Informed search algorithms

(Based on slides by Oren Etzioni, Stuart Russell)

The problem

	# Unique board configurations in search space
8-puzzle	9! = 362880
15-puzzle	16! = 20922789888000 ≈ $\mathbf{10^{13}}$
24-puzzle	25! ≈ $\mathbf{10^{25}}$
35-puzzle	36! ≈ $\mathbf{10^{41}}$
48-puzzle	49! ≈ $\mathbf{10^{63}}$
63-puzzle	64! ≈ $\mathbf{10^{89}}$

- Number of atoms in known universe ≈ $\mathbf{10^{80}}$

Outline

- Greedy best-first search
- A^* search
- Heuristics
- Local search algorithms
- Hill-climbing search
- Simulated annealing search
- Local beam search
- Genetic algorithms

Best-first search

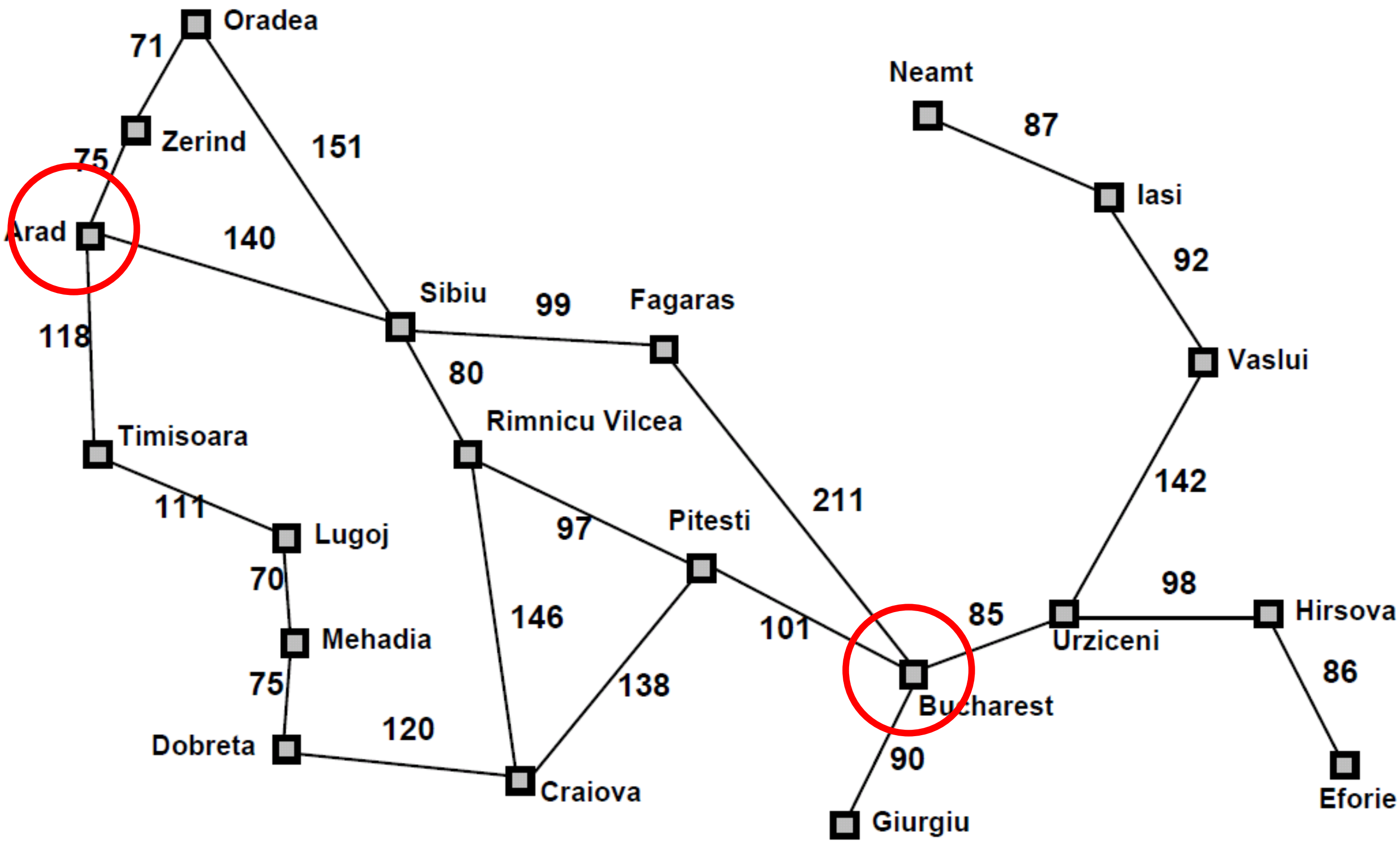
- A search strategy is defined by picking the order of node expansion
- Idea: use an evaluation function *f(n)* for each node
  - estimate of "desirability“

  → Expand most desirable unexpanded node

- Implementation:
  Order the nodes in fringe in decreasing order of desirability

- Special cases:
  - greedy best-first 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 best-first search

- Evaluation function $f(n) = h(n)$ (heuristic)
  = estimate of cost from n to *goal*

