An Introduction to Genetic Algorithms

By: Noureddin Sadawi, PhD

Contents

- A brief Introduction to genetic algorithms
- Genetic Operators
- Details of the example problem we are trying to solve
- Java Implementation

  The java code can be dowloaded via the link below this video
- More details about Genetic Operators
- Java Implementation

  The java code can be dowloaded via the link below this video

Thanks to

- Marek Obitko,

  http://www.obitko.com/tutorials/genetic-algorithms/index.php
- Erik D. Goodman “A brief Introduction to genetic algorithms” - on the web at:

  http://www.egr.msu.edu/~goodman/GECSummitIntroToGA_Tutorial-goodman.pdf
- Darrell Whitley, “Genetic Algorithm Tutorial” – on the web at:

  http://www.cs.colostate.edu/~genitor/MiscPubs/tutorial.pdf
- Wikipedia
- Sacha Barber “Simple Genetic Algorithm (GA) to solve a card problem” - on the web at

  http://www.codeproject.com/Articles/16286/AI-Simple-Genetic-Algorithm-GA-to-solve-a-card-pro

Genetic Algorithms

- Are a method of search, often applied to optimization or learning
- Genetic algorithms are a part of evolutionary computing, they use an evolutionary analogy, “survival of the fittest”
- Have extensions including Genetic Programming (GP) (LISP-like function trees), learning classifier systems (evolving rules) and many others

Search Space I

- If we are solving some problem, we are usually looking for some solution, which will be the best among others
- The space of all feasible solutions (it means objects among those the desired solution is) is called **search space** (also state space)
- Each point in the search space represents one feasible solution
- Each feasible solution can be "marked" by its value or fitness for the problem
- We are looking for our solution, which is one point (or more) among feasible solutions - that is one point in the search space
- The process of looking for a solution is equal to looking for some extreme (minimum or maximum) in the search space
- Usually we only know a few points from the search space and we are generating other points as the process of finding solution continues

Marek Obitko, http://www.obitko.com/tutorials/genetic-algorithms/index.php Dec 2014

Search Space II

- The problem is that the search can be very complicated
- One does not know where to look for the solution and where to start
- There are many methods for finding some suitable solution (ie. not necessarily the best solution), for example hill climbing, simulated annealing .. etc
- The solution found by these methods is often considered as a good solution, because it is not often possible to prove what is the real optimum

Marek Obitko, http://www.obitko.com/tutorials/genetic-algorithms/index.php Dec 2014

Chromosome

- All living organisms consist of cells and in each cell there is the same set of **chromosomes**
- Chromosomes are strings of **DNA** and serve as a model for the whole organism
- A chromosome consists of **genes**, blocks of DNA
- Basically it can be said, that each gene encodes a **trait**, for example color of eyes.
- Possible settings for a trait (e.g. blue, brown) are called **alleles**.
- Each gene has its own position in the chromosome. This position is called **locus**
- The complete set of genetic material (all chromosomes) is called **genome**
- **Genotype,** is the "internally coded, inheritable information" carried by all living organisms
- **Phenotype**, is an organism's actual observed properties (its physical and mental characteristics, such as eye color, intelligence etc)

Marek Obitko, http://www.obitko.com/tutorials/genetic-algorithms/index.php Dec 2014
And http://www.brooklyn.cuny.edu/bc/ahp/BioInfo/GP/Definition.html Jan 2015

Methodology I

- In a genetic algorithm, a **population** of candidate solutions (called individuals, creatures, or phenotypes) to an optimization problem is evolved toward better solutions
- Each candidate solution has a set of properties (its **chromosomes** or **genotype**) which can be mutated and altered
- Traditionally, solutions are represented in binary as strings of **0's** and **1's**, but other encodings are also possible e.g. 1011010010

Source: http://en.wikipedia.org/wiki/Genetic_algorithm Dec 2014

Representation Example

- 1011101010 – a possible 10-bit string "CHROMOSOME" representing a possible solution to a problem
- Bits or subsets of bits might represent choice of some feature, for example, *let's represent choice of a mobile phone:*

bit position	meaning
1-2	Nokia, Samsung, LG or ZTE
3-5	Colour (assume 8 colours)
6	Wifi (Yes or No)
7	Bluetooth (Yes or No)
8	Touch Screen (Yes or No)
9	Water Proof (Yes or No)
10	Front Camera (Yes or No)

1011101010

Methodology II

- The evolution usually starts from a population of **randomly** generated individuals, and is an iterative process, *with the population in each iteration called a* ***generation***
- In each generation, the **fitness** of every individual in the population is evaluated; *the fitness is usually the value of the objective function in the optimization problem being solved*
- The more fit individuals are stochastically selected from the current population, and each individual's *genome* is modified (recombined and possibly randomly mutated) to form a new generation
- The new generation of candidate solutions is then used in the next iteration of the algorithm
- **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**

Source: http://en.wikipedia.org/wiki/Genetic_algorithm Dec 2014

Methodology III

- A typical genetic algorithm requires:
  - a **genetic representation** of the solution domain (i.e. the **genotype** will be different for a different problem domain)
  - a **fitness function** to evaluate the solution domain (i.e. the **fitness function** can be different for different problem domains)

These two items must be developed again, whenever a new problem is specified

Outline of the Basic Genetic Algorithm

1- [Start] Generate random population of **n** chromosomes (suitable solutions for the problem)

2- [Fitness] Evaluate the fitness f(x) of each chromosome **x** in the population

3- [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 bigger 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 at each locus (position in chromosome).
- [Accepting] Place new offspring in a new population

4- [Replace] Use new generated population for a further run of algorithm

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

6- [Loop] Go to step 2

Questions

- How do we create chromosomes?, what type of representation/encoding do we choose?
- Crossover or Mutation? Which one is better/necessary?
- How do we select parents for crossover?

