SWIFT
SHENZHEN UNIVERSITY WEBANK INSTITUTE OF FINTECH
深大微众金融科技学院

Chapter 2: Intelligent agents and problem solving

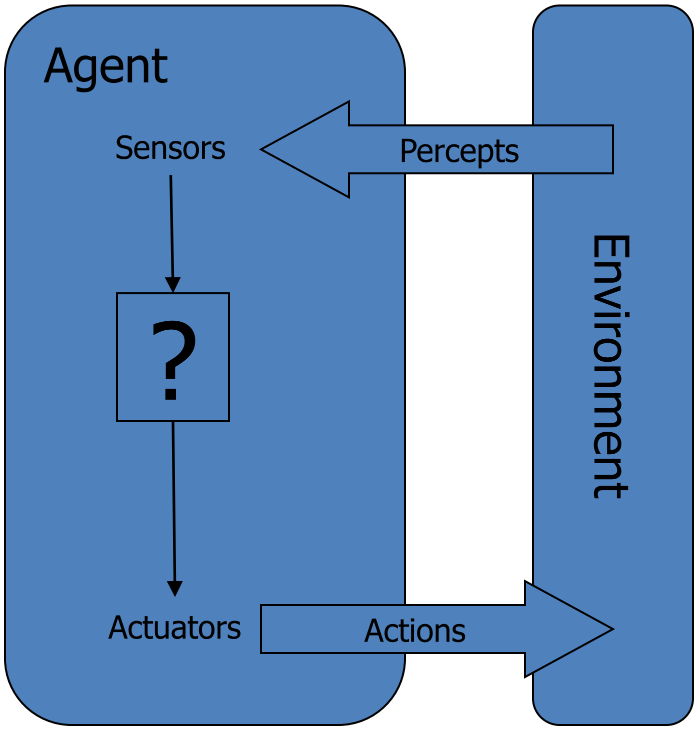
Artificial Intelligence and Machine Learning

Intelligent Agents

- ◇ Agents and environments
- ◇ Rationality
- ◇ PEAS (Performance measure, Environment, Actuators, Sensors)
- ◇ Environment types
- ◇ Agent types

Agents and Environments

- An **Agent** is anything that can be viewed as perceiving its **environment** through **sensors** and acting upon that environment through **actuators**



Agents and Environments

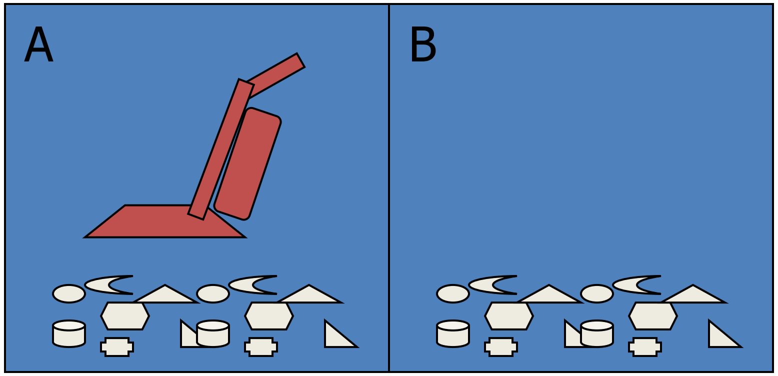
- Percept – the agent's perceptual inputs
  - percept sequence is a sequence of everything the agent has ever perceived

- Agent Function – describes the agent's behavior
  - Maps any given percept sequence to an action
  - f : P* -> A

- Agent Program – an implementation of an agent function for an artificial agent

Agents and Environments

- Example: Vacuum Cleaner World
  - Two locations: squares A and B
  - Perceives what square it is in
  - Perceives if there is dirt in the current square
  - Actions
    - move left
    - move right
    - suck up the dirt
    - do nothing



Agents and Environments

- Agent Function: Vacuum Cleaner World
  – If the current square is dirty, then suck, otherwise move to the other square

Percept Sequence	**Action**
[A, Clean]	Right
[A, Dirty]	Suck
[B, Clean]	Left
[B, Dirty]	Suck
[A, Clean], [A, Clean]	Right
[A, Clean], [A, Dirty]	Suck

Agents and Environments

- But what is the right way to fill out the table?
  - is the agent
    - good or bad
    - intelligent or stupid
  - can it be implemented in a small program?

```
Function Reflex-Vacuum-Agent([location, status]) return an action
    if status == Dirty then return Suck
    else if location = A then return Right
    else if location = B then return Left
```

Good Behavior and Rationality

- Rational Agent – an agent that does the “right” thing
  - Every entry in the table for the agent function is filled out correctly
  - Doing the right thing is better than doing the wrong thing
    - What does it mean to do the right thing?

Good Behavior and Rationality

- Performance Measure
  - A scoring function for evaluating the environment space

- ***<u>Rational Agent</u>*** – for each possible percept sequence, a rational agent should select an action that is expected to maximize its performance measure, given the evidence provided by the percept sequence and what ever built-in knowledge the agent has.

Good Behavior and Rationality

- Rational != omniscient
- Rational != clairvoyant
- Rational != successful

- Rational -> exploration, learning, autonomy

The Nature of Environments

- Task environments
  - The “problems” to which a rational agent is the “solution”
- PEAS
  - Performance
  - Environment
  - Actuators
  - Sensors

The Nature of Environments

- Properties of task environments
  - Fully Observable vs. Partially Observable
  - Deterministic vs. Stochastic
  - Episodic vs. Sequential
  - Static vs. Dynamic
  - Discrete vs. Continuous
  - Single agent vs. Multi-agent

- The real world is partially observable, stochastic, sequential, dynamic, continuous, multi-agent

The Nature of Environments

- Agent Examples
  - Automated Taxi
  - Mars Rover
  - Trader
  - Chess

The Structure of Agents

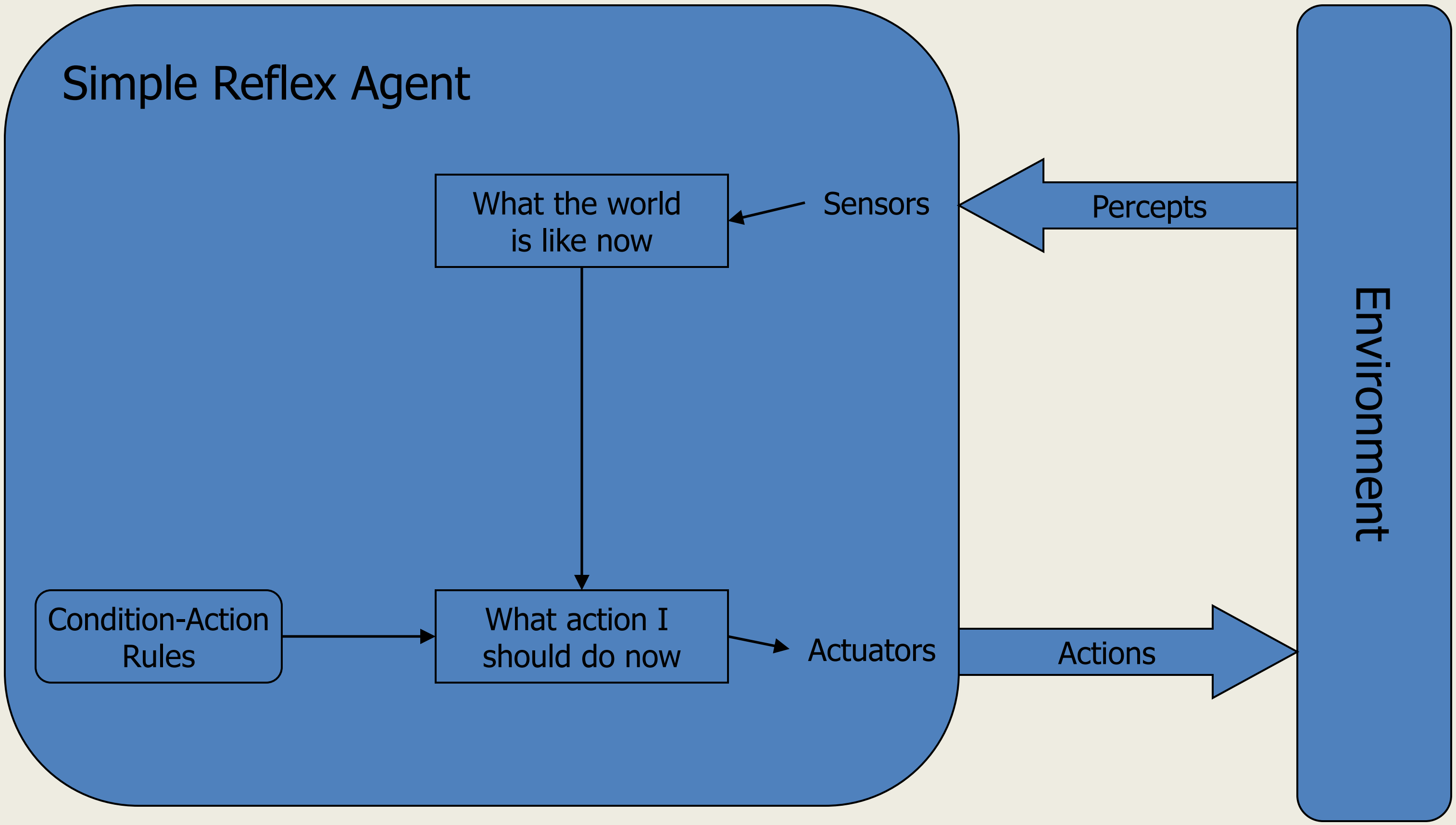
- Agent = Architecture + Program
- Basic algorithm for a rational agent
  - While (true) do
    - Get percept from sensors into memory
    - Determine best action based on memory
    - Record action in memory
    - Perform action
- Most AI programs are a variation of this theme

The Structure of Agents

- Table Driven Agent

```
function Table-Driven-Agent (percept) return action
static:     percepts, a sequence, initially empty
            table, a table of actions, indexed by
            percept sequences, initially fully
            specified

  append percept to the end of the table
  action <- LOOKUP( percept, table )
  return action
```

The Structure of Agents

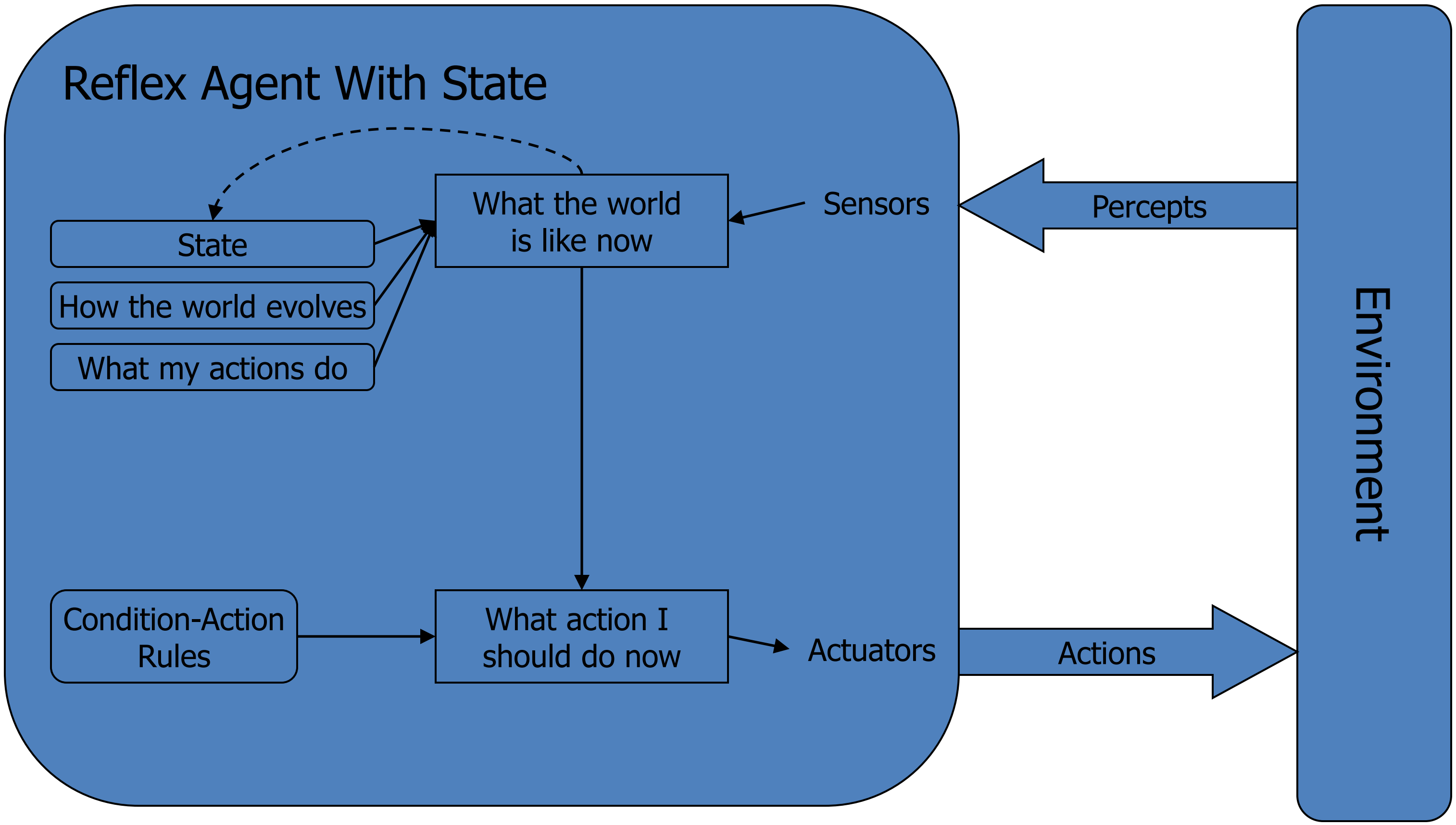


The Structure of Agents

- Simple Reflex Agent

```
function Simple-Reflex-Agent (percept) return action
static:     rules, a set of condition-action rules

  state <- INTERPRET-INPUT( percept )
  rule <- RULE-MATCH( state, rules )
  action <- RULE-ACTION[ rule ]
  return action
```