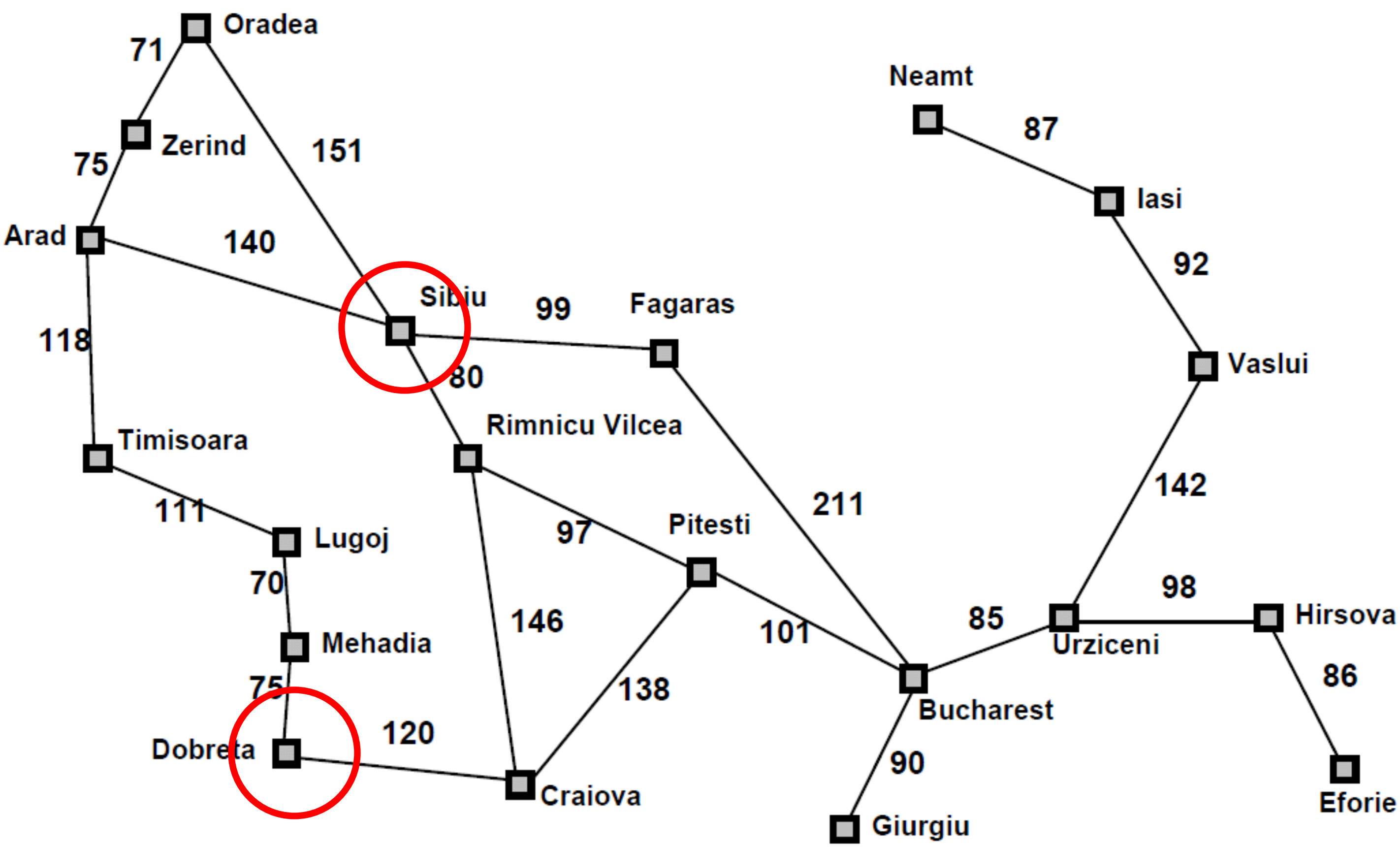
- e.g., $h_{SLD}(n)$ = straight-line distance from n to Bucharest

- Greedy best-first search expands the node that appears to be closest to goal

Properties of greedy best-first search

- Complete?
- No – can get stuck in loops, e.g., Iasi → Neamt → Iasi → Neamt →
- Time?
- $O(b^m)$, but a good heuristic can give dramatic improvement
- Space?
- $O(b^m)$ -- keeps all nodes in memory
- Optimal?
- No

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

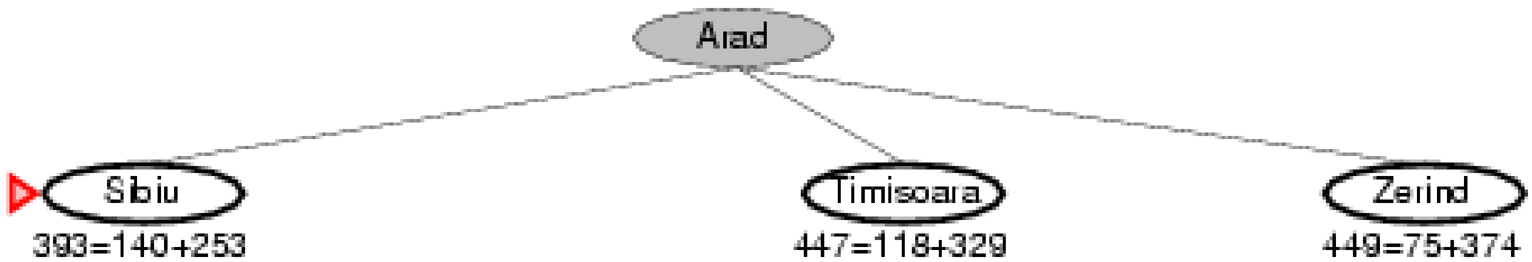
A^* search

- Idea: avoid expanding paths that are already expensive
- Evaluation function $f(n) = g(n) + h(n)$

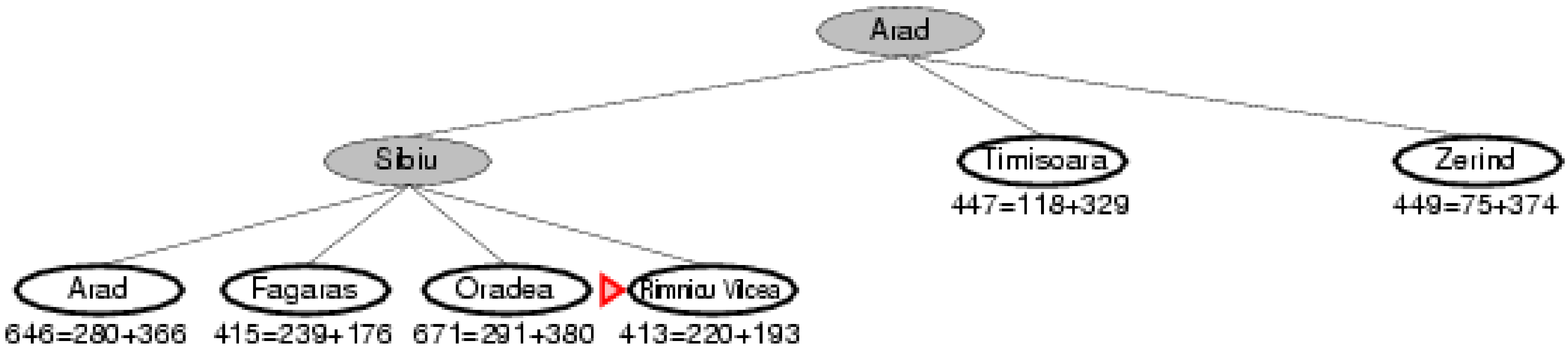
- $g(n)$ = cost so far to reach n
- $h(n)$ = estimated cost from n to goal
- $f(n)$ = estimated total cost of path through n to goal

