Informed Search and Exploration

Chapter 4

Outline

- Best-first search
- A* search
- Heuristics
- (IDA* search)
- Hill-climbing

Review: Tree search

function TREE-SEARCH (*problem, fringe*)
returns a solution, or failure

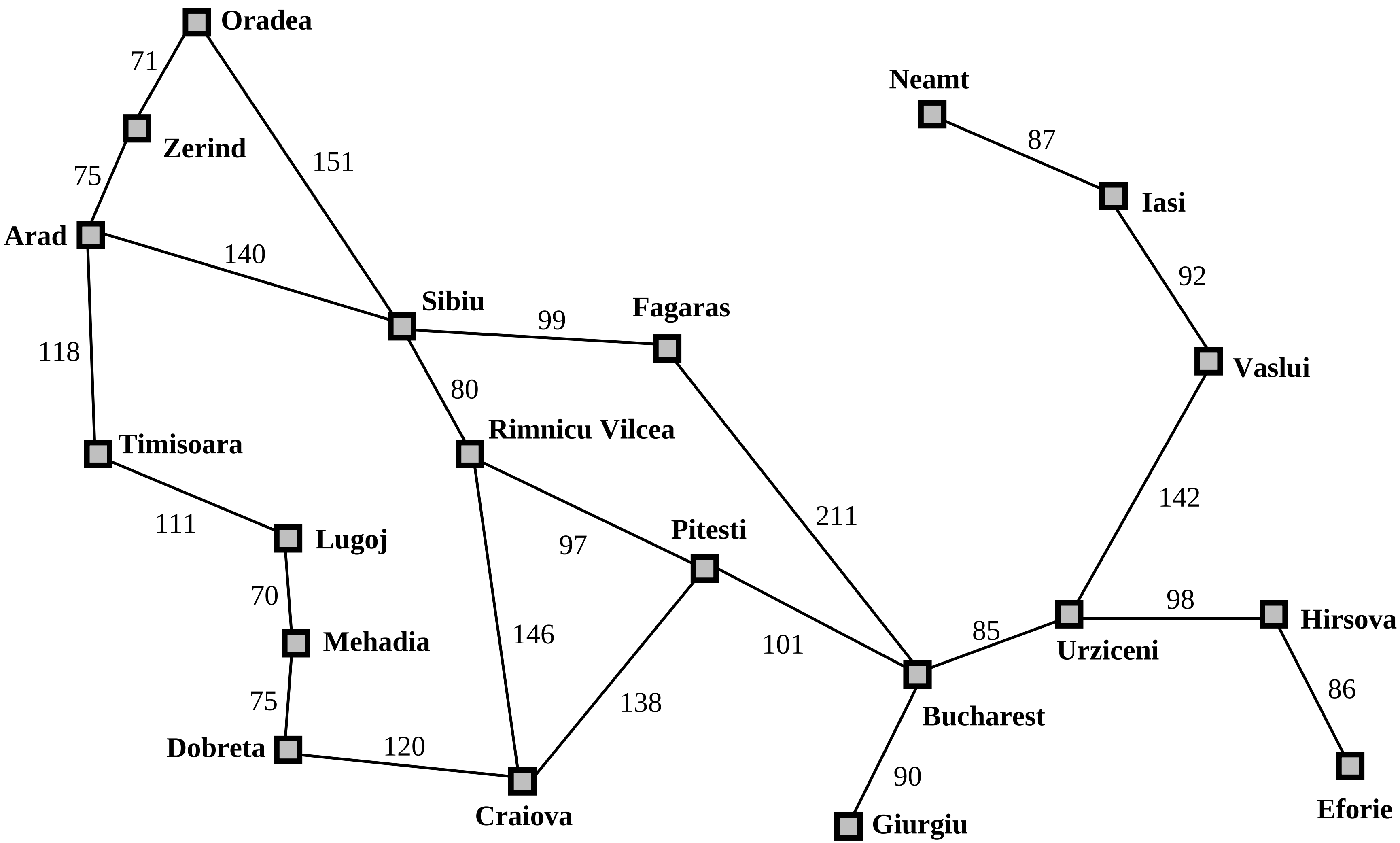
fringe ← INSERT(MAKE-NODE(INITIAL-STATE [*problem*]),*fringe*)
loop do
  if EMPTY?(*fringe*) **then return** failure
  node ← REMOVE-FIRST(*fringe*)
  if GOAL-TEST[*problem*] applied to STATE[*node*] succeeds
    then return SOLUTION(*node*)
  fringe ← INSERT-ALL(EXPAND(*node, problem*), *fringe*)

A strategy is defined by picking the *order of node expansion*.
The nodes are stored in the fringe.

