Imperial College London

ACSE-4: Project 3 - Optimization using Genetic Algorithms

Team Name: Copper

Richard Boyne, Hameed Khandahari, Ye Liu, Yuxuan Liu, Gareth Lomax, Wade Song, Yujie Zhou

March 22, 2019

Abstract

A program was written using C++ to obtain the optimum configuration of separation units in a mineral processing facility. This was achieved using a genetic algorithm as using a brute force solution would be too costly. The principles can be applied to obtain this optimum solution can be applied to many other problems. Further work included the investigating the convergence behaviour of the genetic algorithm and the effects of changing the model parameters on the resulting profit generated by the facility.

1 Problem Specification

Throughout this section, the example given in the project notes will be used, this is an optimised circuit with 5 units and can be seen in Figure 1.

The numbered boxes represent the ‘units’. These are the separating units used in the mineral processing facility. The overall configuration of the units shall be referred to as the ‘circuit’. In the circuit shown in Figure 1, there are 5 units, numbered from 0 to 4. There are two outputs from each unit; the concentrate stream (represented by the blue arrow) which contains 20% of the concentrate and 5% of the waste contained by the unit, and the tailings stream (represented by the red arrow) which contains the rest, i.e. 80% of the concentrate and 95% of the waste. The two outputs of any unit are connected to two other distinct units where they become either their one of their input tailings or concentrate streams. Note that whilst there is a requirement that each unit *must* have two output streams, there is no limit for the number of input streams (as long as the validation tests described in Section 2.3 are passed). Eventually, the material will reach either the overall concentrate stream or the overall tailings stream of the circuit. These can be represented as units with no further output. The fitness value of the circuit, a measure of its performance, is obtained by calculating the profit generated by the circuit. This is calculated using the contents of the overall concentrate stream only (the contents of the tailings stream can be discarded). There is a profit associated with the amount of concentrate, and a penalty associated with the amount of waste, in the concentrate stream. These are £100 /kg and £500/kg however these are parameters which can be changed. The aim of this project is to obtain the optimum configuration of connections in a system of 10 units to maximize the profit generated. This was done using a genetic algorithm described in the next section.

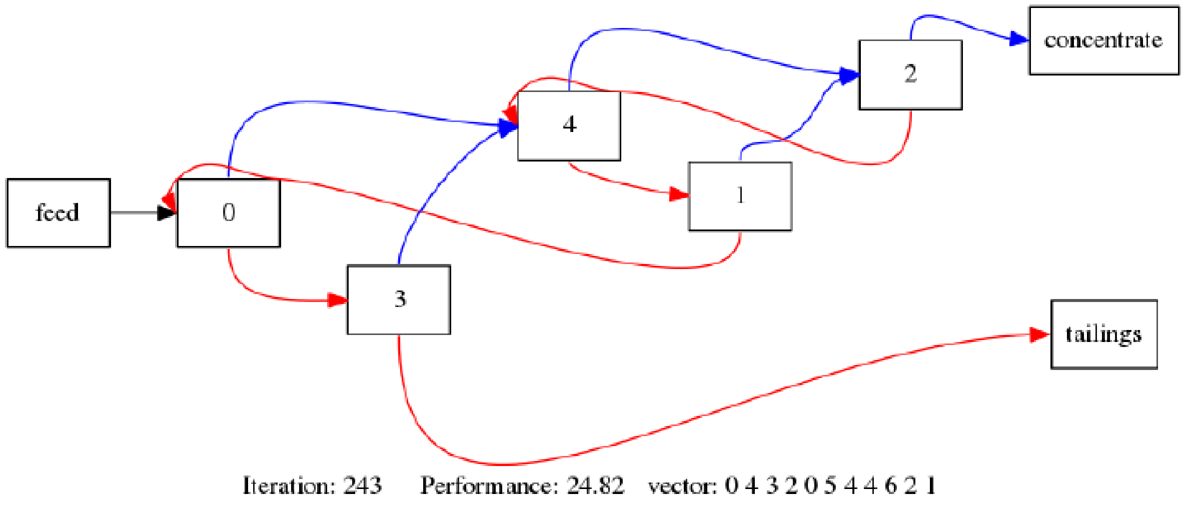

Figure 1: Optimized circuit with a 5 units. Adapted from Dr Neethling’s introductory presentation slides. [REF]

2 Solution Algorithms

Three teams were set up, each assigned to one of the key tasks. These tasks are: evaluating the circuit to obtain the fitness value, checking the validity of the circuit, and optimising the connections between the units in order to find the optimum configuration which will maximize the profit generated. Three teams, with two members each, were initially created to work on one of the tasks. Richard Boyne was involved with each of the teams to ensure coordination between all groups.

2.1 Circuit Simulation

The team members responsible for this section were Hameed Khandahari and Gareth Lomax

This section addresses the parts of the program where the fitness value of a given circuit configuration is evaluated. The evaluation of the fitness value is a crucial part of any genetic algorithm, as it is an indication of how well successful a circuit is performing and can therefore be used by the genetic algorithm to generate more successful offspring in the next generation. In this program, the profit generated it taken as a measure of a circuits 'fitness'.

