Local Search Algorithms

(Chapter 4)

Mustafa Jarrar

Birzeit University
mjarrar@birzeit.edu
www.jarrar.info

Watch this lecture and download the slides

Download: http://www.jarrar.info/courses/AI/Jarrar.LectureNotes.Ch4.LocalSearch.pdf

Watch lectures: http://www.jarrar.info/courses/AI

Acknowledgement:
This lecture is based on (but not limited to) chapter 4 in "S. Russell and P. Norvig: *Artificial Intelligence: A Modern Approach*".

Local Search Algorithms

In this lecture:

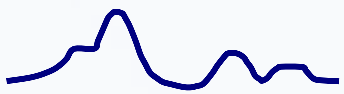
- Part 1: **What/Why Local Search**
- Part 2: Hill-Climbing Search
- Part 3: Simulated Annealing Search
- Part 4: Genetic Algorithms

Local Search Algorithms

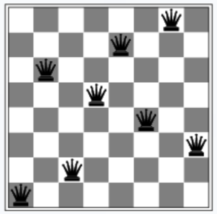
In many optimization problems, the path to the goal is irrelevant; the goal state itself is the solution.

Examples:

- to reduce cost, as in cost functions



- to reduce conflicts, as in *n*-queens



The idea: keep a single "current" state, try to improve it according to an objective function.

Local search algorithms:

- Use little memory
- Find reasonable solutions in large infinite spaces

Local Search Algorithms

- Local search can be used on problems that can be formulated as finding a solution maximizing a criterion among a number of candidate solutions.

- Local search algorithms move from solution to solution in the space of candidate solutions (the *search space*) until a solution deemed optimal is found or a time bound is elapsed.

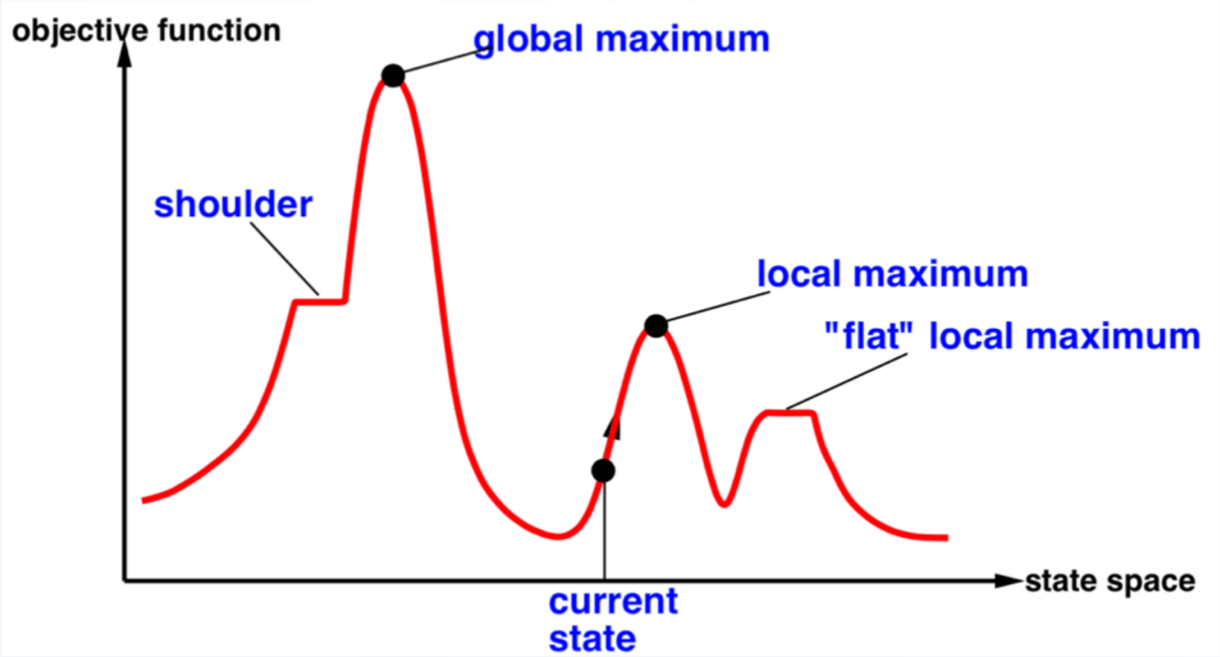
- For example, the travelling salesman problem, in which a solution is a cycle containing all nodes of the graph and the target is to minimize the total length of the cycle. i.e. a solution can be a cycle and the criterion to maximize is a combination of the number of nodes and the length of the cycle.

- A local search algorithm starts from a candidate solution and then iteratively moves to a neighbor solution.