A^* search example

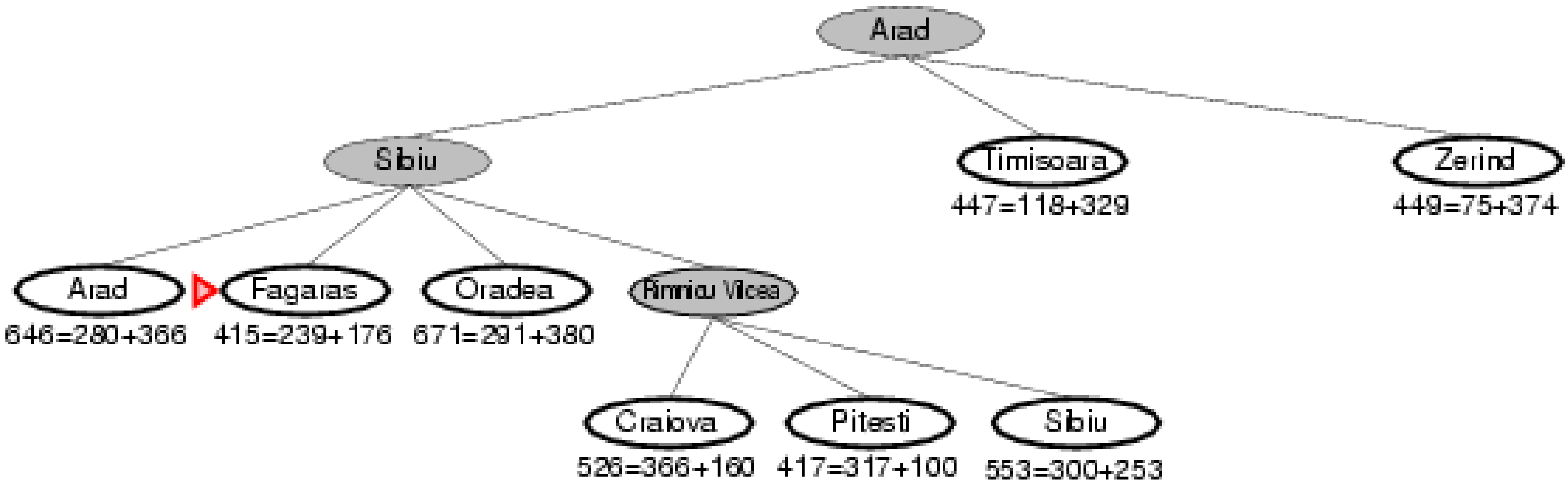
A^* search example



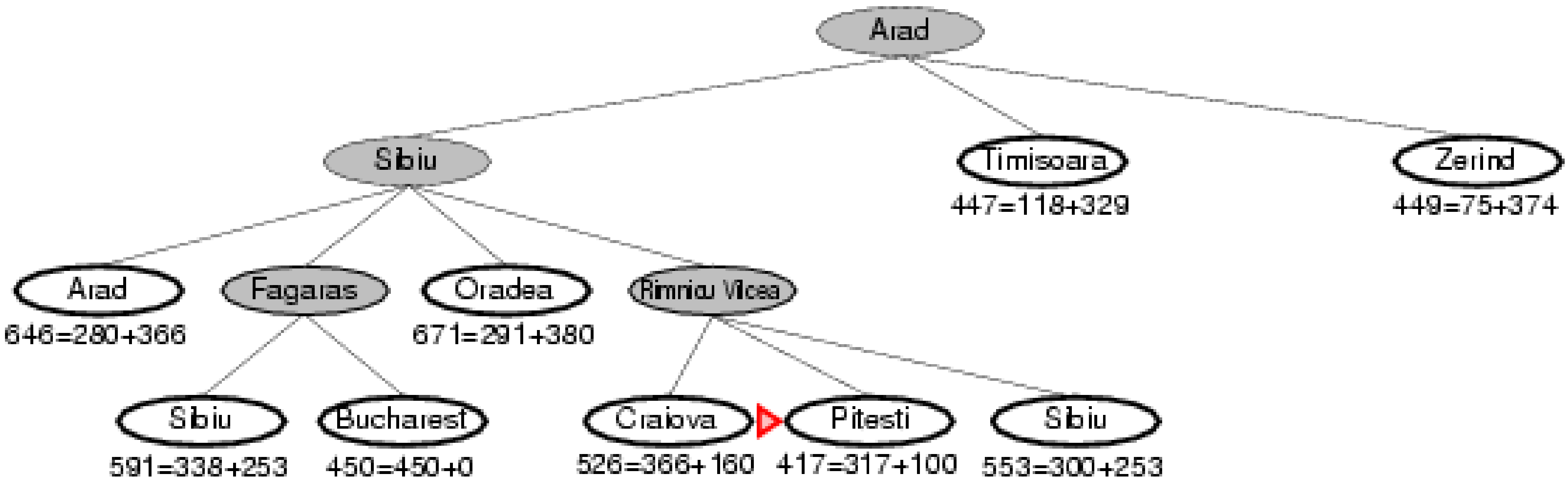
A* search example



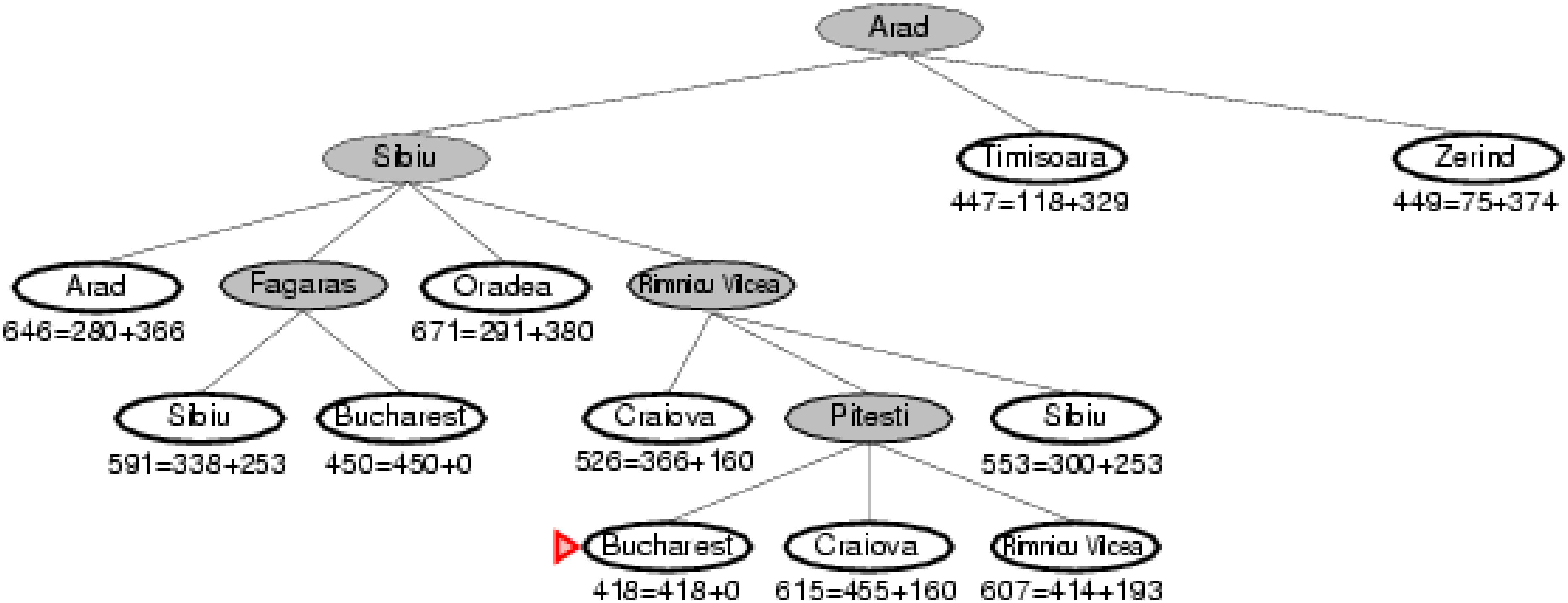
A* search example



A* search example



A* search example



Admissible heuristics

- A heuristic $h(n)$ is admissible if for every node n, $h(n) \leq h^*(n)$, where $h^*(n)$ is the true cost to reach the goal state from n.
- An admissible heuristic never overestimates the cost to reach the goal, i.e., it is optimistic
- Example: $h_{SLD}(n)$ (never overestimates the actual road distance)
- Theorem: If $h(n)$ is admissible, A^* using `TREE-SEARCH` is optimal

Properties of A*

- Complete?

