Artificial Intelligence

CSC-462

Muhammad Najam Dar

Today's Learning

Informed Search (Local Search)
Hill Climbing
Local Beam Search
Genetic Algorithms

Local search

- Search algorithms seen so far are designed to **explore search spaces systematically**.
- Problems: observable, deterministic, known environments where the solution is a sequence of actions.
- Real-World problems are more complex.
- When a goal is found, the path to that goal constitutes a solution to the problem. But, depending on the applications, the path may or may not matter.
- If the path does not matter/systematic search is not possible, then consider another class of algorithms.
- The goal state itself is the solution.

Local search

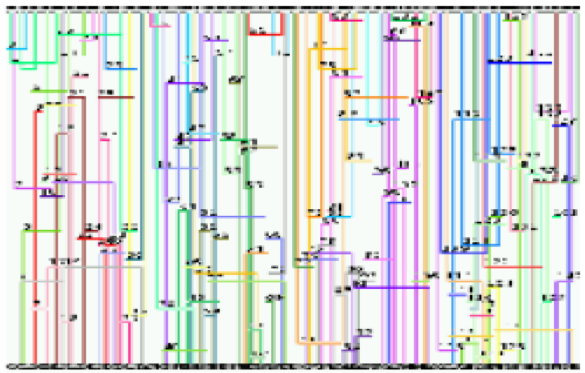
- In such cases, we can use iterative improvement algorithms, **Local search**.

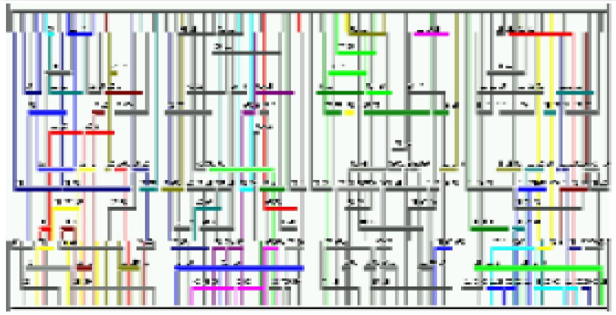
- Also useful in pure **optimization problems** where the goal is to find the best state according to an **optimization function**.

- **Examples:**
  State space = set of configurations
  Find a configuration satisfying your constraints, e.g., n-queens
  - Integrated circuit design, telecommunications network optimization, etc.
  - N-puzzle or 8-queen: what matters is the final configuration of the puzzle, not the intermediary steps to reach it.

Local search

Optimize the number of products purchased by an E-Commerce user
State: Action taken by the user plus the resulting page-view
No track is kept of the path costs between the states
All that is seen is whether the user is buying more products (or not).


