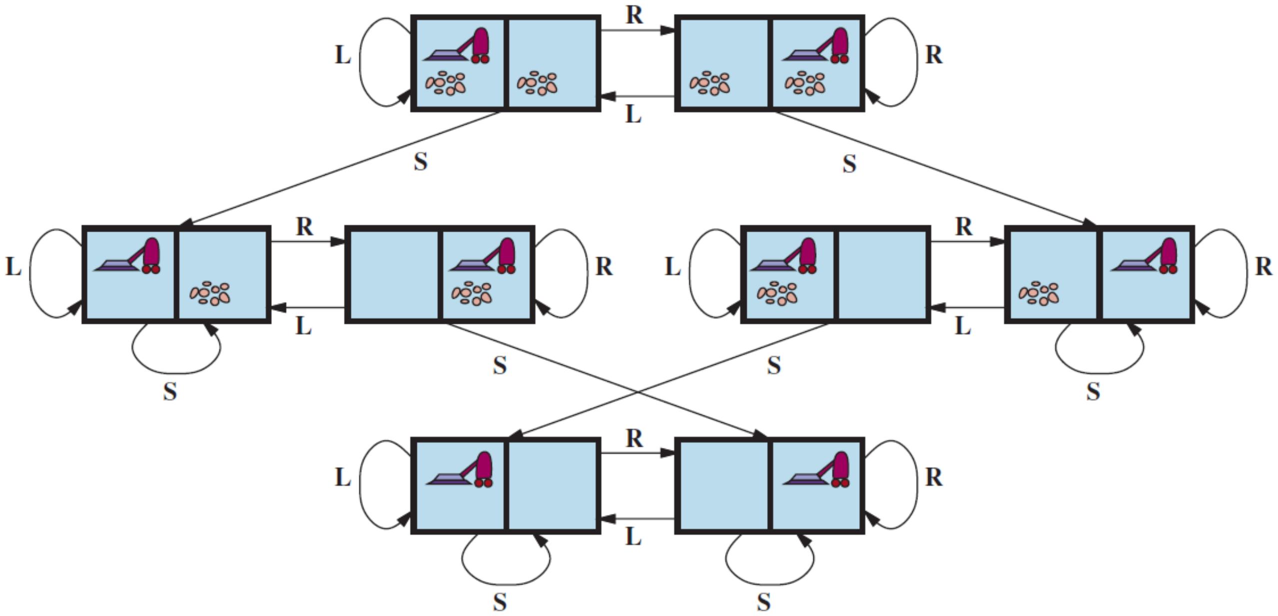
- An action cost function, denoted by ACTION-COST(s, a, s') that gives the numeric cost of applying action a in state s to reach state s'
  - $c(s, a, s')$ when we are doing math
  - Should use a cost function that reflects its own performance measure
- Example: for route-finding agents, the cost of an action might be the length in miles or it might be the time it takes to complete the action.
- A sequence of actions forms a path, and a solution is a path from the initial state to a goal
- An optimal solution has the lowest path cost among all solutions.

Vacuum-World Example

- States: 8 states
  - Agent can be in either of the two cells, and each call can have dirt or not
- Initial state: Any one of the 8 states
- Actions: Suck, MoveLeft, and MoveRight
  - In a 2-D multi-cell world Forward, Backward, TurnRight, and TurnLeft.
- Transition model: Suck removes any dirt from the agent's cell, move left/right takes the agent to the other room unless it hits a wall, in which case the action has no effect
- Goal states: The states in which every cell is clean
- Action cost: +1 for each action

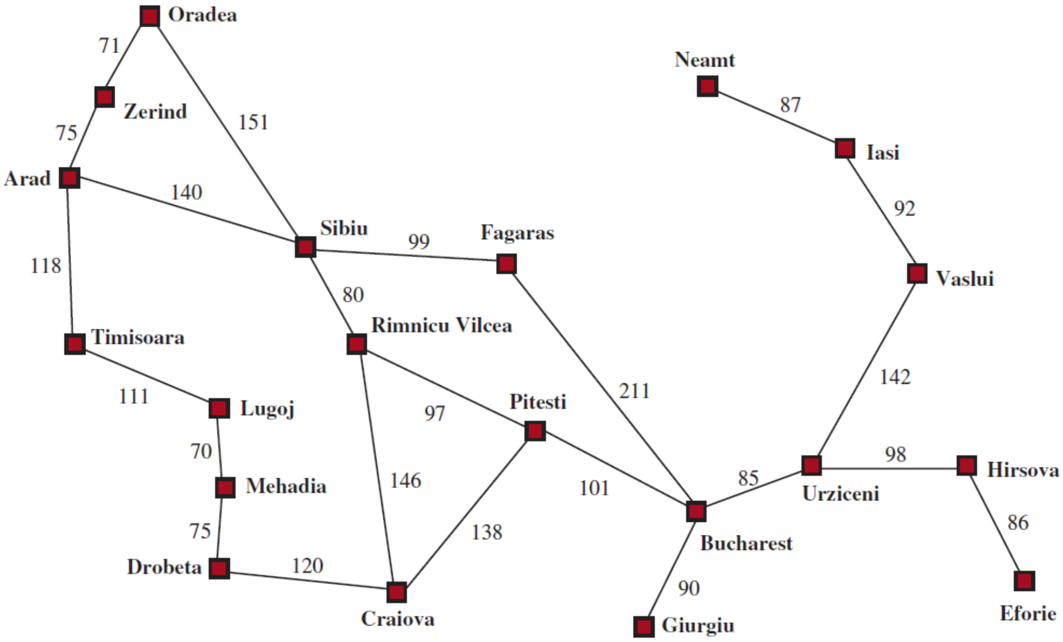
State-Space Graph

- The state space can be represented as a graph in which the vertices are states and the directed edges between them are actions.



Navigation example

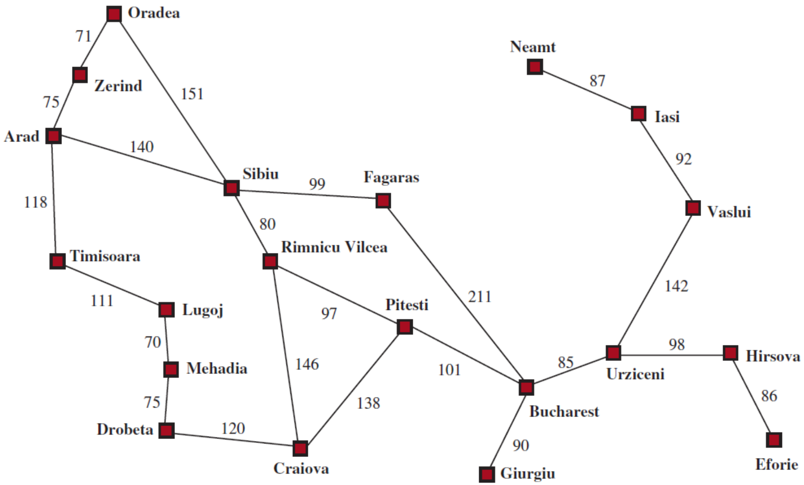
- Here the map is a state-space graph
- Each road indicates two actions, one in each direction.