The Structure of Agents

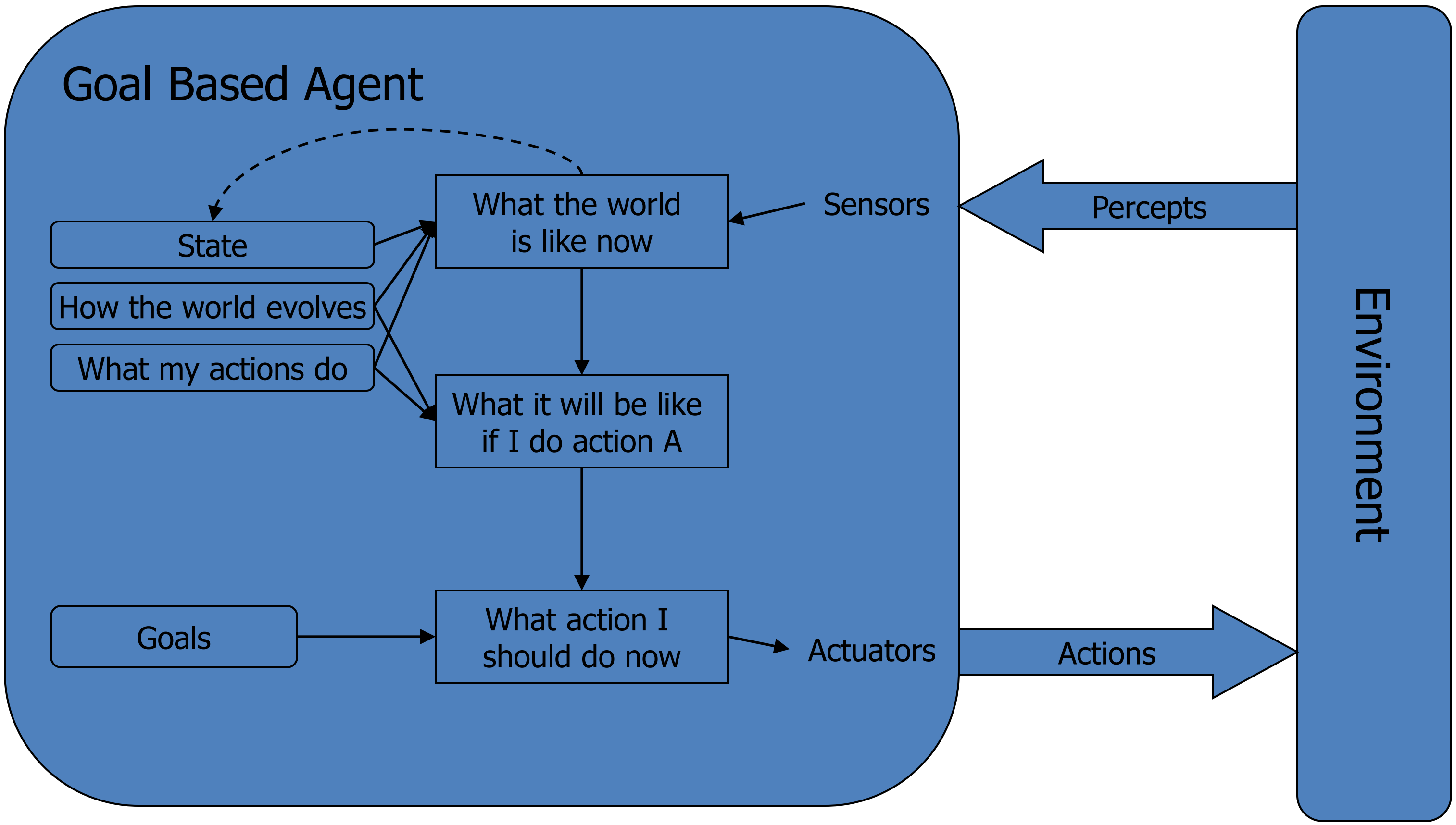


The Structure of Agents

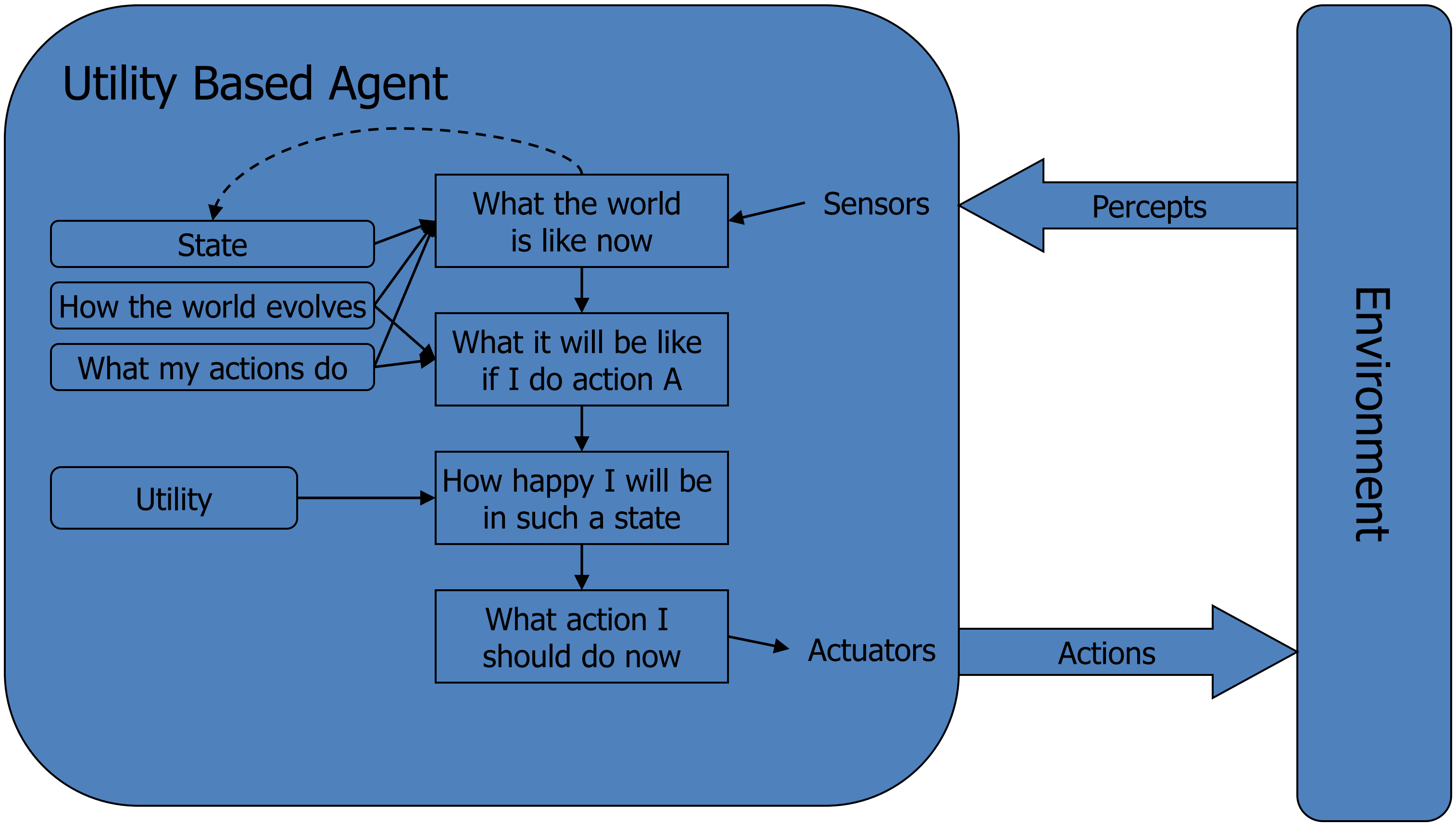
- Reflex Agent With State

```
function Reflex-Agent-With-State (percept) return action
static: state, a description of the current world state
        rules, a set of condition-action rules
        action, the most recent action, initially none

  state <- UPDATE-STATE( state, action, percept )
  rule <- RULE-MATCH( state, rules )
  action <- RULE-ACTION[ rule ]
  return action
```

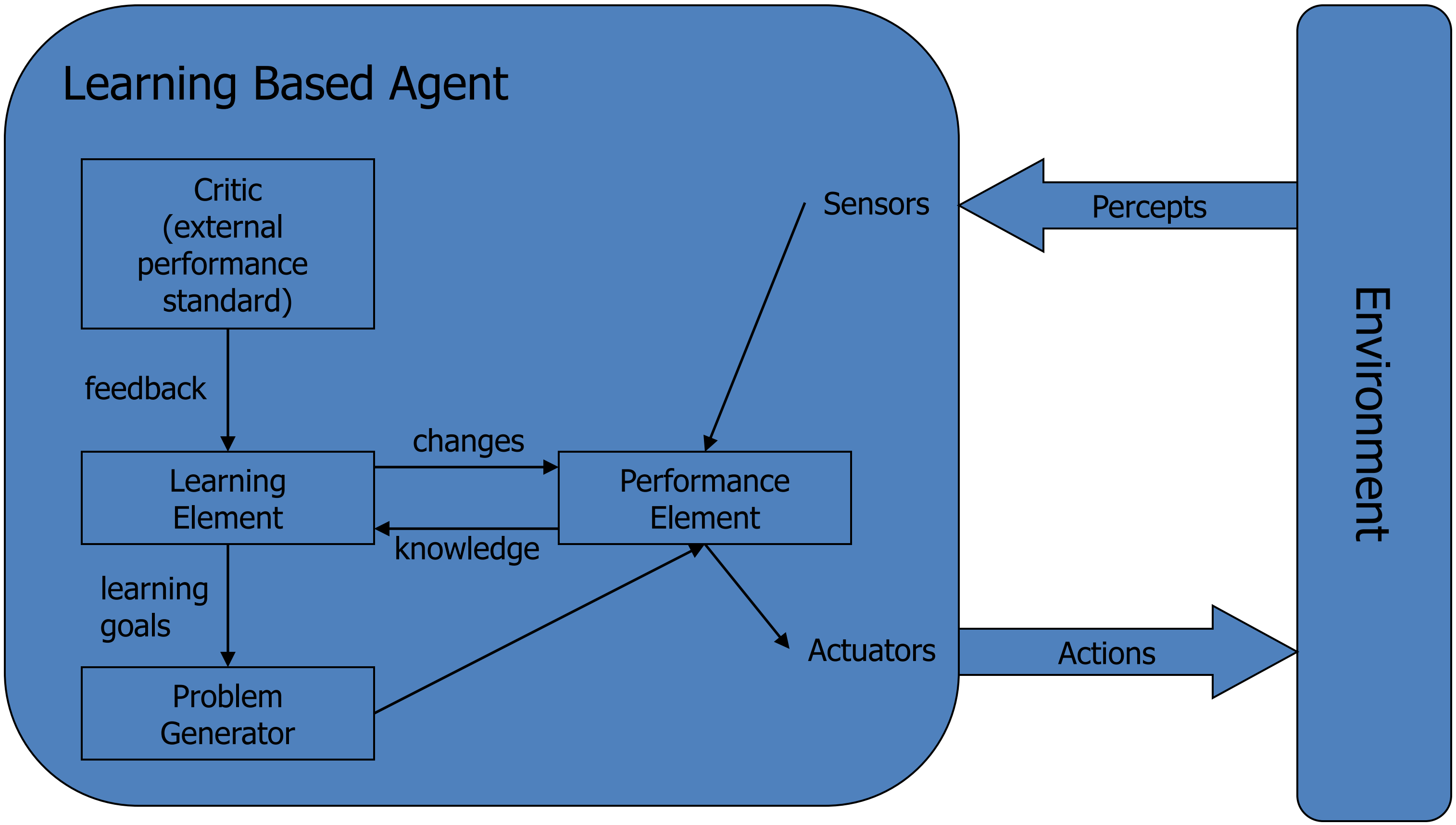
The Structure of Agents



The Structure of Agents



The Structure of Agents



Problem Solving and Search

Problem solving

Problem types

Problem formulation

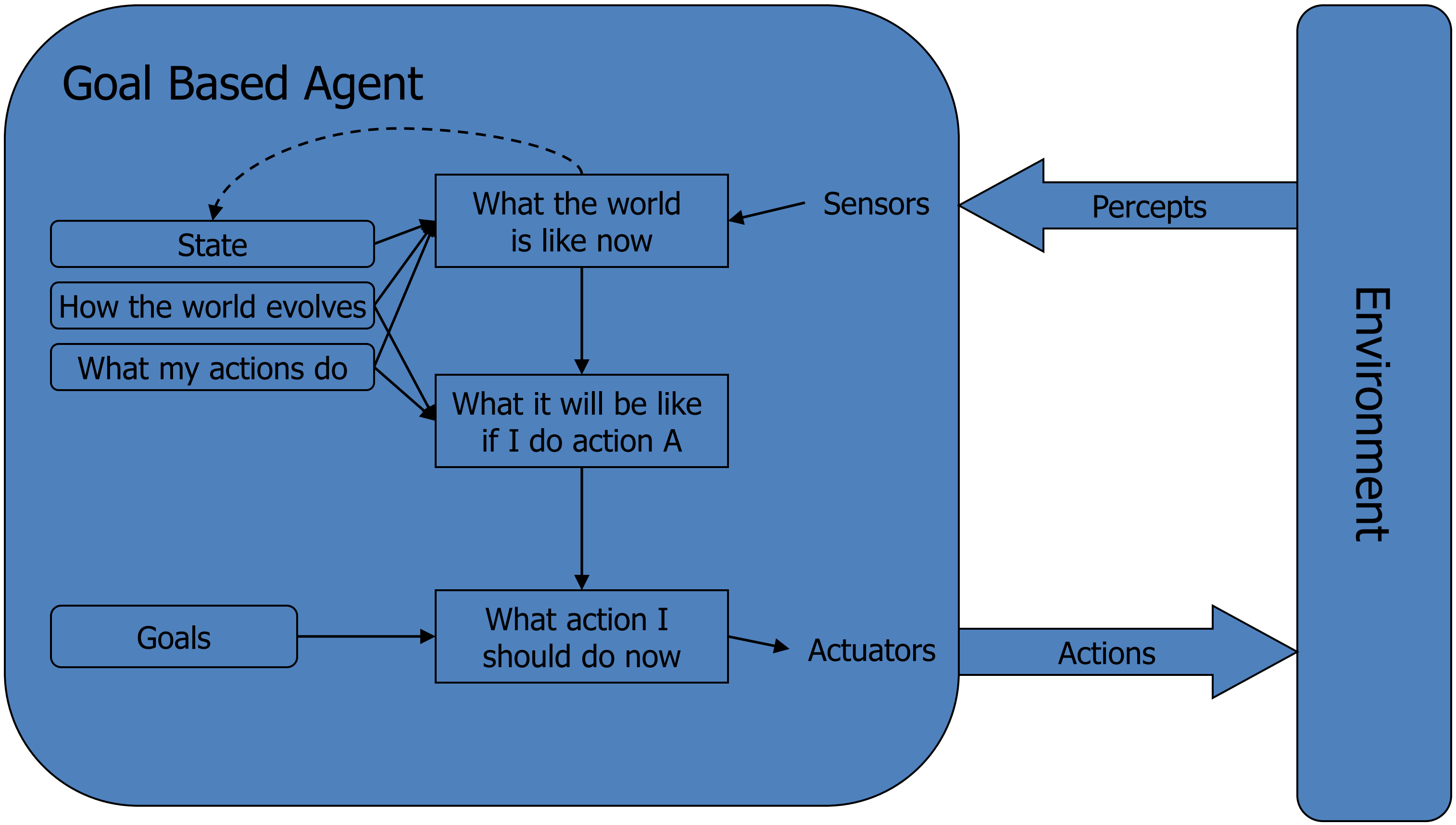
Example problems

Basic search algorithms

Problem Solving Agents

- Problem solving agent
  - A kind of “goal based” agent
  - Finds <u>sequences of actions</u> that lead to desirable states.
- The algorithms are uninformed
  - No extra information about the problem other than the definition
    - No extra information
    - No heuristics (rules)

Goal Based Agent



Goal Based Agent

```
Function Simple-Problem-Solving-Agent( percept ) returns action

Inputs: percept         a percept
Static: seq             an action sequence initially empty
        state           some description of the current world
        goal            a goal, initially null
        problem         a problem formulation

state <- UPDATE-STATE( state, percept )
if seq is empty then do
   goal <- FORMULATE-GOAL( state )
   problem <- FORMULATE-PROBLEM( state, goal )
   seq <- SEARCH( problem )                              # SEARCH
action <- RECOMMENDATION ( seq )                         # SOLUTION
seq <- REMAINDER( seq )
return action                                            # EXECUTION
```

Goal Based Agents

- Assumes the problem environment is:
  - Static
    - The plan remains the same
  - Observable
    - Agent knows the initial state
  - Discrete
    - Agent can enumerate the choices
  - Deterministic
    - Agent can plan a sequence of actions such that each will lead to an intermediate state

- The agent carries out its plans with its eyes closed
  - Certain of what's going on
  - Open loop system

Well Defined Problems and Solutions

- A problem
  - Initial state
  - Actions and Successor Function
  - Goal test
  - Path cost

Example: Water Pouring

- Given a 4 gallon bucket and a 3 gallon bucket, how can we measure exactly 2 gallons into one bucket?
  - There are no markings on the bucket
  - You must fill each bucket completely

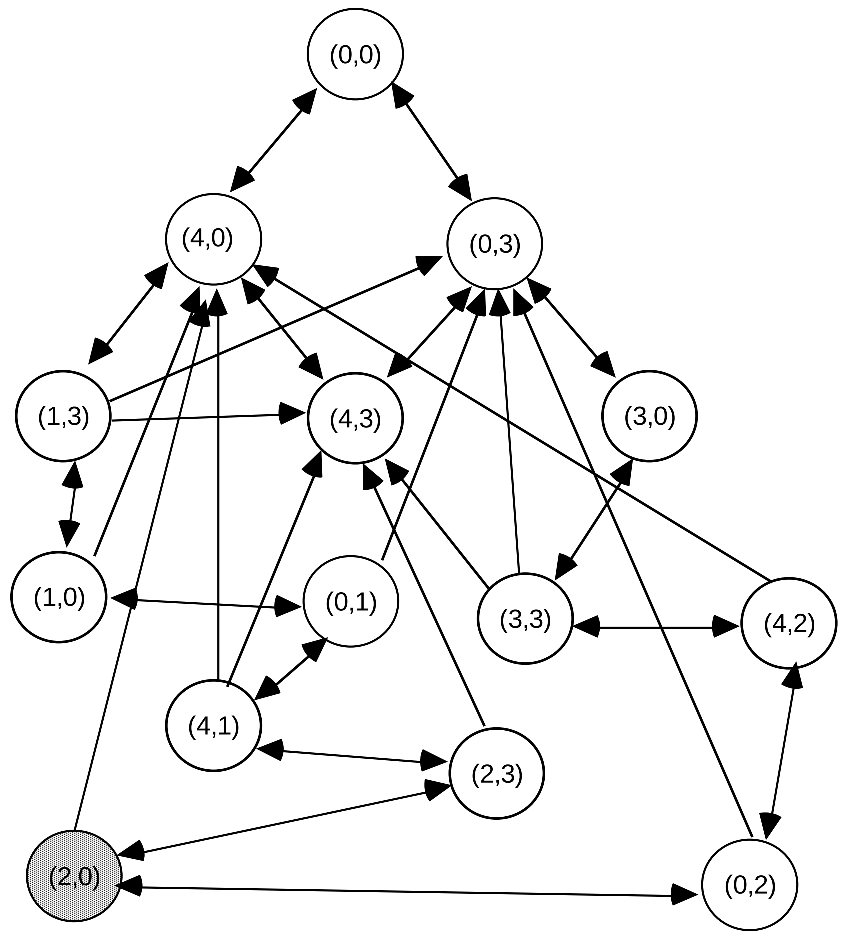
Example: Water Pouring

- Initial state:
  - The buckets are empty
  - Represented by the tuple (0 0)
- Goal state:
  - One of the buckets has two gallons of water in it
  - Represented by either (x 2) or (2 x)
- Path cost:
  - 1 per unit step