Lots of Chip Real-estate


Same connectivity, much less space

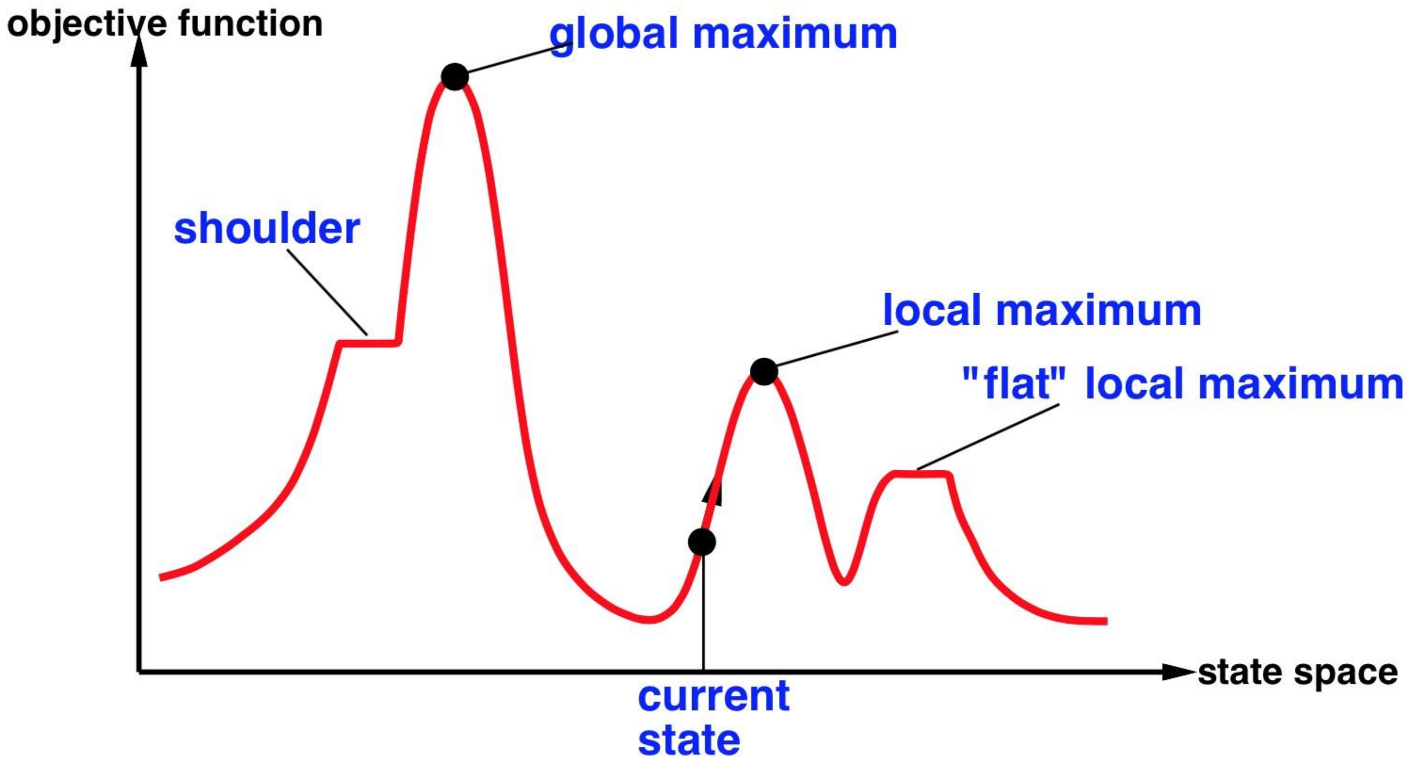
Local search

- **Idea**: keep a single "current" state, and try to improve it.
- Move only to neighbors of that node.
- **Advantages**:
  1. No need to maintain a search tree.
  2. Use very little memory.
  3. Can often find good enough solutions in continuous or large state spaces.

Local search

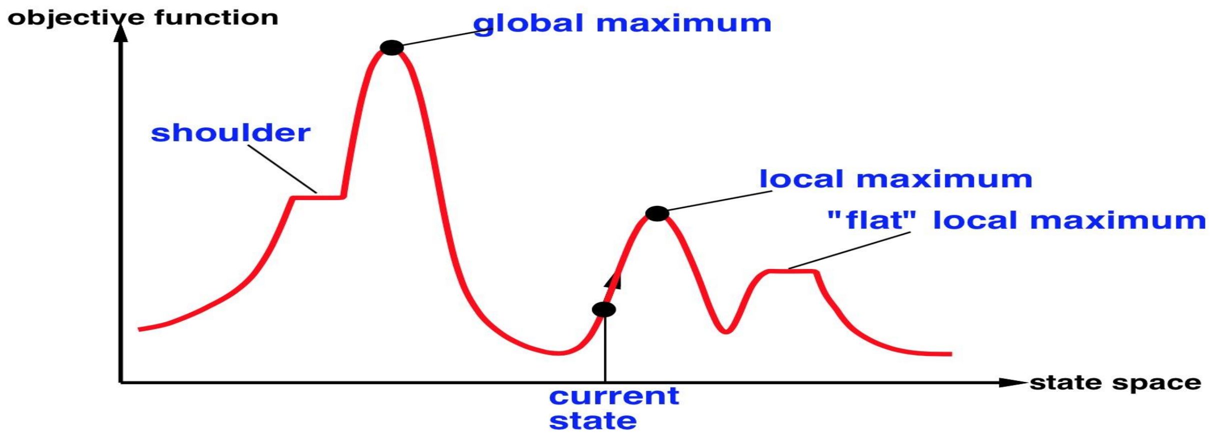
- **Local Search Algorithms:**
  - Hill climbing (steepest ascent/descent).
  - Local beam search.
  - Genetic algorithms: inspired by evolutionary biology.