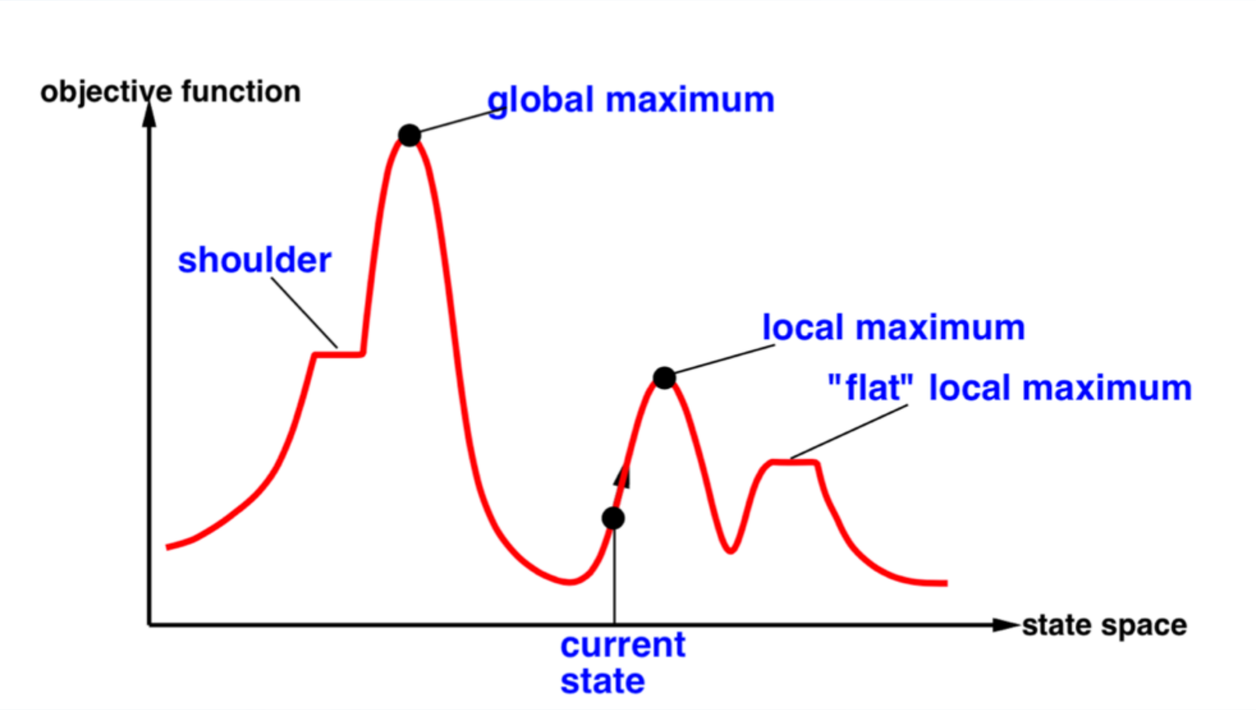
Local Search Algorithms

Terminate on a time bound or if the situation is not improved after number of steps.

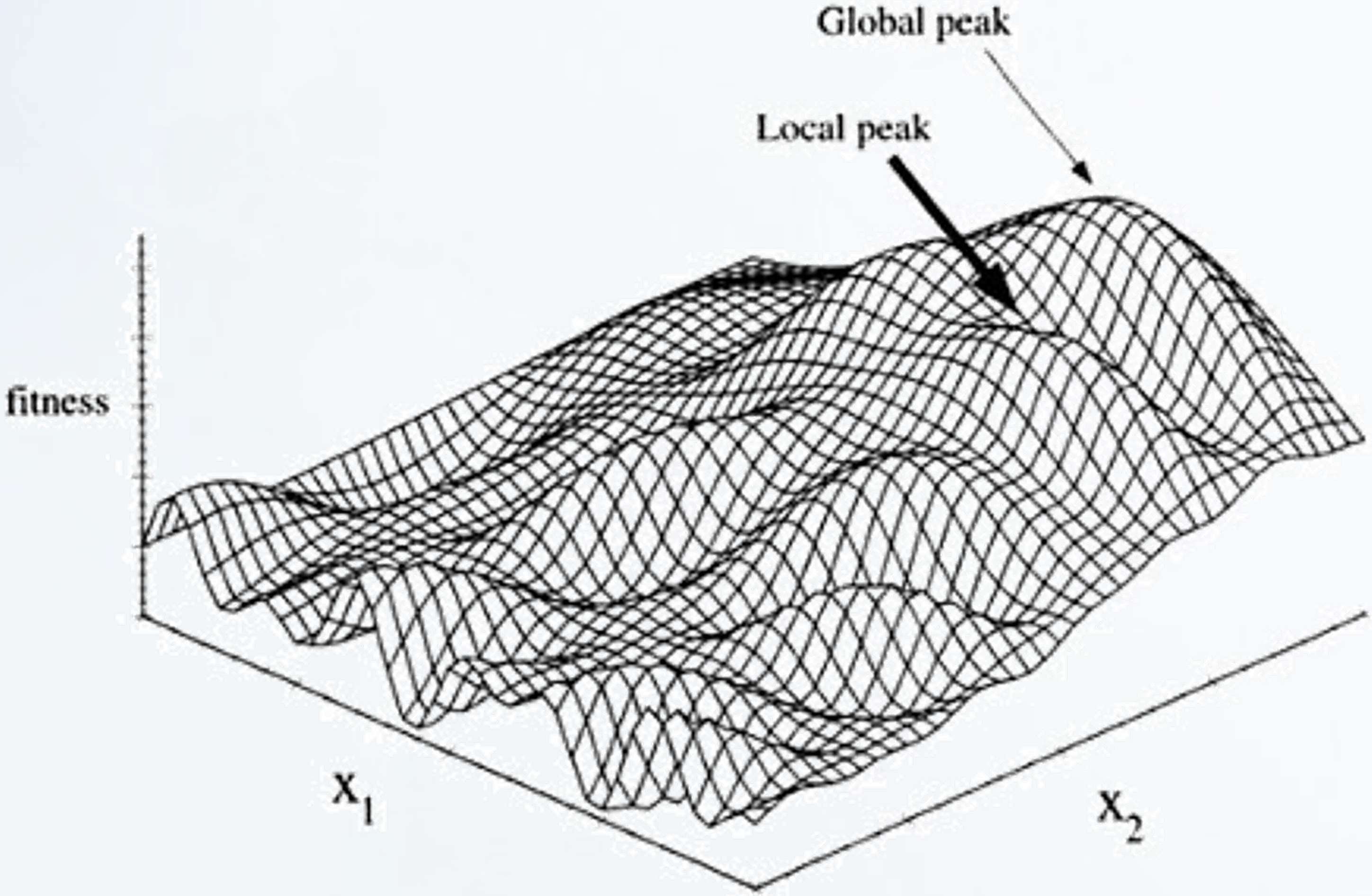
Local search algorithms are typically incomplete algorithms, as the search may stop even if the best solution found by the algorithm is not optimal.



