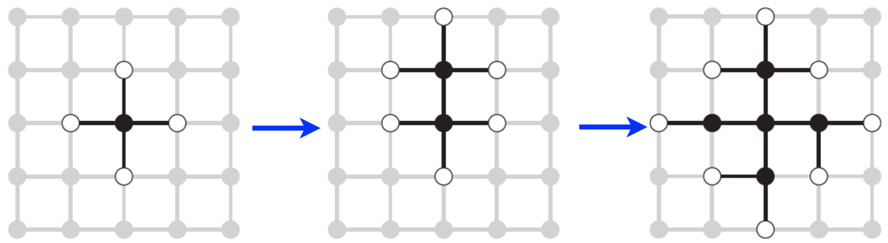
# TDDC17

Seminar III
Search II
Informed or Heuristic Search
Beyond Classical Search

Artificial Intelligence & Integrated Computer Systems Division
Department of Computer and Information Science
Linköping University, Sweden



# Intuitions behind heuristic search



The separation property of GRAPH-SEARCH

Black Nodes - Explored
White Nodes - Frontier
Grey Nodes - Unexplored

*Systematic Search*

Find a <u>heuristic measure</u> h(n) which <u>estimates</u> how close a node n in the frontier is to the nearest goal state and then order the frontier queue accordingly relative to closeness.

Introduce an ***evaluation function*** on nodes f(n) which is a cost estimate. f(n) will order the frontier by least cost.

f(n) = .... + h(n)  h(n) will be part of f(n)

Artificial Intelligence & Integrated Computer Systems Division
Department of Computer and Information Science
Linköping University, Sweden

# Recall Uniform-Cost Search

```
function UNIFORM-COST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  frontier ← a priority queue ordered by PATH-COST, with node as the only element
  explored ← an empty set
  loop do
     if EMPTY?(frontier) then return failure
     node ← POP(frontier)  /* chooses the lowest-cost node in frontier */
     if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
     add node.STATE to explored
     for each action in problem.ACTIONS(node.STATE) do
        child ← CHILD-NODE(problem, node, action)
        if child.STATE is not in explored or frontier then
           frontier ← INSERT(child, frontier)
        else if child.STATE is in frontier with higher PATH-COST then
           replace that frontier node with child
```

g(n) = cost of path from root node to n

f(n) = g(n)





# Best-First Search

```
function BEST-FIRST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE,
  frontier ← a priority queue ordered by f(n), with node as the only element
  explored ← an empty set
  loop do
     if EMPTY?(frontier) then return failure
     node ← POP(frontier)  /* chooses the lowest-cost node in frontier */
     if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
     add node.STATE to explored
     for each action in problem.ACTIONS(node.STATE) do
        child ← CHILD-NODE(problem, node, action)
        if child.STATE is not in explored or frontier then
           frontier ← INSERT(child, frontier)
        else if child.STATE is in frontier with higher f(n) then
           replace that frontier node with child
```

f(n) = .... + h(n)

Most best-first search algorithms include h(n) as part of f(n)

h(n) is a heuristic function

*Estimated cost of the cheapest path through state n to a goal state*

Artificial Intelligence & Integrated Computer Systems Division
Department of Computer and Information Science
Linköping University, Sweden

# Greedy Best-First Search

```
function BEST-FIRST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE,
  frontier ← a priority queue ordered by f(n), with node as the only element
  explored ← an empty set
  loop do
     if EMPTY?(frontier) then return failure
     node ← POP(frontier)  /* chooses the lowest-cost node in frontier */
     if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
     add node.STATE to explored
     for each action in problem.ACTIONS(node.STATE) do
        child ← CHILD-NODE(problem, node, action)
        if child.STATE is not in explored or frontier then
           frontier ← INSERT(child, frontier)
        else if child.STATE is in frontier with higher f(n) then
           replace that frontier node with child
```

*Don't care about anything except how close a node is to a goal state*

$$f(n) = h(n)$$

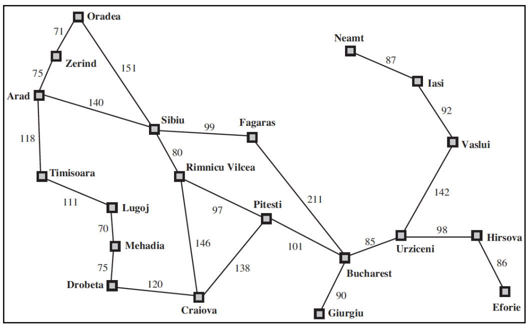
Let's find a heuristic for the Romania Travel Problem



# Romania Travel Problem Heuristic

Straight line distance from city n to goal city n'

Assume the cost to get somewhere is a function of the distance traveled




$h_{SLD}()$ for Bucharest

| | | | |
|---|---|---|---|
| **Arad** | 366 | **Mehadia** | 241 |
| **Bucharest** | 0 | **Neamt** | 234 |
| **Craiova** | 160 | **Oradea** | 380 |
| **Drobeta** | 242 | **Pitesti** | 100 |
| **Eforie** | 161 | **Rimnicu Vilcea** | 193 |
| **Fagaras** | 176 | **Sibiu** | 253 |
| **Giurgiu** | 77 | **Timisoara** | 329 |
| **Hirsova** | 151 | **Urziceni** | 80 |
| **Iasi** | 226 | **Vaslui** | 199 |
| **Lugoj** | 244 | **Zerind** | 374 |

