CSCI 3202: Intro to Artificial Intelligence

Lecture 17 (Review): Midterm

Rachel Cox

Department of Computer Science

Exam Info

What is an agent?

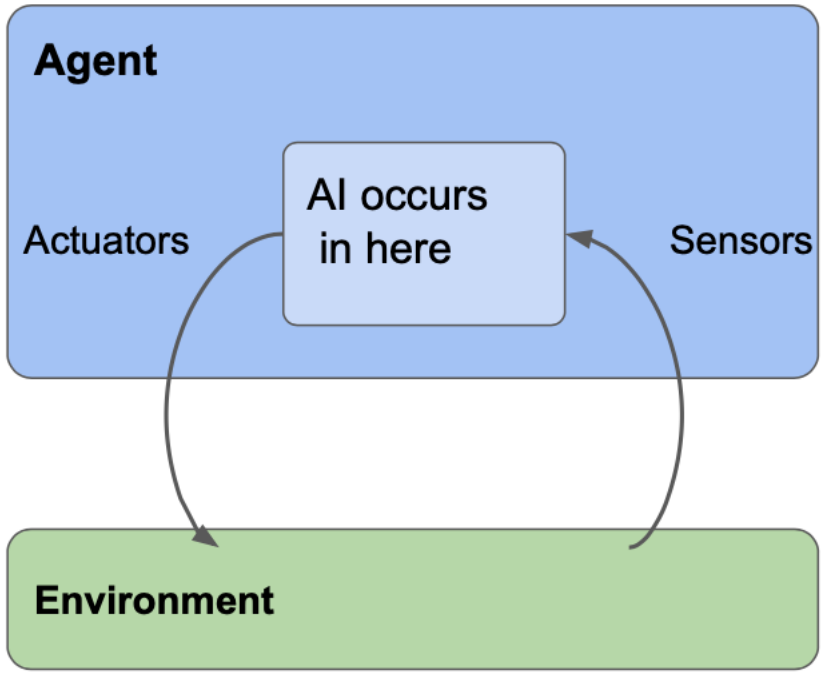
An **agent** is an entity that perceives and acts.

- Perceives via sensors (percepts)
- Acts via actuators (actions)

A **rational agent** is one that does the “right” thing.

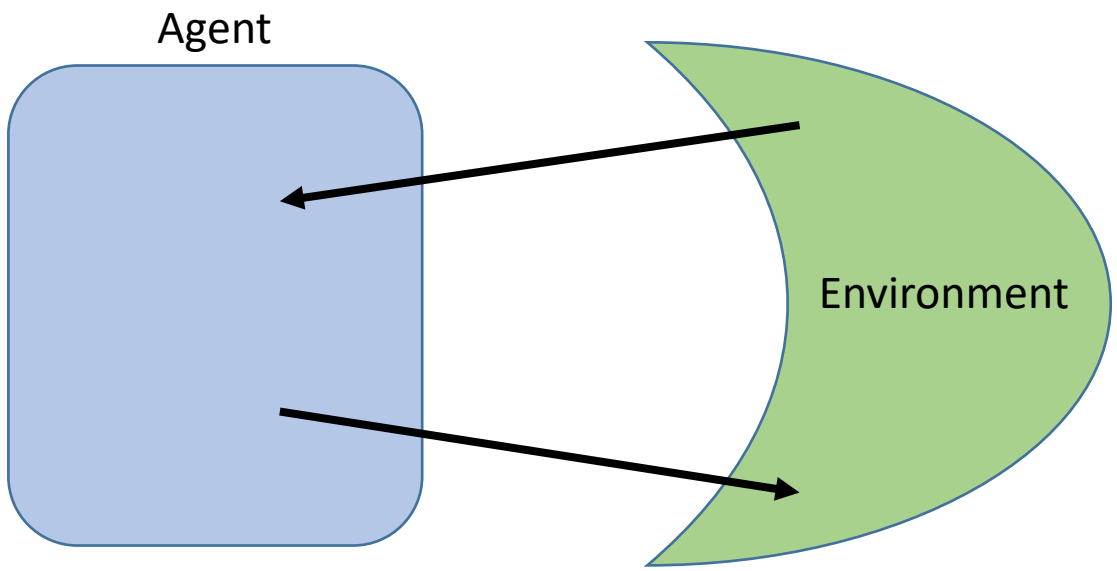
Rationality based on the following four principles:

1) performance measure (utility)
2) environment familiarity
3) actions that are possible
4) sequences of percepts (memory)



What is an agent?

Perception-Action Cycle



How does the agent make decisions based on the sensory data?

- Can’t yet build one system that does it all, but we can build a system that does one thing well.

