

Revised by Hankui Zhuo, March 15, 2019

INFORMED SEARCH ALGORITHMS & HILL-CLIMBING & SIMULATED ANNEALING

CHAPTER 4

Outline

- ◇ Best-first search
- ◇ A^* search
- ◇ Heuristics
- ◇ Hill-climbing
- ◇ Simulated annealing
- ◇ Genetic algorithms (briefly)
- ◇ Local search in continuous spaces (very briefly)

Review: Tree search

```
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)
```

A strategy is defined by picking the **order of node expansion**

Best-first search

Idea: use an **evaluation function** for each node
– estimate of “desirability”

⇒ Expand most desirable unexpanded node

Implementation:

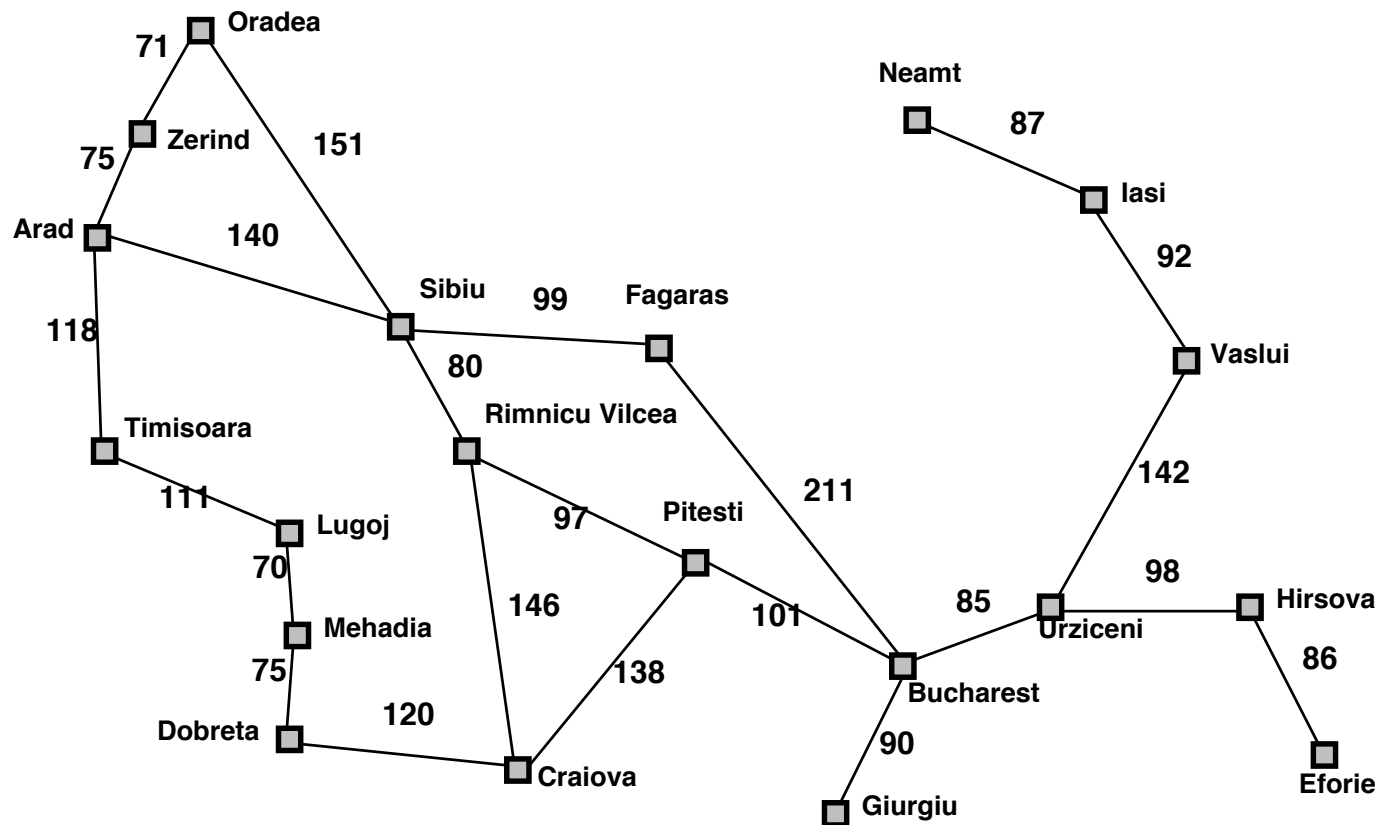
fringe is a queue sorted in decreasing order of desirability

Special cases:

greedy search

A* search

Romania with step costs in km



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 search

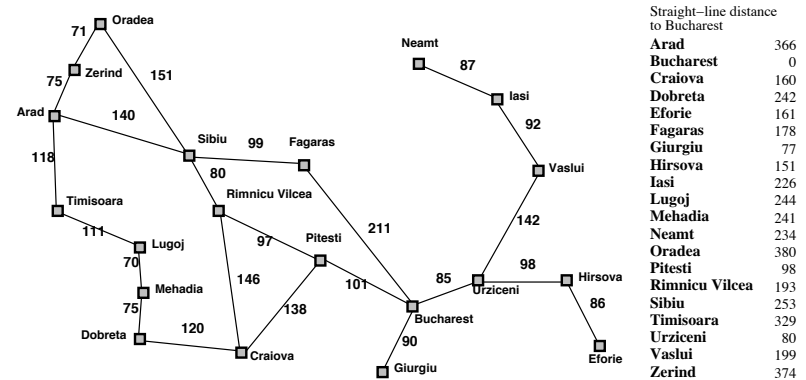
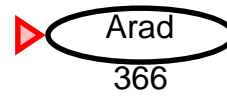
Evaluation function $h(n)$ (**h**euristic)

= estimate of cost from n to the closest goal

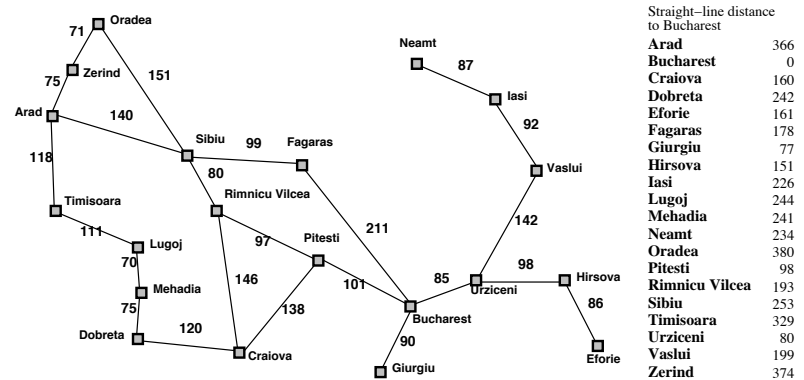
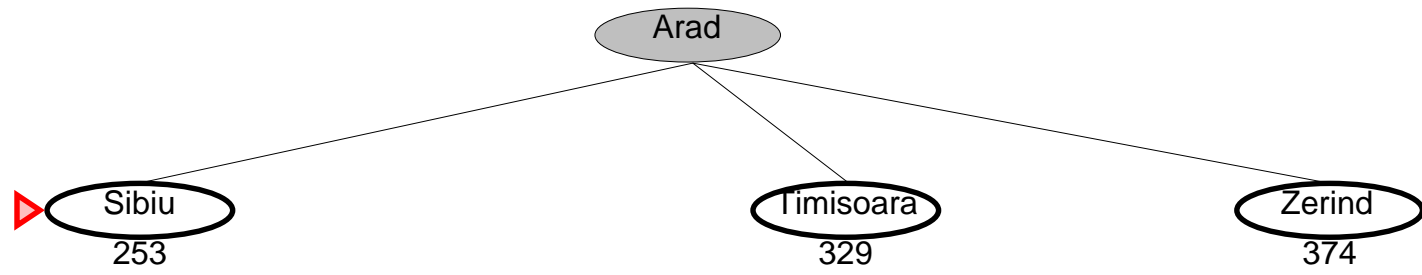
E.g., $h_{\text{SLD}}(n)$ = straight-line distance from n to Bucharest