Example: Water Pouring

- Actions and Successor Function
  - Fill a bucket
    - (x y) -> (3 y)
    - (x y) -> (x 4)
  - Empty a bucket
    - (x y) -> (0 y)
    - (x y) -> (x 0)
  - Pour contents of one bucket into another
    - (x y) -> (0 x+y) or (x+y-4, 4)
    - (x y) -> (x+y 0) or (3, x+y-3)

Example: Water Pouring



Example: Eight Puzzle

- States:
  - Description of the eight tiles and location of the blank tile
- Successor Function:
  - Generates the legal states from trying the four actions {*Left, Right, Up, Down*}
- Goal Test:
  - Checks whether the state matches the goal configuration
- Path Cost:
  - Each step costs 1

7	2	4
5		6
8	3	1

1	2	3
4	5	6
7	8	

Example: Eight Puzzle

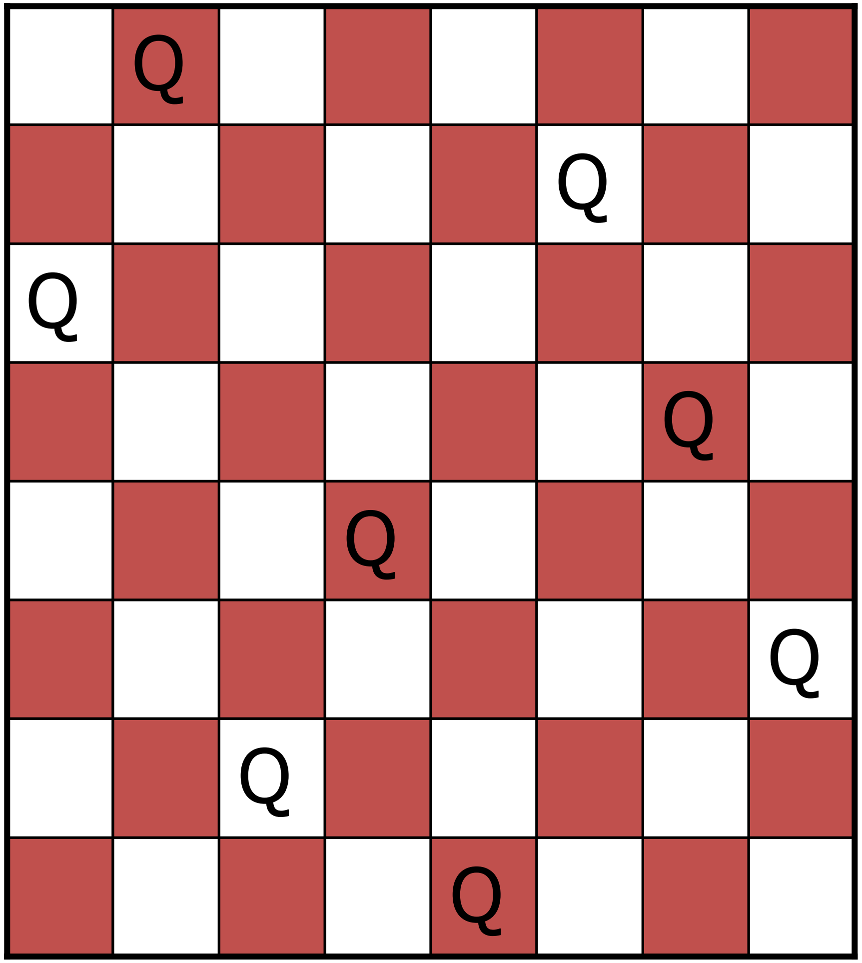
- Eight puzzle is from a family of “sliding –block puzzles”
  - NP Complete
  - 8 puzzle has 9!/2 = 181440 states
  - 15 puzzle has approx. $1.3*10^{12}$ states
  - 24 puzzle has approx. $1*10^{25}$ states

Example: Eight Queens

- Place eight queens on a chess board such that no queen can attack another queen

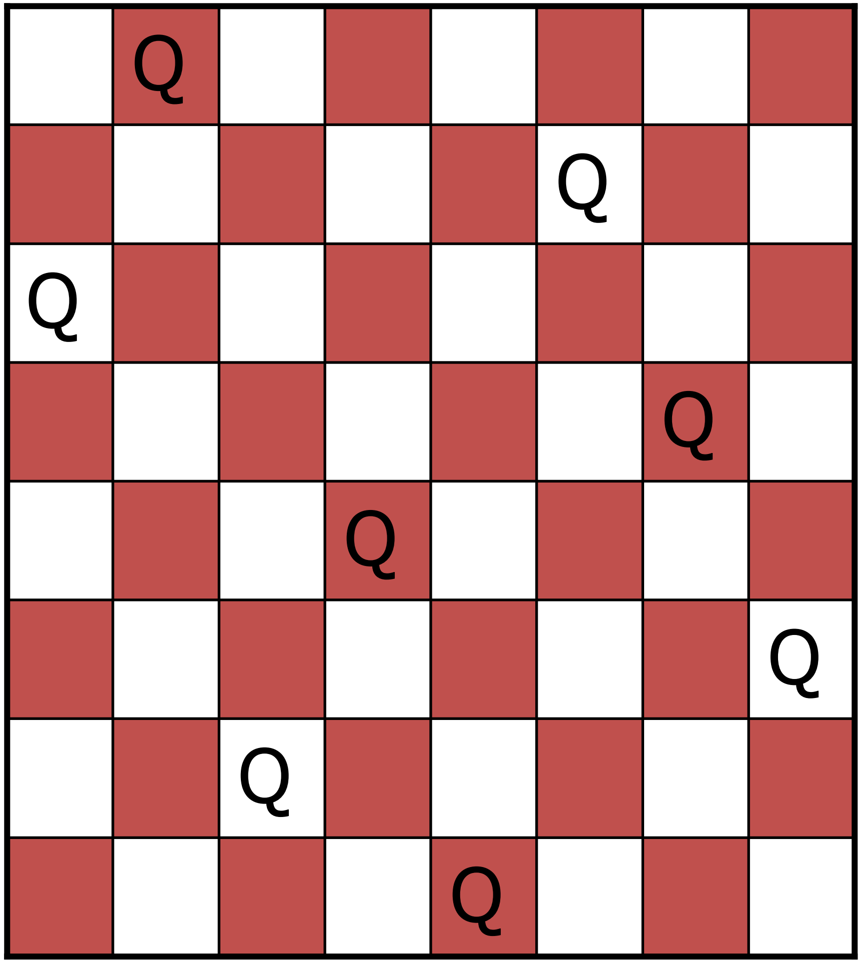
- No path cost because only the final state counts!

- Incremental formulations
- Complete state formulations



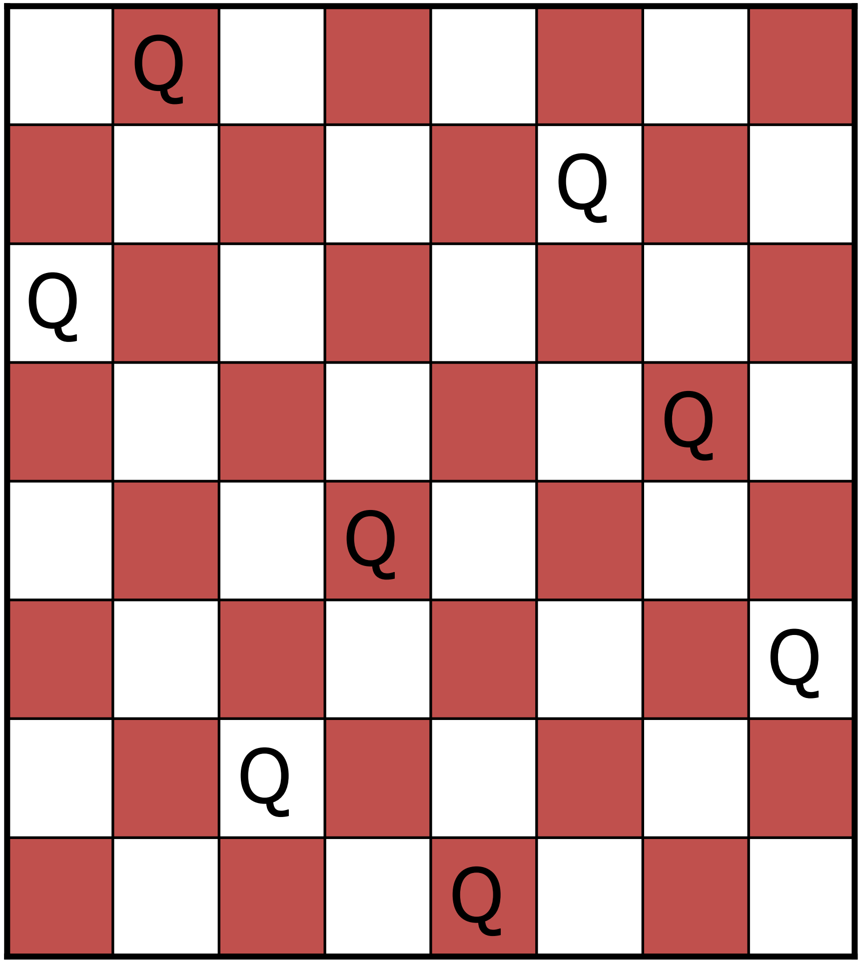
Example: Eight Queens

- States:
  - Any arrangement of 0 to 8 queens on the board
- Initial state:
  - No queens on the board
- Successor function:
  - Add a queen to an empty square
- Goal Test:
  - 8 queens on the board and none are attacked
- $64*63*\ldots*57 = 1.8*10^{14}$ possible sequences
  - Ouch!

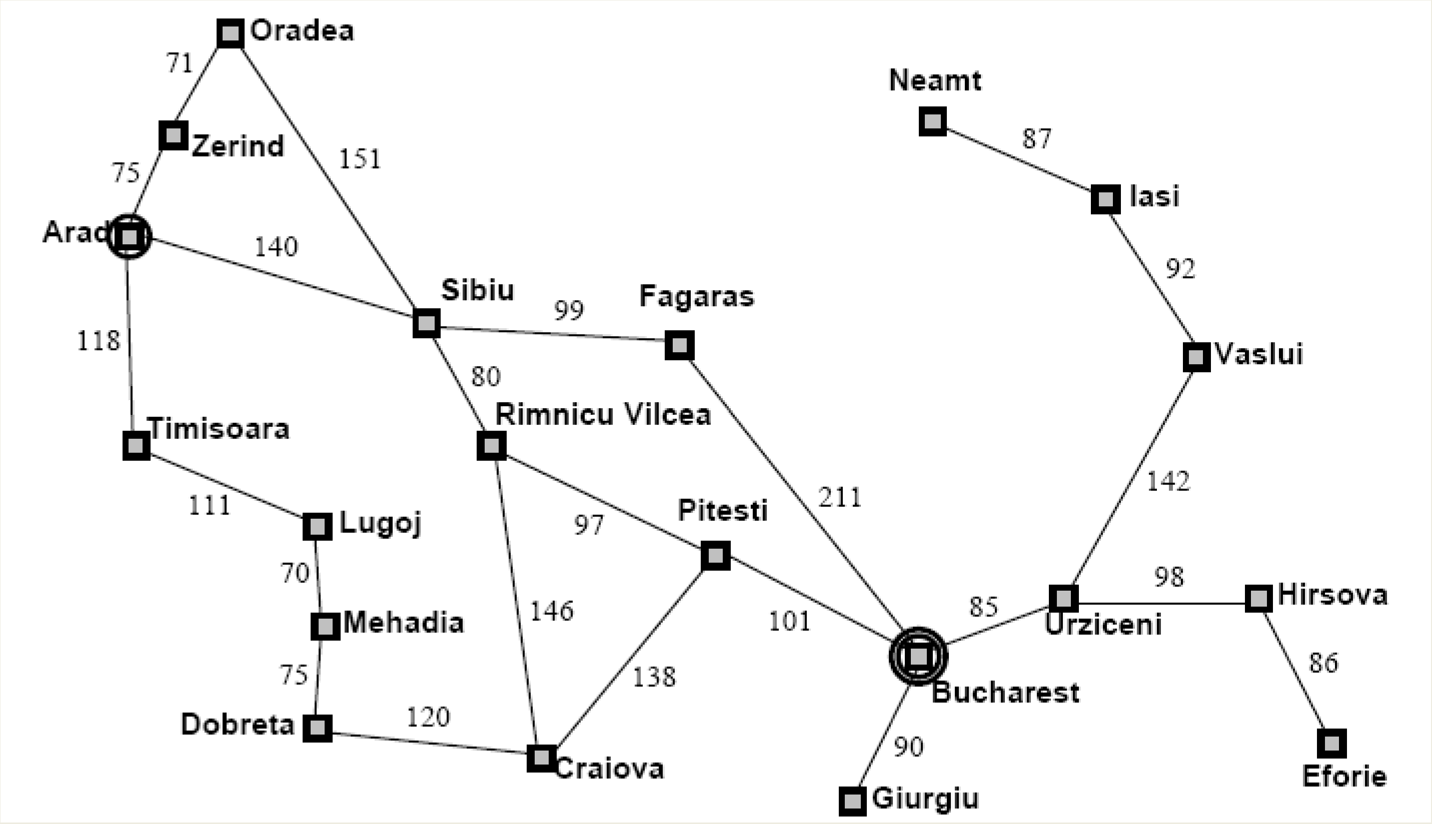


Example: Eight Queens

- States:
  - Arrangements of n queens, one per column in the leftmost n columns, with no queen attacking another are states
- Successor function:
  - Add a queen to any square in the leftmost empty column such that it is not attacked by any other queen.
- 2057 sequences to investigate



Example: Map Planning



Searching For Solutions

- Initial State
  - e.g. “At Arad”
- Successor Function
  - A set of action state pairs
  - S(Arad) = {(Arad->Zerind, Zerind), …}
- Goal Test
  - e.g. x = “at Bucharest”
- Path Cost
  - sum of the distances traveled

Searching For Solutions

- Having formulated some problems…how do we solve them?
- Search through a state space
- Use a search tree that is generated with an initial state and successor functions that define the state space

Searching For Solutions

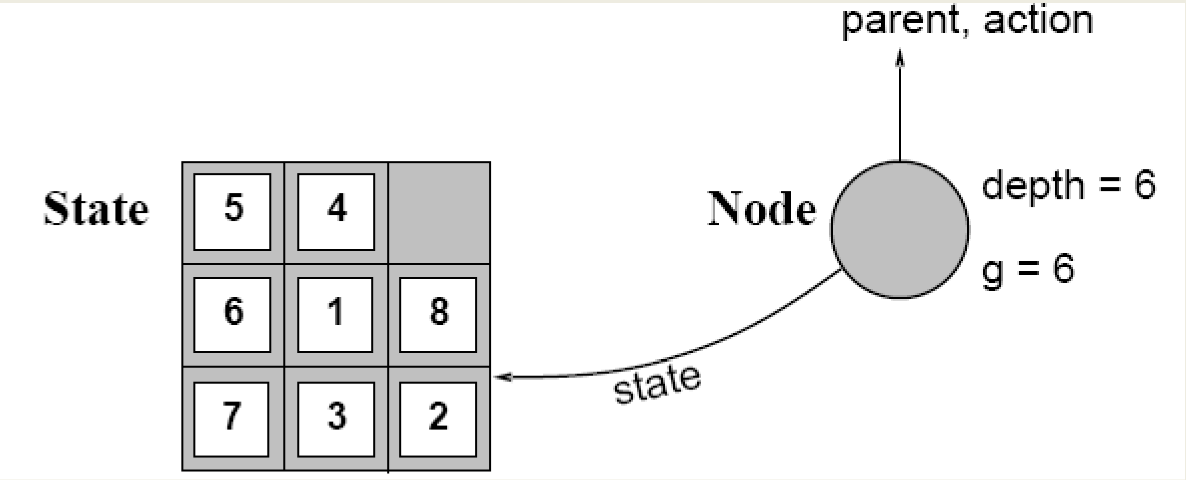
- A ***state*** is (a representation of) a physical configuration