The profit generated is a function of the amount of concentrate and the amount of waste output to the concentrate stream. There is a £100 profit and a £500 penalty associated with every kg of material in the output stream. In this model, we ignore any output to the tailings stream as we assume it is comprised mainly of waste, and that it can be disposed of at no cost. Including a disposal cost would be a useful extension to the model, as waste disposal costs of chemical waste (especially hazardous waste) can be high.

A mass-balance of the system is achieved using the successive substitution algorithm described in the project description notes. Briefly, for any given configuration, this algorithm first naively sets the flow rate of the concentrate and tailings stream to that of the feed rate, then by re-evaluating the input concentrate and tailings streams of each unit iteratively, the resulting, overall, concentrate and tailings stream of the circuit itself can be calculated.

The Circuit is implemented using two classes *Circuit* and *Cunit*.*Cunit* acts as a memory storage, and comprises of the contents of each cell, and each cell's feed buffer. *Circuit* contains a list of *Cunits*, which are subsequently iterated over to extract results. Both *Cunit* and *Circuits* have methods to reset their contents with a new genetic code, which allows us to reset parents with the next generations genetic information, allowing us to re-use existing objects without the overhead of initialising more parents at each iteration.

In our specific implementation, a list of units is initialized with the feed flow rate and the successive substitution is performed until the system is converged. The converge criterion checks the difference between contents of each unit, before and after the substitution, and if this difference is below a tolerance value, the system is assumed to be converged. The profit can then be calculated from this converged system. Initializing the nodes using a combination of the parents converged node feed rates was explored. This was expected to speed up the convergence of the mass balance as the two inherited components of the circuits would both have their equilibrium flow rates at the start of the convergence. This however was not the case; although it outperformed compared to an initially empty circuit, it under performed when compared to an uniformly pre-filled circuit. We anticipate this is due to the nature of the genetic algorithms combination of the two parent circuits. When two genetic codes are spliced together "genes" are grouped together in the genetic code. I.E The genetic code is two contiguous sections of the DNA inherited from its parents. The effects on the structure of the circuit however are not guaranteed to be similar to either parents; the node connections are mixed rather than two different sections being joined together. As a result using the parents converged feed rates to initialize a child circuit is not as beneficial as could be thought. It is however beneficial in the case of small changes (i.e mutations) however these do not occur at a great enough rate to make it a performative change overall when compared to the increased performance of using the feed rates to pre-fill the network.

Several tests were written to ensure the program was performing as expected. These include checking the optimum fitness value of the two circuits included in Prof Neethling's presentation as they had his calculated solution. A further test was written to check that equivalent, but symbolically different, circuits will produce the same fitness values. An example of this is shown in Section 3: Optimum Circuit Configuration. Finally, a test was written to check that a circuit that is known to diverge, will correctly be identified as such, in case such a circuit was accidentally permitted by the validation checks.

2.2 Genetic Algorithm

The team members responsible for this section were Wade Song and Yujie Zhou

The genetic algorithm is based on the theory of evolution by natural selection. Whereas in Darwin's theory, the fitness value in question was a measure of how well an organism is suited to its environment, in our model, the fitness value is the amount of profit generated for the mineral processing company.

The 'genetic code' is analogous to DNA and encodes the circuit configuration in sets of base pairs which describe the connections between the cell units. As all the cells are assumed to be identical, it is the connections between the units which determines how the material traverses the circuit and therefore ultimately determines the profit generated by the circuit.

The circuit configuration is allowed to evolve from generation to generation, producing ever more effective circuits. Initially, a generation of valid circuits are randomly generated. The number of circuits in this generation is a tune-able parameter, the effect of which is investigated later in the report.

The evolution between two successive generations can happen through two routes, mutation or a crossover and the overall process is described in Figure 2