Greedy search expands the node that **appears** to be closest to goal

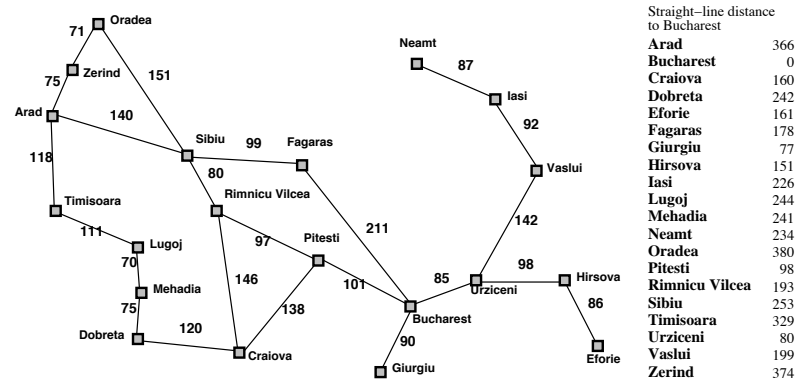
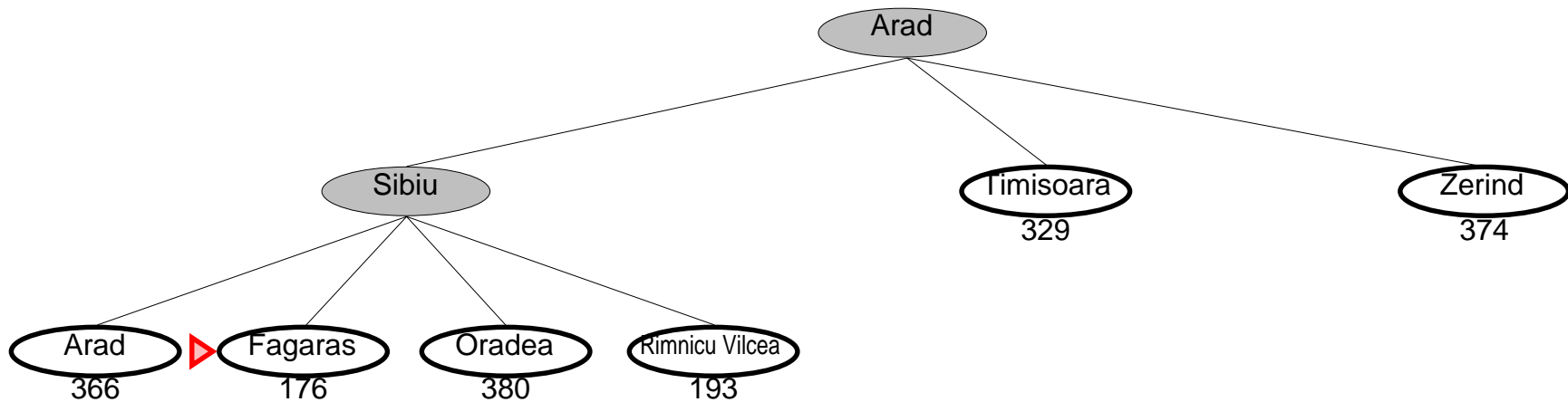
Greedy search example



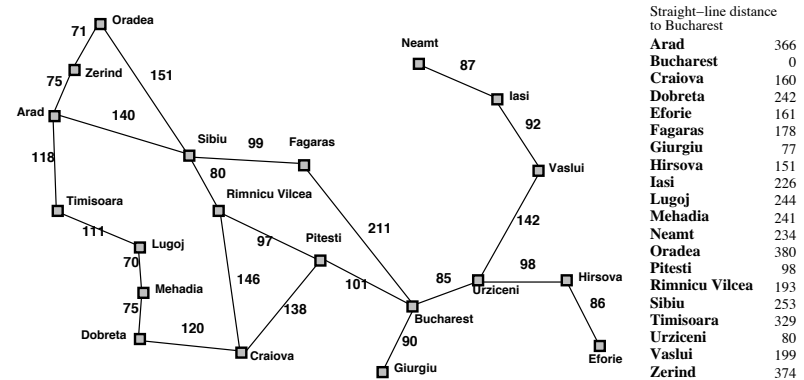
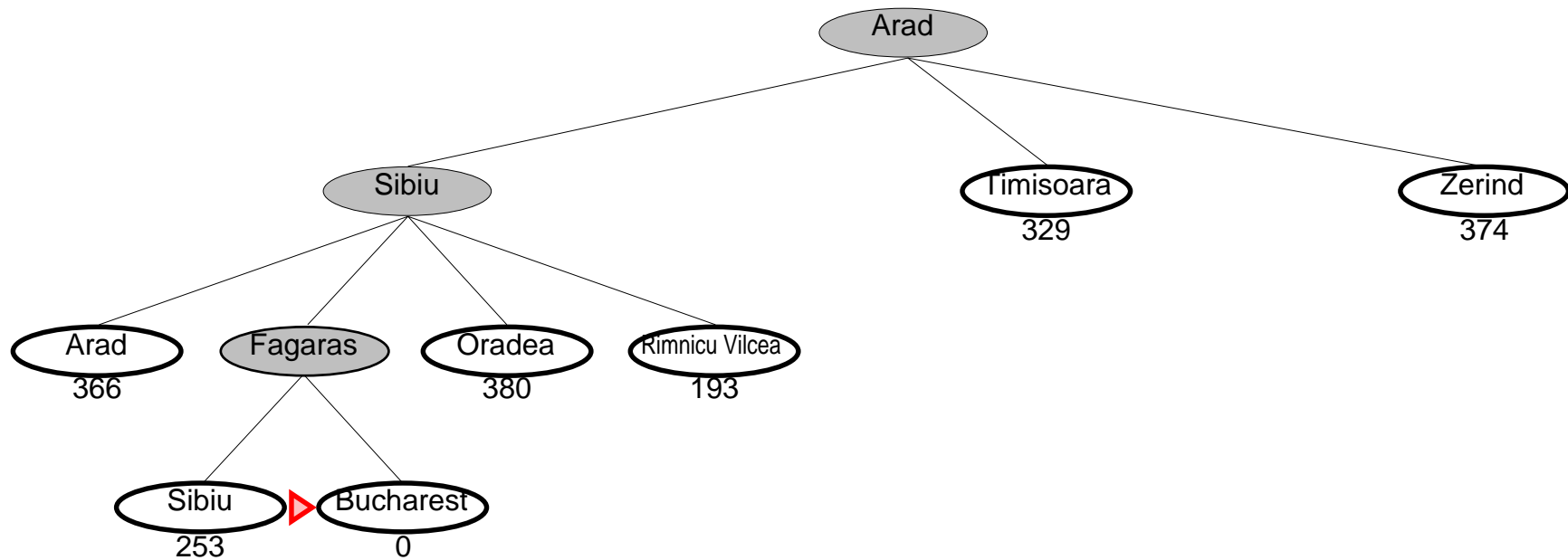
Greedy search example



Greedy search example



Greedy search example



Properties of greedy search

Complete??

