Metaheuristic Optimization Methods: Genetic Algorithms

Genetic Algorithms

- We have now studied many Metaheuristics based on the idea of a Local Search
- It is time to look at methods that are based on different mechanisms
- The first such method will be the Genetic Algorithm

“Living organisms are consummate problem solvers. They exhibit a versatility that puts the best computer programs to shame. This observation is especially galling for computer scientists, who may spend months or years of intellectual effort on an algorithm, whereas organisms come by their abilities through the apparently undirected mechanism of evolution and natural selection.”

-John Holland

Genetic Algorithms

Developed by John Holland, University of Michigan (1970’s)

- To design artificial systems software that retains the robustness of natural systems

  Computer programs that "evolve" in ways that resemble natural selection can solve complex problems even their creators do not fully understand

- To understand the adaptive processes of natural systems

  As researchers probe the natural selection of programs under controlled and well-understood conditions, the practical results they achieve may yield some insight into the details of how life and intelligence evolve in the natural world.

Genetic Algorithms

- Genetic algorithms are inspired by Darwin's theory about evolution and the survival of the fittest.
- Basic idea: intelligent, iterative exploration of the search space based on random search using many simultaneous solutions per iteration
- Widely-used today in business, scientific, and engineering circles

Genetic Algorithms

- A random population of solutions are generated
- The best solutions combine to produce possibly (hopefully) better solutions
- Small percent of the solutions randomly modified
- Better and better solutions to a problem are evolved over multiple generations

Genetic Algorithm Terminology

Chromosone: candidate solution (usually feasible)

Fitness score: quality of solution (evaluation function)

