ACSE 4 - Project 3 - IRON

Deborah Pelacani Cruz, Duncan Hunter,
Hitesh Kumar, Jia Ye Mao,
Nitchakul Pipitvej, Zongpeng Chen

March 22, 2019

1 Project architecture

1.1 Code-base design choices

As we split ourselves into 3 teams, our code also had distinct parts. Our smaller classes were kept separate in CStream (stream class) and CUnit (unit class), with the CCircuit class declared and defined along with the main functions of the Circuit modelling and Validity checking teams. This kept the code not directly related to the overall flow of the genetic algorithm separate from the functions used purely to simulate genetic changes, allowing not only for the teams to work quite independently but also to make the files themselves clearer in terms of purpose and documentation.

After we had a working serial model, we decided to parallelise the entire code base with MPI, following a highly independent, low communications model (we've used OpenMP too within the functions themselves, but we discuss that later). Our idea was to minimise the communication between processes by letting each one run its own, completely independent simulation, with a different random starting set of individual circuits. This way, we make the best use of our multiple processes by parallelising almost all of our code, leaving very little to be serial, hence allowing for the best possible theoretical speedup[1].

Aside from speed improvements, we also see that if we ensure each process has a different set of starting individuals, and has its own independent random seed, then we can ensure that as the number of processes tends to infinity, the probability of our model being trapped in a non-global minimum (or maximum really) tends to 0. This is essentially a particle-swarm based optimisation method[2].

1.2 Circuit class structure

We decided to go for a object-centric mindset to facilitate flows and connections where needed and to abstract it all away when we only need to see a circuit as an object with a fitness score. The smallest particle or "unit" in our code is the CUnit, which represents one separation tank, with its own ID, and feed and output streams (with Streams themselves being simple value/waste pair objects with some operator overloads). Because we can also store the destinations of the output streams, it becomes very easy to iteratively build a circuit from these.

The Circuit itself is also a class, containing information about all of its units, as well as where its feed it and what its overall production stream contains. We decided to represent the circuit both as an integer array and as an array of units, making it much easier for all other functions to take in. Once given a circuit object, the validity checking functions could use the integer vector to quickly and easily do their checks, whilst the Genetic evolution simulation could treat it just as an object. The circuit assessment functions could easily extract the unit array and easily run its own simulation to calculate fitness. At the cost of storage, the object based representations allow incredible ease of use, compatibility and sustainability for future developers.

2 Algorithm description

2.1 Simulating genetic improvement

Before the task, we have two Ccircuit arrays which are defined as our 'parents' and 'offsprings'. We initialize two Ccircuit objects with the same size as a parent object in 'parents' to store the potential offsprings. After that, the function createOffsprings is used to generate two potential offsprings each time. To pair the parents, we need to feed the pairParent function with the fitness result of the parents in this iteration. With implementing of 'Roulette Wheel Selection' method, the parents with higher fitness values will have a higher percentage to be selected. We use a loop calculating the cumsum of fitness each time of selection. When the random number generated is higher than a cumsum calculating since the first fitness value which is always the best one from the last generation, the loop will break and give us the index of the parents. After making sure that the parents are not the same (genetically maybe) with a if statement, the createOffsprings function will then begin to create offsprings using these parents.

The two offsprings will be same as the parents at the first step. Next, Both of them will have 90% chance to crossover since an random index between 1 and (the length of the Ccircuit object - 1). Then, each 'gene' in each offspring will has 1% chance to mutate. The function will create two potential offsprings at last each time. After that, the potential offsprings will be added to the 'offsprings' array if they pass the checkValidity test. The two spaces storing them will get cleaned each time for doing another round of selection which saves a lot of memory. This process will keep going until the 'offsprings' array get the same number of valid objects as the parents before the iteration. In the end, we will swap the 'parents' with the 'offsprings' for beginning another round of genetic improvement. We use a parameter called best_count_lim to control the convergence. We set it as 1000 for this exact example, which means the program will automatically stop generating new 'offsprings' when the best fitness keeps unchanging for 1000 generations. This fitness value and the corresponding circuit will be considered as the best at last.