- A ***node*** is a data structure constituting part of a search tree
  - Includes parent, children, depth, path cost

- States do not have children, depth, or path cost

- The EXPAND function creates new nodes, filling in the various fields and using the SUCCESSOR function of the problem to create the corresponding states

Searching For Solutions



Searching For Solutions

```
function TREE-SEARCH( problem, strategy) returns a solution, or failure
    initialize the search tree using the initial state of problem
    loop do
        if there are no candidates for expansion then return failure
        choose a leaf node for expansion according to strategy
        if the node contains a goal state then return the corresponding solution
        else expand the node and add the resulting nodes to the search tree
    end
```

Searching For Solutions

```
function TREE-SEARCH(problem, fringe) returns a solution, or failure
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe)
        if GOAL-TEST[problem] applied to STATE(node) succeeds return node
        fringe ← INSERTALL(EXPAND(node, problem), fringe)
```

```
function EXPAND(node, problem) returns a set of nodes
    successors ← the empty set
    for each action, result in SUCCESSOR-FN[problem](STATE[node]) do
        s ← a new NODE
        PARENT-NODE[s] ← node;  ACTION[s] ← action;  STATE[s] ← result
        PATH-COST[s] ← PATH-COST[node] + STEP-COST(node, action, s)
        DEPTH[s] ← DEPTH[node] + 1
        add s to successors
    return successors
```

Uninformed Search Strategies

- ***Uninformed*** strategies use only the information available in the problem definition
  - Also known as blind searching

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

Comparing Uninformed Search Strategies

- Completeness
  - Will a solution always be found if one exists?
- Time
  - How long does it take to find the solution?
  - Often represented as the number of nodes searched
- Space
  - How much memory is needed to perform the search?
  - Often represented as the maximum number of nodes stored at once
- Optimal
  - Will the optimal (least cost) solution be found?

- Page 81 in AIMA text

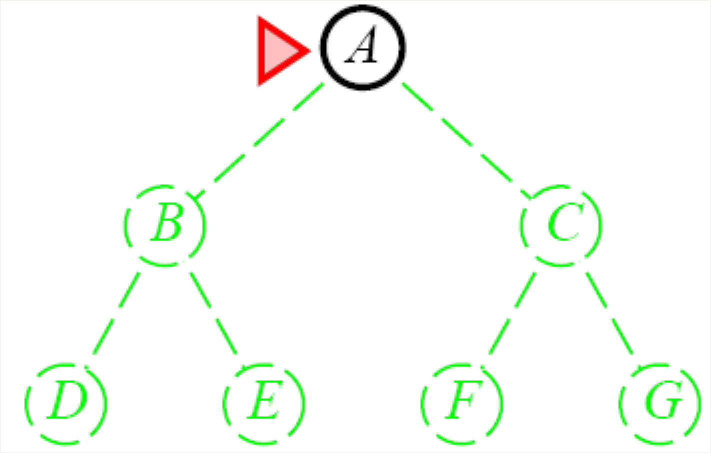
Comparing Uninformed Search Strategies

- Time and space complexity are measured in
  - b – maximum branching factor of the search tree
  - m – maximum depth of the state space
  - d – depth of the least cost solution

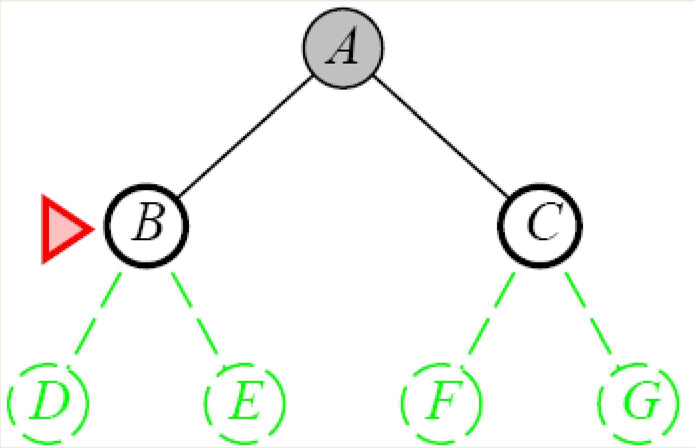
Breadth-First Search

- Recall from Data Structures the basic algorithm for a breadth-first search on a graph or tree
- Expand the ***shallowest*** unexpanded node
- Place all new successors at the end of a FIFO queue

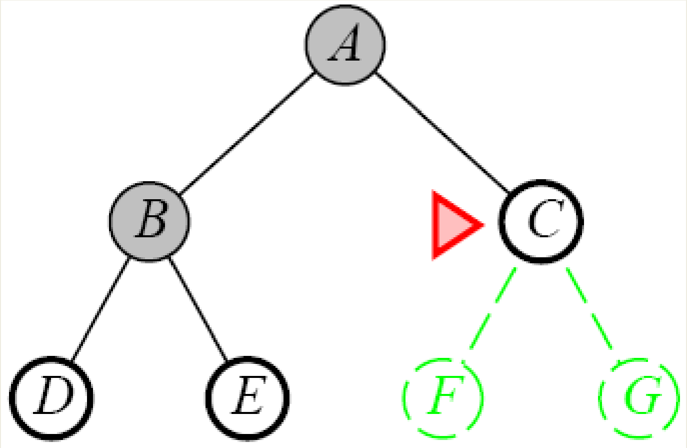
Breadth-First Search



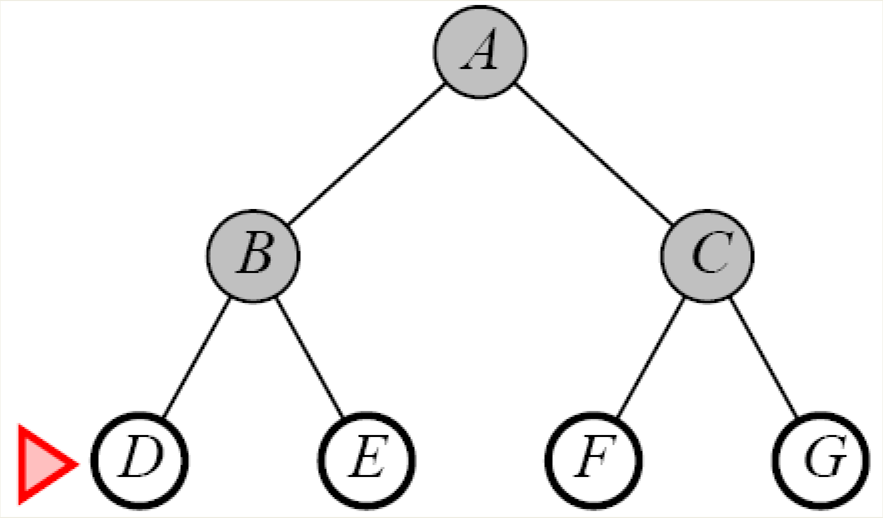
Breadth-First Search



Breadth-First Search



Breadth-First Search



Properties of Breadth-First Search

- Complete
  - Yes if b (max branching factor) is finite
- Time
  - $1 + b + b^2 + \ldots + b^d + b(b^d\text{-}1) = O(b^{d+1})$
  - exponential in d
- Space
  - $O(b^{d+1})$
  - Keeps every node in memory
  - This is the big problem; an agent that generates nodes at 10 MB/sec will produce 860 MB in 24 hours
- Optimal
  - Yes (if cost is 1 per step); not optimal in general

Lessons From Breadth First Search

- The memory requirements are a bigger problem for breadth-first search than is execution time

- Exponential-complexity search problems cannot be solved by uniformed methods for any but the smallest instances

Uniform-Cost Search

- Same idea as the algorithm for breadth-first search…but…
  - Expand the ***least-cost*** unexpanded node
  - FIFO queue is ordered by cost
  - Equivalent to regular breadth-first search if all step costs are equal

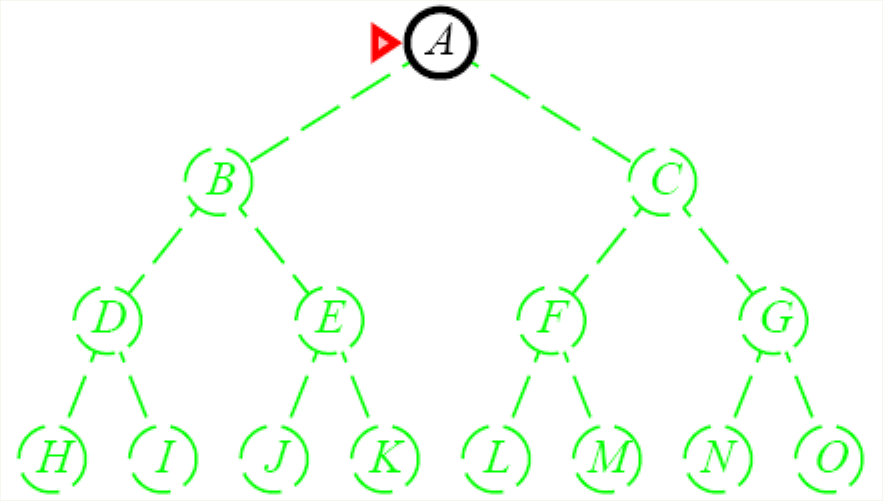
Uniform-Cost Search

- Complete
  - Yes if the cost is greater than some threshold
  - step cost >= ε
- Time
  - Complexity cannot be determined easily by d or d
  - Let C* be the cost of the optimal solution
  - $O(b^{ceil(C^*/\varepsilon)})$
- Space
  - $O(b^{ceil(C^*/\varepsilon)})$
- Optimal
  - Yes, Nodes are expanded in increasing order

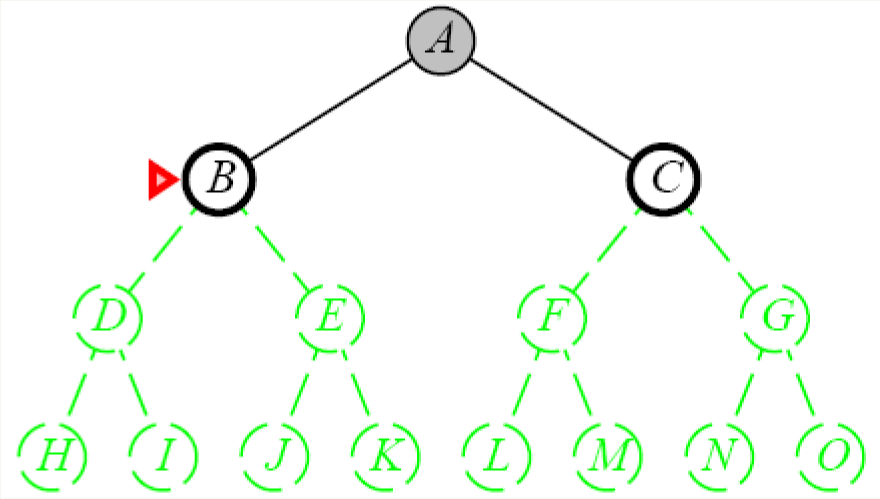
Depth-First Search

- Recall from Data Structures the basic algorithm for a depth-first search on a graph or tree
- Expand the ***<u>deepest</u>*** unexpanded node
- Unexplored successors are placed on a stack until fully explored