  This can be done in many ways, but the main idea is to select the better parents (in hope that the better parents will produce better offspring)
- If we make new population only by new offspring then this can cause loss of the best chromosomes from the last population

  This is true, so so called **elitism** is often used. This means, that at least one best solution is copied without changes to a new population, so the best solution found can survive to end of run

Genetic Operators

- Selection
- Mutation
- Crossover

Selection

- Chromosomes (parents) are selected from the population to be parents to **crossover**
- The problem is how to select these chromosomes

  The best ones should survive and create new offspring
- Traditionally, parents are chosen to mate with probability proportional to their fitness:

  proportional selection
- Traditionally, children replace their parents
- There are many methods for selecting the best chromosomes, for example roulette wheel selection, Boltzmann selection, tournament selection, rank selection and steady state selection
- Overall principle: survival of the fittest

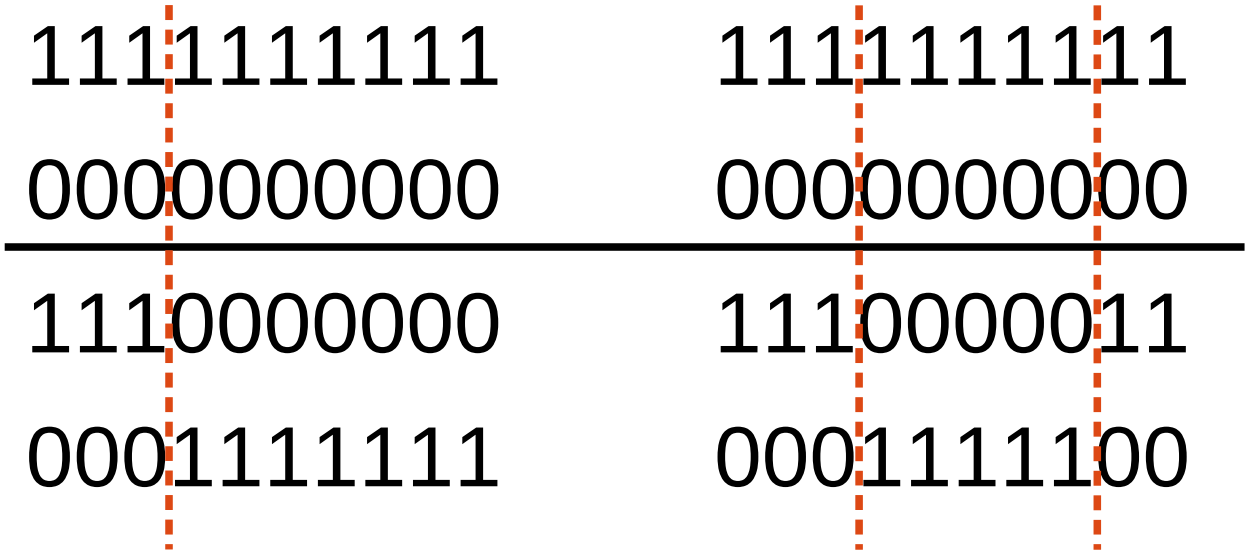
Mutation

- Operates on **ONE** “parent” chromosome
- Produces an “offspring” with changes
- Classically, toggles one [or more] bit in a binary representation
- So, for example: 1101000110

  could mutate to: 1111000110
- Each bit has same probability of mutating

Crossover

- Operates on **TWO** parent chromosomes
- Produces one or two children or offspring
- Classical crossover occurs at 1 or 2 points:

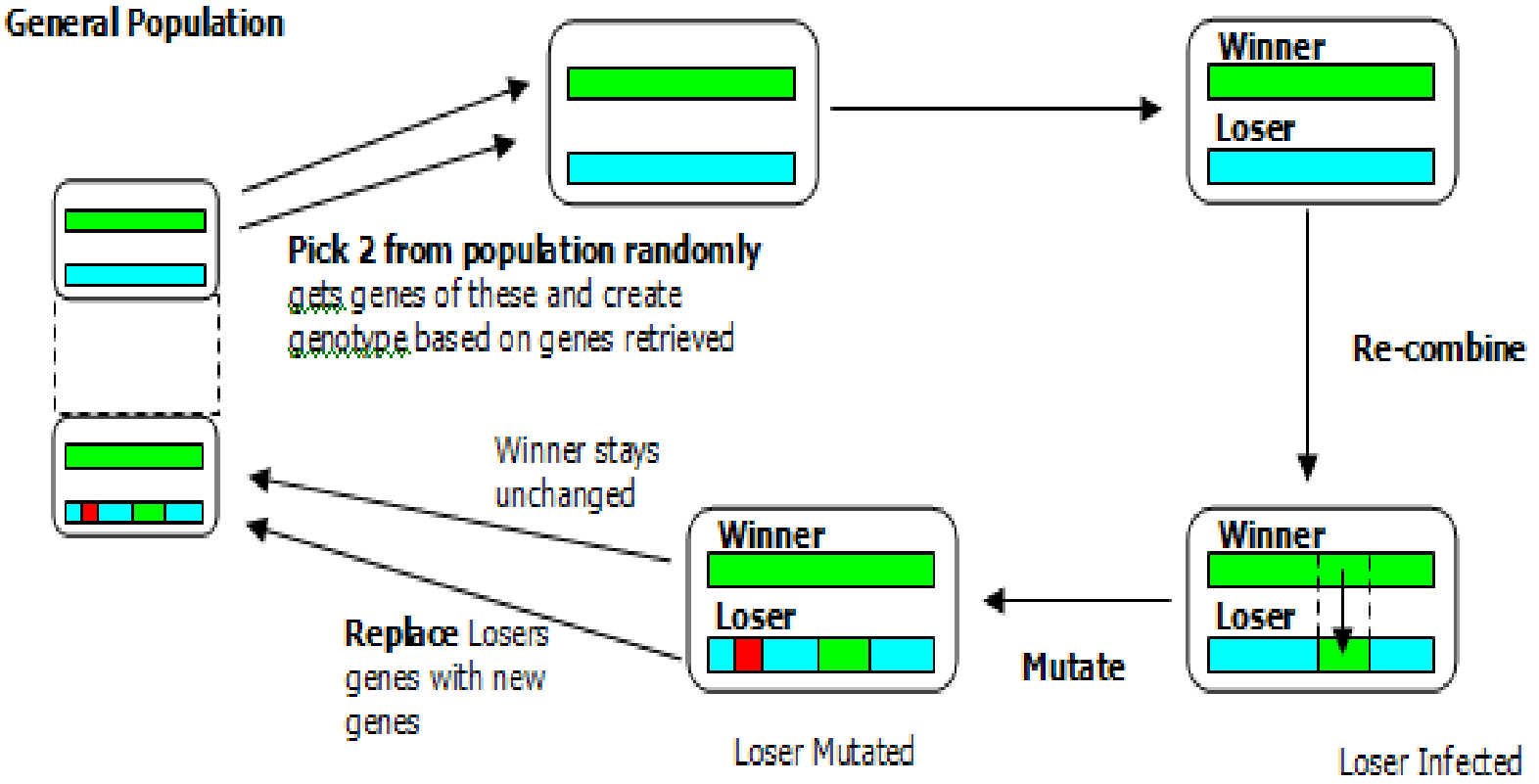


Microbial GA

- The basic operation of the Microbial GA training is as follows:
  - Pick two choromosomes/genotypes (solutions) at random
  - Compare Scores (Fitness) to come up with a **Winner** and **Loser**
  - Go along the Loser choromosome (genotype), at each locus (Point) and:
    - With some probability (randomness), perform cross over on the Loser .. i.e. copy from Winner to Loser (overwrite)
    - With some probability (randomness), perform mutation on the Loser

- So only the Loser gets changed, which gives a version of **Elitism** for free, this ensures that the best in the breed remains in the population