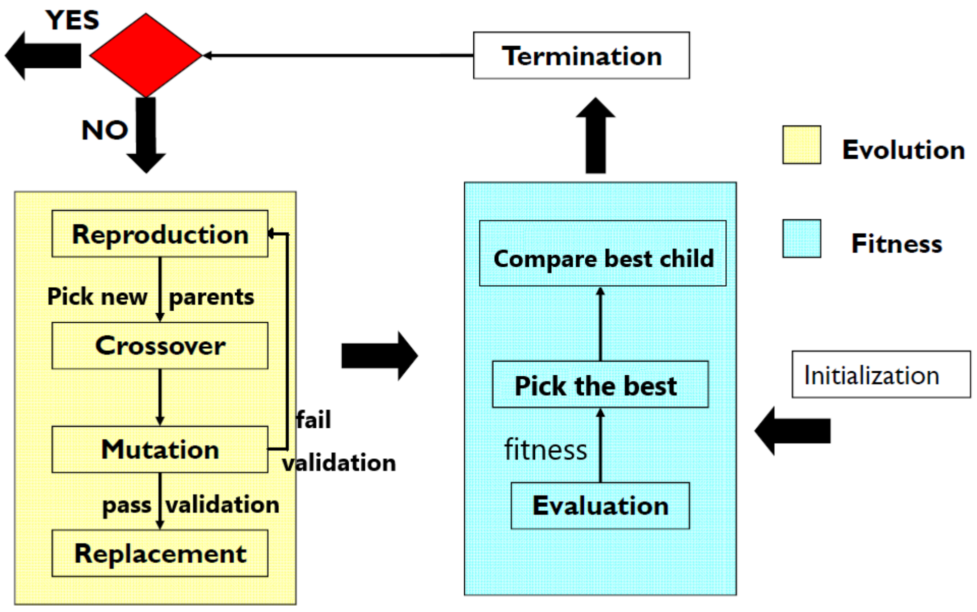

Figure 2: Genetic Algorithm Process. Diagram generated by Wade Song

This was developed initially without any calculation of fitness value, as a proxy for fitness value was being used.

After initializing the set of circuits, the fitness value of each one is calculated. The child with the highest initial fitness value is found, and allowed to survive to the next generation, a variant of the genetic algorithm known as elitist selection. The algorithm then produces children from the previous generation, evaluates their fitness value and then produces a successive generation of children. This process is iterated until there has been no improvement in the best child's fitness value (the highest fitness) for a specific tolerance of rounds. The round tolerance value increases with the number of cells in a circuit. A tolerance of 1000 rounds works well for a 10 unit circuit, whereas a tolerance of 500 rounds is acceptable for a 5 unit circuit. Before this condition is satisfied, the algorithm will carry the circuits to the next iteration for reproduction.

The reproduction is the process of picking new parents. In this process, the best circuit of the last iteration survives and participates in the breeding (elitist selection). All the circuits can have the chance to generate the new circuits, and each couple can produce their offspring more than once while they cannot self reproduce. Roulette Wheel Selection is used to perform parent selection : Find the minimum fitness value, minus this minimum value to everyone to make the temporary fitness value of each circuit. Sum up the circuit value of everyone. Using everyone's fitness value over the sum of fitness to get a relative fitness for everyone. Then, obtain the cumulative fitness which is the sum of relative fitness value from the first one to one itself. Using a random number in 0 to 1 and find which cumulative fitness interval it falls in and pick the corresponding circuit as a parent. Finding two different parents for doing crossover.

Next, each pair of picked parents do single-point crossover. It should be noted that not every couple can generate their own off springs because a user-defined parameter determines the probability of crossover. This parameter is called genetic cross over chance in the ”config.csv”.When permitted by the crossover probability, a index on the parents' circuit vectors (corresponding to the location of bit on chromosomes)is picked randomly and the tails of this location are swapped between the couple. This kind of crossover produces two off springs from the parents. After crossover, some invalid circuits may occur but they move on to next step of mutation because mutation has chance to make them valid.

During mutation, each off spring has probability to mutate, and the bits at random positions can be altered to random values. The genetic mutation rate is the probability of mutation of each bit (cell) in the each circuit chromosome, and this parameter is also user-defined through ”config.csv”. The off springs after mutation are sent to check their validation, the result of which directly determines whether they will be killed. If validation test fails as well as the population of new iteration is not filled fully, the reproduction will be called to generate new off springs to fill the void. Once the population with designated size is full of new off springs which all have passed the validation test, the replacement will be executed to displace parents with children.

The main structure of the genetic algorithm is repeated iteration until hit the termination condition. One significant point is that the parents selection is controlled by fitness value, which is the core of the algorithm : generate high-quality solutions.

2.3 Validity Checking

The team members responsible for this section were Ye Liu and Yuxuan Liu

There are several tests which any given configuration *must* pass in order for a mass-balance to be possible, and hence for the circuit to be valid for use in the genetic algorithm. These include checking that the circuit is traversable.

The validation code is comprised of two validity checking functions, a simple validation function and a more advanced one to check whether the circuit is connected.

The simple validation function is used to do the initial test. The function prevents self-cycles, ensures every cell is mentioned at least once in the genetic code, and that all inputs and outputs of a cell exist. No self-recycle means the number of the node should not be equal to the number of either of the two product streams. The number of the two product streams be the same. If those situations happen, the function will return false and we will generate the new random circuit vector.

If the circuit passes the simple validation test, it will be tested against the validation function to check whether the circuit is connected. This checks if every unit can be accessible from the feed and whether a route exists forward to both outlet streams.

We implement this function by traversing all nodes in the directed graph using recursion. When performing the traversal, all traversed nodes will be marked, nodes that are checked without marks will return false.

When validating the forward path, the recursion begins with a selected node checking the concentrate stream first. If the concentrate number is smaller than the total node numbers, the index changes to the concentrate number and recursive until the index equals to the concentrate number.The nodes backwards test is quite similar, if either the exit pipes were not found the test fails.

Besides, all the stream path that leads to the output can be easily marked, which is good for the further calculation and simulation. The strength of implementing two validation functions is that the time consumption can be significantly decrease in this part because we only do two steps checking instead of the whole four steps one, which, to some extent, optimizes the structure of program and improves the reliability of checking process.

