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)

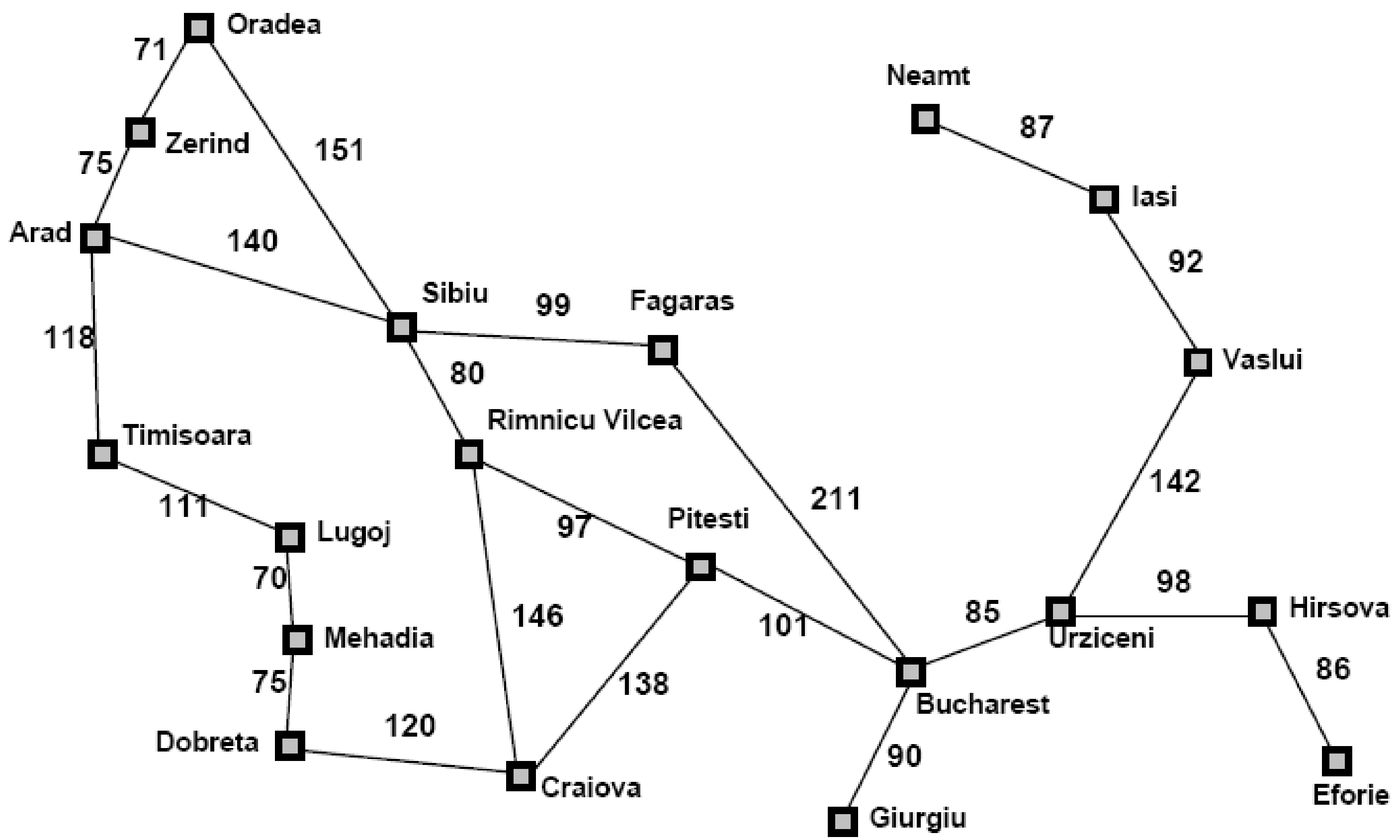
Greedy Best-First Search

- 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

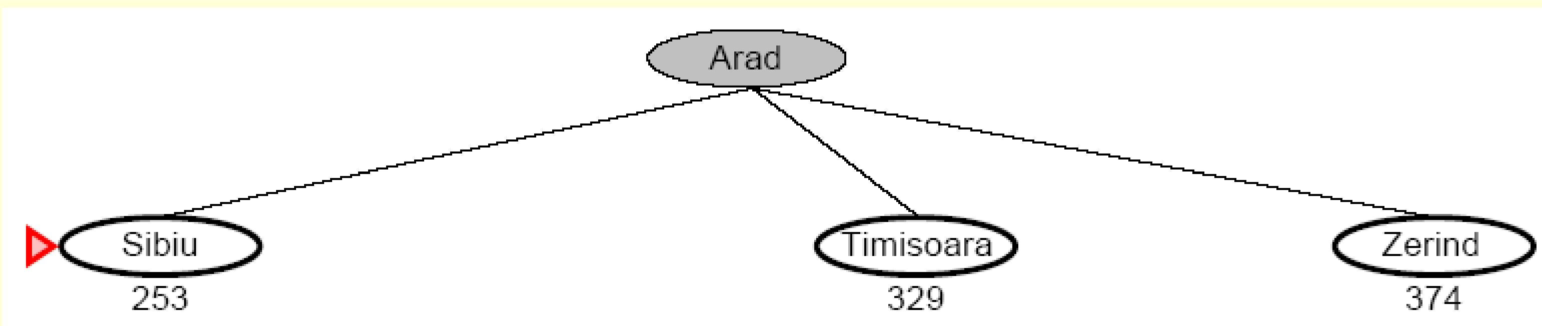