Population: set of chromosones

Selection: choosing a pair of chromosones to breed

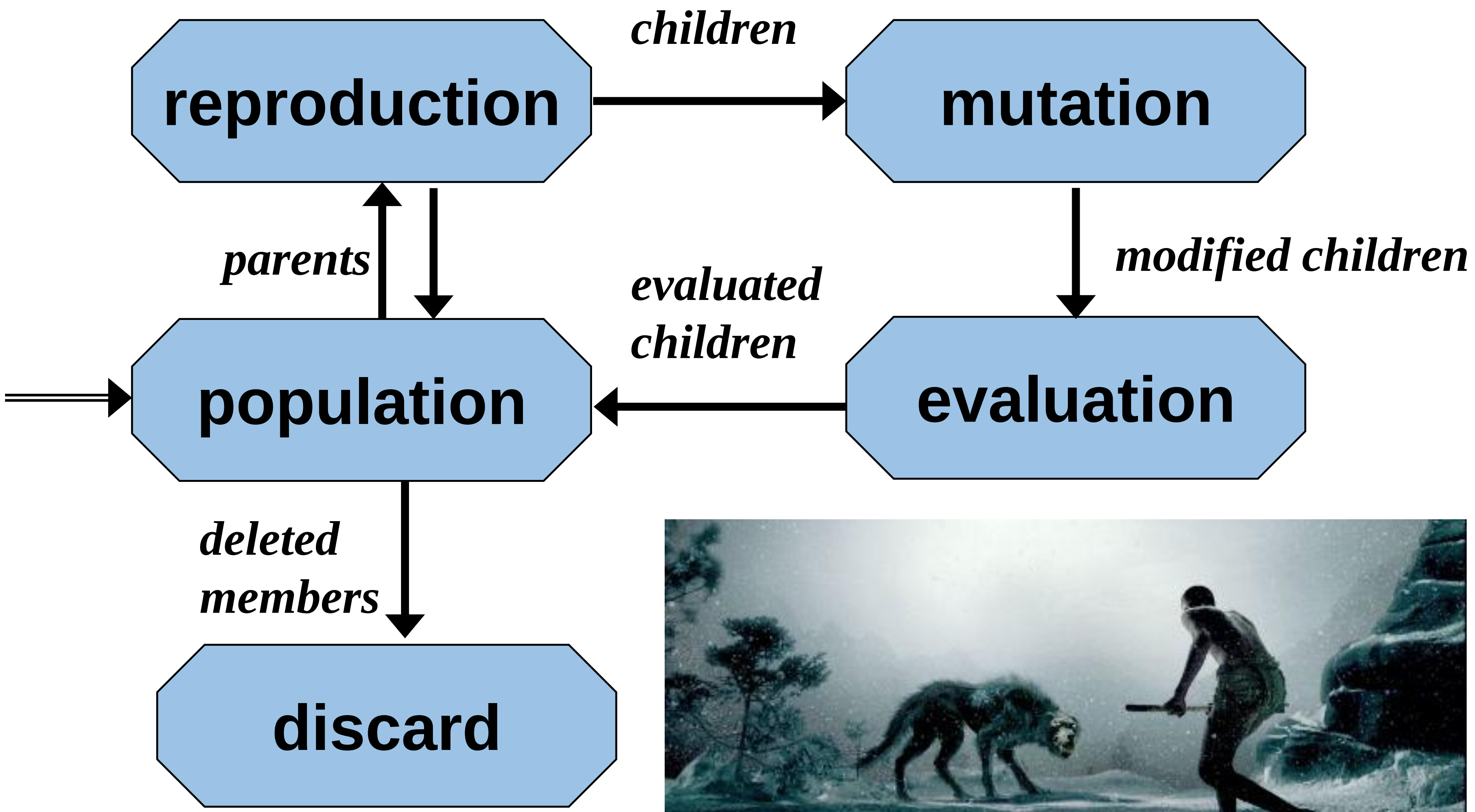
Crossover: “breeding” of chromosones

Offspring: result of crossover

Mutation: random modification of a chromosone

Insertion: selection of offspring to replace previous generation

The GA Cycle of Reproduction



[Start] Generate random population of *n chromosomes*

[Fitness] Evaluate the fitness of each chromosome in population

[New population] Create a new population by repeating following steps until the new population is complete

- **[Selection]** Select two parent chromosomes from a population according to their fitness (the better fitness, the higher the chance to be selected)
- **[Crossover]** With a crossover probability cross over the parents to form a new offspring (children). If no crossover was performed, offspring is an exact copy of parents.
- **[Mutation]** With a mutation probability mutate new offspring
- **[Fitness] and [Insertion]** Place new offspring in a new population

[Replace] Use new population for next run of algorithm

[Test] If the end condition is satisfied, **stop**, and return the best solution in current population

[Loop] Go to step **2 [Fitness]**

Encoding

- Chromosome encodes all relevant information about solution it represents.
- Most common encoding is binary encoding.
  - Each bit (or group of bits) can represent a characteristic of the solution.
  - Or the whole string can represent a number.
- A pair of chromosomes might look like this:

  Chromosome 1: 1101100100110110

  Chromosome 2: 1101111000011110

Population

- A fixed population size throughout all iterations
  - Small populations: undercoverage
  - Large population: computationally demanding
  - Ridiculously bad rule of thumb: use 30
- Populations are usually randomly generated
  - Alternative: seed with good solutions
- Often problems with infeasibility

[Start] Generate random population of *n chromosomes*

[Fitness] Evaluate the fitness of each chromosome in population

[New population] Create a new population by repeating following steps until the new population is complete

- **[Selection]** Select two parent chromosomes from a population according to their fitness (the better fitness, the higher the chance to be selected)
- **[Crossover]** With a crossover probability cross over the parents to form a new offspring (children). If no crossover was performed, offspring is an exact copy of parents.
- **[Mutation]** With a mutation probability mutate new offspring
- **[Fitness] and [Insertion]** Place new offspring in a new population

[Replace] Use new population for next run of algorithm

[Test] If the end condition is satisfied, **stop**, and return the best solution in current population

[Loop] Go to step **2 [Fitness]**

Fitness Evaluation

- The *evaluator* decodes a chromosome and assigns it a fitness measure
- The evaluator is the only link between a classical GA and the problem it is solving
- However, the naïve objective function may not be suitable $\rightarrow$ and may lead to a population of identical individuals

Fitness Scaling

- Limited competition in early generations
  - At the beginning of the GA run, there may be a very high fitness individual i , that biases search towards i

- Increase competition over time
  - Near the end of a run, when the population is converging, there may also not be much separation among individuals in the population

Goals:

- limit the possibility of “super chromosones” dominating the selection process;
- maintain selection pressure throughout the generations

Fitness Scaling

- Traditional linear scaling:

$$f_{\text{linear}} = a + b f_{\text{raw}}$$

- Rank scaling is two step process:
  1. Chromosones are sorted by raw fitness
  2. The new fitness is based on the rank ordering

- Others: top scaling, power scaling, transform ranking, Boltzman scaling, probabilistic nonlinear rank scaling, etc.