Best-first search

- Idea: use an *evaluation function* for each node (the evaluation function is an estimate of "desirability")
- Expand the most desirable unexpanded node
- Implementation:
  fringe is a queue sorted in decreasing order of desirability
- Special cases:
  - greedy search
  - A^* search

Romania with step costs in km

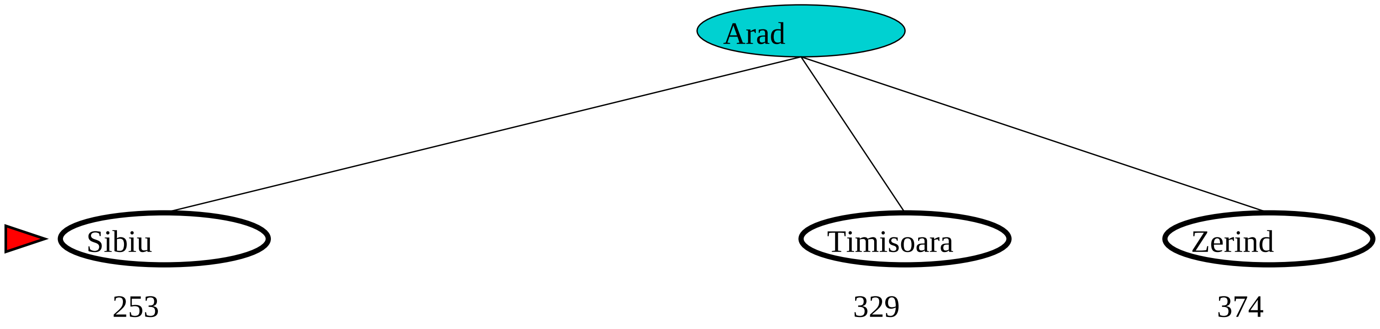


Greedy search

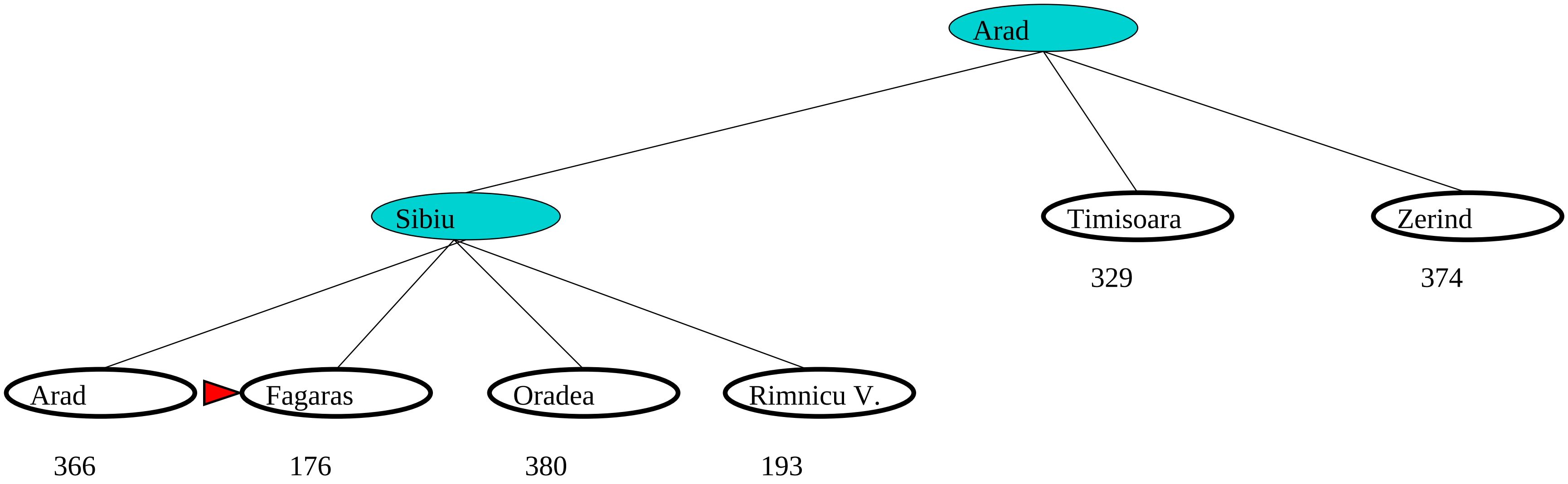
- Evaluation function $h(n)$ (***h***euristic) = estimate of cost from n to the closest goal
- E.g., $h_{\mathrm{SLD}}(n)$ = straight-line distance from n to Bucharest
- Greedy search expands the node that *appears* to be closest to goal