Greedy Best-First Search

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

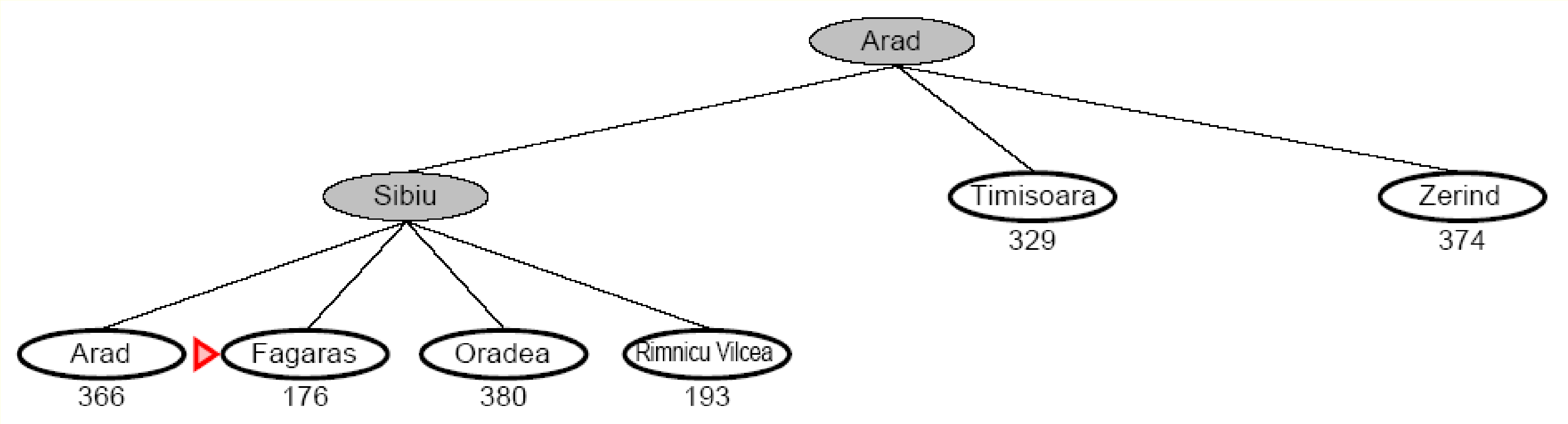
- Notice that the values of h_{SLD} cannot be computed from the problem itself