Local search



State space landscape

Hill climbing



- Also called **greedy local search**.
- Looks only to immediate good neighbors and not beyond.
- Search moves uphill: moves in the direction of increasing elevation/value to find the top of the mountain.
- Terminates when it reaches a **peak**.
- Can terminate with a local maximum, global maximum or can get stuck and no progress is possible.
- A **node is a state and a value**.

Hill climbing

"Like climbing Everest in thick fog with amnesia“

A loop that continually moves in the direction of increasing value, i.e., uphill

Terminates when it reaches a peak where no neighbor has a higher value

Fog with Amnesia: Doesn’t look ahead beyond the immediate neighbors of the current state.

Unfortunately, hill-climbing

- Can get stuck in local maxima
- Can be stuck by ridges (a series of local maxima that occur close together)
- Can be stuck by plateaux (a flat area in the state space landscape)
  - Shoulder: if the flat area rises uphill later on
  - Flat local maximum: no uphill rise exists.

Hill climbing

```
function HILL-CLIMBING(initialState)
    returns State that is a local maximum

    initialize current with initialState

    loop do
        neighbor = a highest-valued successor of current

        if neighbor.value $\leq$ current.value:
            return current.state

        current = neighbor
```

Hill climbing

Other variants of hill climbing include

- **Sideways moves** escape from plateaux where best successor has same value as the current state.

- **Random-restart** hill climbing overcomes local maxima: keep trying! (either find a goal or get several possible solution and pick the max).
- **Steepest Ascent** Backtracking and jump.
- **Stochastic** hill climbing chooses at random among the uphill moves.

Hill climbing

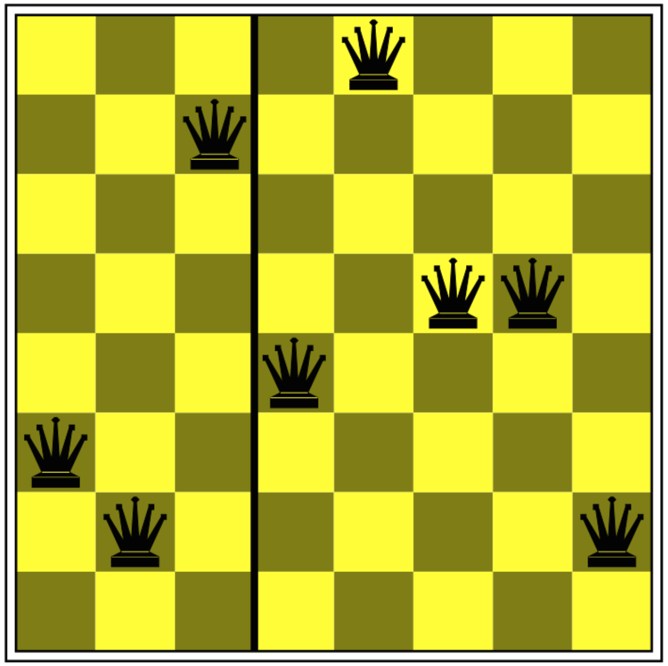
- **Hill climbing** effective in general but depends on shape of the landscape.
- Successful in many real-problems after a reasonable number of restarts.
- **Local beam search** maintains *k* states instead of one state.
- Select the *k* best successor, and useful information is passed among the states.
- **Stochastic beam search** choose *k* successors are random.
- Helps alleviate the problem of the the states agglomerating around the same part of the state space.

Genetic algorithms

- **Genetic algorithms (GA)** is a variant of stochastic beam search.
- Successor states are generated by combining two parents rather by modifying a single state.
- The process is inspired by **natural selection**.
- Starts with *k* **randomly generated states**, called **population**. Each state is an **individual**.
- An individual is usually represented by a **string** of 0's and 1's, or digits, a finite set.
- The objective function is called **fitness function**: better states have high values of fitness function.

Genetic algorithms

- In the 8-queen problem, an individual can be represented by a **string** digits 1 to 8, that represents the position of the 8 queens in the 8 columns.



