Search

School of EECS
Washington State University

Overview

- Problem-solving agent
- Formulating problems
- Search
- Uninformed search
- Informed (heuristic) search
- Heuristics
- Admissibility

Problem-Solving Agent

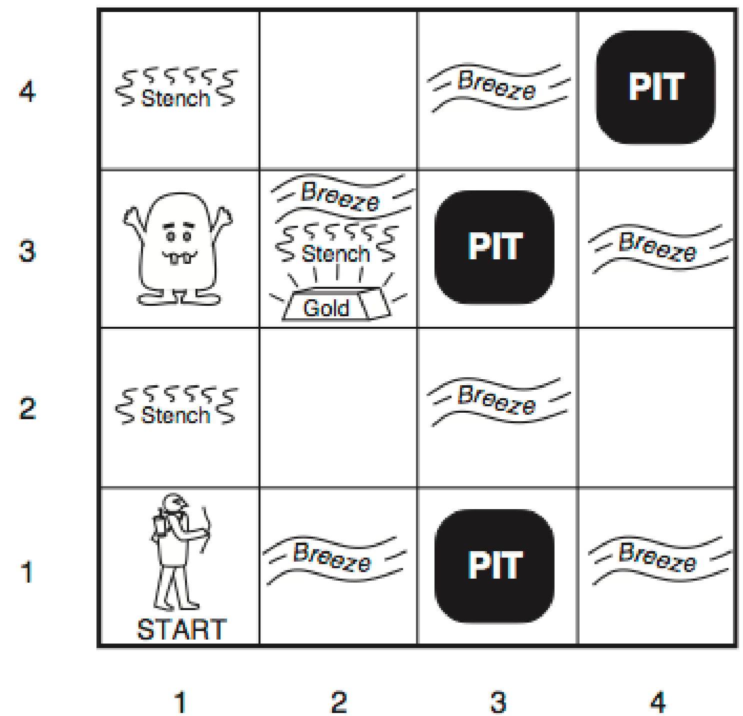
- Goal-based
- Atomic state representation
- Assume solution is a fixed sequence of actions
- Rationality: Achieve goal (minimize cost)
- Search for sequence of actions achieving goal

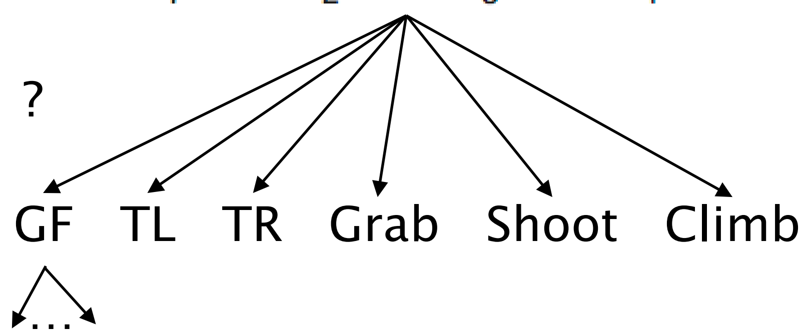
Environment Assumptions

- Observable
  - Agent always knows what state it is in
- Deterministic
  - Each action has one possible outcome (next state)
- Discrete
  - Each state has finite number of applicable actions
- Known
  - Agent knows which state each action will lead to
- Once solution sequence known, execute blindly (ignore percepts) until completion

Wumpus World Example

- Initial state →
- Goal state
  - Any state where agent has gold and not in cave
- Solution?





Problem–Solving Agent

```
function SIMPLE-PROBLEM-SOLVING-AGENT (percept) returns an action
    persistent: seq, an action sequence initially empty
                state, some description of the current world state
                goal, a goal, initially null
                problem, a problem formulation
    state ← UPDATE-STATE (state, percept)
    if seq is empty then
        goal = FORMULATE-GOAL (state)
        problem = FORMULATE-PROBLEM (state, goal)
        seq = SEARCH (problem)
        if seq = failure then return a null action
    action = FIRST (seq)
    seq = REST (seq)
    return action
```

Well-Defined Problems

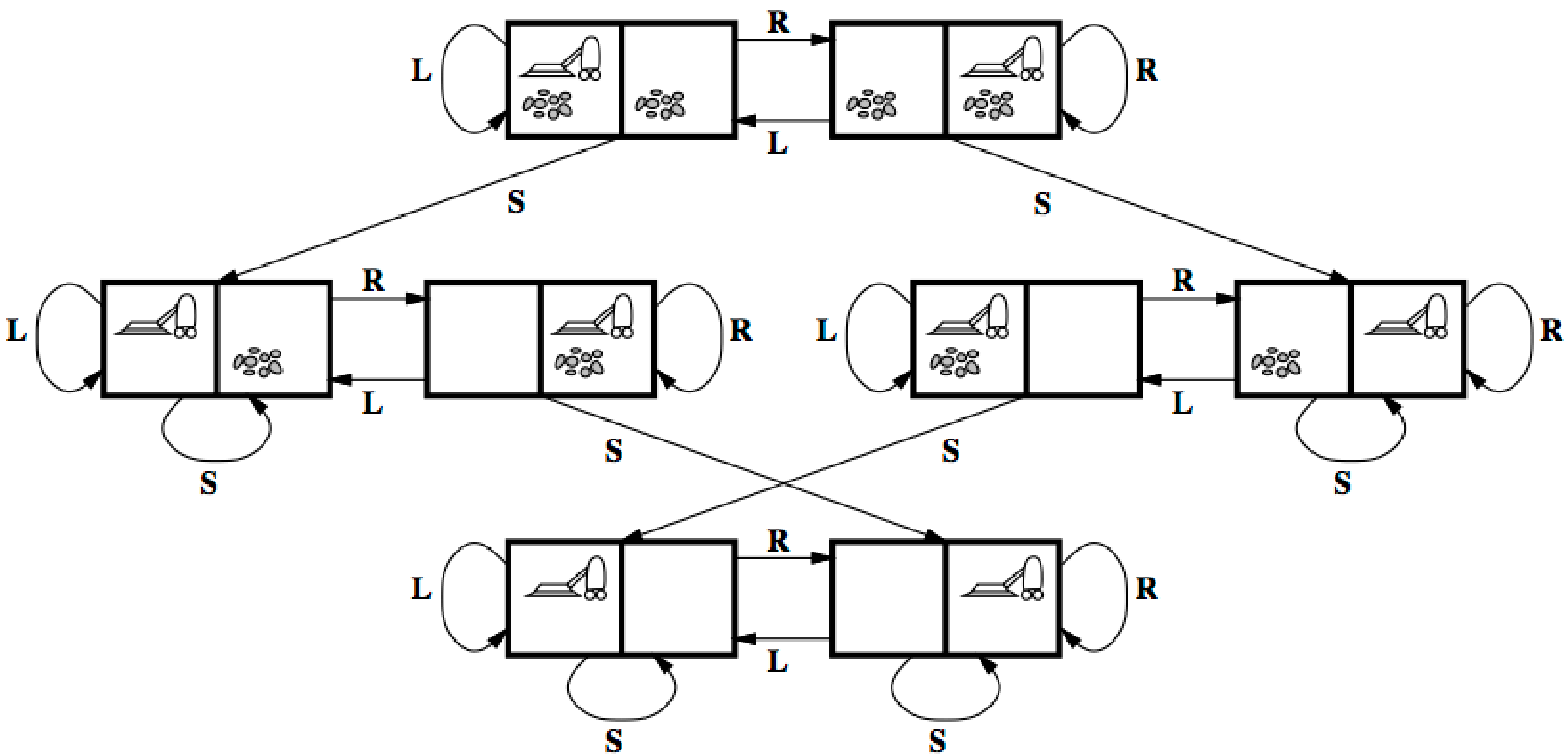
- State representation (atomic, but…)
- Action definitions (action: state → state)
- Five parts
  - Initial state
  - Actions
  - Transition model
  - Goal test
  - Path cost

Well-Defined Problems (5 parts)

1. **Initial state**
2. **Actions**
   - ACTIONS(*s*) returns set of actions applicable to state *s*
3. **Transition model**
   - RESULT(*s*,*a*) returns state resulting from taking action *a* in state *s*
   - Successor state is any state reachable from the current state by a single action

- State space is set of all states reachable from the initial state by any sequence of actions
- State space forms a directed graph of nodes (states) and edges (actions)
- Path in state space is a sequence of states connected by actions

Vacuum World State Space



Well-Defined Problems (5 parts)

4. **Goal test**
   - True for any state satisfying goal
5. **Path cost**
   - Sum of the costs of the individual actions along the path
   - Step cost *c(s,a,s')* is the cost of taking action *a* in state *s* to reach state *s'*
   - Non-negative

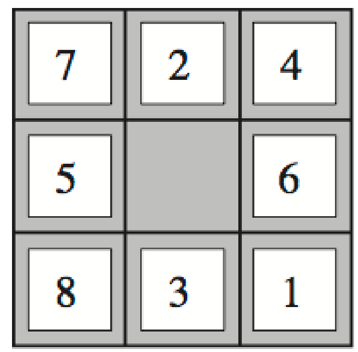
- Solution is sequence of actions leading from the initial state to a goal state
- Optimal solution is a solution with minimal path cost

Vacuum World Problem

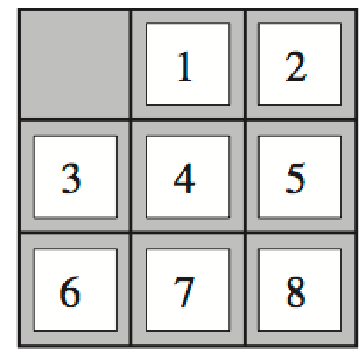
- State representation
  - Location of vacuum: Left, Right
  - Cleanliness of each room: Clean, Dirty
  - Example state: (Left,Clean,Clean)
  - How many unique states?
- Initial state: Any state
- Actions: Left, Right, Suck
- Transition model
  - E.g., Result((Left,Dirty,Clean), Suck) = (Left,Clean,Clean)
- Goal test: State = (?,Clean,Clean)
- Path cost
  - Number of actions in solution (step cost = 1)