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

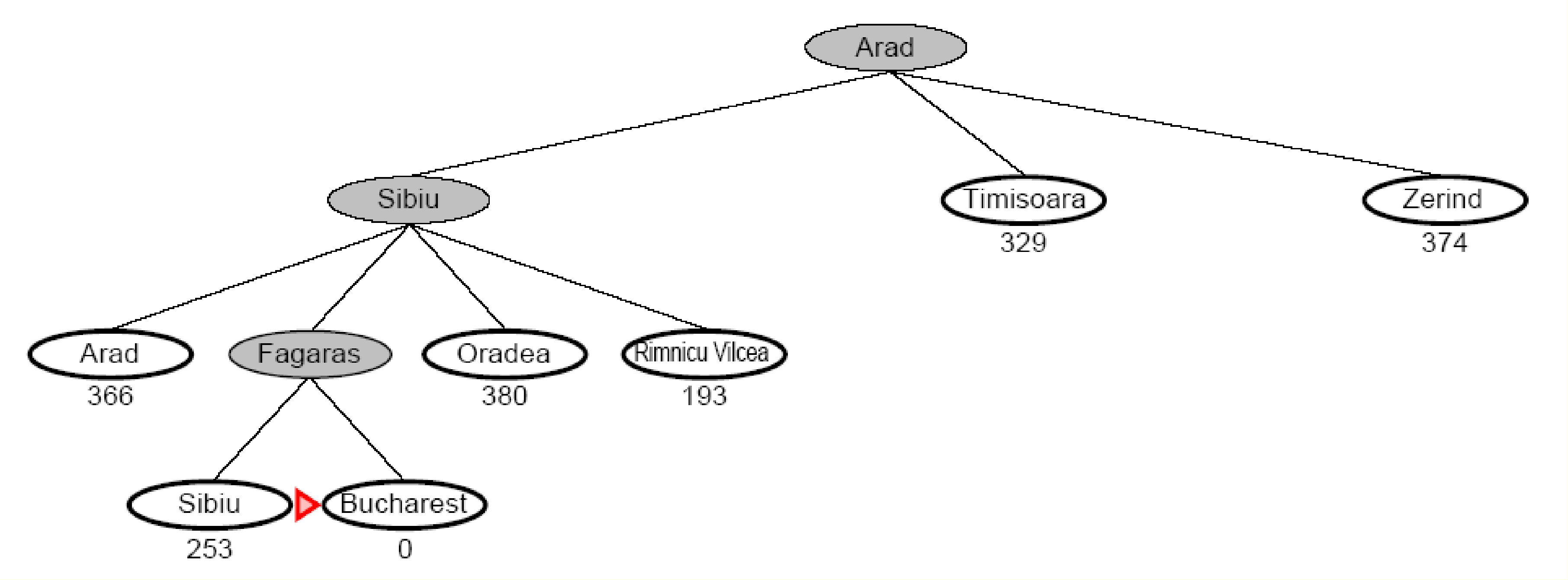
Greedy Best-First Search



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

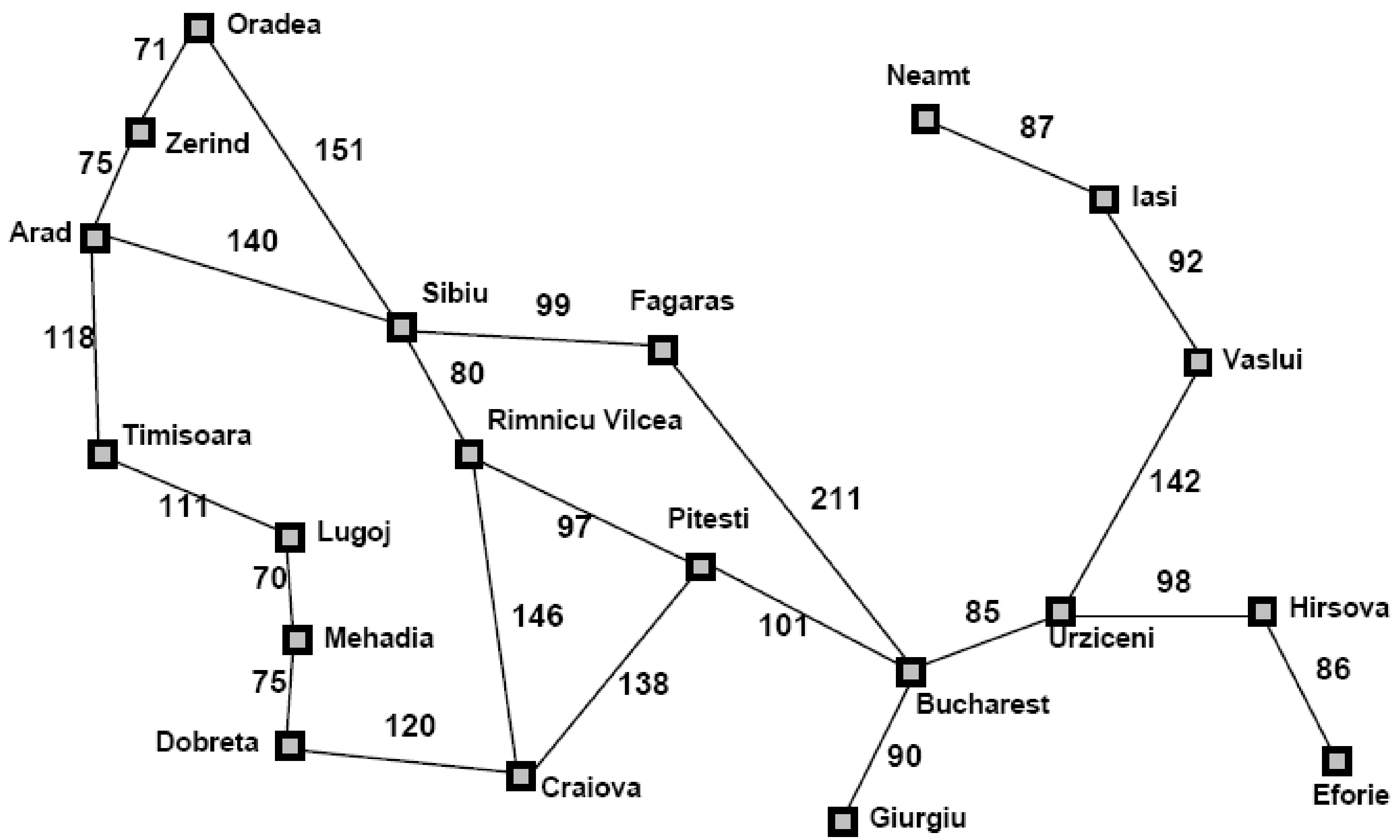
- When h(n) = actual cost to goal
  - Only nodes in the correct path are expanded
  - Optimal solution is found
- When h(n) < actual cost to goal
  - Additional nodes are expanded
  - Optimal solution is found
- When h(n) > actual cost to goal
  - Optimal solution can be overlooked