Search Landscape (two-dimension)



Search Landscape (three-dimensions)



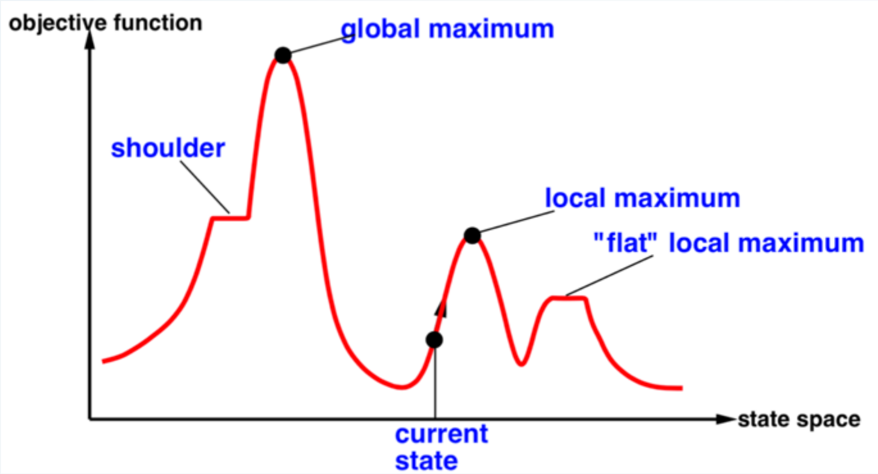
Local Search Algorithms

In this lecture:

- Part 1: What/Why Local Search Algorithms
- Part 2: **Hill-Climbing Search**
- Part 3: Simulated Annealing Search
- Part 4: Genetic Algorithms in nutshell

Hill-Climbing Search

- Continually moves in the direction of increasing value (i.e., uphill).
- Terminates when it reaches a “peak”, no neighbor has a higher value.
- Only records the state and its objective function value.
- Does not look ahead beyond the immediate.



Sometimes called
Greedy Local Search

- **Problem**: can get stuck in *local maxima*,
- Its success depends very much on the shape of the state-space land-scape: if there are few local maxima, random-restart hill climbing will find a “good” solution very quickly.

Hill-Climbing Search Algorithm

Based on [1]

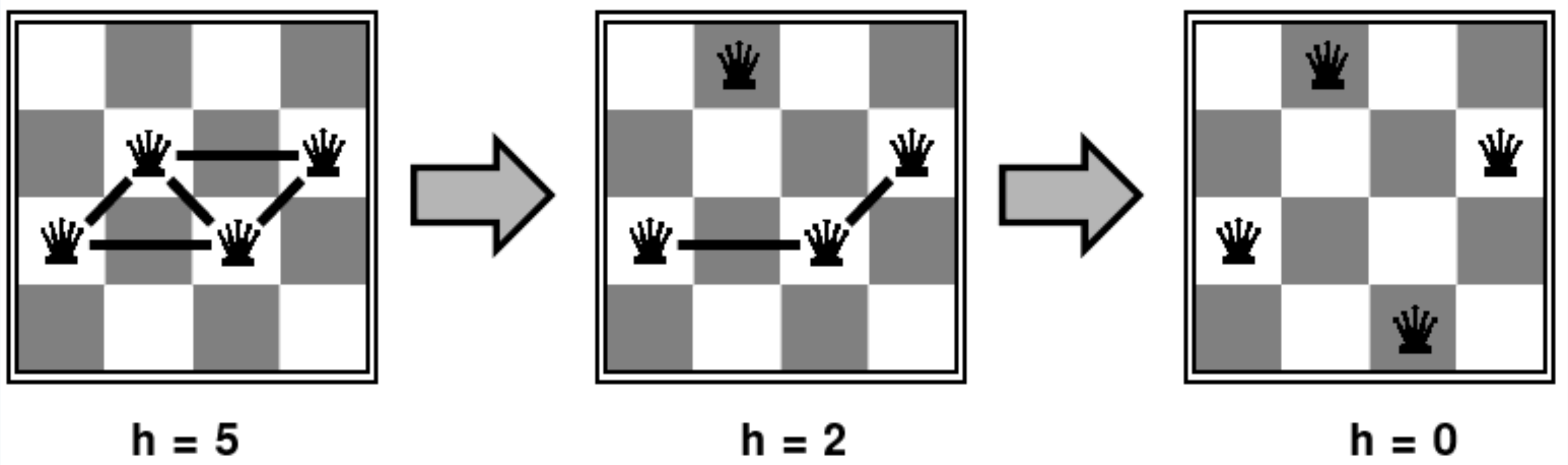
```
function HILL-CLIMBING(problem) returns a state that is a local maximum

  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
```