Greedy search example

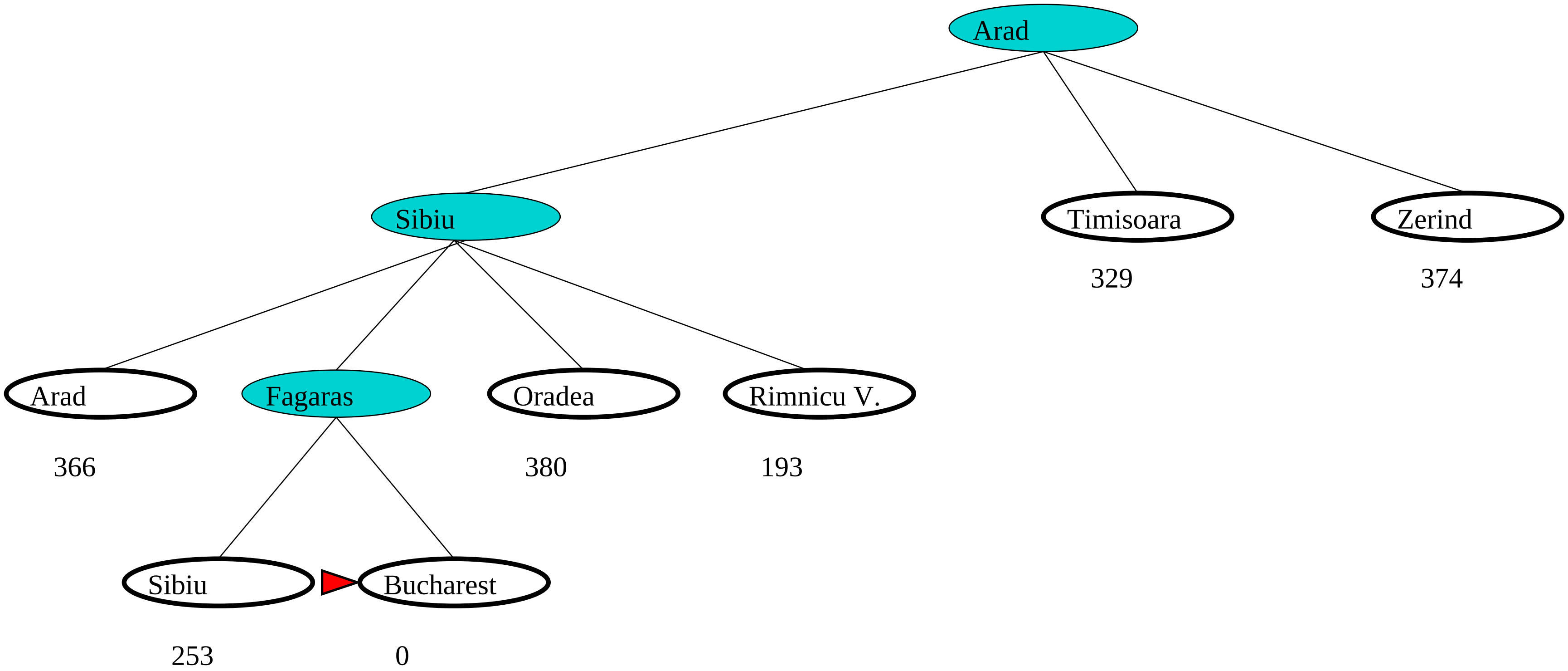
After expanding Arad



After expanding Sibiu



After expanding Fagaras



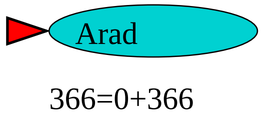
Properties of greedy search

- Complete No — can get stuck in loops, e.g.,
  Iasi $\to$ Neamt $\to$ Iasi $\to$ Neamt $\to$
  Complete in finite space with repeated-state checking
- Time $O(b^m)$, but a good heuristic can give dramatic improvement
- Space $O(b^m)$—keeps all nodes in memory
- Optimal No

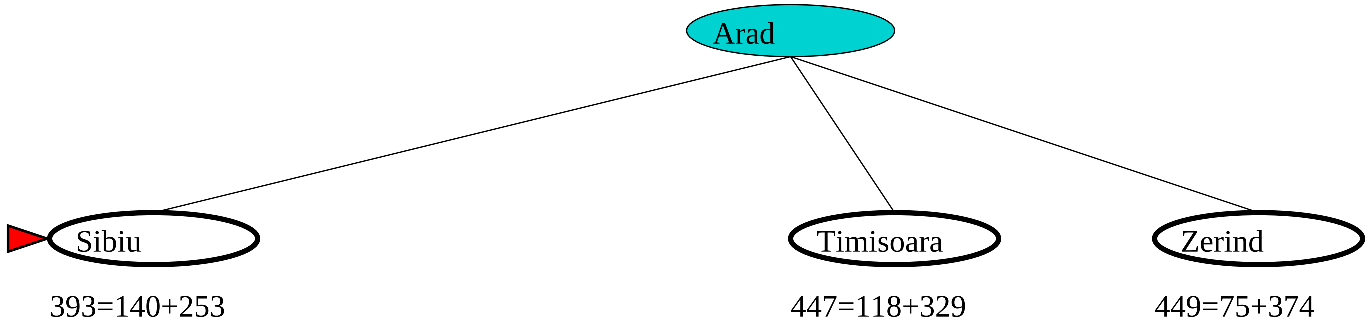
A* search

- Idea: avoid expanding paths that are already expensive
- *Evaluation function* $f(n) = g(n) + h(n)$
  - $g(n)$ = cost so far to reach n
  - $h(n)$ = estimated cost to goal from n
  - $f(n)$ = estimated total cost of path through n to goal
- A* search uses an *admissible* heuristic
  i.e., $h(n) \leq h^*(n)$ where $h^*(n)$ is the *true* cost from n.
  (Also require $h(n) \geq 0$, so $h(G) = 0$ for any goal G.)
  E.g., $h_{\mathrm{SLD}}(n)$ never overestimates the actual road distance.