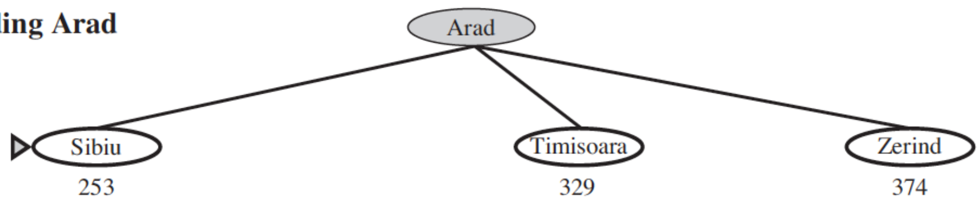
$$f(n) = h_{SLD}(n)$$

# GBFS-1




# GBFS-2

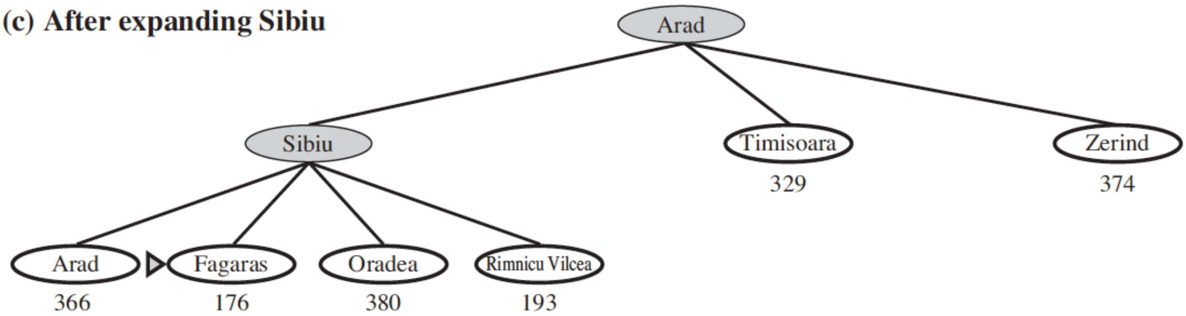
**(b) After expanding Arad**




| | | | |
|---|---|---|---|
| **Arad** | 366 | **Mehadia** | 241 |
| **Bucharest** | 0 | **Neamt** | 234 |
| **Craiova** | 160 | **Oradea** | 380 |
| **Drobeta** | 242 | **Pitesti** | 100 |
| **Eforie** | 161 | **Rimnicu Vilcea** | 193 |
| **Fagaras** | 176 | **Sibiu** | 253 |
| **Giurgiu** | 77 | **Timisoara** | 329 |
| **Hirsova** | 151 | **Urziceni** | 80 |
| **Iasi** | 226 | **Vaslui** | 199 |
| **Lugoj** | 244 | **Zerind** | 374 |

# GBFS-3

**(c) After expanding Sibiu**

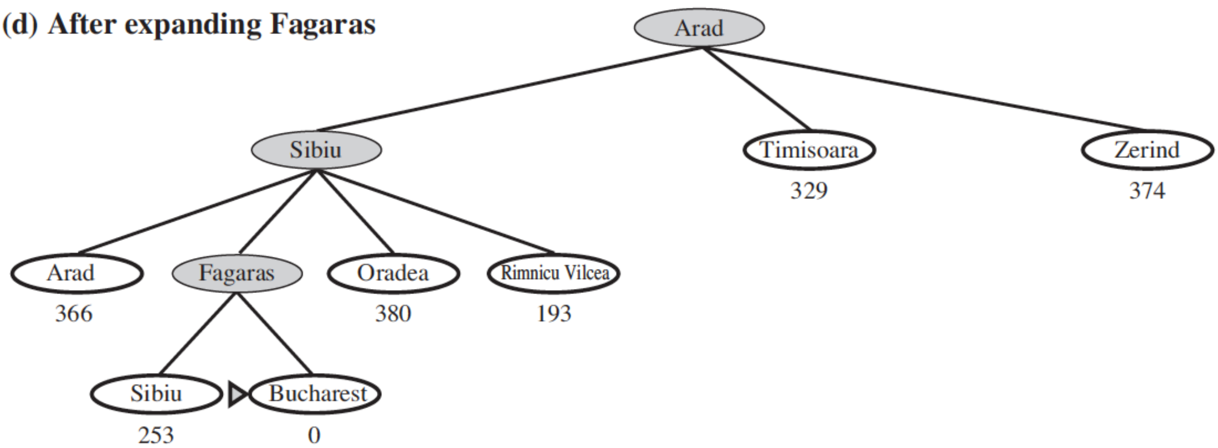



| | | | |
|---|---|---|---|
| **Arad** | 366 | **Mehadia** | 241 |
| **Bucharest** | 0 | **Neamt** | 234 |
| **Craiova** | 160 | **Oradea** | 380 |
| **Drobeta** | 242 | **Pitesti** | 100 |
| **Eforie** | 161 | **Rimnicu Vilcea** | 193 |
| **Fagaras** | 176 | **Sibiu** | 253 |
| **Giurgiu** | 77 | **Timisoara** | 329 |
| **Hirsova** | 151 | **Urziceni** | 80 |
| **Iasi** | 226 | **Vaslui** | 199 |
| **Lugoj** | 244 | **Zerind** | 374 |



# GBFS-4

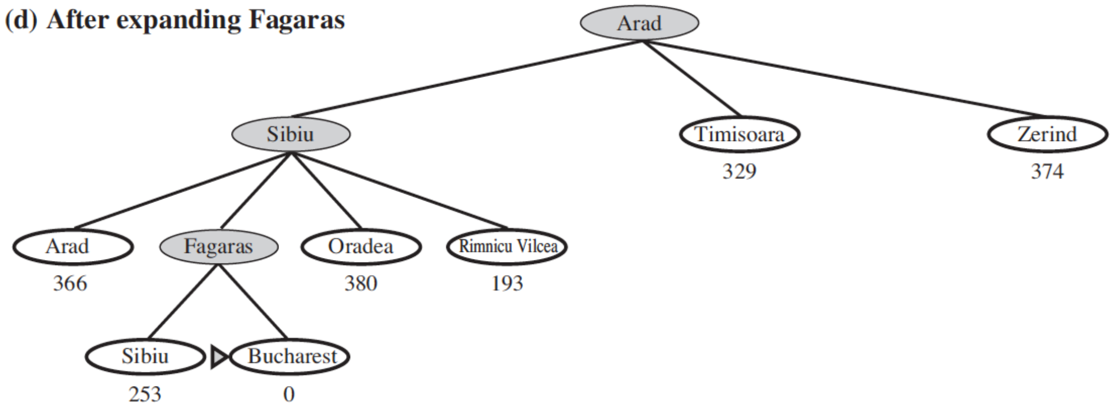
**(d) After expanding Fagaras**




| | | | |
|---|---|---|---|
| **Arad** | 366 | **Mehadia** | 241 |
| **Bucharest** | 0 | **Neamt** | 234 |
| **Craiova** | 160 | **Oradea** | 380 |
| **Drobeta** | 242 | **Pitesti** | 100 |
| **Eforie** | 161 | **Rimnicu Vilcea** | 193 |
| **Fagaras** | 176 | **Sibiu** | 253 |
| **Giurgiu** | 77 | **Timisoara** | 329 |
| **Hirsova** | 151 | **Urziceni** | 80 |
| **Iasi** | 226 | **Vaslui** | 199 |
| **Lugoj** | 244 | **Zerind** | 374 |

In this particular example, we rapidly reached the goal directly following a single path, but......