The hill-climbing search algorithm, which is the **most basic** local search technique. At each step the current node is replaced by the best neighbor; in this version, that means the neighbor with the highest VALUE, but if a heuristic cost estimate h is used, we would find the neighbor with the lowest h.

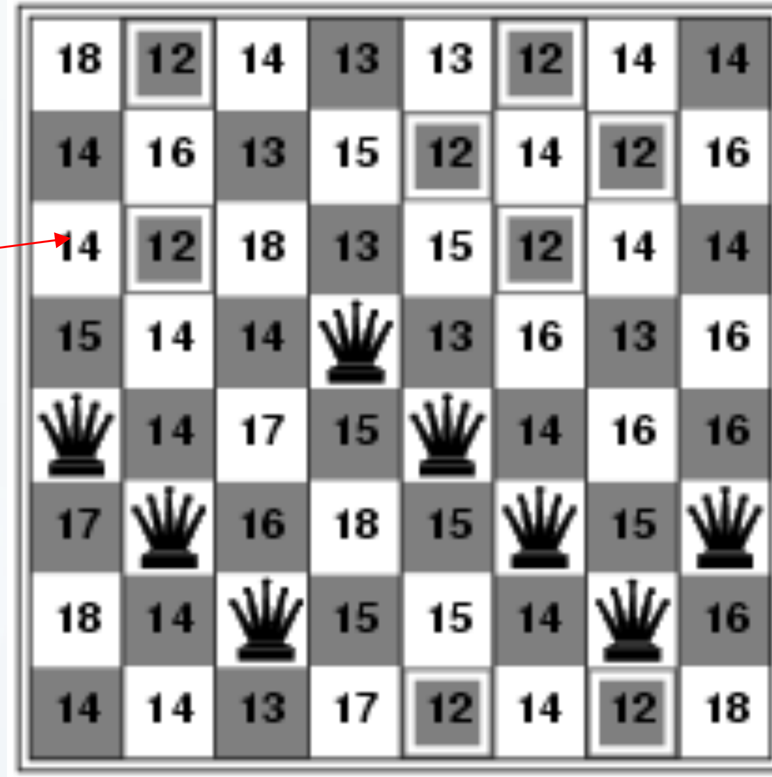
Example: *n*-queens

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

Move a queen to reduce number of conflicts.



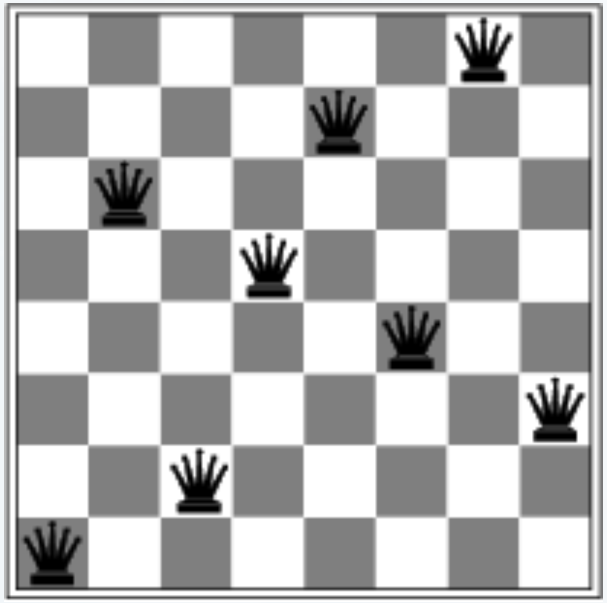
Example: 8-queens



Each number indicates h if we move a queen in its corresponding column

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

Example: *n*-queens



A local minimum with $h = 1$

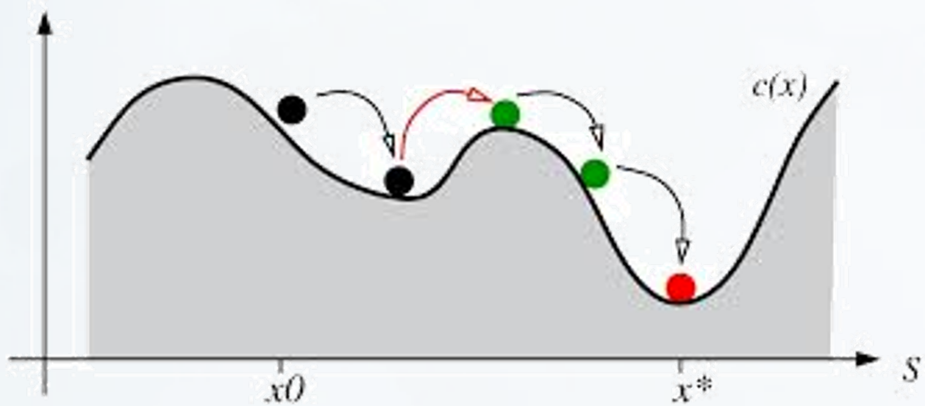
Local Search Algorithms

In this lecture:

- Part 1: What/Why Local Search Algorithms
- Part 2: Hill-Climbing Search
- Part 3: **Simulated Annealing Search**
- Part 4: Genetic Algorithms in nutshell

Simulated Annealing Search

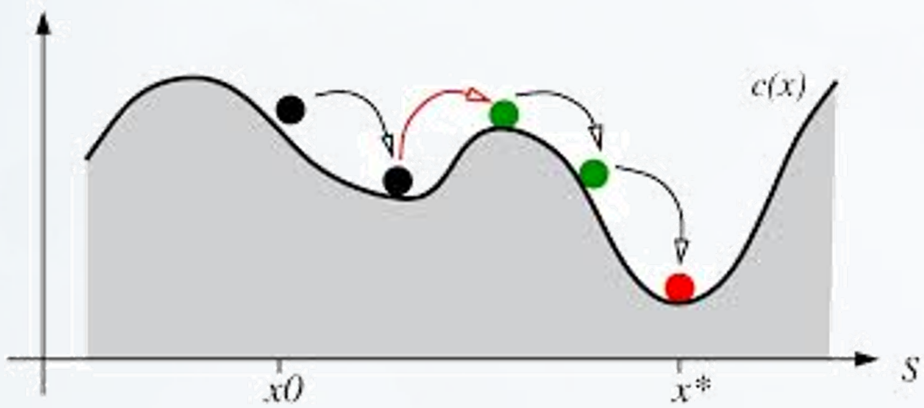
Based on [1]



To avoid being stuck in a local maxima, it tries randomly (using a <u>probability function)</u> to move to another state, if this new state is better it moves into it, otherwise try another move… and so on.

Simulated Annealing Search

Based on [1]



Terminates when finding an acceptably good solution in a fixed amount of time, rather than the best possible solution.

Locating a good approximation to the global minimum of a given function in a large search space.

Widely used in VLSI layout, airline scheduling, etc.

Properties of Simulated Annealing Search

The problem with this approach is that the neighbors of a state are not guaranteed to contain any of the existing better solutions which means that failure to find a better solution among them does not guarantee that no better solution exists.

It will not get stuck to a local optimum.

If it runs for an infinite amount of time, the global optimum will be found.

Local Search Algorithms

In this lecture:

- ❑ Part 1: What/Why Local Search Algorithms
- ❑ Part 2: Hill-Climbing Search
- ❑ Part 3: Simulated Annealing Search
- Part 4: **Genetic Algorithms in nutshell**

Genetic Algorithms

- Inspired by evolutionary biology and natural selection, such as inheritance.
- Evolves toward better solutions.
- A successor state is generated by combining two parent states, rather by modifying a single state.
- Start with k randomly generated states (population), Each state is an individual.

Genetic Algorithms

- 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.
- Commonly, the algorithm terminates when either a maximum number of generations has been produced, or a satisfactory fitness level has been reached for the population.

Genetic Algorithms

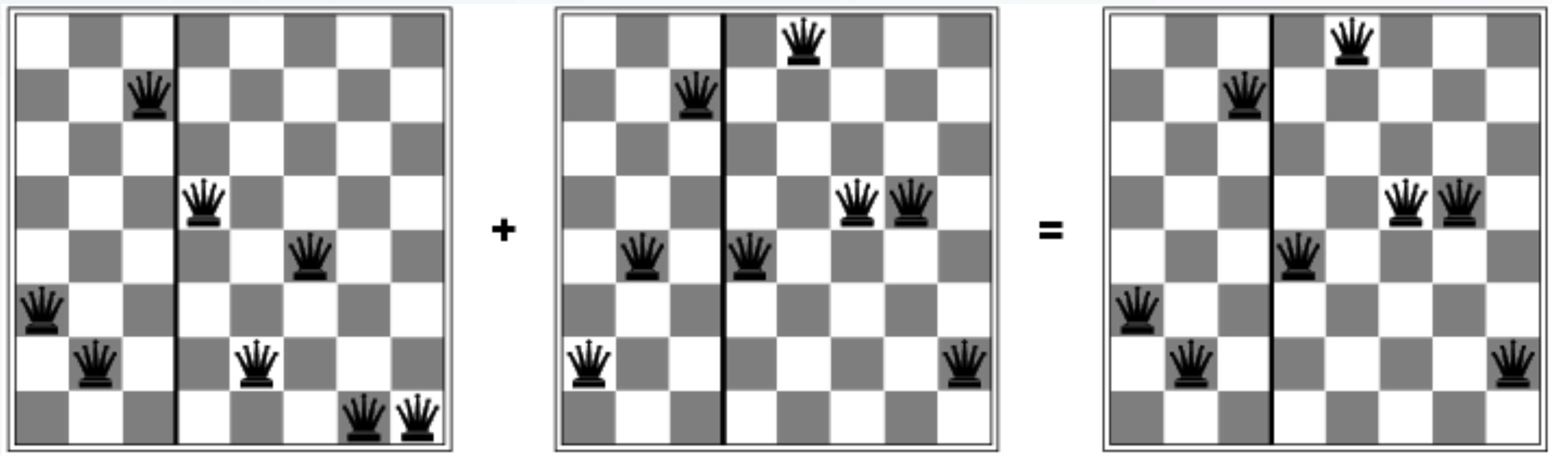
```
function GENETIC-ALGORITHM(population, FITNESS-FN) returns an individual
  inputs: population, a set of individuals
          FITNESS-FN, a function that measures the fitness of an individual

  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))
```