Search Algorithms

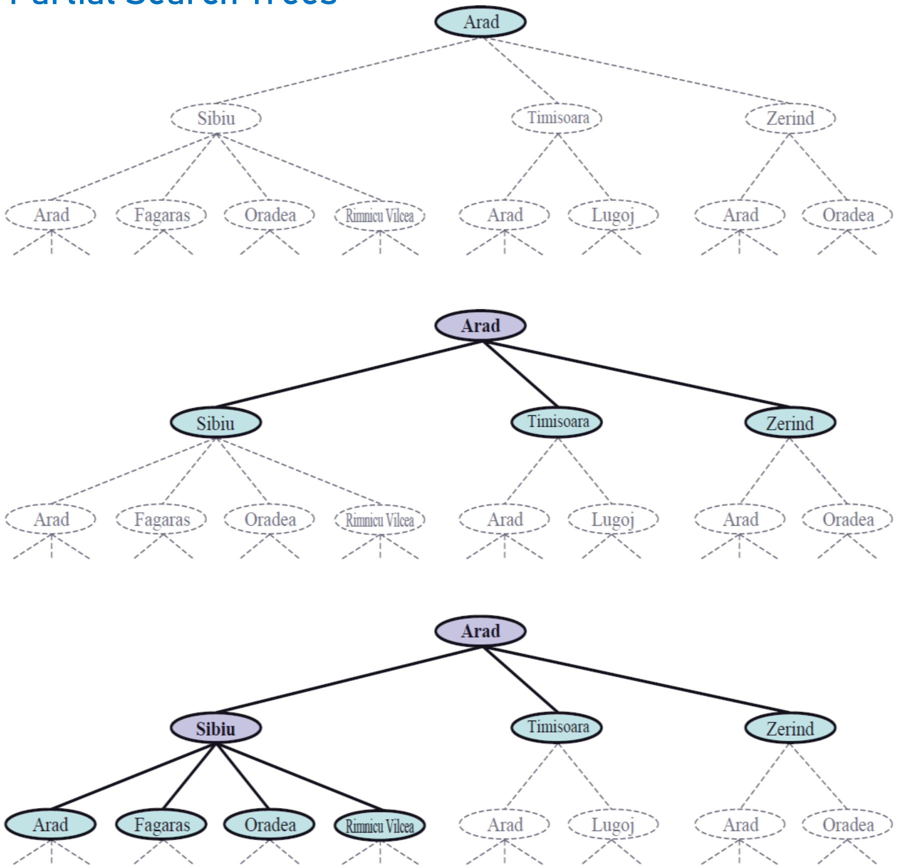
- A search algorithm takes a search problem as input and returns a solution or indicates failure
- Search Tree
  - Node corresponds to a state in the state space
  - Edge corresponds to an action
  - The root of the tree corresponds to the initial state
  - Search tree describes paths between states leading to the goal
  - A state may appear multiple times in the search tree

Search Tree



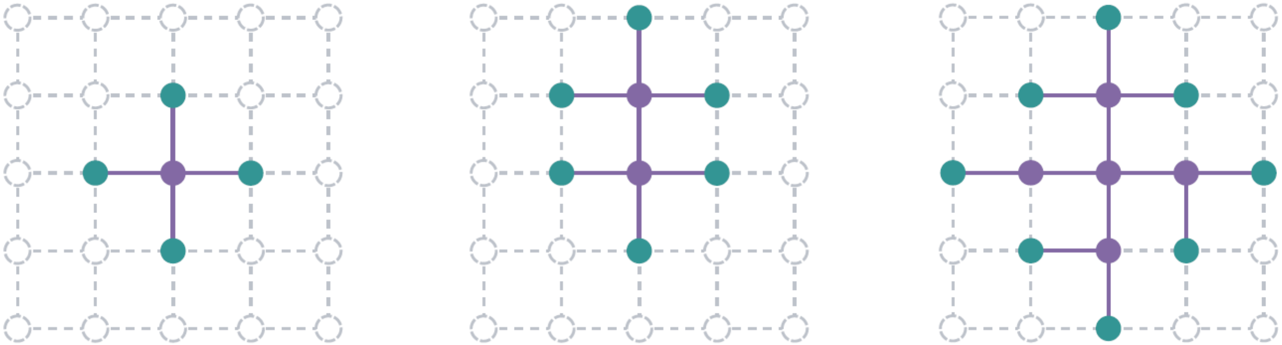
- The search tree is infinite
- State space size is only 20

Partial Search Trees



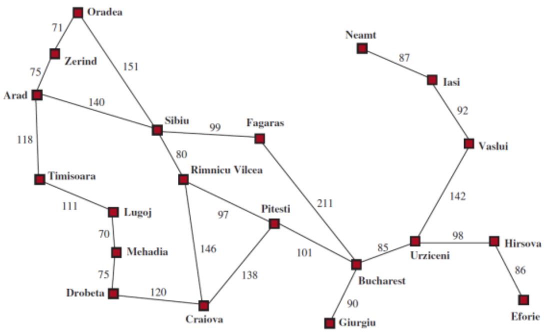
Search Tree

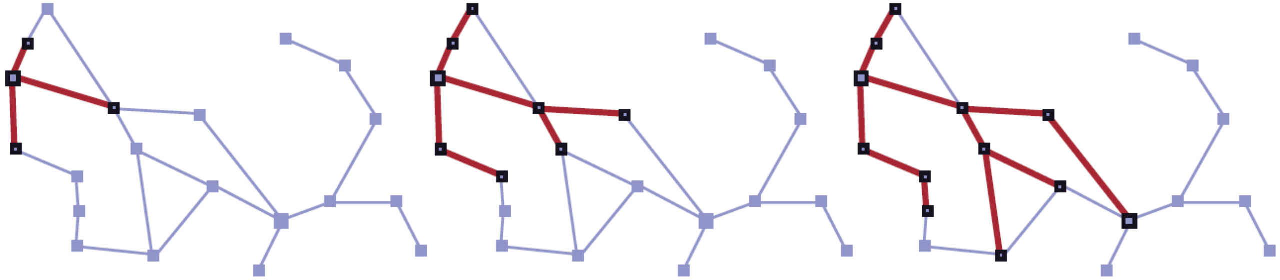
- Frontier (green) separates interior from exterior
  - A frontier node is expanded till goal is reached
- Search algorithm: which frontier node to expand next?

