Artificial Intelligence

2. Informed Search

Lars Schmidt-Thieme

Information Systems and Machine Learning Lab (ISMLL)
Institute of Economics and Information Systems
& Institute of Computer Science
University of Hildesheim
http://www.ismll.uni-hildesheim.de

Uniform Cost Search

```
uniform-cost-search(X, succ, cost, x_0, g) :
border := {x_0}
c(x_0) := 0
while border ≠ ∅ do
    x := argmin_{x∈border} c(x)
    if g(x) = 1
      return branch(x, previous)
    fi
    for y ∈ succ(x, A) do
        border := border ∪ {y}
        c(y) := c(x) + cost(x, y)
        previous(y) := x
    od
    border := border \ {x}
od
return ∅

branch(x, previous) :
P := ∅
while x ≠ ∅ do
    insert-at-beginning(P, x)
    x := previous(x)
od
return P
```


Best-First-Search


```
uniform-cost-search(X, succ, cost, x_0, g) :
border := {x_0}
c(x_0) := 0
while border ≠ ∅ do
    x := argmin_{x∈border} c(x)
    if g(x) = 1
      return branch(x, previous)
    fi
    for y ∈ succ(x, A) do
        border := border ∪ {y}
        c(y) := c(x) + cost(x, y)
        previous(y) := x
    od
    border := border \ {x}
od
return ∅

branch(x, previous) :
P := ∅
while x ≠ ∅ do
    insert-at-beginning(P, x)
    x := previous(x)
od
return P
```

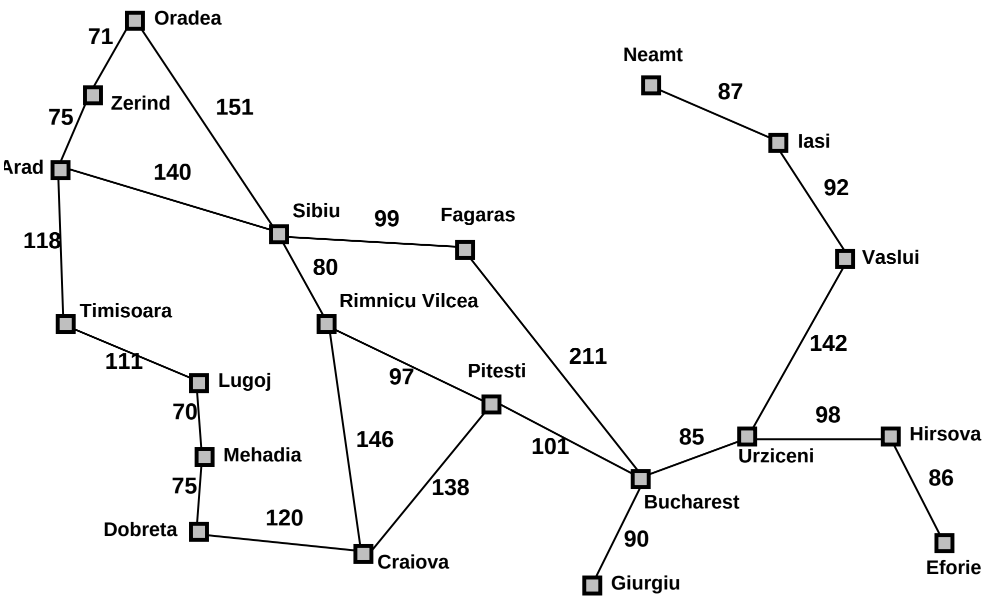

```
best-first-search(X, succ, cost, x_0, g, f) :
border := {x_0}
while border ≠ ∅ do
    x := argmin_{x∈border} f(x)
    if g(x) = 1
      return branch(x, previous)
    fi
    for y ∈ succ(x, A) do
        border := border ∪ {y}
        previous(y) := x
    od
    border := border \ {x}
od
return ∅
```

f: **evaluation function**

uniform cost search is special case with

$$f(x) := \text{cost}(\text{branch}(x, \text{previous}))$$

Additional Information: a Heuristics



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

cost : $X \times X \to \mathbb{R}$

$h : X \to \mathbb{R}$

Greedy Best-First Search

Additional Information:
Heuristics h estimates costs to next goal state.