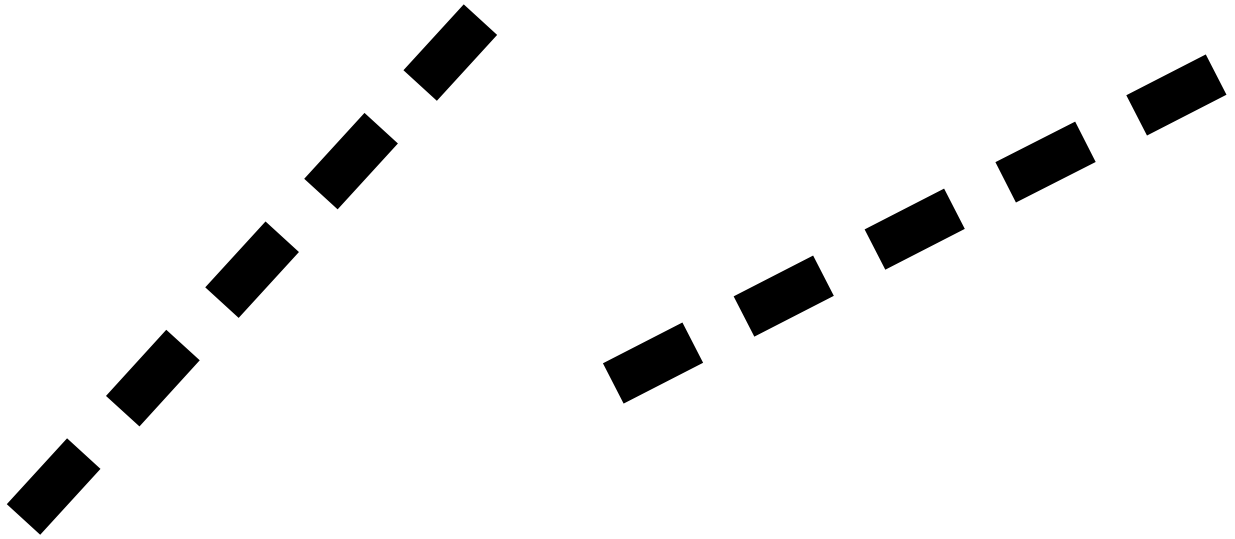
Source: AI - Simple Genetic Algorithm (GA) to solve a card problem By Sacha Barber, 8 Nov 2006



Source: AI - Simple Genetic Algorithm (GA) to solve a card problem By Sacha Barber, 8 Nov 2006

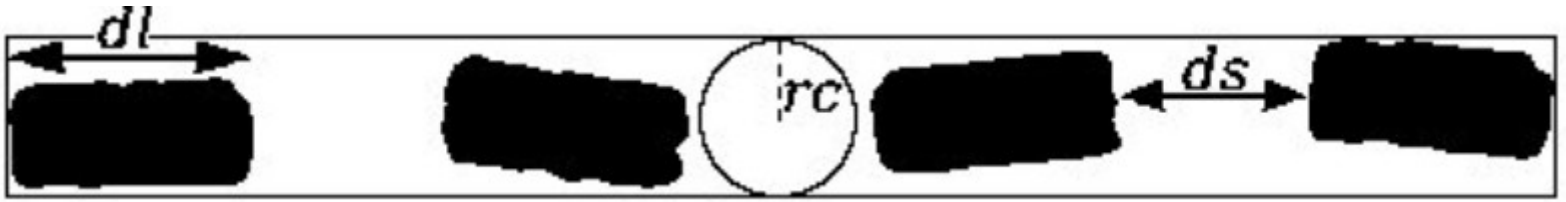
The Problem Statement

- The problem domain states that we have the coordinates of 10 short line segments (dashes)
- Five of these dashes form one dahsed line and the other five form another dashed line



Dashed Line Segment

- A set of line segments
- Elements are approximately collinear (use of rc)
- Every element of it has length of approximately dl
- No two elements of have a separation distance of less than the minimum of ds



Please Cite this Paper

- @InProceedings{SaSeSo-DRR12-ChemicalStructureRecognition-ARuleBasedApproach,
  author = {Sadawi, Noureddin M. and Sexton, Alan P. and Sorge, Volker},
  title = {Chemical Structure Recognition: A Rule Based Approach},
  booktitle = {19th Document Recognition and Retrieval Conference (DRR 2012)},
  year = 2012,
  editor = {Christian Viard-Gaudin and Richard Zanibbi},
  month = {January},
  doi = {10.1117/12.912185},
  publisher = {SPIE},
}

Representation of the Solution

- A solution will be a binary array of size 10:

  e.g. [1,0,0,1,1,0,1,0,1,0]

- The 1's represent five dashes & the 0's represent the other five dashes
- WE CONDITION that DAHSES are in ORDER
- Remember: Five of the short line segments form one dahsed line and the other five form another dashed line

How to do it!

- We will create an array of size 10 to hold the coordinates of the 10 dashes dashes = [((x1,y1),(x2,y2)), ((x1,y1),(x2,y2)),...,((x1,y1),(x2,y2))]
- We will create a chromosome matrix (30 by 10) .. The 30 represents a population size of 30 (It could be any size, but should be big enough to allow some dominant solutions to form)
- Each member of this matrix represents a possible solution (binary array of size 10)

  e.g. [1,0,0,1,1,0,1,0,1,0] ---> [d1,d4,d5,d7,d9] … [d2,d3,d6,d8,d10]
- We will use indices of 1's & 0's to access coordinates in the dashes array
- So in java, this will be:

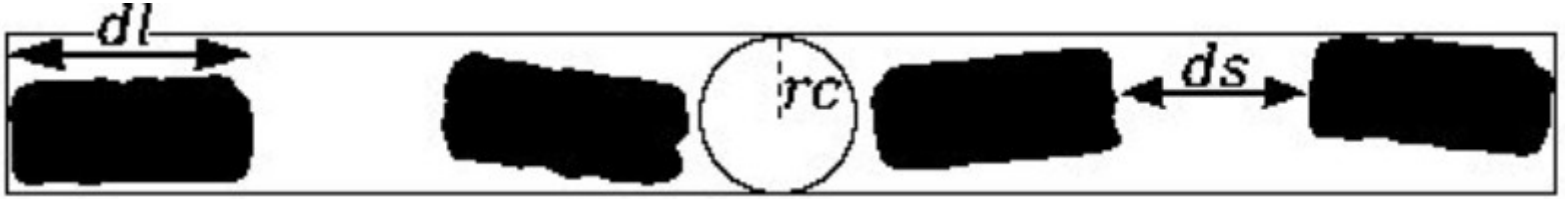
```
//the genes matrix, 30 members, 10 line segments each
int[][] lgenes = new int[30][10];
```