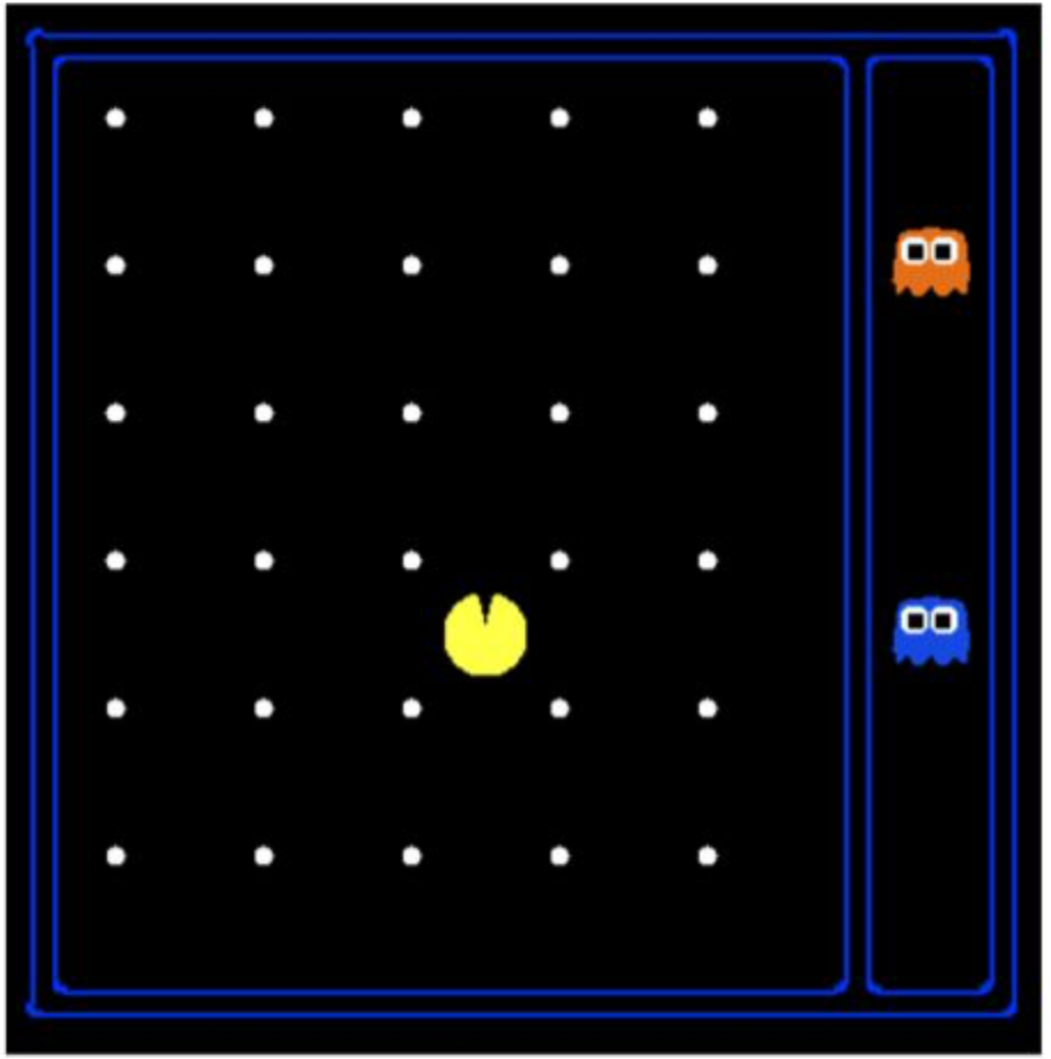
Task Environments

Environment characteristic	Solitaire	Backgammon	Bidding on eBay	Taxi
Fully/partially observable?				
Deterministic/ stochastic?				
Episodic/ sequential task?				
Static / dynamic environment?				
Discrete / continuous?				
Single/ multi-agent?				

Search

A **search problem** consists of:

1. State space
2. Transition model
3. Actions
4. Initial state
5. Goal test



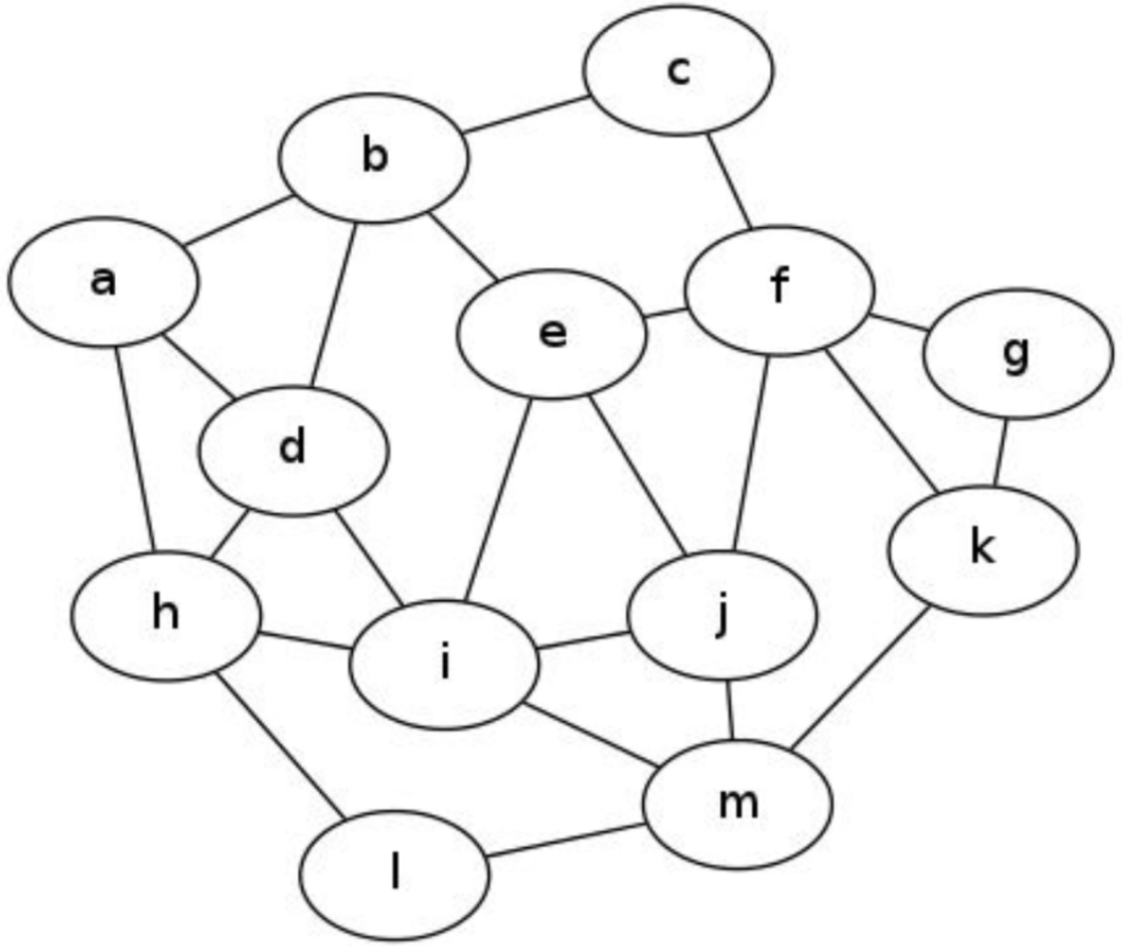
Breadth_First Search

```
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 ← a FIFO queue with node as the only element
  explored ← an empty set
  loop do
    if EMPTY?(frontier) then return failure
    node ← POP(frontier)  /* chooses the 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 ← INSERT(child, frontier)
```

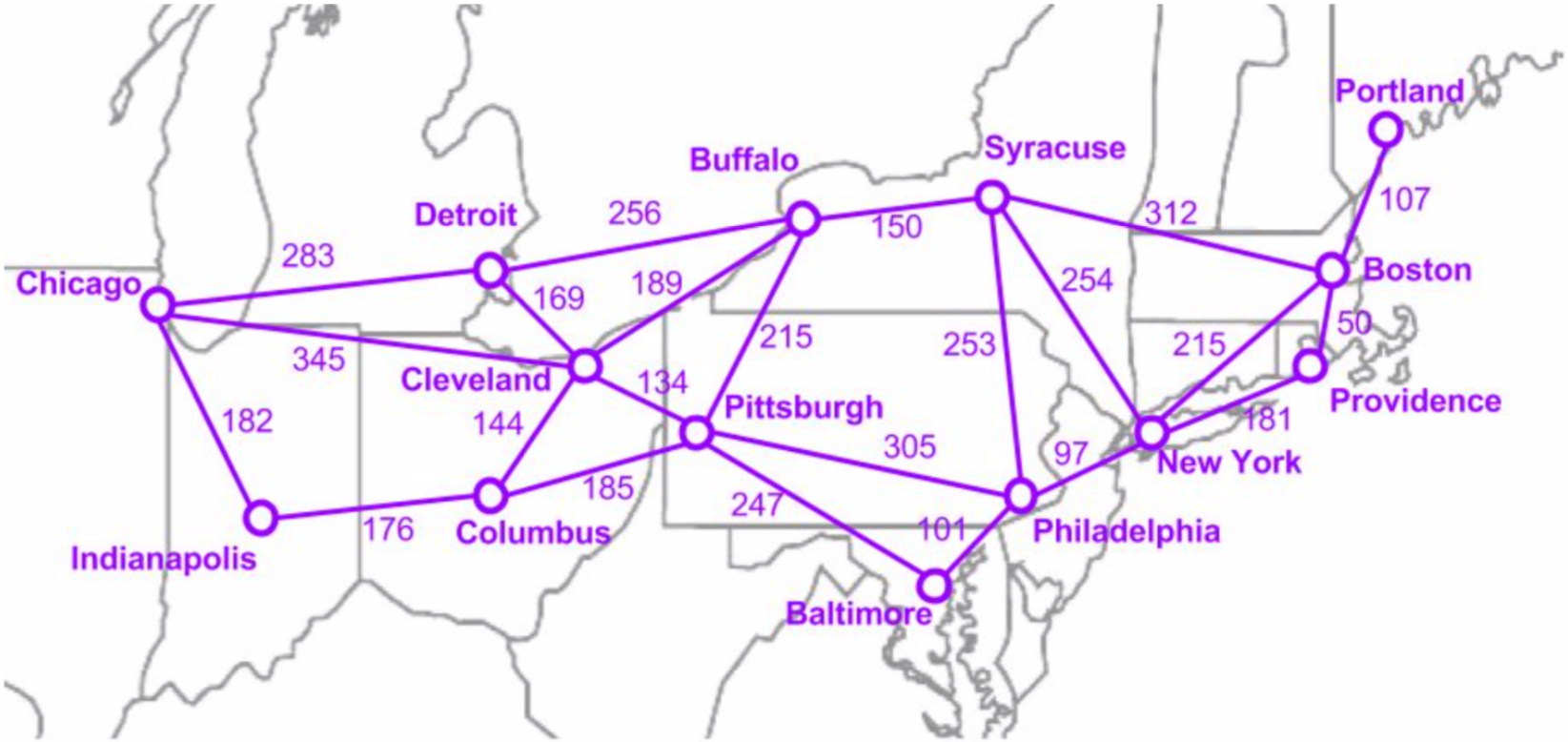
Figure 3.11 Breadth-first search on a graph.