Properties of greedy search

Complete?? No—can get stuck in loops, e.g., with Oradea as goal,

lasi \rightarrow Neamt \rightarrow lasi \rightarrow Neamt \rightarrow

Complete in finite space with repeated-state checking

Time??

Properties of greedy search

Complete?? No—can get stuck in loops, e.g.,

lasi \rightarrow Neamt \rightarrow lasi \rightarrow Neamt \rightarrow

Complete in finite space with repeated-state checking

Time?? $O(b^m)$, but a good heuristic can give dramatic improvement

Space??

Properties of greedy search

Complete?? No—can get stuck in loops, e.g.,

lasi \rightarrow Neamt \rightarrow lasi \rightarrow Neamt \rightarrow

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??

Properties of greedy search

Complete?? No—can get stuck in loops, e.g.,

lasi \rightarrow Neamt \rightarrow lasi \rightarrow Neamt \rightarrow

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 .

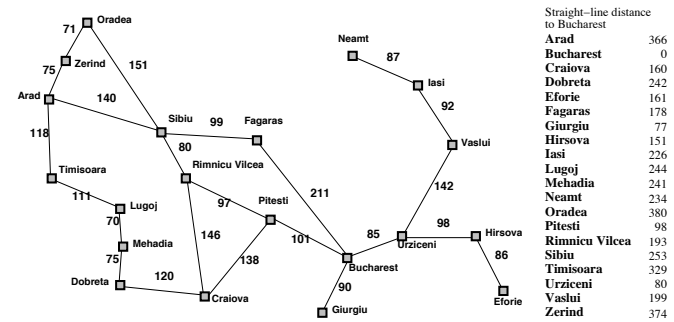
(Also require $h(n) \geq 0$, so $h(G) = 0$ for any goal G .)

E.g., $h_{\text{SLD}}(n)$ never overestimates the actual road distance

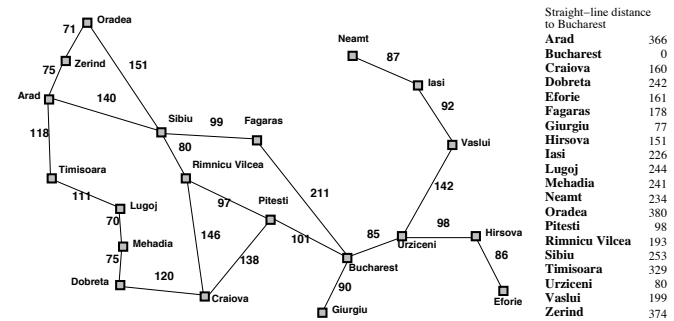
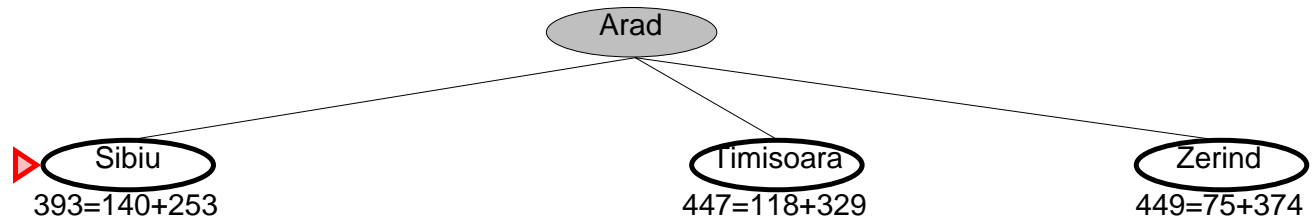
Theorem: A* search is optimal

A* search example

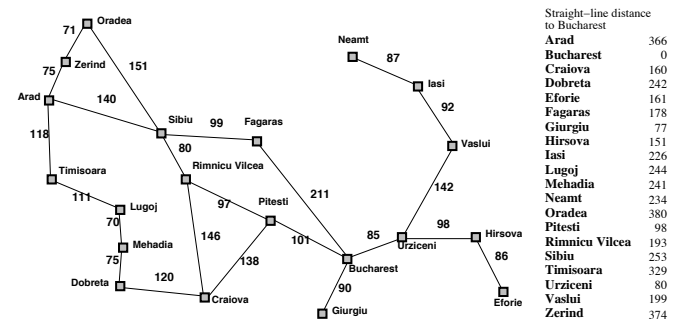
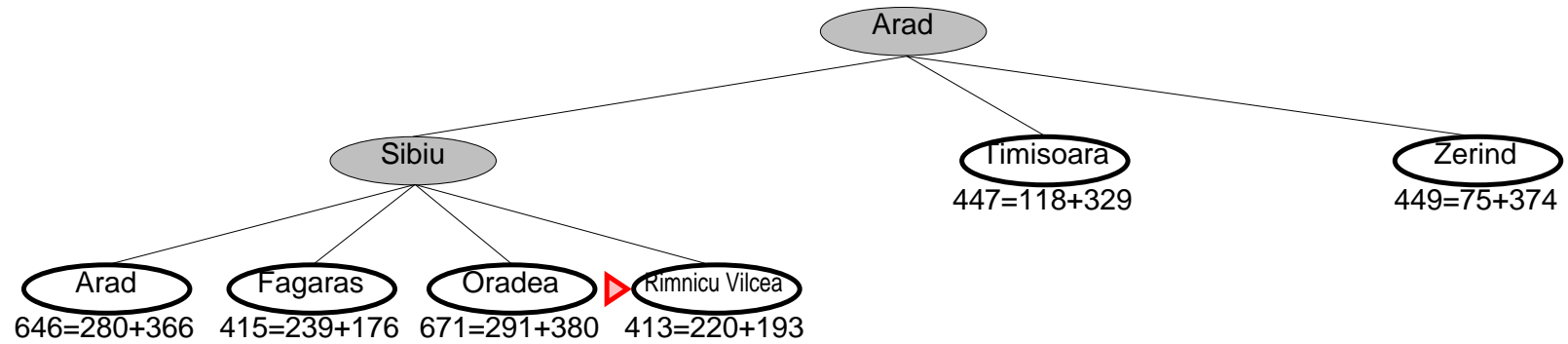
▶ Arad
366=0+366