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

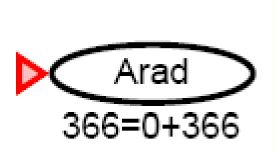
Greedy Best-First Search

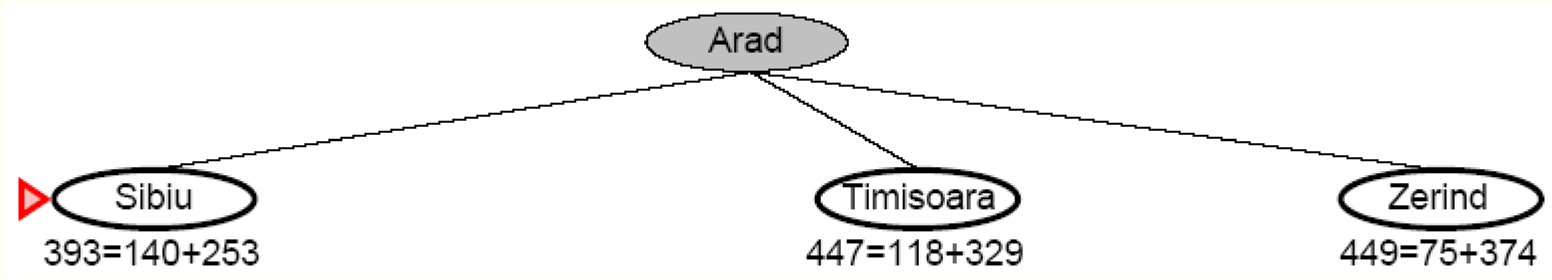


Straight−line distance
to Bucharest

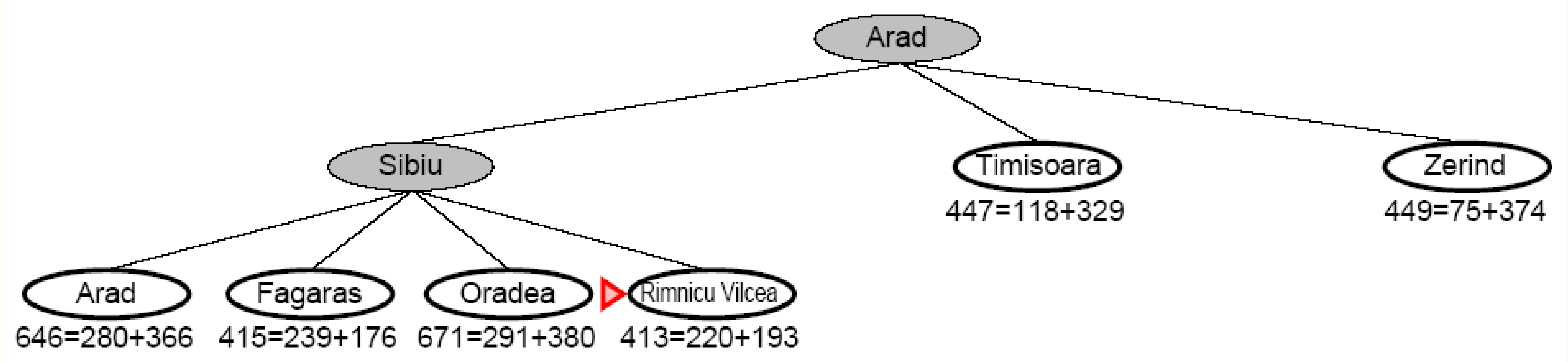
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