8-Puzzle

- State: Location of each tile (and blank)
  - E.g., (B,1,2,3,4,5,6,7,8)
  - How many states?
- Initial state: Any state
- Actions: Move blank Up, Down, Left or Right
- Transition model
- Goal test: State matches Goal State
- Path cost: Number of steps in path (step cost = 1)



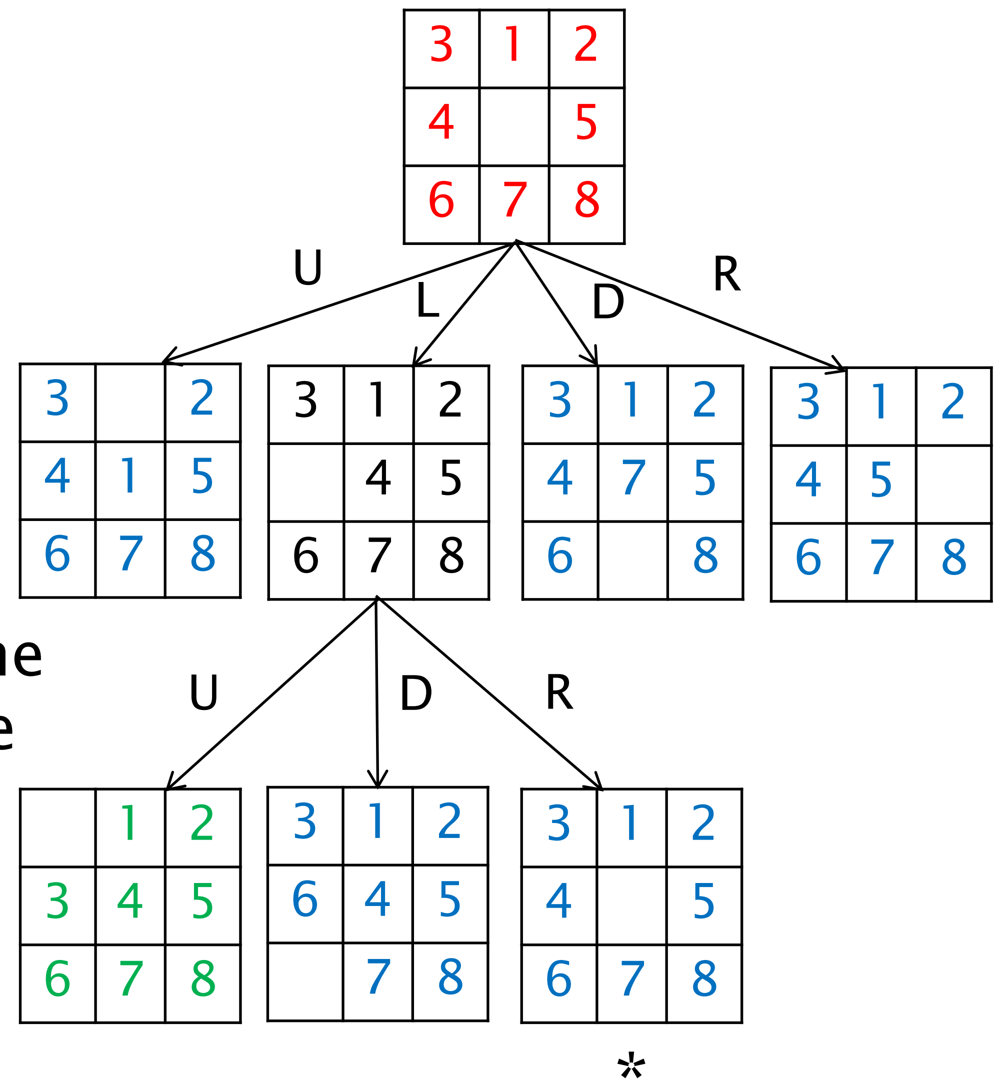
Start State



Goal State

Search

- Search tree
  - Root node is initial state
  - Node branches for each applicable move from node's state
  - Frontier consists of the leaf nodes that can be expanded
  - Repeated states (*)
  - Goal state



Search Demo

- Nice 8-puzzle search web app
  - http://github.com/tristanpenman/n-puzzle

1	2	3
	4	5
7	8	6

Initial State

1	2	3
4	5	6
7	8	

Goal State

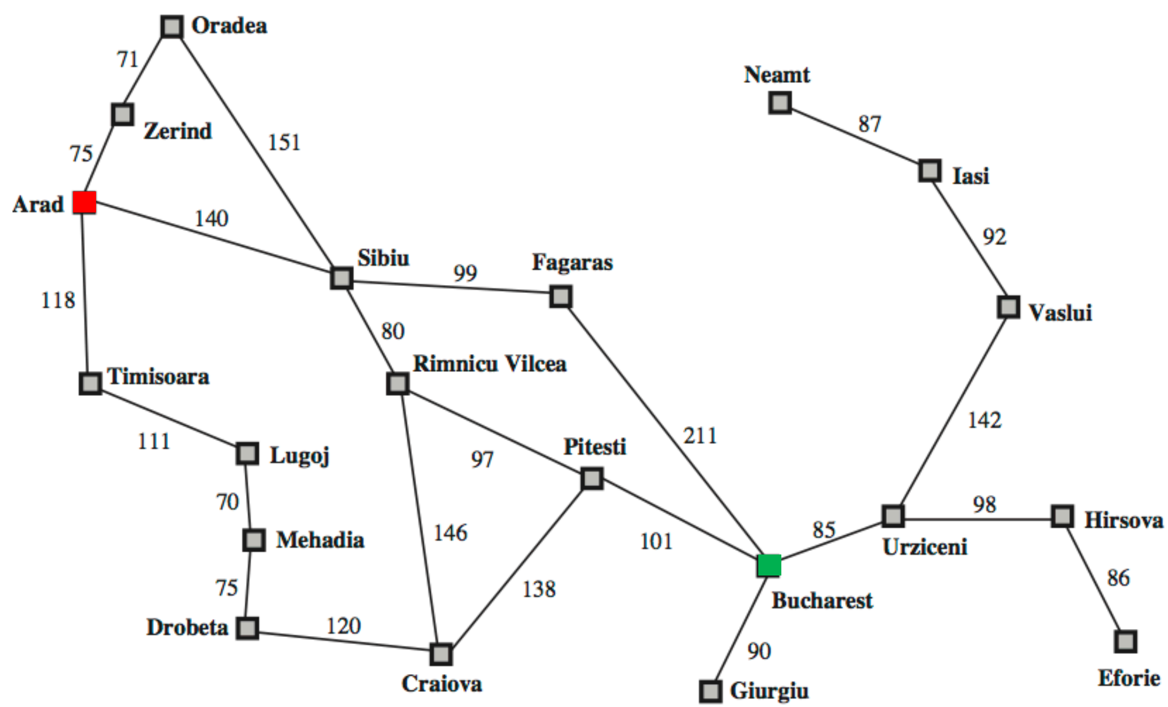
Real-World Search Problems

- Route finding
- Touring
- Layouts
- Robot navigation
- Assembly sequencing
- Chemical design

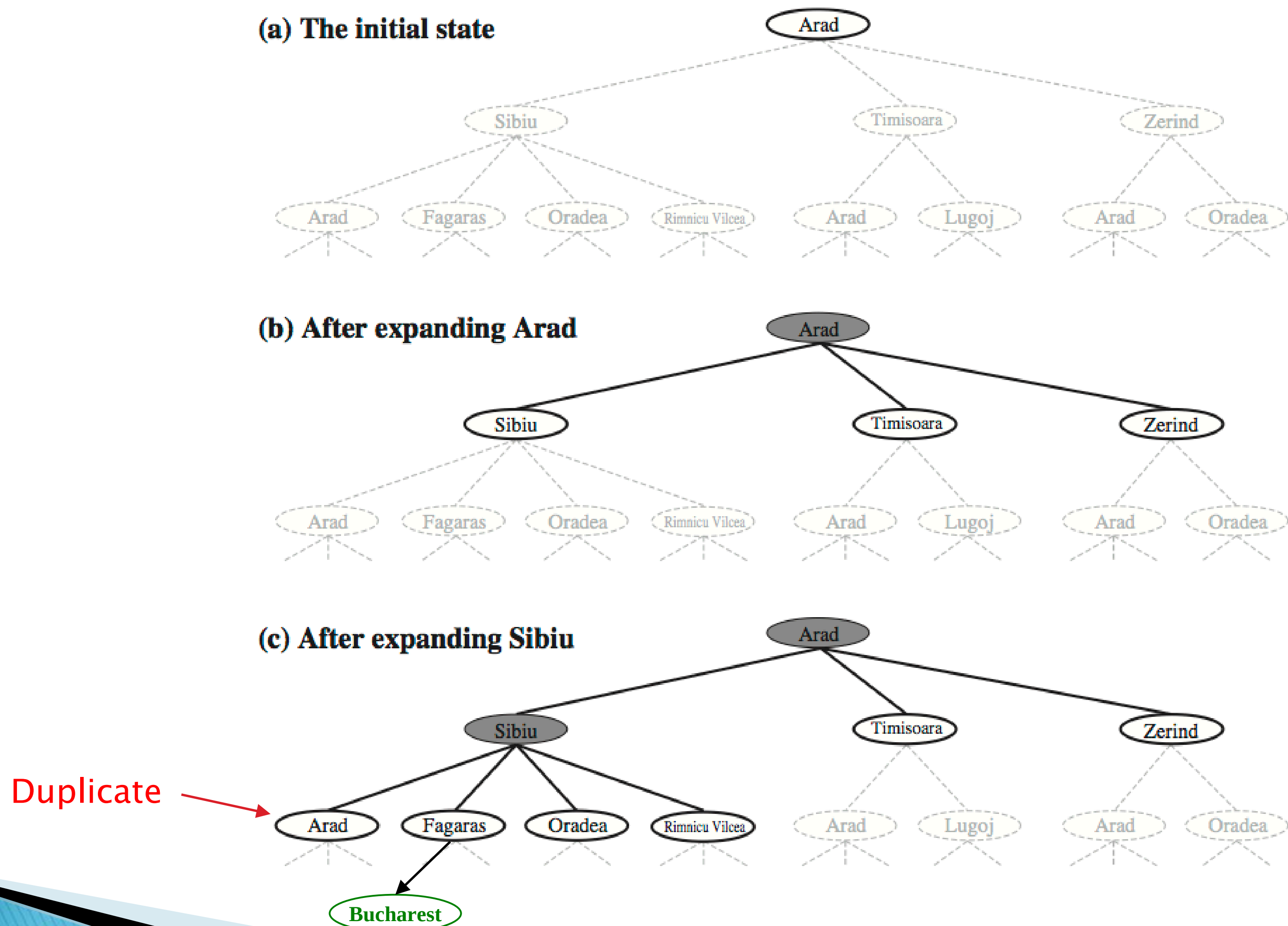
- Most of AI can be cast as a search problem