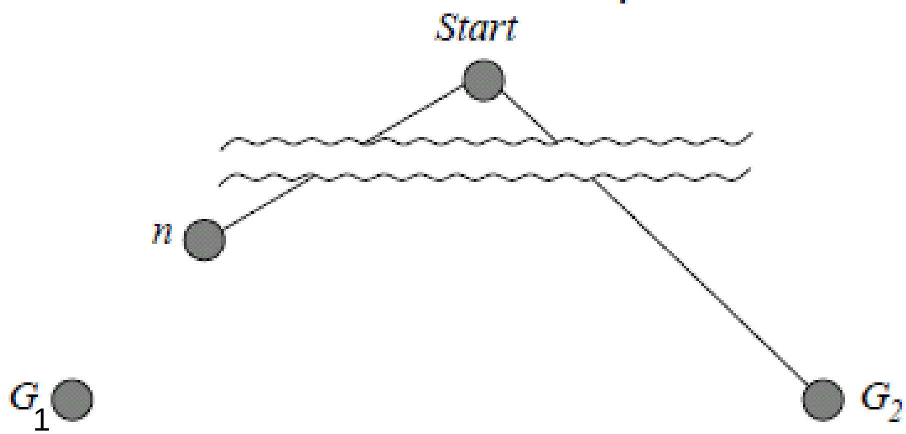
  Yes (unless there are infinitely many nodes with $f \leq f(G)$)

- Time? $O(b^m)$, but a good heuristic can give dramatic improvement
- Space? $O(b^m)$, Keeps all nodes in memory
- Optimal?

  Yes

Why optimal? By contradiction

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.



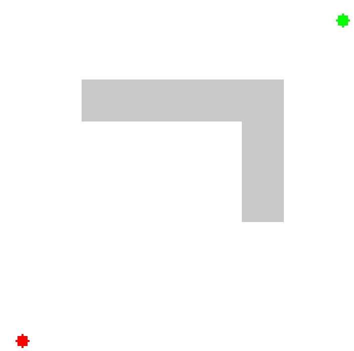
$$\begin{aligned} f(G_2) &= g(G_2) && \text{since } h(G_2) = 0 \\ &> g(G_1) && \text{since } G_2 \text{ is suboptimal} \\ &\geq f(n) && \text{since } h \text{ is admissible} \end{aligned}$$

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

A* is “optimally efficient”

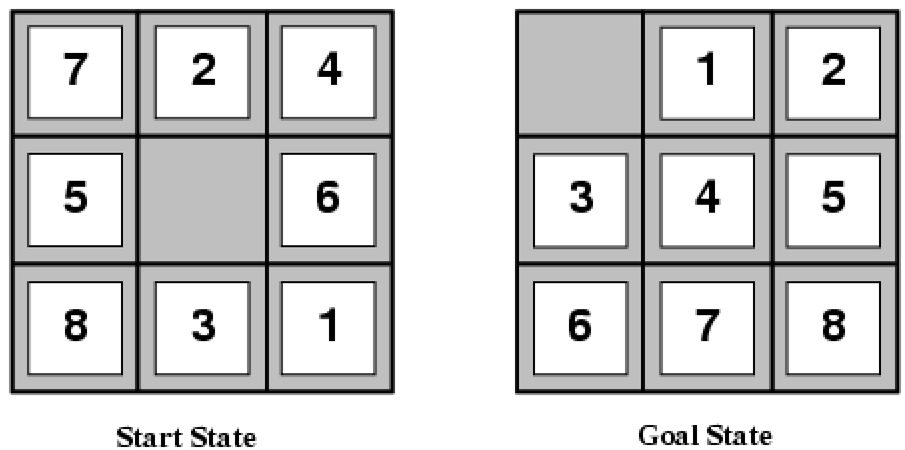
- With an admissible heuristic,
  - A* expands *all* nodes with f(n) < C
  - A* expands *some* nodes with f(n) = C
  - A* expands *no* nodes with f(n) > C

- So, except for the variable (usually small) number of nodes with f(n) = C,
  - No optimal algorithm using *h* expands fewer nodes than A*



From wikipedia.org, contributed by Subh83

Admissible heuristics

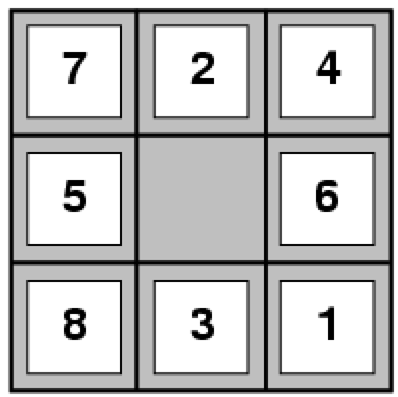


Admissible heuristics

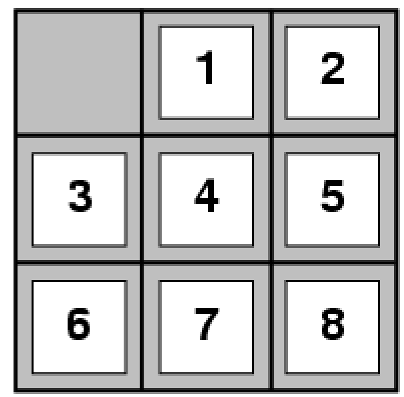
E.g., for the 8-puzzle:

- $h_1(n)$ = 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

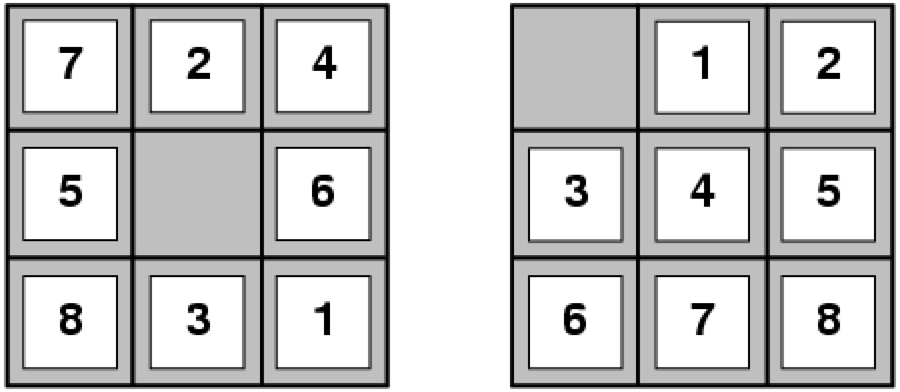
- $h_1(S) = ?$
- $h_2(S) = ?$

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

- <u>$h_1(S)$ = ?</u> 8
- <u>$h_2(S)$ = ?</u> 3+1+2+2+2+3+3+2 = 18

Dominance

- If $h_2(n) \geq h_1(n)$ for all n (both admissible) then h_2 dominates h_1

- h_2 is at least as good as h_1 for search, and likely better
  - Why?