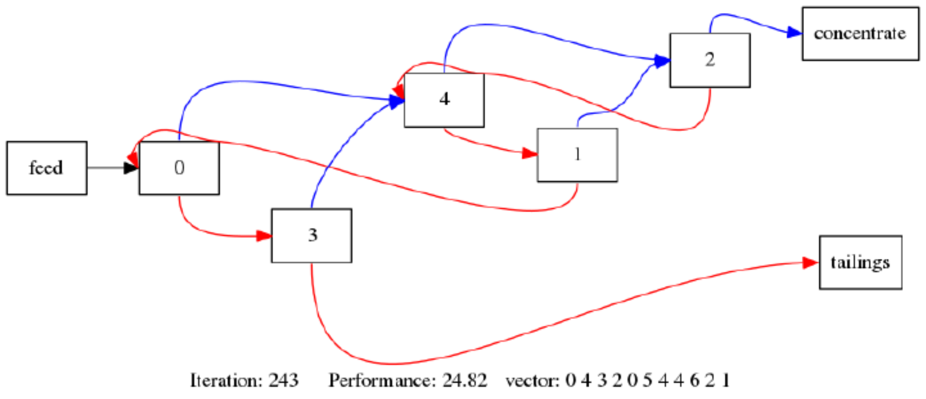

Figure 3: Optimized circuit with a 5 units.

3 Optimum Circuit Configuration

The optimum circuit configuration for a system with 10 units was found to be

Feed	0		1		2		3		4		5		6		7		8		9	
1	4	2	3	5	8	11	9	4	9	8	3	7	3	0	3	6	9	1	10	3

This representation is based on a directional adjacency list, where the first row contains the `id` numbers of the units within the circuit, and the second row contains information about where the concentrate and tailing streams of each unit is directed.

A visual representation of this optimum circuit can be generated using the `graphviz` package in Python. The output can be seen in Figure 4. The fitness value, i.e. the profit, generated for this system was found to be 165.753. This value was consistently obtained across 10 measurements. This system has been evaluated with the parameters given in the project notes and it is the primary result of this investigation

It can be shown that the same configuration can be written in a number of equivalent, but symbolically different ways since the numbers assigned to each unit are arbitrary. For example, the following two circuit configuration is equivalent to the one given above, but symbolically different.

Feed	0		1		2		3		4		5		6		7		8		9	
3	5	3	5	0	5	1	2	6	2	8	10	2	2	7	2	4	1	9	0	11

This equivalency was used to test the correctness of the parts of the code which evaluate the fitness and it is included in the `tests\tests_evaluate.cpp` file as Test 3.

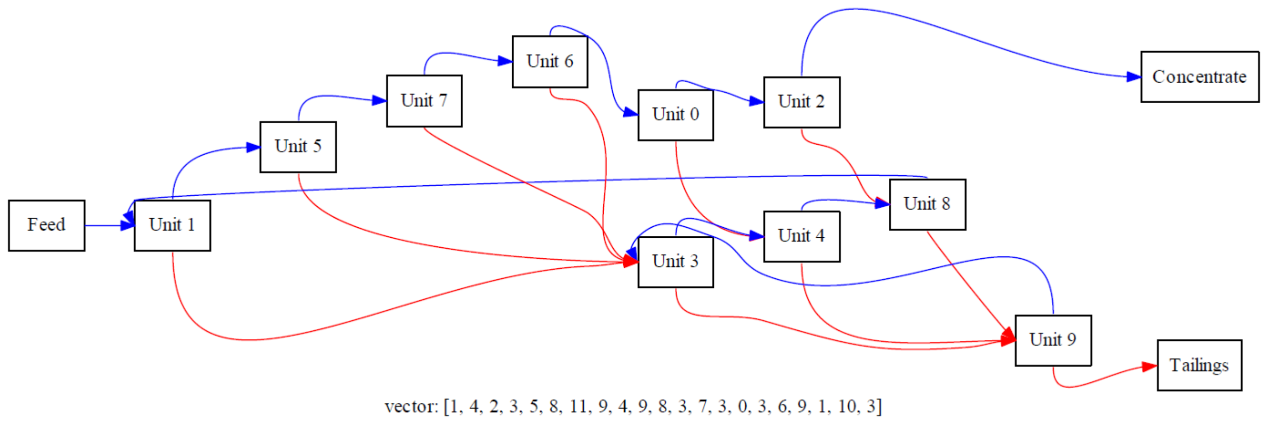

Figure 4: The optimum solution obtained for a system size of 10

Additionally, data was also obtained for system sizes of 5 and 15. The corresponding optimum fitness value was found to be 24.8163 for a system size of 5 units and 449.616 for a system size of 15 units. Note that the value obtained for the system size of 15 has not yet been heavily tested hence the authors are present it with caution.

