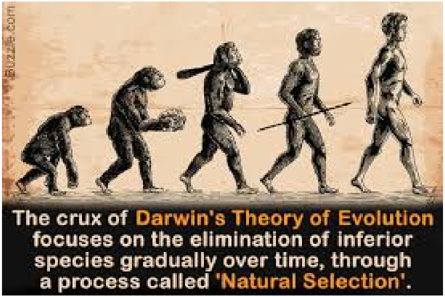
Genetic Algorithm (GA)

Asst.Prof.Dr.Emre Özbilge

Department of Artificial Intelligence Engineering
Faculty of Engineering
Cyprus International University

December 24, 2024

History of Genetic Algorithm I


Buzzle.com
The crux of Darwin's Theory of Evolution focuses on the elimination of inferior species gradually over time, through a process called 'Natural Selection'.

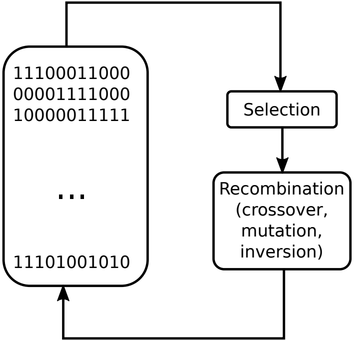
- **Theory of Evolution: How did Darwin come up with it:** https://youtu.be/JOk_0mUT_JU?si=wkdqc-vlSzLlpMZ6

- John Holland introduced the idea of genetic algorithm (GA) in the 1960s as a population-based algorithm with greater biological plausibility than previous approaches.

History of Genetic Algorithm II

- Holland's genetic algorithm used mutation and crossover operators straight from biology to navigate the solution space.
- Potential solutions (or chromosomes) are represented as strings of bits instead of real values.

Figure 1: Holland's bit-string genetic algorithm.

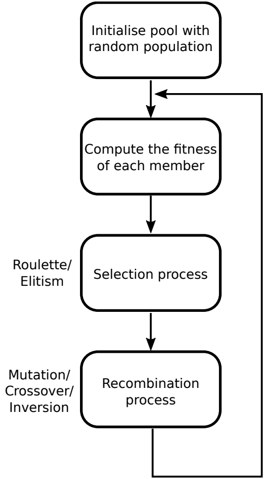


Genetic Algorithm Overview I

- GA is called a population-based technique because instead of operating on a single potential solution, it uses a population of potential solutions.
- The larger the population, the greater the diversity of the members of the population, and the larger the area searched by the population.
- First, a pool of random potential solutions is created that serves as the first generation.
- For the best results, this pool should have adequate diversity (filled with members that differ more than they are similar).
- Then, the fitness of each member is computed.
- The fitness here is a measure of how well each potential solution solves the problem at hand.
- The higher the fitness, the better the solution in relation to others.

Genetic Algorithm Overview II

Figure 2: Simple flow of the genetic algorithm.

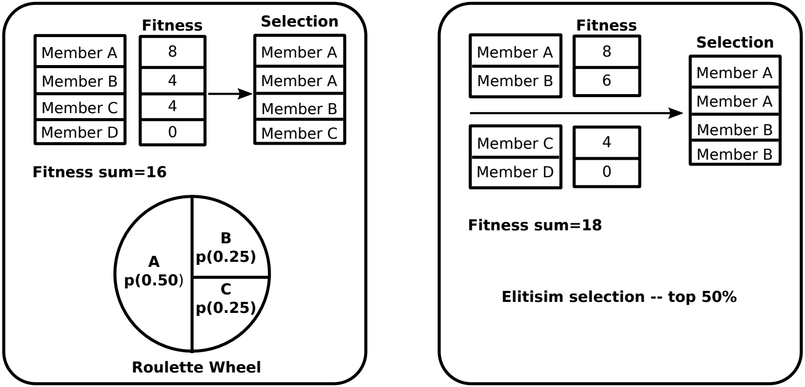

Initialise pool with random population
Compute the fitness of each member
Roulette/ Elitism
Selection process
Mutation/ Crossover/ Inversion
Recombination process

GA Selection: Roulette Wheel and Elitist I

- Two simplest approaches for selecting members of the population are roulette wheel selection and elitist selection.
- Roulette wheel selection is a probabilistic algorithm that selects members of the population proportionate with their fitness (the higher fit the member, the more likely it will be selected).
- In elitist selection, the higher fitness members of the population are selected, forcing the lesser fit members to die off.

GA Selection: Roulette Wheel and Elitist II

Figure 3: Two of the simpler GA selection models.



	Fitness	Selection
Member A	8	Member A
Member B	4	Member A
Member C	4	Member B
Member D	0	Member C

Fitness sum=16

A p(0.50)

B p(0.25)

C p(0.25)

Roulette Wheel

	Fitness
Member A	8
Member B	6
Member C	4
Member D	0

Selection

Member A

Member A

Member B

Member B

Fitness sum=18

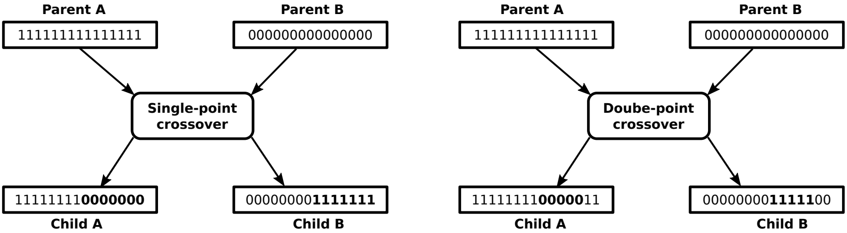
Elitisim selection -- top 50%

GA Operators: Crossover I

- A number of members that have the right to propagate their genetic material to the next population.
- The next step is to recombine these members' material to form the members of the next generation.
- Parents are selected two at a time from the set of individuals that are permitted to propagate (from the selection process).
- Given two parents, two children are created in the new generation with slight alternations courtesy of the recombination process (with a given probability that the genetic operator can occur).
- Using crossover, the parents are combined by picking a crossover point, and then swapping the tails of the two parents to produce the two children.
- Another variant of crossover creates two crossover points, swapping the genetic materials in two places.

GA Operators: Crossover II

Figure 4: Illustrating the crossover operators in genetic recombination.



Parent A

111111111111111

Parent B

000000000000000

Single-point crossover

11111111**0000000**

Child A

00000000**1111111**

Child B

Parent A

111111111111111

Parent B

000000000000000

Doube-point crossover

11111111**00000**11

Child A

00000000**11111**00

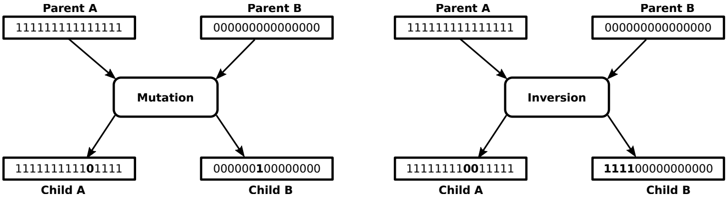
Child B

CYPRUS INTERNATIONAL UNIVERSITY

GA Operators: Mutation and Inversion

- Each of these operators were the original genetic operators from Holland's original work. The mutation operator simply mutates (or flips) a bit.
- In real-valued chromosomes, a slight change to the value can also be performed as mutation (small increment or decrement of the value).
- The inversion operator takes a piece of the chromosome, and inverts it. In this case, the range of bits are flipped.