A* Search


Arad
366=0+366

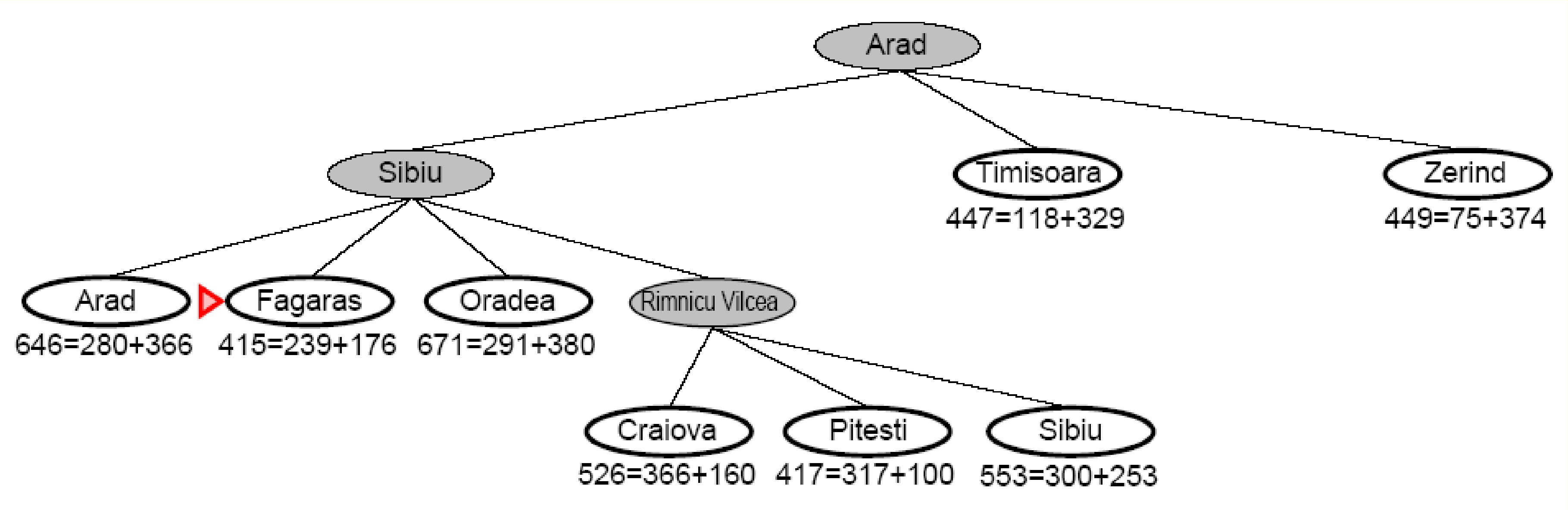

Arad
Sibiu
393=140+253
Timisoara
447=118+329
Zerind
449=75+374

A* Search

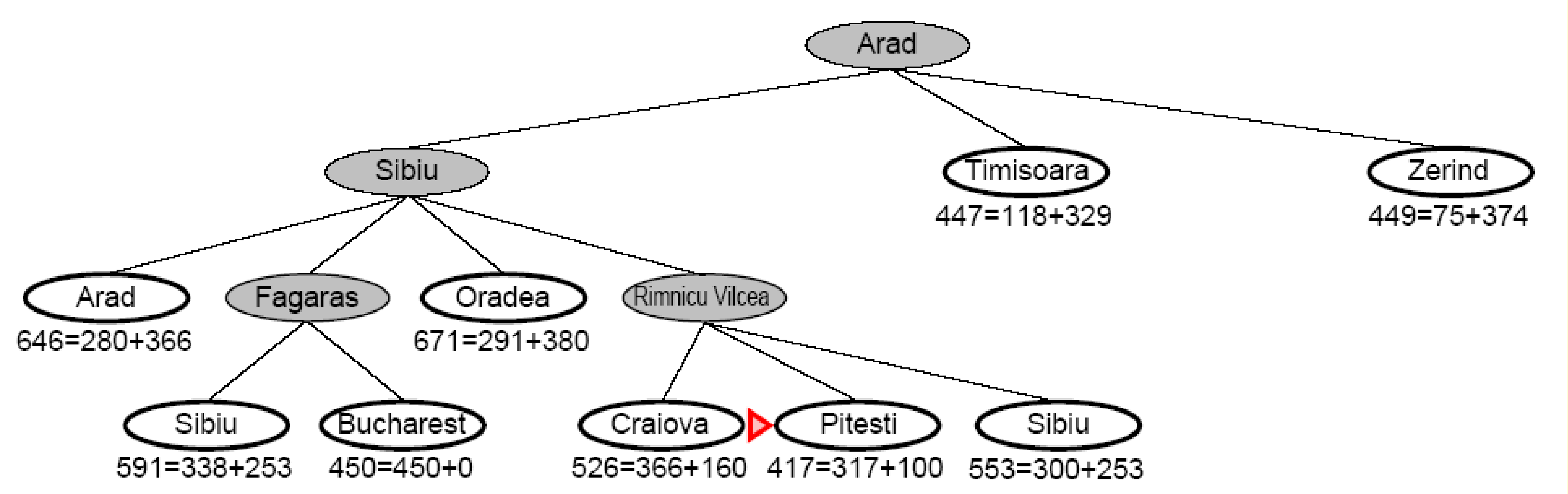


Arad
Sibiu
Timisoara
447=118+329
Zerind
449=75+374
Arad
646=280+366
Fagaras
415=239+176