Route Finding Example

- Romania road map
- Initial state: Arad
- Goal state: Bucharest



Route Finding Example Search Tree



Tree Search

```
function TREE-SEARCH (problem) returns a solution, or failure
  initialize the frontier using the initial state of problem
  loop do
    if the frontier is empty then return failure
    choose a leaf node and remove it from the frontier
    if the node contains a goal state then return the corresponding solution
    expand the node, adding the resulting nodes to the frontier
```

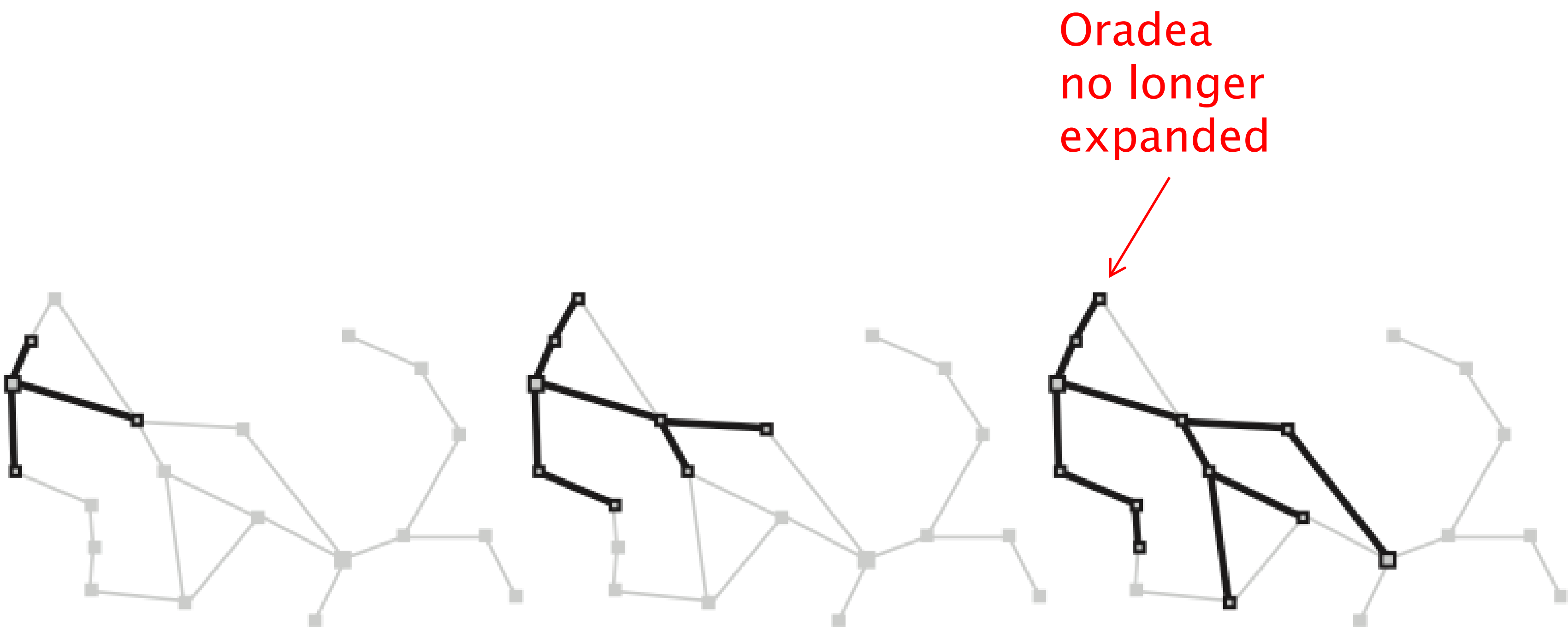
- Search strategy determines how nodes are chosen for expansion
- Suffers from repeated state generation

Graph Search

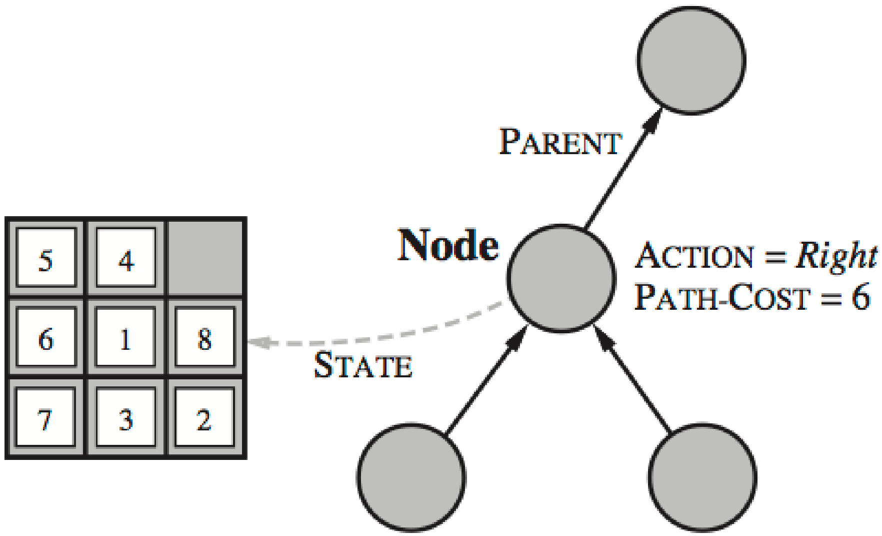
function GRAPH-SEARCH (*problem*) **returns** a solution, or failure
  initialize the frontier using the initial state of *problem*
  initialize the explored set to be empty
  loop do
    if the frontier is empty **then return** failure
    choose a leaf node and remove it from the frontier
    if the node contains a goal state **then return** the corresponding solution
    add the node to the explored set
    expand the node, adding the resulting nodes to the frontier
      only if not in the frontier or explored set

- Keep track of explored set to avoid repeated states
- Changes from TREE-SEARCH highlighted

Graph Search Example



Implementation



function CHILD-NODE (*problem*, *parent*, *action*) **returns** a node
  return a node with
    STATE = *problem*.RESULT (*parent*.STATE, *action*),
    PARENT = *parent*,
    ACTION = *action*,
    PATH-COST = *parent*.PATH-COST +
                problem.STEP-COST (*parent*.STATE, *action*)

Implementation

- Frontier is a queue or stack
  - How nodes are added/removed defines search strategy
- Explored set is a hash table
  - Can be large (# unique states)
  - Key is some canonical state representation

Measuring Performance

- Completeness
  - Is the search algorithm guaranteed to find a solution if one exists?
- Optimality
  - Does the search algorithm find the optimal solution?
- Time and space complexity
  - Branching factor ***b*** (maximum successors of a node)
  - Depth ***d*** of shallowest goal node
  - Maximum path length ***m***
  - Complexity $O(b^d)$ to $O(b^m)$

Uninformed Search Strategies

- No preference over states based on “closeness” to goal
- Strategies
  - Breadth-first search
  - Uniform-cost search
  - Depth-first search
  - Depth-limited search
  - Iterative deepening search
  - Bidirectional search

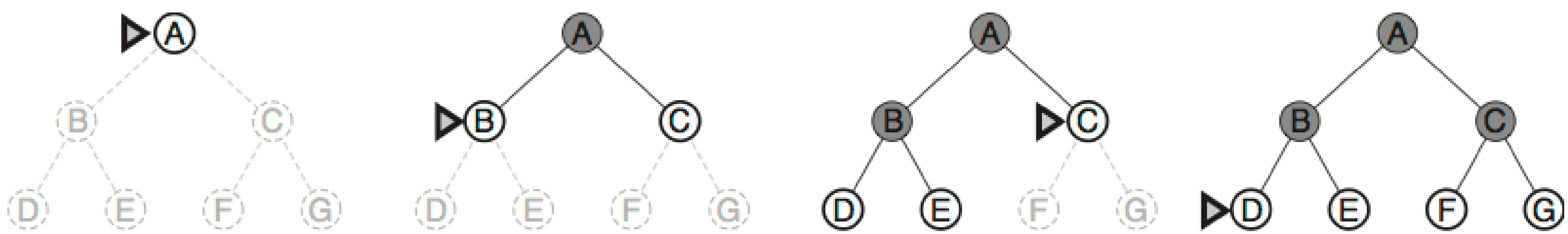
Breadth-First Search

- Expand shallowest nodes in frontier
- Frontier is a simple queue
  - Dequeue nodes from front, enqueue nodes to back
  - First-In, First-Out (FIFO)