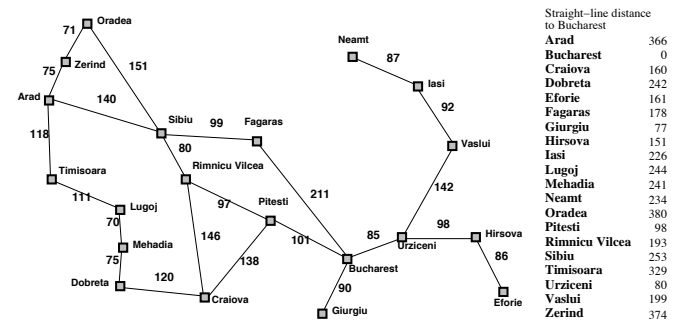
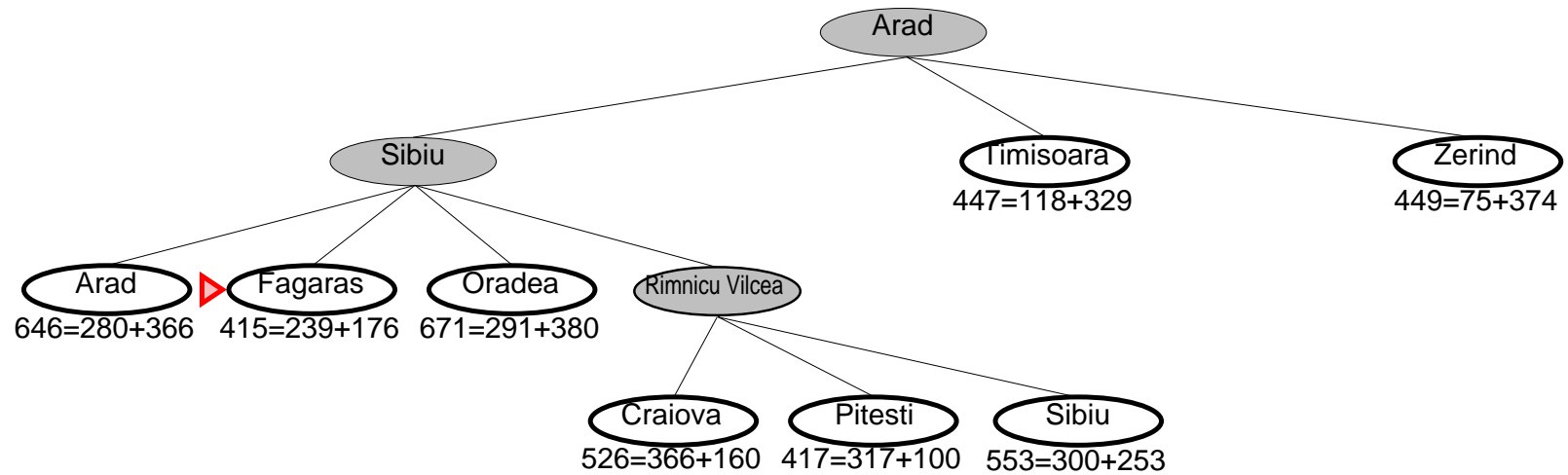
A* search example



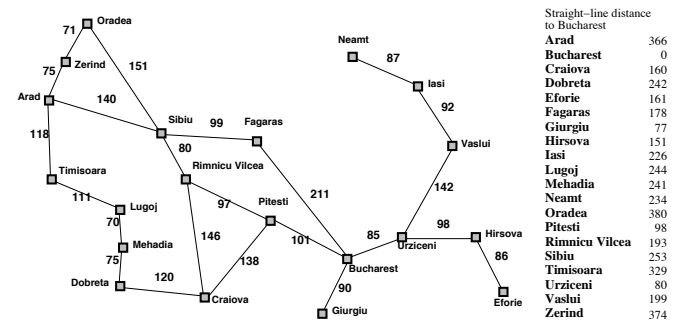
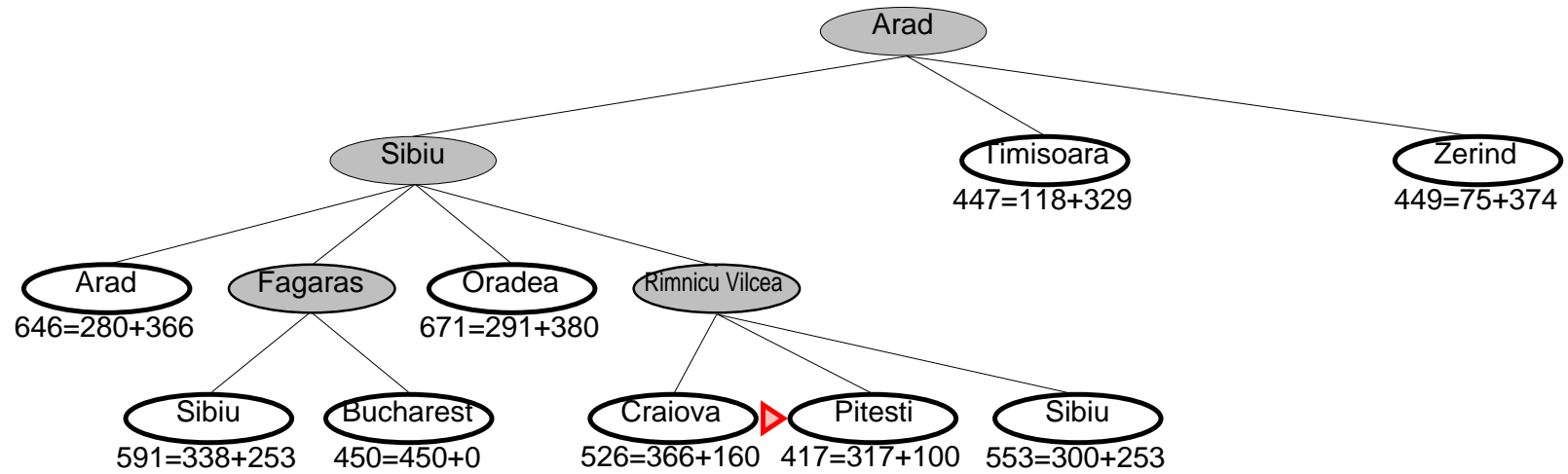
A* search example



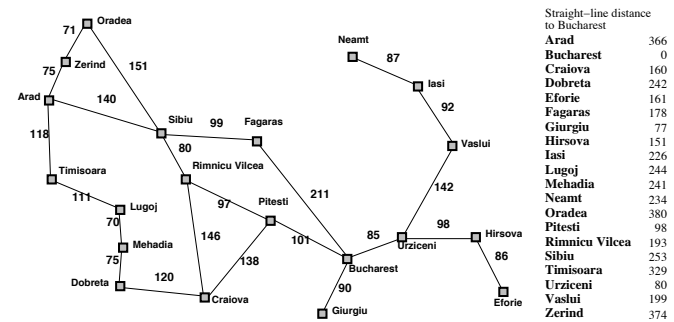
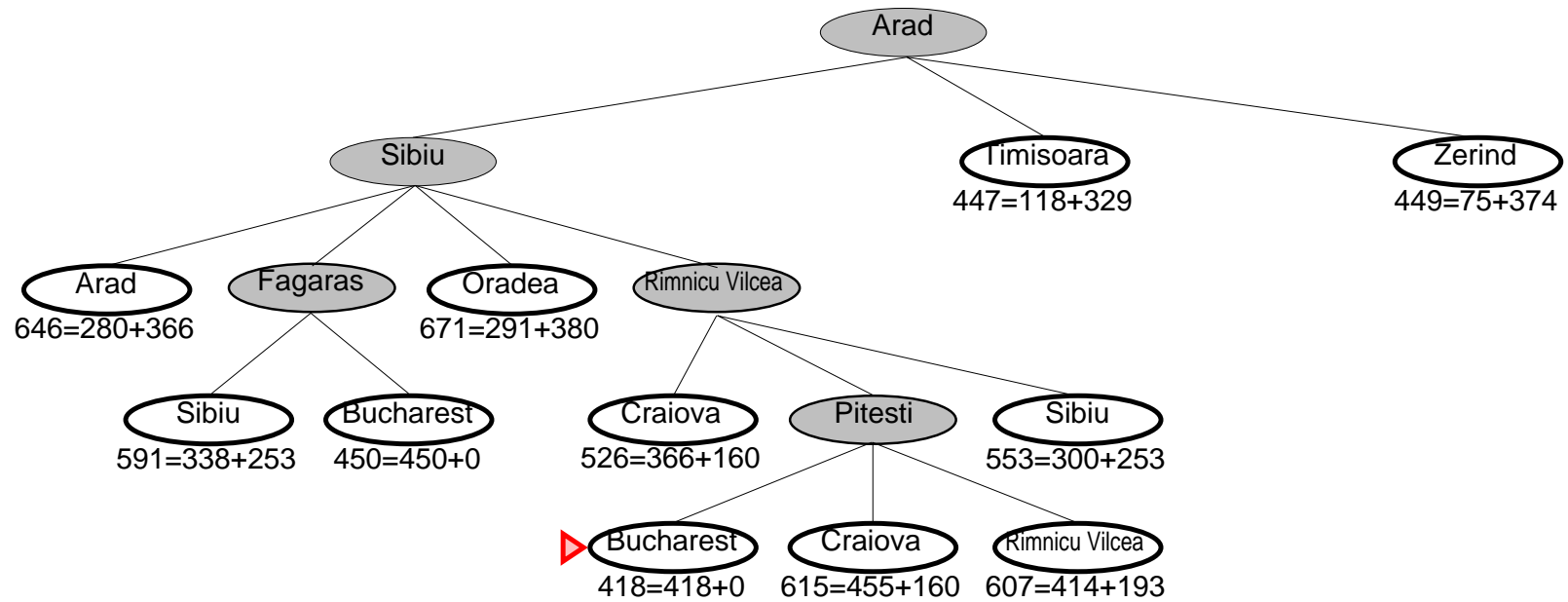
A* search example