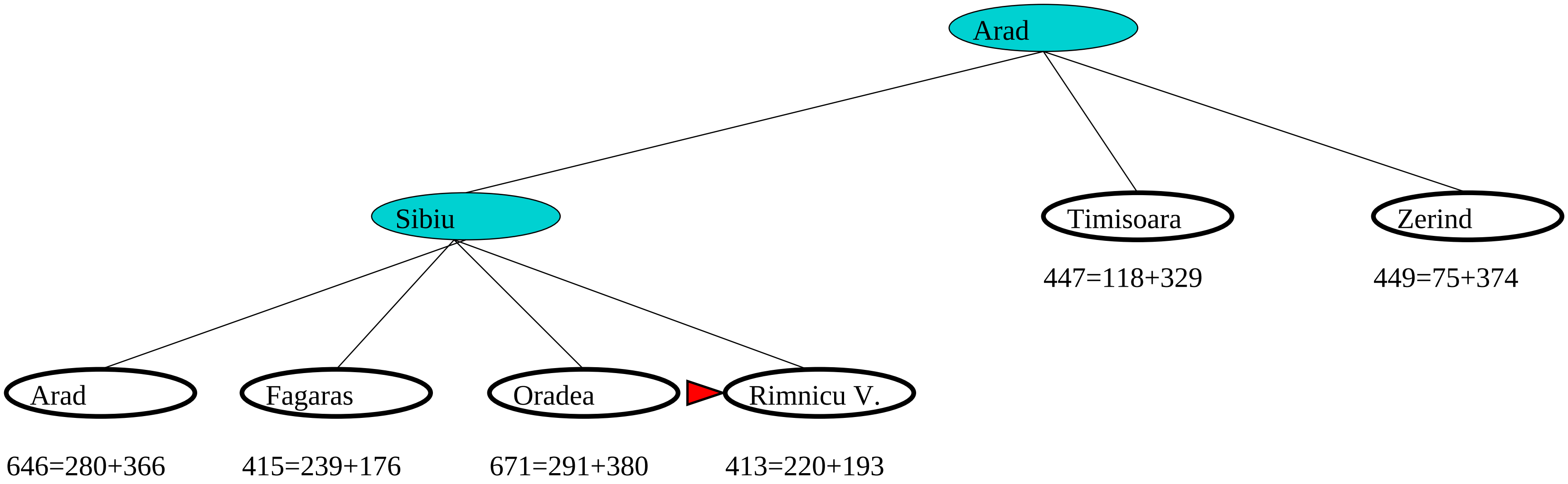
A* search example



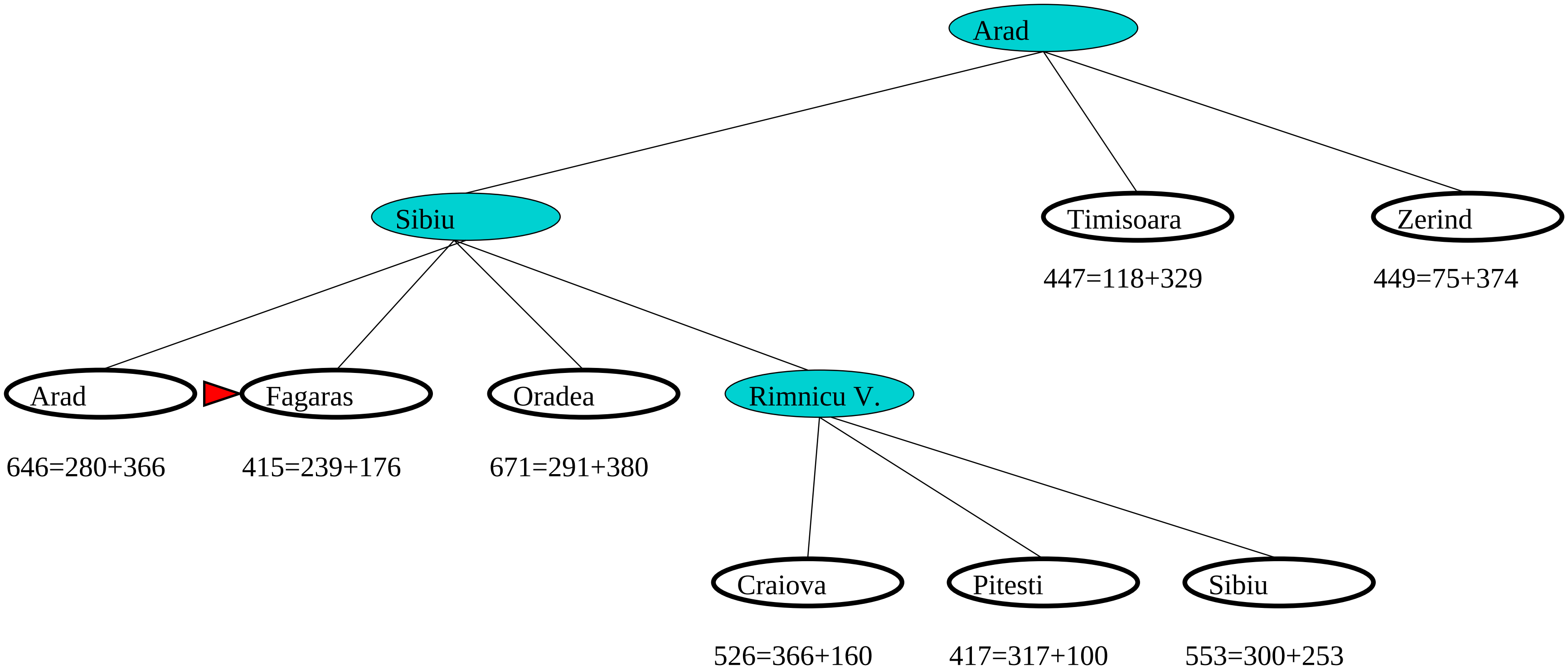
After expanding Arad



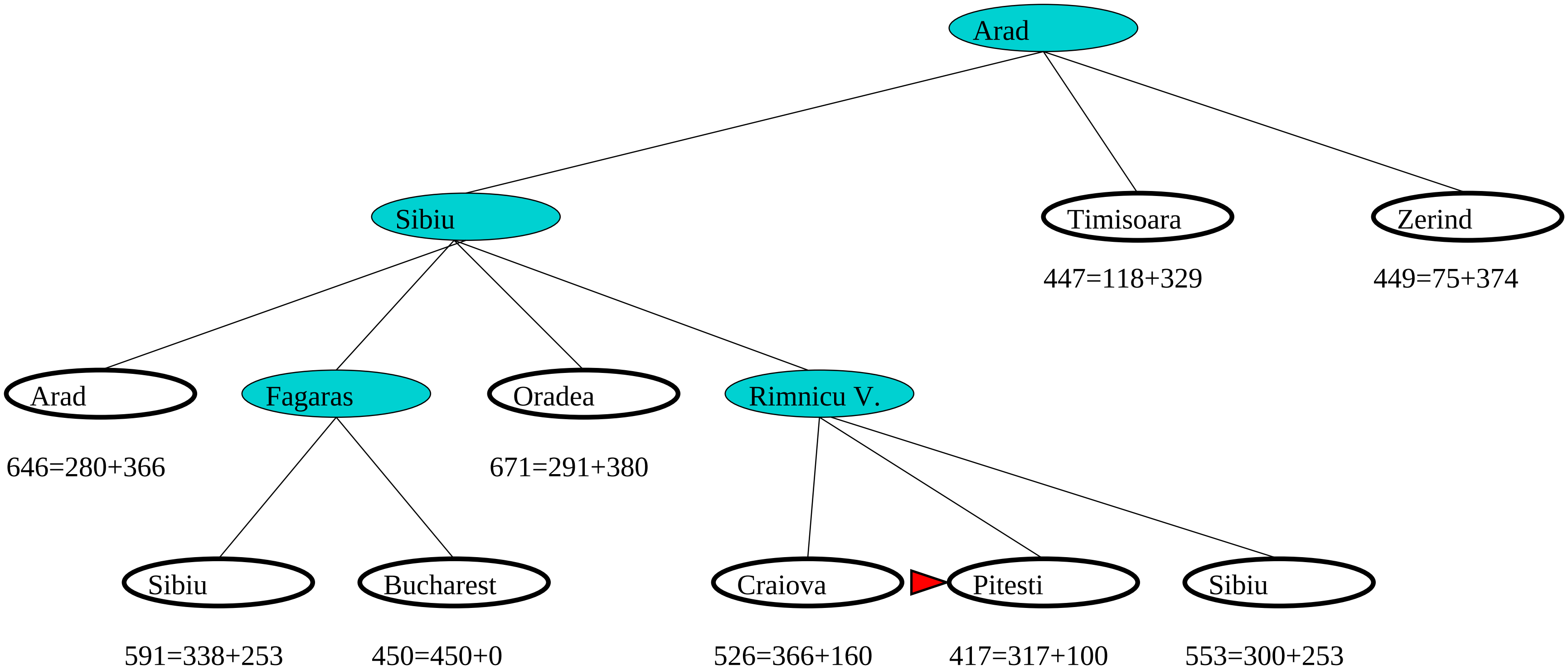
After expanding Sibiu



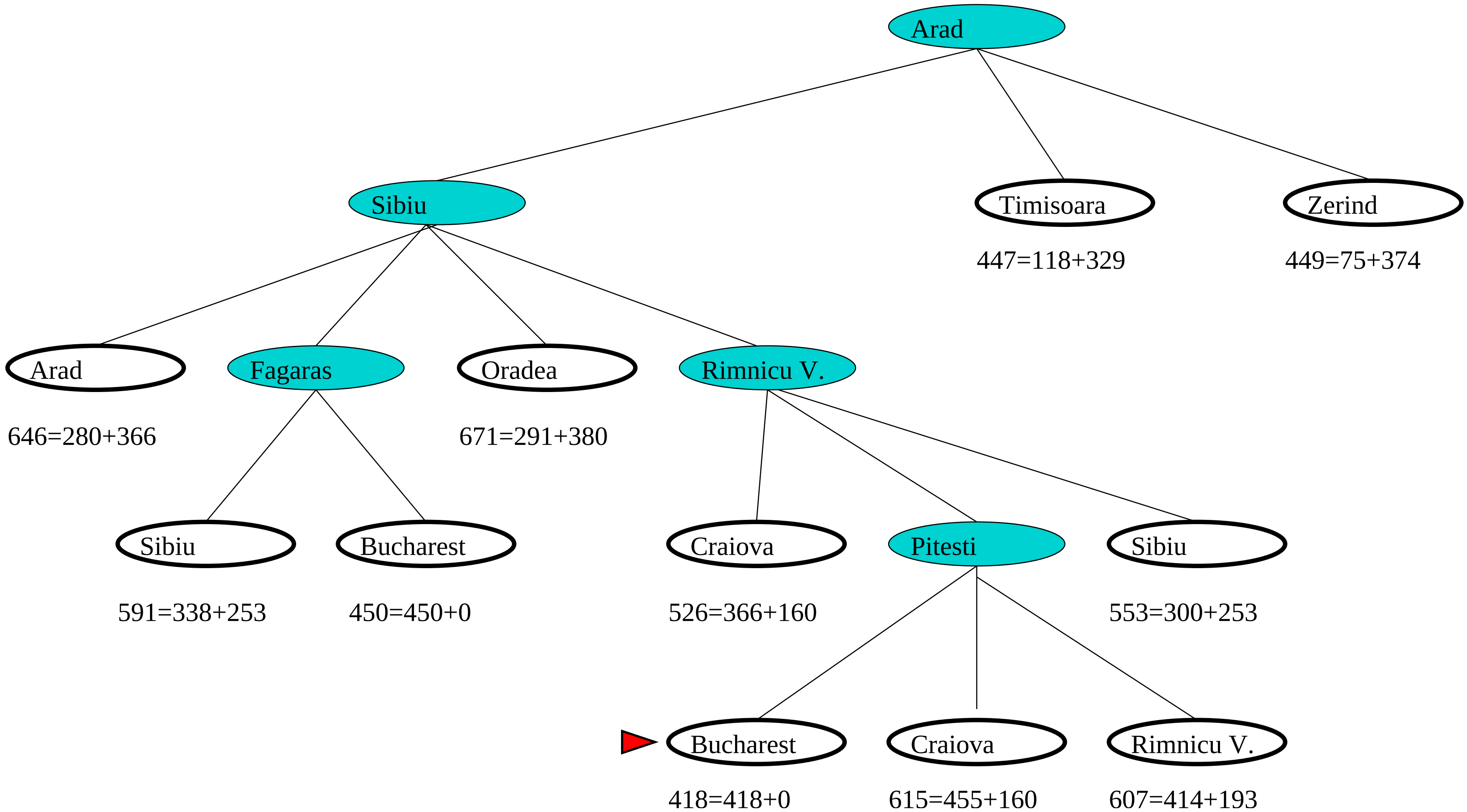
After expanding Rimnicu Vilcea



After expanding Fagaras



After expanding Pitesti

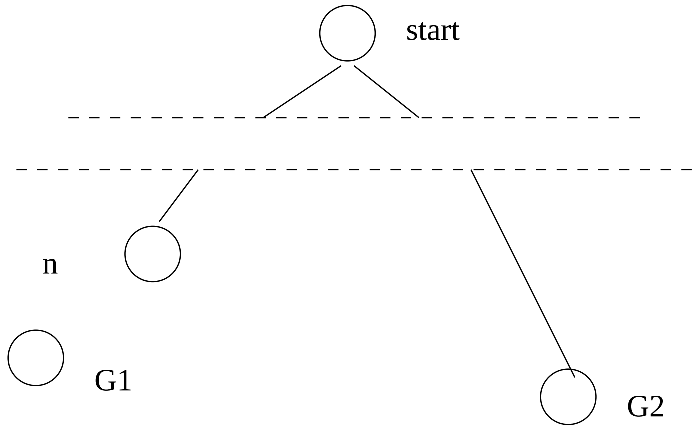


Optimality of A*

Theorem: A* search is optimal

Suppose some suboptimal goal G_2 has been generated and is in the queue. Let n be an unexpanded node on a shortest path to an optimal goal G_1.

Proof for the optimality of A*



$$
\begin{aligned}
f(G_2) &= g(G_2) && \text{since } h(G_2) = 0 \\
&> g(G_1) && \text{since } G_2 \text{ is suboptimal} \\
&\geq f(n) && \text{since } h \text{ is admissible}
\end{aligned}
$$

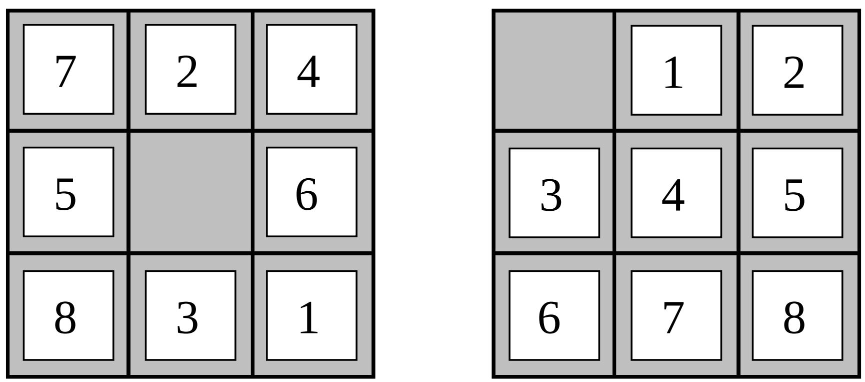
Since $f(G_2) > f(n)$, A* will never select G_2 for expansion

