Search

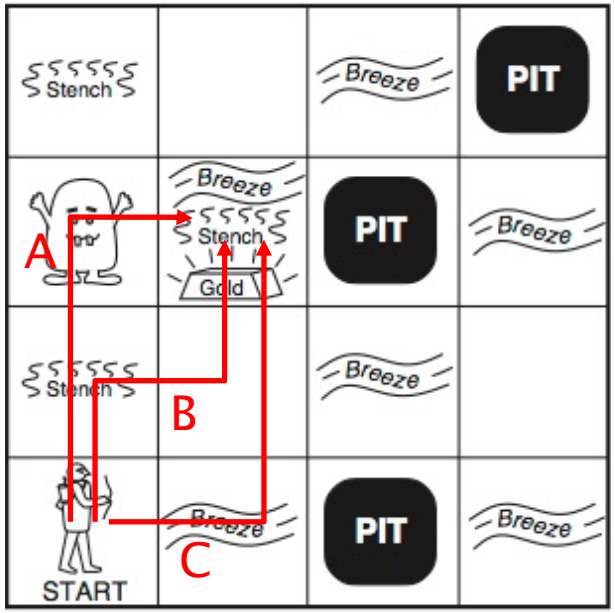
Larry Holder
School of EECS
Washington State University

Overview

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

Problem-Solving Agent

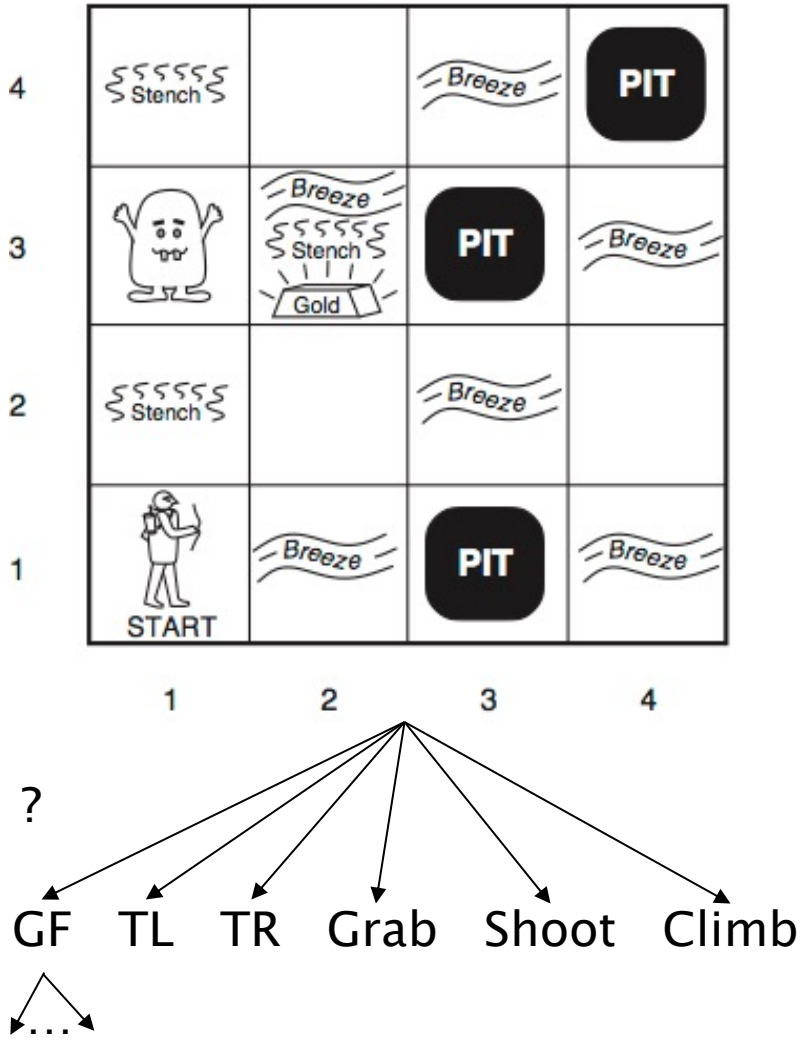
- Type: Goal-based
- Feature-based state representation
  - Agent location, orientation, …
- Assume solution is a fixed sequence of actions
- Rationality: Achieve goal (minimize cost)
- Search for sequence of actions achieving goal



Which solution: A, B or C?

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)* // *action* = *seq*[0]
  seq = REST *(seq)* // *seq* = *seq*[1…]
  return *action*

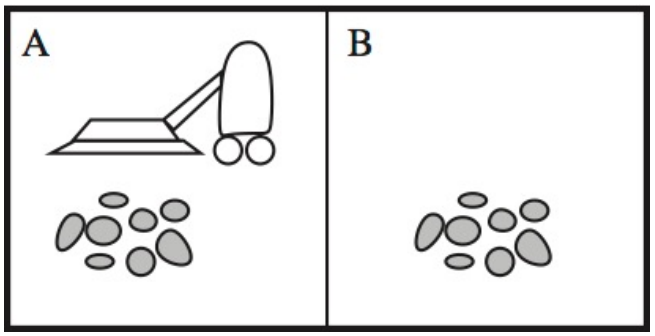
Well-Defined Problems (5 parts)

1. **Initial state**
   - State: relevant features of the problem
2. **Actions**
   - Action: state → state
   - ACTIONS(*s*) returns set of actions applicable to state *s*
3. **Transition model**
   - RESULT(*s*,*a*) returns state after taking action *a* in state *s*
4. **Goal test**
   - True for any state satisfying goal
5. **Path cost**
   - Sum of costs of individual actions along a path
   - Path: sequence of states connected by actions
   - Step cost *c(s,a,s')*: cost of taking action *a* in state *s* to reach state *s'*

Well-Defined Problems: Terminology

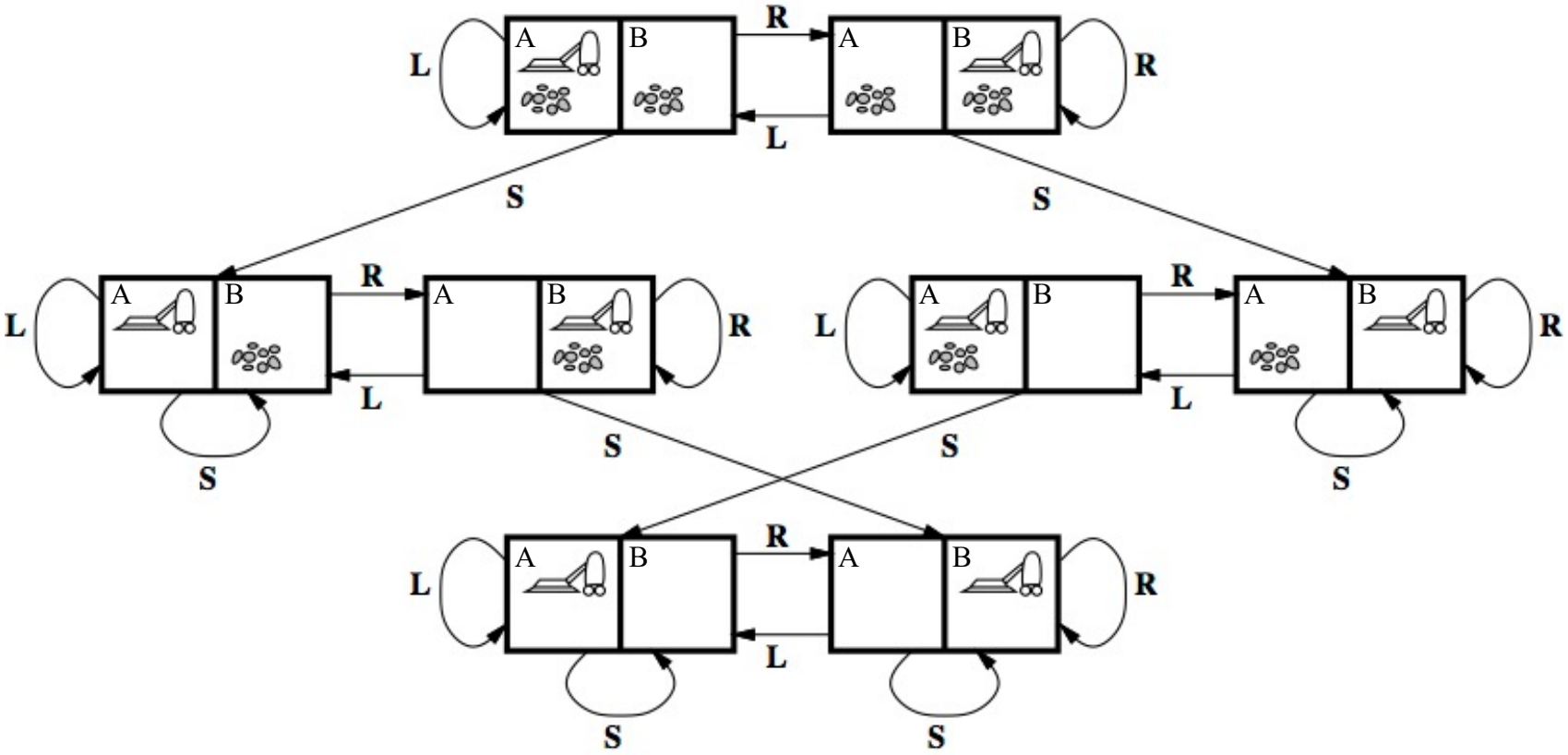
- State space: set of all states reachable from the initial state by any sequence of actions
  - State space forms a directed graph of nodes (states) and edges (actions)
- Solution: sequence of actions leading from the initial state to a goal state
- Optimal solution: solution with minimal path cost