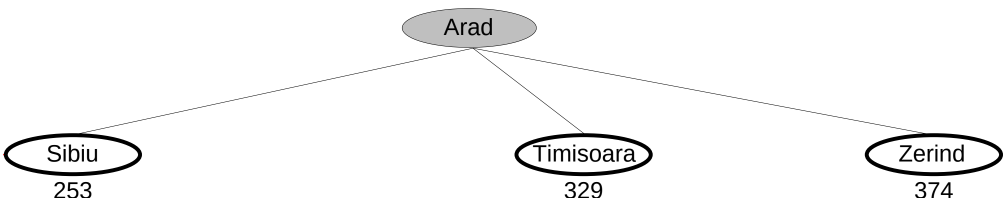
Greedy best-first search:
Take heuristics as evaluation function:

$$f := h$$

Greedy Best-First Search / Example

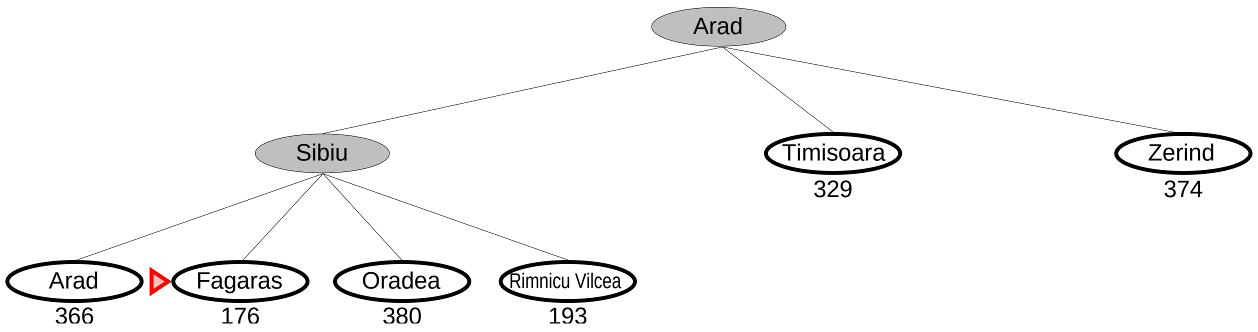
Artificial Intelligence / 1. Greedy Best-First Search

Greedy Best-First Search / Example



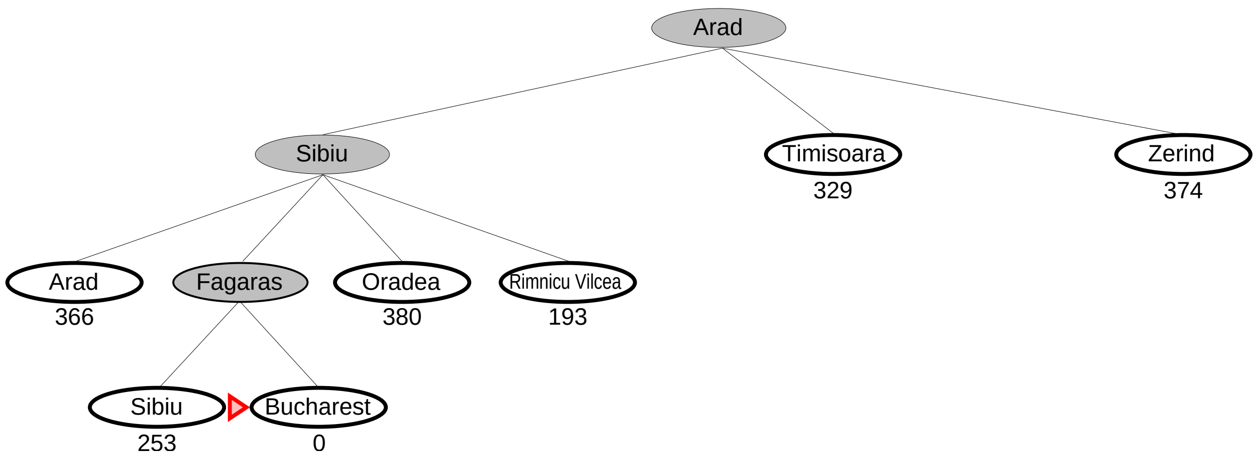
Lars Schmidt-Thieme, Information Systems and Machine Learning Lab (ISMLL), University of Hildesheim, Germany,
Course on Artificial Intelligence, summer term 2008
4/25

Greedy Best-First Search / Example



Artificial Intelligence / 1. Greedy Best-First Search

Greedy Best-First Search / Example



Greedy Best-First Search

Completeness
no (can get stuck in loops:
e.g., goal Oradea; Iasi $\rightarrow$ Neamt $\rightarrow$ Iasi $\rightarrow \ldots$)
yes with repeated state checking

Optimality
no

Time complexity
$O(b^m)$ — but average time complexity may be much better for good heuristics.