4 Convergence Behaviour of Genetic Algorithm

The team members responsible for this section were Wade Song and Ye Liu

The configuration of doing convergence test is running with 5 cells, with the maximum iteration of 20000, and genetic crossover chance at 0.9, profit at 100, cost at 500.

Here we assume the optimal fitness to be 24.8196, with a tolerance of 0.01, we count the iteration runs to reach this reference value and the results show in the following pictures.

In every distribution plot, most data points are distribute between 0% to 10% compare with the extreme data point(though the last 10 most extreme data has not been shown on plot). And the distribution behavior really looks like poisson distribution. But rigorously speaking, we can only treat that as empirical distribution. So that we using confidence interval to interpret the performance of different parameter configuration. The data point as 95% tells if the iteration ends up at 1585 rounds, there's 95% of possibility that the optimal result is already obtained. Same to 99%. The box plot also give us some insight on the distribution behavior.

The following figures show the frequency of iteration that converge. Running at two population and two mutation rate level. In this case, we can compare the distribution of iteration time to convergence under fixed population and two different mutation rate, and under fixed mutation rate and two different population size.

Note: Axis x represents for the iteration runs and axis y represents for frequency.

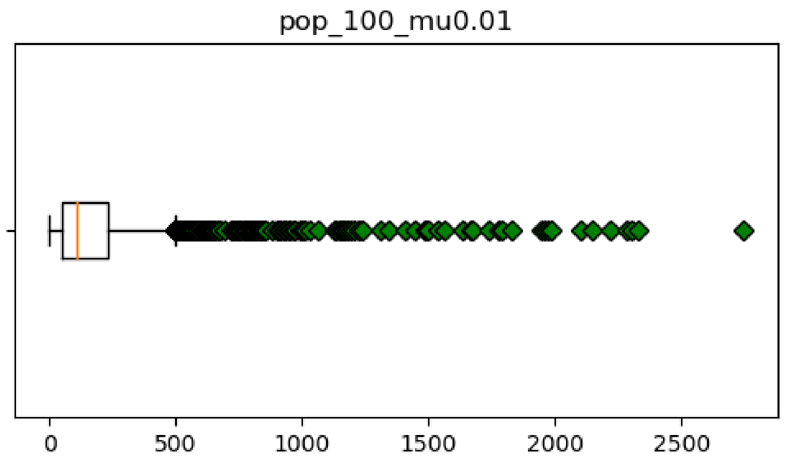

Figure 5: Box plot under population of 100 and mutation rate at 1 %

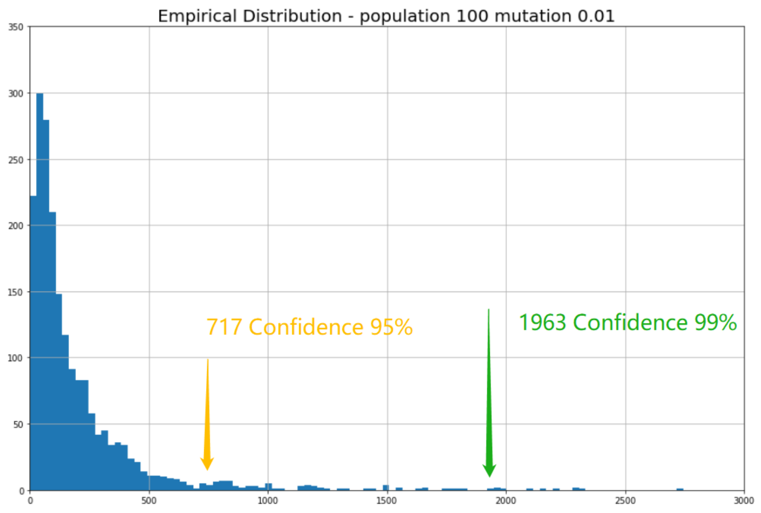

Figure 6: Empirical distribution under population of 100 and mutation rate at 1 %