Breadth-First Search

```
function BREADTH-FIRST-SEARCH (problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  frontier ← FIFO queue with node as only element
  explored ← empty set
  loop do
    if EMPTY(frontier) then return failure
    node ← DEQUEUE(frontier)   // choose shallowest node in frontier
    add node.STATE to explored
    for each action in problem.ACTIONS(node.STATE) do
      child = CHILD-NODE(problem, node, action)
      if child.STATE is not in explored or frontier then
        if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
        frontier ← ENQUEUE(child, frontier)
```

Breadth-First Search



- 8-puzzle demo

1	2	3
4	8	5
7		6

Initial State

1	2	3
4	5	6
7	8	

Goal State

Breadth-First Search

- Complete?
- Optimal?
- Time complexity
  - Number of nodes generated (worst case)

$$\sum_{i=0}^{d} b^i = O(b^{d+1})$$

- Space complexity
  - $O(b^{d-1})$ nodes in explored set
  - $O(b^d)$ nodes in frontier
  - Total $O(b^d)$

Breadth-First Search

- Exponential complexity $O(b^d)$
- For $b=4$, 1KB/node, 1M nodes/sec

Depth	Nodes	Time	Memory
2	16	0.02 ms	16 KB (10^3)
4	256	0.26 ms	256 KB (10^3)
8	65,536	0.07 sec	65 MB (10^6)
16	4.3B	71.6 min	4.3 TB (10^{12})
20	10^{12}	12.7 days	1 PetaByte (10^{15})
30	10^{18}	366 centuries	1 ZettaByte (10^{21})

Uniform-Cost Search

- Expand node *n* with lowest path cost *g(n)*
- Frontier is a priority queue
  - Queue partially ordered by path cost
  - Lowest path cost node always at the front

Uniform-Cost Search

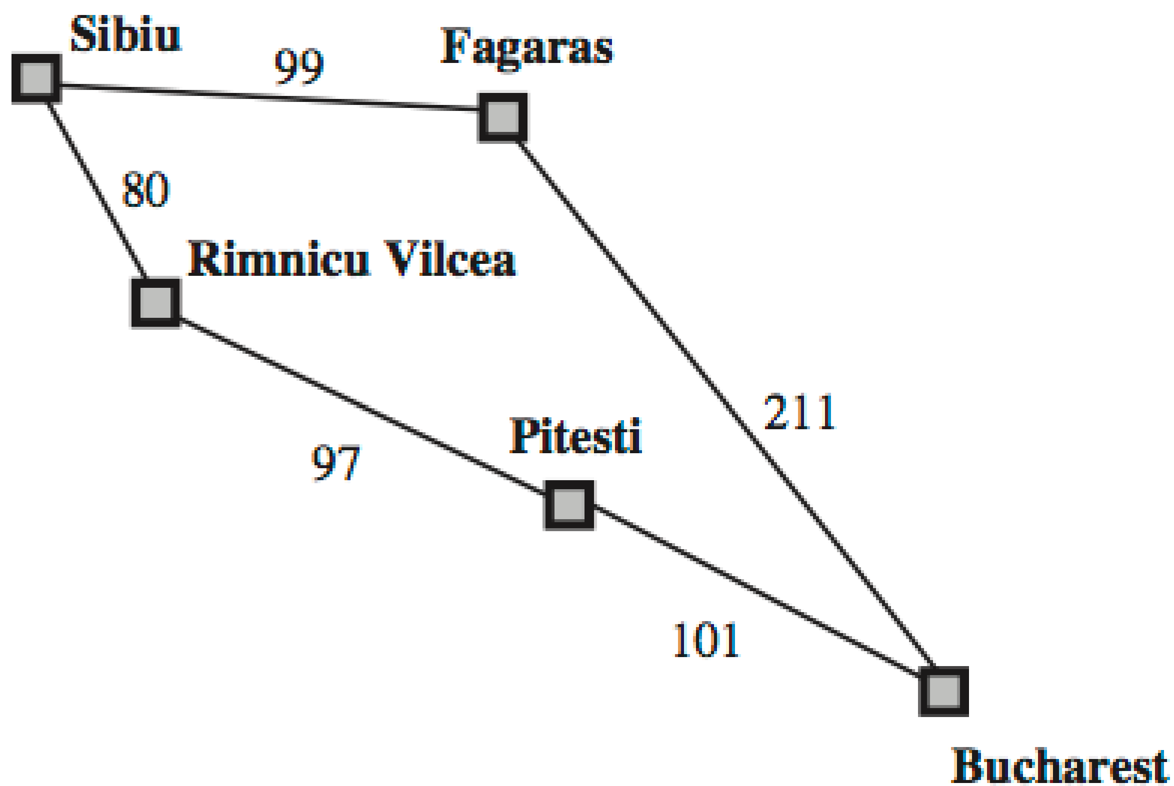
```
function UNIFORM-COST-SEARCH (problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  frontier ← priority queue ordered by PATH-COST, with node as only element
  explored ← empty set
  loop do
    if EMPTY(frontier) then return failure
    node ← DEQUEUE(frontier)   // choose lowest cost node in frontier
    if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
    add node.STATE to explored
    for each action in problem.ACTIONS(node.STATE) do
      child = CHILD-NODE(problem, node, action)
      if child.STATE is not in explored or frontier then
        frontier ← ENQUEUE(child, frontier)
      else if child.STATE is in frontier with higher PATH-COST then
        replace that frontier node with child
```

Why not check Goal-Test here?

Why is this test necessary?

Uniform-Cost Search

- Example (Sibiu → Bucharest)



Uniform-Cost Search

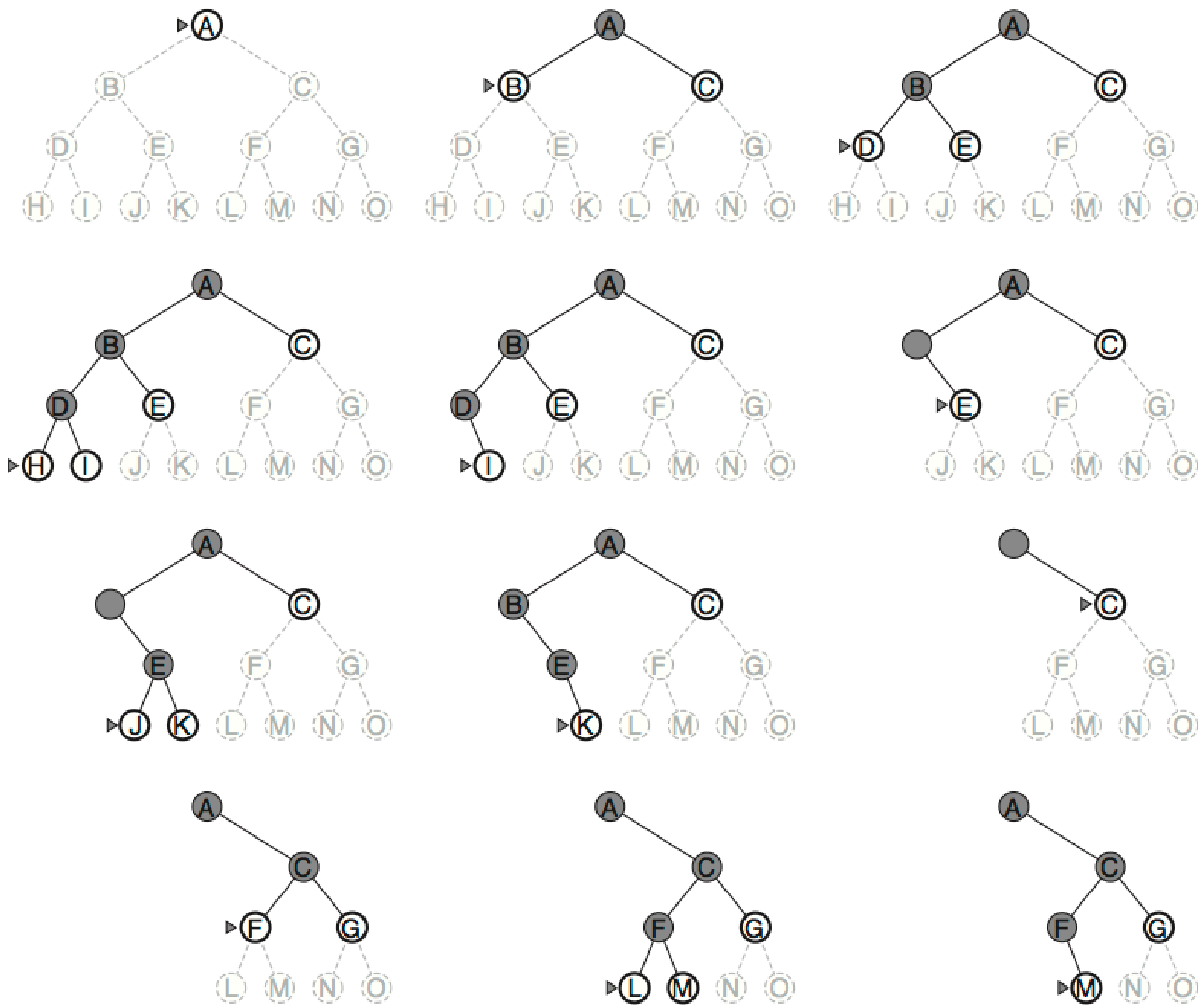
- Complete?
- Optimal?
- Time and space complexity
  - b = branching factor
  - ε = minimum step cost (>0)
  - C* = cost of optimal solution

$$O(b^{1+\lfloor C^*/\varepsilon \rfloor})$$

Depth-First Search

- Always expand the deepest node
- Frontier is a simple queue
  - Enqueue nodes to front, dequeue nodes from front
  - Last-In, First-Out (LIFO)
- Otherwise, same code as BFS
- Or, implement recursively