# Is Greedy Best-First Search Optimal?




**<u>No</u>**, the actual costs:
*Path Chosen*: Arad-Sibiu-Fagaras-Bucharest = **450**
*Optimal Path*: Arad-Sibiu-Rimnicu Vilcea-Pitesti-Bucharest = **418**

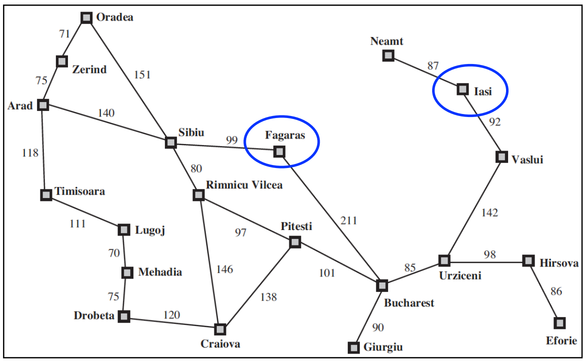
*The search cost is minimal but not optimal!*
*What's missing?*



# Is Greedy Best-First Search Complete?

- GBF Graph search is complete in finite spaces but not in infinite spaces
- GBF Tree search is not even complete in finite spaces. (Can go into infinite loops)

Consider going from Iasi to Fagaras?




Neamt is chosen 1st because h(Neamt) is closer than h(Vaslui), but Neamt is a deadend. Expanding Neamt still puts Iasi 1st on the frontier again since h(Iasi) is closer than h(Vaslui)...which puts Neamt 1st again!

Worst case time and space complexity for GBF tree search is $O(b^m)$

BUT

With heuristics performance is often much better with good choice of heuristic

* m -maximum length of any path in the search space (possibly infinite)

## Improving Greedy Best-First Search

Best-First Search finds a goal as fast as possible by using the h(n) function to estimate n's closeness to the goal.

Best-First Search chooses any goal node without concerning itself with the shallowness of the goal node or the cost of getting to n in the 1st place.

Rather than choosing a node based just on distance to the goal we could include a *quality notion* such as expected depth of the nearest goal

g(n) - the actual cost of getting to node n
h(n) - the estimated cost of getting from n to a goal state

$$f(n) = g(n) + h(n)$$

*f(n) is the estimated cost of the cheapest solution through n*

Artificial Intelligence & Integrated Computer Systems Division
Department of Computer and Information Science
Linköping University, Sweden



## A* Search

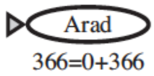
**function** BEST-FIRST-SEARCH(*problem*) **returns** a solution, or failure
  *node* ← a node with STATE = *problem*.INITIAL-STATE,
  *frontier* ← a priority queue ordered by f(n), with *node* as the only element
  *explored* ← an empty set
  **loop do**
    **if** EMPTY?(*frontier*) **then return** failure
    *node* ← POP(*frontier*) /* chooses the lowest-cost node in *frontier* */
    **if** *problem*.GOAL-TEST(*node*.STATE) **then return** SOLUTION(*node*)
    add *node*.STATE to *explored*
    **for each** *action* **in** *problem*.ACTIONS(*node*.STATE) **do**
      *child* ← CHILD-NODE(*problem*, *node*, *action*)
      **if** *child*.STATE is not in *explored* or *frontier* **then**
        *frontier* ← INSERT(*child*, *frontier*)
      **else if** *child*.STATE is in *frontier* with higher f(n) **then**
        replace that *frontier* node with *child*

$$f(n) = g(n) + h(n)$$

Artificial Intelligence & Integrated Computer Systems Division
Department of Computer and Information Science
Linköping University, Sweden

# A*-1

**(a) The initial state**




## Heuristic:

$f(n) = g(n) + h(n)$
$g(n)$ - Actual distance from root node to n
$h(n)$ - $h_{SLD}(n)$ straight line distance from n to (bucharest)

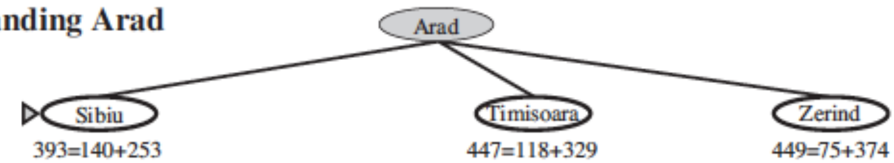
$h_{SLD}(n)$
Bucharest

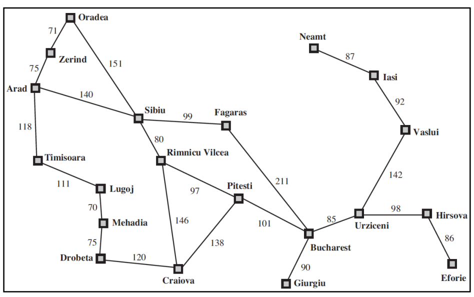
| | | | |
|---|---|---|---|
| **Arad** | 366 | **Mehadia** | 241 |
| **Bucharest** | 0 | **Neamt** | 234 |
| **Craiova** | 160 | **Oradea** | 380 |
| **Drobeta** | 242 | **Pitesti** | 100 |
| **Eforie** | 161 | **Rimnicu Vilcea** | 193 |
| **Fagaras** | 176 | **Sibiu** | 253 |
| **Giurgiu** | 77 | **Timisoara** | 329 |
| **Hirsova** | 151 | **Urziceni** | 80 |
| **Iasi** | 226 | **Vaslui** | 199 |
| **Lugoj** | 244 | **Zerind** | 374 |