The 8-queen Problem

Try to better position the queens using the genetic algorithm. A better state is generated by combining two parent states.

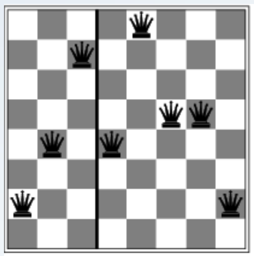
The good genes (features) of the parents are passed onto the children.



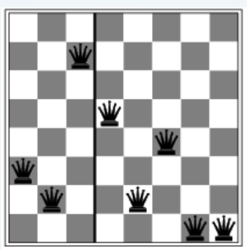
The 8-queen Problem

Represent individuals (chromosomes) :

Can be represented by a string digits 1 to 8, that represents the position of the 8 queens in the 8 columns.



24748552



32752411

24415124

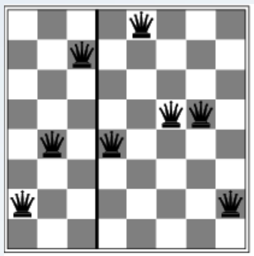
.
.
.

32543213

The 8-queen Problem

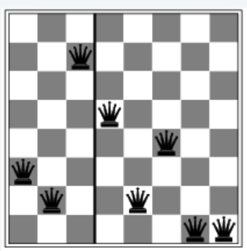
Fitness Function :

Possible fitness function is the number of non-attacking pairs of queens. (min = 0, max = 8 × 7/2 = 28)



24

24748552



23

32752411

20

24415124

.

.

11

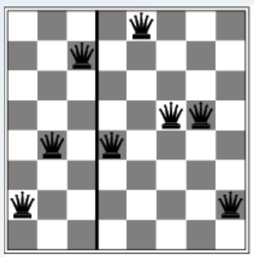
.

32543213

The 8-queen Problem

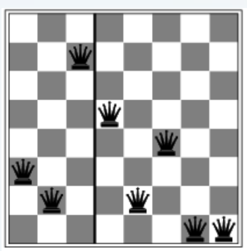
Fitness Function :

Calculate the probability of being regenerated in next generation. For example: 24/(24+23+20+11) = 31% , 23/(24+23+20+11) = 29% , etc.



24

24748552 31%



23

32752411 29%

.

20

24415124 26%

.

11

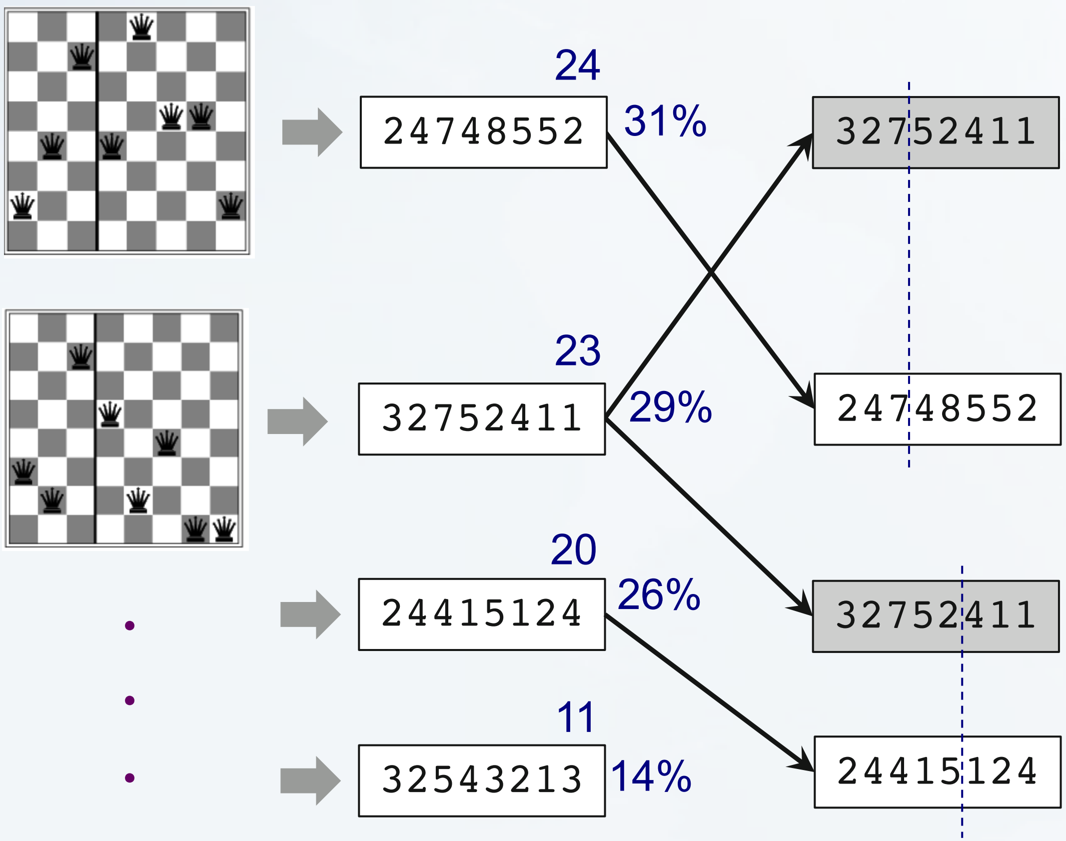
.