Space complexity
same as time complexity as whole search tree is kept in memory.

A^* Search

Additional Information:
Heuristics h estimates costs to next goal state.

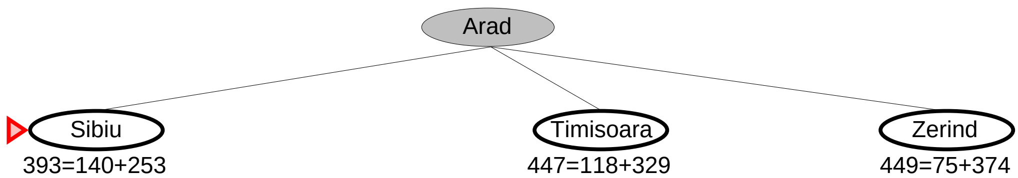
Greedy best-first search:
Take heuristics as evaluation function:

$$f := h$$

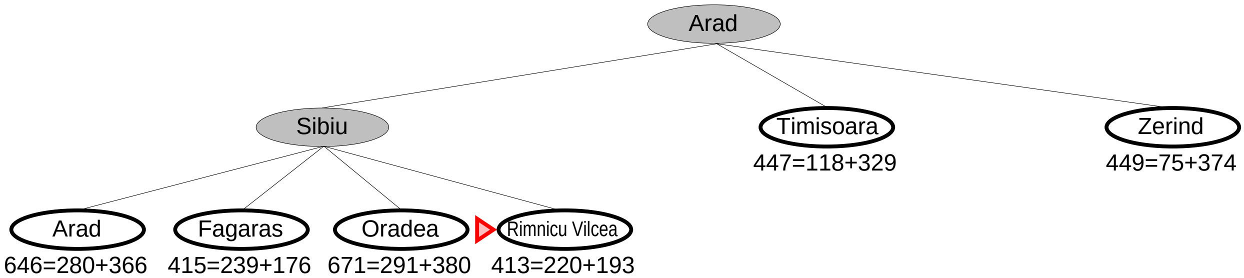
A^* search:
Idea: penalty paths that are already costly.
$\rightsquigarrow$ take sum of costs so far and heuristics as evaluation function:

$$f := \text{cost} + h$$

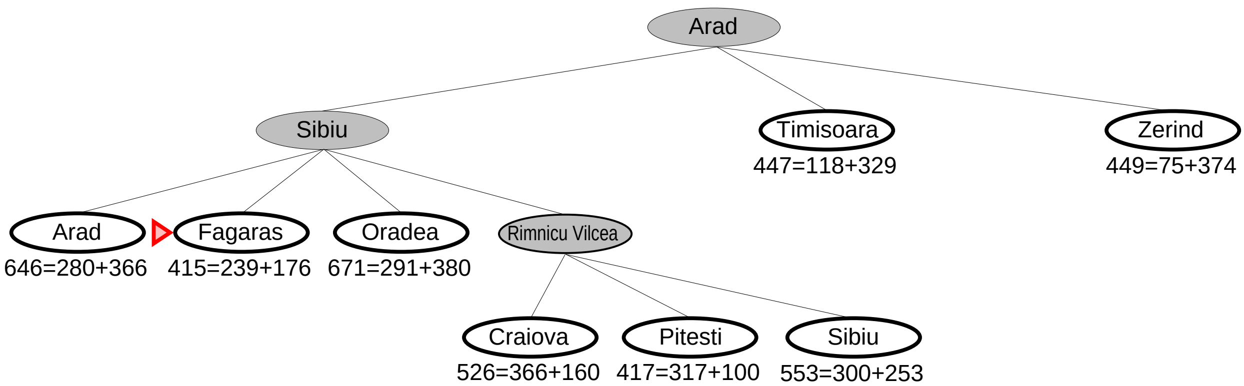
A^* Search / Example



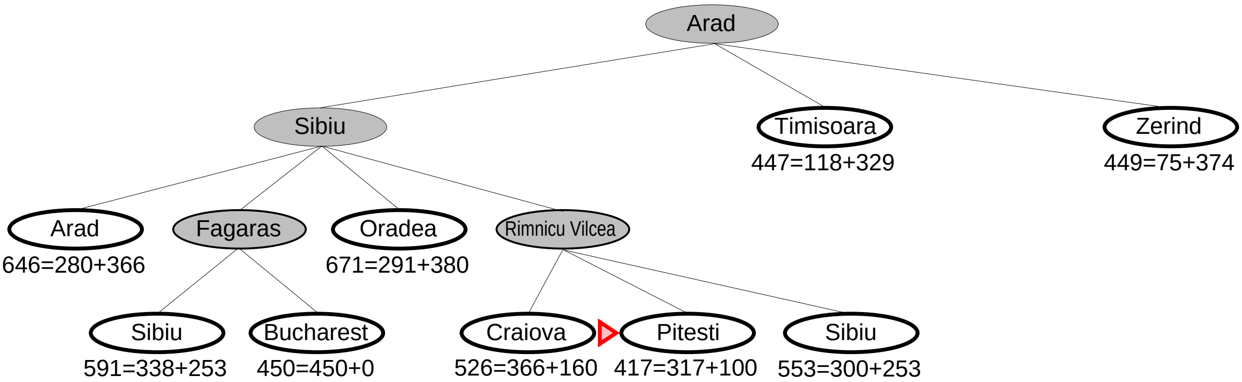
A^* Search / Example



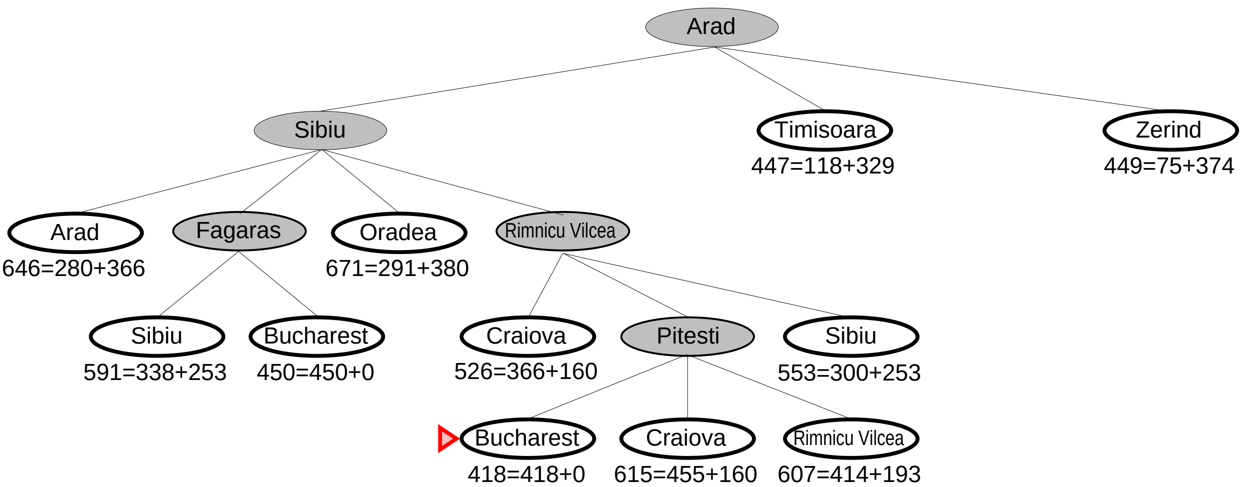
A^* Search / Example



A* Search / Example



A* Search / Example



A* Search

Completeness

yes (if b is finite and step costs are $\geq \epsilon > 0$
$\rightsquigarrow$ there are only finite many states x with $f(x) \leq f(\text{goal})$)

Optimality

no (with any heuristics)
yes with admissible heuristics (see next page)

Time complexity

exponential in $(\text{relative error in } h) \cdot d$.

Space complexity

same as time complexity as whole search tree is kept in memory.

Optimality

Heuristics is **admissible** (“optimistic”, lower bound):

$$h \leq h^*$$

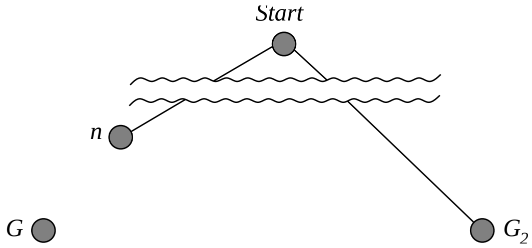
where h^* denotes the true cost to the next goal.

Lemma: If h is admissible, A* search is optimal.

Proof: assume suboptimal G_2 has been found
and let n be any node on an optimal path to optimal solution G.