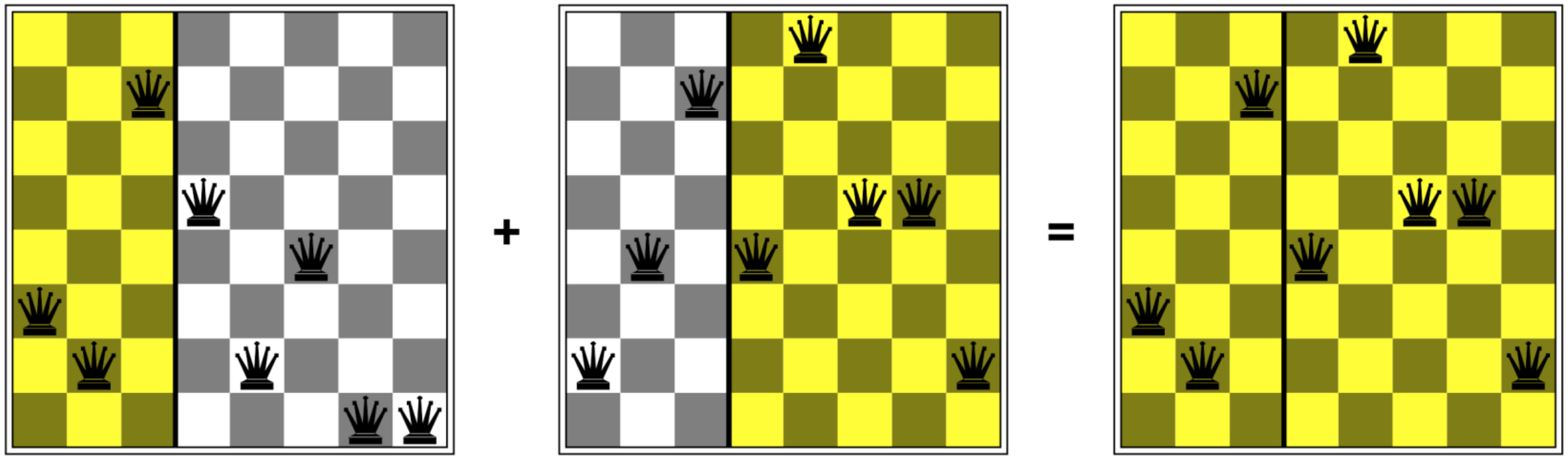
Genetic algorithms

- The objective function is called **fitness function**: better states have high values of fitness function.
- Possible fitness function is the **number of non-attacking pairs of queens**.
- Fitness function of the solution: 28.

Genetic algorithms

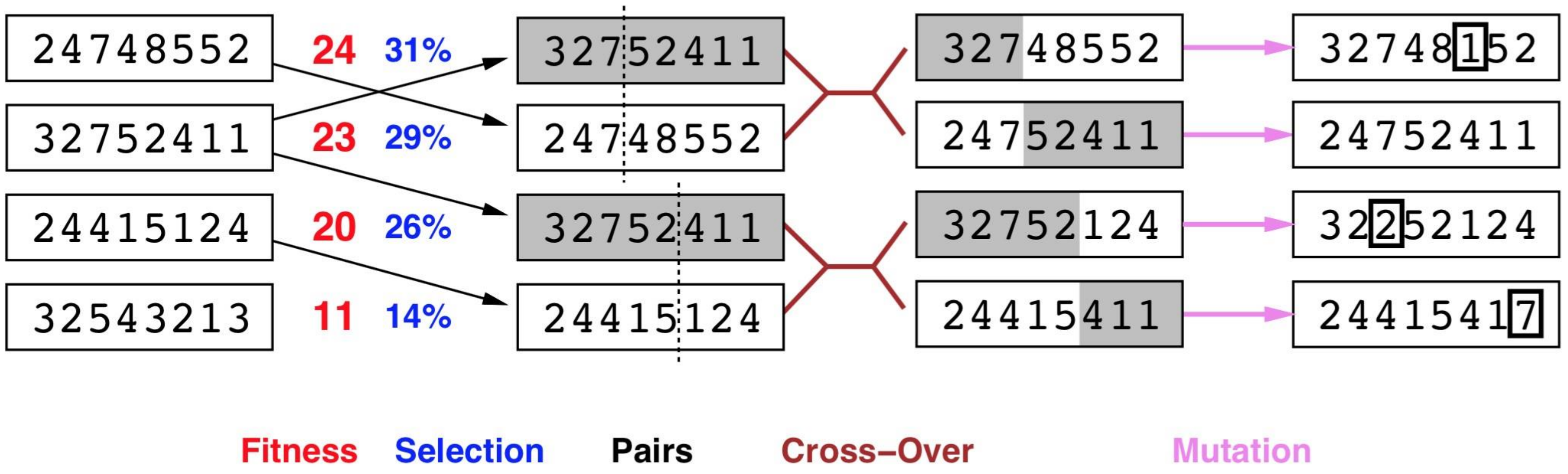
- Pairs of individuals are selected at random for **reproduction** w.r.t some probabilities.
- A **crossover** point is chosen randomly in the string.
- **Offspring** are created by crossing the parents at the crossover point.
- Each element in the string is also subject to some **mutation** with a small probability.