Vacuum World Problem



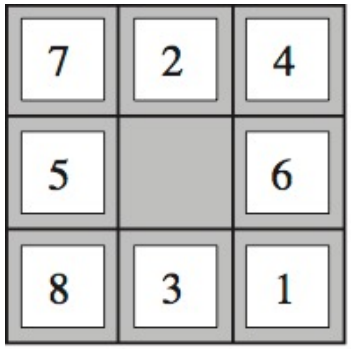
- State representation
  - Location: A, B
  - Cleanliness of rooms: Clean, Dirty
  - Example state: (A,Clean,Clean)
  - How many unique states?
- Initial state: Any state
- Actions: Left, Right, Suction
- Transition model
  - E.g., Result((A,Dirty,Clean), Suction) = ?
- Goal test: State = (?,Clean,Clean)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

Vacuum World State Space

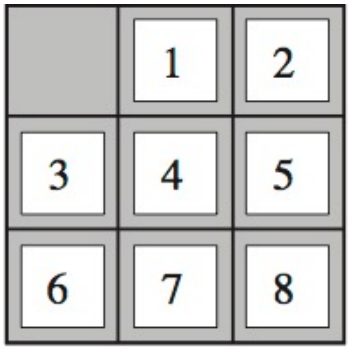


8-Puzzle

- 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 tile Up, Down, Left or Right
- Transition model
- Goal test: State matches Goal State
- Path cost: Number of steps in path (step cost = 1)



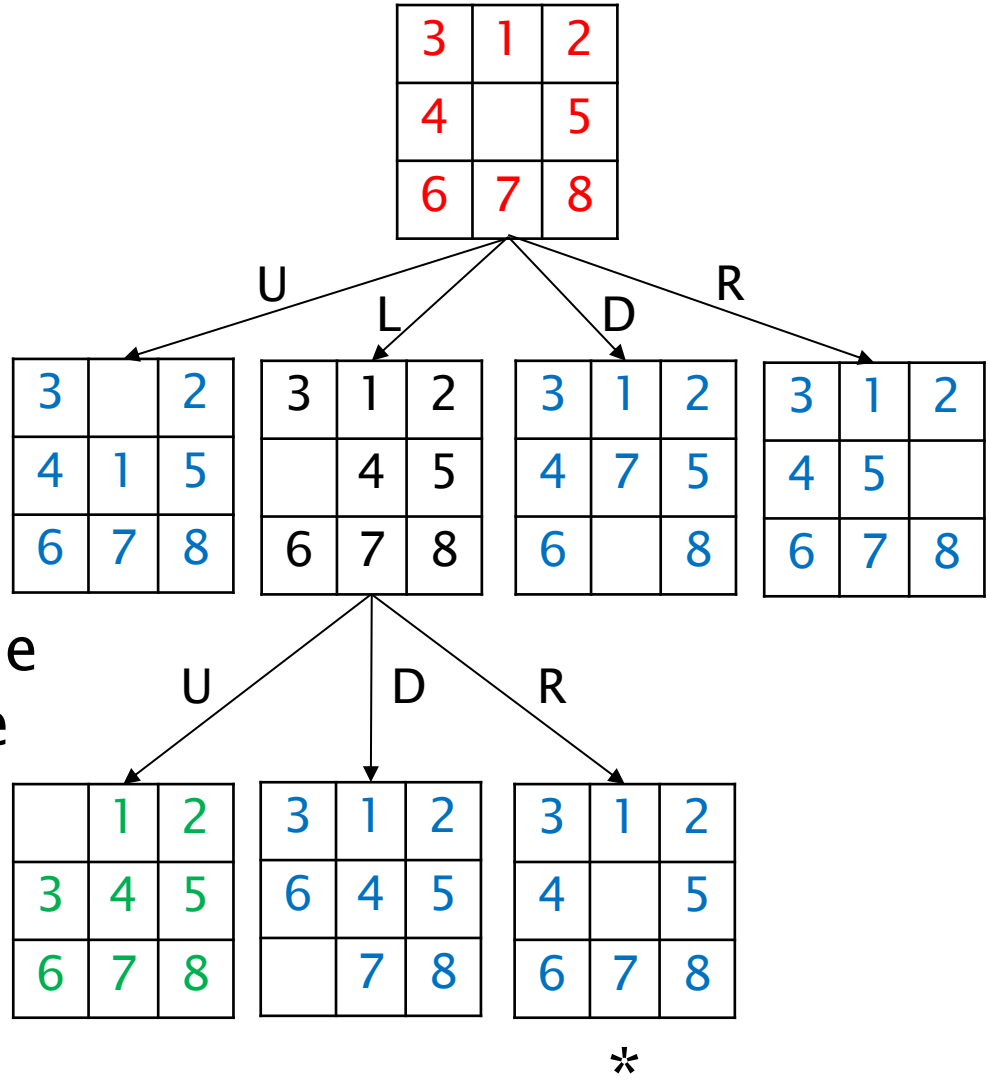
Start State



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://tristanpenman.com/demos/n-puzzle

Caution: This app may not produce answers consistent with algorithms used in this class.