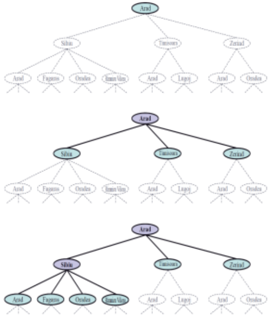


Search Tree

- We will superimpose a search tree over the state-space graph, forming various paths from the initial state, trying to find a path to a goal state



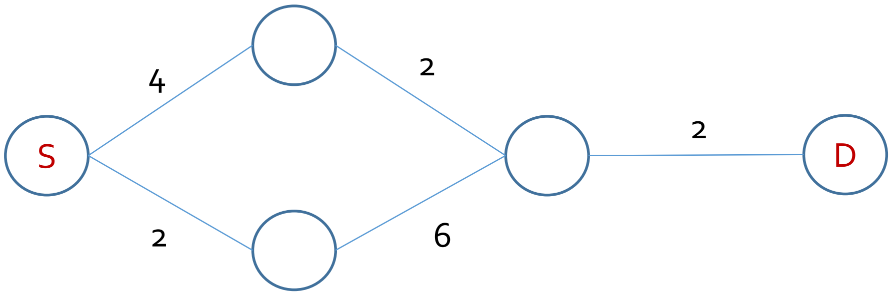




Partial Search Trees

Best-First Search

- Evaluation function for each node f(n)
  - Different f(n) result in different search algorithms…
  - f(n) can change with time
- Out of all nodes in the frontier, select the node with the smallest f(n)
  - A node may be added multiple times to the frontier if it is reached by lower cost path



Search Data Structure

- node
  - node.STATE: the state to which the node corresponds;
  - node.PARENT: the node in the tree that generated this node;
  - node.ACTION: the action that was applied to the parent’s state to generate this node
    - Why?
  - node.PATH-COST: the total cost of the path from the initial state to this node
    - g(node) : a synonym for PATH-COST.
  - Following the PARENT pointers back from a node allows us to recover the states and actions along the path to that node. Doing this from a goal node gives us the solution.

Search Data Structure

- frontier: queue
  - IS-EMPTY(frontier) returns true if no nodes in the frontier
  - POP(frontier) removes the top node from the frontier & returns it
  - TOP(frontier) returns (but does not remove) the top node
  - ADD(node, frontier) inserts node into its proper place in the queue

  - Three kinds of queues are used in search algorithms:
    - Priority queue - pops the node with the minimum cost
      - Used in Best-First Search
    - FIFO queue - pops the node that was added to the queue the earliest
      - Used in BFS
    - LIFO queue (or, stack) - pops the most recently added node
      - Used in DFS

Best-First Search

```
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 node
  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)
```

Brute-Force Search

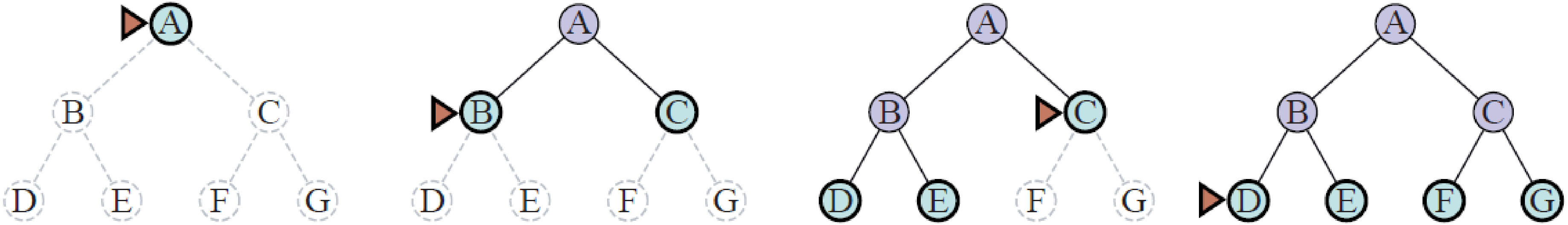
- The search space can be huge
  - Infinite if state space graph has loops
- We remove references to reached states and maintain the best path to deal with redundancy
- Example: 10x10 grid space
  - Any one of the 100 squares can be reached in at-most 9 moves
  - Approx number of paths of length 9 is 8^9 ~ 100 million paths
  - Eliminating redundancy yields roughly 1000000x speedup

Performance metrics

- Completeness: Does the algorithm always find a solution when there is one?
  - And correctly reports failure when there is none?
- Cost optimality: Does it find a solution with the lowest cost?
- Time complexity: How long does it take to find a solution?
  - The number of states and actions considered.
- Space complexity: How much memory it needs to perform the search?

Breadth-First Search

- f(n) = depth of node n
- BFS is complete, not optimal if cost varies
- Time and space complexity are $O(b^d)$, b is the branching factor and d depth. Not good…
- All the search tree nodes need to be kept in memory which is a problem with BFS
  - At 1 KB per node, the memory needed to search till depth 10 and branching factor 10 is 10 TB