Figure 5: Illustrating the mutation and inversion genetic operators.



GA Termination

- There are a number of ways that we can terminate the process.
- The most obvious is to end when a solution is found, or one that meets the designer's criteria.
- Another termination criterion, potentially returning a suboptimal solution, is when the population lacks diversity, and therefore the inability to adequately search the solution space.
- When the members of the population become similar, there's a loss in the ability to search.
- To combat this, we terminate the algorithm early by detecting if the average fitness of the population is near the maximum fitness of any member of the population. For example, when average fitness is greater than 0.99 times the maximum fitness.
- Once the population becomes too similar, the members have focused on similar areas of the search space, and are therefore incapable of branching out to new areas in search of more fit individuals.
- Another option is that run the GA with a several episodes (*e.g.* 1000, 10000) and get the fittest chromosome at the end of these runs.

Evolution Inside Your Computer I

- The way genetic algorithms work is essentially mimicking evolution.
- First, you figure out a way of encoding any potential solution to your problem as a “digital” chromosome.
- Then, you create a start population of random chromosomes (each one representing a different candidate solution) and evolve them over time by “breeding” the fittest individuals and adding a little mutation here and there.
- With a bit of luck, over many generations, the genetic algorithm will converge upon a solution.
- Genetic algorithms do not guarantee a solution, nor do they guarantee to find the best solution, but if utilised the correct way, you will generally be able to code a genetic algorithm that will perform well.
- The best thing about genetic algorithms is that you do not need to know how to solve a problem; you only need to know how to encode it in a way the genetic algorithm mechanism can utilise.
- Typically, the chromosomes are encoded as a series of binary bits.

Evolution Inside Your Computer II

- At the start of a run, you create a population of chromosomes, and each chromosome has its bits set at random.
- The length of the chromosome is usually fixed for the entire population. As an example, this is what a chromosome of length twenty may look like: **01010010100101001111**
- The important thing is that each chromosome is encoded in such a way that the string of bits may be decoded to represent a solution to the problem at hand.
- It may be a very poor solution, or it may be a perfect solution, but every single chromosome represents a possible solution.
- Usually the initial population is terrible because it is created with random bits without knowing the task which is going to be solved.

GA Evolution Loop

- **Loop until a solution is found:**
    1. Test each chromosome to see how good it is at solving the problem and assign a fitness score accordingly.
    2. Select two members from the current population. The probability of being selected is proportional to the chromosome's fitness – the higher the fitness, the better the probability of being selected. A common method for this is called Roulette wheel selection.
    3. Dependent on the Crossover Rate, crossover the bits from each chosen chromosome at a randomly chosen point.
    4. Step through the chosen chromosome's bits and flip dependent on the Mutation Rate.
    5. Repeat steps 2, 3, and 4 until a new population of one hundred members has been created.
- **End loop**

What's the Crossover Rate?

- The crossover rate is simply the probability that two chosen chromosomes will swap their bits to produce two new offspring.
- Experimentation has shown that a good value for this is typically around 0.7, although some problem domains may require much higher or lower values.
- Every time you choose two chromosomes from the population, you test to see if they will crossover bits by generating a random number between 0 and 1.
- If the number is lower than the crossover rate (0.7), then you choose a random position along the length of the chromosome and swap all the bits after that point.
- For example, given two chromosomes:

100010011 10010010
010100010 01000011

- you choose a random position along the length, say 10, and swap all the bits to the right of that point accordingly. So the chromosomes become:

100010011 01000011
010100010 **10010010**

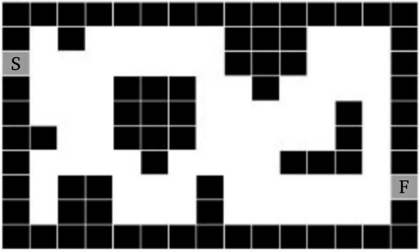
What’s the Mutation Rate?

- The mutation rate is the probability that a bit within a chromosome will be flipped (a 0 becomes 1, and a 1 becomes 0).
- This is usually a very low value for binary encoded genes, for example 0.001.
- Whenever you choose chromosomes from the population, you first check for crossover, and then you move through each bit of the offspring’s chromosomes and test to see if it needs to be mutated.

Case Study I: Path-finding I

- The maze is defined as a matrix; a 0 will represent open space, a 1 will represent a wall or an obstacle, a 5 will be the start point, and an 8 will be the finish.
- The intelligent agent starts from S to go F.

1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	0	1	0	0	0	0	0	1	1	1	0	0	0	1
8	0	0	0	0	0	0	0	1	1	1	0	0	0	1
1	0	0	0	1	1	1	0	0	1	0	0	0	0	1
1	0	0	0	1	1	1	0	0	0	0	0	1	0	1
1	1	0	0	1	1	1	0	0	0	0	0	1	0	1
1	0	0	0	0	1	0	0	0	0	1	1	1	0	1
1	0	1	1	0	0	0	1	0	0	0	0	0	0	5
1	0	1	1	0	0	0	1	0	0	0	0	0	0	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1



Case Study I: Path-finding II

- Each chromosome must be encoded to represent the movement of the agent. Agent's movement is restricted to four directions: North, South, East, and West, so the encoded chromosomes should be strings of information representing these four directions. The traditional method of encoding is changing the directions into binary code. Only two bits are necessary to represent four directions.

Code	Decoded	Direction
00	0	North
01	1	South
10	2	East
11	3	West

- If you take a random string of bits, you can decode them into a series of directions for agent to follow. For example the chromosome:

111110011011101110010101

- This represents the genes:

11, 11, 10, 01, 10, 11, 10, 11, 10, 01, 01, 01

- When decoded from binary to decimal become:

3, 3, 2, 1, 2, 3, 2, 3, 2, 1, 1, 1

Case Study I: Path-finding III

Code	Decoded	Direction
11	3	West
11	3	West
10	2	East
01	1	South
10	2	East
11	3	West
10	2	East
11	3	West
10	2	East
01	1	South
01	1	South
01	1	South

- The length of the chromosome must be long enough to find a solution and also provide various of solution for validiy of the optimal solution.
- In this particularly problem, we use 70 bits length chromosome which allow us maximum upto 35 movements.

Case Study I: Path-finding IV

- In each generation, test each member of the population and assign its fitness score.
- Check how far the agent traverses through the map and calculate a fitness socre proportional to the agent's finished distance from the exit.

$$fitness^i = \frac{1}{|goal_x - agent^i_x| + |goal_y - agent^i_y| + 1}$$

- **Case study:** Can you provide better fitness function, how to modify this function to get better solution?