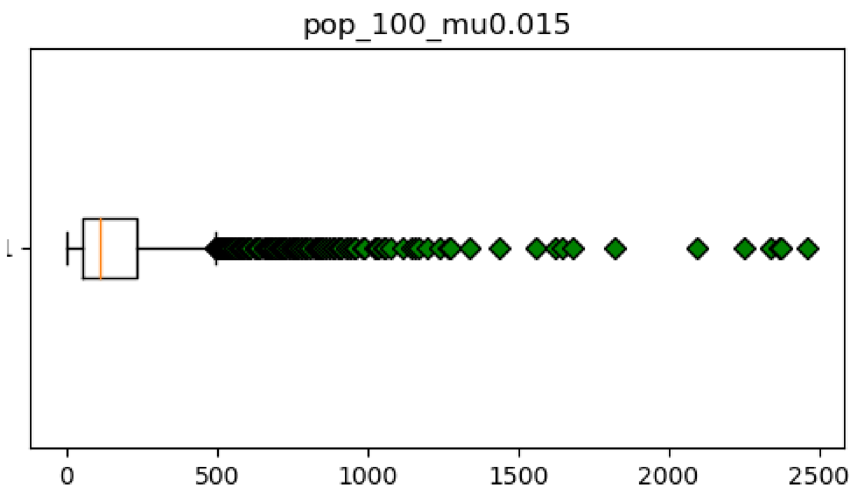

Figure 7: Box plot under population of 100 and mutation rate at 1 %

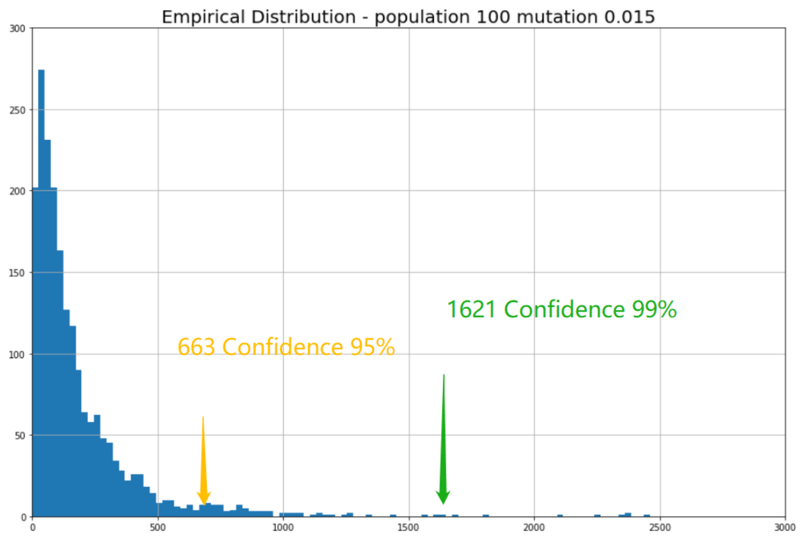

Figure 8: Empirical distribution under population of 100 and mutation rate at 1.5 %

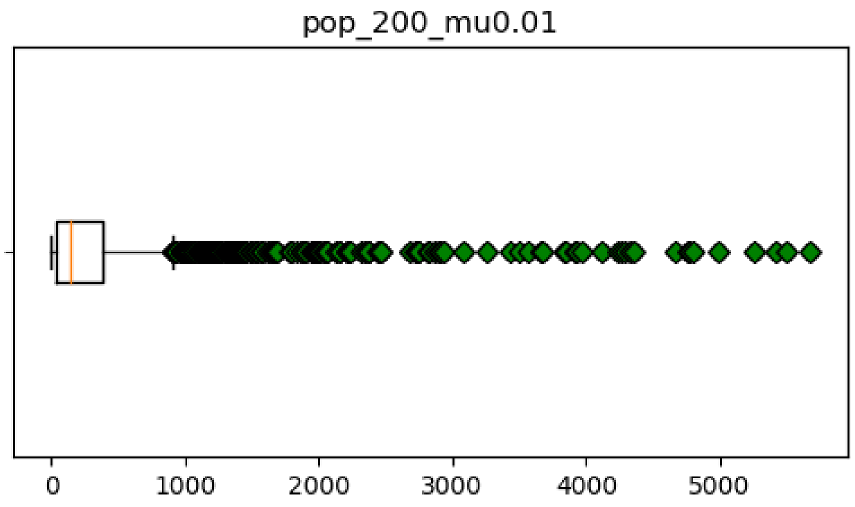

Figure 9: Box plot under population of 100 and mutation rate at 1 %

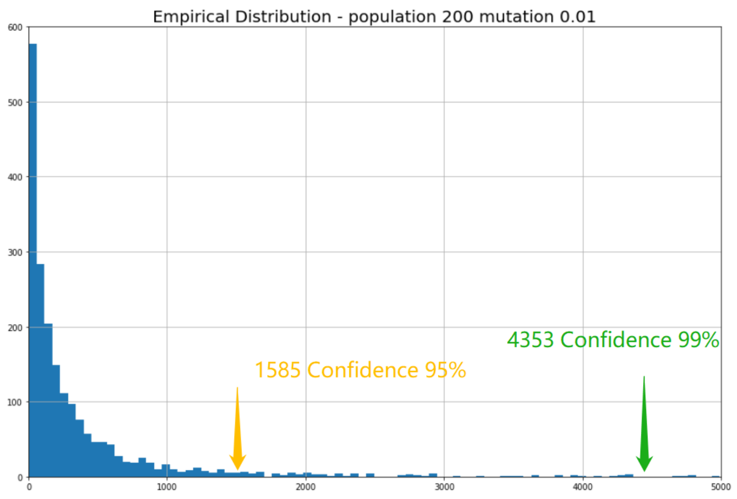

Figure 10: Empirical distribution under population of 200 and mutation rate at 1 %