# A*-2

**(b) After expanding Arad**

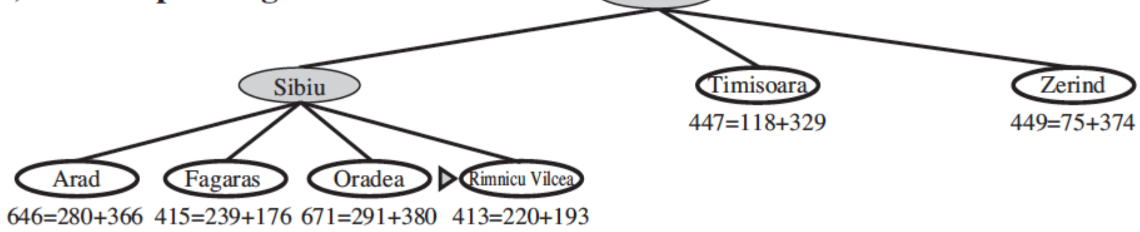


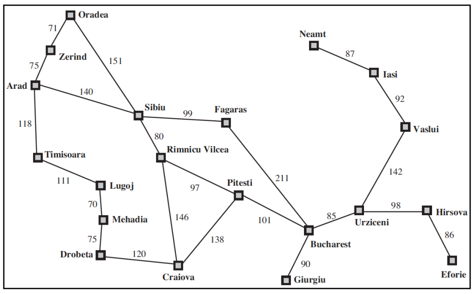




| | | | |
|---|---|---|---|
| **Arad** | 366 | **Mehadia** | 241 |
| **Bucharest** | 0 | **Neamt** | 234 |
| **Craiova** | 160 | **Oradea** | 380 |
| **Drobeta** | 242 | **Pitesti** | 100 |
| **Eforie** | 161 | **Rimnicu Vilcea** | 193 |
| **Fagaras** | 176 | **Sibiu** | 253 |
| **Giurgiu** | 77 | **Timisoara** | 329 |
| **Hirsova** | 151 | **Urziceni** | 80 |
| **Iasi** | 226 | **Vaslui** | 199 |
| **Lugoj** | 244 | **Zerind** | 374 |

# A*-3

## (c) After expanding Sibiu



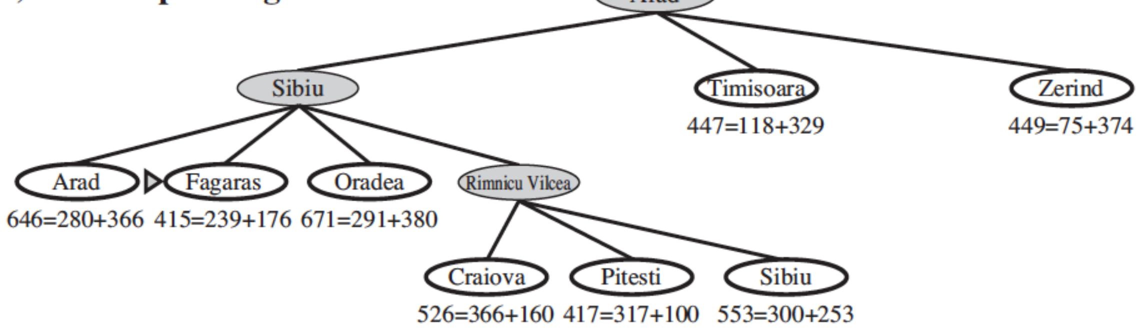



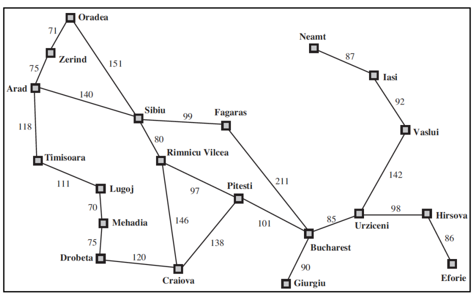

| | | | |
|---|---|---|---|
| **Arad** | 366 | **Mehadia** | 241 |
| **Bucharest** | 0 | **Neamt** | 234 |
| **Craiova** | 160 | **Oradea** | 380 |
| **Drobeta** | 242 | **Pitesti** | 100 |
| **Eforie** | 161 | **Rimnicu Vilcea** | 193 |
| **Fagaras** | 176 | **Sibiu** | 253 |
| **Giurgiu** | 77 | **Timisoara** | 329 |
| **Hirsova** | 151 | **Urziceni** | 80 |
| **Iasi** | 226 | **Vaslui** | 199 |
| **Lugoj** | 244 | **Zerind** | 374 |



# A*-4

## (d) After expanding Rimnicu Vilcea

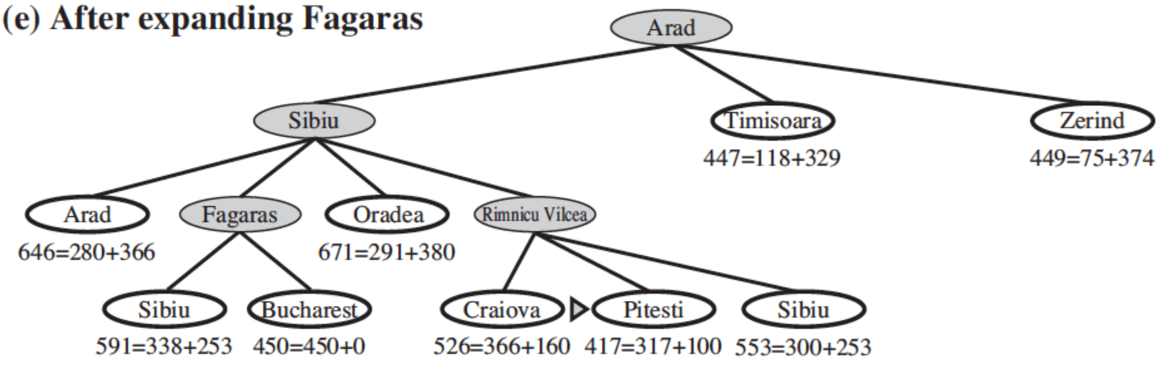


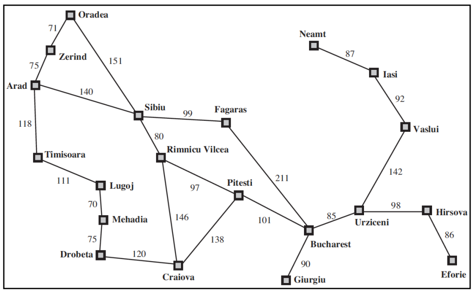




| | | | |
|---|---|---|---|
| **Arad** | 366 | **Mehadia** | 241 |
| **Bucharest** | 0 | **Neamt** | 234 |
| **Craiova** | 160 | **Oradea** | 380 |
| **Drobeta** | 242 | **Pitesti** | 100 |
| **Eforie** | 161 | **Rimnicu Vilcea** | 193 |
| **Fagaras** | 176 | **Sibiu** | 253 |
| **Giurgiu** | 77 | **Timisoara** | 329 |
| **Hirsova** | 151 | **Urziceni** | 80 |
| **Iasi** | 226 | **Vaslui** | 199 |
| **Lugoj** | 244 | **Zerind** | 374 |