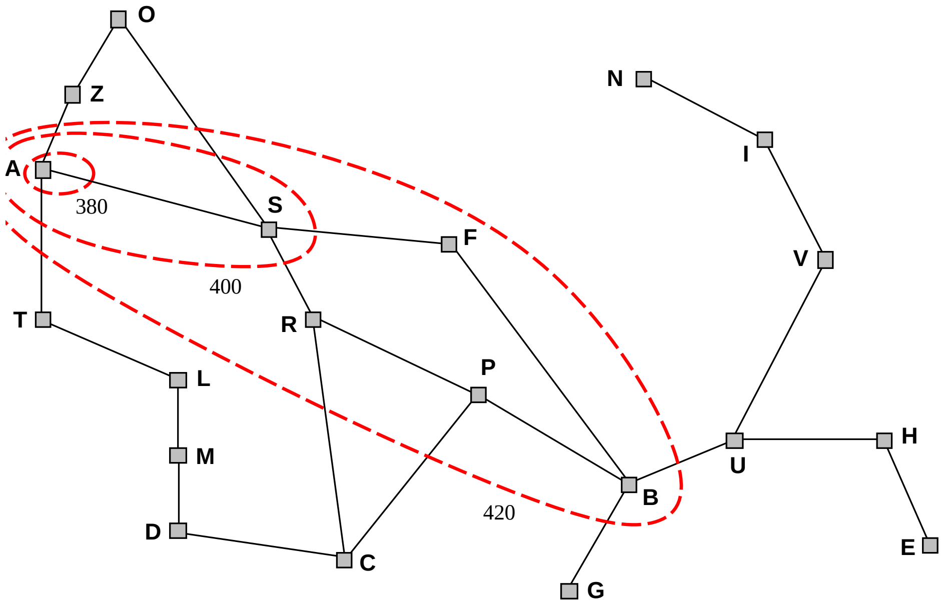
$$f(G_2) = \text{cost}(G_2) > \text{cost}(G) \geq f(n)$$

Hence n must be visited before G_2.

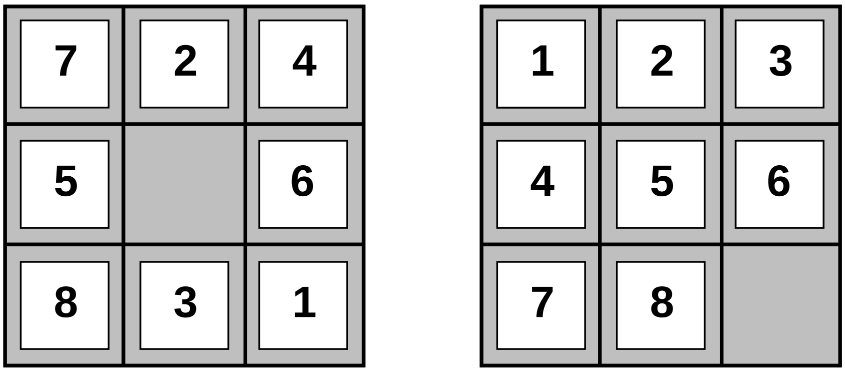


Optimality

A* expands nodes in layers/contours of increasing f value.