32543213 14%

The 8-queen Problem

Selection:

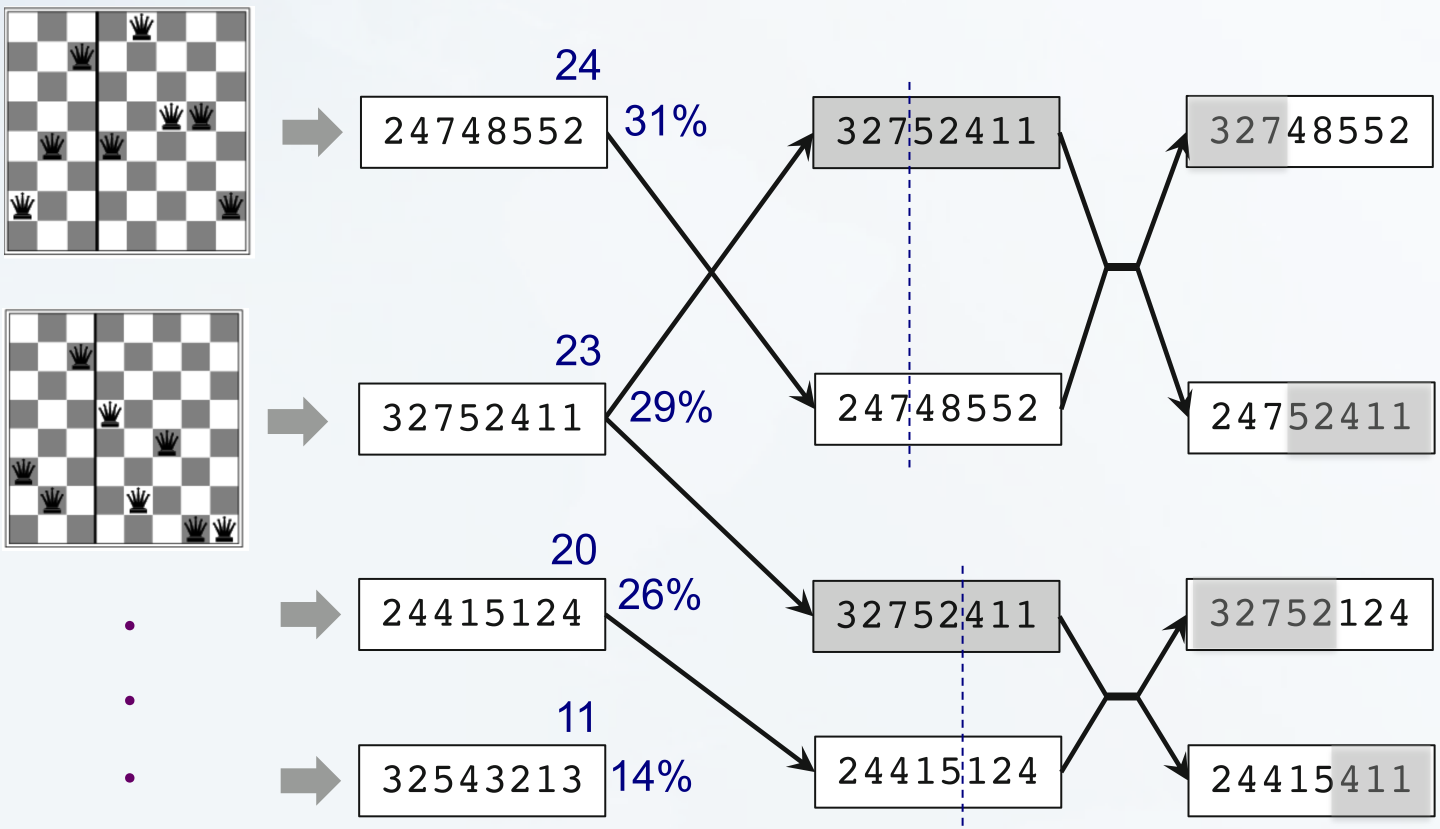
Pairs of individuals are selected at random for reproduction w.r.t. some probabilities. Pick a crossover point per pair.