1	2	3
	4	5
7	8	6

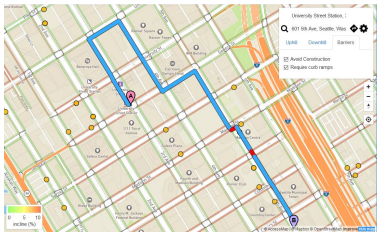
Initial State

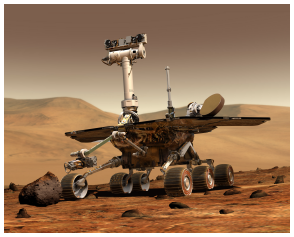
1	2	3
4	5	6
7	8	

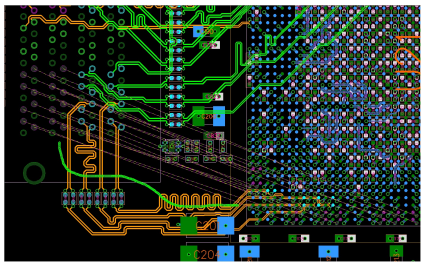
Goal State

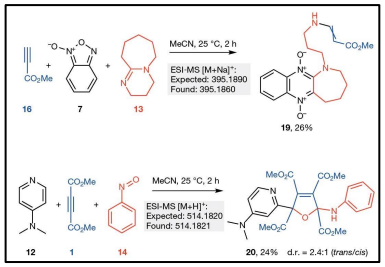
Real-World Search Problems

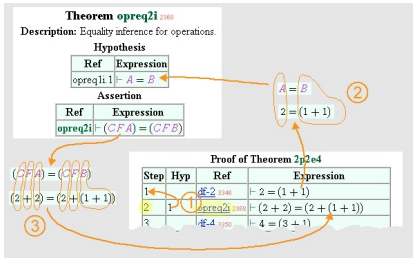
- Route finding
- Robot navigation
- Factory assembly
- Circuit layout
- Chemical design
- Mathematical proofs
- Game playing







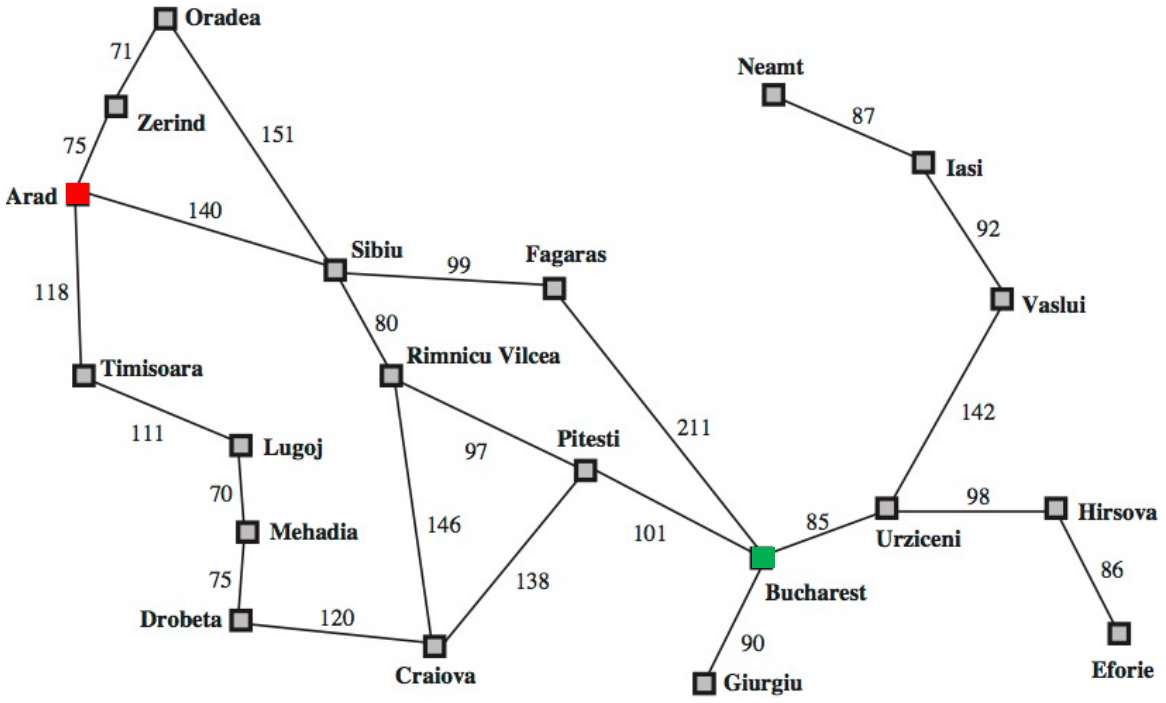




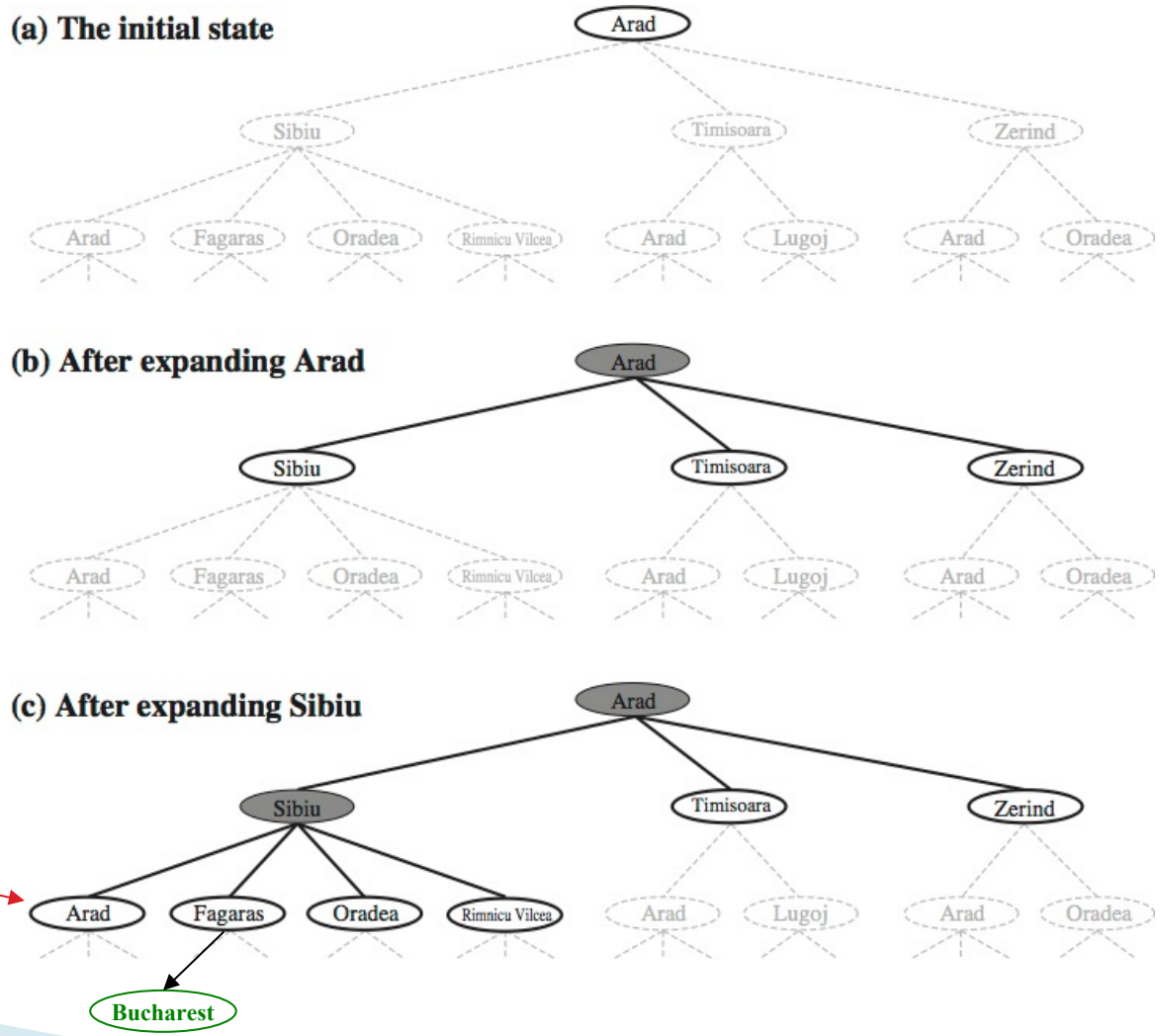
- Most of AI can be cast as a search problem