The Fitness Function

- For each candidate solution, we will check the two arrays for colinearity:
  - Check the first element against the second, the second against the third, the third against the fourth and the fourth against the fifth

    (FOUR pairwise comparisons)

    e.g. [1,0,0,1,1,0,1,0,1,0] ---> [d1,d4,d5,d7,d9] … [d2,d3,d6,d8,d10]
  - We will return the number of **adjacent & colinear** pairs
  - If this number is **4** then these elements form a dashed line segment
  - **If the two arrays return 4 then we have found the solution**
  - **If the two arrays DO NOT return 4 then we use the SUM of results to compare against other candidate solutions so we can choose Winner and Loser**



Representation/Encoding

- Encoding of chromosomes (solutions) is of very high importance when you are trying to solve a problem with GA
- Encoding very much depends on the problem
- We have seen **binary encoding**

Example Binary Encoding

- In binary encoding every chromosome is a string of bits, 0 or 1

  Chromosome A: 1011001011001010111100101

  Chromosome B: 1111111000001100000011111

- **Example of Problem:** Knapsack problem
- **The problem:** There are things with given value and size
  - The knapsack has given capacity
  - Select things to maximize the value of things in knapsack, but do not exceed knapsack capacity
- **Encoding:** Each bit says, if the corresponding thing is in knapsack

Permutation Encoding

- Permutation encoding can be used in ordering problems, such as travelling salesman problem or task ordering problem
- In permutation encoding, every chromosome is a string of numbers, which represent numbers in a sequence

  Chromosome A: 1 5 3 2 6 4 7 9 8

  Chromosome B: 8 5 6 7 2 3 1 4 9

- Permutation encoding is only useful for ordering problems

Example Permutation Encoding

- **Example of Problem:** Travelling salesman problem (TSP)
- **The problem:**
  - There are cities and given distances between them
  - Travelling salesman has to visit all of them, but he does not to travel very much
  - Find a sequence of cities to minimize travelled distance
- **Encoding:** Chromosome says order of cities, in which salesman will visit them

Value Encoding

- Direct value encoding can be used in problems, where some complicated value, such as real numbers, are used (the use of binary encoding for this type of problems would be very difficult)
- In value encoding, every chromosome is a string of some values
- Values can be anything connected to problem, form numbers, real numbers or chars to some complicated objects

  Chromosome A: 1.2324 5.3243 0.4556 2.3293 2.4545

  Chromosome B: ABDJEIFJDHDIERJFDLDFLFEGT

  Chromosome C: (back), (back), (right), (forward), (left)

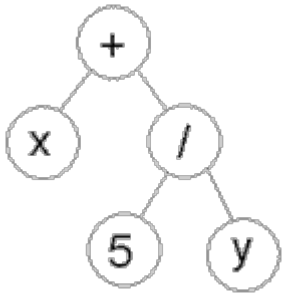
- On the other hand, for this encoding it is often necessary to develop some new crossover and mutation techniques specific for the problem

Example Value Encoding

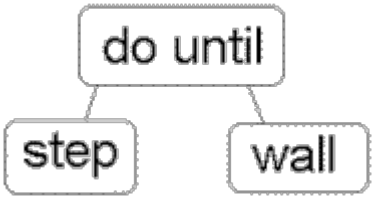
- **Example of Problem:** Finding weights for neural network
- **The problem:**
  - There is some neural network with given architecture
  - Find weights for inputs of neurons to train the network for wanted output
- **Encoding:** Real values in chromosomes represent corresponding weights for inputs

Tree Encoding

- Tree encoding is used mainly for evolving programs or expressions, for genetic programming
- In tree encoding every chromosome is a tree of some objects, such as functions or commands in programming language



(+ x (/ 5 y))



(do_until step wall)

- Tree encoding is good for evolving programs
- Programing language LISP is often used to this, because programs in it are represented in this form and can be easily parsed as a tree, so the crossover and mutation can be done relatively easily

Example Tree Encoding

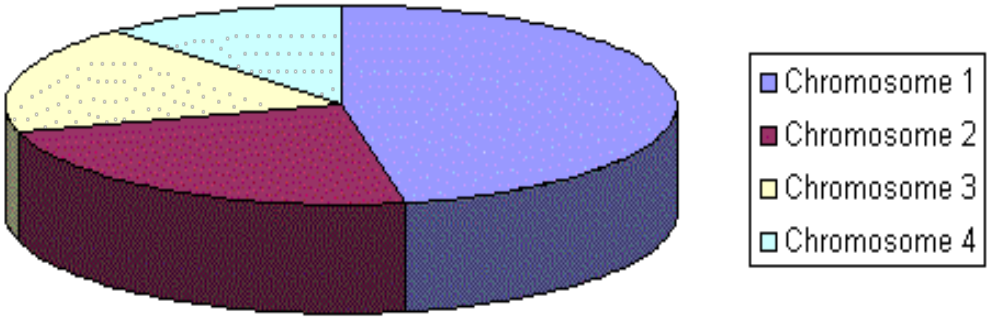
- **Example of Problem:** Finding a function from given values
- **The problem:**
  - Some input and output values are given
  - Task is to find a function, which will give the best (closest to wanted) output to all inputs
- **Encoding:** Chromosomes are functions represented in a tree

Selection

- As you already know from the GA outline, chromosomes are selected from the population to be parents to crossover
- The problem is how to select these chromosomes
- We want the best ones to survive and create new offspring
- There are many methods how to select the best chromosomes, for example *roulette wheel selection, rank selection and steady state selection*

Roulette Wheel Selection

- Parents are selected according to their fitness
- The better the chromosomes are, the more chances to be selected they have
- Imagine a roulette wheel where all chromosomes in the population are placed and the area each chromosome has corresponds to its fitness function, like on the following picture
- Then a marble is thrown there and it selects the chromosome
- Chromosomes with bigger fitness will be selected more times



Roulette Wheel Selection

- This can be simulated by following algorithm:
  - **[Sum]** Calculate sum **S** of all chromosome fitnesses in population
  - **[Select]** Generate random number **r** from interval (0,S)
  - **[Loop]** Go through the population and sum fitnesses from 0 - sum s
    - When the sum s is greater than r, stop and return the chromosome where you are