The 8-queen Problem

Crossover

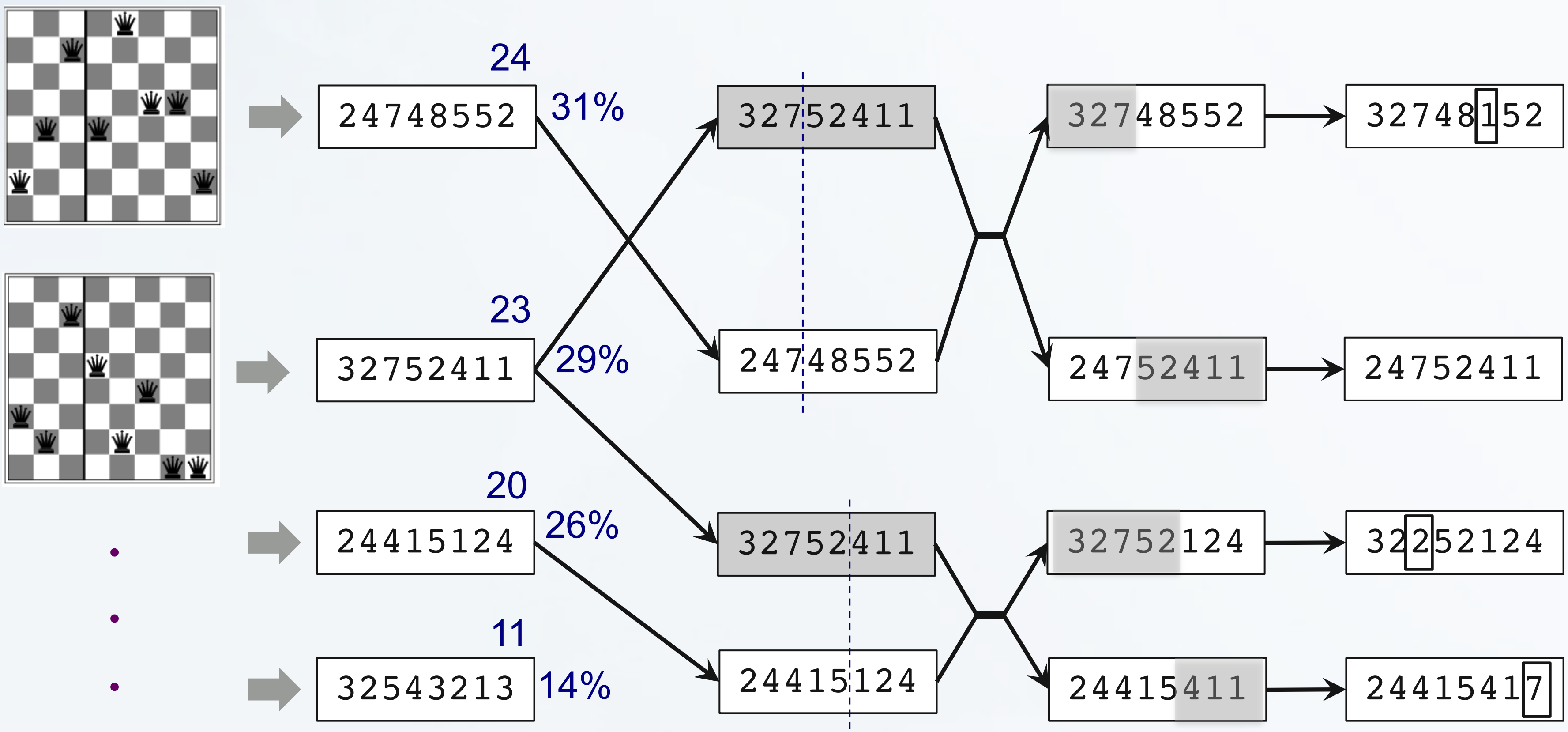
A **crossover** point is chosen randomly in the string. **Offspring** are created by crossing the parents at the crossover point.



The 8-queen Problem

Mutation

Each element in the string is also subject to some mutation with a small probability.



References

[1] S. Russell and P. Norvig: *Artificial Intelligence: A Modern Approach* Prentice Hall, *2003, Second Edition*

[2] http://en.wikipedia.org/wiki/SMA*

[3] Moonis Ali: Lecture Notes on Artificial Intelligence
http://cs.txstate.edu/~ma04/files/CS5346/SMA%20search.pdf

[4] Max Welling: Lecture Notes on Artificial Intelligence
https://www.ics.uci.edu/~welling/teaching/ICS175winter12/A-starSearch.pdf

[5] Kathleen McKeown: Lecture Notes on Artificial Intelligence
http://www.cs.columbia.edu/~kathy/cs4701/documents/InformedSearch-AR-print.ppt

[6] Franz Kurfess: Lecture Notes on Artificial Intelligence
http://users.csc.calpoly.edu/~fkurfess/Courses/Artificial-Intelligence/F09/Slides/3-Search.ppt