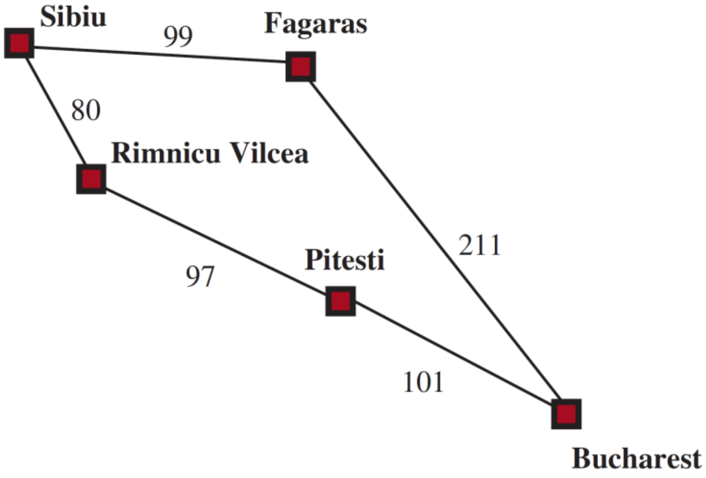


Breadth-First Search

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

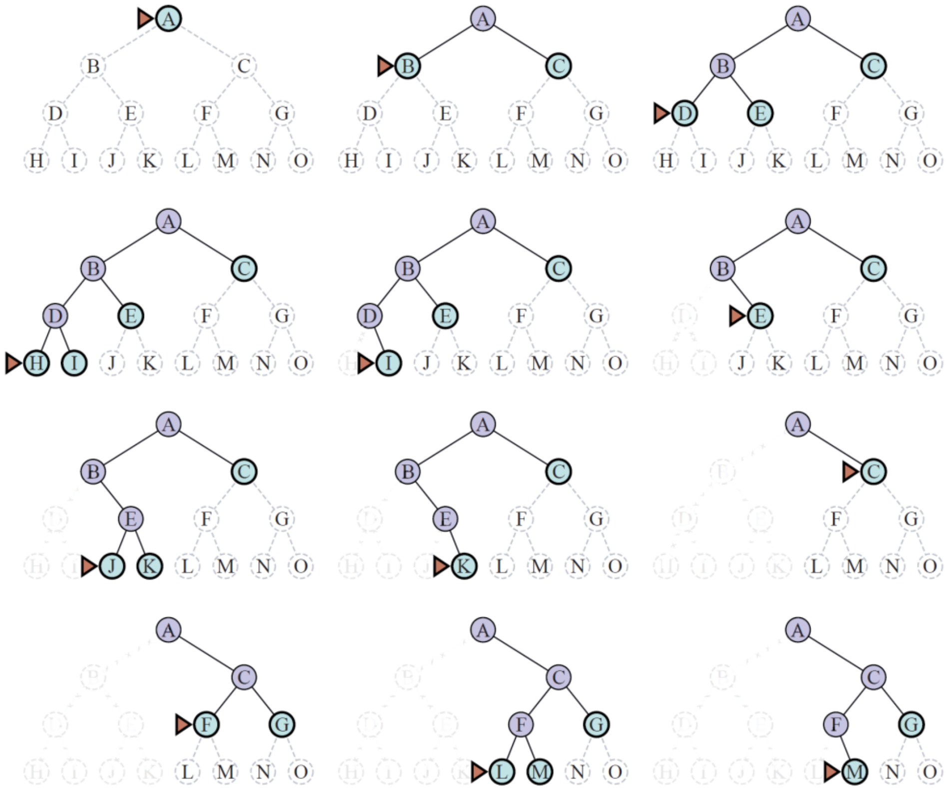
Dijkstra's Algorithm

- Uniform-cost search
  - Expand the node with the least cost first
- Complete and optimal
- Time and space complexity: $O(b^{1+\lfloor C^*/\varepsilon \rfloor})$
  - Can be worse than BFS



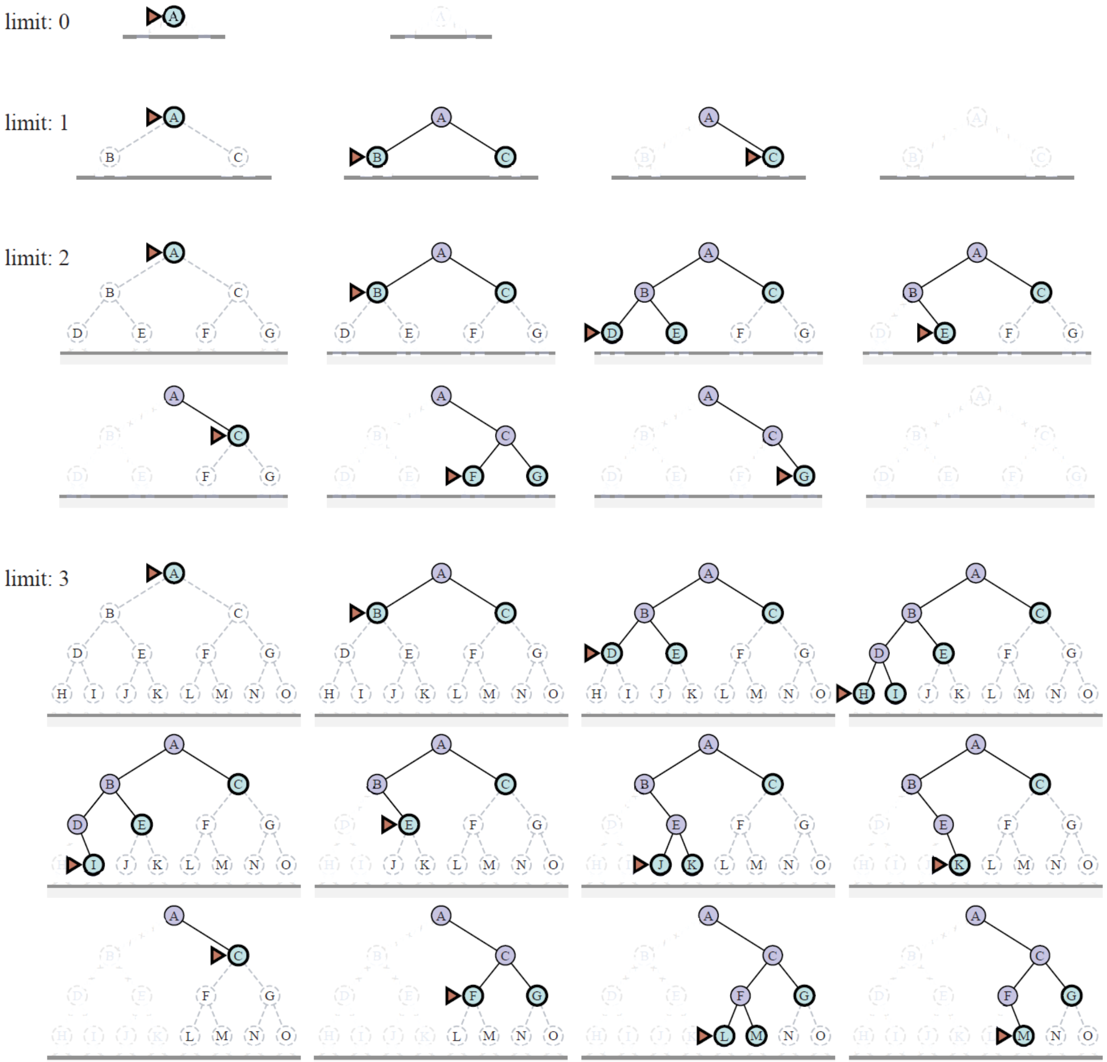
Depth-First Search

- $f(n) = -$ (depth of node n)
- Complete if state space is
  - Tree or DAG
  - Else incomplete
- Not optimal
- Smaller memory requirement
  - O(bm), m is the max depth
- Time complexity is $O(b^m)$



Improvements

- Depth-limited search
  - Set the maximum depth limit and do DFS
  - E.g., set depth = 19 for the Romania map navigation problem
  - Neither complete nor optimal
- Iterative deepening search
  - Set the depth limit as 0, 1, 2, 3, … and do depth-limited search
    - Most nodes are at the bottom level
  - Combines BFS and DFS
  - Memory requirements of DFS O(bd), is complete, but not optimal in general
    - optimal if action costs are all the same


limit: 0
limit: 1
limit: 2
limit: 3

Bidirectional Search

- Simultaneous search from the initial and gaol states
    - Why?
    - $b^{d/2} + b^{d/2}$ vs b^d
- Can use BFS or some other search algorithm
- Keep track of two sets of frontiers and two sets of reached states
    - Opposite parent-child relationships
    - Solution when the two frontiers meet
    - If BFS: $O(b^{d/2})$ time and space complexity