A* search example

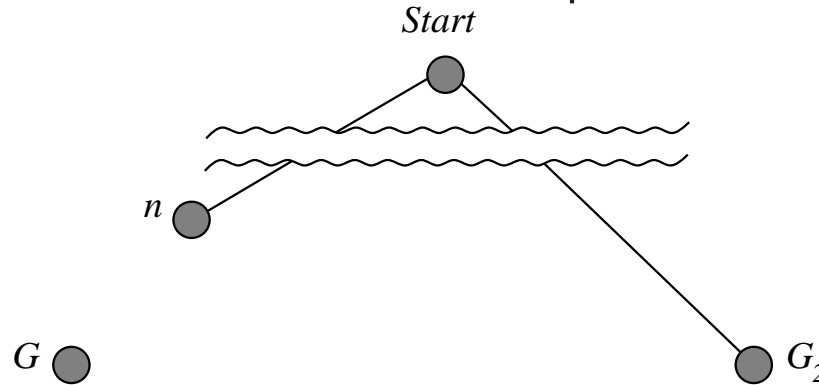


A* search example



Optimality of A^* (standard proof)

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 .



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

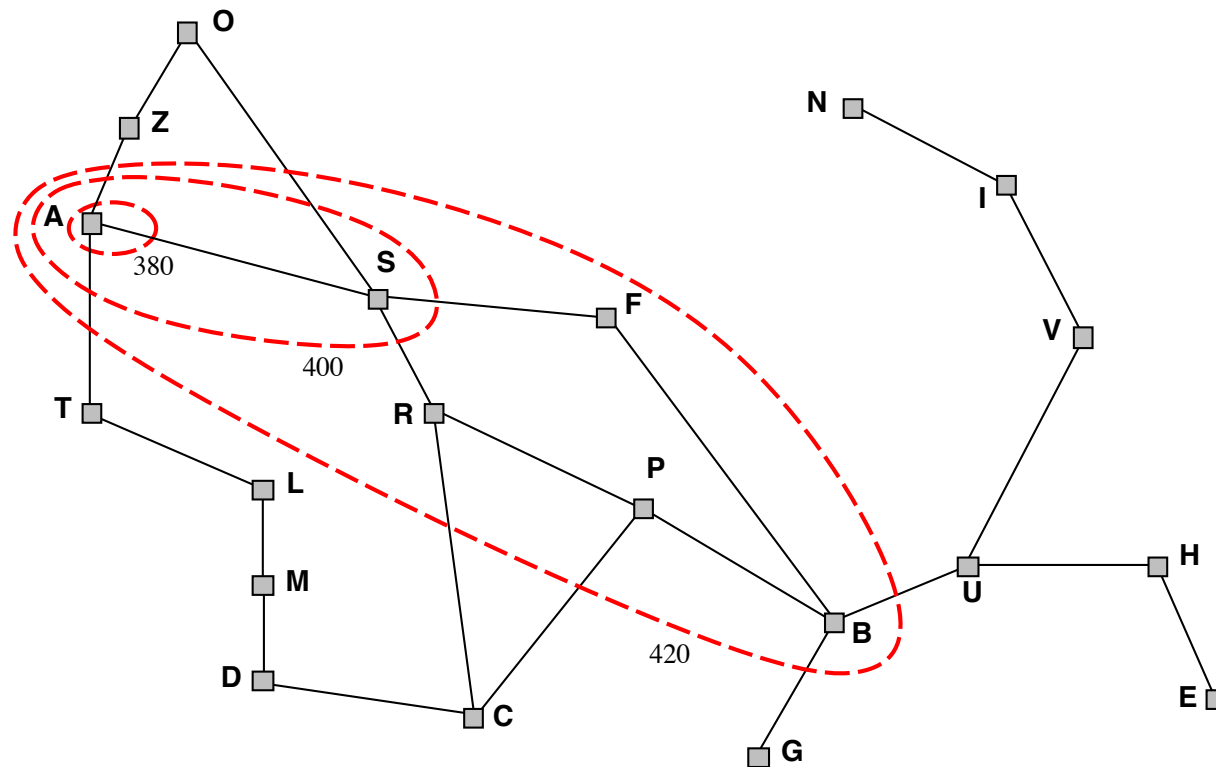
Since $f(G_2) > f(n)$, A^* will never select G_2 for expansion

Optimality of A^* (more useful)

Lemma: A^* expands nodes in order of increasing f value*

Gradually adds “ f -contours” of nodes (cf. breadth-first adds layers)