# A*-5

## (e) After expanding Fagaras



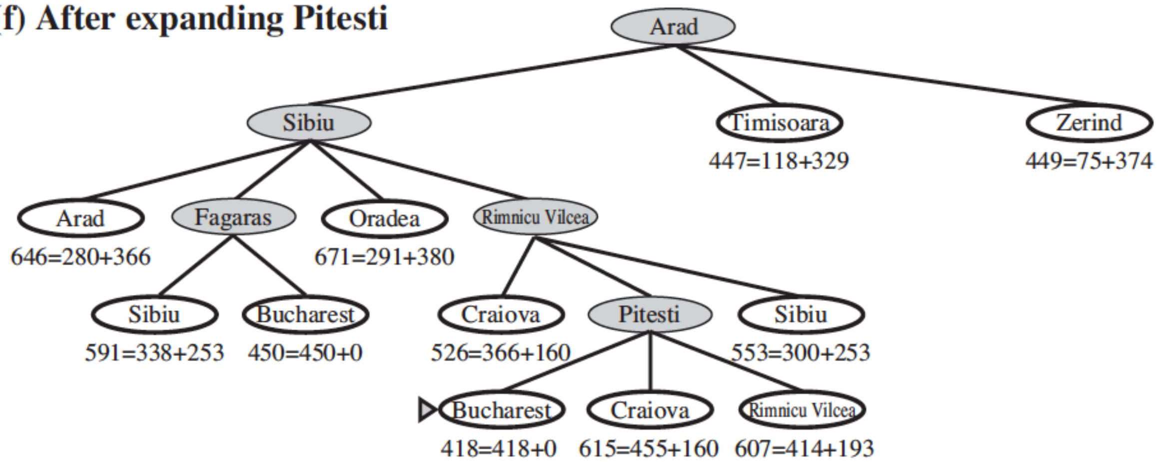



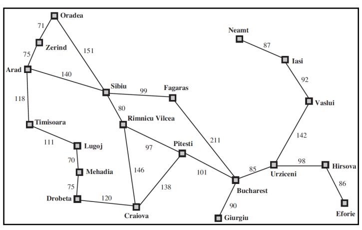

| | | | |
|---|---|---|---|
| **Arad** | 366 | **Mehadia** | 241 |
| **Bucharest** | 0 | **Neamt** | 234 |
| **Craiova** | 160 | **Oradea** | 380 |
| **Drobeta** | 242 | **Pitesti** | 100 |
| **Eforie** | 161 | **Rimnicu Vilcea** | 193 |
| **Fagaras** | 176 | **Sibiu** | 253 |
| **Giurgiu** | 77 | **Timisoara** | 329 |
| **Hirsova** | 151 | **Urziceni** | 80 |
| **Iasi** | 226 | **Vaslui** | 199 |
| **Lugoj** | 244 | **Zerind** | 374 |



# A*-6

## (f) After expanding Pitesti

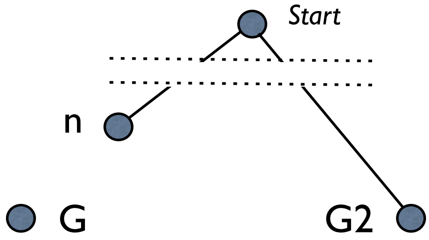


| | | | |
|---|---|---|---|
| **Arad** | 366 | **Mehadia** | 241 |
| **Bucharest** | 0 | **Neamt** | 234 |
| **Craiova** | 160 | **Oradea** | 380 |
| **Drobeta** | 242 | **Pitesti** | 100 |
| **Eforie** | 161 | **Rimnicu Vilcea** | 193 |
| **Fagaras** | 176 | **Sibiu** | 253 |
| **Giurgiu** | 77 | **Timisoara** | 329 |
| **Hirsova** | 151 | **Urziceni** | 80 |
| **Iasi** | 226 | **Vaslui** | 199 |
| **Lugoj** | 244 | **Zerind** | 374 |

# A* Proof of Optimality for Tree Search

A* using TREE-SEARCH is optimal
if h(n) is admissible




## Proof:

Assume the cost of the optimal solution is C*.
Suppose a suboptimal goal node G2 appears on the fringe.

Since G2 is suboptimal and h(G2)=0 (G2 is a goal node),
f(G2) = g(G2) + h(G2) = g(G2) > C*

Now consider the fringe node *n* that is on an optimal solution path. If h(n) does not over-estimate the cost of completing the solution path then f(n) = g(n) + h(n) < or = C*

Then f(n) <  or = C* < f(G2), so G2 will not be expanded and A* is optimal!

<u>See example:</u>
n = Pitesti (417)
G2 = Bucharest (450)



# A* Proof of Optimality for Graph Search

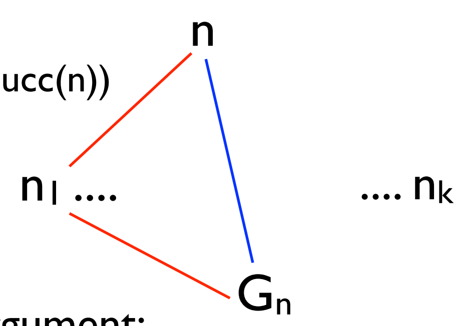
A* using GRAPH-SEARCH is optimal
if h(n) consistent (monotonic)

*Step Cost*

h(n) is consistent if h(n) < or = c(n,a, succ(n)) + h(succ(n)) for <u>all</u> a,n, succ(n)

*Step cost:* c(n,a, succ(n))

*successors(n):*




*:Goal node closest to n*

## Triangle inequality argument:

*Length of a side of a triangle is always less than the sum of the other two.*

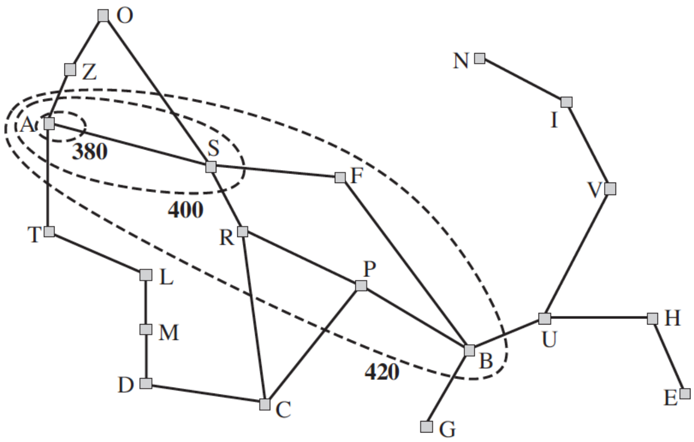
Estimated cost of getting to $G_n$ from n can not be more than going through a successor of n to $G_n$
otherwise it would violate the property that h(n) is a lower bound on the cost to reach

# Optimality of graph search

*Steps to show in the proof:*

- If h(n) is consistent, then the values f(n) along any path are non-decreasing
- Whenever A* selects a node n for expansion, the optimal path to that node has been found

If this is the case, then the values along any path are non-decreasing
and A* fans out in concentric bands of increasing f-cost




Map of Romania showing contours at f=380, f=400, and f=420 with Arad as start state. Nodes inside a given contour have f-costs < or = to the contour value.