Loop of Genetic Algorithm

- Repeat until find the solution
  1. Update the fitness score of all members of the population
  2. Find the sum of the fitness scores
  3. Normalised all the fitness score of each member
  4. Repeat the followings until generate N number of new chromosomes (N is population size)
     - 4.1 Select mum and dad choromosomes using one of the selection methods
     - 4.2 Apply crossover operator and generate two child chromosomes
     - 4.3 Perform mutate operator for both child chromosomes
     - 4.4 Store these two child chromosomes into the new population

- **Simulation's video:** https://youtu.be/_pTrsCANL-M?si=BD-sGjZjghK7sLLN

Evolution Inside Your Computer I

- The way genetic algorithms work is essentially mimicking evolution.
- First, you figure out a way of encoding any potential solution to your problem as a “digital” chromosome.
- Then, you create a start population of random chromosomes (each one representing a different candidate solution) and evolve them over time by “breeding” the fittest individuals and adding a little mutation here and there.
- With a bit of luck, over many generations, the genetic algorithm will converge upon a solution.
- Genetic algorithms do not guarantee a solution, nor do they guarantee to find the best solution, but if utilised the correct way, you will generally be able to code a genetic algorithm that will perform well.
- The best thing about genetic algorithms is that you do not need to know how to solve a problem; you only need to know how to encode it in a way the genetic algorithm mechanism can utilise.
- Typically, the chromosomes are encoded as a series of binary bits.

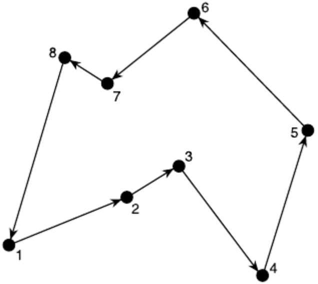
Evolution Inside Your Computer II

- At the start of a run, you create a population of chromosomes, and each chromosome has its bits set at random.
- The length of the chromosome is usually fixed for the entire population. As an example, this is what a chromosome of length twenty may look like: **01010010100101001111**
- The important thing is that each chromosome is encoded in such a way that the string of bits may be decoded to represent a solution to the problem at hand.
- It may be a very poor solution, or it may be a perfect solution, but every single chromosome represents a possible solution.
- Usually the initial population is terrible because it is created with random bits without knowing the task which is going to be solved.

Case Study II: Traveling Salesman Problem (TSP) I

Definition:

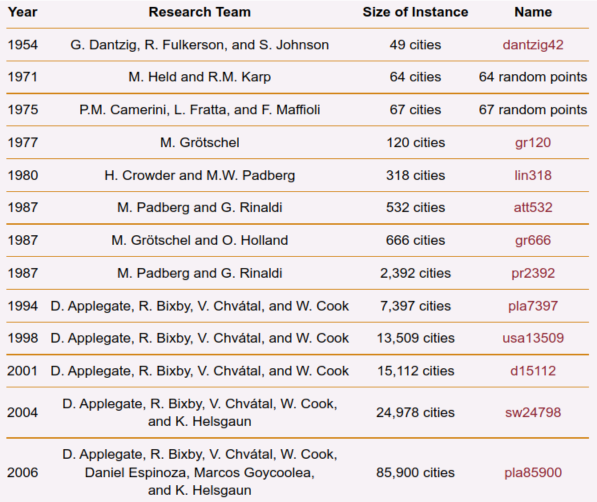
- Given a collection of cities, the traveling salesman must determine the shortest route that will enable him to visit each city precisely once and then return back to his starting point.
- The distance between each city is given and is assumed to be the same in both directions.
- Objective is to minimize the total distance to be travelled.



Case Study II: Traveling Salesman Problem (TSP) II

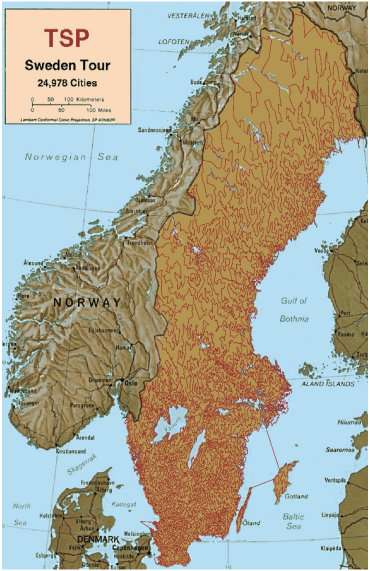
- This type of problem frequently occurs when coding the AI for strategy games. Often it's necessary to create the shortest path for a unit that will start at one waypoint, end at another, and pass through several predefined areas along the way, to pick up resources, food, energy, and so on. It can also be used as part of the route planning AI for a Quake-like FPS bot.
- Some of the landmarks in TSP solving, starting from the 1950s.

Case Study II: Traveling Salesman Problem (TSP) III



Year	Research Team	Size of Instance	Name
1954	G. Dantzig, R. Fulkerson, and S. Johnson	49 cities	dantzig42
1971	M. Held and R.M. Karp	64 cities	64 random points
1975	P.M. Camerini, L. Fratta, and F. Maffioli	67 cities	67 random points
1977	M. Grötschel	120 cities	gr120
1980	H. Crowder and M.W. Padberg	318 cities	lin318
1987	M. Padberg and G. Rinaldi	532 cities	att532
1987	M. Grötschel and O. Holland	666 cities	gr666
1987	M. Padberg and G. Rinaldi	2,392 cities	pr2392
1994	D. Applegate, R. Bixby, V. Chvátal, and W. Cook	7,397 cities	pla7397
1998	D. Applegate, R. Bixby, V. Chvátal, and W. Cook	13,509 cities	usa13509
2001	D. Applegate, R. Bixby, V. Chvátal, and W. Cook	15,112 cities	d15112
2004	D. Applegate, R. Bixby, V. Chvátal, W. Cook, and K. Helsgaun	24,978 cities	sw24798
2006	D. Applegate, R. Bixby, V. Chvátal, W. Cook, Daniel Espinoza, Marcos Goycoolea, and K. Helsgaun	85,900 cities	pla85900

Case Study II: Traveling Salesman Problem (TSP) IV



Case Study II: Traveling Salesman Problem (TSP) V

Problem description:

- The main difference with the TSP is that solutions rely on permutations, and therefore, you have to make sure that all your genomes represent a valid permutation of the problem – a valid tour of all the cities.
- If you were to represent possible solutions using the binary encoding and crossover operator from the Pathfinder problem, you can see how you would run into difficulties very quickly.
- Take the eight city example. You could encode each city as a 3-bit binary number, numbering the cities from 0 to 7.
- Choose a crossover point (represented by an x) after the fourth city, and see what offspring you get.