[Start] Generate random population of *n chromosomes*

[Fitness] Evaluate the fitness of each chromosome in population

[New population] Create a new population by repeating following steps until the new population is complete

- **[Selection] Select two parent chromosomes from a population according to their fitness (the better fitness, the higher the chance to be selected)**
- **[Crossover]** With a crossover probability cross over the parents to form a new offspring (children). If no crossover was performed, offspring is an exact copy of parents.
- **[Mutation]** With a mutation probability mutate new offspring
- **[Fitness] and [Insertion]** Place new offspring in a new population

[Replace] Use new population for next run of algorithm

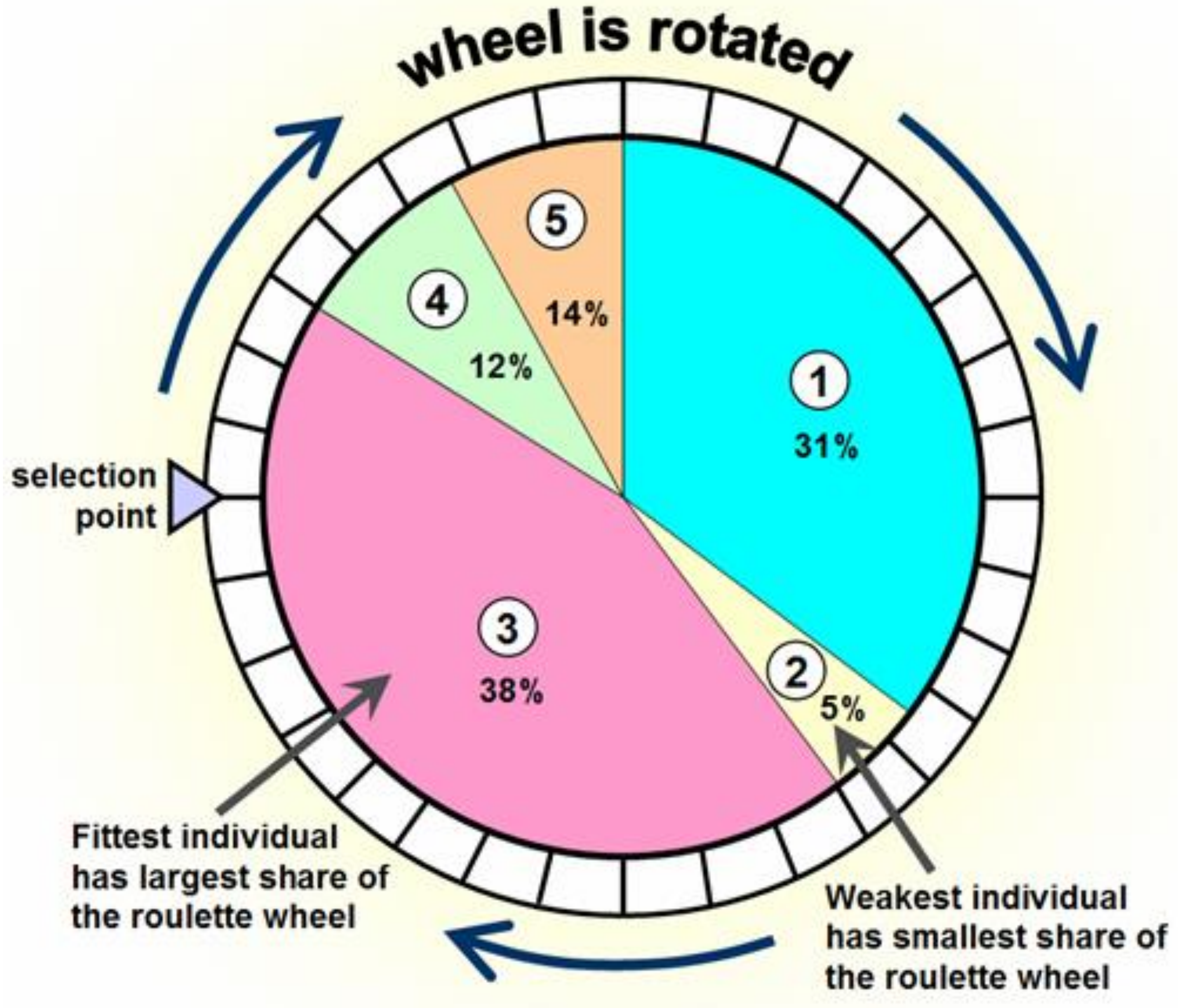
[Test] If the end condition is satisfied, **stop**, and return the best solution in current population

[Loop] Go to step **2 [Fitness]**

Selection

- Each chromosome i is evaluated to determine fitness (or modified fitness) f_i
- **Roulette Wheel Selection**: fittest solutions have highest probability of selection.