d	Search Cost (nodes)		
	IDS	A*(h1)	A*(h2)
2	10	6	6
4	112	13	12
6	680	20	18
8	6386	39	25
10	47127	93	39
12	3644035	227	73
14	-	539	113
16	-	1301	211
18	-	3056	363

Summary

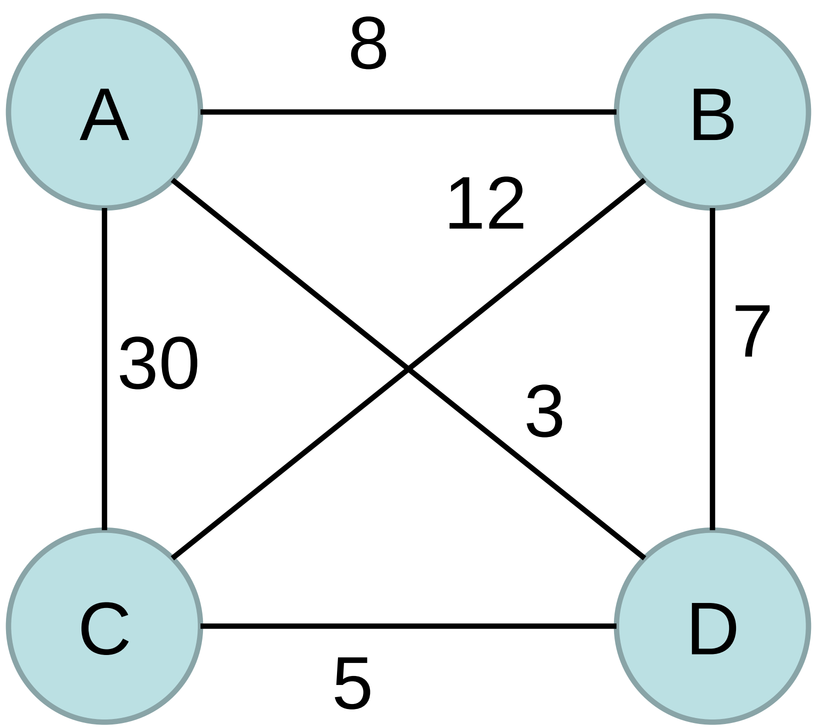
- A* search
  - Expand nodes in increasing order of:
    $f(n) = g(n) + h(n)$
    = cost so far + estimated cost to goal
  - Optimal for *admissible* heuristics
    - Admissible = “optimistic”
  - Designing heuristics is key for performance
    - More next time

Relaxed problems

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

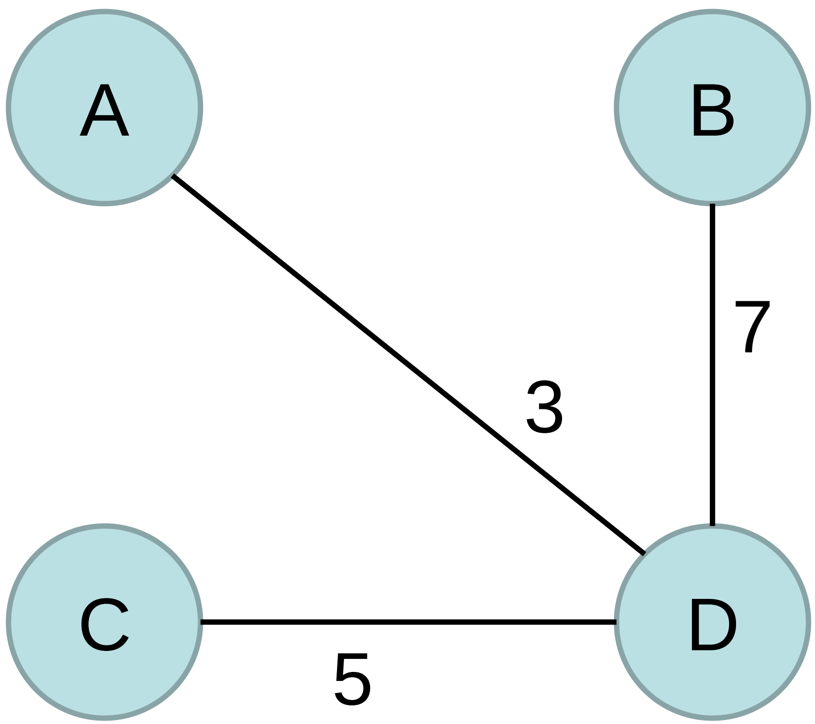
Traveling Salesman Problem

- Goal: find the least-cost cycle in the graph that visits each node exactly once



TSP Relaxed Problem Heuristic

- Relaxed problem: find least-cost *tree* that connects all nodes (minimum spanning tree).
  - **Cost(MST) <= Cost(Best Tour – 1 edge) < Cost(Best Tour)**

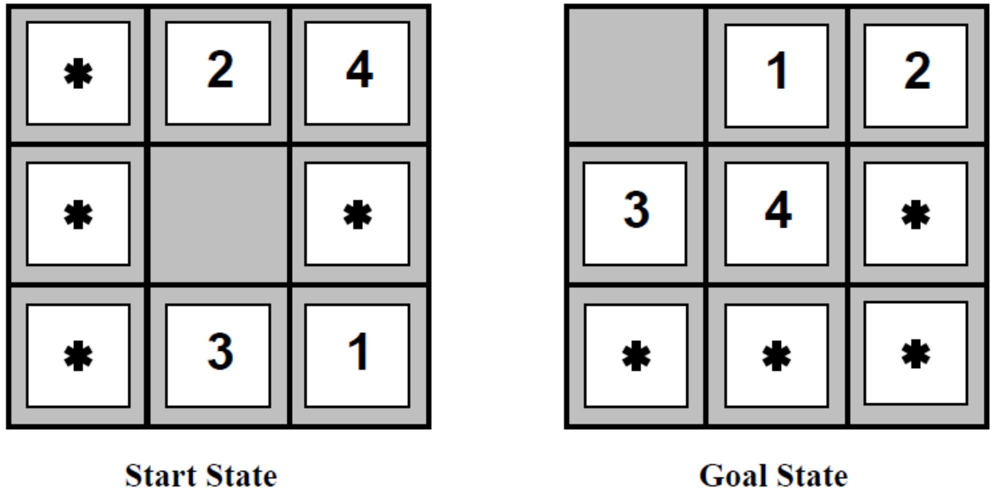


Combining Heuristics

- Say we have two heuristics, h1 and h2, and neither dominates the other.
  - What can we do?