Depth-First Search



DEMO

Depth-First Search

- Tree-Search version
  - Not complete (infinite loops)
  - Not optimal
- Graph-Search version
  - Complete
  - Not optimal
- Time complexity (m = max depth): $O(b^m)$
- Space complexity
  - Tree-search: $O(bm)$
  - Graph-search: $O(b^m)$

Depth-Limited Search

```
function DEPTH-LIMITED-SEARCH (problem, limit) returns a solution, or failure/cutoff
  return RECURSIVE-DLS (MAKE-NODE (problem.INITIAL-STATE), problem, limit)

function RECURSIVE-DLS (node, problem, limit) returns a solution, or failure/cutoff
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  else if limit = 0 then return cutoff
  else
    cutoff_occurred ← false
    for each action in problem.ACTIONS(node.STATE) do
       child = CHILD-NODE(problem, node, action)
       result ← RECURSIVE-DLS (child, problem, limit – 1)
       if result = cutoff  then cutoff_occurred ← true
       else if result ≠ failure then return result
    if cutoff_occurred then return cutoff else return failure
```

Depth-Limited Search

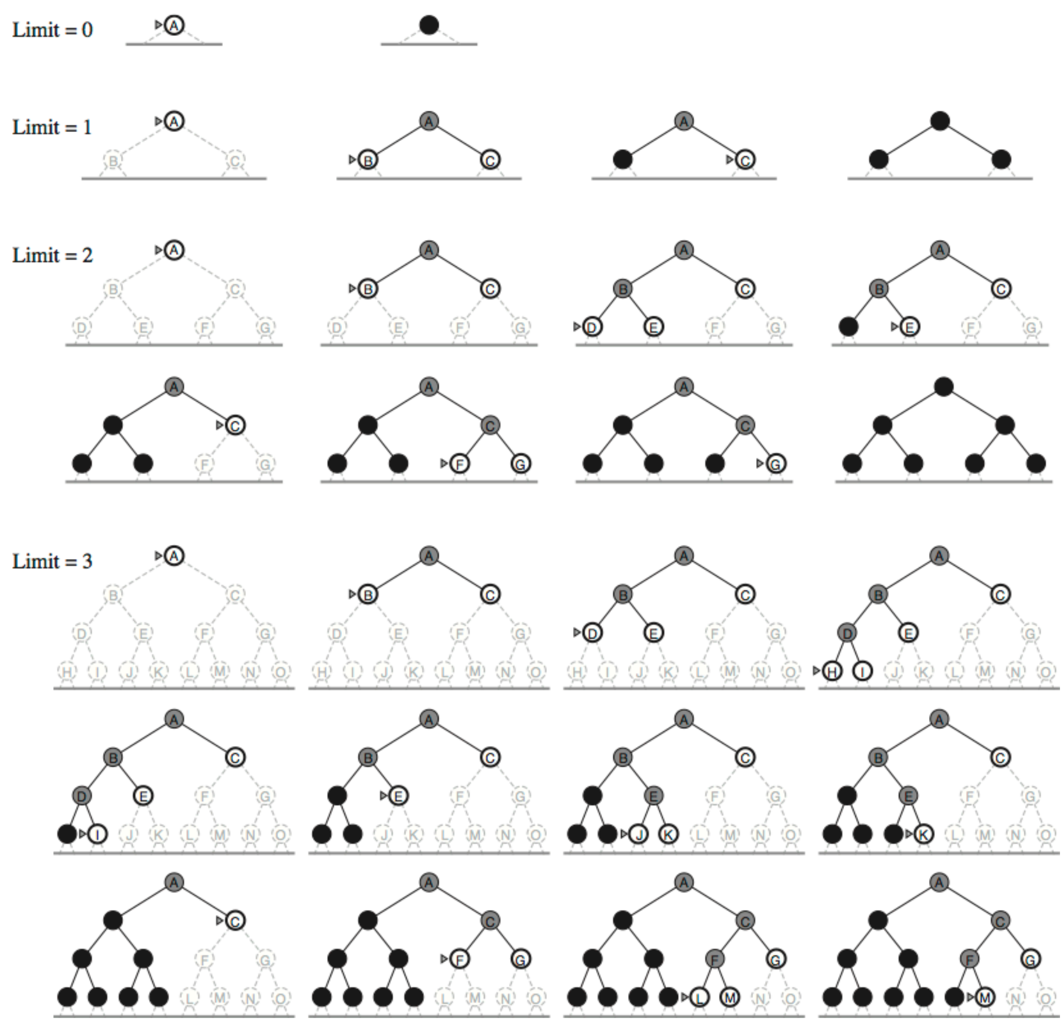
- Limit DFS depth to ℓ
- Still incomplete, if $\ell < d$
- Non-optimal if $\ell > d$
- Time complexity: $O(b^{\ell})$
- Space complexity: $O(b\ell)$

Iterative-Deepening Search

- Run DEPTH-LIMITED-SEARCH iteratively with increasing depth limit

```
function ITERATIVE-DEEPENING-SEARCH (problem) returns a solution, or failure
  for depth = 0 to ∞ do
    result = DEPTH-LIMITED-SEARCH (problem, depth)
    if result ≠ cutoff then return result
```

Iterative-Deepening Search



DEMO

Iterative-Deepening Search

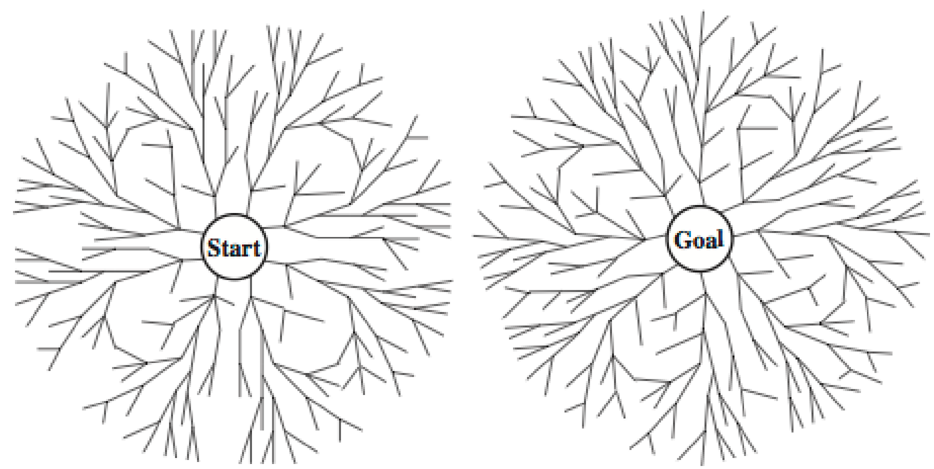
- Complete?
- Optimal?
- Space complexity: *O(bd)*
- Time complexity

$$\sum_{i=0}^{d-1}(d-i)b^{i+1} = (d)b + (d-1)b^2 + \ldots + (1)b^d = O(b^d)$$

  - Nodes at depth d = all nodes at depths 1 to (d-1)
- Iterative deepening best uninformed search when solution depth unknown

Bidirectional Search

- Search forward from initial state and backward from goal state
- Meet (hopefully) in the middle
- Each search has complexity $O(b^{d/2}) \ll O(b^d)$
- Replace goal test with frontier intersection
- How to reverse actions?



Uninformed Search Strategies

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional
Complete	Yes[1]	Yes[1,2]	No	No	Yes[1]	Yes[1,4]
Time	$O(b^d)$	$O(b^{1+\lfloor C^*/\varepsilon \rfloor})$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^d)$	$O(b^{1+\lfloor C^*/\varepsilon \rfloor})$	$O(bm)$	$O(b\ell)$	$O(bd)$	$O(b^{d/2})$
Optimal	Yes[3]	Yes	No	No	Yes[3]	Yes[3,4]

1. Complete if b is finite
2. Complete if step costs $\geq \varepsilon > 0$
3. Optimal if step costs all the same
4. If both directions use BFS

Informed (Heuristic) Search

- Guided by problem-specific knowledge other than the problem formulation
- Problem-specific knowledge usually expressed as heuristics

Heuristic Function

- Heuristic function h(n) estimates cost of the path from state n to a goal state
  - E.g., 8-puzzle
    - Number of tiles out
    - Euclidean distance of each tile
    - City-block (Manhattan) distance of each tile
  - Non-negative function
  - For goal node h(n)=0
- Recall path cost g(n) is the cost so far from the initial state to state n
- Evaluation function f(n) = g(n) + h(n) estimates the total cost of a solution going through state n

1	5	2
4	3	
7	8	6

1	2	3
4	5	6
7	8	

Goal State

Best-First Search

- Choose next frontier node with smallest $f(n)$
- Depth-first search = Best-first search with
  - $f(n) = g(n) + h(n) = ?$
- Breadth-first search = Best-first search with
  - $f(n) = g(n) + h(n) = ?$
- Uniform-cost search = Best-first search with
  - $f(n) = g(n) + h(n) = ?$