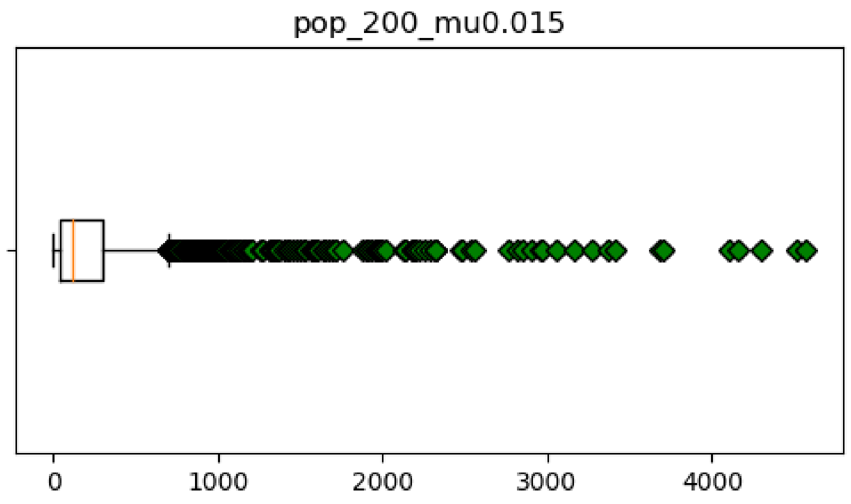

Figure 11: Box plot under population of 100 and mutation rate at 1 %

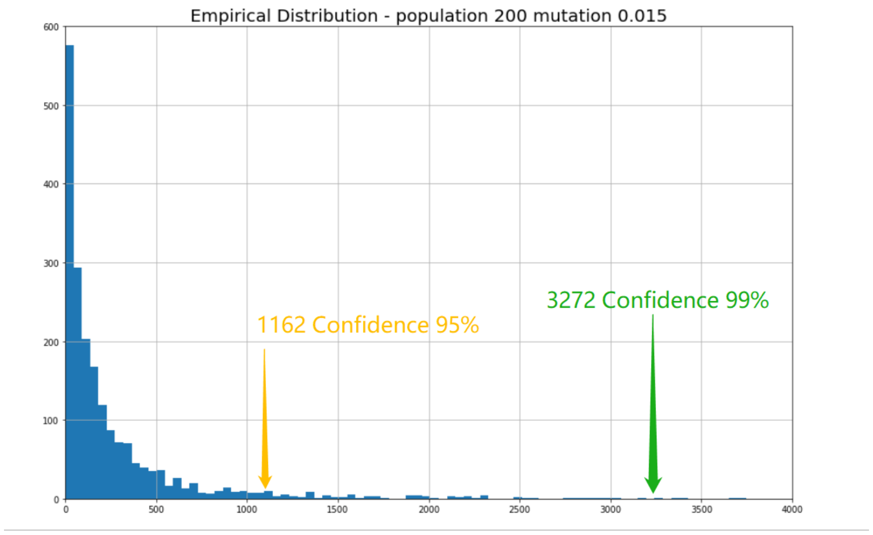

Figure 12: Empirical distribution under population of 200 and mutation rate at 1.5 %

Compare figure 5 with figure 7, fixed population with 100 and vary the mutation rate from 0.01 to 0.015, the iteration times take to converge at both 95% confidence interval and 99% confidence interval decrease. The consistent result can be obtained from fixing population size at 200 with different mutation rate, referring to figure 9 and 11.

The decrease of the confidence interval basically shows the improvement of the performance in the evolution process.When the mutation rate increased, the production rate of better gene of gene will increase accordingly. In the meantime, the production rate of worse gene of gene will increase but in our genetic algorithm they are not tend to generate their child. We call a part of a circuit 'gene' here.

Also, compare figure 9 with figure 9, fixed mutation rate of 0.01 and vary the population from 100 to 200, the iteration times take to converge at both 95% confidence interval and 99% confidence interval increase. The consistent result can be obtained from fixed mutation rate of 0.015 with different population size, referring to figure 7 and 11 as well.

The increase of the population size basically the worse performance in the evolution process.When the population size increased, the possibility of better genes meet with each other would decrease.

5 Changing Model Parameters

The program has been constructed in such a form that it takes a `.csv` file as its input. This file contains all of the parameters required to configure the model. These include, the number of units in a circuit, the number of circuits in a given generation, the maximum number of iterations of the genetic algorithm, the genetic cross-over chance, the genetic mutation rate, the profit generated per kg of gormanium in the concentrate stream, the penalty charged per kg of waste in the in the concentrate stream, the genetic algorithm steady state tolerance, the circuit evaluation algorithm steady state tolerance.

These configuration `.csv` files were generated using a Python script, to vary parameters in a given range, and a BASH script was written to evaluate run the genetic algorithm for each of these configurations, producing an output `.csv` file for each configuration containing information about the optimum circuit generated.

Another Python script allowed for these files to processed and plots to be produced.

5.1 Varying penalty

Penalty configuration was varied, giving a two-dimensional parameter space for which fitness values could be calculated. Figure showed a trade-off between the mineral recovery and its purity: at high recovery the purity will be low, and vice versa. When the money spent on per kilogram waste was decreased, the recovery rate increased, and purity decreased synchronously. It is obvious that the recovery rate almost reached 100 when there is no penalty, but meanwhile the purity was very low. So in industrial community, the company can set what their strategy is. Is it focusing more on recovery or purity, and the corresponding penalty can be calculated at the same time, which determines how they are going to adjust the future strategy as well.