- h3(n) = max(h1(n), h2(n))
  - h3 dominates h1, h2

Pattern Databases



- h(n) = cost to get {1,2,3,4} in right place
  - Compute once for all possible configurations and store
- Can use multiple sub-problems (e.g., {5,6,7,8}) and combine with max
  - Or, ignore * moves and *add* disjoint subproblems

Summary of A* Search

- Expands node n with minimum f(n) = g(n) + h(n)

  = path cost so far + heuristic estimate

- Optimal for *admissible* heuristic h(n)
  - I.e. h that underestimates true path cost

- Designing good heuristics is crucial for performance
  - One method: Relaxed problems

- Combining heuristics
  - Take max or add “disjoint” heuristics

Outline

- Greedy best-first search
- A^* search
- Heuristics
- Local search algorithms
- Hill-climbing search
- Simulated annealing search
- Local beam search
- Genetic algorithms

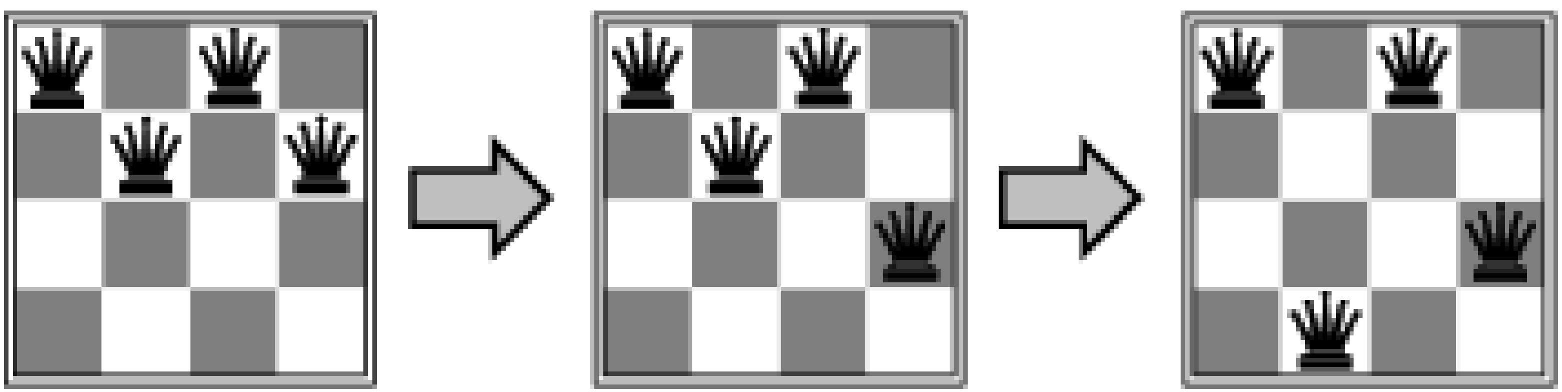
Local search algorithms

- In many optimization problems, the path to the goal is irrelevant
  - the goal state itself is the solution

- State space = set of "complete" configurations
- Find configuration satisfying constraints, e.g., n-queens

- In such cases, we can use local search algorithms
- keep a single "current" state, try to improve it

Example: *n*-queens

- Put *n* queens on an *n* × *n* board with no two queens on the same row, column, or diagonal
- 

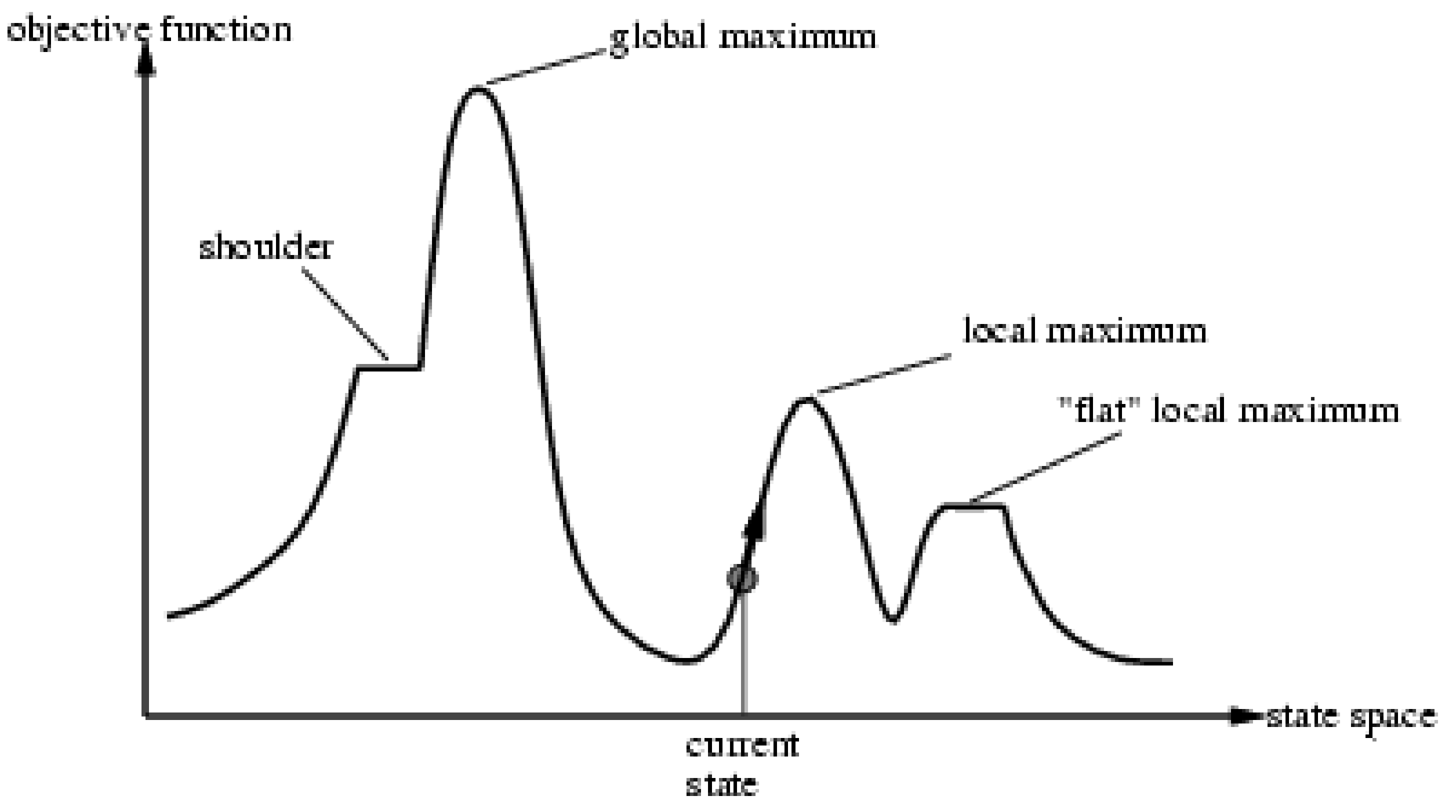
Hill-climbing search

- "Like climbing Everest in thick fog with amnesia"