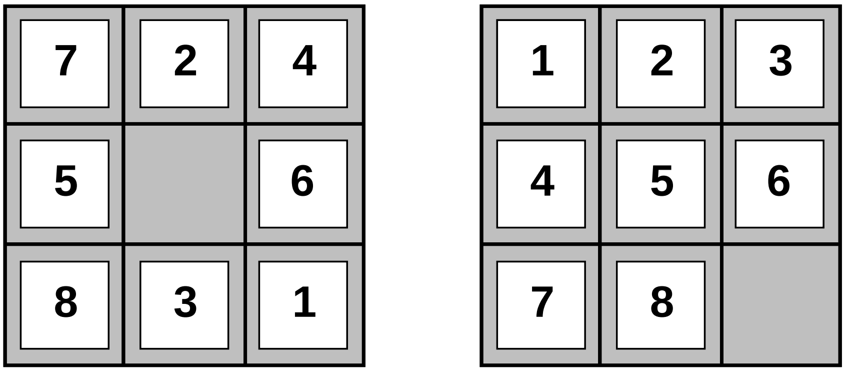


Example 8-Puzzle



Start State **Goal State**

Example 8-Puzzle

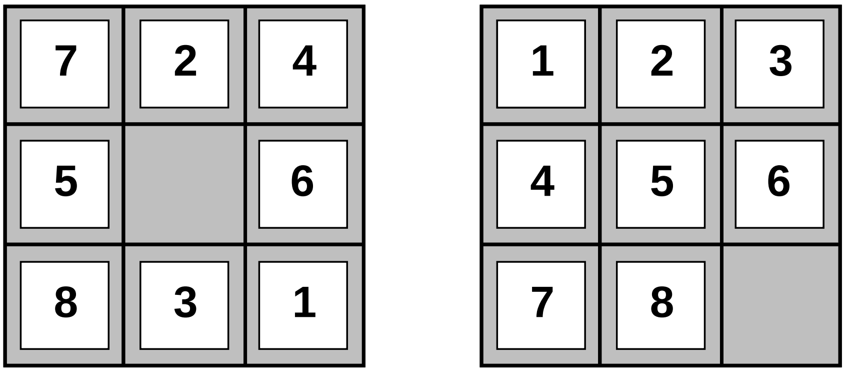


Start State **Goal State**

$h_1(x) :=$ number of misplaced tiles

$h_1(x) = 6$.

Example 8-Puzzle



Start State **Goal State**

$h_2(x) :=$ sum of distances of all misplaced tiles to goal
Here: distance in required moves, i.e., Manhattan distance.

$h_2(x) = 4 + 0 + 3 + 3 + 1 + 0 + 2 + 1 = 14$

Which heuristics is better?

Size of search tree in nodes for two examples:

algorithm	length of optimal solution $d = 14$	$d = 24$
IDS	3,473,941	$\approx$ 54,000,000,000
A*(h_1)	539	39,135
A*(h_2)	113	1,641

For two admissble heurstics h_1 and h_2:
h_1 **dominates** h_2 if $h_1(x) \geq h_2(x)$ for all x.

Using a dominant heuristics with A* always is faster.
(as only nodes x with $f(x) = \text{cost}(x) + h(x) \leq f(x^*)$ are expanded!)

$h := \max(h_1, h_2)$ also is admissible and dominates h_1 and h_2.

How to design a heuristics? / 1. Relaxation

Conditions for legal moves:

A tile can move from A to B
(a) if A and B are horizontally or vertically adjacent and B is blank.

Relax conditions to:

(b) if A and B are horizontally or vertically adjacent.
— OR —
(c) if B is blank.
— OR —
(d) if true.

h_1 gives the true costs for relaxed problem (d).
h_2 gives the true costs for relaxed problem (b).

How to design a heuristics? / 2. Subproblems

Look at a subproblem, e.g.,
8-puzzle with four tiles labeled 1 to 4 and four unlabeled tiles.

Each state x can be projected to a state subproblem$_{1234}(x)$ of the subproblem.

$$\begin{pmatrix} 7 & 2 & 4 \\ 5 & & 6 \\ 8 & 3 & 1 \end{pmatrix} \overset{\text{project}}{\leadsto} \begin{pmatrix} * & 2 & 4 \\ * & & * \\ * & 3 & 1 \end{pmatrix} \overset{\text{solve}}{\leadsto} \begin{pmatrix} 1 & 2 & 3 \\ 4 & * & * \\ * & * & \end{pmatrix}$$

$h_3(x) :=$cost(subproblem$_{1234}(x)$)
— the cost to solve just the subproblem.
(all configurations of such subproblems, called **patterns** and their costs can be precomputed and stored in a database).

1. **Greedy Best-First Search**

2. **A* Search**

3. **Admissible Heuristic Functions**

4. **Local Search**

Local Search

For some problems just the final state is interesting,
not the action/state sequence to reach the final state.

Examples:

- 8-queens problem
- traveling salesman problem
- ...

Then it is a waste to keep all the information about solution paths.
Instead:

- keep only one state x, the **actual** or **current state**
- consider only neighboring states as next actual state
  i.e., reachable by an action from the actual state: succ(x, A).
- needs objective function to steer movement: f
  may need an heuristics if the true objective is not accessible.

Called **local search** or **neighborhood search**.

Local Search

If the state space consists just of “complete configurations”, local search can be understood as iterative improvement.

In any case:
Local search requires just constant space.