Probability of selection = $$\frac{f_i}{\sum_{j=1}^{n} f_j}$$


wheel is rotated
selection point
1
31%
2
5%
3
38%
4
12%
5
14%
Fittest individual has largest share of the roulette wheel
Weakest individual has smallest share of the roulette wheel

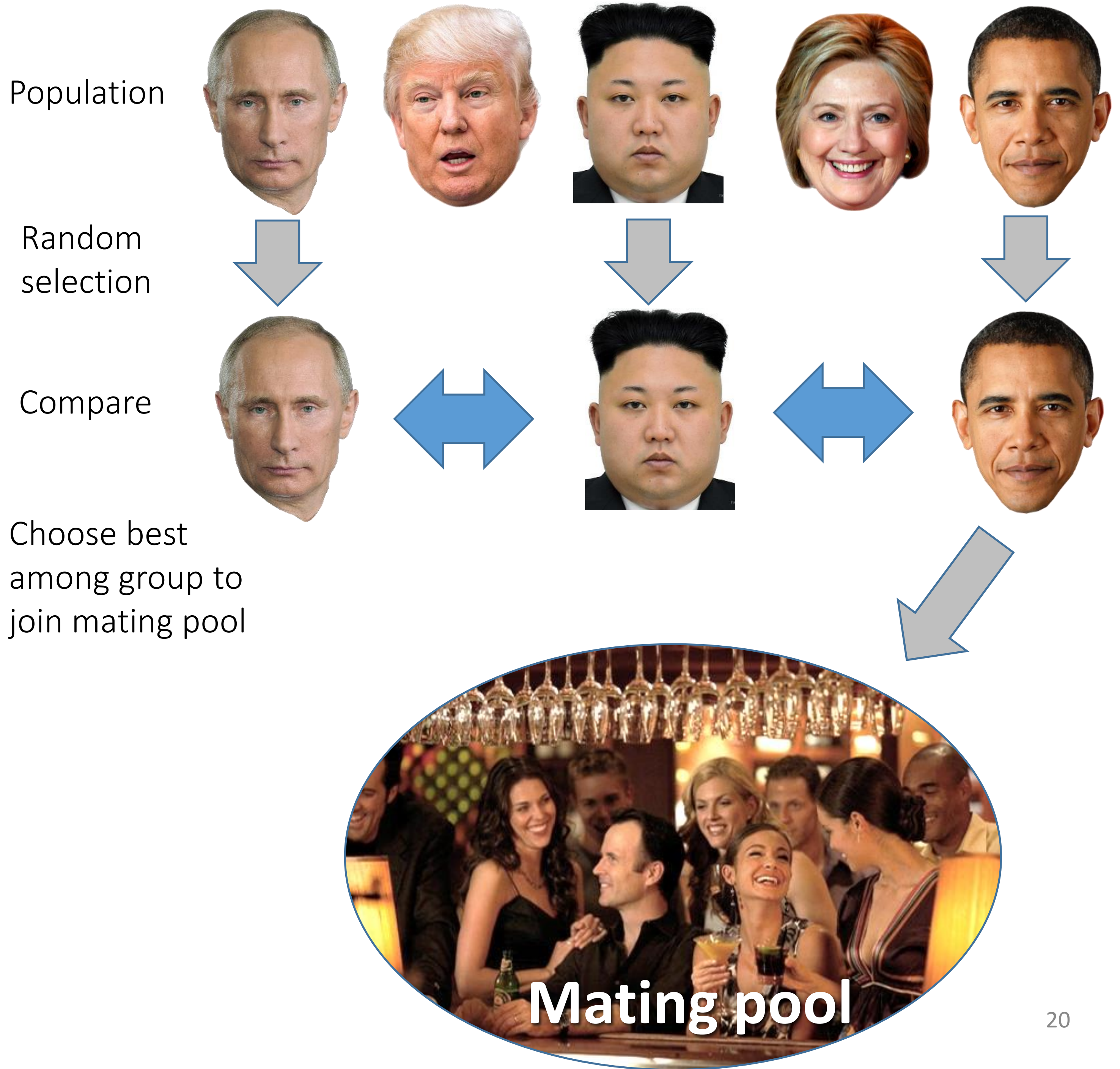
Selection

Tournament selection

- Pick *k* members at random
- Select the best of these to advance to mating pool
- Repeat to select more individuals

Tournament selection

- Pick *k* members at random
- Select the best of these to advance to mating pool
- Repeat to select more individuals



[Start] Generate random population of *n chromosomes*

[Fitness] Evaluate the fitness of each chromosome in population

[New population] Create a new population by repeating following steps until the new population is complete

- [Selection] Select two parent chromosomes from a population according to their fitness (the better fitness, the higher the chance to be selected)
- **[Crossover] With a crossover probability cross over the parents to form a new offspring (children). If no crossover was performed, offspring is an exact copy of parents.**
- **[Mutation]** With a mutation probability mutate new offspring
- **[Fitness] and [Insertion]** Place new offspring in a new population