# Some Properties of A*

- Optimal - for a given admissible heuristic (every consistent heuristic is an admissible heuristic)
- Complete - Eventually reach a contour equal to the path of the cost to the goal state.
- Optimally efficient - No other algorithm, that extends search paths from a root is guaranteed to expand fewer nodes than A* for a given heuristic function.
- The exponential growth for most practical heuristics will eventually overtake the computer (run out of memory)
  - The number of states within the goal contour is still exponential in the length of the solution.
  - There are variations of A* that bound memory....

# Admissible Heuristics

**h(n) is an admissible heuristic if it never over-estimates the cost to reach the goal from n.**

*Admissible Heuristics are optimistic because they always think the cost of solving a problem is less than it actually is.*

| | | |
|---|---|---|
| 7 | 2 | 4 |
| 5 | | 6 |
| 8 | 3 | 1 |

Start State

| | | |
|---|---|---|
| | 1 | 2 |
| 3 | 4 | 5 |
| 6 | 7 | 8 |

Goal State

The 8 Puzzle

*How would we choose an admissible heuristic for this problem?*



# 8 Puzzle Heuristics

| | | |
|---|---|---|
| 7 | 2 | 4 |
| 5 | | 6 |
| 8 | 3 | 1 |

Start State

| | | |
|---|---|---|
| | 1 | 2 |
| 3 | 4 | 5 |
| 6 | 7 | 8 |

Goal State

True solution is 26 moves. (C*)

$h_1(n)$: The number of pieces that are out of place.

(8) Any tile that is out of place must be moved at least once. Definite under estimate of moves!

$h_2(n)$: The sum of the manhatten distances for each tile that is out of place.

(3+1+2+2+2+3+3+2=18) .The manhatten distance is an under-estimate because there are tiles in the way.

## Inventing Admissible Heuristics

- A problem with fewer restrictions is called a *relaxed problem*
- The cost of an optimal solution to a relaxed problem is in fact an admissible heuristic to the original problem

If the problem definition can be written down in a formal language, there are possibilities for automatically generating relaxed problems automatically!

*Sample rule:*

A tile can move from square A to square B if A is horizontally or vertically adjacent to B and B is blank



## Some Relaxations

*Sample rule:*

A tile can move from square A to square B if A is horizontally or vertically adjacent to B and B is blank

1. A tile can move from square A to square B if A is adjacent to B
2. A tile can move from square A to square B if B is blank
3. A tile can move from square A to square B

*(1) gives us manhatten distance*

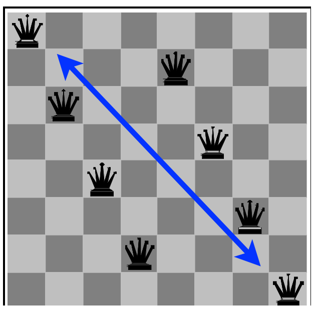
# Beyond Classical Search

## Chapter 4

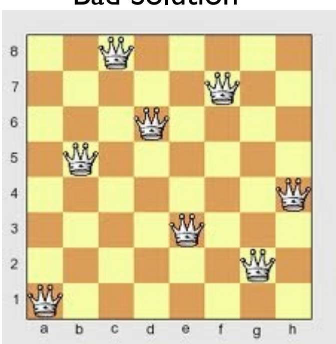
Artificial Intelligence & Integrated Computer Systems Division
Department of Computer and Information Science
Linköping University, Sweden



# Local Search: 8 Queens Problem



Bad Solution




Good Solution

**Problem**:
Place 8 queens on a chessboard such that
No queen attacks any other.

**Note**:
- The path to the goal is irrelevant!
- Complete state formulation is a straightforward representation: 8 queens, one in each column

*Candidate for use of local search!*

Artificial Intelligence & Integrated Computer Systems Division
Department of Computer and Information Science
Linköping University, Sweden

# Local Search Techniques

***Global Optimum***: The best possible solution to a problem.

***Local Optimum***: A solution to a problem that is better than all other solutions that are slightly different, but worse than the global optimum

***Greedy Algorithm***: An algorithm that always takes the best immediate, or local, solution while finding an answer. Greedy algorithms find the overall, or globally, optimal solution for some optimization problems, but may find less-than-optimal solutions for some instances of other problems. (They may also get stuck!)



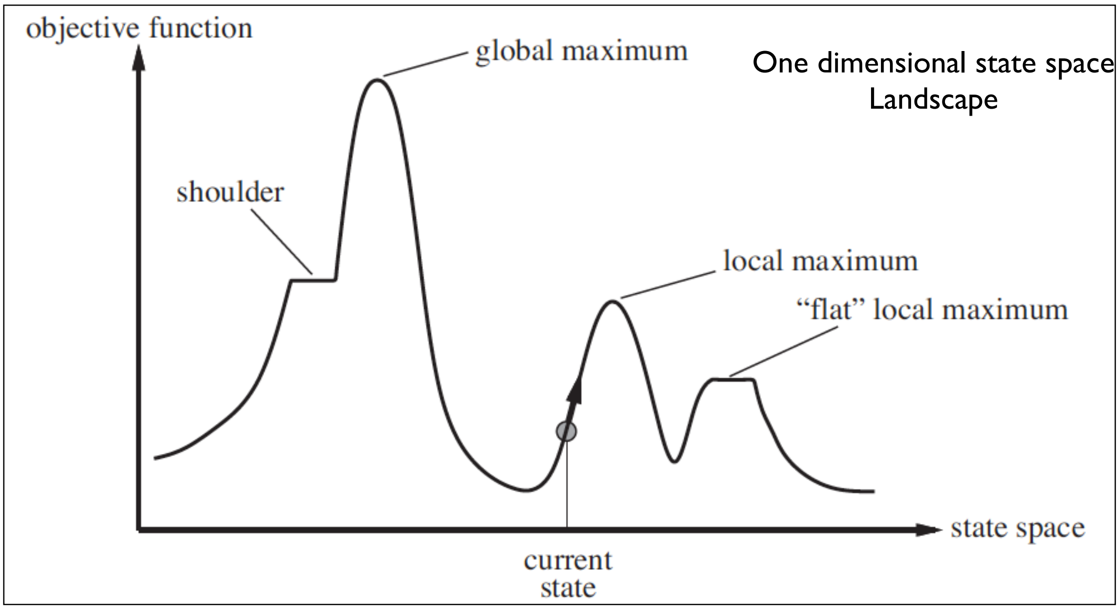
# Hill-Climbing Algorithm (steepest ascent version)