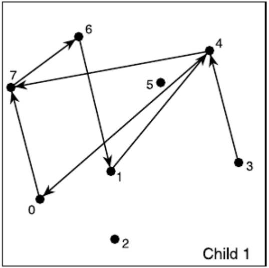
Case Study II: Traveling Salesman Problem (TSP) VI

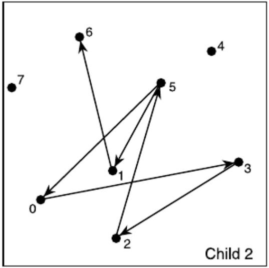
Before Crossover

	Binary Encoded Tour	Decoded Tour
Parent 1	011 100 000 111 x 010 101 001 110	3, 4, 0, 7, 2, 5, 1, 6
Parent 2	010 101 000 011 x 110 001 100 111	2, 5, 0, 3, 6, 1, 4, 7

After Crossover

	Binary Encoded Tour	Decoded Tour
Child 1	011 100 000 111 x 110 001 100 111	3, 4, 0, 7, 6, 1, 4, 7
Child 2	010 101 000 011 x 010 101 001 110	2, 5, 0, 3, 2, 5, 1, 6





TSP: The Permutation Crossover Operator (PMX) I

- To generate valid offspring, Partially-Mapped Crossover or PMX as it's more widely known is used.
- Assuming the eight city problem has been encoded using integers, two possible parents may be:

  Parent1: 2 . 5 . 0 . 3 . 6 . 1 . 4 . 7
  Parent2: 3 . 4 . 0 . 7 . 2 . 5 . 1 . 6

- To implement PMX, you must first choose two random crossover points – let's say after cities 3 and 6. So, the split is made at the x's, like so:

  Parent1: 2 . 5 . 0 . x **3 . 6 . 1** x . 4 . 7
  Parent2: 3 . 4 . 0 . x **7 . 2 . 5** x . 1 . 6

- Then you look at the two center sections and make a note of the mapping between parents. In this example:

  3 is mapped to **7**
  6 is mapped to **2**
  1 is mapped to **5**

TSP: The Permutation Crossover Operator (PMX) II

- Now, iterate through each parent's genes and swap the genes wherever a gene is found that matches one of those listed. Step by step it goes like this:

Step 1

Child1: 2 . 5 . 0 . 3 . 6 . 1 . 4 . 7
Child2: 3 . 4 . 0 . 7 . 2 . 5 . 1 . 6

(*here the children are just direct copies of their parents*)

Step 2 [3 and 7]

Child1: 2 . 5 . 0 . **7** . 6 . 1 . 4 . **3**
Child2: **7** . 4 . 0 . **3** . 2 . 5 . 1 . 6

Step 3 [6 and 2]

Child1: **6** . 5 . 0 . 7 . **2** . 1 . 4 . 3
Child2: 7 . 4 . 0 . 3 . **6** . 5 . 1 . **2**

TSP: The Permutation Crossover Operator (PMX) III

Step 4 [1 and 5]

Child1: 6 . **1** . 0 . 7 . 2 . **5** . 4 . 3

Child2: 7 . 4 . 0 . 3 . 6 . **1** . **5** . 2

TSP: The Exchange Mutation Operator (EM)

- After PMX, mutation operator must be also produced valid tours. The Exchange Mutation operator does this by choosing two genes in a chromosome and swapping them. For example, given the following chromosome:

$$5 \,.\, \mathbf{3} \,.\, 2 \,.\, 1 \,.\, 7 \,.\, \mathbf{4} \,.\, 0 \,.\, 6$$

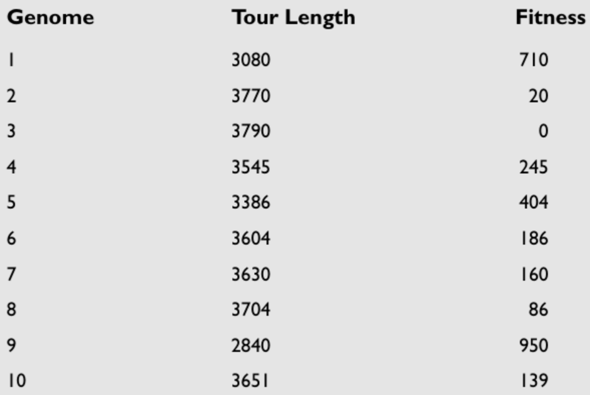
- The mutation function chooses two genes at random, for example 4 and 3, and swaps them:

$$5 \,.\, \mathbf{4} \,.\, 2 \,.\, 1 \,.\, 7 \,.\, \mathbf{3} \,.\, 0 \,.\, 6$$

TSP: Deciding on a Fitness Function I

- Tour length from city A to B is computed by using *Euclidean distance*. So total tour lenght for each choromosome is the total Euclidean distance between the cities which are visited to complete the tour.
- The worst tour length each generation is found and then iterate through the population again subtracting each genome's tour distance from the worst. It also effectively removes the worst chromosome from the population, because it will have a fitness score of zero and, therefore, will never get selected during the roulette wheel selection procedure.

TSP: Deciding on a Fitness Function II



Genome	Tour Length	Fitness
1	3080	710
2	3770	20
3	3790	0
4	3545	245
5	3386	404
6	3604	186
7	3630	160
8	3704	86
9	2840	950
10	3651	139

- In this problem, adding *elitism* can help the genetic algorithm converge more quickly. So, *n* instances of the fittest genome from the previous generation will be copied unchanged into the new population.

Building a Better GA: Alternative Permutation Mutation Operators I

Scramble Mutation (SM)

- Choose two random points and "scramble" the cities located between them:

0 .1 . 2 . **3** . **4** . **5** . **6** . 7
becomes
0 .1 . 2 . **5** . **6** . **3** . **4** . 7

Displacement Mutation (DM)

- Select two random points, grab the chunk of chromosome between them, and then reinsert at a random position displaced from the original.

0 . 1 . 2 . **3** . **4** . **5** . 6 . 7
becomes
0 . **3** . **4** . **5** . 1 . 2 . 6 . 7

Building a Better GA: Alternative Permutation Mutation Operators II

Insertion Mutation (IM)

- This is a very effective mutation and is almost the same as the DM operator, except here only one gene is selected to be displaced and inserted back into the chromosome. In tests, this mutation operator has been shown to be consistently better than any of the alternatives mentioned here.

0 . 1 . **2** . 3 . 4 . 5 . 6 . 7

becomes

0 . 1 . 3 . 4 . 5 . **2** . 6 . 7

Building a Better GA: Alternative Permutation Mutation Operators III

Inversion Mutation (IVM)

- This is a very simple mutation operator. Select two random points and reverse the cities between them.

0 . **1** . **2** . **3** . **4** . 5 . 6 . 7
becomes
0 . **4** . **3** . **2** . **1** . 5 . 6 . 7

Displaced Inversion Mutation (DIVM)

- Select two random points, reverse the city order between the two points, and then displace them somewhere along the length of the original chromosome. This is similar to performing IVM and then DM using the same start and end points.

0 . 1 . 2 . 3 . **4** . **5** . **6** . 7
becomes
0 . **6** . **5** . **4** . 1 . 2 . 3 . 7

Building a Better GA: Alternative Permutation Crossover Operators I

Order-Based Crossover (OBX)

- To perform order-based crossover, several cities are chosen at random from one parent and then the order of those cities is imposed on the respective cities in the other parent.