Heuristic Search Strategies

- Greedy best-first search
- A* search
- Memory-bounded heuristic search

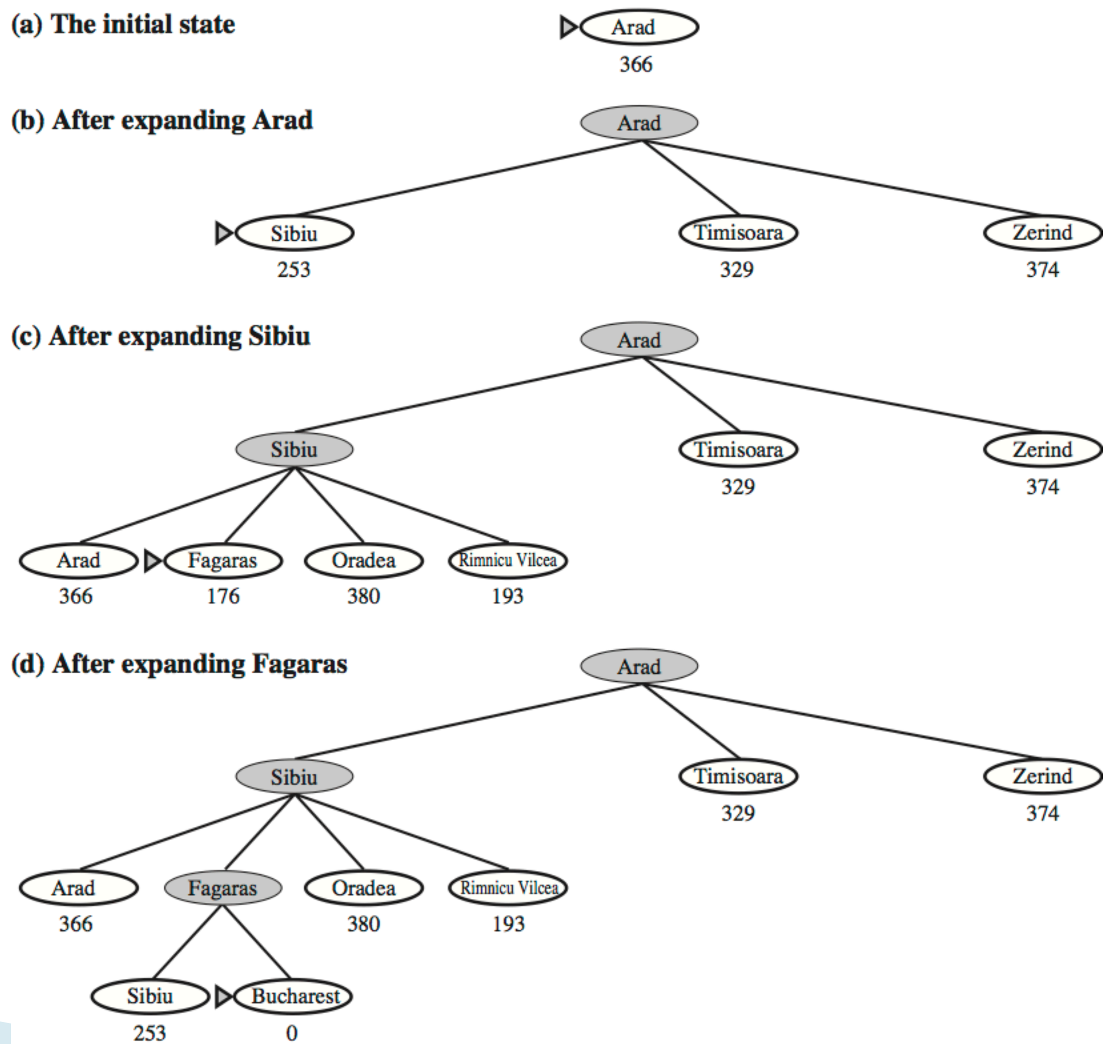
Greedy Best-First Search

- Best-first search with f(n) = h(n)
- Example: Route-finding problem
  - h(n) = straight-line distance from city n to goal city

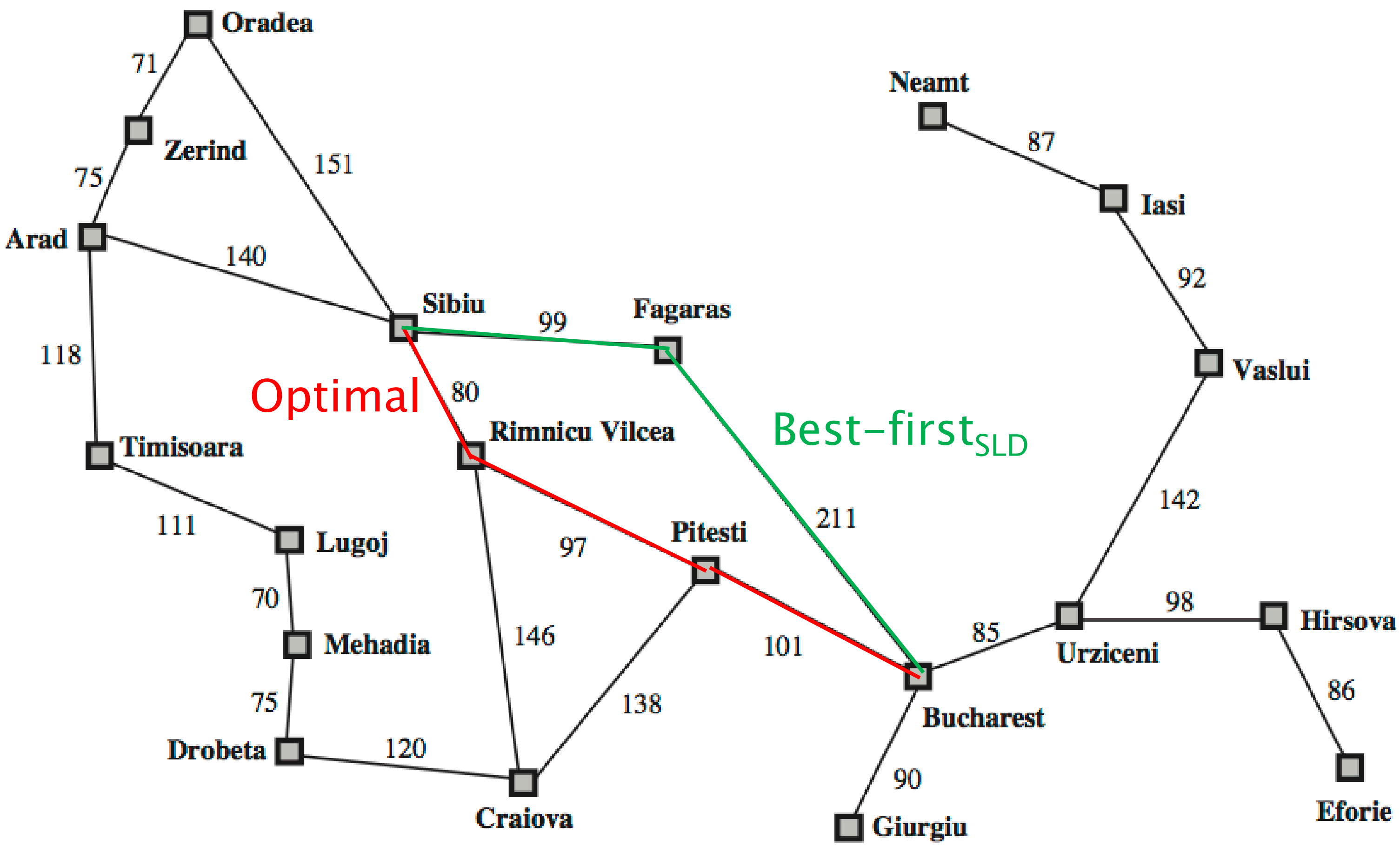
Straight-line distances to Bucharest:

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

Greedy Best-First Search Example: Arad to Bucharest

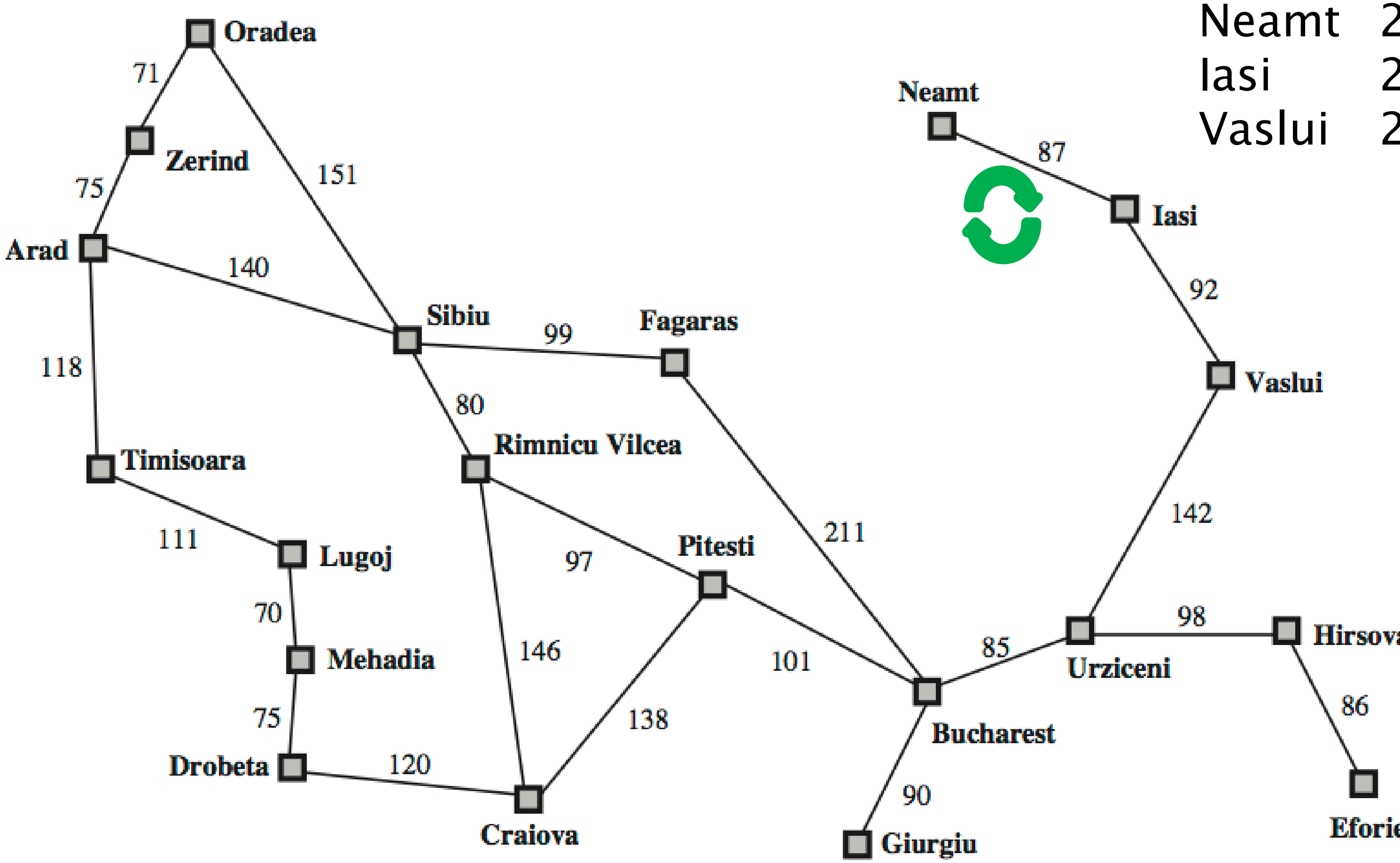


Greedy Best-First Search Example: Arad to Bucharest



Greedy Best-First Tree Search Example: Iasi to Fagaras

SLD to Fagaras	
Neamt	200
Iasi	220
Vaslui	230



Greedy Best-First Search

- Complete?
- Optimal?
- Time and space complexity: $O(b^m)$
  - b = branching factor
  - m = maximum depth of search space
  - Worst case
    - Good heuristic can substantially improve

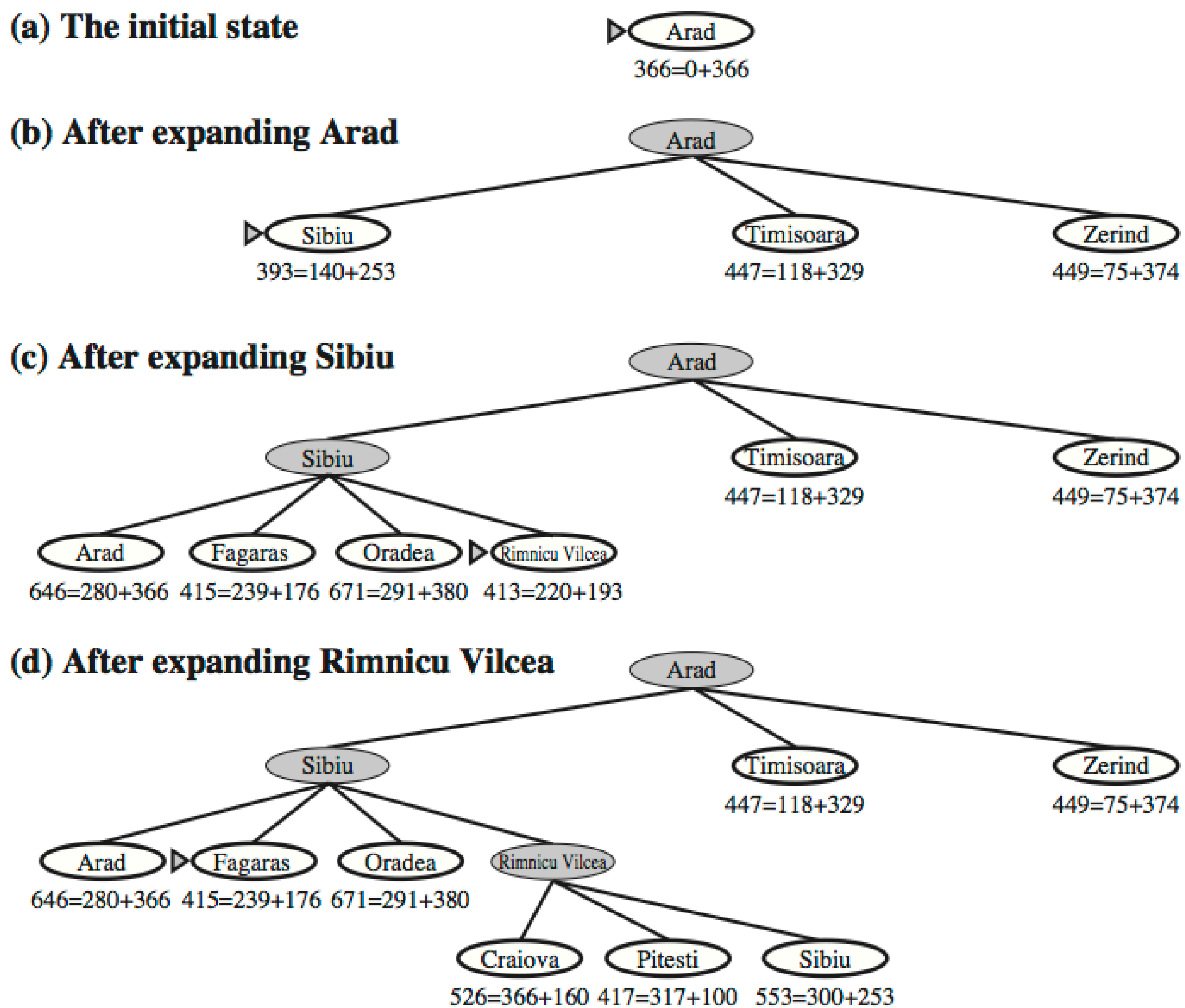
DEMO

A* Search

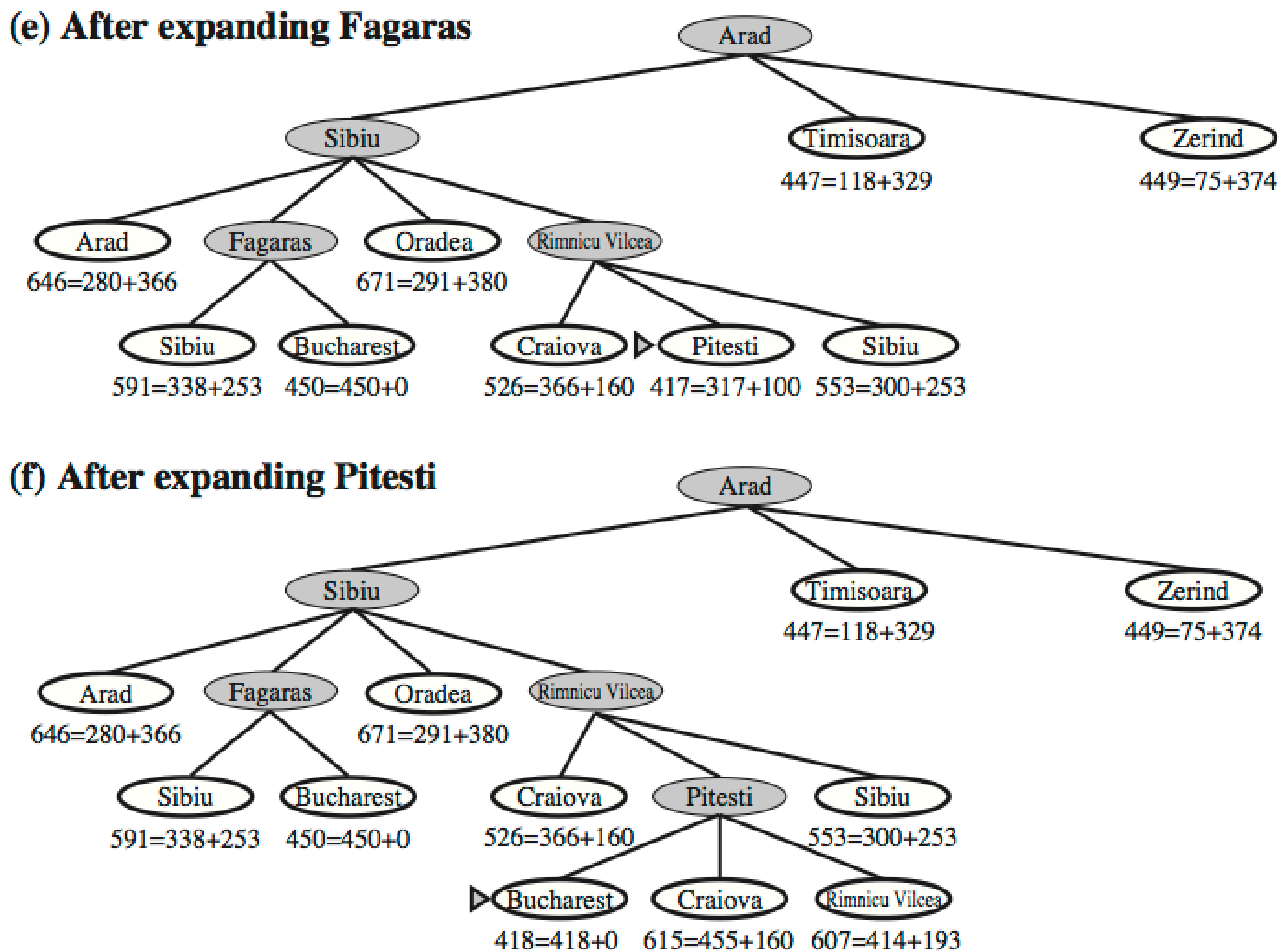
History: A* generalizes over algorithms A1 and A2, which were heuristic extensions to Dijkstra's shortest path algorithm.