Properties of A*

- Complete Yes, unless there are infinitely many nodes with $f \leq f(G)$
- Time Exponential in (relative error in $h \times$ length of solution)
- Space Keeps all nodes in memory
- Optimal Yes—cannot expand f_{i+1} until f_i is finished
  - A* expands all nodes with $f(n) < C^*$
  - A* expands some nodes with $f(n) = C^*$
  - A* expands no nodes with $f(n) > C^*$

Admissible heuristics

E.g., for the 8-puzzle:
$h_1(n)$ = number of misplaced tiles
$h_2(n)$ = total *Manhattan* distance
(i.e., no. of squares from desired location of each tile)



Start State   Goal State

$h_1(S)$ = ??
$h_2(S)$ = ??

Dominance

If $h_2(n) \geq h_1(n)$ for all n (both admissible)
then h_2 *dominates* h_1 and is better for search

Typical search costs:

$d = 14$	IDS = 3,473,941 nodes
	A$^*(h_1)$ = 539 nodes
	A$^*(h_2)$ = 113 nodes
$d = 24$	IDS $\approx$ 54,000,000,000 nodes
	A$^*(h_1)$ = 39,135 nodes
	A$^*(h_2)$ = 1,641 nodes

Relaxed problems

- Admissible heuristics can be derived from the *exact* solution cost of a *relaxed* version of the problem
- If the rules of the 8-puzzle are relaxed so that a tile can move *anywhere*, then $h_1(n)$ gives the shortest solution
- If the rules are relaxed so that a tile can move to *any adjacent square*, then $h_2(n)$ gives the shortest solution
- Key point: the optimal solution cost of a relaxed problem is no greater than the optimal solution cost of the real problem

Iterative Deepening A* (IDA*)

- Idea: perform iterations of DFS. The cutoff is defined based on the f-cost rather than the depth of a node.
- Each iteration expands all nodes inside the contour for the current f-cost, peeping over the contour to find out where the contour lies.