Contour i has all nodes with $f = f_i$, where $f_i < f_{i+1}$

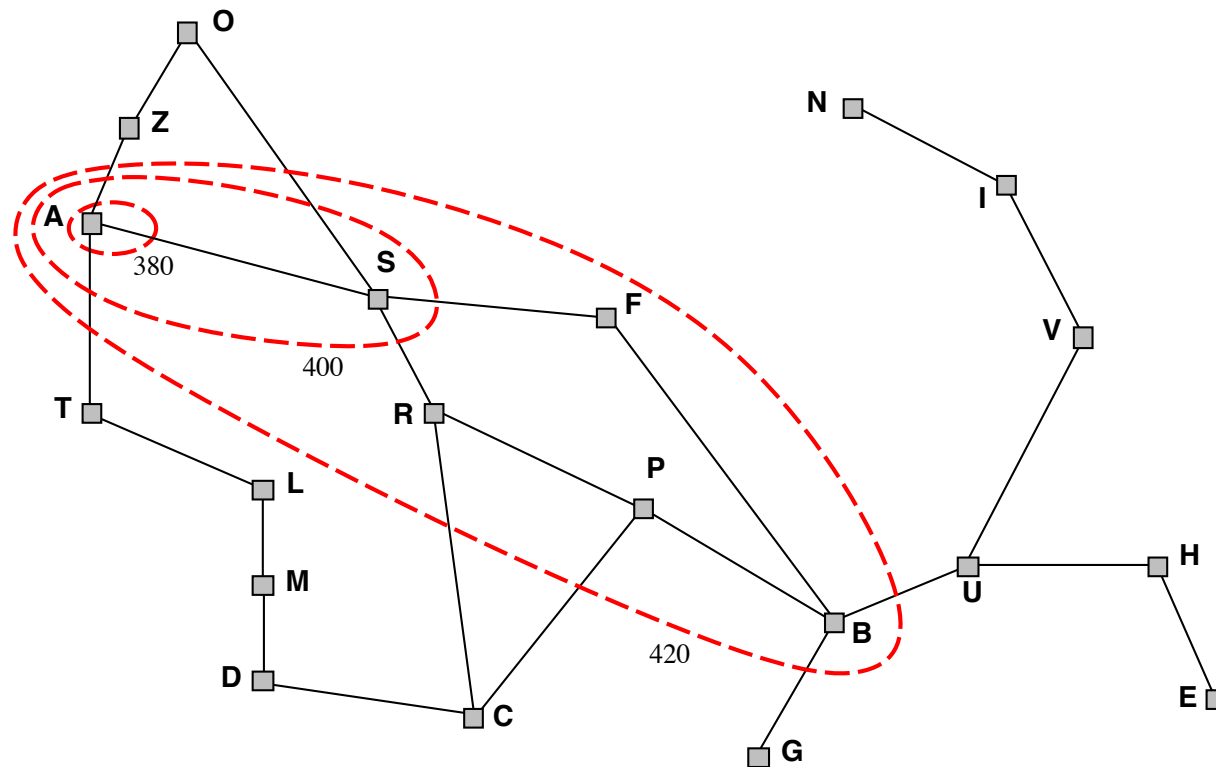


Optimality of A^* (more useful)

Lemma: A^* expands nodes in order of increasing f value* Homework: prove it??

Gradually adds “ f -contours” of nodes (cf. breadth-first adds layers)

Contour i has all nodes with $f = f_i$, where $f_i < f_{i+1}$



Properties of A^*

Complete??

Properties of A^*

Complete?? Yes, unless there are infinitely many nodes with $f \leq f(G)$

Time??

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 soln.]

Space??

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 soln.]

Space?? Keeps all nodes in memory

Optimal??

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 soln.]

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 $f(n) > C^*$

Proof of lemma: Consistency

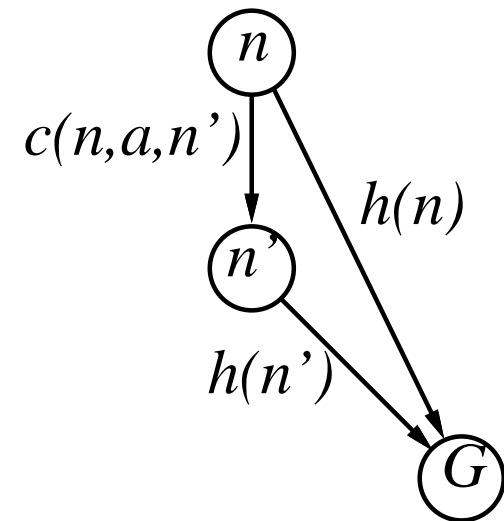
A heuristic is **consistent** if

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

If h is consistent, we have

$$\begin{aligned} f(n') &= g(n') + h(n') \\ &= g(n) + c(n, a, n') + h(n') \\ &\geq g(n) + h(n) \\ &= f(n) \end{aligned}$$

I.e., $f(n)$ is nondecreasing along any path.



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)

7	2	4
5		6
8	3	1

Start State

1	2	3
4	5	6
7	8	

Goal State

$$h_1(S) = ??$$

$$h_2(S) = ??$$

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)

7	2	4
5		6
8	3	1

Start State

1	2	3
4	5	6
7	8	

Goal State

$$h_1(S) = ?? \quad 6$$

$$h_2(S) = ?? \quad 4+0+3+3+1+0+2+1 = 14$$

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

Given any admissible heuristics h_a, h_b ,

$$h(n) = \max(h_a(n), h_b(n))$$

is also admissible and dominates h_a, h_b

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

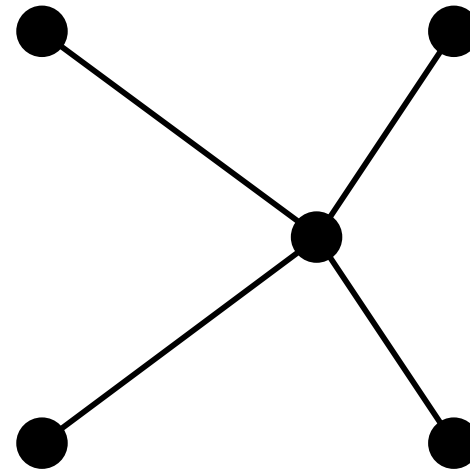
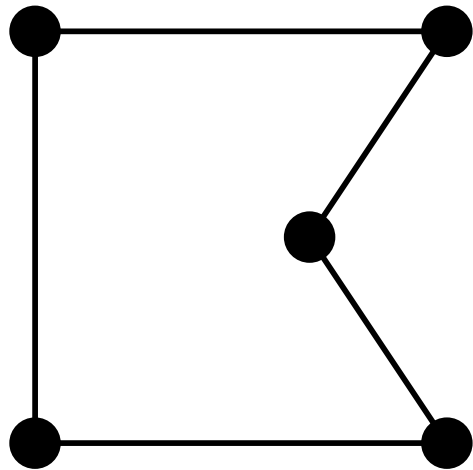
If the rules are relaxed so that a tile can move to **any adjacent square**, then $h_2(n)$ gives the shortest solution

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

Relaxed problems contd.

Well-known example: travelling salesperson problem (TSP)

Find the shortest tour visiting all cities exactly once



Minimum spanning tree can be computed in $O(n^2)$
and is a lower bound on the shortest (open) tour

Summary

Heuristic functions estimate costs of shortest paths

Good heuristics can dramatically reduce search cost

Greedy best-first search expands lowest h

- incomplete and not always optimal

A* search expands lowest $g + h$

- complete and optimal
- also optimally efficient (up to tie-breaks, for forward search)

Admissible heuristics can be derived from exact solution of relaxed problems

Iterative improvement algorithms

In many optimization problems, **path** is irrelevant;
the goal state itself is the solution

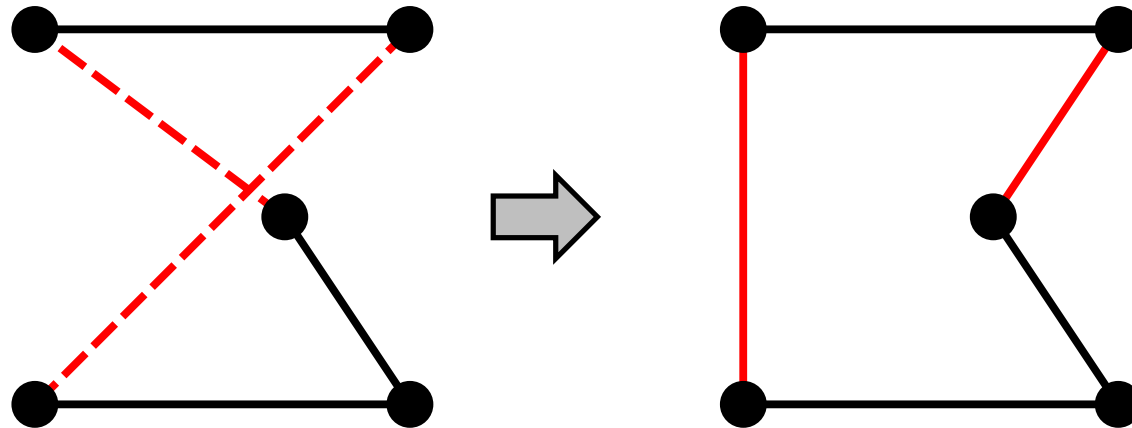
Then state space = set of “complete” configurations;
find **optimal** configuration, e.g., TSP
or, find configuration satisfying constraints, e.g., timetable