Oradea
671=291+380
Rimnicu Vilcea
413=220+193

A* Search

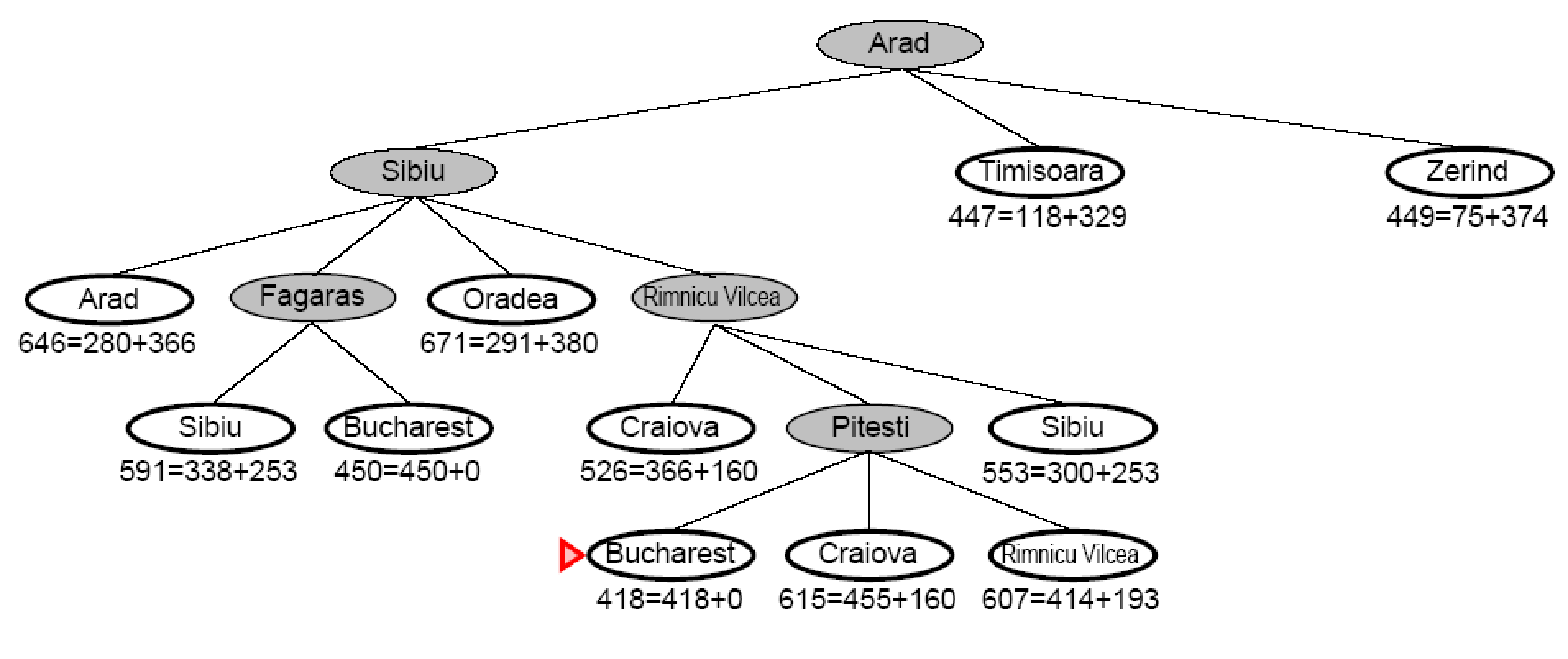

Arad
Sibiu
Timisoara
447=118+329
Zerind
449=75+374
Arad
646=280+366
Fagaras
415=239+176
Oradea
671=291+380
Rimnicu Vilcea
Craiova
526=366+160
Pitesti
417=317+100
Sibiu
553=300+253