- $f(n) = g(n) + h(n)$
  - Estimated cost of solution through n
- Same as Uniform-Cost search using f(n)
- Complete and optimal under some constraints on h(n)
- Example: Route-finding using SLD

A* Search Example: Arad to Bucharest



A* Search Example: Arad to Bucharest (cont.)



Optimality of A*

- For A* tree search to be optimal, h(n) must be admissible
  - A heuristic function h(n) is <u>admissible</u> if it never over-estimates the cost of reaching the goal from n
  - E.g., Straight-line distance for route finding
  - E.g., Tiles out of place in 8-puzzle
- For A* graph search to be optimal, heuristic must further satisfy triangle inequality (also called consistent or monotonic)
  - A heuristic function h(n) satisfies the <u>triangle inequality</u> if $h(n) \leq cost(n,a,n') + h(n')$

A* Search

- Complete and optimal?
  - Yes, if heuristic is admissible
- Time and space complexity?
  - Still $O(b^d)$ worst case
  - Space is typically the bottleneck
- A* is optimally efficient
  - No other algorithm using the same consistent heuristic is guaranteed to expand fewer nodes

DEMO

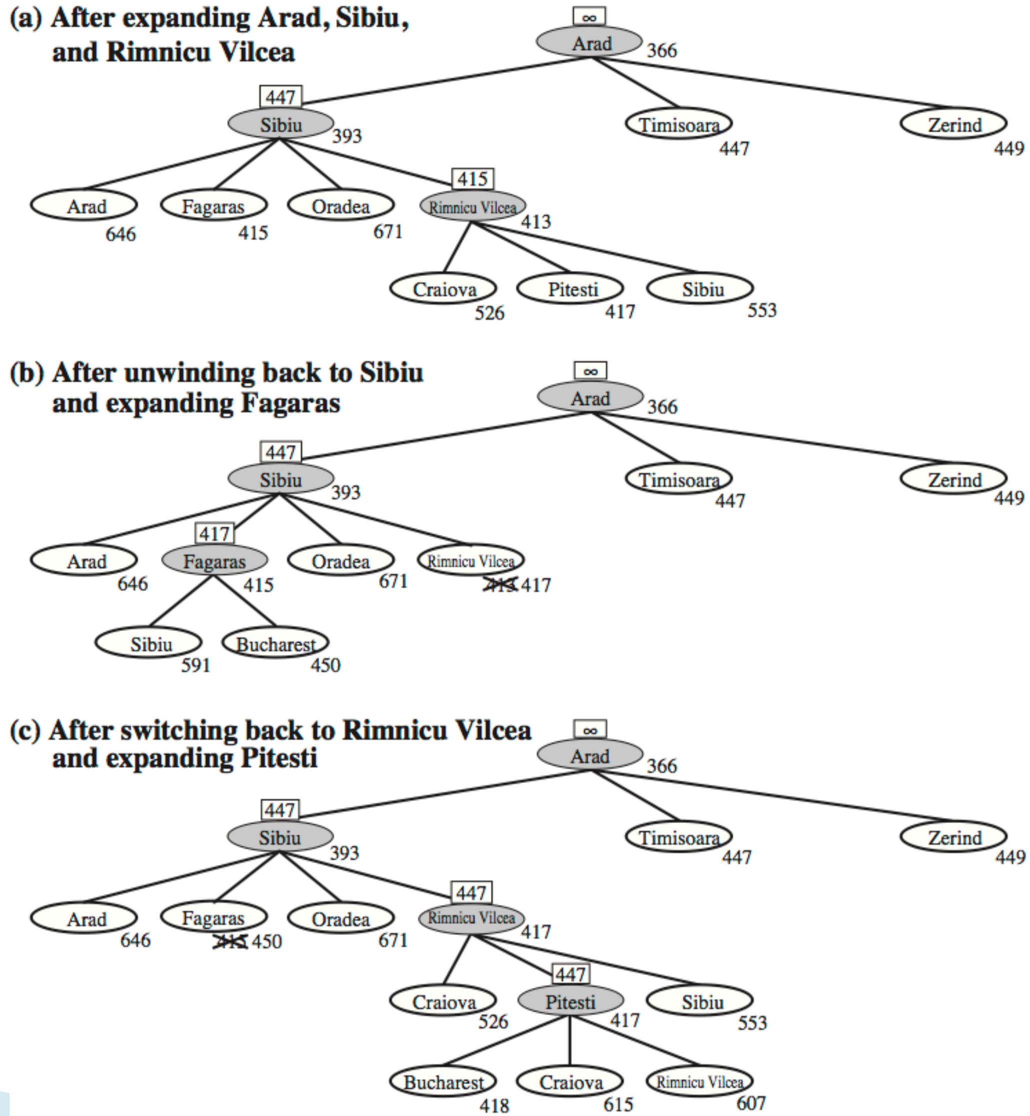
Memory-Bounded Heuristic Search

- Iterative-Deepening A* (IDA*)
  - Like iterative deepening, except uses limit on f rather than depth
  - Next f limit is the smallest f of a node exceeding the limit in the previous iteration
  - Space efficient, but may take long if f values increase slowly

Memory-Bounded Heuristic Search

- Recursive Best-First Search (RBFS)
  - Similar to recursive depth-first search
  - Each node along current path maintains best f value (f-limit) of alternative path from an ancestor
  - If current node's f value exceeds f-limit then backtrack to ancestor and expand alternative path

RBFS Example



Memory-Bounded Heuristic Search

- Recursive Best-First Search (RBFS)
  - Like IDA*, RBFS explores same states many times
  - IDA* and RBFS use only linear memory
  - They cannot take advantage of more memory, if available

Memory-Bounded Heuristic Search

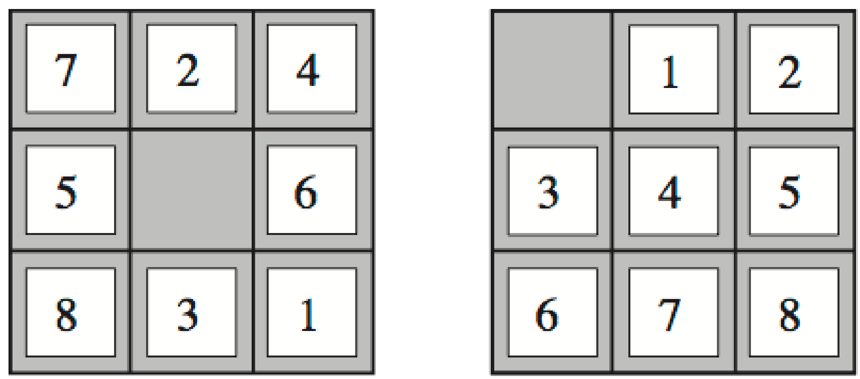
- Simplified Memory-bounded A* (SMA*)
  - Similar to A*
  - Nodes maintain best f-value of any explored node in their subtree
  - If out of memory, remove node with worst f-value and update parent's f-value
  - Complete and optimal if best solution is reachable within memory

Heuristics Revisited

- Why not use $h(n) = 1$?
- How to measure quality of heuristic?
- Effective branching factor b^*
  - Assume A* generates N nodes to find solution at depth d
  - What branching factor needed for a uniform tree of depth d to include N+1 nodes?
  - $N + 1 = 1 + b^* + (b^*)^2 + \ldots + (b^*)^d$
  - Ideally, $b^* = 1$
- E.g., $N=52$, $d=5$, $b^*=1.92$

Heuristics Revisited

- E.g., 8-puzzle
  - h_1 = tiles out of place
  - h_2 = sum of tiles' city block distances



Start State   Goal State

$h_1 = 8$
$h_2 = 3+1+2+2+3+2+2+3 = 18$
Solution cost = 26

Heuristics Revisited

- Values averaged over 100 8-puzzle problems for each d
- Note: $b^*(h_2) \leq b^*(h_1)$

	Search Cost (nodes generated)			Effective Branching Factor		
d	IDS	A*(h_1)	A*(h_2)	IDS	A*(h_1)	A*(h_2)
2	10	6	6	2.45	1.79	1.79
4	112	13	12	2.87	1.48	1.45
6	680	20	18	2.73	1.34	1.30
8	6384	39	25	2.80	1.33	1.24
10	47127	93	39	2.79	1.38	1.22
12	3644035	227	73	2.78	1.42	1.24
14	–	539	113	–	1.44	1.23
16	–	1301	211	–	1.45	1.25
18	–	3056	363	–	1.46	1.26
20	–	7276	676	–	1.47	1.27
22	–	18094	1219	–	1.48	1.28
24	–	39135	1641	–	1.48	1.26

Heuristics Revisited

- Heuristic h_2 dominates h_1 if, for all nodes n, $h_2(n) \geq h_1(n)$
- Implies A* using h_2 will typically generate fewer nodes than A* using h_1
- "City block distance" dominates "misplaced tiles"
- In general, want h(n) to be consistent and close to true solution cost from node n
  - But still be fast to compute

Designing Heuristics

- Relaxed problems
  - h(n) = cost of solution to relaxed problem
  - E.g., 8-puzzle where you can swap tiles
- Subproblems
  - h(n) = cost of solution to subproblem
  - E.g., get half the tiles in correct position
- Learning from experience
  - Collect experience as (state, solution cost) pairs
  - Learn h(n): state → solution cost

Summary

- Problem-solving agent
- Formulating problems
- Search
- Uninformed search (Iterative-Deepening)
- Informed (heuristic) search (A*)
- Admissible heuristics