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

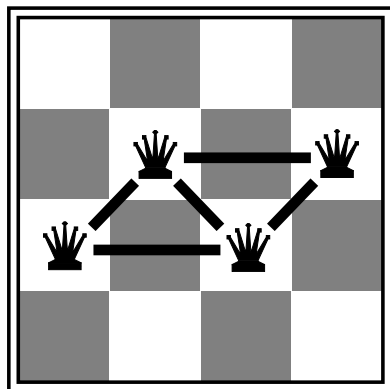


Variants of this approach get within 1% of optimal very quickly with thousands of cities

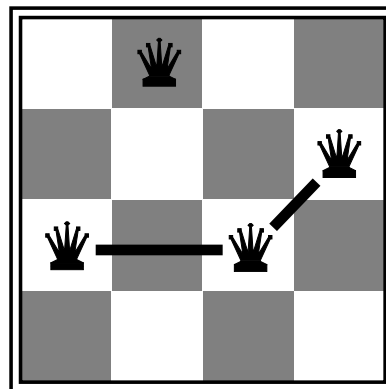
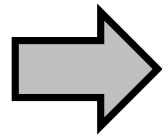
Example: n -queens

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

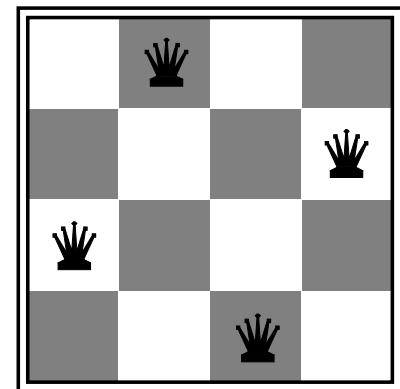
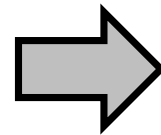
Move a queen to reduce number of conflicts



$h = 5$



$h = 2$



$h = 0$

Almost always solves n -queens problems almost instantaneously for very large n , e.g., $n = 1\text{million}$

Hill-climbing (or gradient ascent/descent)

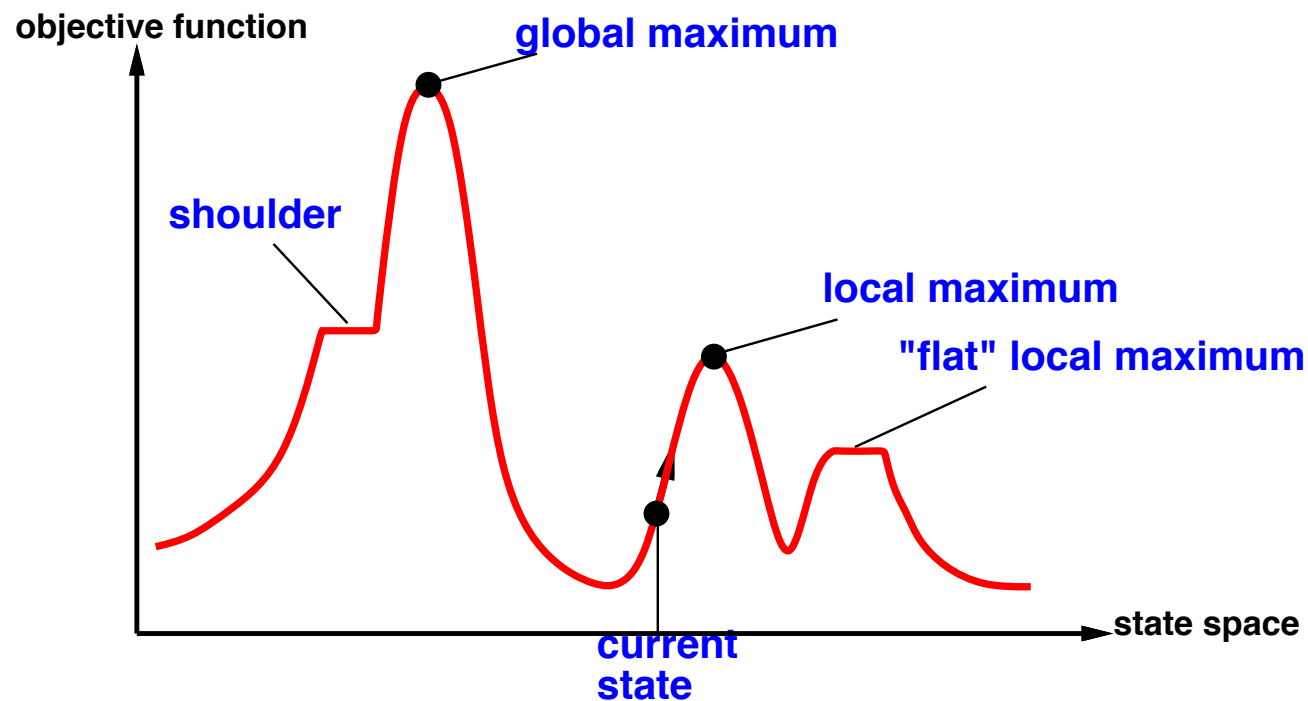
“Like climbing Everest in thick fog with amnesia”

```
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 contd.

Useful to consider state space landscape



Random-restart hill climbing overcomes local maxima—trivially complete

Random sideways moves 😊 escape from shoulders 😞 loop on flat maxima

Simulated annealing

Idea: escape local maxima by allowing some “bad” moves
but gradually decrease their size and frequency