Bidirectional Search

function BIBF-SEARCH($problem_F$, f_F, $problem_B$, f_B) **returns** a solution node, or *failure*
    $node_F \leftarrow$ NODE($problem_F$.INITIAL)   // *Node for a start state*
    $node_B \leftarrow$ NODE($problem_B$.INITIAL)   // *Node for a goal state*
    $frontier_F \leftarrow$ a priority queue ordered by f_F, with $node_F$ as an element
    $frontier_B \leftarrow$ a priority queue ordered by f_B, with $node_B$ as an element
    $reached_F \leftarrow$ a lookup table, with one key $node_F$.STATE and value $node_F$
    $reached_B \leftarrow$ a lookup table, with one key $node_B$.STATE and value $node_B$
    $solution \leftarrow failure$
    while not TERMINATED($solution$, $frontier_F$, $frontier_B$) **do**
        if f_F(TOP($frontier_F$)) $<$ f_B(TOP($frontier_B$)) **then**
            $solution \leftarrow$ PROCEED(F, $problem_F$ $frontier_F$, $reached_F$, $reached_B$, $solution$)
        else $solution \leftarrow$ PROCEED(B, $problem_B$, $frontier_B$, $reached_B$, $reached_F$, $solution$)
    return $solution$

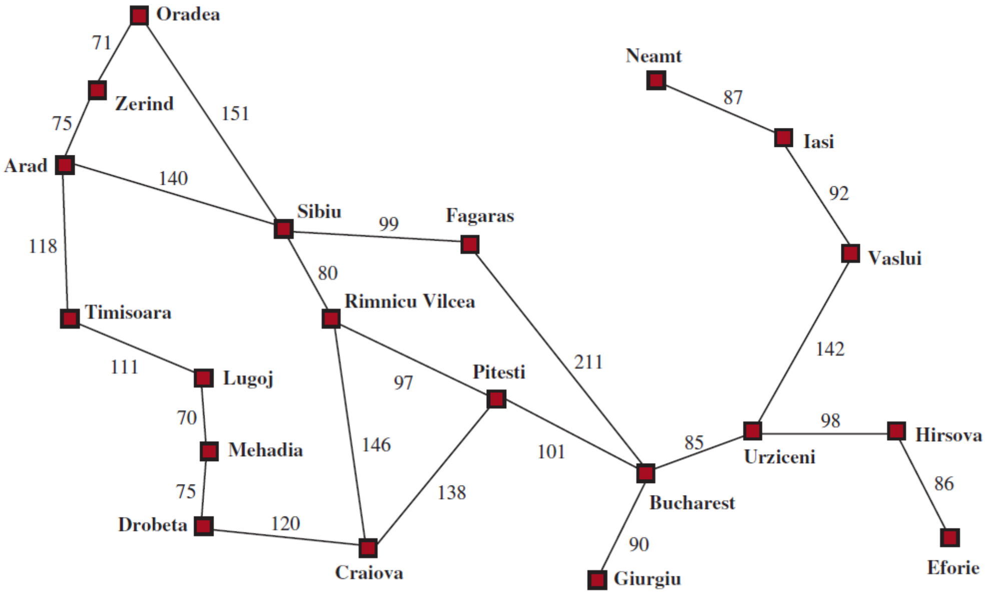
Informed Search

- Search process uses domain specific hints about goals
- Hints are given by heuristic function h(n) where
  - h(n) = estimated cost of the cheapest path from n to goal
- Study of informed search = study of heuristic functions

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

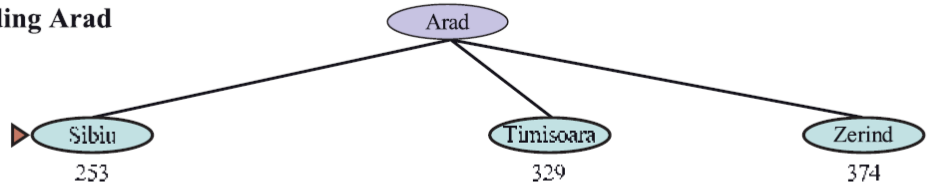
Greedy Best First Search

- Expand the node with the smallest h(n) first
- O(|V|) time and space complexity

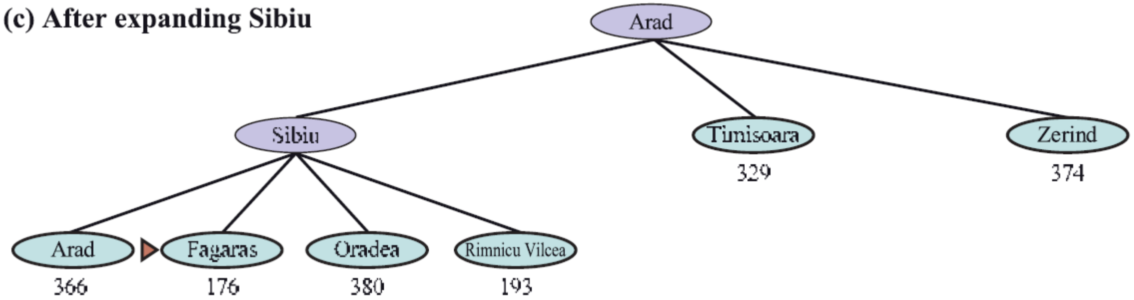


Arad	366	**Mehadia**	241
Bucharest	0	**Neamt**	234
Craiova	160	**Oradea**	380
Drobeta	242	**Pitesti**	100
Eforie	161	**Rimnicu Vilcea**	193
Fagaras	176	**Sibiu**	253
Giurgiu	77	**Timisoara**	329
Hirsova	151	**Urziceni**	80
Iasi	226	**Vaslui**	199
Lugoj	244	**Zerind**	374

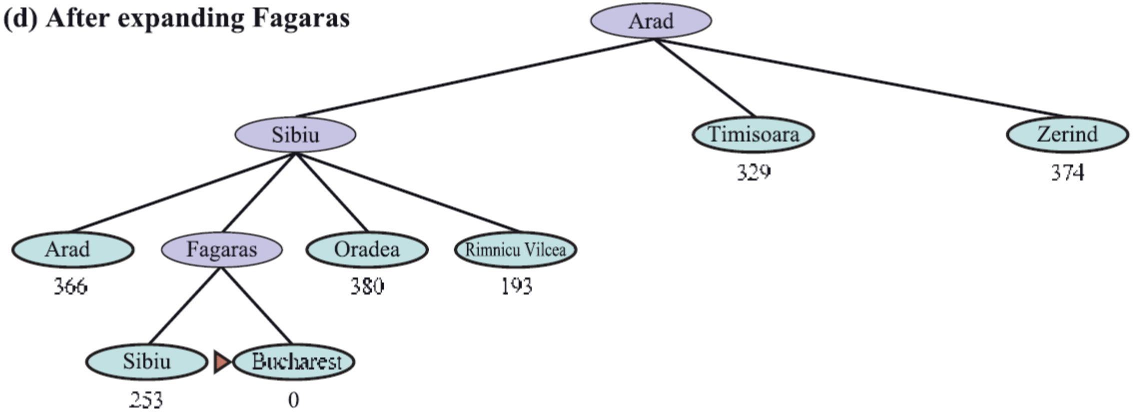
(b) After expanding Arad



(c) After expanding Sibiu



(d) After expanding Fagaras



A* Search

- Numerous applications
- $h(n)$ = estimated cost of the cheapest path from n to goal
- $g(n)$ = actual cost from start state to n
- A* uses $f(n) = g(n) + h(n)$ as the estimated cost from start to goal via n