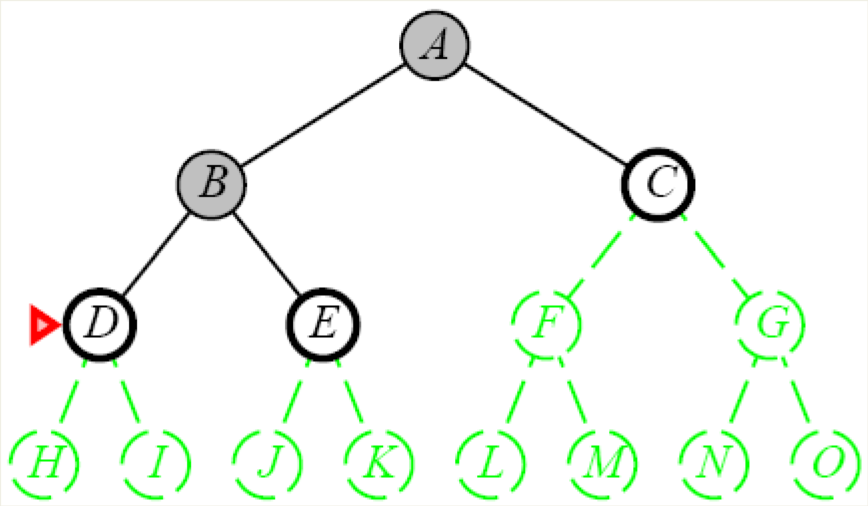
Depth-First Search



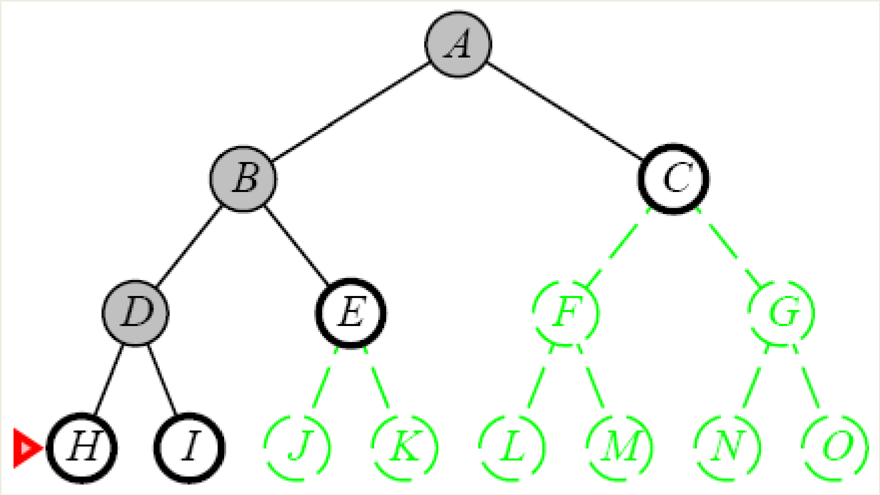
Depth-First Search



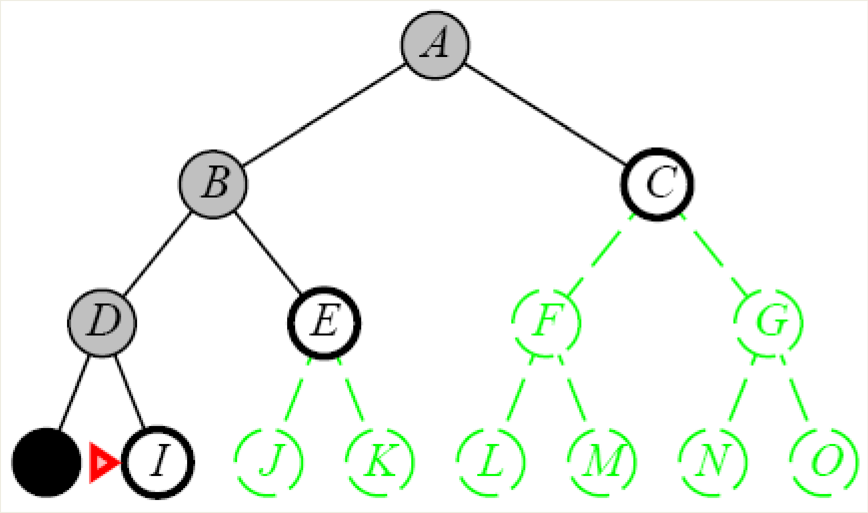
Depth-First Search



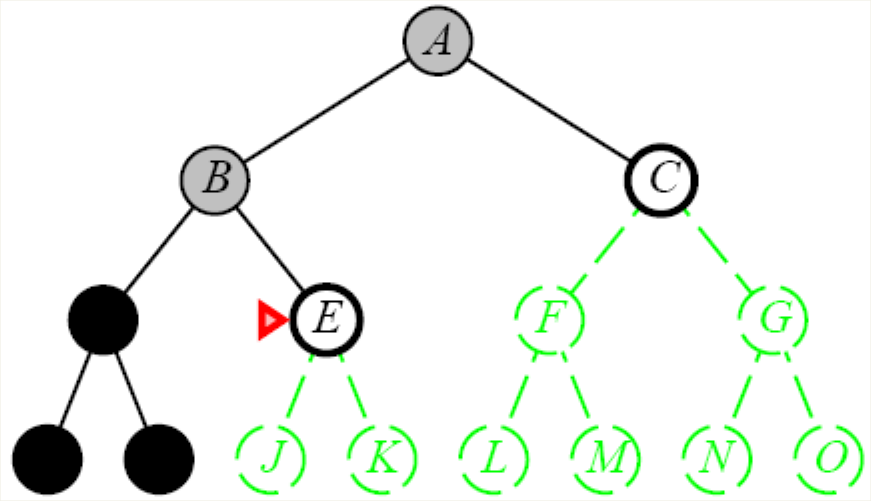
Depth-First Search



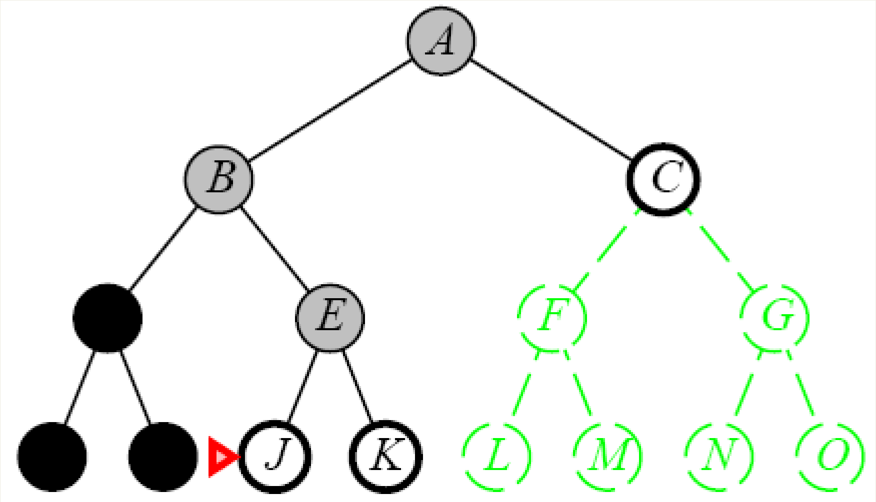
Depth-First Search



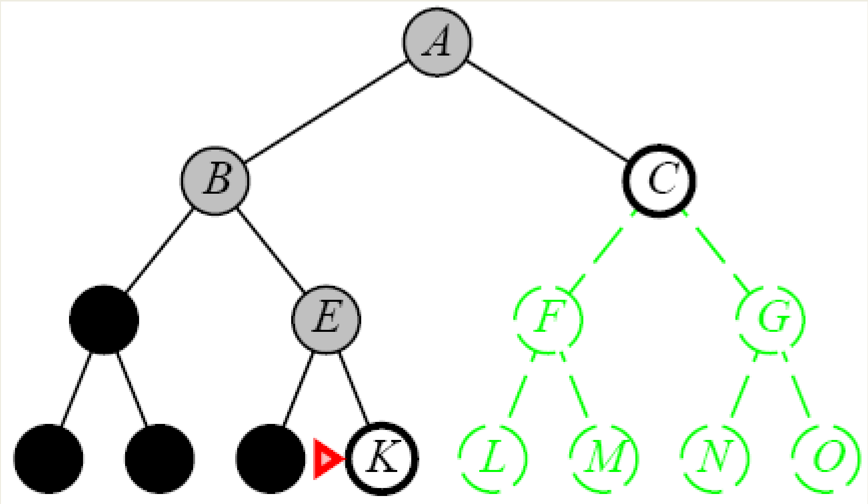
Depth-First Search



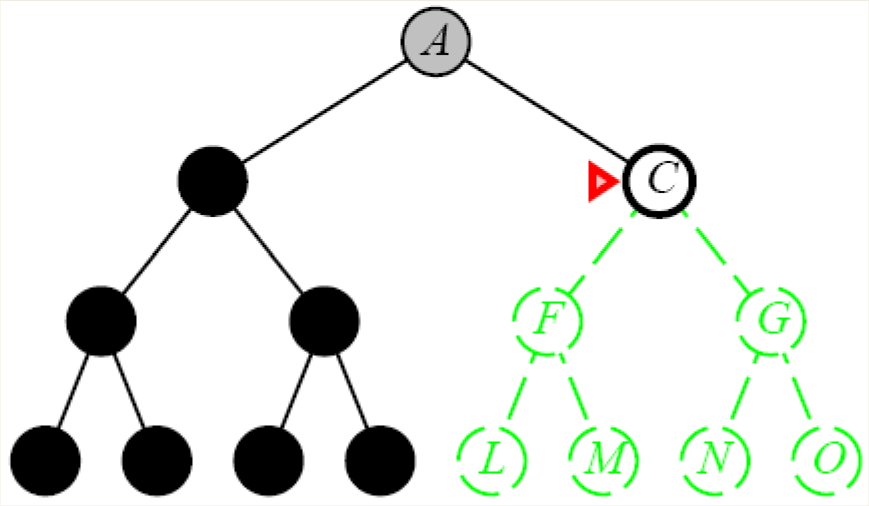
Depth-First Search



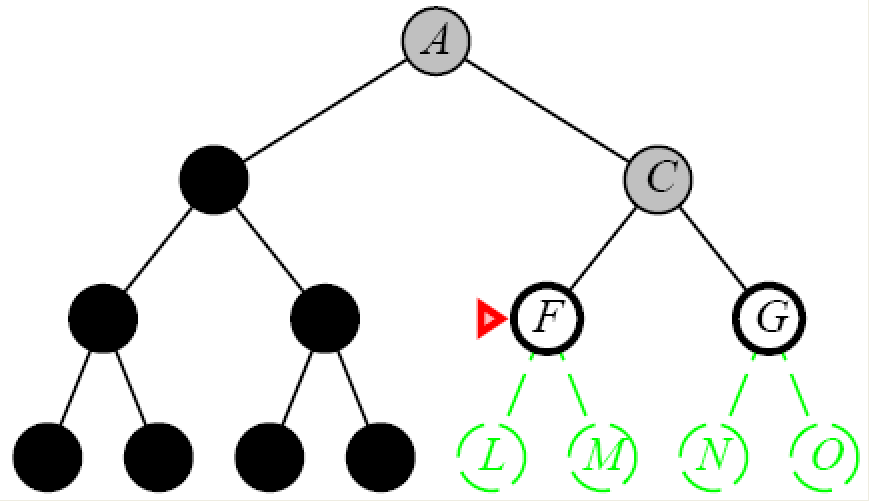
Depth-First Search



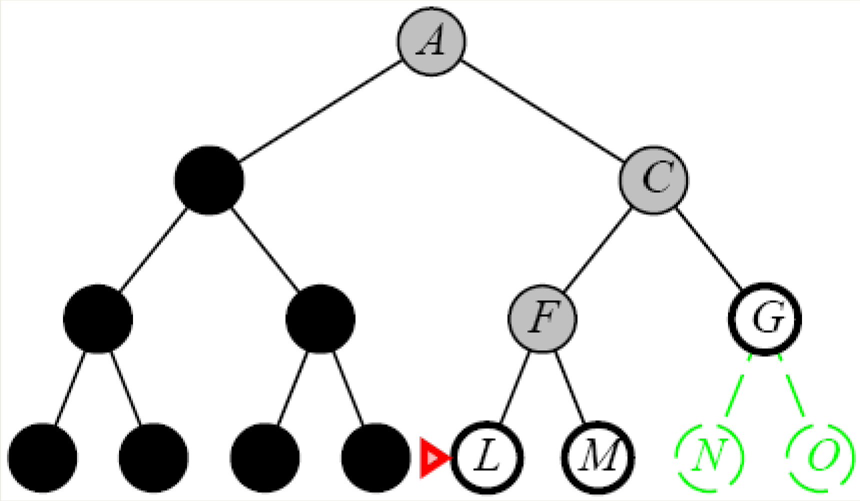
Depth-First Search



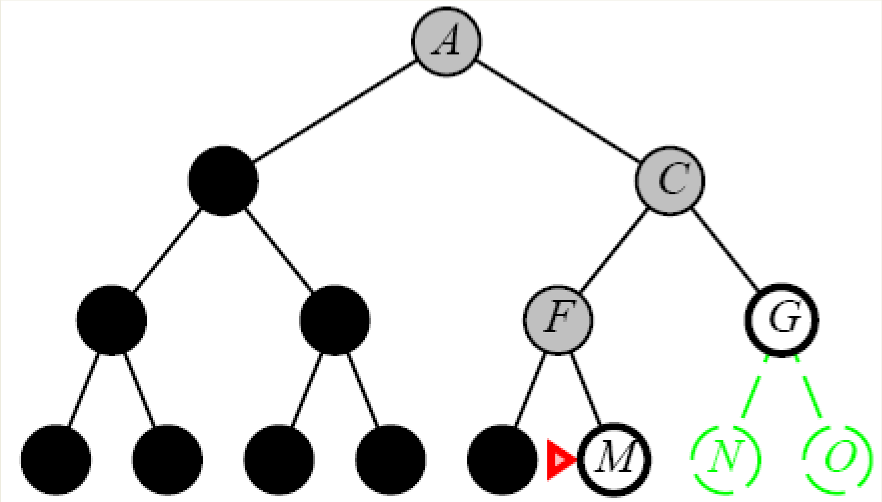
Depth-First Search



Depth-First Search



Depth-First Search



Depth-First Search

- Complete
  - No: fails in infinite-depth spaces, spaces with loops
    - Modify to avoid repeated spaces along path
  - Yes: in finite spaces
- Time
  - $O(b^m)$
  - Not great if m is much larger than d
  - But if the solutions are dense, this may be faster than breadth-first search
- Space
  - O(bm)…linear space
- Optimal
  - No

Depth-Limited Search

- A variation of depth-first search that uses a depth limit
  - Alleviates the problem of unbounded trees
  - Search to a predetermined depth ℓ (“ell”)
  - Nodes at depth ℓ have no successors

- Same as depth-first search if $\ell = \infty$
- Can terminate for failure and cutoff