We will have java implementation of this method

Roulette Wheel Selection

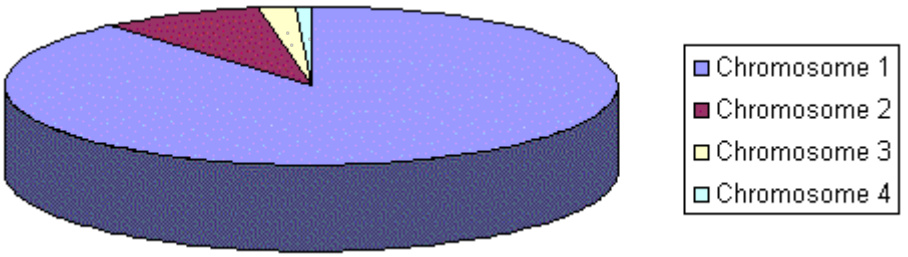
```
public static int selectMemberUsingRouletteWheel() {
    int totalSum = 0;
    for (int x = population.length-1 ; x >= 0 ; x--) {
        totalSum += listIsDashedLine(asList(population[x]));
    }
    int rand = randomNumber(0,totalSum);
    int partialSum = 0;
    for (int x=population.length-1 ; x >= 0 ; x--) {
        partialSum += listIsDashedLine(asList(population[x]));
        if (partialSum >= rand) {
            return x;
        }
    }
    return -1;

}
```

Rank Selection

- The previous selection will have problems when the fitnesses differs very much
- For example, if the best chromosome fitness is 90% of all the roulette wheel then the other chromosomes will have very few chances to be selected
- Rank selection first ranks the population and then every chromosome receives fitness from this ranking
- The worst will have fitness 1, second worst 2 etc. and the best will have fitness N (number of chromosomes in population)

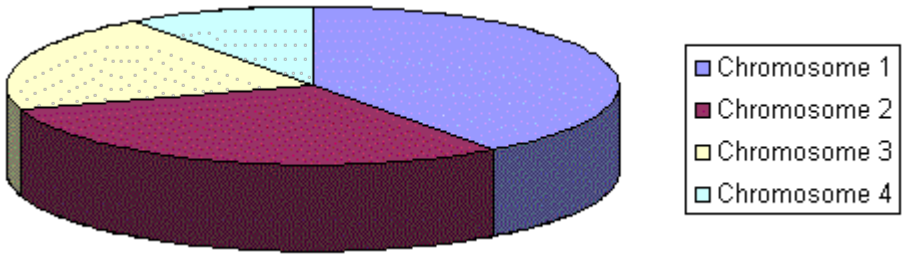
- You can see in following picture, how the situation changes after changing fitness to order number



Situation before ranking (graph of fitnesses)

Rank Selection

- After this all the chromosomes have a chance to be selected
- But this method can lead to slower convergence, because the best chromosomes do not differ so much from other ones



Situation after ranking (graph of order numbers)

Steady-State Selection

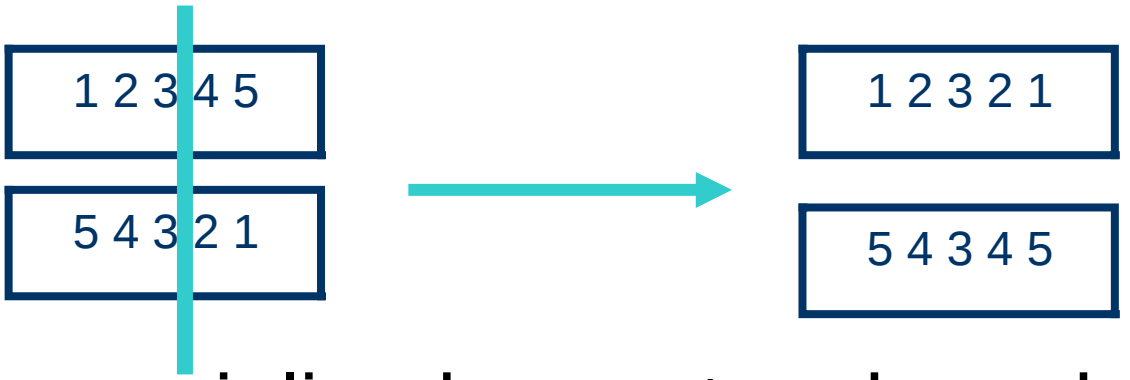
- This is not a particular method of selecting parents
- Main idea of this selection is that big part of chromosomes should survive to next generation
- GA then works in the following way:
  - In every generation a few chromosomes are selected (good, with high fitness) for creating a new offspring
  - Then some chromosomes (bad, with low fitness) are removed and the new offspring is placed in their place
  - The rest of population survives to new generation

Elitism

- Idea of elitism has been already introduced
- When creating new population by crossover and mutation, we have a big chance, that we will loose the best chromosome
- Elitism is name of method, which first copies the best chromosome (or a few best chromosomes) to new population
- The rest is done in classical way
- Elitism can very rapidly increase performance of GA, because it prevents losing the best found solution

Crossover Operators for Permutations

- Normal crossover operators will often lead to inadmissible solutions



- Many specialised operators have been devised which focus on combining order or adjacency information from the two parents

Source: Introduction to Evolutionary Computing by A.E. Eiben and J.E. Smith

Order One Crossover

The Idea is to preserve the relative order in which elements occur

Informal procedure:

1) Choose an arbitrary part from the first parent

2) Copy this part to the child

3) Copy the numbers that are not in the child, from the second parent to the child:

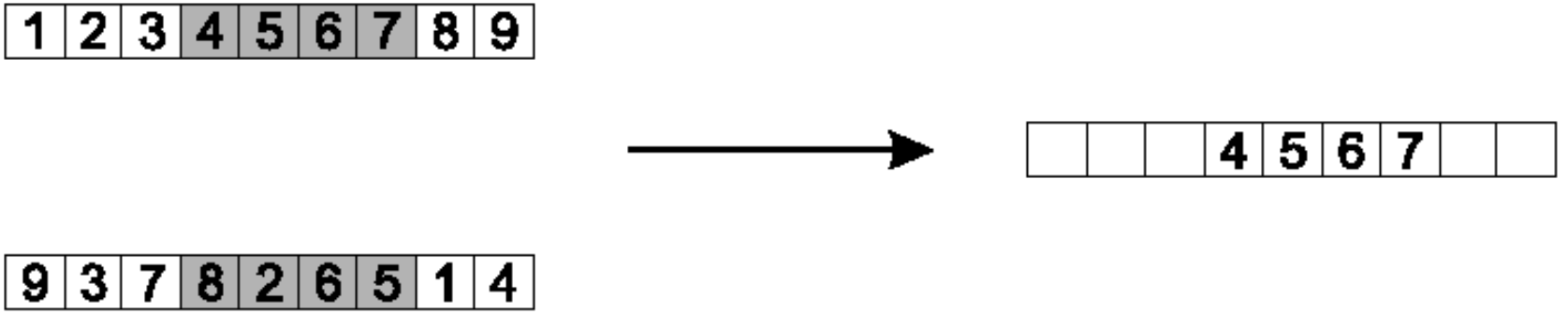
- starting right from the cut point of the copied part,
- using the **order** of the second parent
- and wrapping around at the end