Breadth–first Search (BFS)

Example: Number the nodes in the search graph according to the order in which they would be expanded using BFS to find a path from a to k. Assume that nodes within a layer are expanded in alphabetical order.



Breadth–first Search (BFS)



Complete?

Optimal?

Depth–First Search

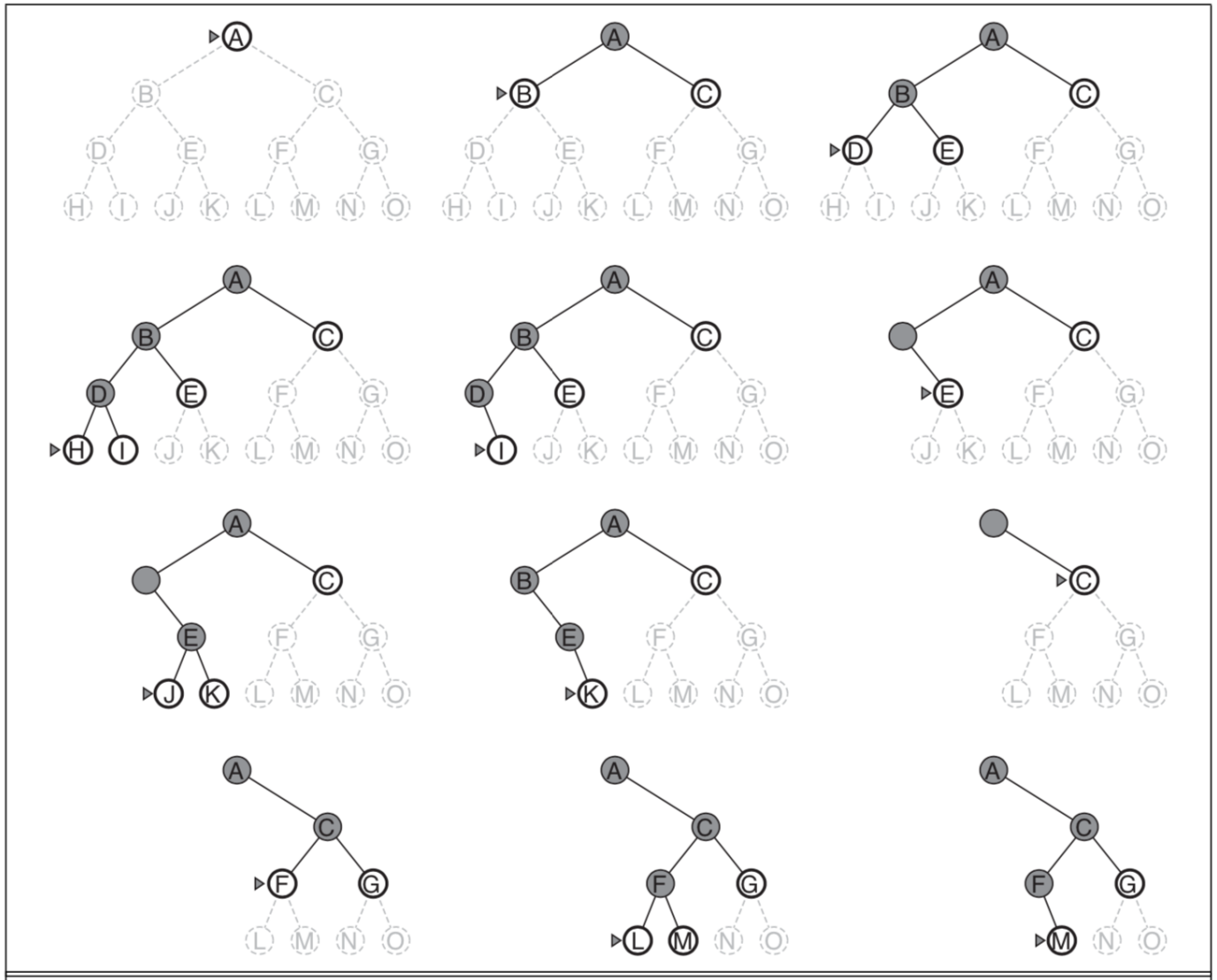
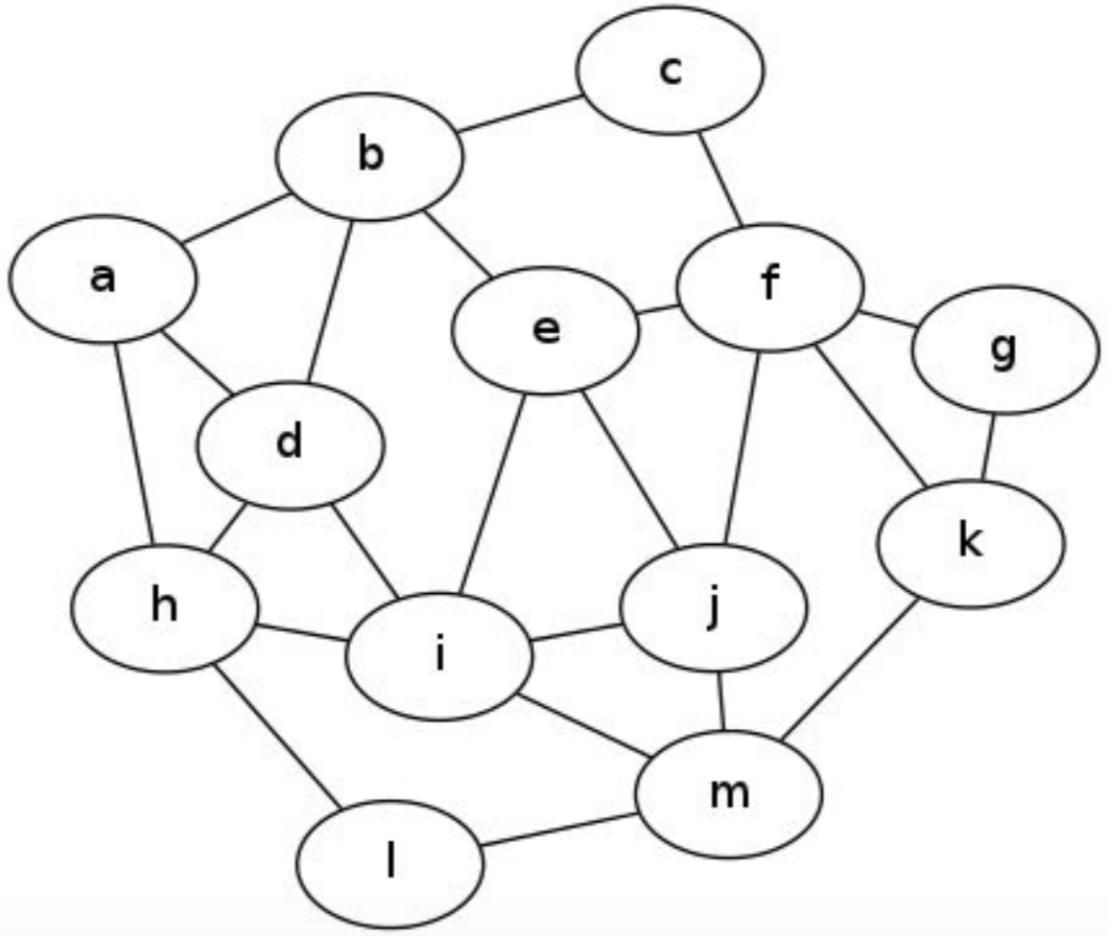