Depth-Limited Search

Recursive implementation:

```
function Depth-Limited-Search( problem, limit) returns soln/fail/cutoff
   Recursive-DLS(Make-Node(Initial-State[problem]), problem, limit)

function Recursive-DLS(node, problem, limit) returns soln/fail/cutoff
   cutoff-occurred? ← false
   if Goal-Test[problem](State[node]) then return node
   else if Depth[node] = limit then return cutoff
   else for each successor in Expand(node, problem) do
        result ← Recursive-DLS(successor, problem, limit)
        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

- Complete
  - Yes if $\ell < d$
- Time
  - $O(b^\ell)$
- Space
  - $O(b\ell)$
- Optimal
  - No if $\ell > d$

Iterative Deepening Search

- Iterative deepening depth-first search
  - Uses depth-first search
  - Finds the best depth limit
    - Gradually increases the depth limit; 0, 1, 2, … until a goal is found

Informed Search

- Best-first search
- A^* search
- Heuristics
- Hill climbing
- Iterative improvement algorithms

Informed (Heuristic) Search Strategies

- ***Informed Search*** – a strategy that uses problem-specific knowledge beyond the definition of the problem itself
- ***Best-First Search*** – an algorithm in which a node is selected for expansion based on an evaluation function f(n)
  - Traditionally the node with the lowest evaluation function is selected
  - Not an accurate name…expanding the best node first would be a straight march to the goal.
  - Choose the node that *appears* to be the best

Informed (Heuristic) Search Strategies

- There is a whole family of Best-First Search algorithms with different evaluation functions
  - Each has a heuristic function h(n)

- h(n) = estimated cost of the cheapest path from node n to a goal node

- Example: in route planning the estimate of the cost of the cheapest path might be the straight line distance between two cities

A Quick Review

- $g(n)$ = cost from the initial state to the current state n
- $h(n)$ = estimated cost of the cheapest path from node n to a goal node
- $f(n)$ = evaluation function to select a node for expansion (usually the lowest cost node)

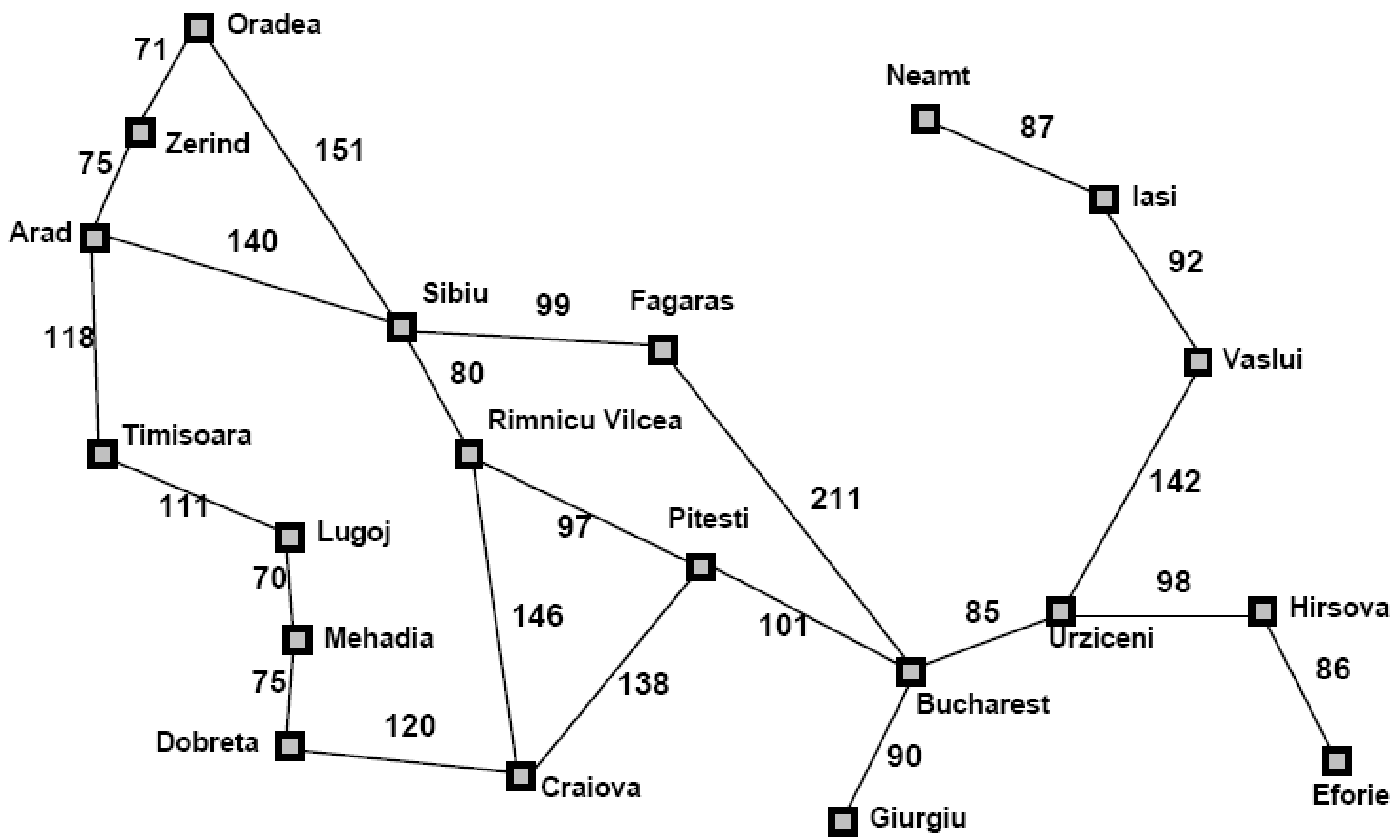
Greedy Best-First Search

- Greedy Best-First search tries to expand the node that is closest to the goal assuming it will lead to a solution quickly
  – f(n) = h(n)
  – aka “Greedy Search”

- Implementation
  – expand the “most desirable” node into the fringe queue
  – sort the queue in decreasing order of desirability

- Example: consider the straight-line distance heuristic h_{SLD}
  – Expand the node that appears to be closest to the goal

Greedy Best-First Search



Straight−line distance to Bucharest

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

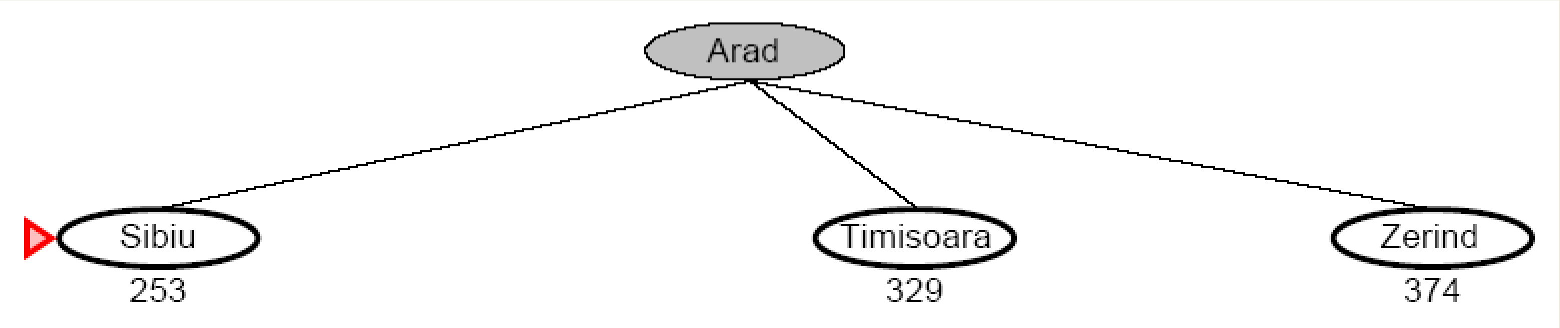
Greedy Best-First Search

- h_{SLD}(In(Arid)) = 366

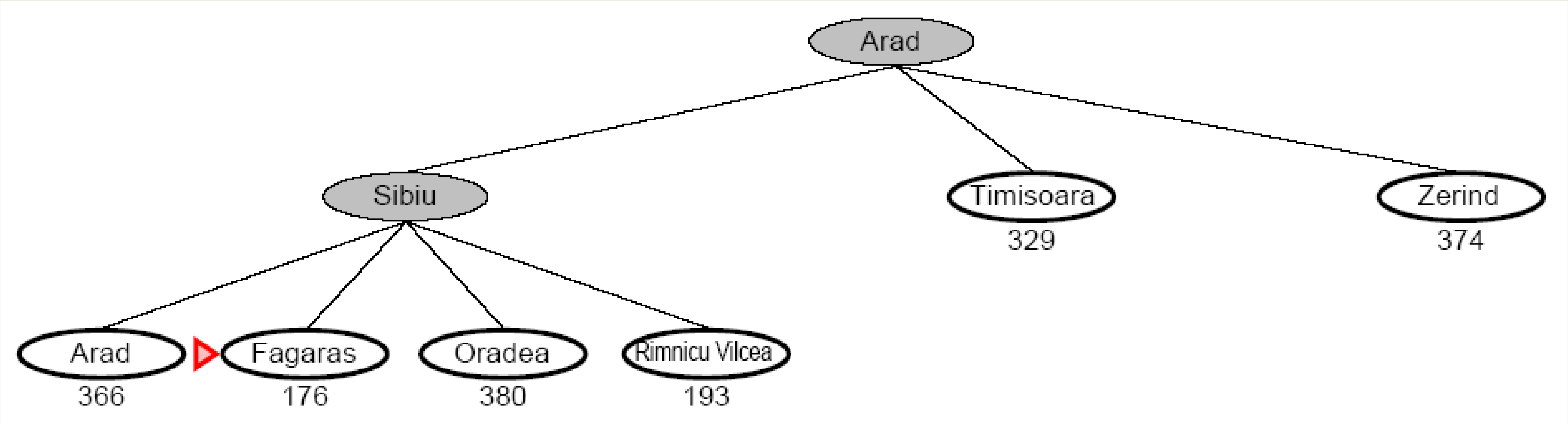
- Notice that the values of h_{SLD} cannot be computed from the problem itself

- It takes some experience to know that h_{SLD} is correlated with actual road distances
  - Therefore a useful heuristic

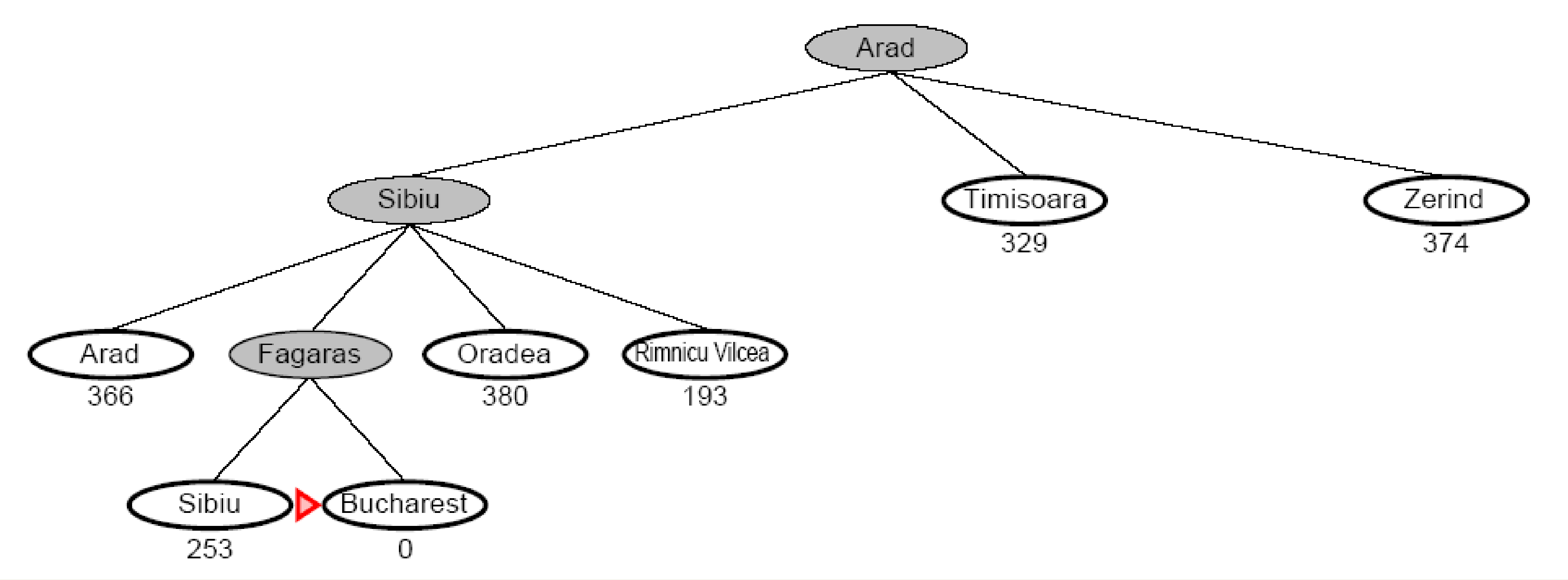
Greedy Best-First Search



Greedy Best-First Search



Greedy Best-First Search



Greedy Best-First Search

- Complete
  - No, GBFS can get stuck in loops (e.g. bouncing back and forth between cities)
- Time
  - $O(b^m)$ but a good heuristic can have dramatic improvement
- Space
  - $O(b^m)$ – keeps all the nodes in memory
- Optimal
  - No!

A Quick Review - Again

- g(n) = cost from the initial state to the current state n
- h(n) = estimated cost of the cheapest path from node n to a goal node
- f(n) = evaluation function to select a node for expansion (usually the lowest cost node)

A* Search

- A* (A star) is the most widely known form of Best-First search
  - It evaluates nodes by combining $g(n)$ and $h(n)$
  - $f(n) = g(n) + h(n)$
  - Where
    - $g(n)$ = cost so far to reach n
    - $h(n)$ = estimated cost to goal from n
    - $f(n)$ = estimated total cost of path through n

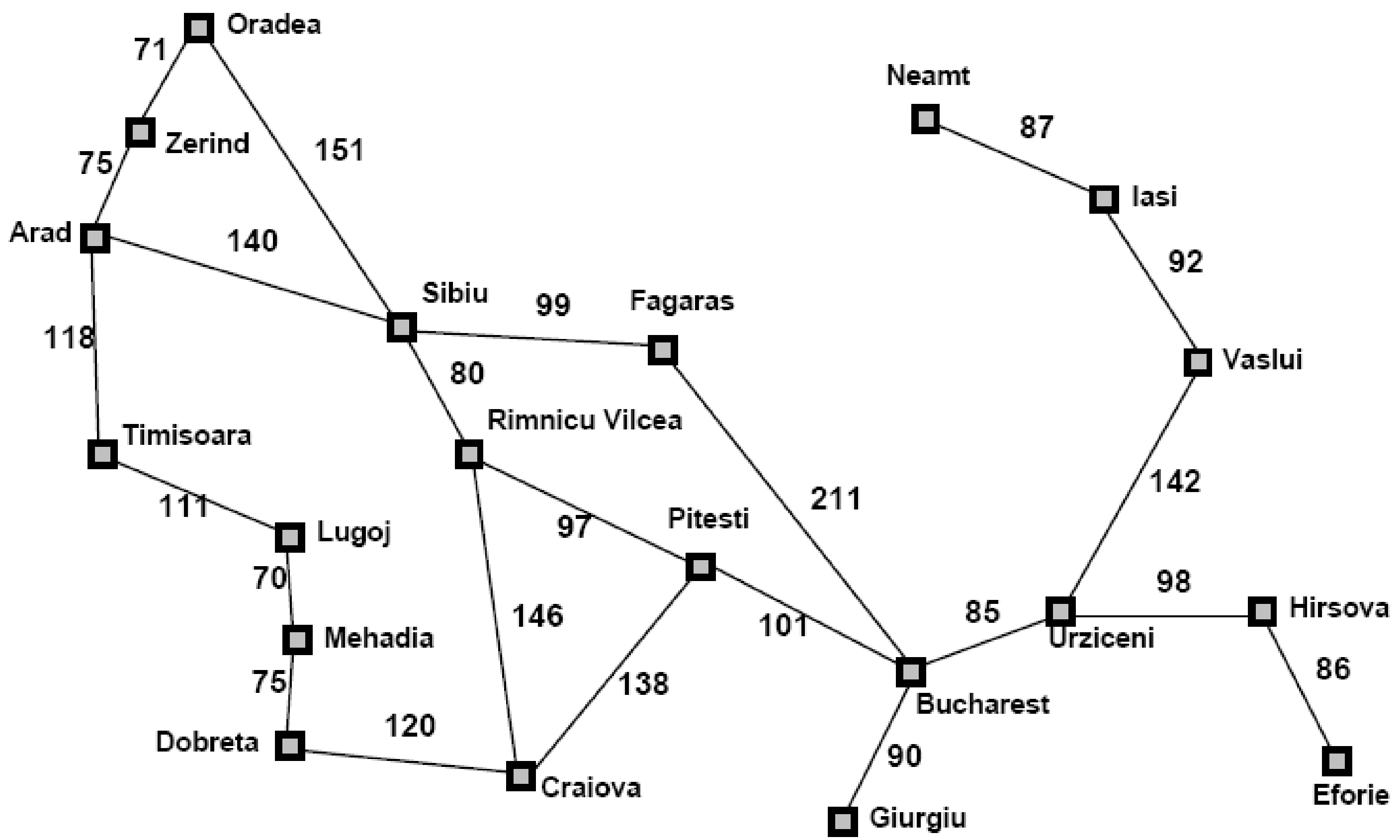
A* Search

- When h(n) = actual cost to goal
  - Only nodes in the correct path are expanded
  - Optimal solution is found
- When h(n) < actual cost to goal
  - Additional nodes are expanded
  - Optimal solution is found
- When h(n) > actual cost to goal
  - Optimal solution can be overlooked

A* Search

- A* is optimal if it uses an ***admissible heuristic***
  - h(n) <= h*(n) the true cost from node n
  - if h(n) *never overestimates* the cost to reach the goal
- Example
  - h_{SLD} never overestimates the actual road distance

Greedy Best-First Search

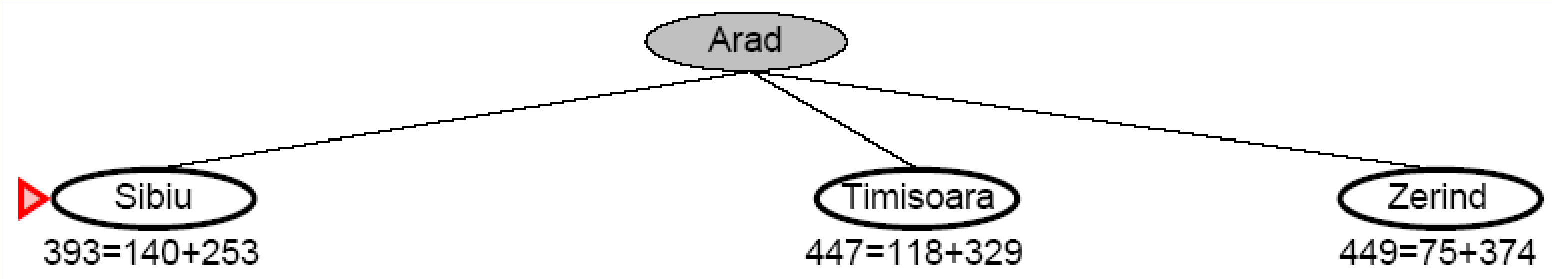


Straight−line distance
to Bucharest

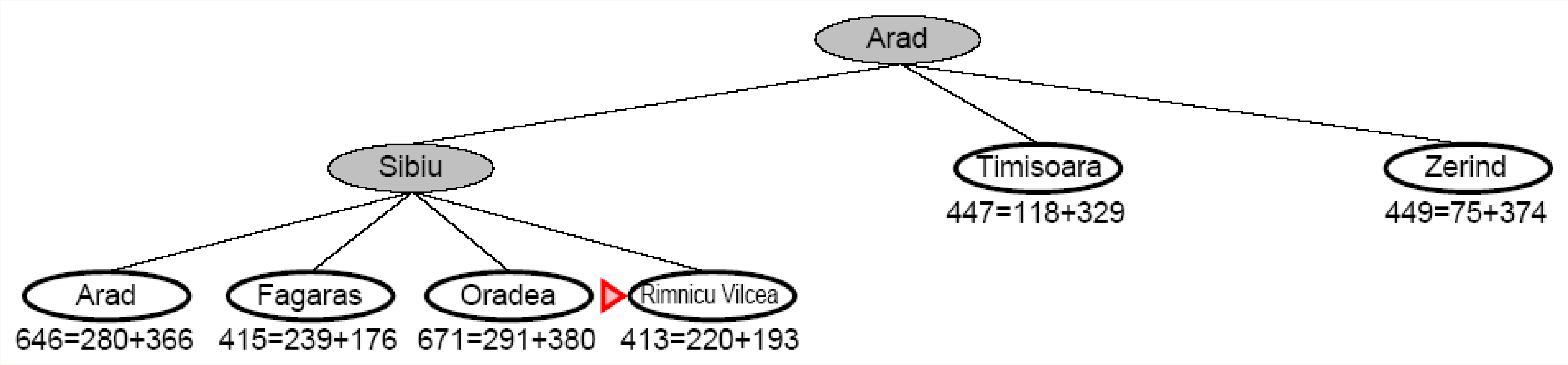
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

A* Search


Arad
366=0+366

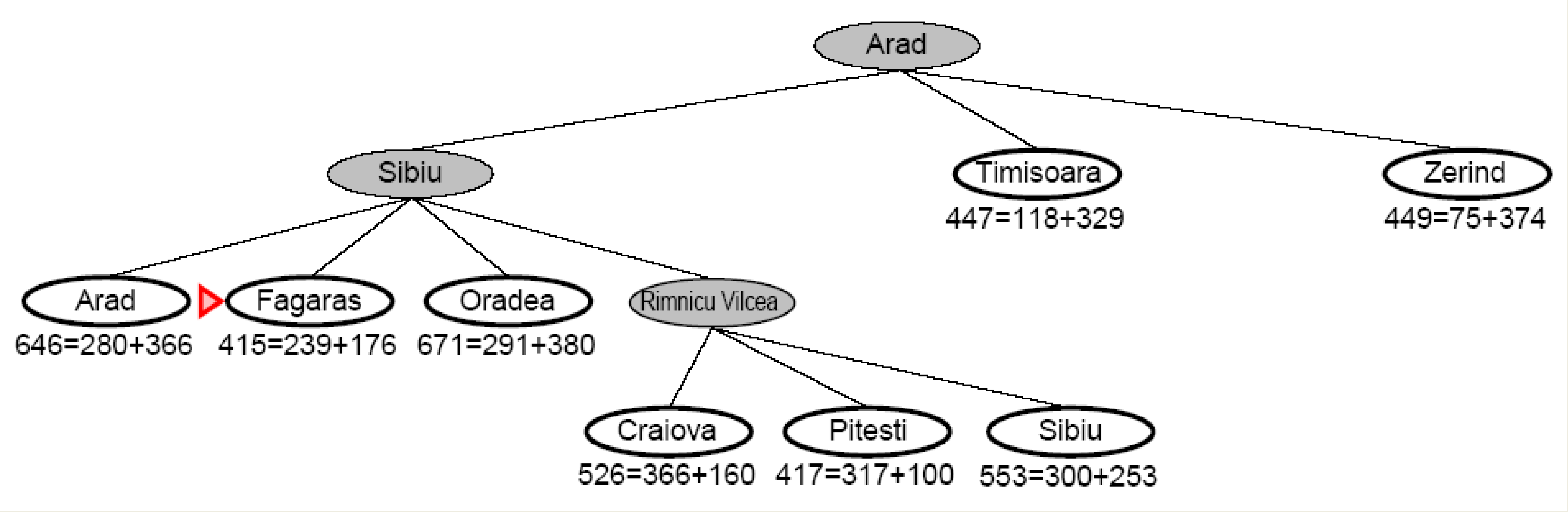

Arad
Sibiu
393=140+253
Timisoara
447=118+329
Zerind
449=75+374

A* Search



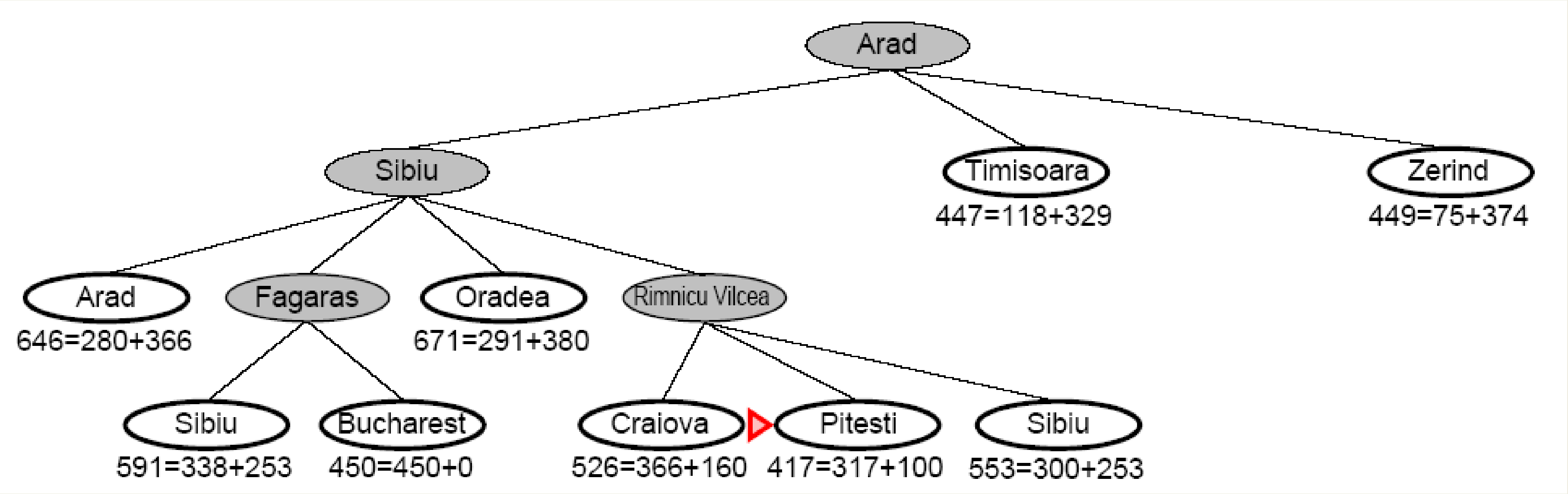
Arad
Sibiu
Timisoara
447=118+329
Zerind
449=75+374
Arad
646=280+366
Fagaras
415=239+176
Oradea
671=291+380
Rimnicu Vilcea
413=220+193

A* Search



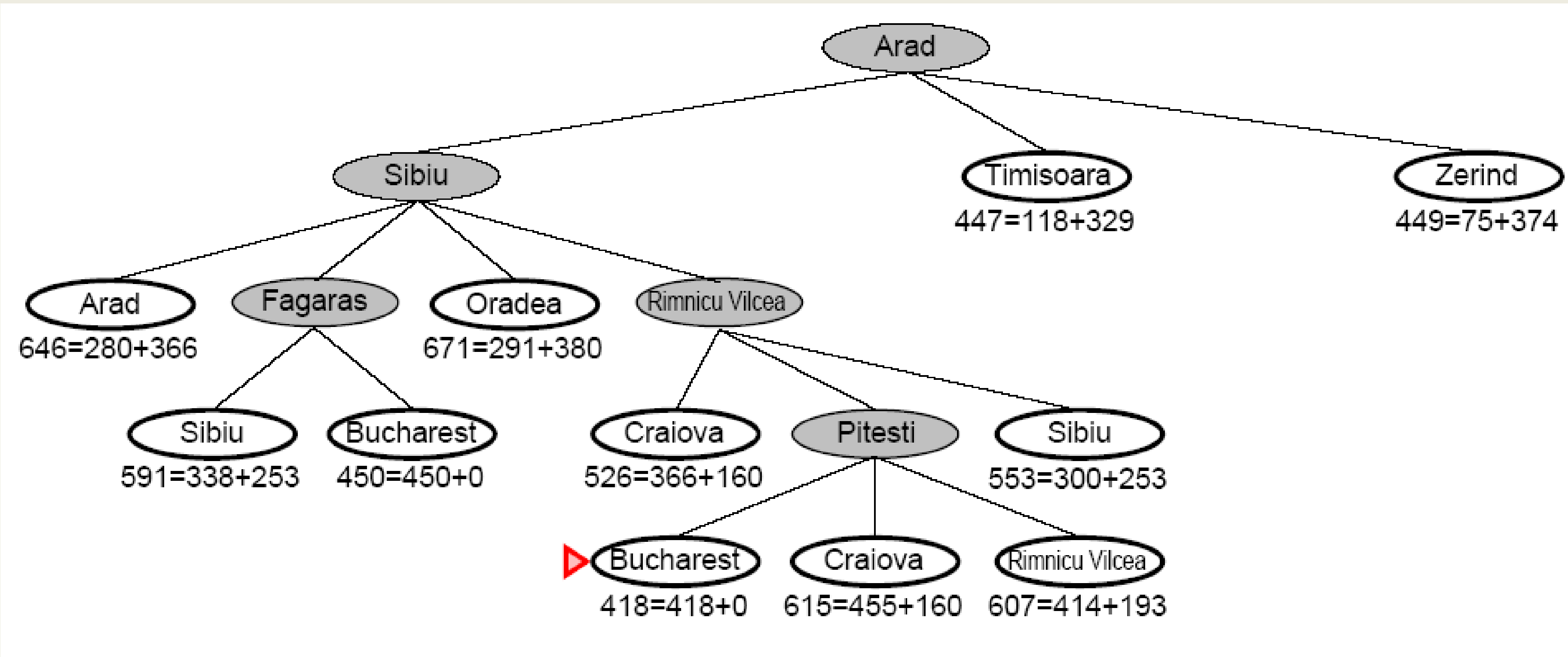
Arad
Sibiu
Timisoara
447=118+329
Zerind
449=75+374
Arad
646=280+366
Fagaras
415=239+176
Oradea
671=291+380
Rimnicu Vilcea
Craiova
526=366+160
Pitesti
417=317+100
Sibiu
553=300+253

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
417=317+100
Sibiu
553=300+253