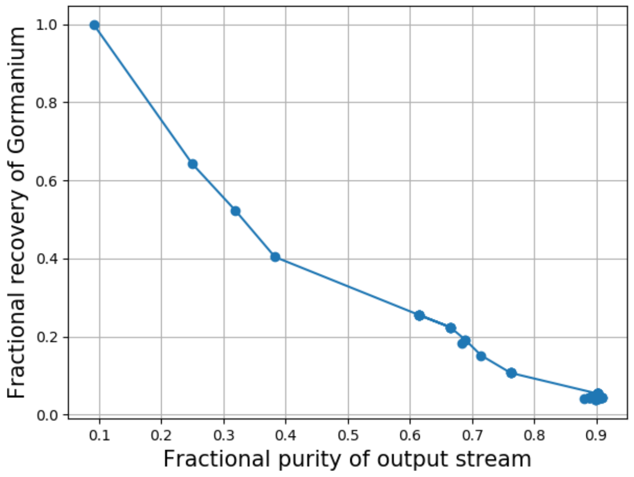

Figure 13: Purity - Recovery

6 Parallelisation

A parallel implementation of the previous genetic algorithm code was produced using the MPI structure. Two approaches of parellisation were considered; domain decomposition, and master-slave. Following the production of the child codes at each iteration step the children must be verified as valid. This process is costly as it involves full recursive traversal of the produced children. Using peer-to-peer communication allows this validation step to be distributed across the available nodes, avoiding the bottleneck introduced by performing all validations in serial on the master node. Peer to peer communication allows the costly step of initialising all the parents and children to be distributed across the system, again avoiding bottlenecks.

Two buffer arrays were used to gather the discontinuous local genetic codes and fitness values from the local dynamically allocated children. Using these arrays `MPI_Allgatherv()` was used to allow efficient simultaneous communication across all nodes, and to consolidate the children produced across all the nodes into a local genetic code. While this is not strictly full domain decomposition, the cost of storing all genetic codes locally is very small, and a consolidated global list of codes is required to both allow the best parent to continue into the next generation via elitist selection, and produce a probability distribution to seed the next generation.

A more elegant solution to this issue would have been to produce a new MPI datatype to send the discontinuous class information between nodes. This method requires the discontinuous data to be statically allocated. Our implementation is designed to be easily scale-able, controlled by a local `config.csv` file for ease of use. As a result, the circuit data is allocated dynamically and does not lend itself to being sent using `MPI_datatypes`. This could be achieved using a work around as a possible improvement in the future as a possible extension to the work shown in this report.

Bench marking was performed to investigate the effect of the parallel architecture on the performance of the code. In order to observe the increase in performance a relatively costly (15 cell) operation was used. A large genetic population (320 codes) was used. As may be seen in fig 14 there is a clear increase in performance with increasing processor count. We note that there is a large initial performance increase, which begins to level off, and sharply reduced once we exceed 12 processors. This is as a result of the architecture of the CX1 allocation the program was run on, and the use of run time as out bench marking variable. Once the processor count exceeds 12, the parallelisation occurs across multiple nodes, and incurs the increased time penalty associated with node to node communication.

The performance increase associated with the parellisation allowed exploration for higher cell count networks. We have tentative results for a 25 cell network, however further repeat simulation is needed to ensure that this has converged to a constant value.

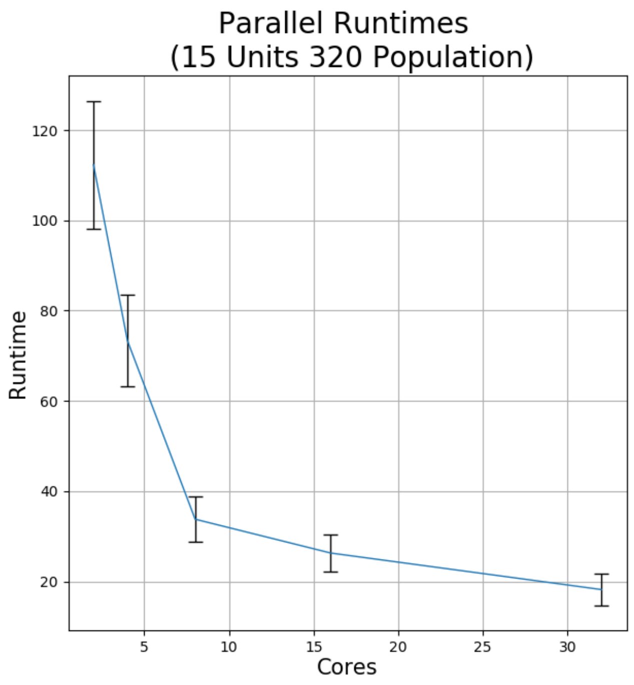

Figure 14: Scaling behaviour of parallelisation behaviour