Iterative Deepening A* (IDA*)

function IDA* (*problem*)
returns a solution sequence

inputs: *problem*, a problem
local variables:
f-limit, the current f-COST limit
root, a node

root $\leftarrow$ MAKE-NODE(INITIAL-STATE[*problem*])
f-limit $\leftarrow$ f-COST(*root*)
loop do
solution, f-limit $\leftarrow$ DFS-CONTOUR(*root, f-limit*)
if *solution* is non-null **then return** *solution*
if *f-limit* = ∞ **then return** failure

Iterative Deepening A* (IDA*)

function DFS-CONTOUR (*node, f-limit*)
returns a solution sequence and a new f-COST limit

inputs: *node*, a node
f-limit, the current f-COST limit
local variables:
next-f, the f-COST limit for the next contour, initally ∞

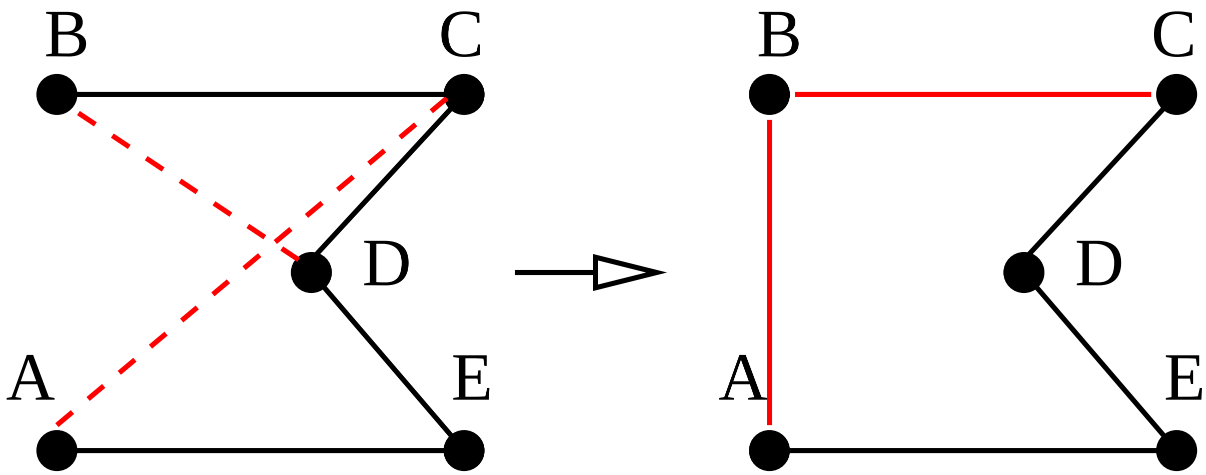
if f-COST[*node*] $>$ *f-limit* **then return** null, f-COST[*node*]
if GOAL-TEST[*problem*](STATE[*node*]) **then return** *node, f-limit*
for each node s **in** SUCCESSORS(*node*) **do**
solution, new-f $\leftarrow$ DFS-CONTOUR(*s, f-limit*)
if *solution* is non-null **then return** *solution, f-limit*
next-f $\leftarrow$ MIN(*next-f, new-f*)
return null, *next-f*

Iterative improvement algorithms

- In many optimization problems, the path is irrelevant; the goal state itself is the solution
- Then state space = set of “complete” configurations; find optimal configuration, e.g., TSP or, find configuration satisfying constraints, e.g., timetable
- In such cases, can use *iterative improvement* algorithms; keep a single “current” state, try to improve it
- Constant space, suitable for online as well as offline search

Example: Travelling Salesperson Problem

Start with any complete tour, perform pairwise exchanges.



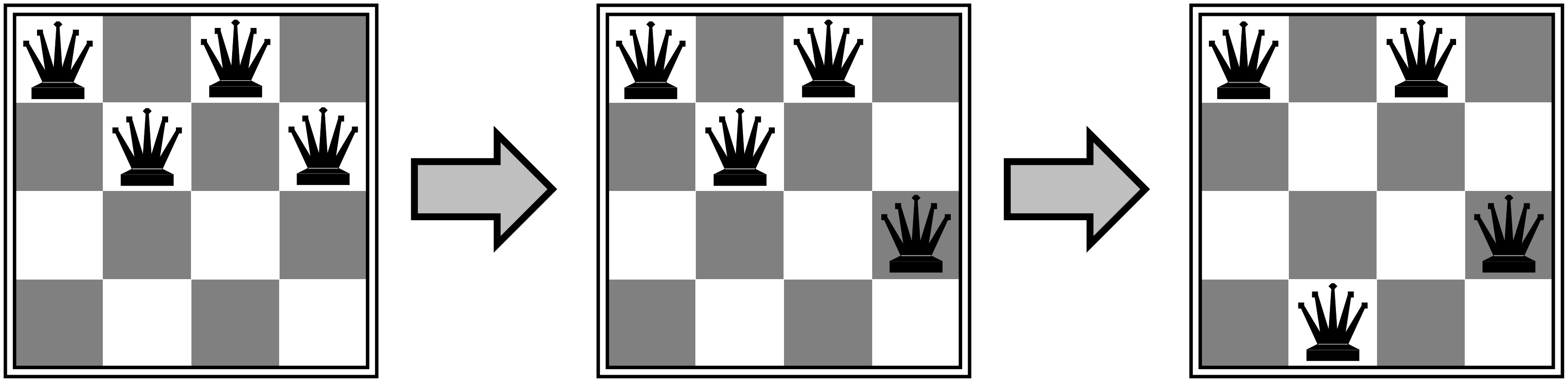
A C B D E   A B C D E

Variants of this approach get within 1% of optimal very quickly with thousands of cities.