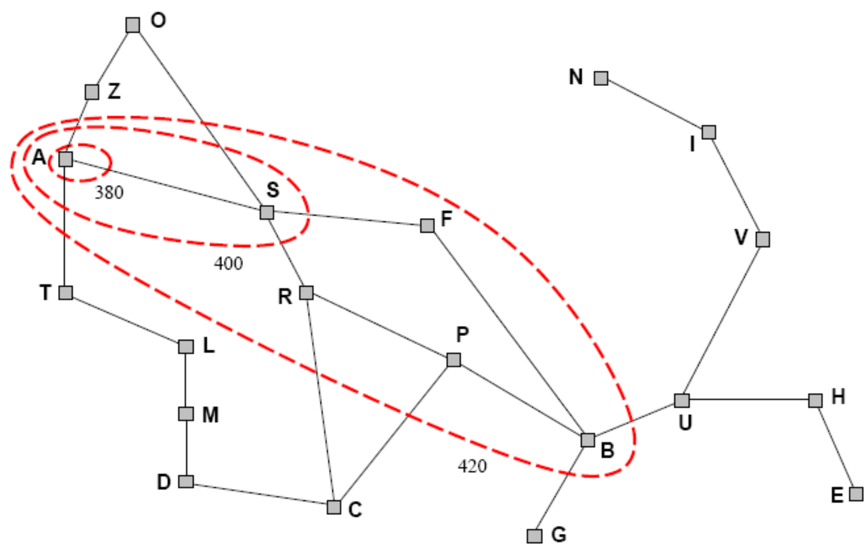
A* Search



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

A* Search

- A* expands nodes in increasing f value
  - Gradually adds f-contours of nodes (like breadth-first search adding layers)
  - Contour i has all nodes $f=f_i$ where $f_i < f_{i+1}$



A* Search

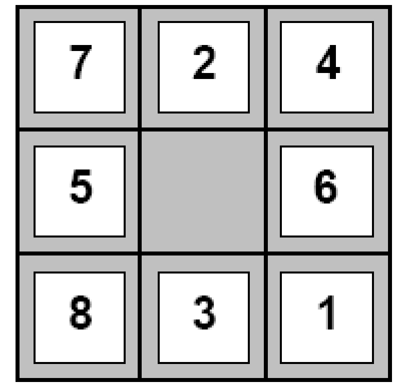
- Complete
  - Yes, unless there are infinitely many nodes with f <= f(G)
- Time
  - Exponential in [relative error of h x length of soln]
  - The better the heuristic, the better the time
    - Best case h is perfect, O(d)
    - Worst case h = 0, $O(b^d)$ same as BFS
- Space
  - Keeps all nodes in memory and save in case of repetition
  - This is $O(b^d)$ or worse
  - A* usually runs out of space before it runs out of time
- Optimal
  - Yes, cannot expand f_{i+1} unless f_i is finished

Memory-Bounded Heuristic Search

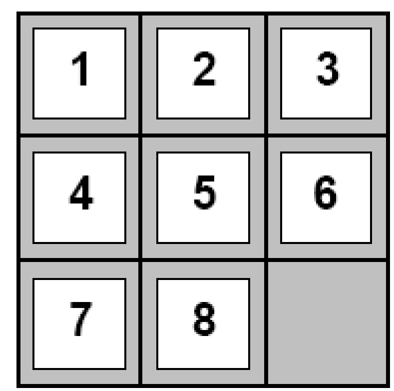
- Iterative Deepening A* (IDA*)
  - Similar to Iterative Deepening Search, but cut off at (g(n)+h(n)) > max instead of depth > max
  - At each iteration, cutoff is the first f-cost that exceeds the cost of the node at the previous iteration
- RBFS – see text figures 4.5 and 4.6
- Simple Memory Bounded A* (SMA*)
  - Set max to some memory bound
  - If the memory is full, to add a node drop the worst (g+h) node that is already stored
  - Expands newest best leaf, deletes oldest worst leaf

Heuristic Functions

- Example: 8-Puzzle
  - Average solution cost for a random puzzle is 22 moves
  - Branching factor is about 3
    - Empty tile in the middle -> four moves
    - Empty tile on the edge -> three moves
    - Empty tile in corner -> two moves
  - 3^{22} is approx 3.1e10
    - Get rid of repeated states
    - 181440 distinct states



Start State



Goal State

Heuristic Functions

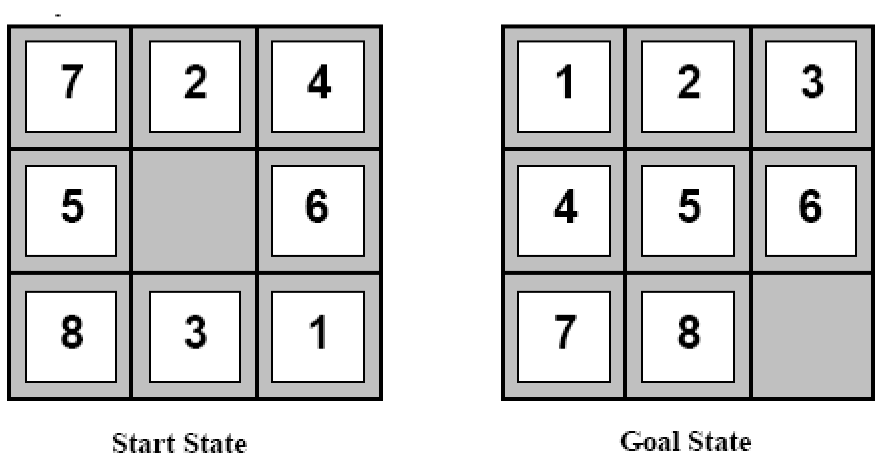
- To use A* a heuristic function must be used that never overestimates the number of steps to the goal

- h1=the number of misplaced tiles

- h2=the sum of the Manhattan distances of the tiles from their goal positions

Heuristic Functions

- h1 = 7
- h2 = 4+0+3+3+1+0+2+1 = 14



Dominance

- If h2(n) > h1(n) for all n (both admissible) then h2(n) dominates h1(n) and is better for the search

- Take a look at figure 4.8!

Relaxed Problems

- A Relaxed Problem is a problem with fewer restrictions on the actions
  - The cost of an optimal solution to a relaxed problem is an admissible heuristic for the original problem

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

Relaxed Problems

- Example: 8-puzzle
  - Consider only getting tiles 1, 2, 3, and 4 into place
  - If the rules are relaxed such that a tile can move anywhere then h1(n) gives the shortest solution
  - If the rules are relaxed such that a tile can move to any adjacent square then h2(n) gives the shortest solution

Relaxed Problems

- Store sub-problem solutions in a database
  - # patterns is much smaller than the search space
  - Generate database by working backwards from the solution
  - If multiple sub-problems apply, take the max
  - If multiple disjoint sub-problems apply, heuristics can be added

Learning Heuristics From Experience

- h(n) is an estimate cost of the solution beginning at state n
- How can an agent construct such a function?
- Experience!
  - Have the agent solve many instances of the problem and store the actual cost of h(n) at some state n
  - Learn from the features of a state that are relevant to the solution, rather than the state itself
    - Generate “many” states with a given feature and determine the average distance
    - Combine the information from multiple features
      - $h(n) = c(1)*x1(n) + c(2)*x2(n) + \ldots$ where x1, x2, … are features

Optimization Problems

- Instead of considering the whole state space, consider only the current state
- Limits necessary memory; paths not retained
- Amenable to large or continuous (infinite) state spaces where exhaustive search algorithms are not possible
- Local search algorithms can't backtrack

Local Search Algorithms

- They are useful for solving ***optimization problems***
  - Aim is to find a best state according to an ***objective function***

- Many optimization problems do not fit the standard search model outlined in chapter 3
  - E.g. There is no goal test or path cost in Darwinian evolution
- State space landscape

Optimization Problems

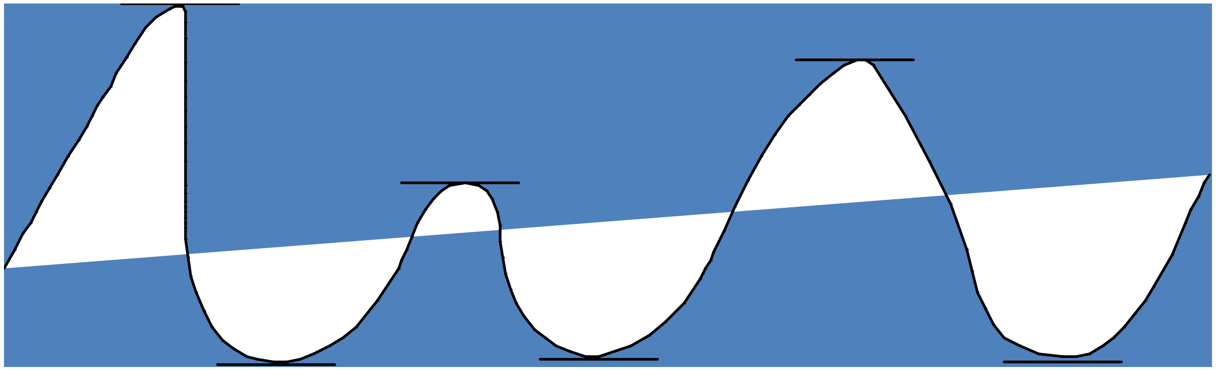
- Given measure of goodness (of fit)
  - Find optimal parameters (e.g correspondences)
  - That maximize goodness measure (or minimize badness measure)

- Optimization techniques
  - Direct (closed-form)
  - Search (generate-test)
  - Heuristic search (e.g Hill Climbing)
  - Genetic Algorithm

Direct Optimization

- The slope of a function at the maximum or minimum is 0
  - Function is neither growing nor shrinking
  - True at global, but also local extreme points

- Find where the slope is zero and you find extrema!
- (If you have the equation, use calculus (first derivative=0)



Hill Climbing

- Consider all possible successors as "one step" from the current state on the landscape.
- At each iteration, go to
  - The best successor (steepest ascent)
  - Any uphill move (first choice)
  - Any uphill move but steeper is more probable (stochastic)
- All variations get stuck at local maxima

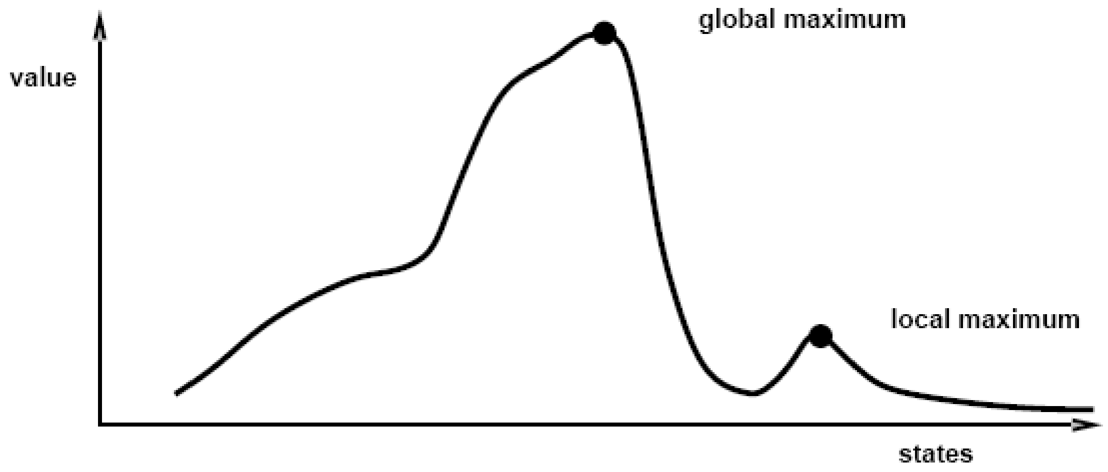
Hill Climbing

"Like climbing Everest in thick fog with amnesia"