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

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

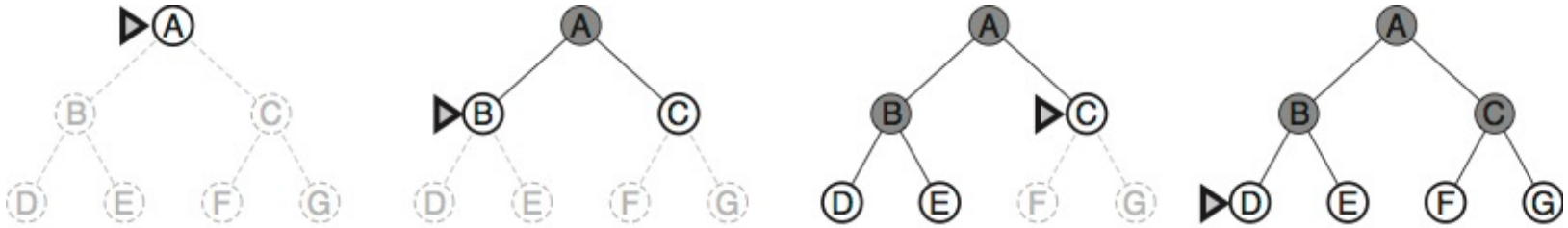
Breadth-First Search

- 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)

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

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

Breadth-First Search

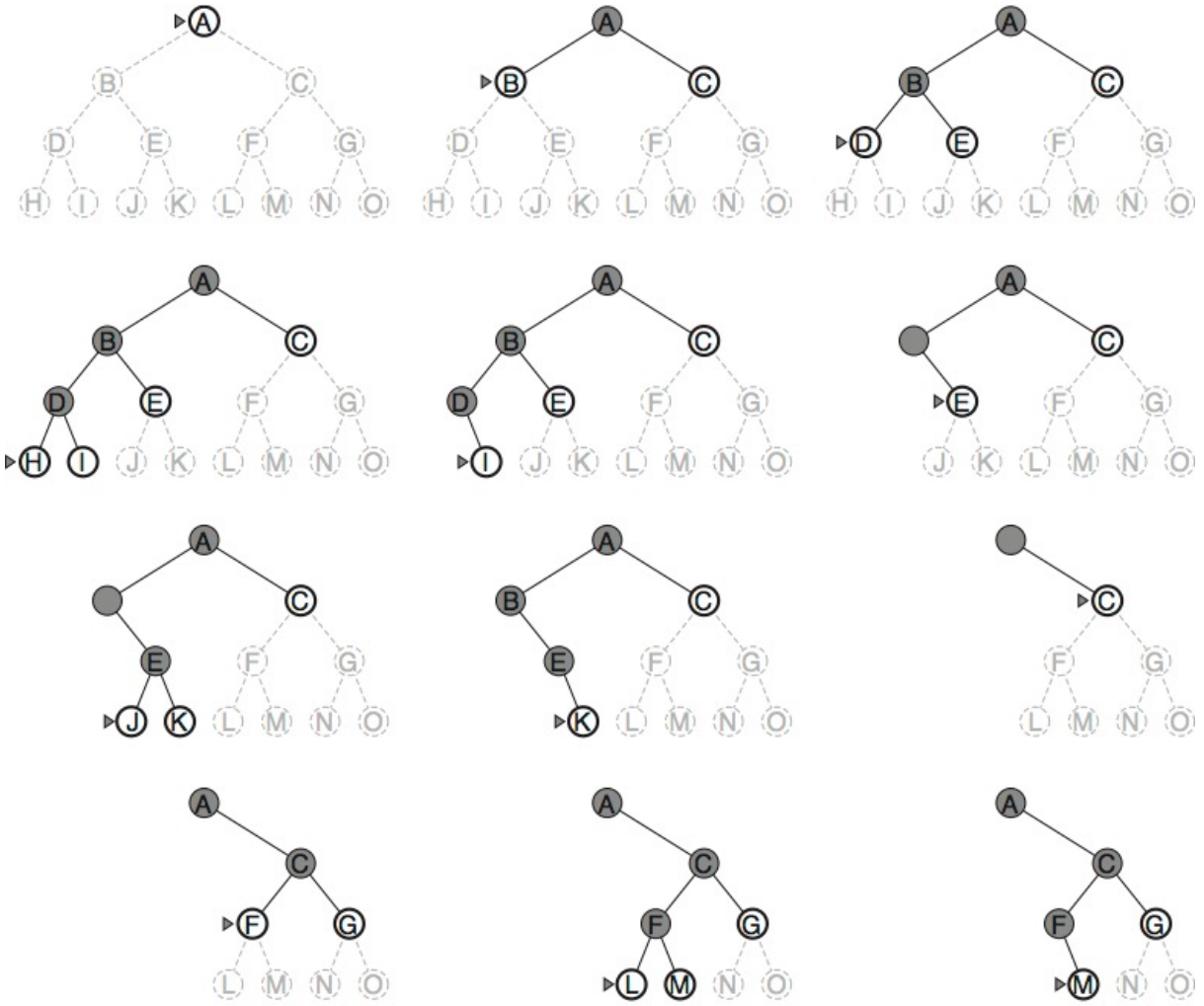
- Exponential complexity $O(b^d)$
- For $b=4$, 1KB/node, 1M nodes/sec

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})

Depth-First Search

- Always expand the deepest node
- Frontier is a simple stack
  - Push nodes to front, pop 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

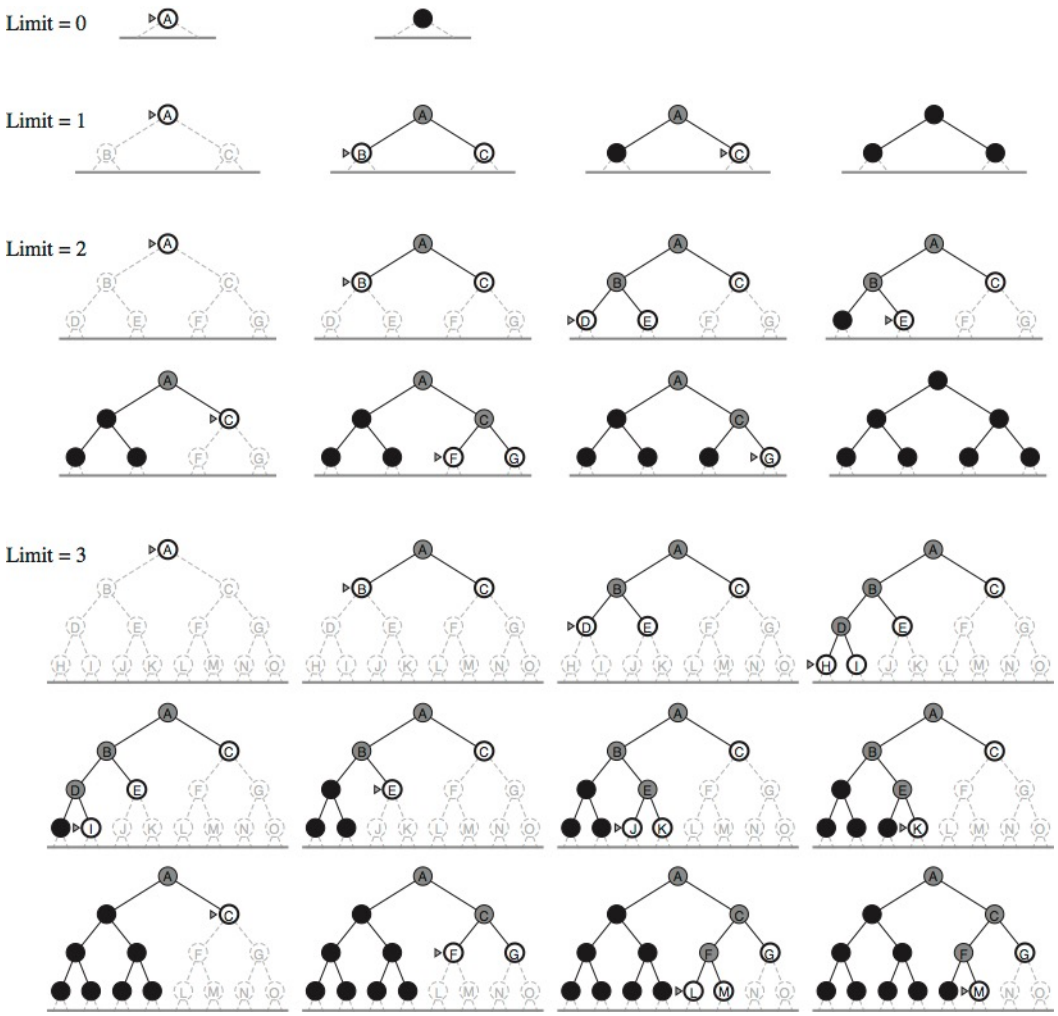
- Limit DFS depth to ℓ
- Still incomplete, if $\ell < d$
- Non-optimal if $\ell > d$
- Time complexity: $O(b^{\ell})$
- Space complexity: $O(b\ell)$