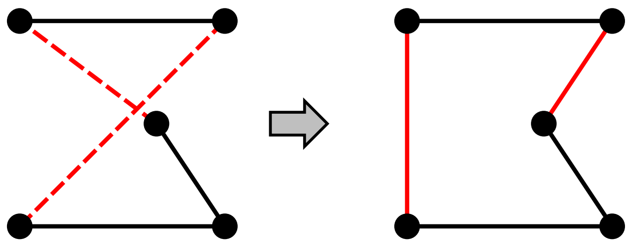
Example / Traveling Salesman Problem

Problem:
given a graph with labeled edges,
find a cycle that visits each node exactly once (hamiltonian cycle; tour) with minimal sum of edge labels (costs).

State space:
all tours.

Actions:
remove two edges and join the resulting two paths in the other possible way (2-Opt; Croes 1958).

Objective function:
cost of resulting tour.

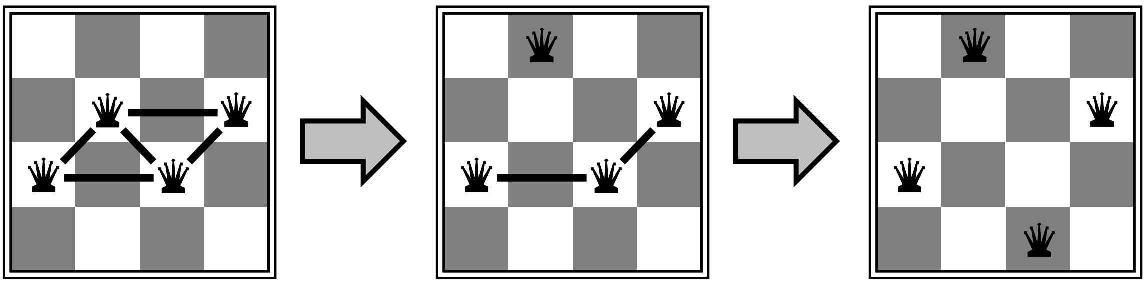


Example / 8-Queens

State space:
8 queens on the board, each in one column.

Actions:
move a queen to another row in her column.

Heuristics h:
number of possible attacks.



h = 5 h = 2 h = 0

Hill-climbing / Steepest Descent/Ascent

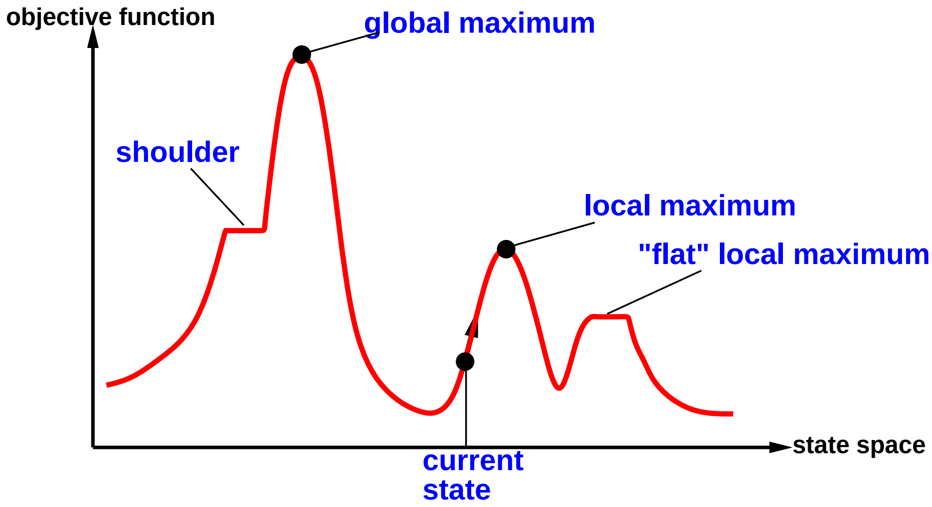
Greedy local search:
always move to the neighbor with the maximal objective value.

hill-climbing(X, succ, f, x_0) :
$y := x_0$
do
  $x := y$
  $y := \text{argmax}_{y \in \text{succ}(x,A)} f(y)$
while $f(y) > f(x)$
return x

For continuous state spaces / actions and differentiable objective functions:
gradient descent/ascent.

Hill-climbing / Steepest Descent/Ascent

State space landscape:



Random restart: try to overcome local maxima.

Random sideways move: try to overcome shoulders.
(but restrict their number to avoid infinite loops on flat local maxima)

Stochastic Hill-climbing

Idea:
like hill-climbing
but choose randomly among all improving actions
proportional to their improvement.

hill-climbing-stochastic(X, succ, f, x_0) :
$y := x_0$
do
  $x := y$
  $y \sim \text{multinomial}(\text{succ}(x, A))$ with $p(y) := \frac{\max(0, f(y) - f(x))}{\sum_y \max(0, f(y) - f(x))}, \quad y \in \text{succ}(x, A)$
while $f(y) > f(x)$
return x

$p(y)$ is called the **acceptance probability** for neighboring state y of x.