  Parent1: 2 . **5** . **0** . 3 . 6 . **1** . 4 . 7
  Parent2: 3 . 4 . 0 . 7 . 2 . 5 . 1 . 6

- The cities in bold are the cities which have been chosen at random. Now, impose the order – 5, 0, then 1 – on the same cities in Parent2 to give Offspring1 like so:

  Offspring1: 3 . 4 . **5** . 7 . 2 . **0** . **1** . 6

Building a Better GA: Alternative Permutation Crossover Operators II

- City one stayed in the same place because it was already positioned in the correct order. Now the same sequence of actions is performed on the other parent. Using the same positions as the first,

  Parent1: 2 . 5 . 0 . 3 . 6 . 1 . 4 . 7
  Parent2: 3 . **4** . **0** . 7 . 2 . **5** . 1 . 6

- Parent1 becomes:

  Offspring2: 2 . **4** . **0** . 3 . 6 . 1 . **5** . 7

Building a Better GA: Alternative Permutation Crossover Operators III

Position-Based Crossover (PBX)

- This is similar to Order-Based Crossover, but instead of imposing the order of the cities, this operator imposes the position. So, using the same example parents and random positions, here's how to do it.

Parent1: 2 . **5** . **0** . 3 . 6 . **1** . 4 . 7
Parent2: 3 . **4** . **0** . 7 . 2 . **5** . 1 . 6

- First, move over the selected cities from Parent1 to Offspring1, keeping them in the same position.

OffSpring1: * . **5** . **0** . * . * . **1** . * . *

- Now, iterate through Parent2's cities and fill in the blanks if that city number has not already appeared. In this example, filling in the blanks results in:

Offspring1: 3 . **5** . **0** . 4 . 7 . **1** . 2 . 6

Building a Better GA: Alternative Permutation Crossover Operators IV

- Get it? Let's run through the derivation of Offspring2, just to be sure. First, copy over the selected cities into the same positions.

  Offspring2: * . **4** . **0** . * . * . **5** . * . *

- Now, fill in the blanks.

  Offspring2: 2 . **4** . **0** . 3 . 6 . **5** . 1 . 7

Building a Better GA: Selection Techniques I

Elitism

- Elitism is a way of guaranteeing that the fittest members of a population are retained for the next generation. Selecting *n* copies of the top *m* individuals of the population to be retained. Retaining about 2-5% of the population size yields good results.

Steady State Selection

- Main idea of this selection is that big part of chromosomes should survive to next generation. GA then works in a following way. In every generation are selected a few (good - with high fitness) chromosomes for creating a new offspring. Then some (bad - with low fitness) chromosomes are removed and the new offspring is placed in their place. The rest of population survives to new generation.

Building a Better GA: Selection Techniques II

Example:

Individual	Fitness
A	10
B	30
C	50
D	70
E	90

1. Step 1: Parent Selection: Suppose we use tournament selection or some other method to select parents for reproduction. Let's say **C** and **D** are selected as parents.
2. Step 2: Crossover and Mutation: The offspring generated from these parents through crossover (and possibly mutation) are evaluated. Let's say the offspring have the following fitness:

$$\text{Offspring 1: Fitness} = 60$$
$$\text{Offspring 2: Fitness} = 80$$

3. Step 3: Replacement: Now, instead of replacing the entire population, we identify the weakest individuals in the current population. From the original population, individuals A (fitness $= 10$) and B (fitness $= 30$) have the lowest fitness values.

Building a Better GA: Selection Techniques III

4. Step 4: Population Update: The weakest individuals, A and B, are replaced by the offspring:

Individual	Fitness
C	50
D	70
E	90
OffSpring1	60
Offspring2	80

⋆ In most implementations, 2 parents are selected to produce 2 offspring. Then, the 2 worst individuals in the population are replaced by the 2 offspring.

⋆ Typical approach, new offspring always replace the selected worst individuals, even if the offspring are of lower fitness.

Building a Better GA: Selection Techniques IV

Advantages:

- Maintains Diversity: By only replacing a small portion of the population, steady-state selection helps maintain genetic diversity for a longer period, avoiding premature convergence.
- Gradual Evolution: Changes occur gradually because the entire population is not replaced at once, leading to more stable and incremental improvements.
- Exploration and Exploitation Balance: It strikes a balance between exploring new solutions and exploiting existing ones. Good individuals have a higher chance of staying in the population and passing on their genes.

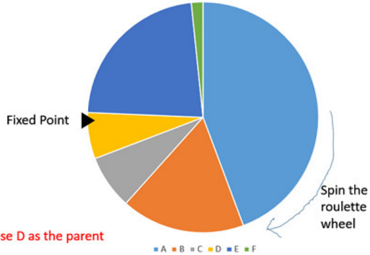
Disadvantages:

- Slower Convergence: Since only a few individuals are replaced at a time, it might take longer to converge to a solution compared to generational replacement methods.
- Risk of Stagnation: If not managed carefully, steady-state selection may lead to stagnation where the population becomes too similar (genetic drift), and little progress is made.

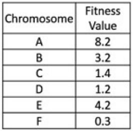
Building a Better GA: Selection Techniques V

Fitness Proportionate Selection (FPS)

- Roulette wheel selection is a widely used method for fitness proportionate selection in genetic algorithms. However, this method has limitations. Due to its dependence on random numbers and the relatively *small population sizes* (typically 50 to 200), the actual number of offspring per individual can significantly differ from expectations. More critically, roulette wheel selection may entirely miss the most fit individuals. This risk underscores the importance of incorporating elitism, which ensures that the best individuals are retained.



Choose D as the parent



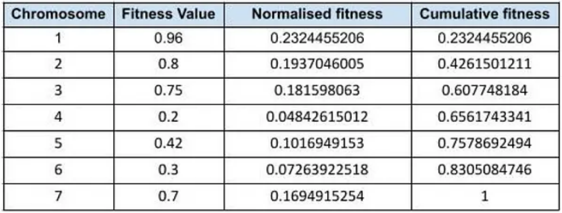
Chromosome	Fitness Value
A	8.2
B	3.2
C	1.4
D	1.2
E	4.2
F	0.3

Building a Better GA: Selection Techniques VI

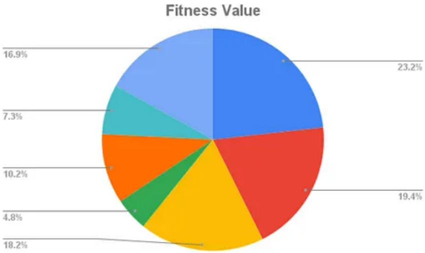
Algorithm:

1. Find the sum of all fitness values in a population(S)
2. Find normalised fitness values (=fitness value/S)
3. Find cumulative fitness values
4. Generate a random number p between [0,1]
5. The individual for which p is just smaller than the cumulative sum is selected.
   - *e.g.* consider the Cumulative Fitness Values. If $p < 0.232$, Individual 1 is selected; if $0.232 < p < 0.426$, Individual 2 is selected, and so on.
6. Alternatively, whenever the sum of normalised fitness score passes the value p, the last added chromosome is selected as a parent.