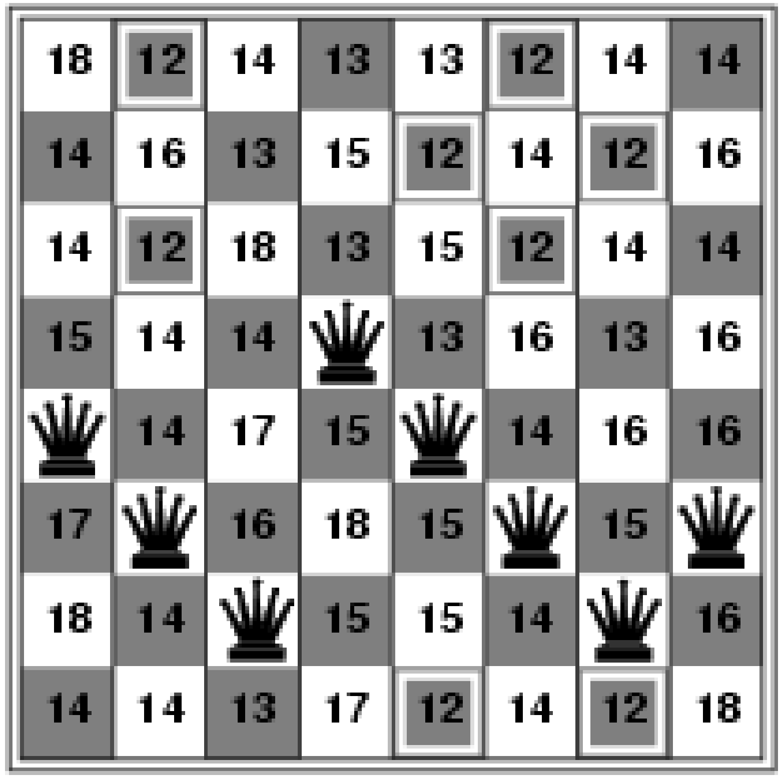
```
function HILL-CLIMBING( problem) returns a state that is a local maximum
    inputs: problem, a problem
    local variables: current, a node
                     neighbor, a node

    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
```

Hill-climbing search

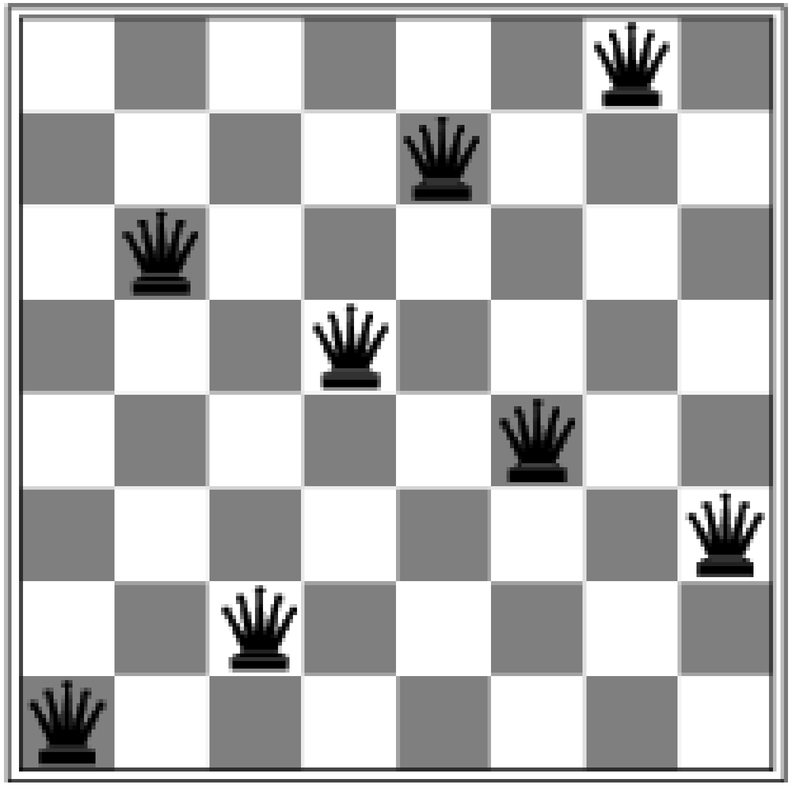
- Problem: depending on initial state, can get stuck in local maxima
- 

Hill-climbing search: 8-queens problem



- *h* = number of pairs of queens that are attacking each other, either directly or indirectly
- *h = 17* for the above state