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

Uninformed Search Strategies

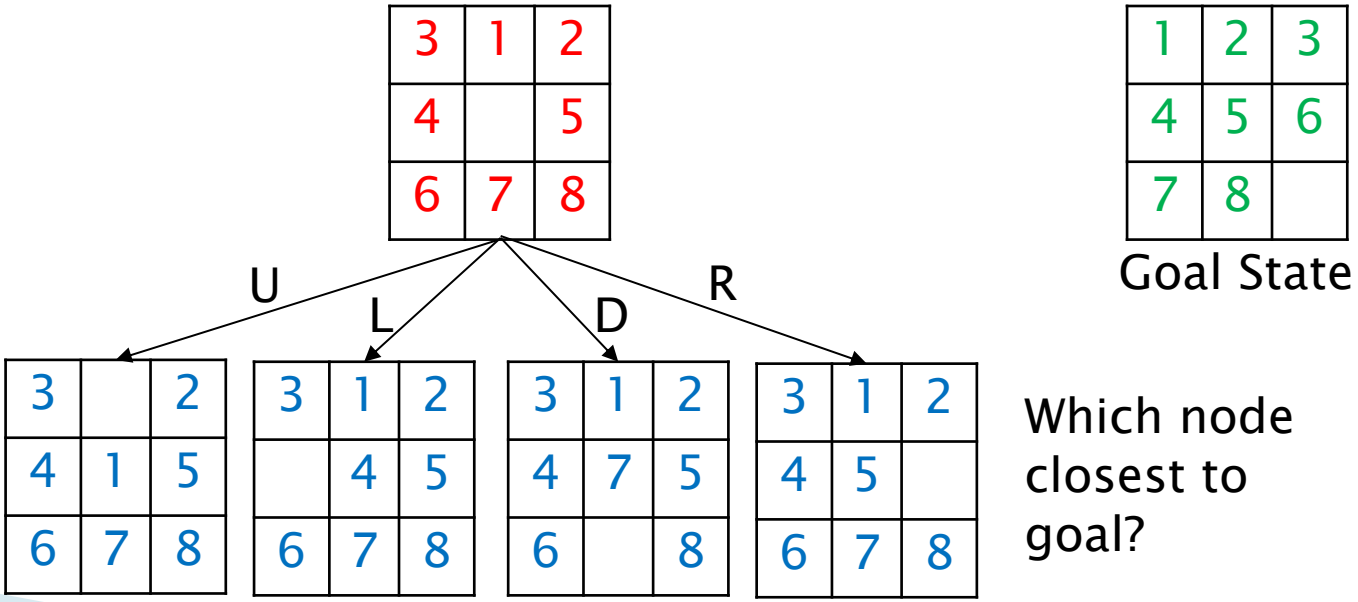
Criterion	Breadth-First	Depth-First	Depth-Limited	Iterative Deepening
Complete	Yes*	No	No	Yes*
Time	$O(b^d)$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$
Space	$O(b^d)$	$O(bm)$	$O(b\ell)$	$O(bd)$
Optimal	Yes**	No	No	Yes**

* Complete if b is finite

** Optimal if step costs all the same

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

Heuristic Search Strategies

- Greedy best-first search
  - Choose node on frontier with smallest h(n)
- A* search
  - Choose node on frontier with smallest f(n)
- Hill-climbing
  - Choose node with smallest h(n)
  - Discard other nodes on frontier

Greedy Best-First Graph Search

```
function BEST-FIRST-SEARCH (problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, COST = h(node)
  frontier ← priority queue ordered by 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 COST then
        replace that frontier node with child
```

Why not check Goal-Test here?

Why is this test necessary?