4) Analogous for the second child, with parent roles reversed

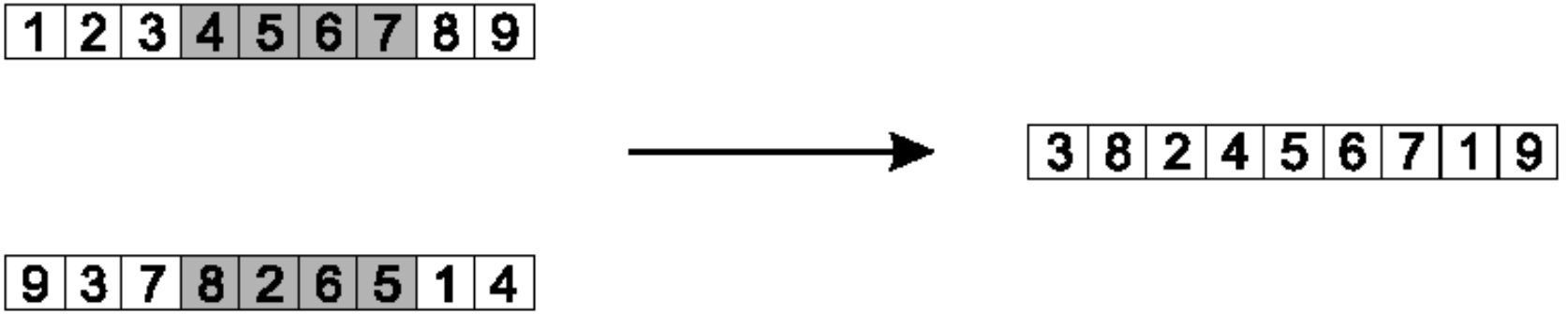
Source: Introduction to Evolutionary Computing by A.E. Eiben and J.E. Smith

Order 1 Crossover Example

- Copy randomly selected set from first parent



- Copy rest from second parent in order 1,9,3,8,2



Source: Introduction to Evolutionary Computing by A.E. Eiben and J.E. Smith

Partially Mapped Crossover (PMX)

- This crossover method builds an offspring by choosing a subsequence of elements from one parent and preserving the order and position of as many elements as possible from the other parent
- A subsequence of elements is selected by choosing two random cut points, which serve as boundaries for the swapping operations

Source: http://paginas.fe.up.pt/~mac/ensino/docs/OR/HowToSolveIt/Ch%206%20-%20An%20evolutionary%20approach.pdf

Partially Mapped Crossover (PMX)

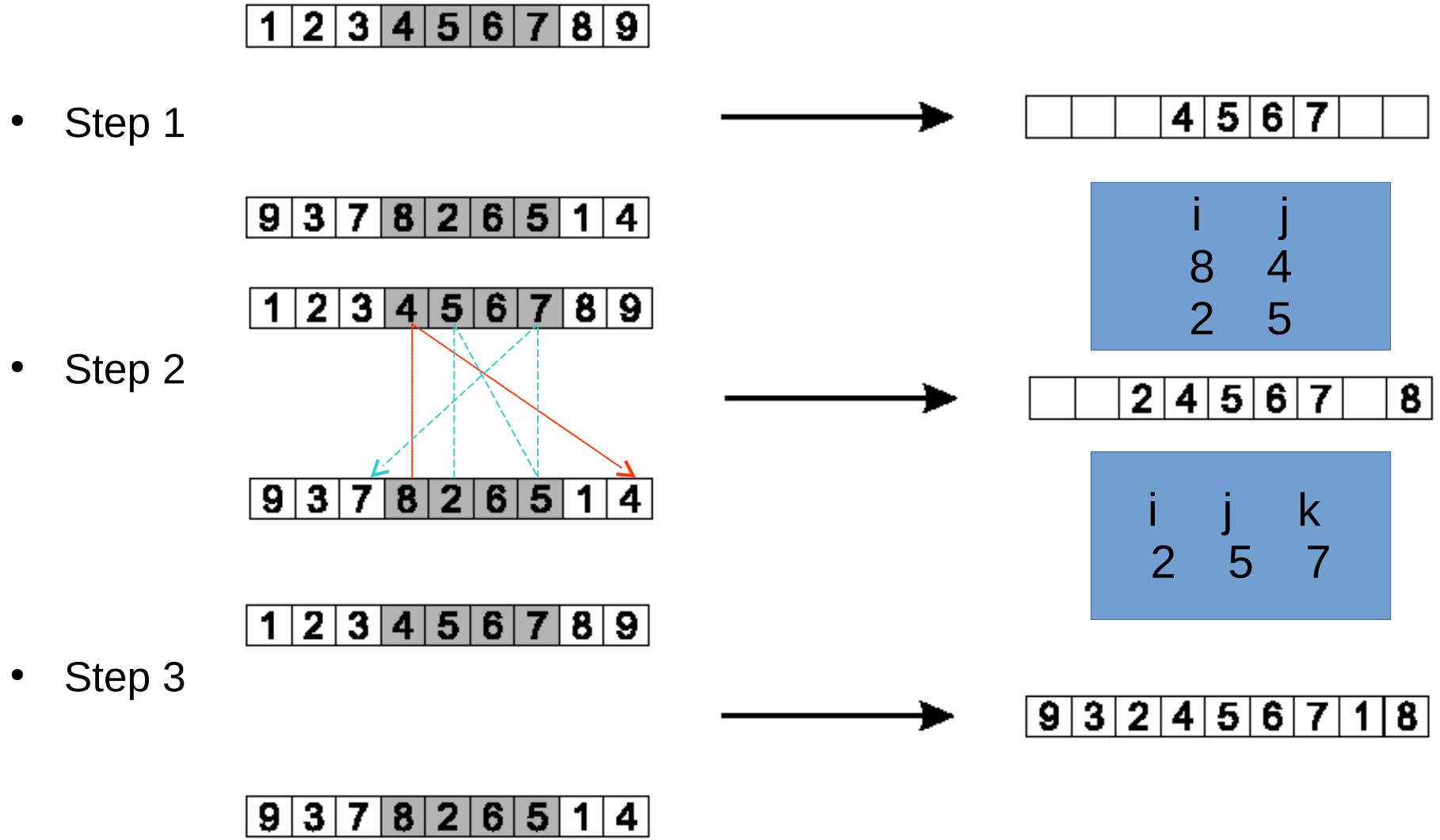
Informal procedure for parents P1 and P2:

1. Choose random segment and copy it from P1 to child
2. Starting from the first crossover point look for elements in that segment of P2 that have not been copied
3. For each of these ***i*** look in the offspring to see what element ***j*** has been copied in its place from P1
4. Place ***i*** into the position occupied by ***j*** in P2, since we know that we will not be putting ***j*** there (as j is already in offspring)
5. If the place occupied by ***j*** in P2 has already been filled in the offspring by ***k***, put ***i*** in the position occupied by ***k*** in P2
6. Having dealt with the elements from the crossover segment, the rest of the offspring can be filled from P2