A* Search



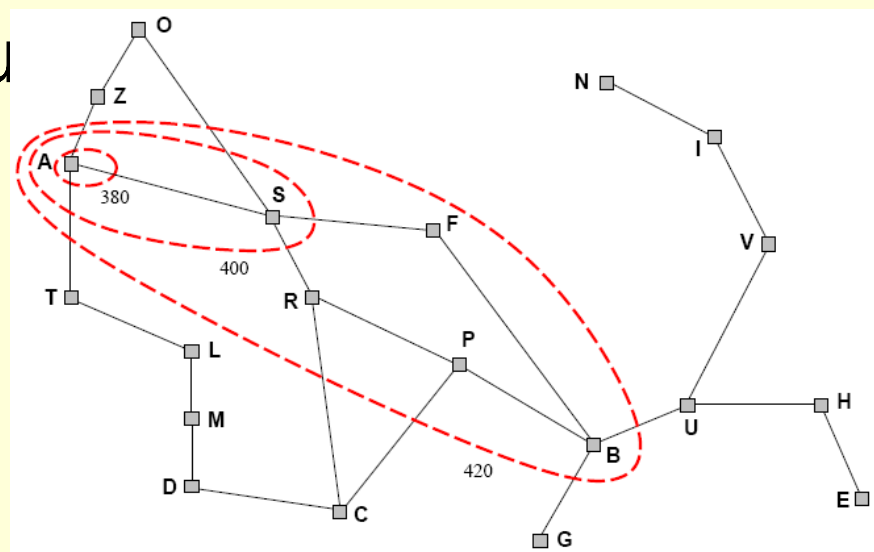
Arad
Sibiu
Timisoara
447=118+329
Zerind
449=75+374
Arad
646=280+366
Fagaras
Oradea
671=291+380
Rimnicu Vilcea
Sibiu
591=338+253
Bucharest
450=450+0
Craiova
526=366+160
Pitesti
417=317+100
Sibiu
553=300+253

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)
  - Contou here $f_i < f_{i+1}$



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

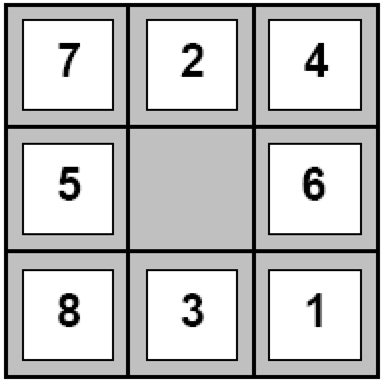
- Iterative Deepening A* (IDA*)
  - Similar to Iterative Deepening Search, but cut off at (g(n)+h(n)) > max instead of depth > max
  - At each iteration, cutoff is the first f-cost that exceeds the cost of the node at the previous iteration

- RBFS – see text figures 4.5 and 4.6

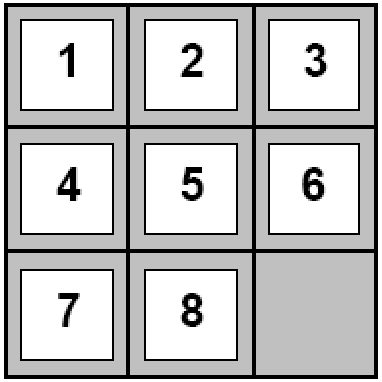
- Simple Memory Bounded A* (SMA*)
  - Set max to some memory bound
  - If the memory is full, to add a node drop the worst (g+h) node that is already stored
  - Expands newest best leaf, deletes oldest worst leaf