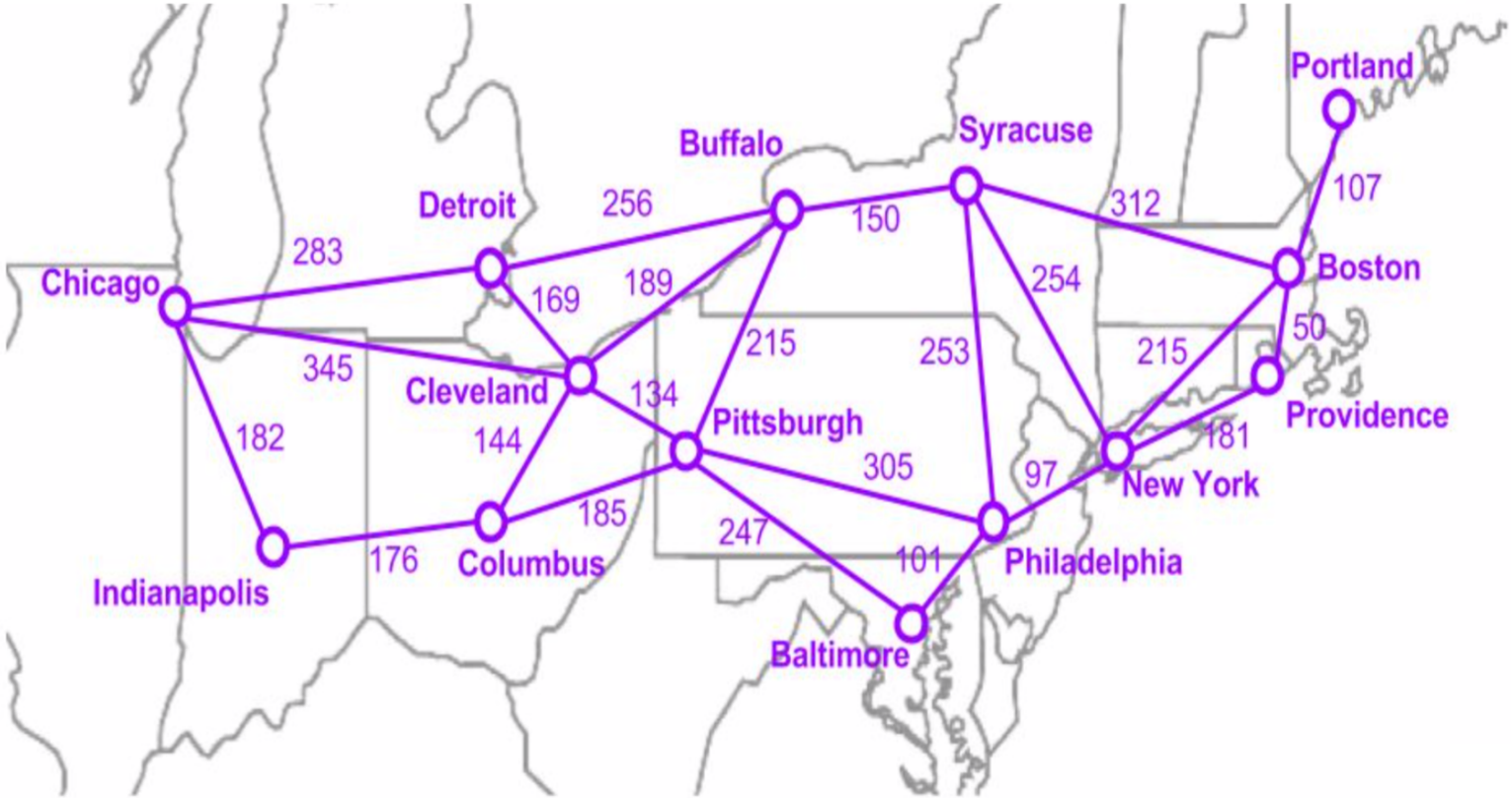
Figure 3.16 Depth-first search on a binary tree. The unexplored region is shown in light gray. Explored nodes with no descendants in the frontier are removed from memory. Nodes at depth 3 have no successors and M is the only goal node.

Depth–First Search (DFS)

Example: Number the nodes in the search graph according to the order in which they would be expanded using DFS to find a path from a to k. Assume that nodes within a layer are added to the queue by alphabetical order. What is the route that DFS yields, if any?



Depth–First Search (DFS)



Complete?

Optimal?

Uniform Cost Search (UCS)

```
function UNIFORM-COST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  frontier ← a priority queue ordered by PATH-COST, with node as the only element
  explored ← an empty set
  loop do
    if EMPTY?(frontier) then return failure
    node ← POP(frontier)  /* chooses the 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 ← INSERT(child, frontier)
      else if child.STATE is in frontier with higher PATH-COST then
        replace that frontier node with child
```

Figure 3.14 Uniform-cost search on a graph. The algorithm is identical to the general graph search algorithm in Figure 3.7, except for the use of a priority queue and the addition of an extra check in case a shorter path to a frontier state is discovered. The data structure for *frontier* needs to support efficient membership testing, so it should combine the capabilities of a priority queue and a hash table.