[Replace] Use new population for next run of algorithm

[Test] If the end condition is satisfied, **stop**, and return the best solution in current population

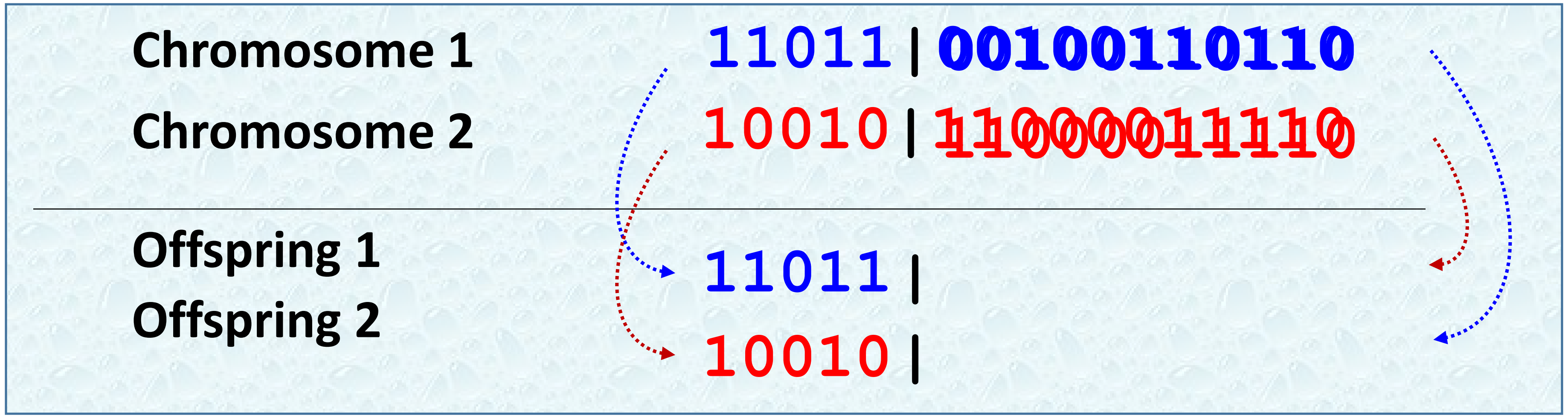
[Loop] Go to step **2 [Fitness]**

Crossover

- Two chromosomes are selected and with probability *p* (crossover rate) the chromosomes are bred.
- The simplest way how to do this is to choose randomly a crossover point.
    - everything before crossover comes from first parent
    - everything after crossover comes from second parent