2.2 Validating circuits

Since the genes generated from the genetic algorithm are created from mutation and crossover, there is no way to confirm if the created circuit is valid or not. There are a number of conditions that need to be met before the circuit can be considered as valid, we will discuss the method we use to check each condition one at a time.

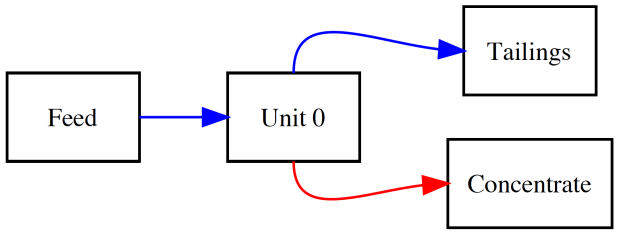
The checkValidity function takes in the CCircuit object. The object contains the information both in the form of integer array which stores the genetic code, and an array of CUnit objects as well. While both of these objects store the same information, we can make use of each data structure to speed up the check validity process.

To check that there is no unit that sends to itself and does not send both its output to the same unit, the genetic code array is utilized. The array was looped through and check its adjacent members. This can be done quite fast as it only involves one integer array.

As for validating that all unit is accessible from the feed, we use the recursive function "markUnits" to validates. Using the property of CUnit object, the flag "mark" was used to check if a unit has been visited yet or not. The algorithm will traverse from the feed index along the concentrated path and the waste path until it reaches an end or reaches a unit that was marked as visited. As it traverses, it will mark the unit as visited to prevent looping on the same path. It will also keep track of which exit it has reached. After the circuit traversal was done, it will check if both exits were reached, and check if all the units were visited. At this point, while both exits can be reached, we have no way to confirm that all unit can reach both exits. To check for each unit, we make use of the "markUnits" function again, but will only check if both exits can be reached afterwards.

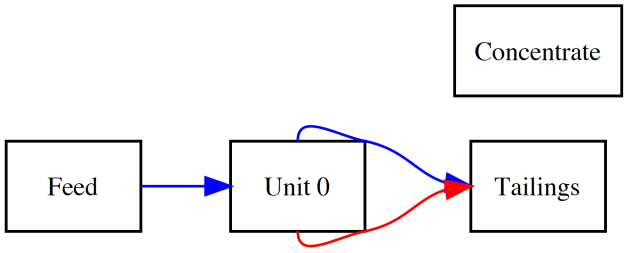
2.2.1 Testing

We tested this function on a number of small tests. The first two test were taken from the provided test file, which will test the simplest case of valid and invalid circuit. The following are the genetic codes tested:



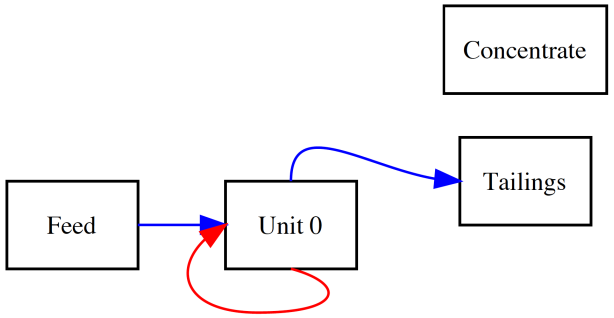
vector: [0, 1, 2]

with one unit that reaches both exits (1 and 2 as concentrated and waste exit).