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



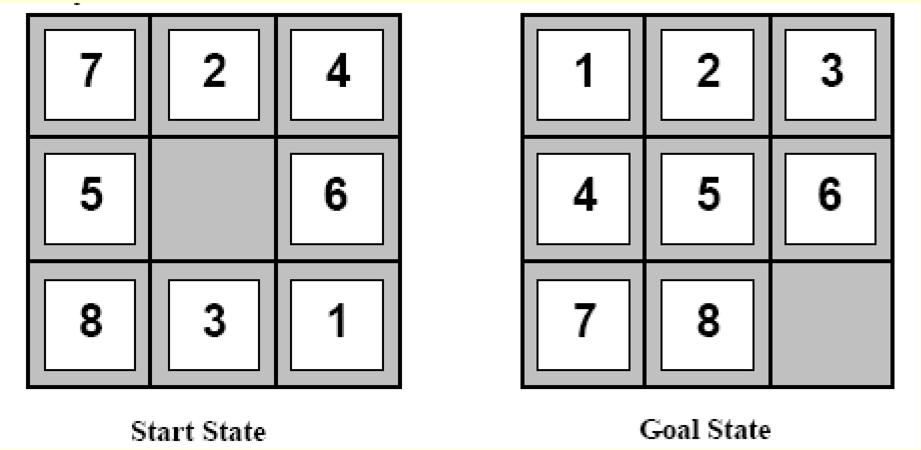
Goal State

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) + … 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

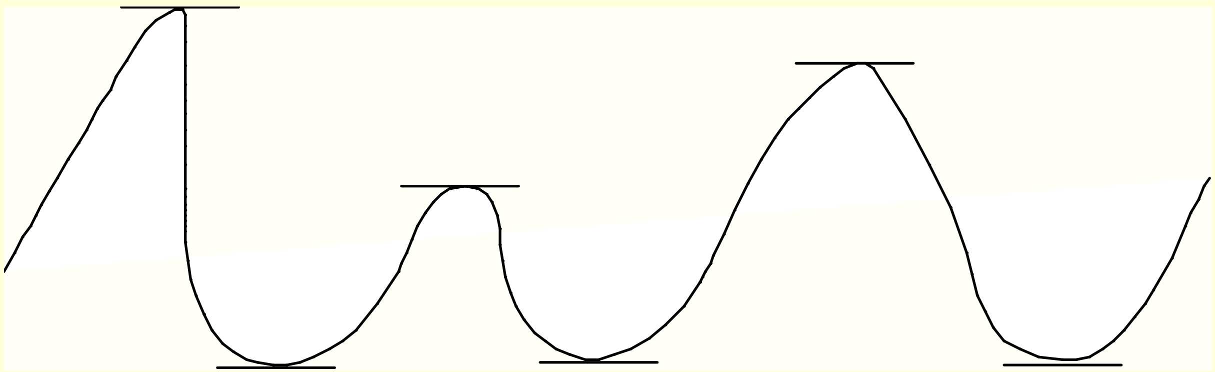
- Given measure of goodness (of fit)
  - Find optimal parameters (e.g correspondences)
  - That maximize goodness measure (or minimize badness measure)

- 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

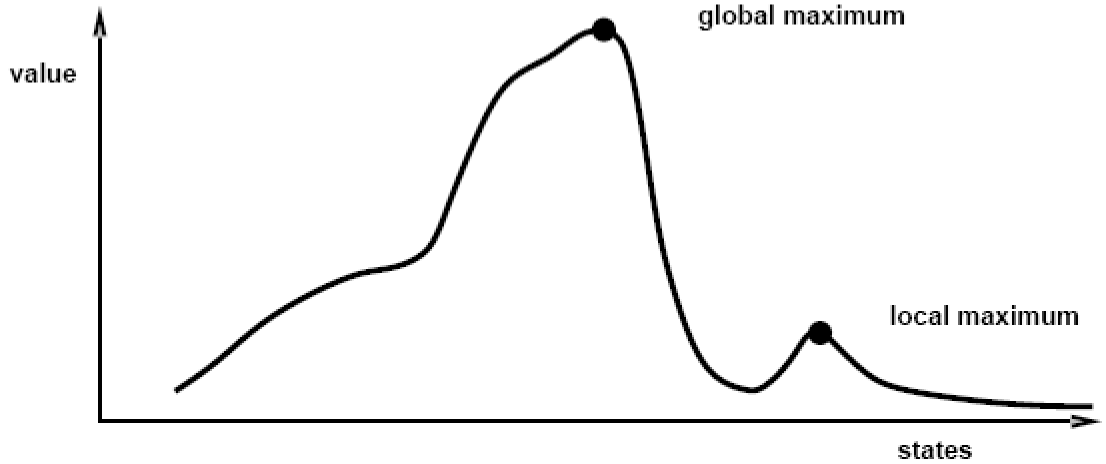
"Like climbing Everest in thick fog with amnesia"

```
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^(Δ E/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.