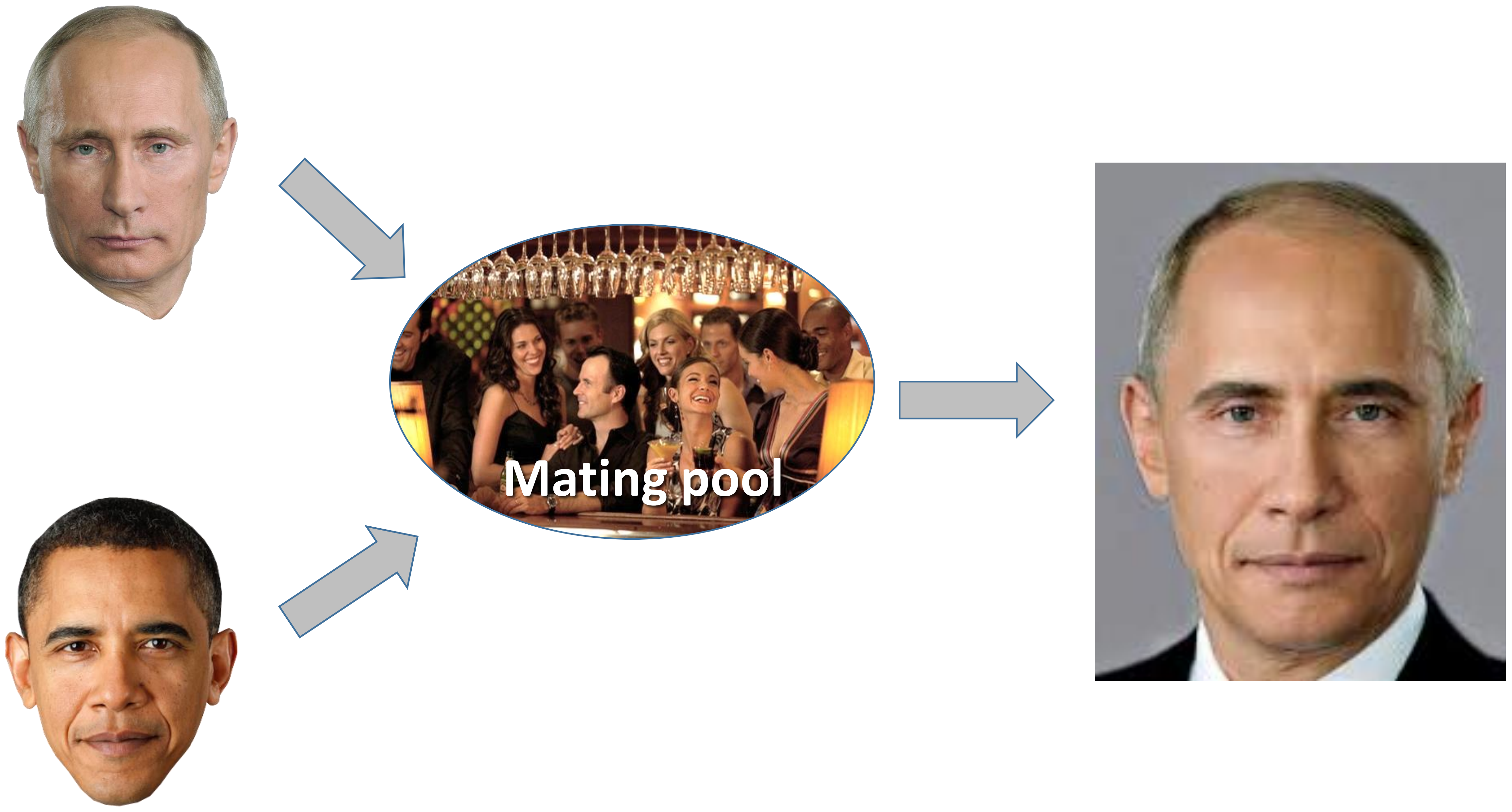
Crossover

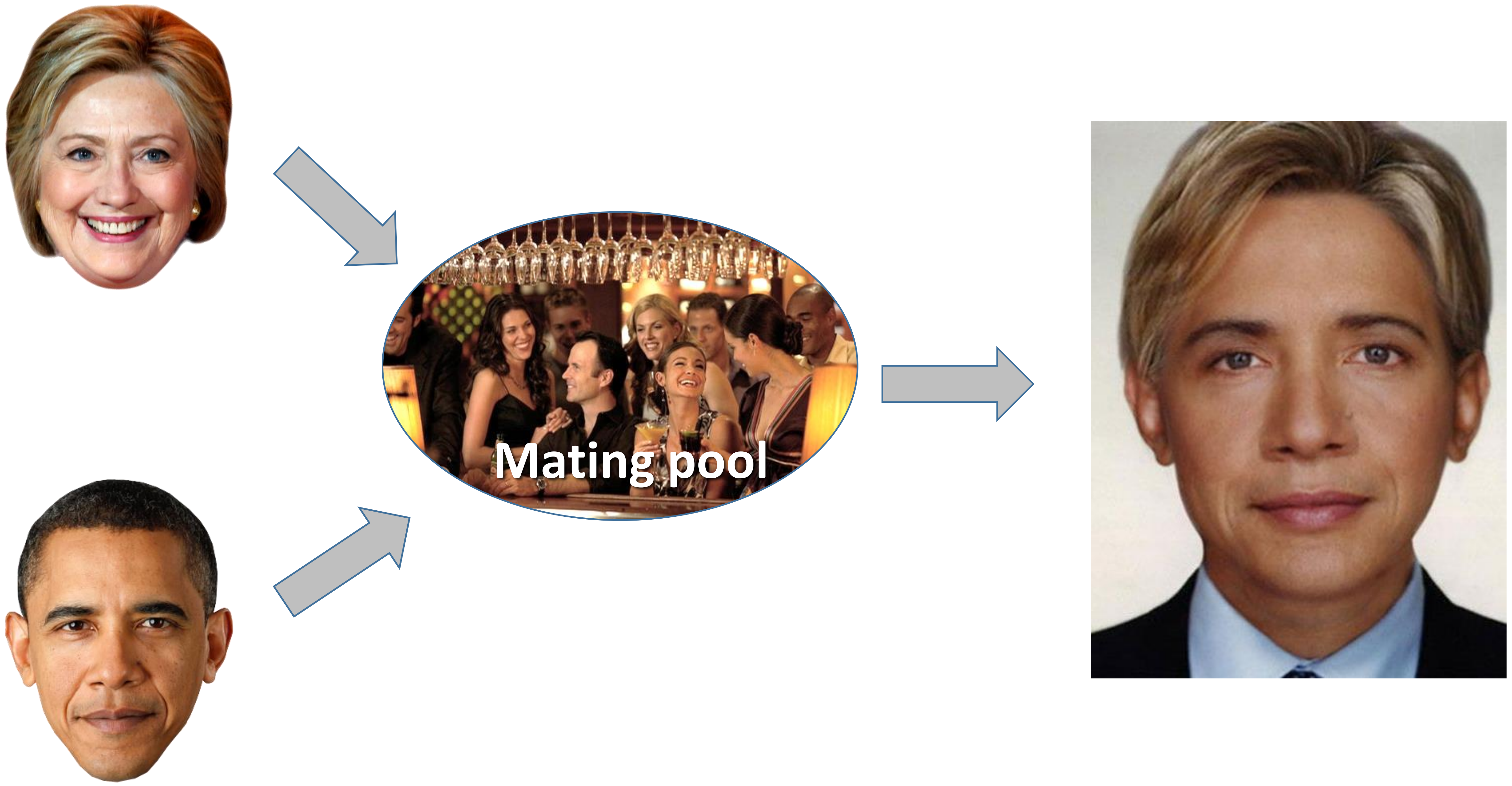
Crossover can then look like this:

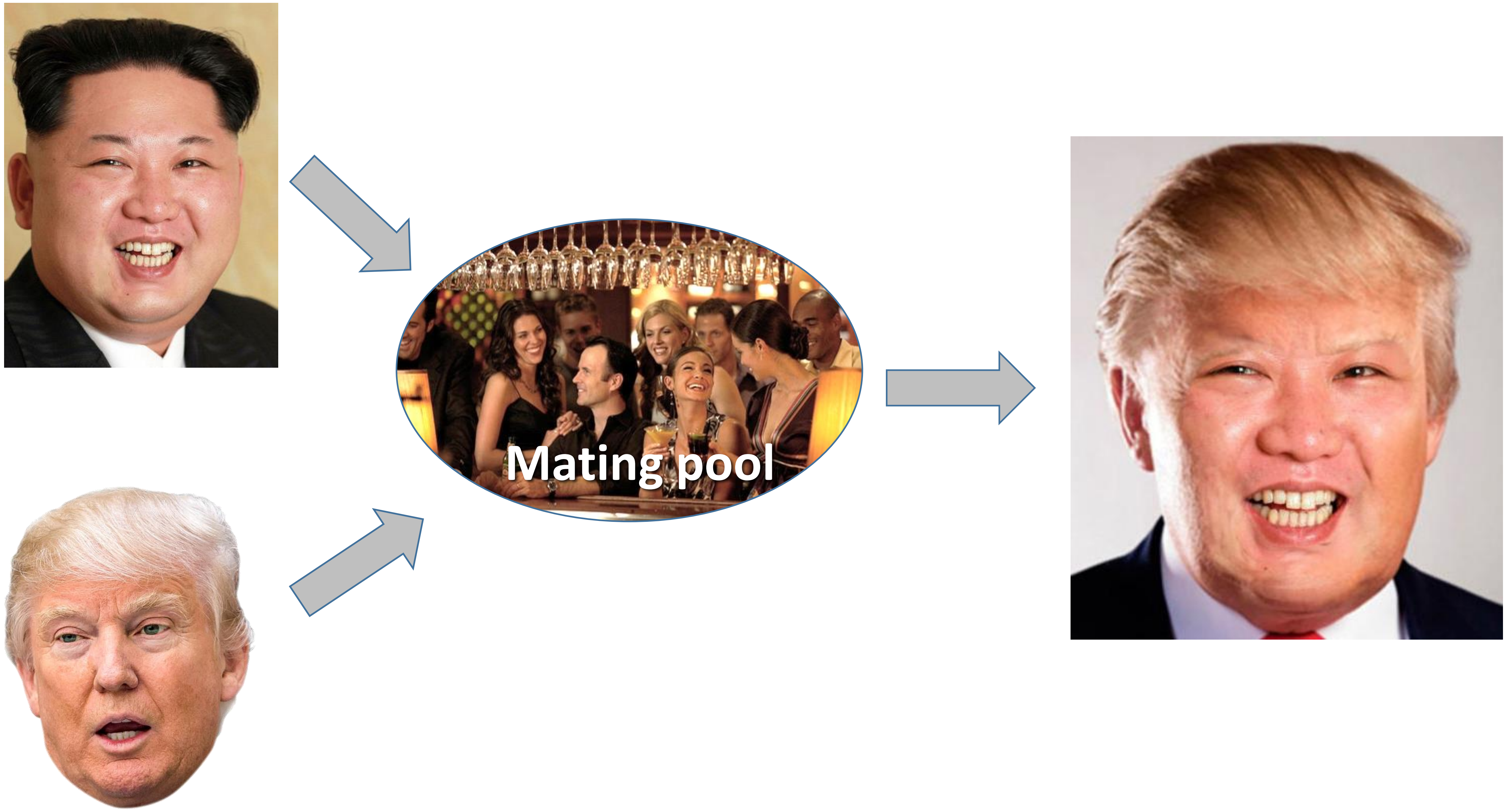


Crossover can be more complex:

- multiple crossover points
- crossover can be specific to encoding
- crossover can be specific to problem type
- crossover should (usually) generate feasible solutions


Mating pool


Mating pool


Mating pool

[Start] Generate random population of *n chromosomes*

[Fitness] Evaluate the fitness of each chromosome in population

[New population] Create a new population by repeating following steps until the new population is complete

- [Selection] Select two parent chromosomes from a population according to their fitness (the better fitness, the higher the chance to be selected)
- [Crossover] With a crossover probability cross over the parents to form a new offspring (children). If no crossover was performed, offspring is an exact copy of parents.
- **[Mutation] With a mutation probability mutate new offspring**
- **[Fitness] and [Insertion]** Place new offspring in a new population

[Replace] Use new population for next run of algorithm

[Test] If the end condition is satisfied, **stop**, and return the best soluti

current population

[Loop] Go to step **2 [Fitness]**

Mutation

- With probability *q* (mutation rate) offspring is mutated.
- This helps increase population diversity.
- Mutation randomly changes the chromosome.

Mutation example:

Original Offspring 1	**1101111000011110**
Mutated offspring 1	**1101111000111110**

- The mutation depends on the encoding and should produce a feasible solution.

[Start] Generate random population of *n chromosomes*

[Fitness] Evaluate the fitness of each chromosome in population

[New population] Create a new population by repeating following steps until the new population is complete

- [Selection] Select two parent chromosomes from a population according to their fitness (the better fitness, the higher the chance to be selected)
- [Crossover] With a crossover probability cross over the parents to form a new offspring (children). If no crossover was performed, offspring is an exact copy of parents.
- [Mutation] With a mutation probability mutate new offspring
- **[Fitness] and [Insertion] Place new offspring in a new population**

[Replace] Use new population for next run of algorithm

[Test] If the end condition is satisfied, **stop**, and return the best sol current population

[Loop] Go to step **2 [Fitness]**

Insertion

Options:

- **Replace the whole population each iteration?**
- **Duplicates**
  - Accept all duplicates
  - Avoid too many duplicates, because that degenerates the population
  - No duplicates at all
- **Elitism**
  - Always keep the best solution
  - Always keep best k solutions

[Start] Generate random population of *n chromosomes*

[Fitness] Evaluate the fitness of each chromosome in population

[New population] Create a new population by repeating following steps until the new population is complete

- [Selection] Select two parent chromosomes from a population according to their fitness (the better fitness, the higher the chance to be selected)
- [Crossover] With a crossover probability cross over the parents to form a new offspring (children). If no crossover was performed, offspring is an exact copy of parents.
- [Mutation] With a mutation probability mutate new offspring
- [Fitness] and [Insertion] Place new offspring in a new population

[Replace] Use new population for next run of algorithm

[Test] If the end condition is satisfied, stop, and return the best solution in current population

[Loop] Go to step 2 [Fitness]

GA Examples

- TSP example: https://www.youtube.com/watch?v=94p5NUogClM
- UAV example https://www.youtube.com/watch?v=x8rSVho-TFs
- Wheeled Vehicle https://www.youtube.com/watch?v=uxourrlPlf8

Benefits of Genetic Algorithms

- Concept is easy to understand
- Modular, separate from application
- Good for “noisy” environments
- Always an answer; answer gets better with time
- Inherently parallel; easily distributed
- Many ways to speed up and improve a GA-based application as knowledge about the problem domain is gained
- Easy to exploit previous or alternate solutions
- Flexible building blocks for hybrid applications
- Substantial history and range of use

When to Use a GA?

- Alternate methods are too slow or overly complicated
- Need an exploratory tool to examine new approaches
- Problem is similar to one that has already been successfully solved by using a GA
- Want to hybridize with an existing method
- Benefits of the GA technology meet key problem requirements

Memetic Algorithms – let's bring it all together!

- Basically, a Memetic Algorithm is a GA with Local Search as improvement mechanism
  - Also known under different names
  - An example of **hybridization**
- The experience is that GAs do not necessarily perform well in some problem domains
- Using Local Search in addition to the population mechanisms proves to be an improvement