A* Search

Uniform-cost search:

$f(n) = g(n)$ (cost to get to *n*)

Greedy:

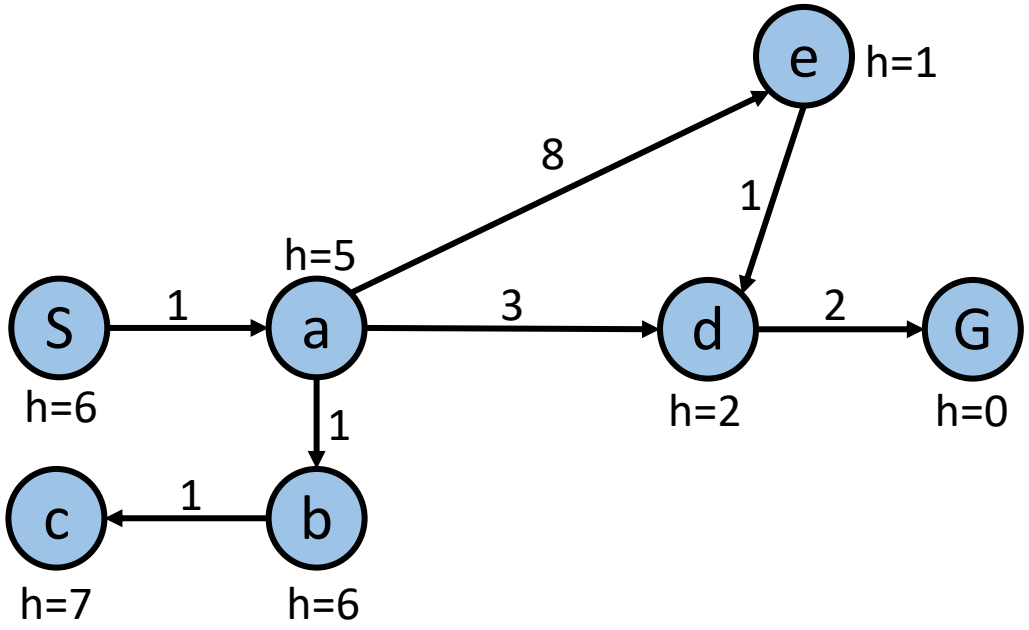
$f(n) = h(n)$ (estimated cost to get from *n* to goal)

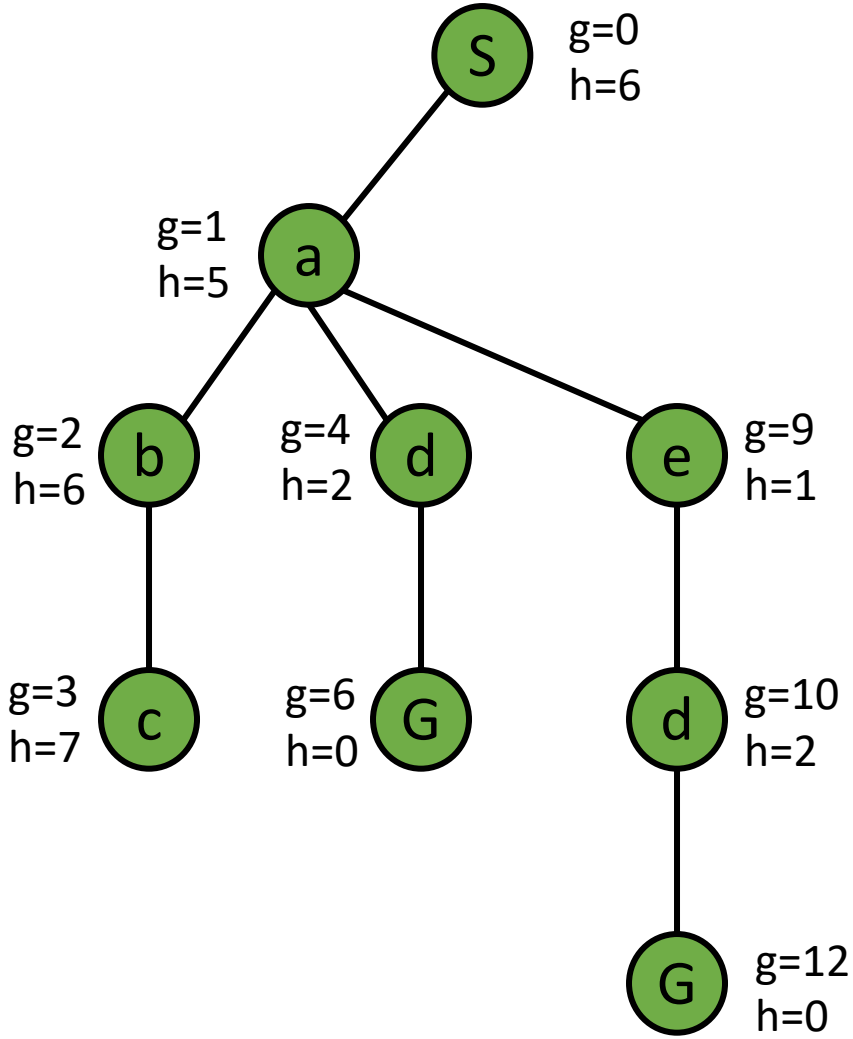
A*:

$f(n) = g(n) + h(n)$ (estimated total cost of cheapest solution through *n*)

A* Search

Example: Compare Uniform Cost, Greedy Search, and A* on the graph below.





A* Search

Consistent: for every node n and successor n’ of n, generated by some action a, the estimated cost of reaching the goal from n is no greater than the step cost from n to n’, plus the estimated cost of reaching the goal from n’

- That is: $h(n) \leq c(n, a, n') + h(n')$

- General **triangle inequality** between n, n’, and the goal

A heuristic h is **admissible** (optimistic) if $0 \leq h(n) \leq h^*(n)$, where $h^*(n)$ is the true cost to the nearest goal.

A* Search

Any consistent heuristic is also admissible (but not the other way around).

Example: Prove the above statement by induction.

A* Search

Optimality

Conditions for Optimality: Admissibility & Consistency

- $h(n)$ must be **admissible** - an admissible heuristic is one that never overestimates the cost to reach the goal.

- $h(n)$ is **consistent** if, for every node n and every successor n' of n generated by any action a, the estimated cost of reaching the goal from n is no greater than the step cost of getting to n' plus the estimated cost of reaching the goal from n':

$$h(n) \leq c(n, a, n') + h(n')$$

Optimality of A* Search

A* (graph) is optimal if the heuristic $h(n)$ is consistent.

Based on two key facts:

1. If $h(n)$ is consistent, then the values of $f(n)$ along any path are nondecreasing.

2. Whenever A* selects a node *n* for expansion, the optimal path to that node has been found.

- **So the first goal node to be expanded took the lowest-cost path, and all later goal node expansions are at least as expensive.**

Heuristics

How do we come up with heuristics?

1) Generate heuristics from relaxed problems.

2) Generate heuristics from sub-problems.