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

*current* ← MAKE-NODE(*problem*.INITIAL-STATE)
**loop do**
  *neighbor* ← a highest-valued successor of *current*
  **if** neighbor.VALUE ≤ current.VALUE **then return** *current*.STATE
  *current* ← *neighbor*

# Greedy Progress: Hill Climbing

## Aim: Find the Global Maximum



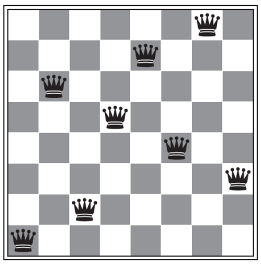

Hill Climbing: Modify the current state to try and improve it

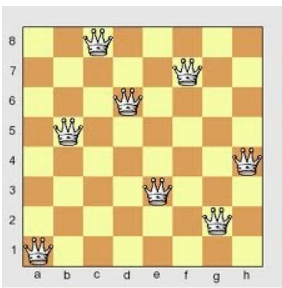


# Hill Climbing: 8 Queens

**Problem**:

Place 8 queens on a chessboard such that
No queen attacks any other.






**Successor Function**

Return all possible states generated by moving a single queen to another square in the same column. (8*7=56)

**Heuristic Cost Function**

The number of pairs of queens that are attacking each other either directly or indirectly.

Global minimum - 0

# Successor State Example

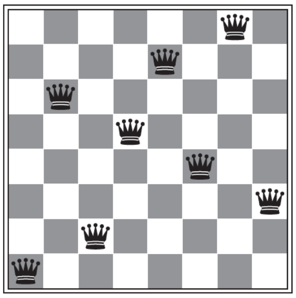
Current state: h=17

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 18 | 12 | 14 | 13 | 13 | 12 | 14 | 14 |
| 14 | 16 | 13 | 15 | 12 | 14 | 12 | 16 |
| 14 | 12 | 18 | 13 | 15 | 12 | 14 | 14 |
| 15 | 14 | 14 | ♛ | 13 | 16 | 13 | 16 |
| ♛ | 14 | 17 | 15 | ♛ | 14 | 16 | 16 |
| 17 | ♛ | 16 | 18 | 15 | ♛ | 15 | ♛ |
| 18 | 14 | ♛ | 15 | 15 | 14 | ♛ | 16 |
| 14 | 14 | 13 | 17 | 12 | 14 | 12 | 18 |

The value of h is shown for each possible successor. The 12's are the best choices for the local move. (Use steepest descent) Choose randomly on ties.

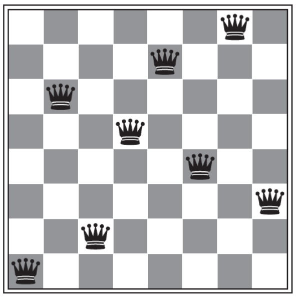
Local minimum: h=1

Any move will increase h.





# Results



State Space: $8^8 = 17 \times 10^6$ states!
Branching factor of 8*7=56

- Starting from a random 8 queen state:
  - Steepest hill ascent gets stuck 86% of the time.
  - It is quick: average of 3 steps when it fails, 4 steps when it succeeds.
  - $8^8$ = 17 million states!

How can we avoid local maxima, shoulders, flat maxima, etc.?

# Variants on Hill-Climbing

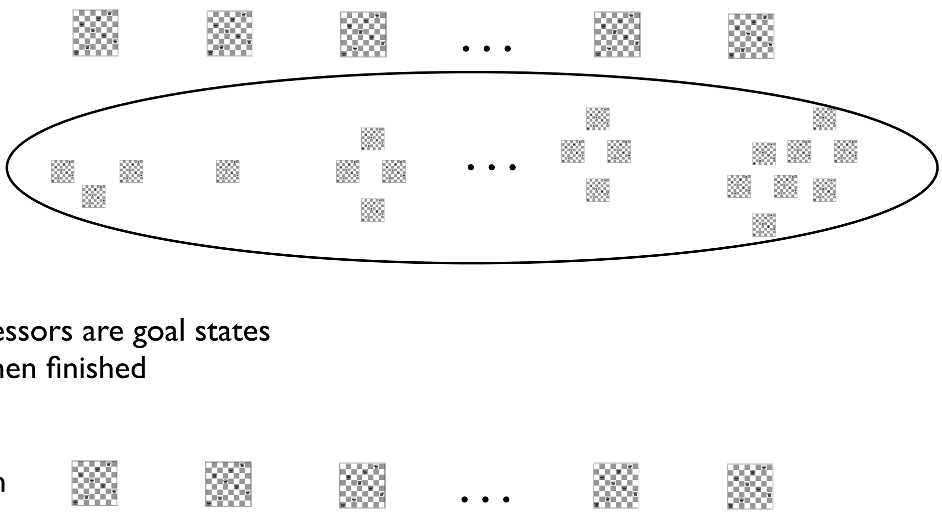
- **Stochastic hill climbing**
  - Chooses at random from among the uphill moves. Probability can vary with the steepness of the moves.
- **Simulated Annealing**
  - Combination of hill climbing and random walk.
- **Local Beam search**
  - Start with k randomly generated start states and generate their successors.
  - Choose the k best out of the union and start again.



# Local Beam Search

Start with k random states

Determine successors of all k random states




If any successors are goal states then finished

Else select k best states from union of successors and repeat

Can suffer from lack of diversity (concentrated in small region of search space). Stochastic variant: choose k successors at random with probability of choosing the successor being an increasing function of its value.