Second child is created analogously

Source: Introduction to Evolutionary Computing by A.E. Eiben and J.E. Smith

PMX example



Source: Introduction to Evolutionary Computing by A.E. Eiben and J.E. Smith

Cycle Crossover

Basic idea:

Each element comes from one parent *together with its position*

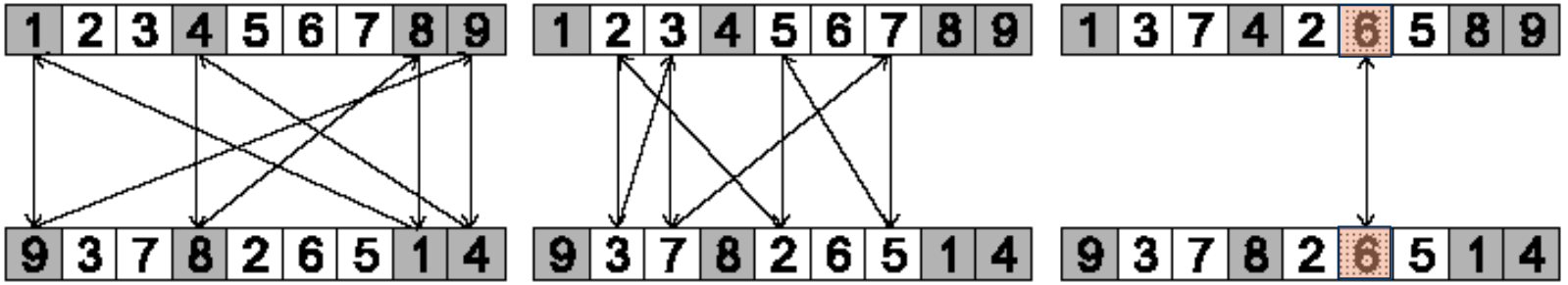
Informal procedure:

1. Make a cycle of elements from P1 in the following way:
   (a) Start with the first element of P1
   (b) Look at the element at the *same position* in P2
   (c) Go to the position with the *same element* in P1
   (d) Add this element to the cycle
   (e) Repeat step b through d until you arrive at the first element of P1
2. Put the elements of the cycle in the first child on the positions they have in the first parent
3. Take next cycle from second parent

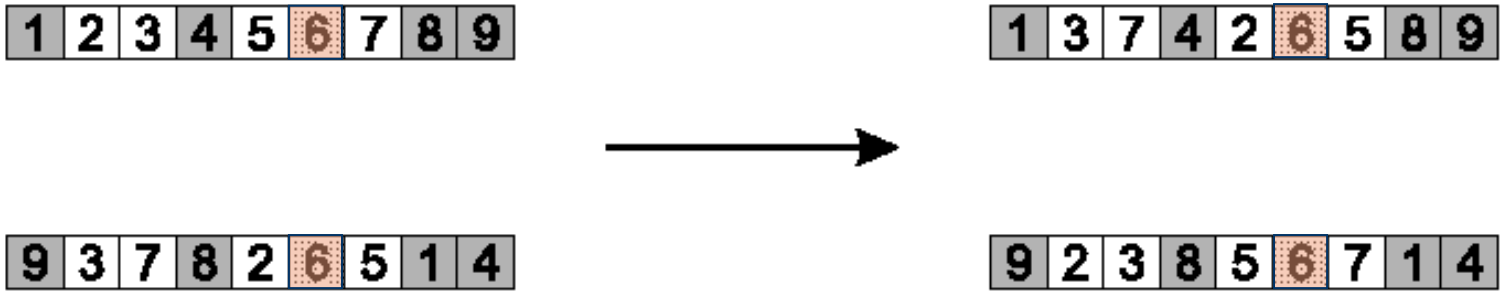
Source: Introduction to Evolutionary Computing by A.E. Eiben and J.E. Smith

Cycle crossover example

- Step 1: identify cycles



- Step 2: copy alternate cycles into offspring



Source: Introduction to Evolutionary Computing by A.E. Eiben and J.E. Smith

Insert Mutation for Permutations

- Pick two elements at random
- Move the second to follow the first, shifting the rest along to accommodate
- Note that this preserves most of the order and the adjacency information

Source: Introduction to Evolutionary Computing by A.E. Eiben and J.E. Smith

Insert Mutation for Permutations Java Implementation

```
public static int[] insertMutation(int[] parent){
    int[] array = parent.clone();
    int l = array.length;
    //get 2 random integers between 0 and size of array
    int r1 = randomNumber(0,l);
    int r2 = randomNumber(0,l);
    //to make sure the r1 < r2
    while(r1 >= r2) {r1 = randomNumber(0,l); r2 = randomNumber(0,l);}
    //this code moves element at r2 to just after r1
    //and shifts the elements after r1 by 1 place
    for(int i = r2-1; i > r1 ; i--){
        int temp2 = array[i+1];
        array[i+1] = array[i];
        array[i] = temp2;
    }

    return array;
}
```

Swap Mutation for Permutations

- Pick two elements at random and swap their positions
- Preserves most of adjacency information

Source: Introduction to Evolutionary Computing by A.E. Eiben and J.E. Smith

Swap Mutation for Permutations Java Implementation

```java
public static int[] swapMutation(int[] parent){
    int[] array = parent.clone();
    int l = array.length;
    //get 2 random integers between 0 and size of array
    int r1 = randomNumber(0,l);
    int r2 = randomNumber(0,l);
    //to make sure the 2 numbers are different
    while(r1 == r2) r2 = randomNumber(0,l);

    //swap array elements at those indices
    int temp = array[r1];
    array[r1] = array[r2];
    array[r2] = temp;

    return array;
}
```

Inversion Mutation for Permutations

- Pick two elements at random and then invert the substring between them
- Preserves most adjacency information
- Disruptive of order information

Source: Introduction to Evolutionary Computing by A.E. Eiben and J.E. Smith

Inversion Mutation for Permutations Java Implementation

```java
public static int[] inversionMutation(int[] parent){
    int[] array = parent.clone();
    int l = array.length;
    for(int k = 0; k < 5; k++){//repeat process 5 times
        //get 2 random integers between 0 and size of array
        int r1 = randomNumber(0,l);
        int r2 = randomNumber(0,l);
        //to make sure the r1 < r2
        while(r1 >= r2) {r1 = randomNumber(0,l); r2 = randomNumber(0,l);}
        //this code inverts (i.e. reverses) elements between r1..r2 inclusive
        int mid = r1 + ((r2 + 1) - r1) / 2;
        int endCount = r2;
        for (int i = r1; i < mid; i++) {
            int tmp = array[i];
            array[i] = array[endCount];
            array[endCount] = tmp;
            endCount--;
        }
    }
    return array;
}
```