Genetic algorithms

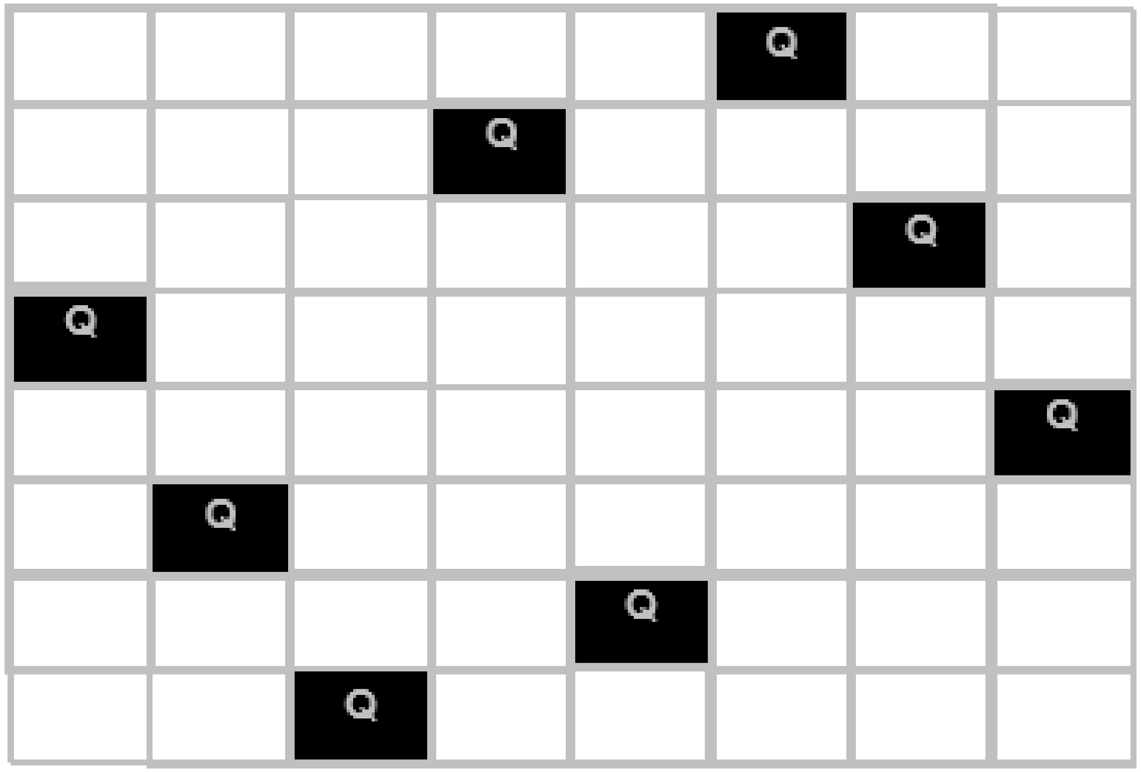


Genetic algorithms

Generate successors from pairs of states.