```
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
     $\Delta E \leftarrow \text{VALUE}[\textit{next}] - \text{VALUE}[\textit{current}]$ 
    if  $\Delta E > 0$  then current ← next
    else current ← next only with probability  $e^{\Delta E/T}$ 
```

Properties of simulated annealing

At fixed “temperature” T , state occupation probability reaches Boltzman distribution

$$p(x) = \alpha e^{-\frac{E(x)}{kT}}$$

T decreased slowly enough \implies always reach best state x^*
because $e^{-\frac{E(x^*)}{kT}} / e^{-\frac{E(x)}{kT}} = e^{\frac{E(x^*) - E(x)}{kT}} \gg 1$ for small T

Is this necessarily an interesting guarantee??

Devised by Metropolis et al., 1953, for physical process modelling

Widely used in VLSI layout, airline scheduling, etc.

Local beam search

Idea: keep k states instead of 1; choose top k of all their successors

Not the same as k searches run in parallel!

Searches that find good states recruit other states to join them

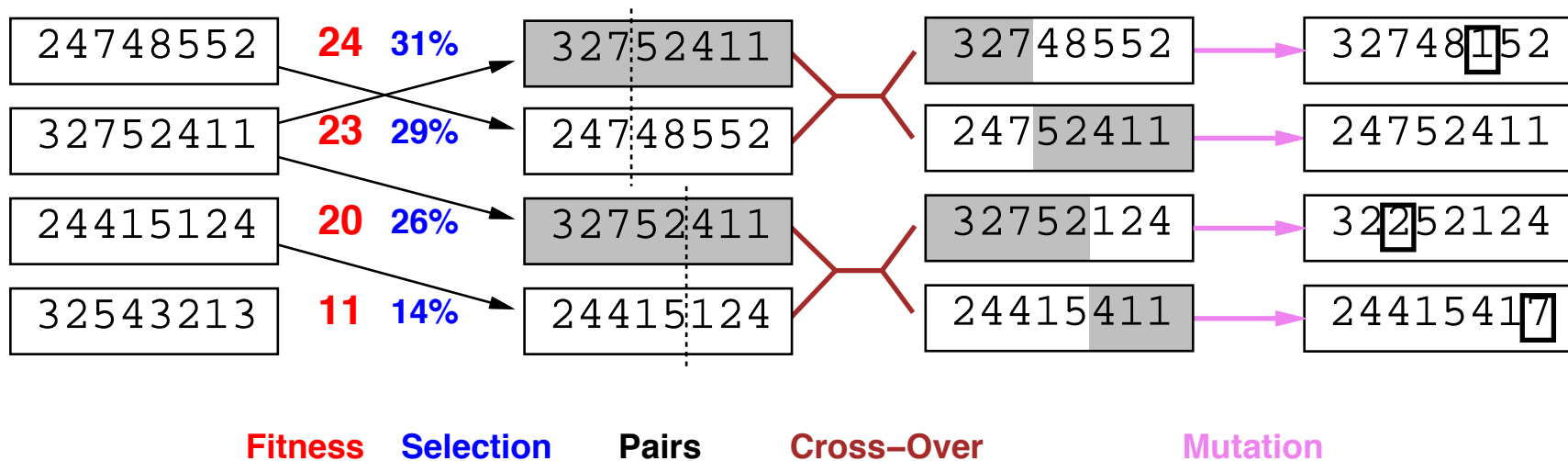
Problem: quite often, all k states end up on same local hill

Idea: choose k successors randomly, biased towards good ones

Observe the close analogy to natural selection!

Genetic algorithms

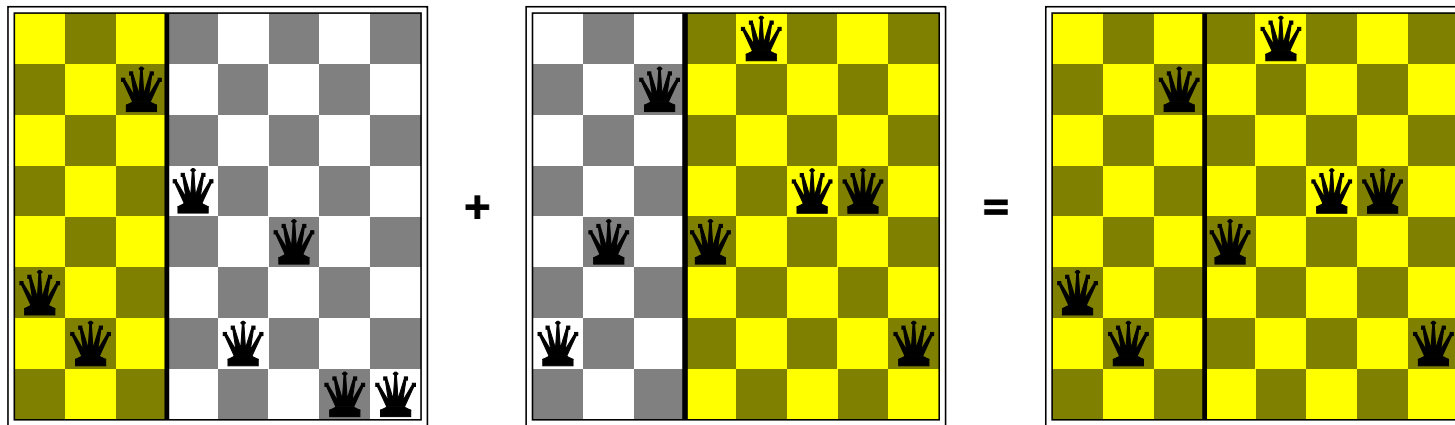
= stochastic local beam search + generate successors from **pairs** of states



Genetic algorithms contd.

GAs require states encoded as strings (GPs use programs)

Crossover helps **iff substrings are meaningful components**



GAs \neq evolution: e.g., real genes encode replication machinery!

Continuous state spaces

Suppose we want to site three airports in Romania:

- 6-D state space defined by $(x_1, y_1), (x_2, y_2), (x_3, y_3)$
- objective function $f(x_1, y_1, x_2, y_2, x_3, y_3) =$
sum of squared distances from each city to nearest airport

Discretization methods turn continuous space into discrete space, e.g., **empirical gradient** considers $\pm\delta$ change in each coordinate

Gradient methods compute

$$\nabla f = \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial y_1}, \frac{\partial f}{\partial x_2}, \frac{\partial f}{\partial y_2}, \frac{\partial f}{\partial x_3}, \frac{\partial f}{\partial y_3} \right)$$

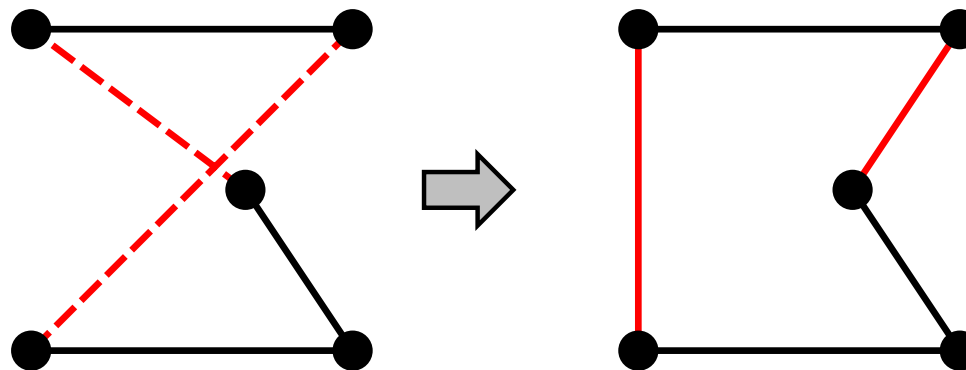
to increase/reduce f , e.g., by $\mathbf{x} \leftarrow \mathbf{x} + \alpha \nabla f(\mathbf{x})$

Sometimes can solve for $\nabla f(\mathbf{x}) = 0$ exactly (e.g., with one city).

Newton–Raphson (1664, 1690) iterates $\mathbf{x} \leftarrow \mathbf{x} - \mathbf{H}_f^{-1}(\mathbf{x}) \nabla f(\mathbf{x})$ to solve $\nabla f(\mathbf{x}) = 0$, where $\mathbf{H}_{ij} = \partial^2 f / \partial x_i \partial x_j$

Exercise

- ◇ How to solve the traveling salesperson problem with GA??
 - Suppose we have 1,000 cities to travel.
 - The connections among the cities are specified by $1,000 \times 1,000$ matrix M .
 - Please specify how to define strings and fitness function.



End of Chapter 4

Thanks & Questions!