Scramble Mutation for Permutations

- Pick two elements at random
- Randomly rearrange the substring between them

(note subset does not have to be contiguous)

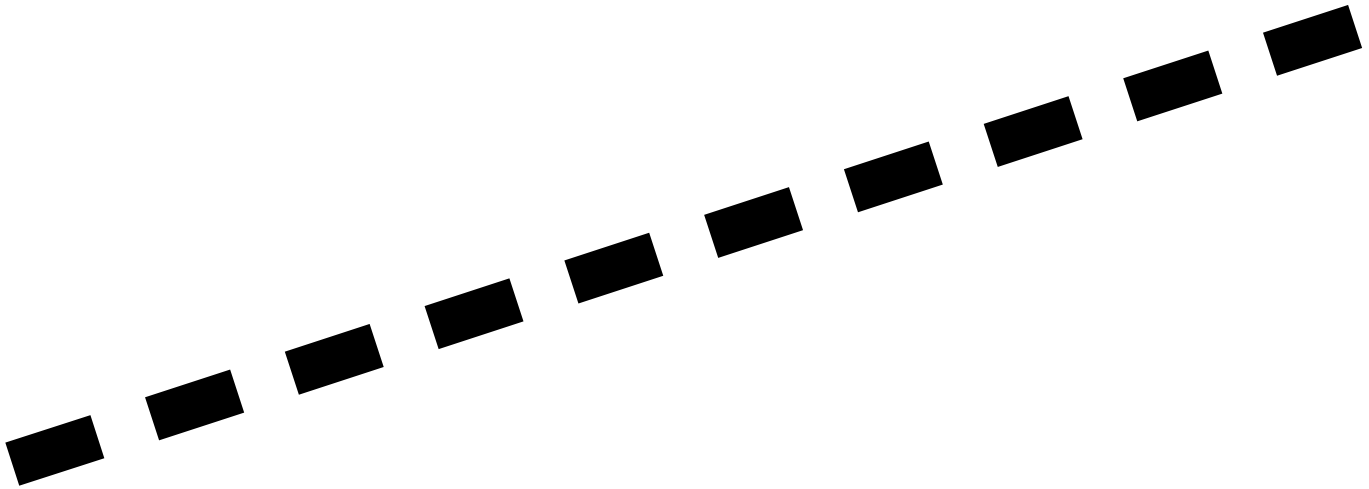
Source: Introduction to Evolutionary Computing by A.E. Eiben and J.E. Smith

Scramble Mutation for Permutations Java Implementation

```
public static int[] scrambleMutation(int[] parent){
    int[] array = parent.clone();
    int l = array.length;
    for(int k = 0; k < 5; k++){//repeat process 5 times
        //get 2 random integers between 0 and size of array
        int r1 = randomNumber(0,l);
        int r2 = randomNumber(0,l);
        //to make sure the r1 < r2
        while(r1 >= r2) {r1 = randomNumber(0,l); r2 = randomNumber(0,l);}
        //this code scrambles (i.e. randomises) elements between r1..r2
        for(int i = 0; i < 10; i++){
            int i1 = randomNumber(r1,r2+1);// add 1 to include actual value of r2
            int i2 = randomNumber(r1,r2+1);// see comments on method randomNumber
            int a = array[i1];
            array[i1] = array[i2];
            array[i2] = a;
        }
    }
    return array;
}
```

The Problem Statement

- The problem domain states that we have the coordinates of 10 short line segments (dashes)
- These dashes form one big dashed line
- We have their coordinates but NOT in the correct order
- We would like to find the right order to form the dashed line



Representation of the Solution

- A solution will be an int array of size 10:

  e.g. [2,1,4,7,3,0,9,5,8,6]

How to do it!

- We will create an array of size 10 to hold the coordinates of the 10 dashes dashes = [((x1,y1),(x2,y2)), ((x1,y1),(x2,y2)),...,((x1,y1),(x2,y2))]
- We will create a chromosome matrix (30 by 10) .. The 30 represents a population size of 30 (It could be any size, but should be big enough to allow some dominant solutions to form)
- Each member of this matrix represents a possible solution (int array of size 10)

  e.g. [2,1,4,7,3,0,9,5,8,6]
- We will use members of each chromosome to access coordinates in the dashes array
- So in java, this will be:

```
//the genes matrix, 30 members, 10 line segments each
int[][] lgenes = new int[30][10];
```

Evolving the Population

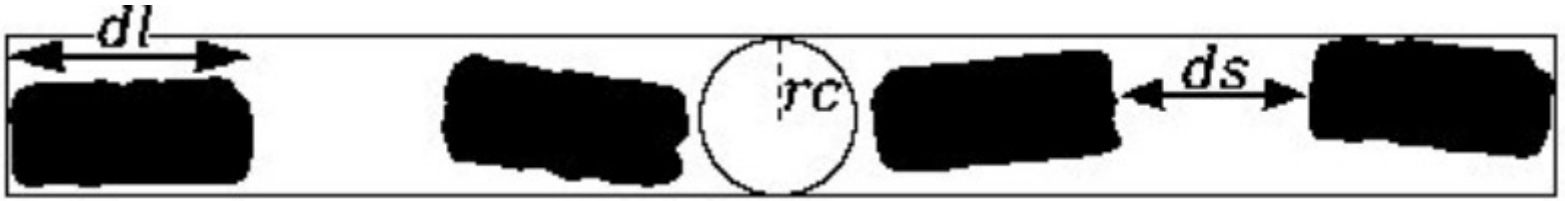
- At each iteration, we will try to have a new population of candidate solutions
- We will try to keep some of the good solutions from the previous iteration (elitism)
- We will choose two chromosomes from the current population
- We will try to apply crossover (randomly)
- If no crossover happens, we will choose the best of them and add it to the new population
- And then apply mutation to all members of the new population (randomly)

The Fitness Function

- For each candidate solution, we will check the array elements for colinearity:
  - Check the first element against the second, the second against the third ... the ninth against the tenth

    (NINE pairwise comparisons)

  - We will count the number of **adjacent & colinear** pairs
  - **If this number is 9 then these elements form a dashed line and we have found the solution**
  - **If this number is NOT 9 then these elements DO NOT form a dashed line BUT we will keep track of the best solution so far**



Implementation

- Classes:
  - Point: a simple class to model a point
  - Dash: a simple class to model a line segment (dash)
  - Utility: a utility class that has several methods such as checking adjacency and colinearity of two line segments, generating random numbers, randomising array contents and more!
  - Mutation: a simple class that has implementation of four mutation methods
  - PermutationGA: the actual class where the GA is run!

Implementation

- Start with a random population
- Repeat the following maxNoTour times:
  - Find the fittest element in the population
  - If the fitness of this element is 9 then we have found the solution
  - Otherwise:
    - keep track of it
    - Evolve the population