A* Search

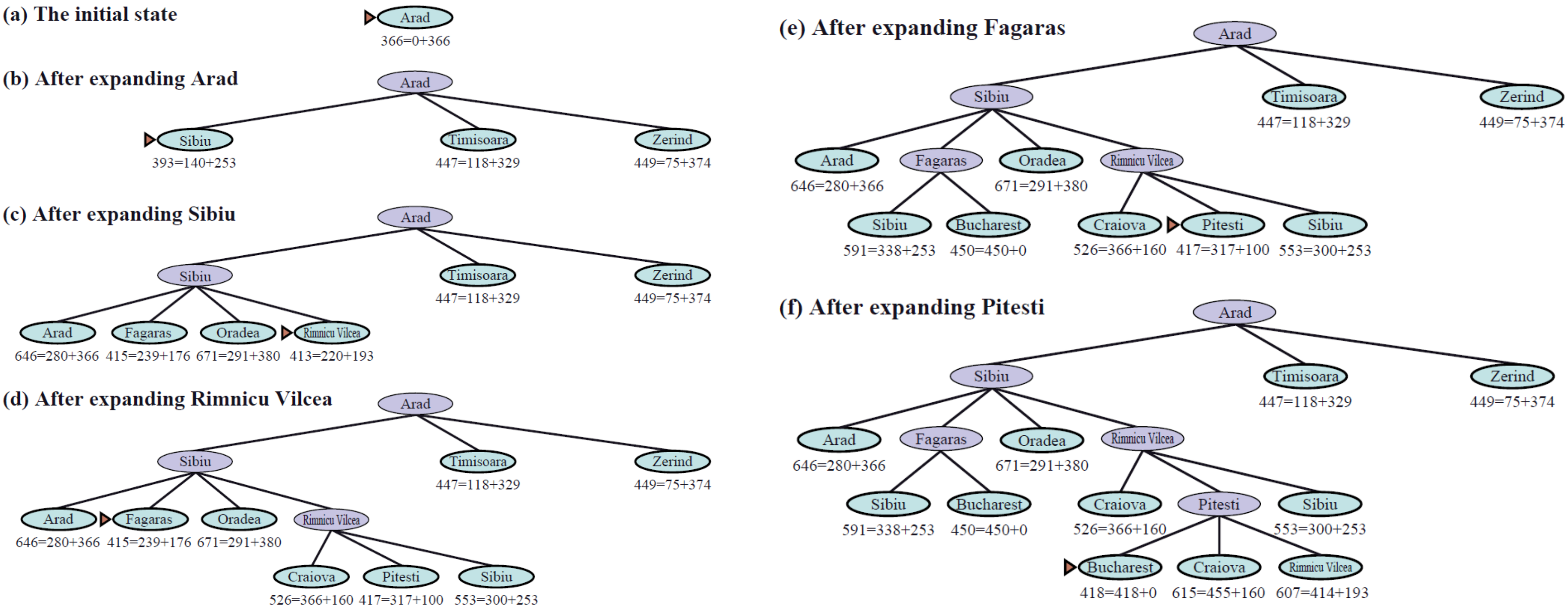
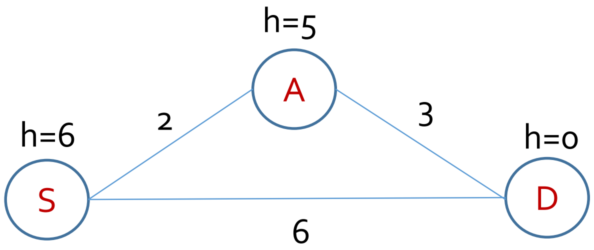


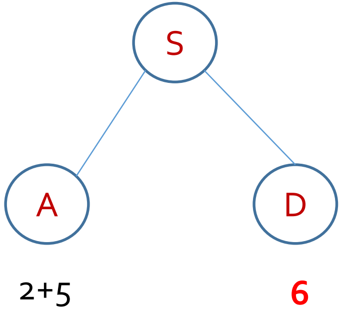
Figure 3.18 Stages in an A* search for Bucharest. Nodes are labeled with $f = g + h$. The h values are the straight-line distances to Bucharest taken from Figure 3.16.

A* Search

- Heuristic estimates must be optimistic or realistic
  - Estimates ≤ Actual costs
- A heuristic is called admissible if it never overestimates the cost to a goal
  - $0 \leq h(n) \leq h^*(n)$, $h^*(n)$: actual cost

- A counterexample





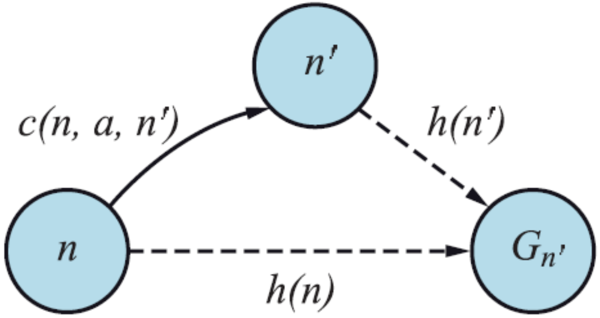
A* Search Properties

- Complete
- Optimal if heuristic is admissible
- Proof by contradiction
  - The cost of A* solution C > C* where C* is the optimal cost.
  - Let n be a node which is on the path to optimal solution but not in A* solution. Therefore, f(n) ≥ C > C* which can't be true.

$$
\begin{aligned}
f(n) &> C^* \quad \text{(otherwise } n \text{ would have been expanded)} \\
f(n) &= g(n) + h(n) \quad \text{(by definition)} \\
f(n) &= g^*(n) + h(n) \quad \text{(because } n \text{ is on an optimal path)} \\
f(n) &\leq g^*(n) + h^*(n) \quad \text{(because of admissibility, } h(n) \leq h^*(n)) \\
f(n) &\leq C^* \quad \text{(by definition, } C^* = g^*(n) + h^*(n))
\end{aligned}
$$