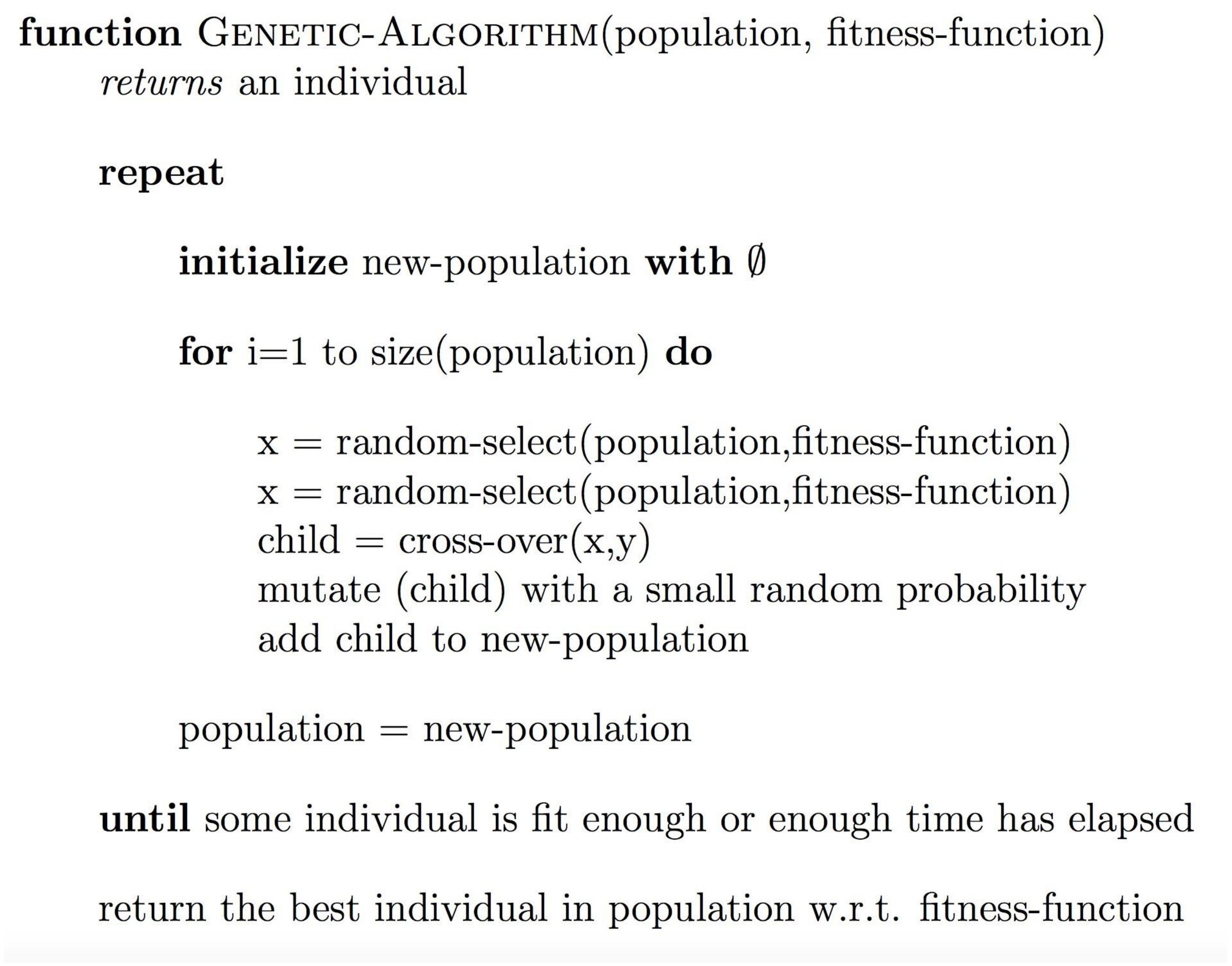
Solution!



8

4 6 8 2 7 1 3 5

Genetic algorithms



```
function GENETIC-ALGORITHM(population, fitness-function)
    returns an individual

    repeat

        initialize new-population with $\emptyset$

        for i=1 to size(population) do

            x = random-select(population,fitness-function)
            x = random-select(population,fitness-function)
            child = cross-over(x,y)
            mutate (child) with a small random probability
            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 w.r.t. fitness-function
```

Genetic algorithms

- Common terminating conditions are:
  - A solution is found that satisfies minimum criteria
  - Fixed number of generations reached
  - Allocated budget (computation time/money) reached
  - The highest ranking solution's fitness is reaching or has reached a plateau such that successive iterations no longer produce better results

Example (initialization)

