# Discrete Genetic Algorithm

## Introduction

This is a stochastic algorithm that was used in seeking the best solutions (paths) from a particular node to either base station. It uses the classical genetic algorithm approach with the following steps:

1. Population initialization
2. The selection process that encompasses fitness function evaluation
3. Crossover
4. Mutation

These steps are done iteratively until our termination criteria which is ‘maximum number of generations’ is reached. The specifications of the steps above are explained in the process flow below.

## Process Flow

Population Initialization: The algorithm starts with random initialization of the population which in this case is the various routes from the specified origin to a specified destination (base station). This is done in the function `dga\_init\_population()`where the number of routes generated is also specified as an argument. The function generated random values excluding the origin and destination regardless of whether they have end connectivity and at the end prepending and appending the origin and destination respectively. The validity of the paths will be checked in subsequent stages. A list of randomly generated routes is returned form the function.

Fitness Function Evaluation: The next stage is calculating the end-to-end fitness function which incorporates the end-to-end bandwidth and total latency using the cost function defined for this project. This is done in the ` bandwidth\_latency\_calc()` function and while doing all this, it updates the global variables ‘solutions’ and ‘dga\_data’ which carry map visualization data and best solutions (for the solutions file) data. The function returns the population (routes) with their corresponding end-to-end fitness function values.

Filtering out valid paths: The next stage is filtering out valid paths from the ones that are created. Since the algorithm has no specified formulae in the way the path is created, many invalid paths have a high chance of being created. Therefore, there is need to exclude them so that the subsequent stages run on quality routes. This process is run in the `get\_valid\_paths()` function. The function eliminates any chromosome with an end-to-end fitness value of 0 as this means there is no end-to-end connectivity. It is at this stage that we determine if there is no valid path completely to a particular station that has been generated. If that is the case, the algorithm reruns from scratch and seeks a path to the alternative base station using the processes described above.

Fitness Selection Process: Using the valid paths determined by the previous stage, a selection process is carried out using the tournament selection method. Here the selection is based on the solutions that have the highest fitness function values since our main aim is to maximize the values. This is carried out in the `tournament\_fitness\_selection()` function which has a set number of maximum possible winners in the function parameters. The result of this function is quality solutions which will be improved in the subsequent stages. If the quality solution left is just 1, the solution is returned without proceeding further.

Crossover: Just before this, the process of preparing the chromosomes for crossover is done. This entails pairing best solutions in 2s since the crossover process requires pairs of chromosomes. This is alongside the type of crossover (single-point/ two-point/ uniform) as arguments to the function `crossover()` where the process happens. The result of the process is 2 children with mixed genes (nodes/sensors in the routes) that could potentially be better solutions than their parents.

Mutation: This process seeks to introduce new information to the gene. In the shortest route problem that we seek to solve, mutation is done in the form of swapping genes (sensors) for 2 reasons:

1. There is a limited number of valid sensors that can be introduced to the solution
2. We aim at having sensors with valid end-to-end connectivity values.

The number of swaps made is determined by the mutation rate defined in the function `mutate()` where this entire process runs. The nodes that are to be swapped exclude the origin and destination/base station nodes which are the fixed.

The above processes are consolidated into the function `dga\_run()` which takes in all the requires parameters. The parameters are:

1. origin: the origin from which you want to traverse the space.
2. destination: the destination node which you want to get to.
3. population length: the length of the population for the first generation.
4. generations (optional): the maximum number of generations that you would

want to you do work with (termination criteria). Default is set to 10.

1. mutation rate (optional): The percentage of the population that will

be mutated. Default is set to (0.1).

1. crossover type (optional): The type of crossover that will be implemented.

The default is set to 'single-point'.

1. selection limit (optional): The number of winners that should be selected

at the minimum during the fitness selection stage. Default is set to 20.

The algorithm is tested for each base station. The population generated point to a certain base station and if a valid solution to that base station is found, the alternate base station is tried. This is done using a try-except statement in the wrapping function `discrete\_genetic\_algorithm()` which is the main function of this algorithm and is called in the main function file. The result of this function is the plotting data that has the best and possible paths alongside the solutions variable that has the data of the best path that will be recorded in the solutions file. The above processes are illustrated in the flow chart below.

A screenshot of a cell phone

Description automatically generated

Figure : DGA Process Flow

The algorithm performs well but due to its mechanism of searching for randomly generating paths, the population initialization stage, a high number of routes need to be generated to get some valid solutions by the time we are filtering out the invalid solutions. A satisfactory number of chromosomes required for standard performance (fast time and relatively good solutions) is 100,000 nodes. This is a process that could be improved by used of a graph indicating paths with connectivity. This will increase the speed of the algorithm and remove the need to initializing a high number of chromosomes. This is an improvement that we intend to include in future iterations of the algorithm for time and resource efficiency.