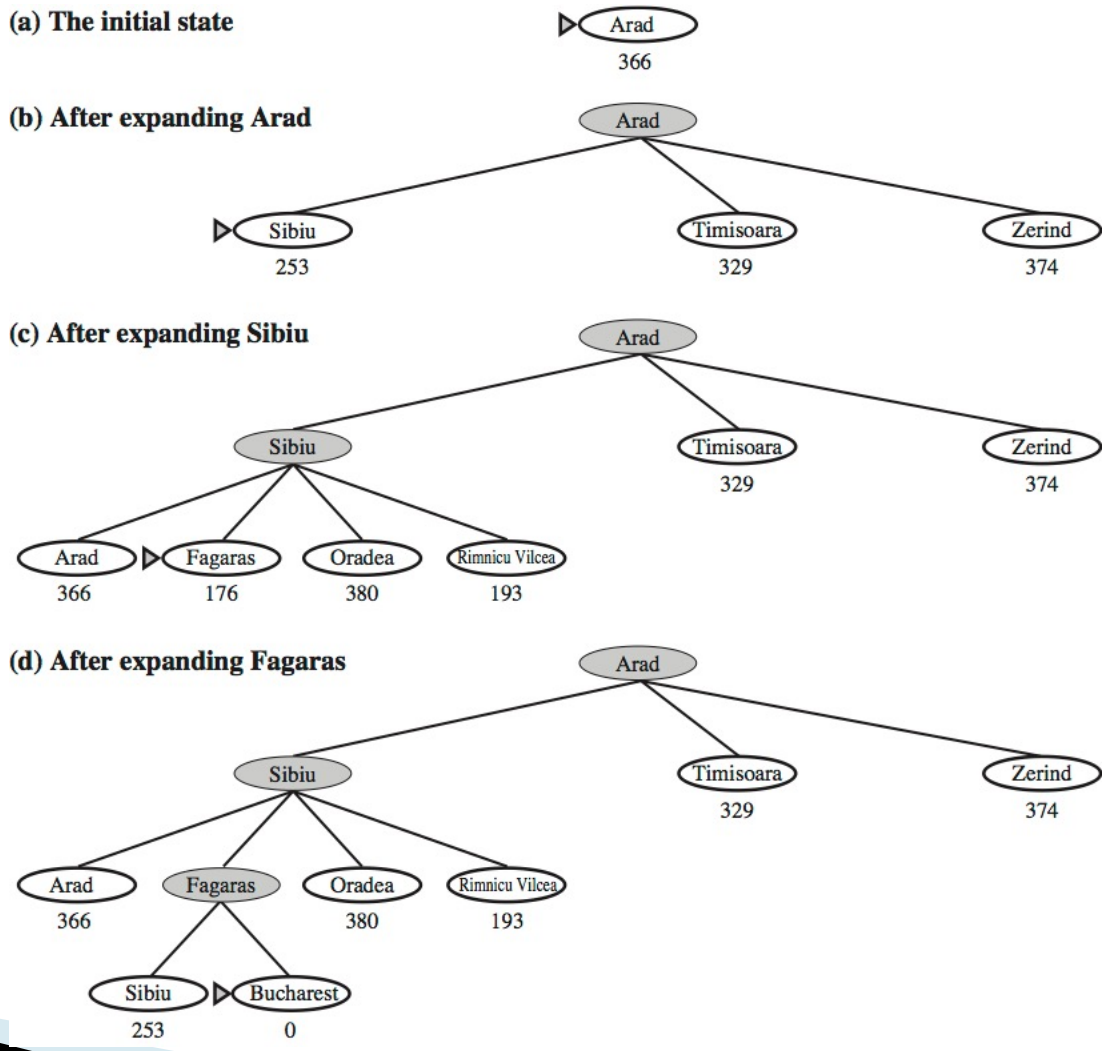
Greedy Best-First Search

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

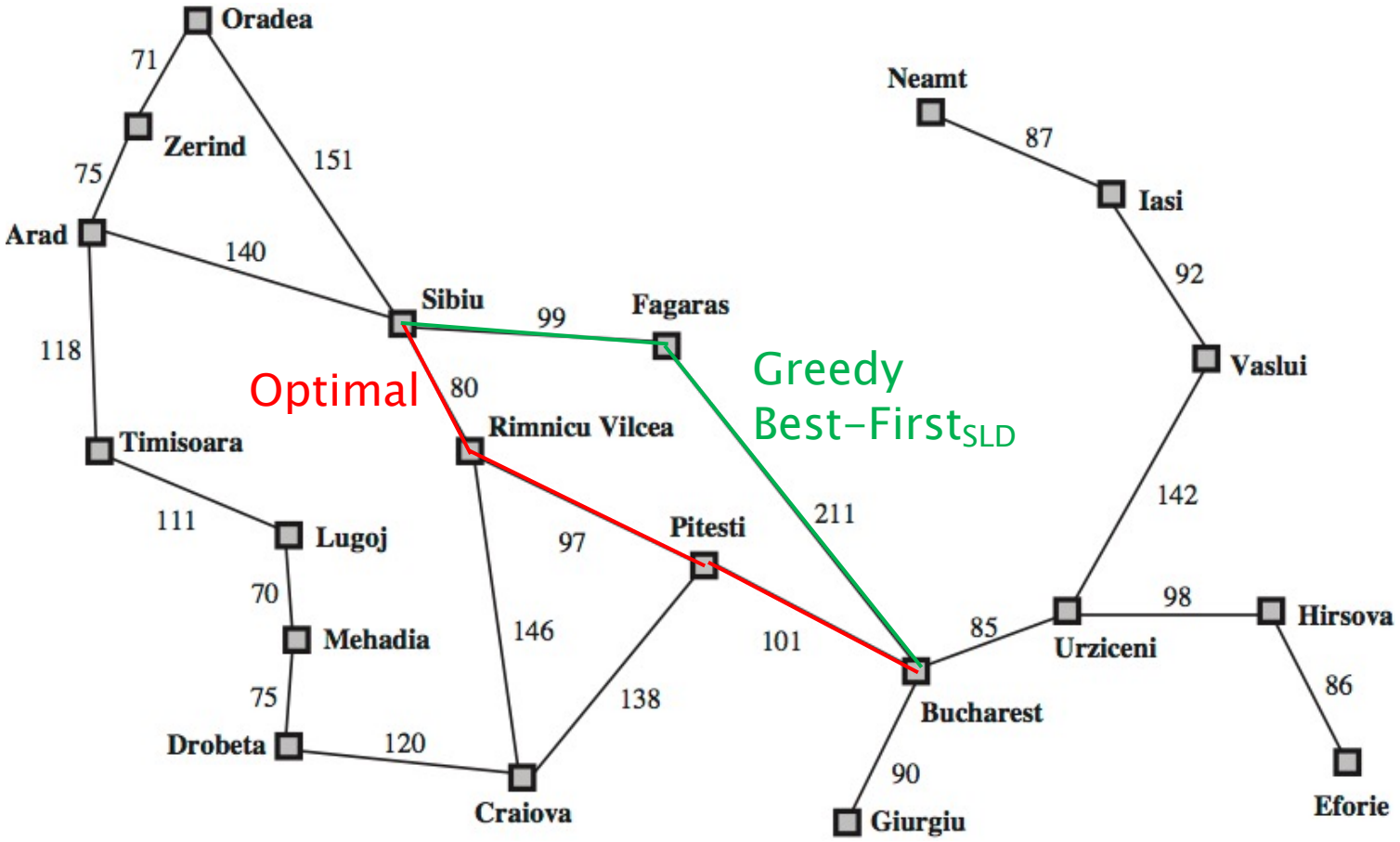
Straight-line distances to Bucharest:

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



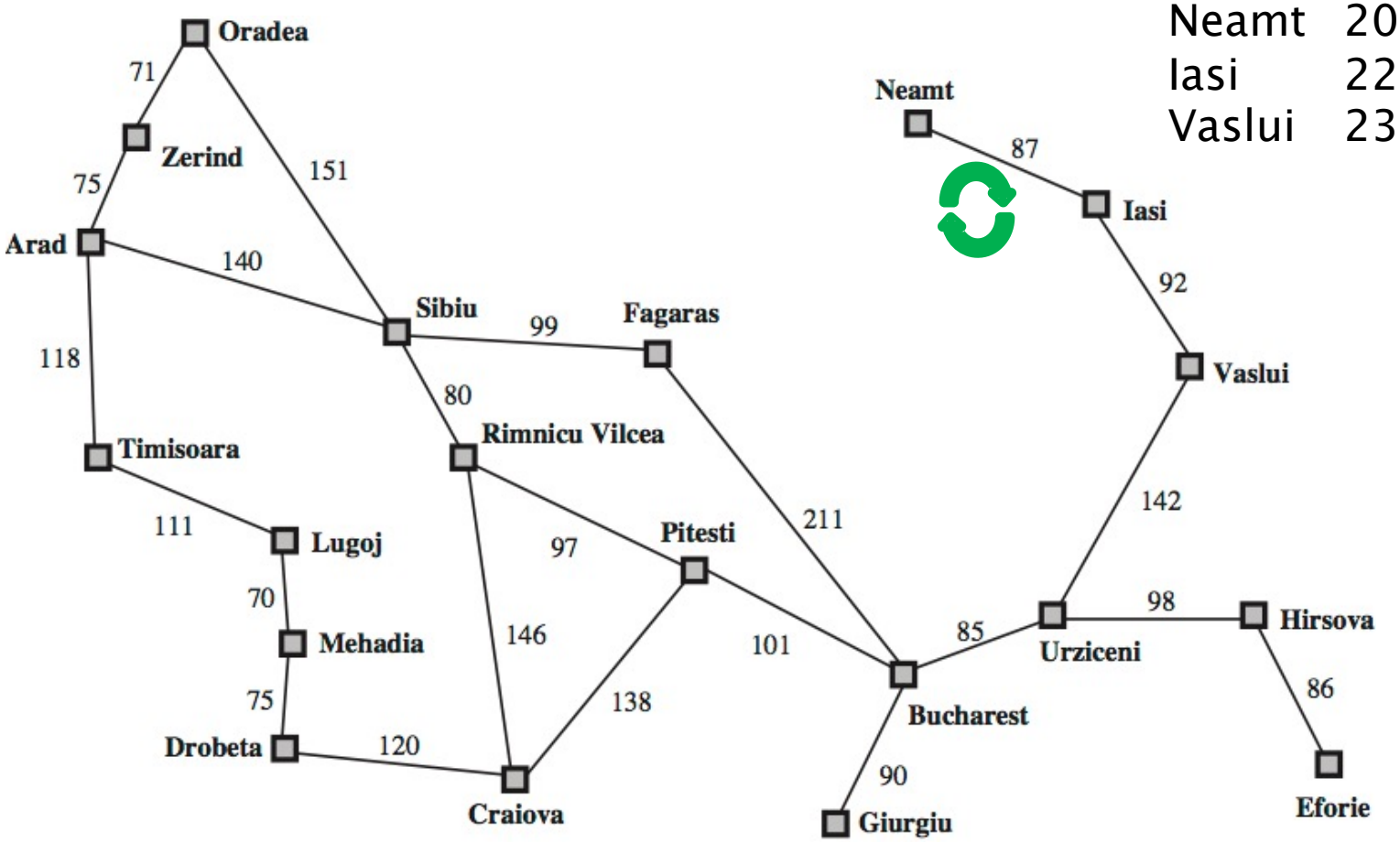
Greedy Best-First Search Example: Arad to Bucharest