Building a Better GA: Selection Techniques VII



Chromosome	Fitness Value	Normalised fitness	Cumulative fitness
1	0.96	0.2324455206	0.2324455206
2	0.8	0.1937046005	0.4261501211
3	0.75	0.181598063	0.607748184
4	0.2	0.04842615012	0.6561743341
5	0.42	0.1016949153	0.7578692494
6	0.3	0.07263922518	0.8305084746
7	0.7	0.1694915254	1



Building a Better GA: Selection Techniques VIII

Rank-Based Selection

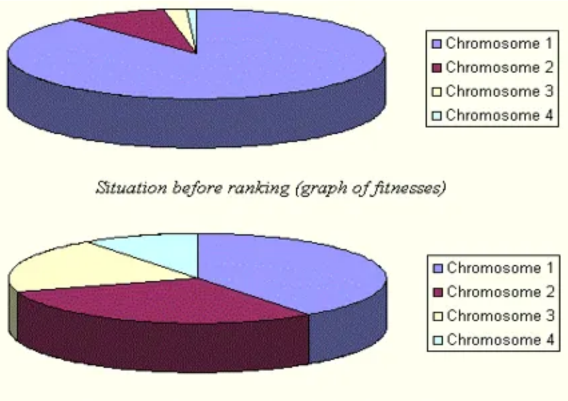
- If an initial population contains one or two very fit but not the best individuals and the rest of the population are not good, then these fit individuals will quickly dominate the whole population and prevent the population from exploring other potentially better individuals. Such a strong domination causes a very high loss of genetic diversity which is definitely not advantageous for the optimization process.

Rank Selection:

1. Rank selection first ranks the population and then every chromosome receives fitness from this ranking.
2. The worst will have fitness 1, second worst 2 etc. and the best will have fitness N (number of chromosomes in population).
3. After this all the chromosomes have a chance to be selected.
4. Rank-based selection schemes can avoid premature convergence.
5. But can be computationally expensive because it sorts the populations based on fitness value.

Building a Better GA: Selection Techniques IX

6. But this method can lead to slower convergence, because the best chromosomes do not differ so much from other ones.



Situation before ranking (graph of fitnesses)

Situation after ranking (graph of order numbers)

Building a Better GA: Selection Techniques X

Algorithm:

1. First sort the Fitness value of the Population in ascending order.
2. Then assign 1 to the worst individual which is the beginning of the the sorted population, then assign 2 to the next individual until the last individual which is the best one, that is N (N is total individual in the population).
3. After that compute the normalised fitness score.
4. Then calculate cumulative fitness.
5. Perform *Roulette Wheel Selection*.

Building a Better GA: Selection Techniques XI

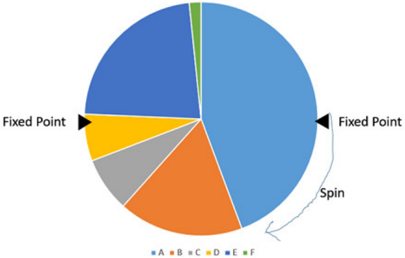
Example: In this way, the fraction of individuals is distributed, so make more chance for the lower individual to be selected. Therefore, the diversity of population can be increased.

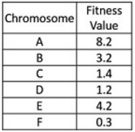
Chromosome	Fitness	Fraction	Ranked Fitness	New Fraction
A	5	50%	5	33.33%
B	2	20%	4	26.66%
C	0.5	5%	1	6.66%
D	1.5	15%	3	20%
E	1	10%	2	13.33%

Building a Better GA: Selection Techniques XII

Stochastic Universal Sampling (SUS)

- SUS addresses challenges in fitness proportionate selection within small populations by using multiple evenly spaced pointers rotated once, rather than repeatedly spinning a single wheel. The number of pointers corresponds to the desired number of offspring.





Chromosome	Fitness Value
A	8.2
B	3.2
C	1.4
D	1.2
E	4.2
F	0.3

Steps of SUS:

1. Calculate Total Fitness: The total fitness of the population is computed by summing the fitness values of all individuals.

Building a Better GA: Selection Techniques XIII

2. Assign Selection Pointers: A series of equally spaced ”pointers” (selection points) are distributed across the fitness scale. The distance between these pointers is calculated by dividing the total fitness by the number of individuals to be selected.
3. Place the Wheel: A random starting point is chosen in the range of [0, total fitness / number of individuals], and then the equally spaced pointers are placed at this starting point and each subsequent distance.
4. Select Individuals: Each pointer selects the individual whose cumulative fitness encloses the pointer.

Example:

Let’s assume we have a population of 5 individuals with the following fitness values:

Individual	**Fitness**
A	10
B	20
C	30
D	25
E	15

Building a Better GA: Selection Techniques XIV

1. Step 1: Calculate Total Fitness

$$Total\_fitness = 10 + 20 + 30 + 25 + 15 = 100$$

2. Step 2: Assign Selection Pointers. Let's say we need to select 3 individuals.

$$Pointer\_distance = \frac{Total\_fitness}{Number\_of\_selections} \approx 33.33$$

3. Step 3: Random Starting Point. We choose a random starting point between 0 and 33.33. For this example, let's say the starting point is 10. Thus, the pointers will be located at the following positions on the cumulative fitness scale:
   - First pointer at 10
   - Second pointer at $10 + 33.33 = 43.33$
   - Third pointer at $43.33 + 33.33 = 76.66$
4. Step 4: Calculate Cumulative Fitness

Individual	Fitness	Cumulative Fitness
A	10	10
B	20	30
C	30	60
D	25	85
E	15	100

Building a Better GA: Selection Techniques XV

5. Step 5: Select Individuals. We check which individuals are selected by looking at where the pointers land on the cumulative fitness scale:
   - The first pointer is at 10, which is the cumulative fitness of **Individual A**.
   - The second pointer is at 43.33, which is between 30 (B) and 60 (C), so **Individual C** is selected.
   - The third pointer is at 76.66, which is between 60 (C) and 85 (D), so **Individual D** is selected.

⋆ Once the new population is selected, the individuals are paired sequentially for crossover.

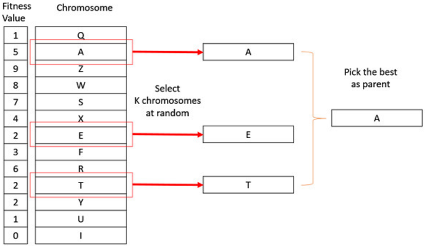
⋆ SUS works with positive fitness score, if you have negative fitness score, adjust all of them to positive by adding the minimum fitness score to the all.

⋆ If you use SUS in your own genetic algorithms, it is inadvisable to use elitism with it because this tends to mess up the algorithm.

Building a Better GA: Selection Techniques XVI

Tournament Selection

- K-Way tournament selection randomly picks K individuals from the population and selects the fittest as a parent. This process is repeated to choose additional parents. It is widely used in research for its ability to operate with negative fitness values, enhancing its versatility.



Fitness Value	Chromosome
1	Q
5	A
9	Z
8	W
7	S
4	X
2	E
3	F
6	R
2	T
2	Y
1	U
0	I

Algorithm:

1. For each parent selection.
    1.1 Select k individuals from the population and perform a tournament among them.
    1.2 Select the best individual from the k individuals.