```
function HILL-CLIMBING(problem) returns a state that is a local maximum
   inputs: problem, a problem
   local variables: current, a node
                    neighbor, a node

   current ← MAKE-NODE(INITIAL-STATE[problem])
   loop do
      neighbor ← a highest-valued successor of current
      if VALUE[neighbor] < VALUE[current] then return STATE[current]
      current ← neighbor
   end
```

Hill Climbing

Problem: depending on initial state, can get stuck on local maxima



In continuous spaces, problems w/ choosing step size, slow convergence

Hill Climbing

- Local maxima = no uphill step
  - Algorithms on previous slide fail (not complete)
  - Allow “random restart” which is complete, but might take a very long time
- Plateau = all steps equal (flat or shoulder)
  - Must move to equal state to make progress, but no indication of the correct direction
- Ridge = narrow path of maxima, but might have to go down to go up (e.g. diagonal ridge in 4-direction space)

Simulated Annealing

- Idea: Escape local maxima by allowing some “bad” moves
  - But gradually decreasing their frequency
- Algorithm is randomized:
  - Take a step if random number is less than a value based on both the objective function and the Temperature
- When Temperature is high, chance of going toward a higher value of optimization function J(x) is greater
- Note higher dimension: “perturb parameter vector” vs. “look at next and previous value”

Simulated Annealing

```
function SIMULATED-ANNEALING( problem, schedule) returns a solution state
    inputs: problem, a problem
            schedule, a mapping from time to "temperature"
    local variables: current, a node
                     next, a node
                     T, a "temperature" controlling prob. of downward steps

    current ← MAKE-NODE(INITIAL-STATE[problem])
    for t ← 1 to ∞ do
        T ← schedule[t]
        if T = 0 then return current
        next ← a randomly selected successor of current
        ΔE ← VALUE[next] – VALUE[current]
        if ΔE > 0 then current ← next
        else current ← next only with probability e^(ΔE/T)
```

Genetic Algorithms

- Quicker but randomized searching for an optimal parameter vector
- Operations
  - Crossover (2 parents -> 2 children)
  - Mutation (one bit)
- Basic structure
  - Create population
  - Perform crossover & mutation (on fittest)
  - Keep only fittest children

Genetic Algorithms

- Children carry parts of their parents’ data
- Only “good” parents can reproduce
  - Children are at least as “good” as parents?
    - No, but “worse” children don’t last long
- Large population allows many “current points” in search
  - Can consider several regions (watersheds) at once

Genetic Algorithms

- Representation
  - Children (after crossover) should be similar to parent, not random
  - Binary representation of numbers isn't good - what happens when you crossover in the middle of a number?
  - Need "reasonable" breakpoints for crossover (e.g. between R, xcenter and ycenter but not within them)
- "Cover"
  - Population should be large enough to "cover" the range of possibilities
  - Information shouldn't be lost too soon
  - Mutation helps with this issue

Experimenting With GAs

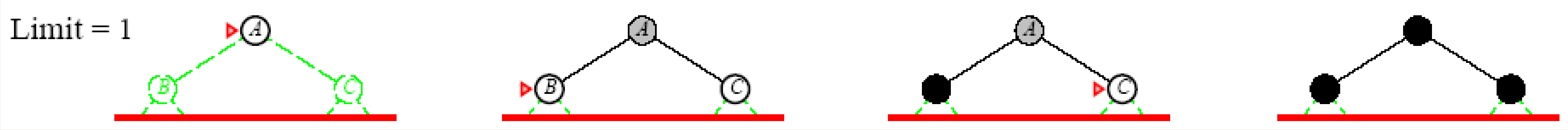
- Be sure you have a reasonable “goodness” criterion
- Choose a good representation (including methods for crossover and mutation)
- Generate a sufficiently random, large enough population
- Run the algorithm “long enough”
- Find the “winners” among the population
- Variations: multiple populations, keeping vs. not keeping parents, “immigration / emigration”, mutation rate, etc.

Iterative Deepening Search

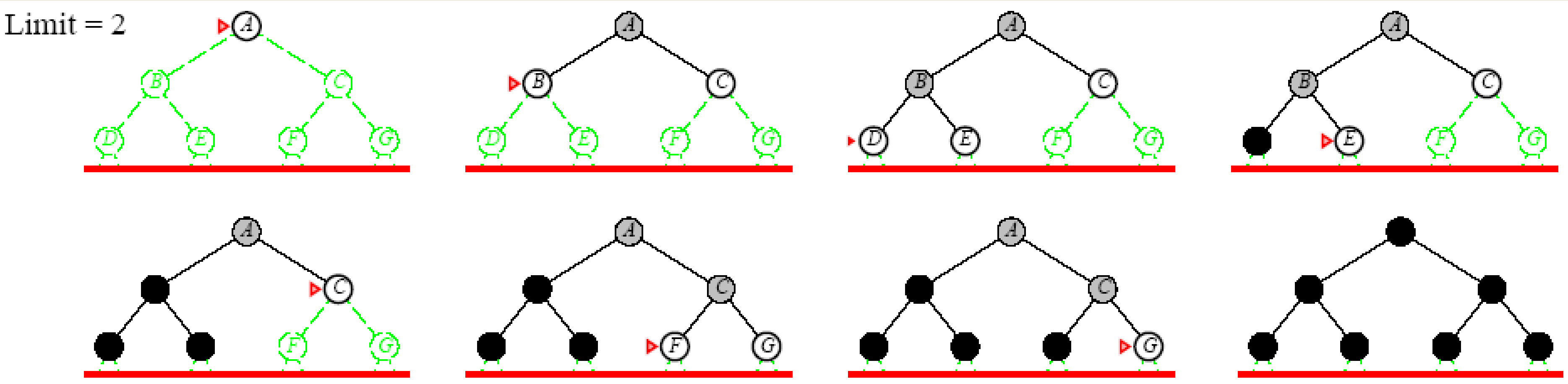
```
function ITERATIVE-DEEPENING-SEARCH( problem) returns a solution
    inputs: problem, a problem

    for depth ← 0 to ∞ do
        result ← DEPTH-LIMITED-SEARCH( problem, depth)
        if result ≠ cutoff then return result
    end
```

Iterative Deepening Search

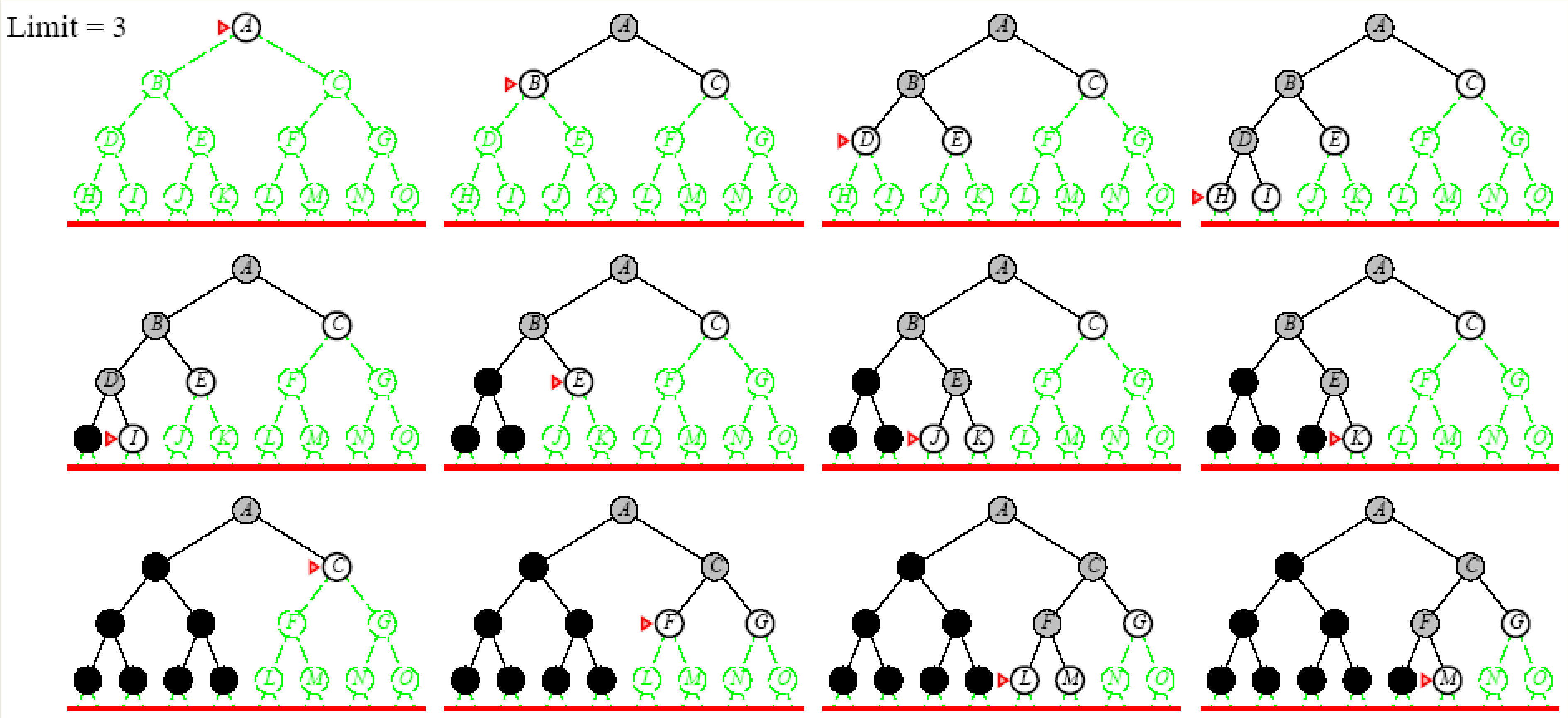
Iterative Deepening Search



Iterative Deepening Search



Iterative Deepening Search



Iterative Deepening Search

- Complete
  - Yes
- Time
  - $O(b^d)$
- Space
  - $O(bd)$
- Optimal
  - Yes if step cost = 1
  - Can be modified to explore uniform cost tree

Lessons From Iterative Deepening Search

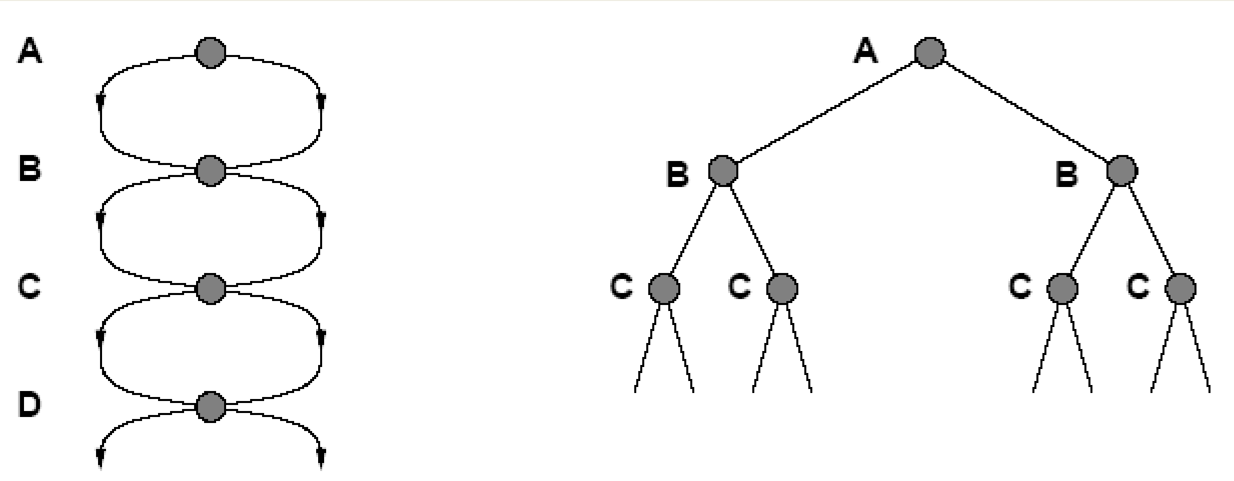
- Faster than BFS even though IDS generates repeated states
  - BFS generates nodes up to level d+1
  - IDS only generates nodes up to level d
- In general, iterative deepening search is the preferred uninformed search method when there is a large search space and the depth of the solution is not known

Avoiding Repeated States

- Complication of wasting time by expanding states that have already been encountered and expanded before
  - Failure to detect repeated states can turn a linear problem into an exponential one

- Sometimes, repeated states are unavoidable
  - Problems where the actions are reversable
    - Route finding
    - Sliding blocks puzzles

Avoiding Repeated States



State Space

Search Tree

Avoiding Repeated States

```
function GRAPH-SEARCH( problem, fringe) returns a solution, or failure
    closed ← an empty set
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe)
        if GOAL-TEST[problem](STATE[node]) then return node
        if STATE[node] is not in closed then
            add STATE[node] to closed
            fringe ← INSERTALL(EXPAND(node, problem), fringe)
    end
```

THANKS

金 融 先 锋　科 技 向 善