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

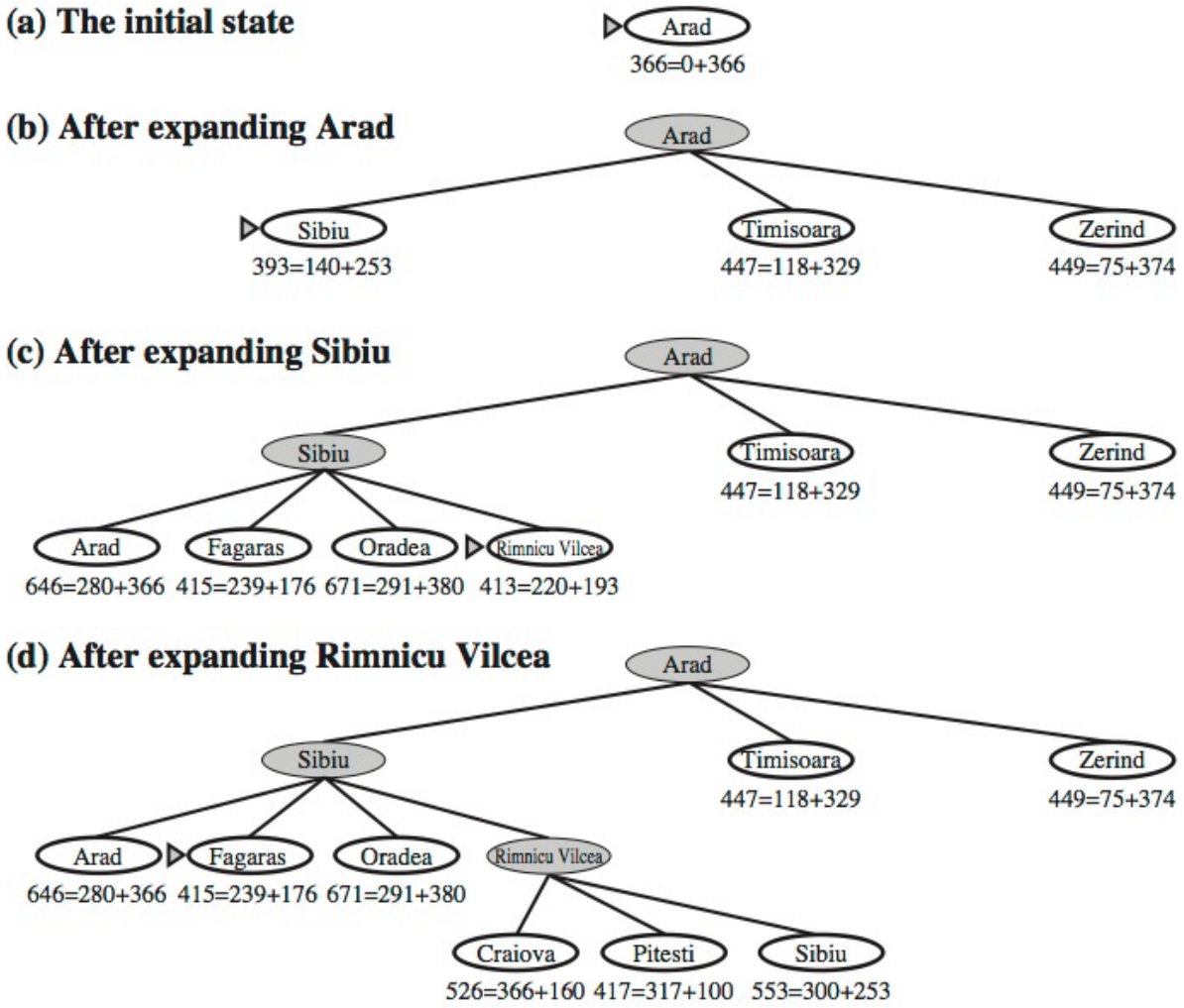
DEMO

A* Search

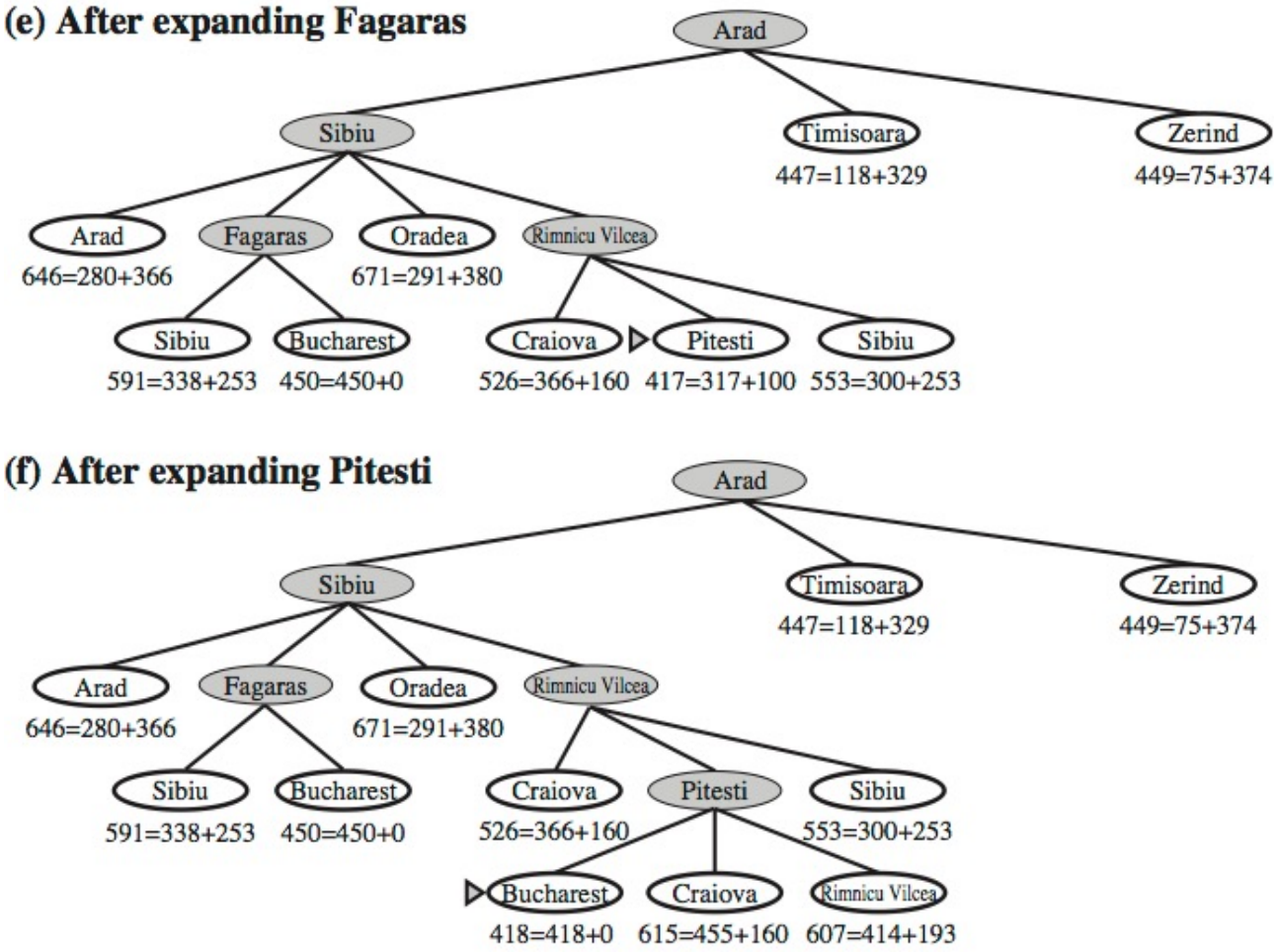
History: A* generalizes over algorithms A1 and A2, which were heuristic extensions to Dijkstra's shortest path algorithm.