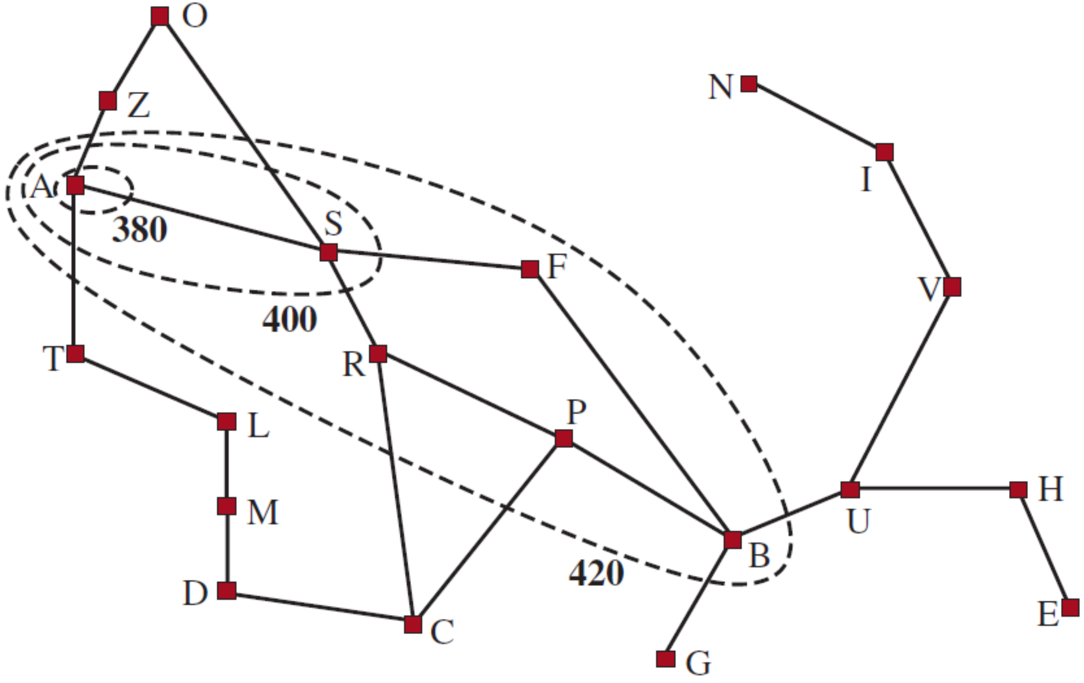
Consistent heuristic

- A heuristic is consistent if it obeys triangle inequality
  - h(n) ≤ c(n, a, n') + h(n')
  - Going via n' should not reduce the cost
- Every consistent heuristic is admissible but not vice-versa
  - Stronger condition than consistency
- With a consistent heuristic, the first time we reach a state, it will be on an optimal path
  - If C* is the optimal cost, A* won't expand any node with f(n) > C*



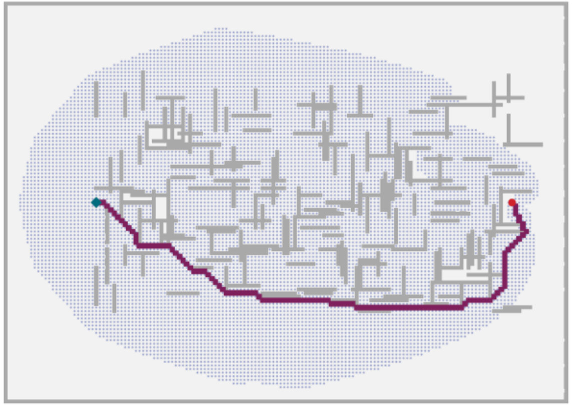
A* Search Contours

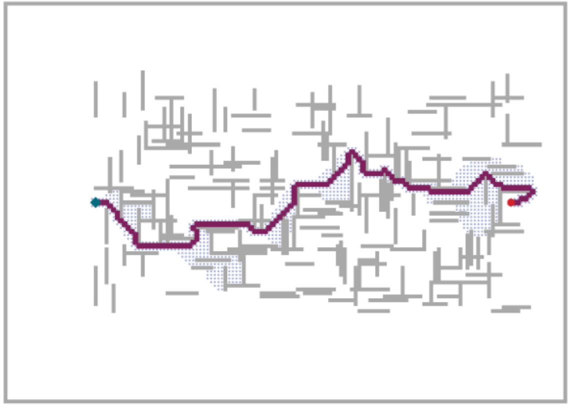
- A* expands lowest f-cost node at the frontier
  - Contours have bias towards the goal



Weighted A* - Satisficing Search

- A* expands too may nodes
- Satisficing: accept suboptimal but "good enough" solutions
- Detour index: multiplier to straight line distance to account for road curvatures
- Weighted A* search: $f(n) = g(n) + W * h(n), W > 1$
  - "Somewhat greedy best-first search"





Weighted A* Search

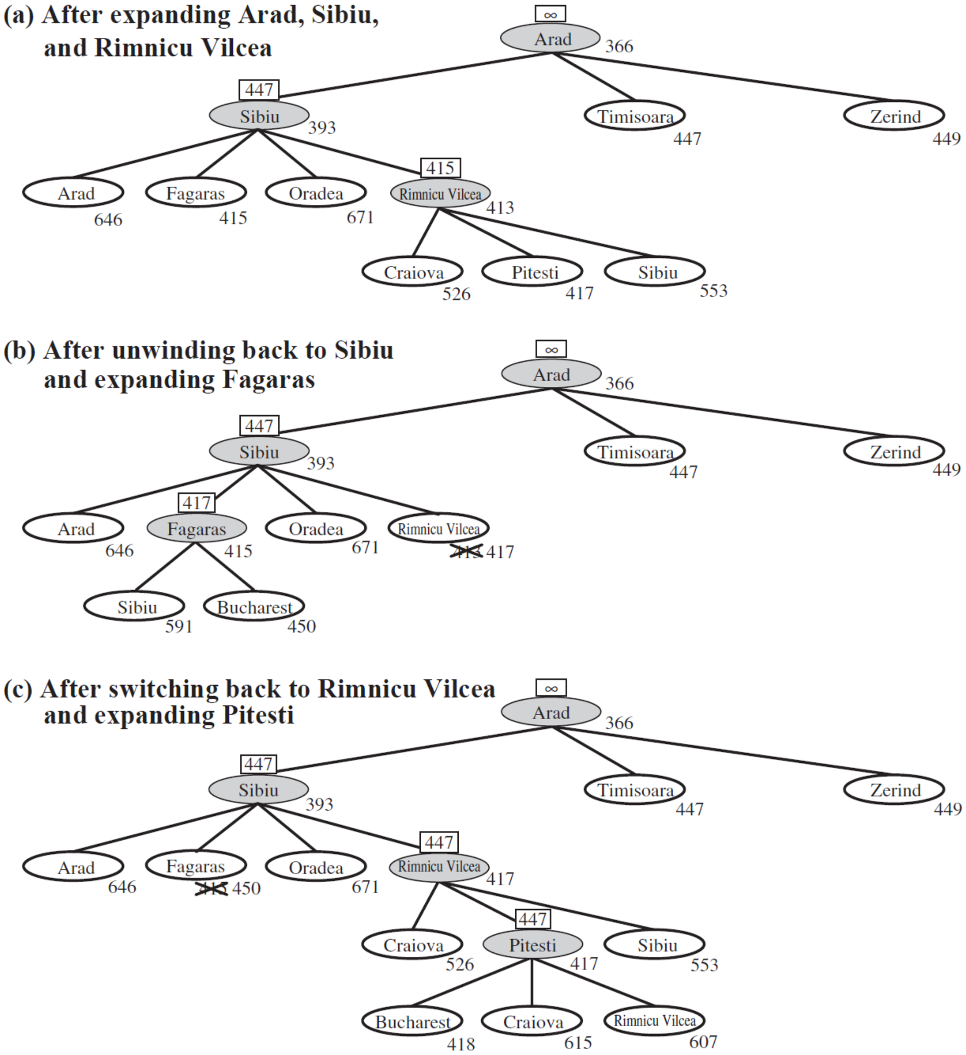
A* search:	$g(n) + h(n)$	$(W = 1)$
Uniform-cost search:	$g(n)$	$(W = 0)$
Greedy best-first search:	$h(n)$	$(W = \infty)$
Weighted A* search:	$g(n) + W \times h(n)$	$(1 < W < \infty)$