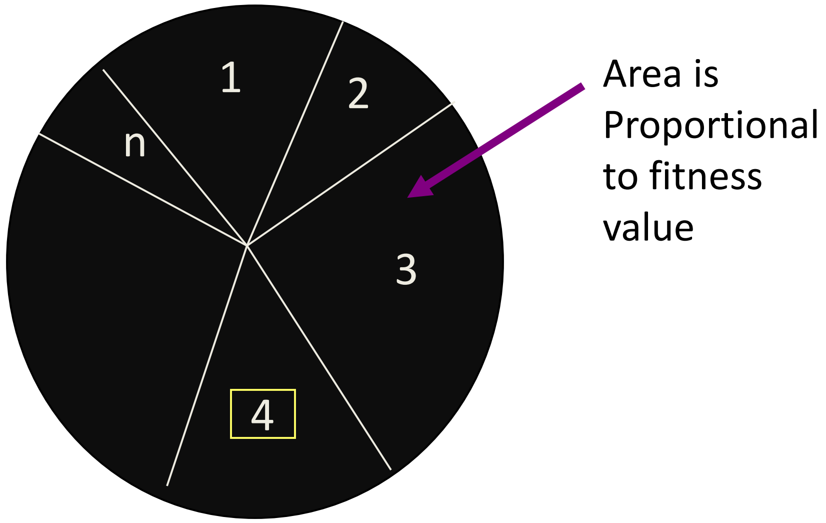
We toss a fair coin 60 times and get the following initial population:

$s_1 = 1111010101$	$f(s_1) = 7$
$s_2 = 0111000101$	$f(s_2) = 5$
$s_3 = 1110110101$	$f(s_3) = 7$
$s_4 = 0100010011$	$f(s_4) = 4$
$s_5 = 1110111101$	$f(s_5) = 8$
$s_6 = 0100110000$	$f(s_6) = 3$

Example (selection1)

Next we apply fitness proportionate selection with the roulette wheel method:

We repeat the extraction as many times as the number of individuals we need to have the same parent population size (6 in our case)



Example (selection2)

Suppose that, after performing selection, we get the following population:

s_1\`	= 1111010101	(s_1)
s_2\`	= 1110110101	(s_3)
s_3\`	= 1110111101	(s_5)
s_4\`	= 0111000101	(s_2)
s_5\`	= 0100010011	(s_4)
s_6\`	= 1110111101	(s_5)

Example (crossover1)

Next we mate strings for crossover. For each couple we decide according to crossover probability (for instance 0.6) whether to actually perform crossover or not

Suppose that we decide to actually perform crossover only for couples (s_1\`, $s_2$\`) and (s_5\`, $s_6$\`). For each couple, we randomly extract a crossover point, for instance 2 for the first and 5 for the second

Example (crossover2)

Before crossover:

s_1` = 1111010101

s_2` = 1110110101

s_5` = 0100010011

s_6` = 1110111101

After crossover:

s_1`` = 1110110101

s_2`` = 1111010101

s_5`` = 0100011101

s_6`` = 1110110011

Example (mutation1)

The final step is to apply random mutation: for each bit that we are to copy to the new population we allow a small probability of error (for instance 0.1)

Before applying mutation:

s_1`` = 1110110101

s_2`` = 1111010101

s_3`` = 1110111101

s_4`` = 0111000101

s_5`` = 0100011101

s_6`` = 1110110011

Example (mutation2)

After applying mutation:

$s_1``` = 1110100101 \quad f(s_1```) = 6$

$s_2``` = 1111110100 \quad f(s_2```) = 7$

$s_3``` = 1110101111 \quad f(s_3```) = 8$

$s_4``` = 0111000101 \quad f(s_4```) = 5$

$s_5``` = 0100011101 \quad f(s_5```) = 5$

$s_6``` = 1110110001 \quad f(s_6```) = 6$

Example (end)

In one generation, the total population fitness changed from 34 to 37, thus improved by ~9%

At this point, we go through the same process all over again, until a stopping criterion is met

References

- **Artificial Intelligence**: A Modern Approach. S. **Russell**, and P. **Norvig**. Series in **Artificial Intelligence** Prentice Hall, Upper Saddle River, NJ, Third edition, (2010). URL: http://aima.cs.berkeley.edu/