Building a Better GA: Selection Techniques XVII

Random Selection

This method involves randomly selecting parents from the current population, lacking any bias towards fitter individuals, and is therefore generally not favored.

Algorithm:

```
1. parentA = RandInt(0,PopSize) # return random idx
2. parentB = parentA
3. while parentB == parentA
    3.1 parentB = RandInt(0,PopSize) # return random idx
```

CYPRUS INTERNATIONAL UNIVERSITY

Building a Better GA: Fitness Scaling Techniques I

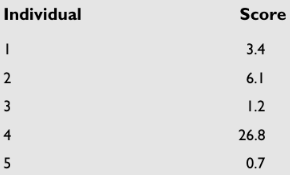
Rank Scaling

- Rank scaling can be a great way to prevent too quick convergence, particularly at the start of a run when it's common to see a very small percentage of individuals outperforming all the rest.
- The individuals in the population are simply ranked according to fitness, and then a new fitness score is assigned based on their rank.
- So, for example, if you had a population of five individuals with the fitness scores as shown in Figure 6, all you do is sort them and assign a new fitness based on their rank within the sorted population (see Figure 7).
- Once the new ranked fitness scores have been applied, you select individuals for the next generation using roulette wheel selection or a similar fitness proportionate selection method.

Building a Better GA: Fitness Scaling Techniques II

- This technique avoids the possibility that a large percentage of each new generation is being produced from a very small number of highly fit individuals, which can quickly lead to premature convergence. You will find that the greater diversity provided by this technique leads to a more successful result for your genetic algorithm.

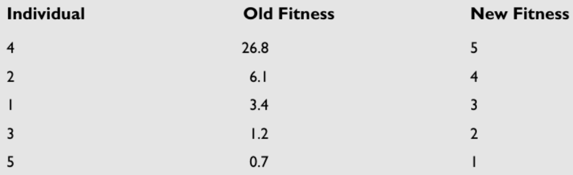
Figure 6: Fitness Scores Before Ranking.



Individual	Score
1	3.4
2	6.1
3	1.2
4	26.8
5	0.7

Building a Better GA: Fitness Scaling Techniques III

Figure 7: Fitness Scores After Ranking.



Individual	Old Fitness	New Fitness
4	26.8	5
2	6.1	4
1	3.4	3
3	1.2	2
5	0.7	1

Building a Better GA: Fitness Scaling Techniques IV

Sigma Scaling

- If you use raw fitness scores as a basis for selection, the population may converge too quickly, and if they are scaled as in rank selection, the population may converge too slowly.
- Sigma scaling is an attempt to keep the selection pressure constant over many generations.
- At the beginning of the genetic algorithm, when fitness scores can vary wildly, the fitter individuals will be allocated less expected offspring.
- Toward the end of the algorithm, when the fitness scores are becoming similar, the fitter individuals will be allocated more expected offspring.

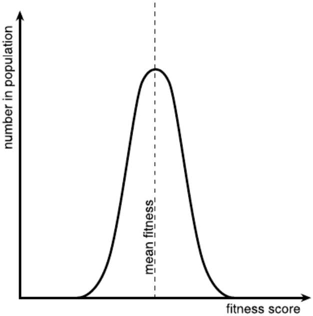
$$NewFitness = \frac{OldFitness - AverageFitness}{2\sigma}$$

where σ is standard deviation.

- The standard deviation is the square root of the population's variance. The variance is a measure of spread within the fitness scores.

Building a Better GA: Fitness Scaling Techniques V

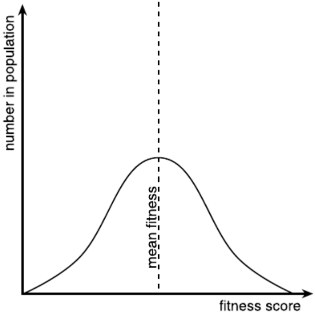
Figure 8: Population with a low spread.



- The hump in the middle of the graph represents the mean (the average) fitness score. Most of the population's scores are clustered around this hump. The spread, or variance, is the width of the hump at the base.

Building a Better GA: Fitness Scaling Techniques VI

Figure 9: Population with a high spread.



- A population with a high variance, and as you can see, the hump is lower and more spread out.

Building a Better GA: Fitness Scaling Techniques VII

- Now that you know what variance is, let me show you how to calculate it. Imagine we are only dealing with a population of three, and the fitness scores are 1, 2, and 3. To calculate the variance, first calculate the mean of all the fitness scores.

$$mean = \frac{1+2+3}{3} = 2$$

$$variance = \frac{(1-2)^2 + (2-2)^2 + (3-2)^2}{3} = 0.667$$

- Standard deviation is the square root of the variance:

$$\sigma = \sqrt{variance}$$

Building a Better GA: Fitness Scaling Techniques VIII

Boltzmann Scaling

- You have learned how to keep the selection pressure constant over a run of your genetic algorithm by using sigma scaling, but sometimes you may want the selection pressure to vary.
- A common scenario is one in which you require the selection pressure to be low at the beginning so that diversity is retained, but as the genetic algorithm converges closer toward a solution, you want mainly the fitter individuals to produce offspring.
- One way of achieving this is by using Boltzmann scaling. This method of scaling uses a continuously varying temperature to control the rate of selection.
- Increases the selection pressure as the temperature decreases, helping the algorithm focus more on fitter individuals in later generations.

$$f_i^{'} = \frac{e^{f_i / T_k}}{\sum_j e^{f_j / T_k}}$$

Building a Better GA: Fitness Scaling Techniques IX

- Each generation, the temperature is decreased by a small value, which has the effect of increasing the selection pressure toward the fitter individuals. Pseudocode is given as follows:

```
1. reduce the temperature a little each generation:
   $T_k = T_{k-1} - \alpha$
2. make sure it doesn't fall below minimum value:
   if( $T_k < T_{\min}$ )
       $T_k = T_{\min}$
```

- For example: decreasing temperature (α) is 0.05, minimum possible temperature ($T_{\min}$) is defined as 1 and the initial temperature (T_0) starts from 50.