Simulated Annealing

Idea:
like hill-climbing
but also allow deteriorating actions
slight deteriorations more often than severe deteriorations
less and less deteriorations as the search proceeds

simulated-annealing$(X, \text{succ}, f, x_0, T)$:
$x := x_0$
for $k := 1$ to ∞ **while** $T(k) > 0$ **do**
    $y \sim \text{uniform}(\text{succ}(x, A))$
    if $f(y) > f(x)$ or random() $\leq \exp((f(y) - f(x))/T(k))$
        $x := y$
    fi
od
return x

T is called the **temperature schedule**, $T \to 0$ for k growing.

Beam Search

Idea:
like hill-climbing
but retain k best solutions in parallel.

beam-search$(X, \text{succ}, f, g, k)$:
$S :=$ random subset of X of size k
while $g(x) = 0 \; \forall x \in S$ **do**
    $S := \text{argmax}^k_{y \in \text{succ}(S,A)} f(y)$
od
return $x \in S$ with $g(x) = 1$

where succ$(S, A) := \bigcup_{x \in S}$ succ(x, A) and
argmax^k selects the k elements with maximum argument.

S is called **population**, each state an **individual**.

This is different from k random restarts of hill-climbing!

Genetic Algorithms

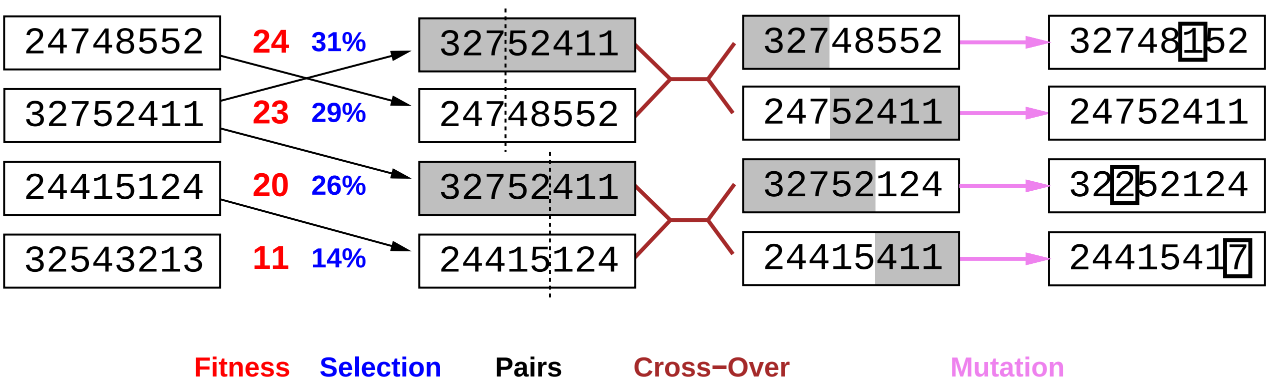
Idea:
like beam search
but combine two states to a new state
(represented as string/vector)

genetic-algorithm(X, f, g, k) :
$S :=$ random subset of X of size k
while $g(x) = 0\ \forall x \in S$ **do**
  $S' := \emptyset$
  for $i = 1 \ldots k$ **do**
    $x_1, x_2 \sim$ multinomial(S) with $p(x) := \frac{f(x)}{\sum_{x' \in S} f(x')}, \quad x \in S$
    $y :=$ combine(x_1, x_2)
    if (random() $< p_{mutation}$) $y :=$ mutation(y) **fi**
    $S' := S' \cup \{y\}$
  od
  $S := S'$
od
return $x \in S$ with $g(x) = 1$

combine(x_1, x_2) :
$n :=$ length(x_1)
$c \sim$ uniform$(\{1, 2, \ldots, n\})$
return concat$(x_1[1 \ldots c], x_2[c + 1 \ldots n])$

f also is called **fitness** (and should be ≥ 0).

Genetic Algorithms / Example



Genetic algorithms create triadic neighborhoods
pair of states $\rightarrow$ state
by means of combination/reproductio/cross-over.

To make sense, the string encoding must be such that close positions encode related properties of the candidate solution.

Genetic Algorithms / Example