Improvements to A* Search

- A* is memory hungry
- Iterative deepening A* search (IDA*)
  - Cutoff is f-cost (g+h) instead of depth
  - Increase the cutoff by the smallest f-cost of the node beyond the search contour
    - No of iterations is bounded by C* if f-cost is an integer
- Recursive best-first search (RBFS)
  - f-limit keeps track of the f-value of the best alternative path from any ancestor of the current node
    - If the recursion exceeds this limit, the search unwinds

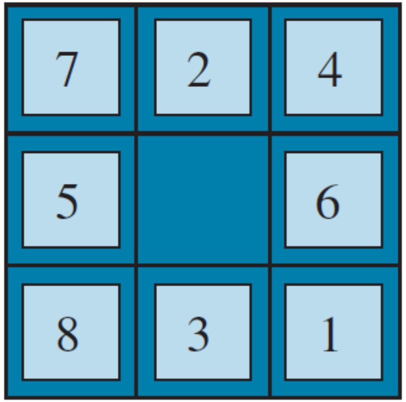
RBFS

- Frequent switches
  - Increases near the goal

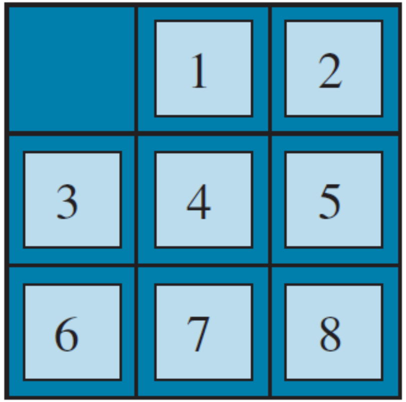


Creating admissible heuristic

- Much of the hard work
- Solve a relaxed version of the problem, use pattern databases, use precomputed landmark solutions, learn (what to look for)
- Example: 8-Puzzle
  - 9!/2 = 181,400 reachable states
  - Good heuristics?



Start State

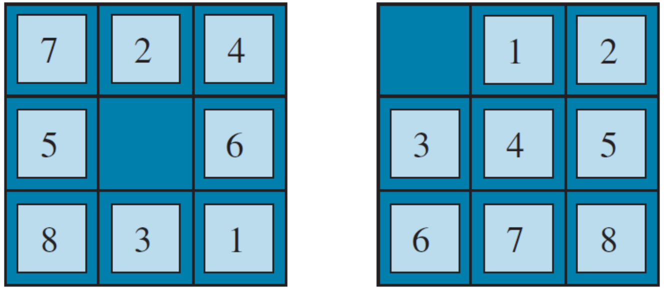


Goal State

Creating good heuristic

- Heuristic choices for 8-Puzzle
  - No. of misplaced tiles (h_1)
  - Sum of Manhattan distances to the correct position (h_2)
  - Here h_2 dominates h_1, i.e., $h_2 \geq h_1$
    - A* with h_1 will expand all the nodes that A* with h_2 does and possibly some more

- The effect of using a heuristic in A* search is a reduced effective depth of the search compared to that of the uniform search (Korf & Reid, 1998)
  - $O(b^{d-k})$ vs $O(b^d)$



Start State Goal State

$h_1 = 8$, $h_2 = 18$

Dominating heuristic is more efficient

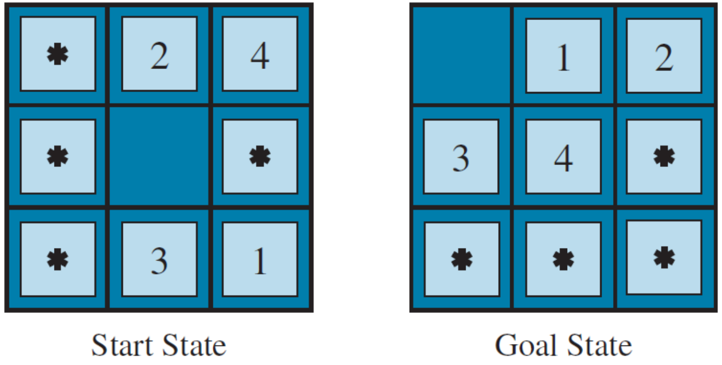
	Search Cost (nodes generated)			Effective Branching Factor		
d	BFS	$A^*(h_1)$	$A^*(h_2)$	BFS	$A^*(h_1)$	$A^*(h_2)$
6	128	24	19	2.01	1.42	1.34
8	368	48	31	1.91	1.40	1.30
10	1033	116	48	1.85	1.43	1.27
12	2672	279	84	1.80	1.45	1.28
14	6783	678	174	1.77	1.47	1.31
16	17270	1683	364	1.74	1.48	1.32
18	41558	4102	751	1.72	1.49	1.34
20	91493	9905	1318	1.69	1.50	1.34
22	175921	22955	2548	1.66	1.50	1.34
24	290082	53039	5733	1.62	1.50	1.36
26	395355	110372	10080	1.58	1.50	1.35
28	463234	202565	22055	1.53	1.49	1.36

Generate heuristic from relaxed problems

- The state-space graph of the relaxed problem is a supergraph of the original problem state-space graph
  - Relaxation results in extra edges added to the graph
- The cost of an admissible solution to a relaxed problem becomes less. Hence, the solution of relaxed problem is an admissible heuristic to the original problem
- Heuristic cost needs to be generated fast
- Generating heuristic costs can be automated
  - Absolver (Prieditis, 1993) generated heuristic was better than known ones for 8-Puzzle and could generate for Rubik's cube
- Can combine admissible heuristics: $h(n) = \max(h_1(n), \ldots, h_k(n))$

Generate heuristic from subproblems

- Cost of the optimal solution of a subproblem is a lower bound on the cost of the complete problem
- Store the exact solution cost of every subproblem in a pattern database
  - Example: pattern for 1-2-3-4
  - Can combine the heuristic cost for multiple patterns (take max)
  - More accurate than Manhattan distance
  - Large speedups in practice



- Reading: Chapter 3
- Assignments: PS 2, search.ipynb

- Project: Phase-I report due in 3 weeks

- Next: CSP, Chapter 6

- Quiz 1 coming up