Example of using Boltzmann Scaling Technique I

- Suppose the fitness values for 4 chromosomes.
- Fitness: [10, 7, 12, 1]
- **High Temperature ($T = 10$)**
- Using Boltzmann scaling:
  - $e^{10/10}, e^{7/10}, e^{12/10}, e^{1/10} \approx [2.718, 2.014, 3.320, 1.105]$
  - $\text{f} = \frac{[2.718,2.014,3.320,1.105]}{\sum([2.718,2.014,3.320,1.105])} \approx [0.31, 0.23, 0.38, 0.13]$
- **Low Temperature ($T = 1$)**
- Using Boltzmann scaling:
  - $e^{10/1}, e^{7/1}, e^{12/1}, e^{1/1} \approx [22026.47, 1096.63, 162754.79, 2.72]$
  - $\text{f} = \frac{[22026.47,1096.63,162754.79,2.72]}{\sum([22026.47,1096.63,162754.79,2.72])} \approx [0.12, 0.01, 0.87, \approx 0]$

CYPRUS INTERNATIONAL UNIVERSITY

Example of using Boltzmann Scaling Technique II

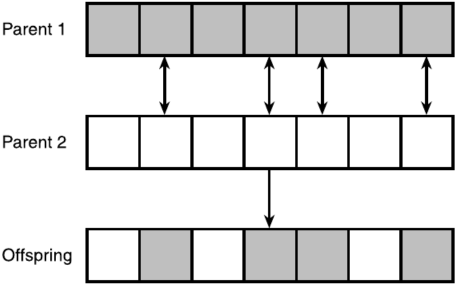
Chromosome	Fitness	Probability (T=10)	Probability (T=1)
1	10	0.31	0.12
2	7	0.23	0.01
3	12	0.38	0.87
4	1	0.13	0.00

- **High Temperature ($T = 10$)**:
  - Probabilities are more evenly distributed, allowing diverse selection.
  - Even low-fitness chromosomes (e.g., fitness 1) have a chance to be selected.
- **Low Temperature ($T = 1$)**:
  - Selection focuses almost entirely on the highest fitness chromosome (fitness 12).
  - Low-fitness chromosomes are nearly ignored.

Building a Better GA: Alternative Crossover Operators I

Multi-Point Crossover

- Why stop at just two crossover points? There's no need to limit the amount of crossover points you can have. Indeed, for some types of encoding, your genetic algorithm may perform better if you use multiple crossover points. The easiest way of achieving this is to move down the length of the parents, and for each position in the chromosome, randomly swap the genes based on your crossover rate.



Building a Better GA: Alternative Crossover Operators II

Algorithm:

```python
def crossover_multi(mum,dad,crossover_rate,chromo_length):
    # 1. Check whether to skip crossover or if parents are identical (then no crossover is needed)
    if random.random() > crossover_rate or mum == dad:
        return list(mum), list(dad)
    # 2. Determine a "swap rate" for this chromosome
    swap_rate = random.random() * chromo_length
    baby1, baby2 = [], []
    # 3. Iterate through each gene and decide whether to swap
    for m_gene, d_gene in zip(mum, dad):
        if random.random() < swap_rate:
            # Swap the genes
            baby1.append(d_gene)
            baby2.append(m_gene)
        else:
            # Copy the genes as is
            baby1.append(m_gene)
            baby2.append(d_gene)
    return baby1, baby2
```

Building a Better GA: Niching Method (Fitness Sharing) I

- Maintaining population diversity by grouping similar individuals is known as niching. Fitness sharing stands out as one of the most widely used niching techniques. This approach involves clustering individuals in a population based on the similarity of their genetic makeups. Subsequently, the fitness score of each individual was modified by distributing it among the members of its cluster. This process effectively penalizes similar individuals within a population, thereby preserving the diversity.

  Implementation Fitness Sharing:

  - Hamming distance between two binary strings is simply the number of positions at which the corresponding bits are different.
    e.g, the Hamming distance between **10101** and **11100** is 3 because three positions differ between the two strings.

$$d_H(x, y) = \sum_{i=1}^{n} |x_i - y_i|$$

Building a Better GA: Niching Method (Fitness Sharing) II

- Sharing function:

$$sh(d_H) = \begin{cases} 1 - \left(\frac{d_h}{\sigma_{\text{share}}}\right)^{\alpha} & \text{if } d_H < \sigma_{\text{share}} \\ 0 & \text{otherwise} \end{cases}$$

  - σ_{share} is the sharing radius in Hamming space (*i.e.*, the maximum allowed Hamming distance within which individuals are considered similar).
  - α controls the shape of the sharing function.
- Modified Fitness Calculation: The fitness of each individual is then scaled by the sum of the sharing functions applied to the distances between it and all other individuals:

$$f^{'}(i) = \frac{f(i)}{\sum_j sh(d_{ij})}$$

  - $f(i)$ is the original fintess of i-th individual, and d_{ij} is the distance between individuals i and j.

⋆ Fitness sharing is very effective at penalizing similarly constructed genomes and can be a terrific way of making sure your population remains diverse.

Exercise of Binary Niching Method I

Lets we assume:

1. Chromosome Length: 5 bits.
2. Population Size: 4 individuals.
3. Sharing Radius (σ_{share}): 2 in Hamming distance.
4. Alpha (α): 1, used in the sharing function.

Step 1: Initialize Population

$$Pop = \begin{bmatrix} 1 & 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 \end{bmatrix}$$

Step 2: Compute fitness score

Assume we have a fitness function and we evaluated each individual.

Exercise of Binary Niching Method II

$$Fit = \begin{bmatrix} 3 \\ 2 \\ 4 \\ 2 \end{bmatrix}$$

Step 3: Calculate Sharing Function and Adjust Fitness

1. **Calculate Hamming Distances between all pairs:**
   - Between 10101 and 11000: Hamming distance = 3
   - Between 10101 and 11101: Hamming distance = 1
   - Between 10101 and 00011: Hamming distance = 4
   - Between 11000 and 11101: Hamming distance = 2
   - Between 11000 and 00011: Hamming distance = 4
   - Between 11101 and 00011: Hamming distance = 4
2. **Apply Sharing Function: Using the sharing function:**
   - Use the sharing function:

$$sh(d_H) = \begin{cases} 1 - \left(\frac{d_h}{\sigma_{\text{share}}}\right)^{\alpha} & \text{if } d_H < \sigma_{\text{share}} \\ 0 & \text{otherwise} \end{cases}$$

Exercise of Binary Niching Method III

- With $\sigma_{share} = 2$ and $\alpha = 1$:
  - $sh(3) = 0$
  - $sh(1) = 1 - \frac{1}{2}$
  - $sh(4) = 0$
  - $sh(2) = 1 - \frac{2}{2}$

3. **Calculate Modified Fitness for Each Individual:**
   - For 10101:
     - Sum of sharing functions = $sh(0) + sh(3) + sh(1) + sh(4) = 1 + 0 + 0.5 + 0 = 1.5$
     - Modified fitness = $\frac{3}{1.5} = 2$
   - For 11000:
     - Sum of sharing functions = $sh(3) + sh(0) + sh(2) + sh(3) = 0 + 1 + 0 + 0 = 1$
     - Modified fitness = $\frac{2}{1} = 2$
   - For 11101:
     - Sum of sharing functions = $sh(1) + sh(2) + sh(0) + sh(4) = 0.5 + 0 + 1 + 0 = 1.5$
     - Modified fitness = $\frac{4}{1.5} = 2.67$

CYPRUS INTERNATIONAL UNIVERSITY

Exercise of Binary Niching Method IV

- For 00011:
  - Sum of sharing functions $= sh(4) + sh(3) + sh(4) + sh(0) = 0 + 0 + 0 + 1 = 1$
  - Modified fitness $= \frac{2}{1} = 2$

The modified fitness values are:

$$ModifiedFit = \begin{bmatrix} 2 \\ 2 \\ 2.67 \\ 2 \end{bmatrix}$$

4. **Selection Based on Modified Fitness**
   - Using roulette wheel selection based on the modified fitness values, the algorithm would select individuals with probabilities proportional to their modified fitness.