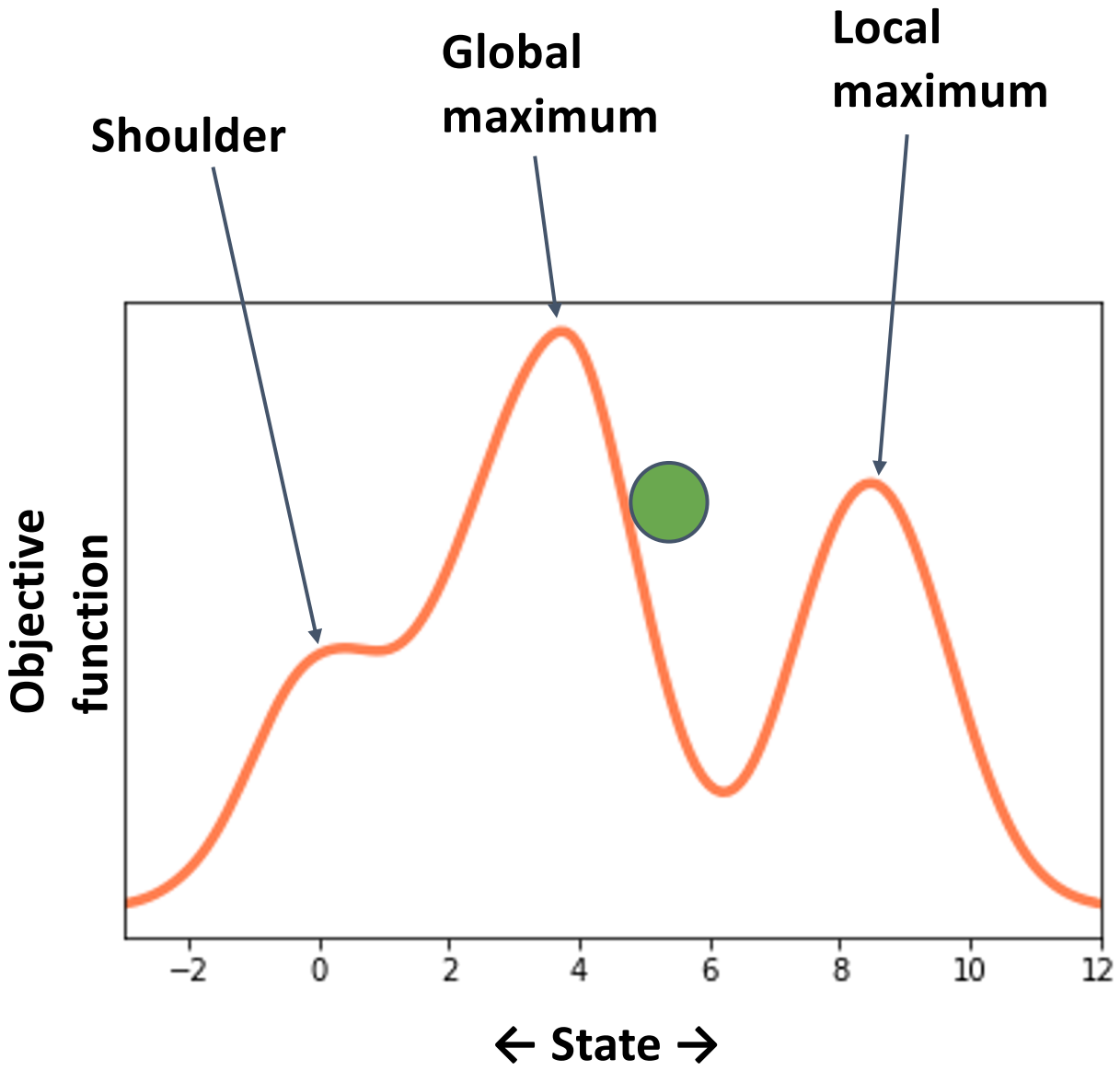
3) Learning heuristics from experience.

Hill Climbing

AKA: steepest ascent

Some words…

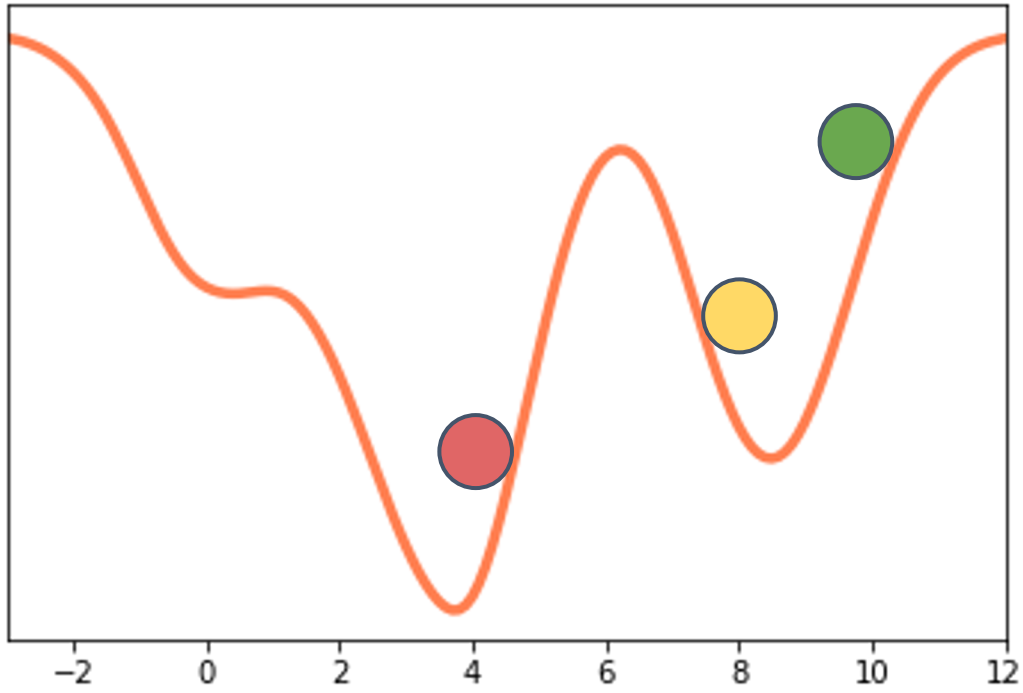
1) Start in some initial state
2) Get the neighbors of that state
3) Move to the neighbor with the best value of the objective function



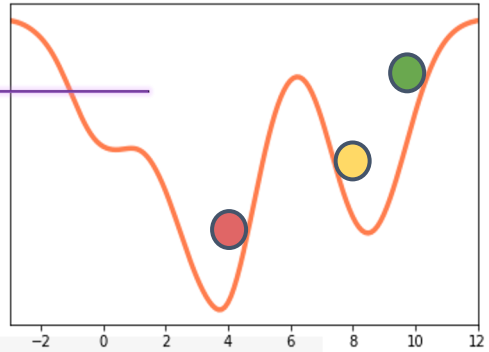
Simulated Annealing

Trying to find global **minimum** (in this case)