- f(n) = g(n) + h(n)
  - Estimated cost of solution through n
- Best-First-Search using Cost = f(n)
- Complete and optimal assuming some constraints on h(n)

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 admissible 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 triangle inequality 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

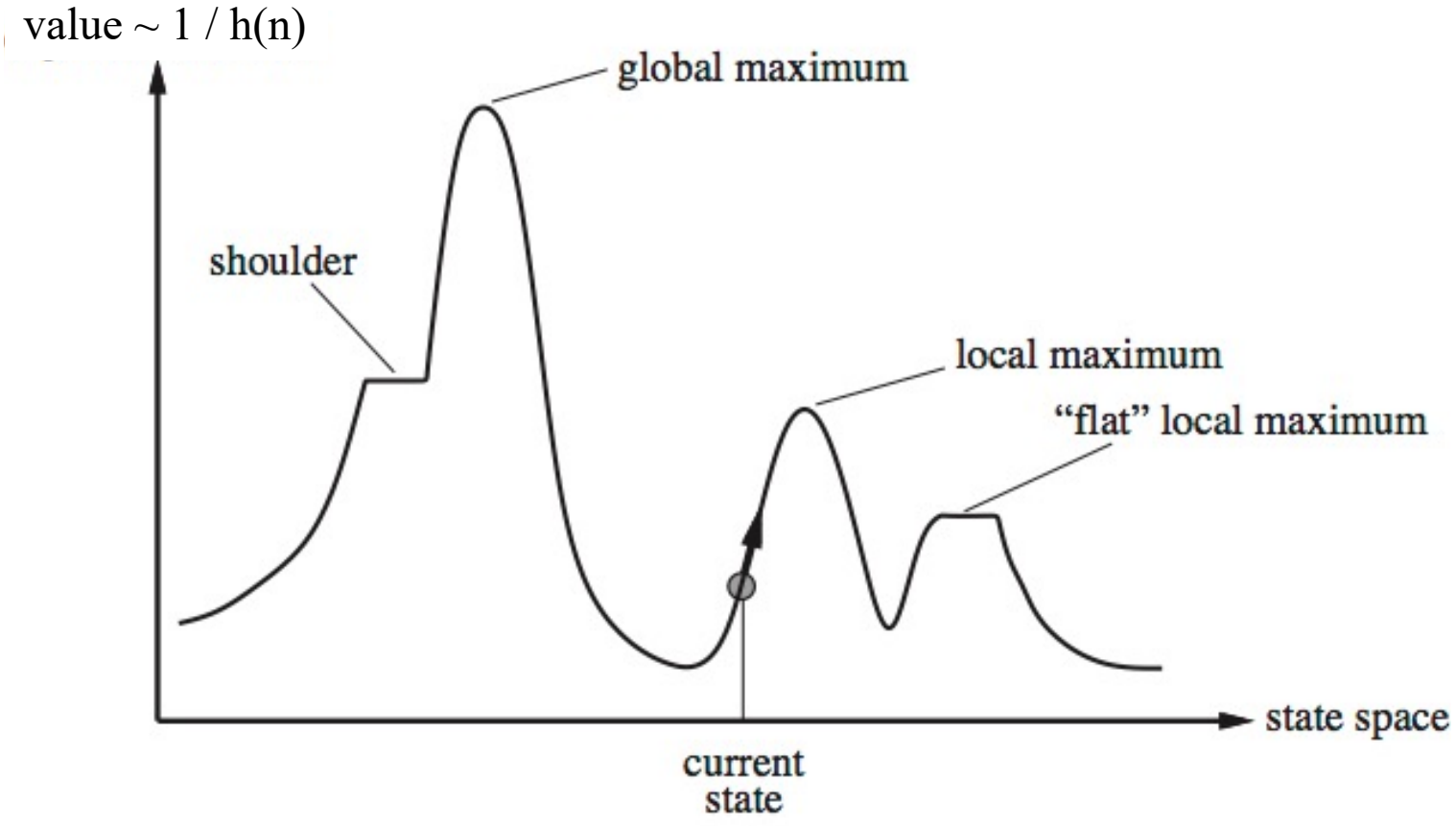
DEMO

Water Jug Problem

- **States**: Water jugs of various sizes with some amount of water in them
  - Jug j has capacity c(j) and contains w(j) gallons of water
- **Initial state**: Water jugs all empty: w(j) = 0
- **Actions**:
  - Fill a jug to the top with water from water source
  - Pour water from one jug into another until second jug is full or first jug is empty
  - Empty all water from a jug
- **Transition model**:
  - Fill(j): w(j) = c(j)
  - Pour(j1,j2):
    - w(j1) = max(0,w(j1)–c(j2)+w(j2))
    - w(j2) = min(c(j2),w(j1)+w(j2))
  - Empty(j): w(j) = 0
- **Goal test**: Some w(j) = X
- **Path cost**: Number of actions

Die Hard with a Vengeance (1995)
c(1)=3, c(2)=5, Goal: w(2)=4

State-Space Landscape



Hill-Climbing Search

```
function HILL-CLIMBING (problem) returns a state which is a local maximum
  current ← MAKE-NODE(problem.INITIAL-STATE)
  loop do
    next ← current
    for each action in problem.ACTIONS(node.STATE) do
      child ← CHILD-NODE(problem, node, action)
      if child.VALUE > next.VALUE then next ← child        // VALUE ~ 1 / h(n)
    if next.VALUE ≤ current.VALUE then return current.STATE   // Goal test?
    current ← next
```

- Also called “steepest ascent” or “greedy local search”
- Gets stuck on local maxima and plateaus
- Stochastic hill climbing
  - Random selection of *next* node

Hill-Climbing

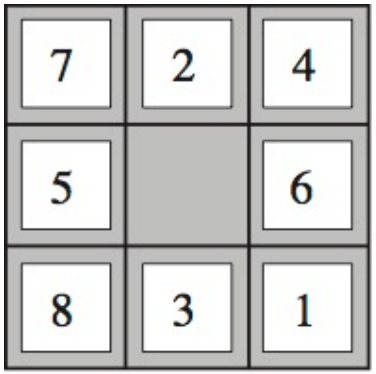
- Complete?
- Optimal?
- Time complexity?
- Space complexity?

Designing Heuristics

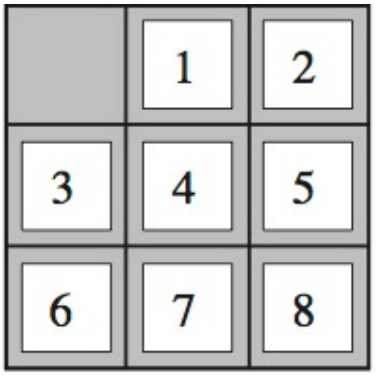
- Why not use h(n) = 1?
- Why not use h(n) = actual optimal cost to goal from n?
- How to measure quality of heuristic?

Designing Heuristics

- 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

Designing Heuristics

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

	Search Cost (nodes generated)		
d	IDS	$A^*(h_1)$	$A^*(h_2)$
2	10	6	6
4	112	13	12
6	680	20	18
8	6384	39	25
10	47127	93	39
12	3644035	227	73
14	–	539	113
16	–	1301	211
18	–	3056	363
20	–	7276	676
22	–	18094	1219
24	–	39135	1641

Designing Heuristics

- 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 "tiles out of place"
- In general, want h(n) to be:
  - Admissible and consistent
  - Close to actual solution cost from node n
  - But still 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