# Simulated Annealing

- Escape local maxima by allowing "bad" moves
  - *Idea*: but gradually decrease their size and frequency
  - Origin of concept: metallurgical annealing
- Bouncing ball analogy (gradient descent):
  - Shaking hard (= high temperature)
  - Shaking less (= lower the temperature)
- If *Temp* decreases slowly enough, best state is reached





# Simulated Annealing

**function** SIMULATED-ANNEALING(*problem*, *schedule*) **returns** a solution state
  **inputs**: *problem*, a problem
    *schedule*, a mapping from time to "temperature"

  *current* ← MAKE-NODE(*problem*.INITIAL-STATE)
  **for** $t = 1$ **to** $\infty$ **do**
    $T \leftarrow$ *schedule*($t$)
    **if** $T = 0$ **then return** *current*
    *next* ← a randomly selected successor of *current*
    $\Delta E \leftarrow$ *next*.VALUE – *current*.VALUE
    **if** $\Delta E > 0$ **then** *current* ← *next*
    **else** *current* ← *next* only with probability $e^{\Delta E/T}$

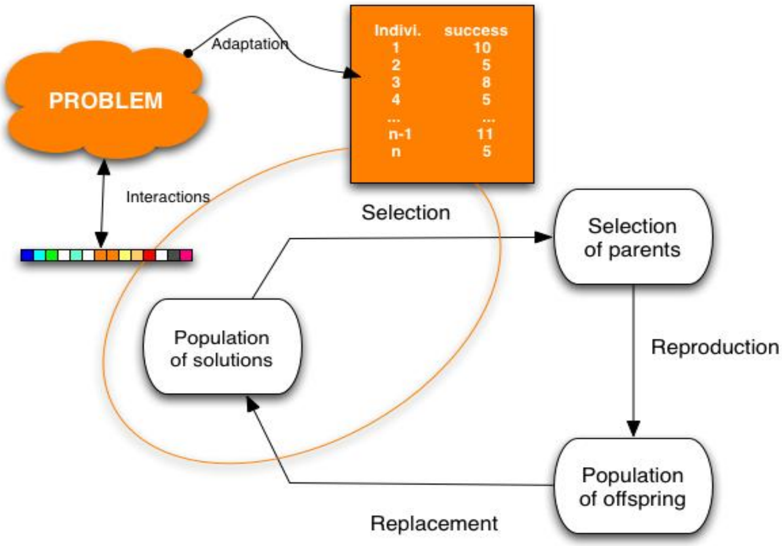
*The probability decreases exponentially with the "badness" of the move - the amount Delta E by which the evaluation is worsened.*

*The probability also decreases as the "temperature" T goes down: "bad" moves are more likely to be allowed at the start when the temperature is high, and more unlikely As T decreases.*

Artificial Intelligence & Integrated Computer Systems Division
Department of Computer and Information Science
Linköping University, Sweden

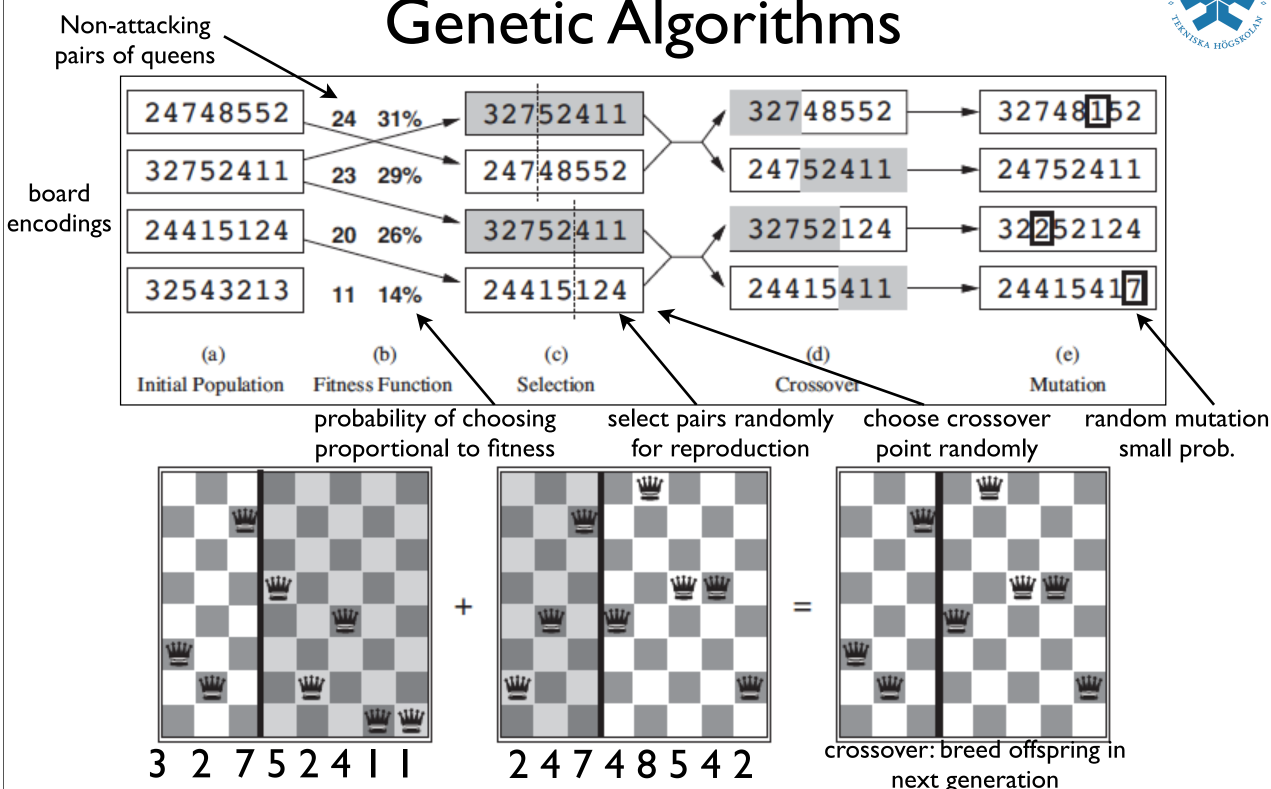
# Genetic Algorithms

Variant of Local Beam Search with the addition of sexual recombination






# Genetic Algorithms





Artificial Intelligence & Integrated Computer Systems Division
Department of Computer and Information Science
Linköping University, Sweden

# Genetic Algorithms

```
function GENETIC-ALGORITHM(population, FITNESS-FN) returns an individual
  inputs: population, a set of individuals
          FITNESS-FN, a function that measures the fitness of an individual

  repeat
     new_population ← empty set
     for i = 1 to SIZE(population) do
        x ← RANDOM-SELECTION(population, FITNESS-FN)
        y ← RANDOM-SELECTION(population, FITNESS-FN)
        child ← REPRODUCE(x, y)
        if (small random probability) then child ← MUTATE(child)
        add child to new_population
     population ← new_population
  until some individual is fit enough, or enough time has elapsed
  return the best individual in population, according to FITNESS-FN
```

```
function REPRODUCE(x, y) returns an individual
  inputs: x, y, parent individuals

  n ← LENGTH(x); c ← random number from 1 to n
  return APPEND(SUBSTRING(x, 1, c), SUBSTRING(y, c + 1, n))
```



# Using A* in Path Planning for a UAV

(If time permits)