- Propose **random** moves
- Accept with a probability that depends on:
  1) "temperature"
  2) how "good" the move is
     - ➢ Accept move without question if it's better than our current situation
     - ➢ Accept move with some probability < 1 if it's worse



Simulated Annealing



Some pseudocode…

```
def schedule(time):
    '''some sort of mapping from time to temperature, to represent how we should be
    "cooling off" - that is, accepting wacky solutions with lower and lower probability'''
    # math goes here
    return temperature


def simulated_annealing(problem, some number of iterations):

    for t in some number of iterations:
        #1. update the "temperature", T(t) = schedule(time)
        #2. which moves can we make from the current node?
        #3. pick a random move
        #4. calculate difference in objective function between
        #   proposed new state and the current state (deltaE)
        #5. if proposed new state is better than the current state,
        #   then accept the proposed move with probability 1
        #6. otherwise...
        #         ACCEPT the move with probability exp(-deltaE/T(t)),
        #     or REJECT with prob 1-exp(-deltaE/T(t))
```

Random Restart Hill–Climbing

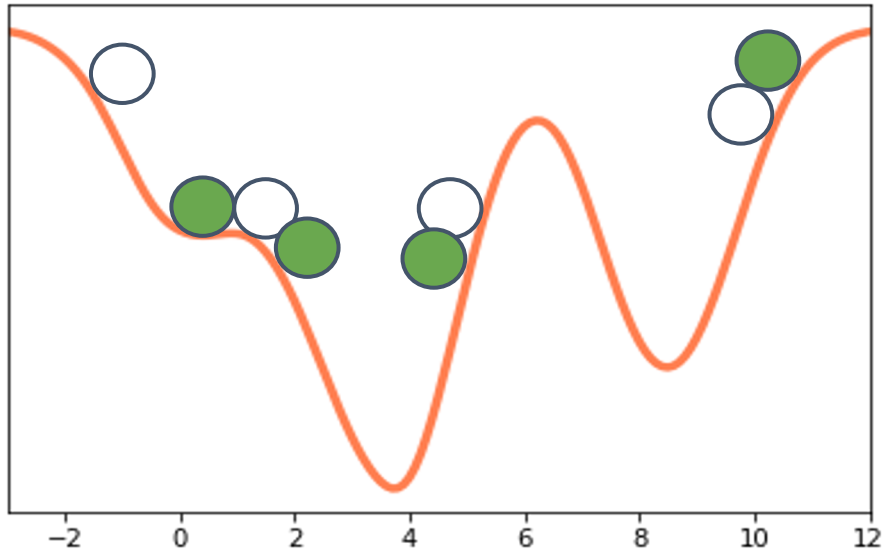
- Conduct a series of hill-climbing searches with a randomized starting point.
- If your random restart initial points end up all being close together, this method may not be any more effective than basic hill-climbing.
- You want to cover the space as much as you can.
- Cost is a constant multiple of hill-climbing - not much more expensive.

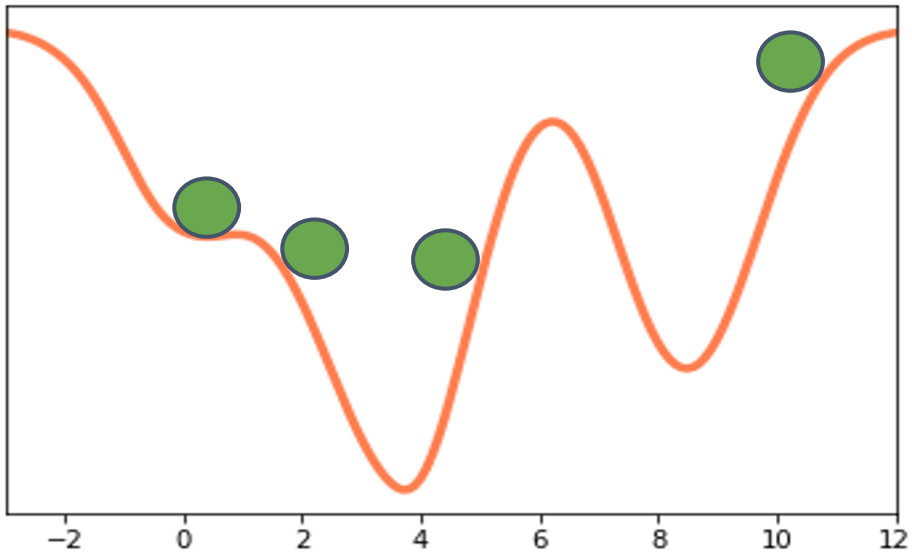
Local Beam Search

Fire off *k* states to explore

Each step:

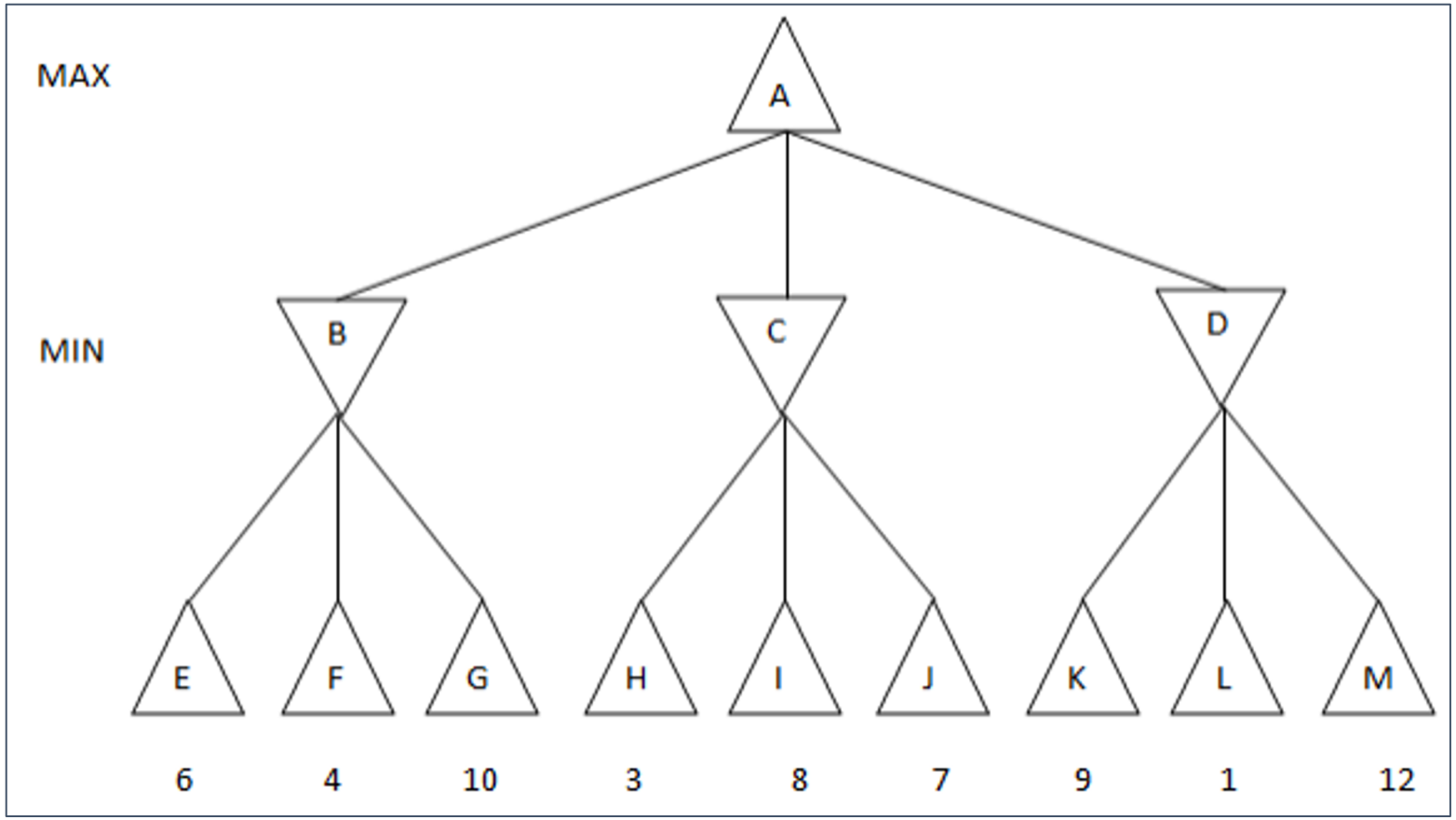
- Propose ***all*** successors to each of the *k* states
- If any of successors is goal -> finished
- Pick *k* of them to keep exploring in the next step
    - Can be **best** *k*, or **random** (possibly with some performance-weighting)
- Think of these as **generations** of states…