Example: n-queens

Put n queens on an $n \times n$ board with no two queens on the same row, column, or diagonal

Move a queen to reduce number of conflicts



Almost always solves n-queens almost instantenously for very large n, e.g., $n = 1$ million.

Hill-climbing (or gradient ascent/descent)

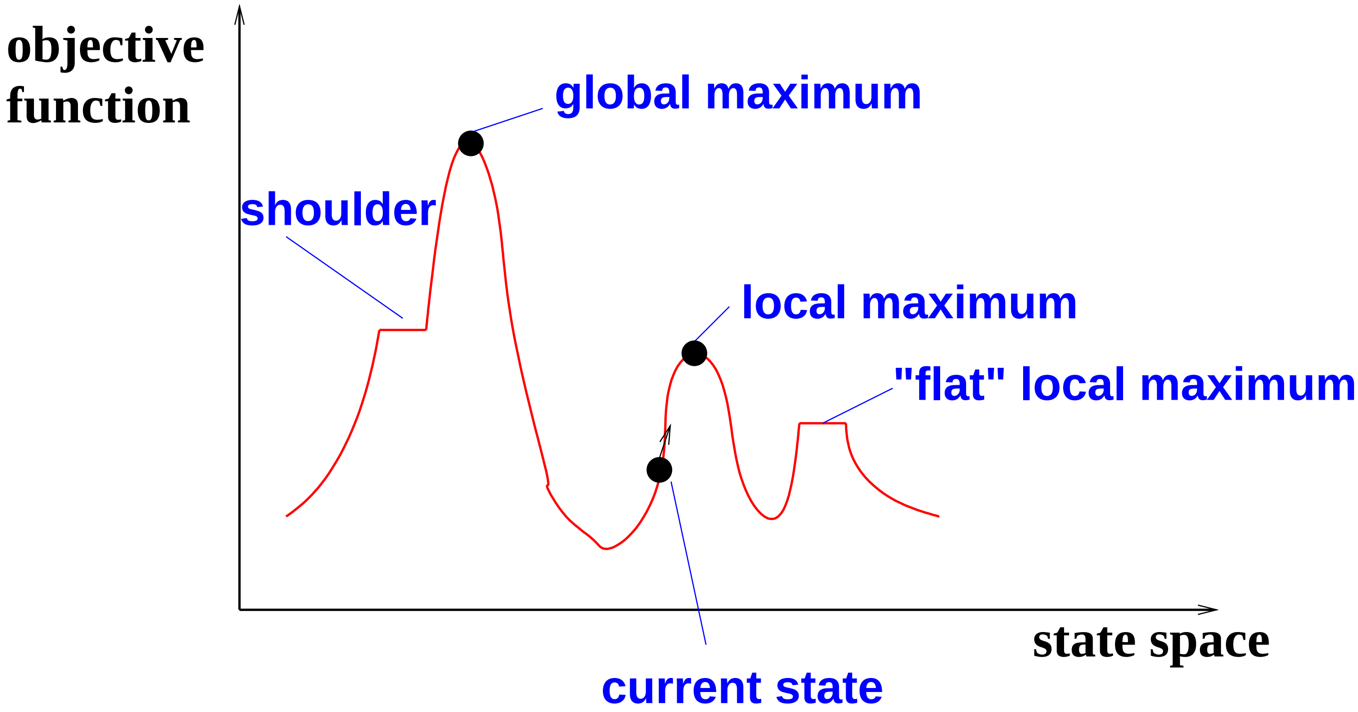
function HILL-CLIMBING (*problem*)
returns a state that is a local maximum

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*

Hill-climbing (cont'd)

- "Like climbing Everest in thick fog with amnesia"
- Problem: depending on initial state, can get stuck on local maxima



Local Search Algorithms

- Hill Climbing
  - stochastic: choose randomly from uphill moves
  - first-choice: generate successors randomly one-by-one until one better than the current state is found
  - random-restart: restart with a randomly generated initial state
  - In continuous spaces, problems w/ choosing step size, slow convergence
- Simulated annealing : escape local maxima by allowing some “bad” moves (with predefined probabilities)
  but gradually decrease their size and frequency

Local Search Algorithms (cont'd)

- Local beam search : keeps k states rather than 1; choose top k of their successors
  - not the same as k searches run in parallel searches that find good states recruit other searches to join them
  - often, all k states end up on the same local hill
  - idea: choose k successors randomly, biased towards good ones
- Genetic algoritms: keeps a population and generates children from two parents
  - fitness function
  - cross-over, mutation

Summary

- Heuristic search algorithms
- Local search algorithms
- Can think about speed of the search in addition to: time & space complexity, optimality, completeness