Hill-climbing search: 8-queens problem



- A local minimum with *h = 1*

Simulated annealing search

- Idea: escape local maxima by allowing some "bad" moves but gradually decrease their 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
        ΔE ← VALUE[next] – VALUE[current]
        if ΔE > 0 then current ← next
        else current ← next only with probability e^(ΔE/T)
```

Properties of simulated annealing search

- One can prove: If *T* decreases slowly enough, then simulated annealing search will find a global optimum with probability approaching 1

- Widely used in VLSI layout, airline scheduling, etc

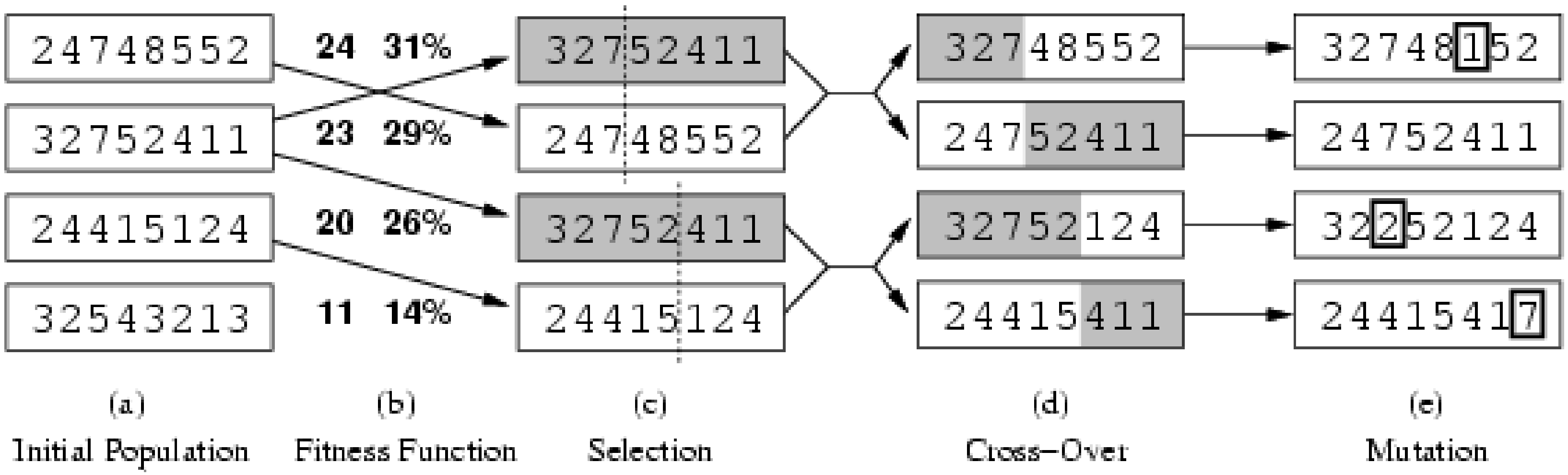
Local beam search

- Keep track of *k* states rather than just one
- Start with *k* randomly generated states
- At each iteration, all the successors of all *k* states are generated
- If any one is a goal state, stop; else select the *k* best successors from the complete list and repeat.

Genetic algorithms

- A successor state is generated by combining two parent states
- Start with *k* randomly generated states (population)
- A state is represented as a string over a finite alphabet (often a string of 0s and 1s)
- Evaluation function (fitness function). Higher values for better states.
- Produce the next generation of states by selection, crossover, and mutation

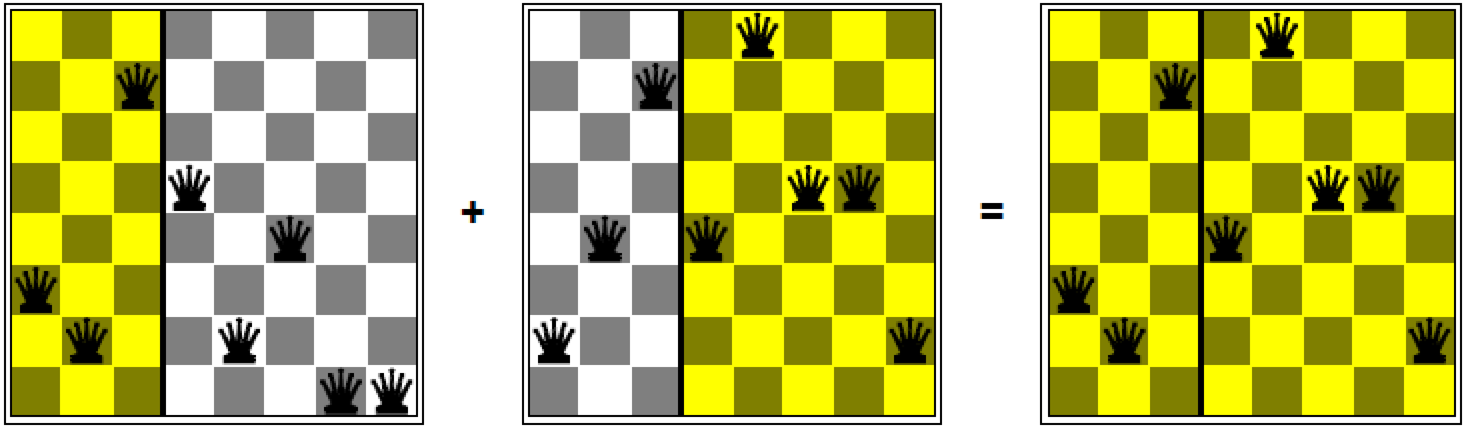
Genetic algorithms



- Fitness function: number of non-attacking pairs of queens (min = 0, max = 8 × 7/2 = 28)
- 24/(24+23+20+11) = 31%
- 23/(24+23+20+11) = 29% etc

Genetic algorithms

- Genetic algorithm is "stochastic beam search"
  - Key difference: combine multiple parents



For which problems is this helpful?

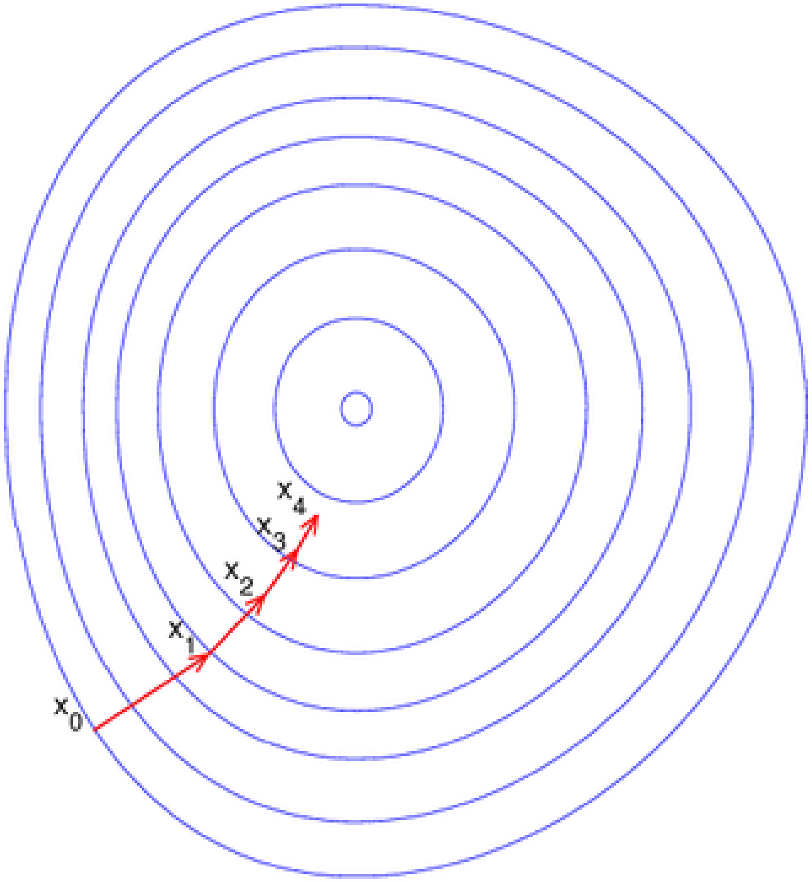
Continuous Optimization

- Many AI problems require optimizing a function $f(\mathbf{x})$, which takes continuous values for input vector $\mathbf{x}$
- Huge research area
- Examples:
  - **Machine Learning**
  - Signal/Image Processing
  - Computational biology
  - Finance
  - Weather forecasting
  - Etc., etc.

Gradient Ascent

- Idea: move in direction of steepest ascent (gradient)

- $\mathbf{x}_k = \mathbf{x}_{k-1} + \eta \nabla f(\mathbf{x}_{k-1})$



Types of Optimization

- Linear vs. non-linear
- Analytic vs. Empirical Gradient
- Convex vs. non-convex
- Constrained vs. unconstrained

Continuous Optimization in Practice

- *Lots* of previous work on this
- Use packages