Minimax

- Two players: **Max** and **Min**
- Alternate making moves
- Optimal strategy from state *s* determined by **minimax value** of that state.



Utility to Max

Minimax

```
def minimax_decision(state):
    all_actions = what are the available actions?
    best_action = action that maximizes min_value(result(action, state))
    return best_action

def min_value(state):
    if terminal_state(state):
        return utility(state)
    value = infinity
    for action in all available actions:
        value = min(value, max_value(result(action, state)))
    return value

def max_value(state):
    if terminal_state(state):
        return utility(state)
    value = -infinity
    for action in all available actions:
        value = max(value, min_value(result(action, state)))
    return value
```

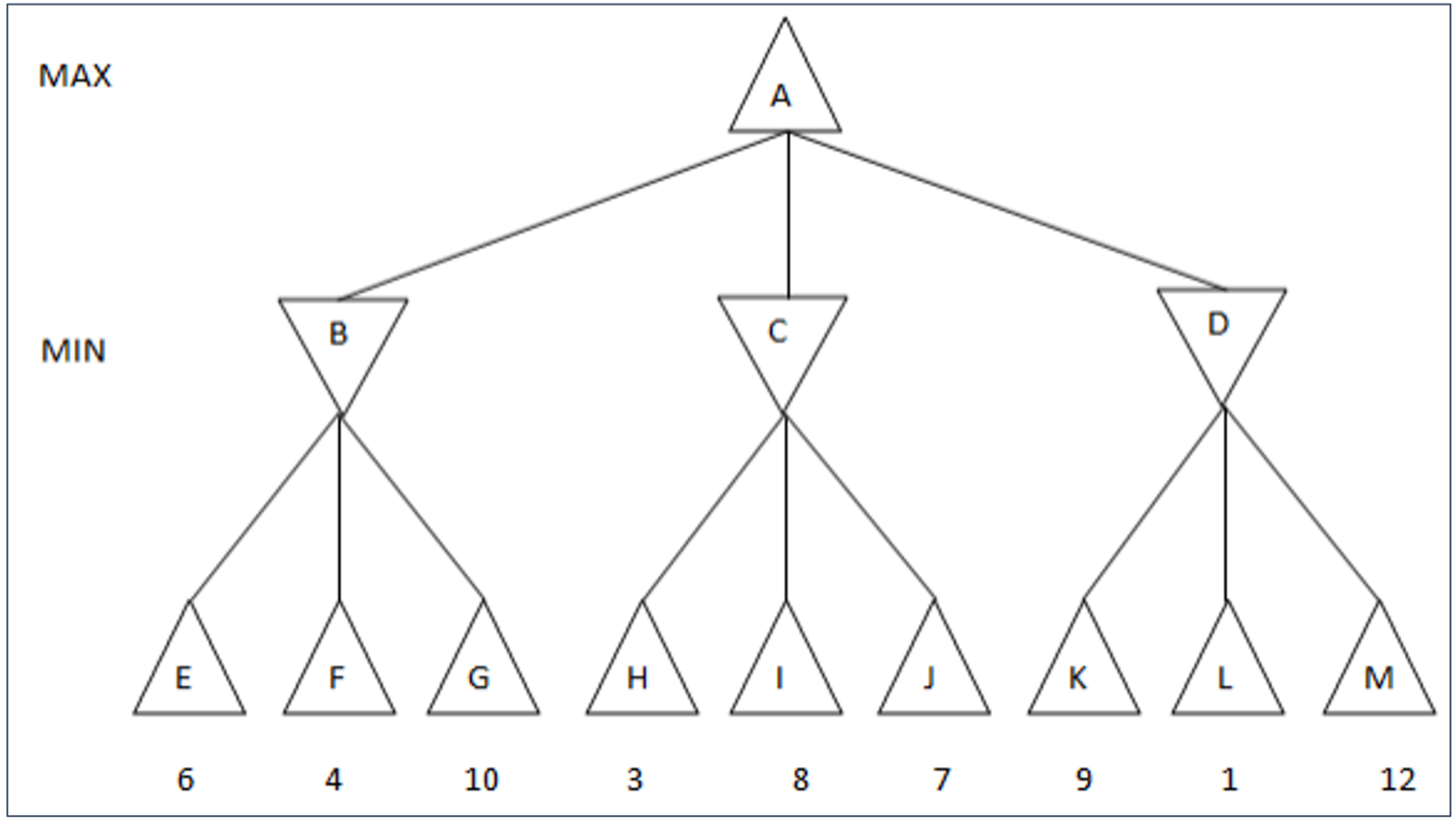
Alpha–Beta Pruning

Doing a full search on the game tree is generally intractable.

DFS time complexity: $O(b^m)$
- m = maximum depth
- b = branching factor

BUT if we assume optimal play, then there are some branches of the game tree that we don't need to explore

→ ***pruning*** those branches



Alpha–Beta Pruning

```
def alphabeta_search(state):
    alpha = -infinity
    beta  = +infinity
    value = max_value(state, alpha, beta)
    best_action = action that has utility=value to Max
    return

def max_value(state, alpha, beta):
    if terminal_state(state):
        return utility(state)
    value = -infinity
    for action in all available actions:
        value = max(value, min_value(result(action, state), alpha, beta))
        if value >= beta: return value
        alpha = max(value, alpha)
    return value

def min_value(state, alpha, beta):
    if terminal_state(state):
        return utility(state)
    value = +infinity
    for action in all available actions:
        value = min(value, max_value(result(action, state), alpha, beta))
        if value <= alpha: return value
        beta = min(value, beta)
    return value
```

Alpha–Beta Pruning

Example: Consider the Game Tree to below. Find the resulting value at the root node by following the minimax algorithm.