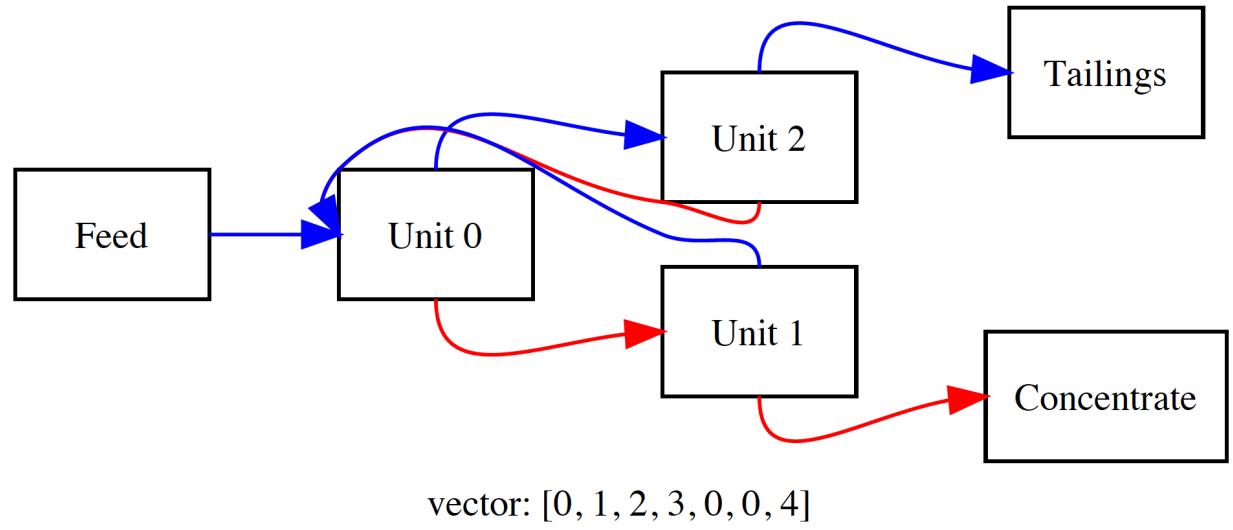
vector: [0, 2, 2]

with one unit but only reaches one exit.

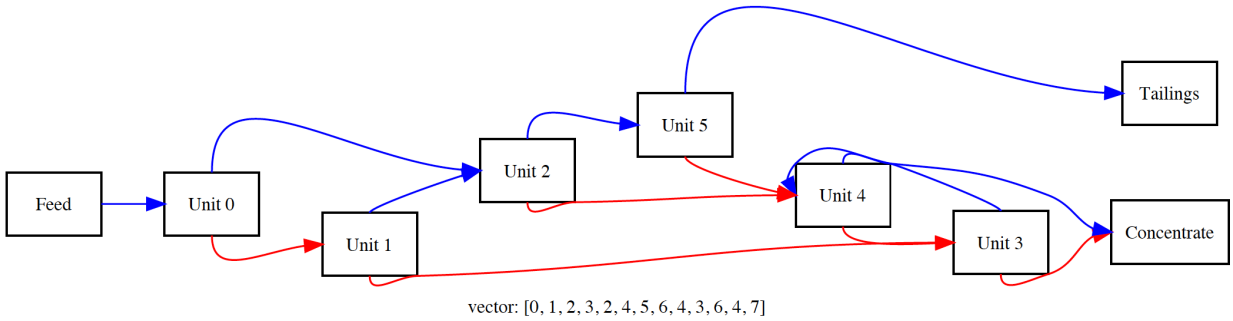


vector: [0, 0, 2]

with one unit but recycle itself.



with three units, this valid case was taken from the project description file.



with six units, where some units does not reach both end. These are included in test1.cpp.

2.3 Assessing circuit fitness

The circuit fitness is assessed by its the amount of output valuable and waste. The formula for it is

$$f = 100(v - 5w)$$

Where f is the fitness score, and v and w are the valuable and waste components of the circuits concentrate stream.

The balance mass function is implemented to calculate the fitness. It updates the feed, the concentrate and the tail stream of every unit in each iteration, until the feeds of every unit converge to a certain value(steady state) or the iteration times reaches a max value.

In order to update parameters for every unit. we need to first initialize the circuit. The initial feed values of every unit are set to 10 and 100 in order to reach the convergence as quick as possible. After that, the program goes into the loop to update unit parameters. The updating process includes three steps. It first finds the outputs of each unit, and then set circuit feed to 10 and 100 and set other feeds to 0. Finally, it search for every unit if a recursion happens in the output, if it does, then add these outputs to the feed of the recursive unit.

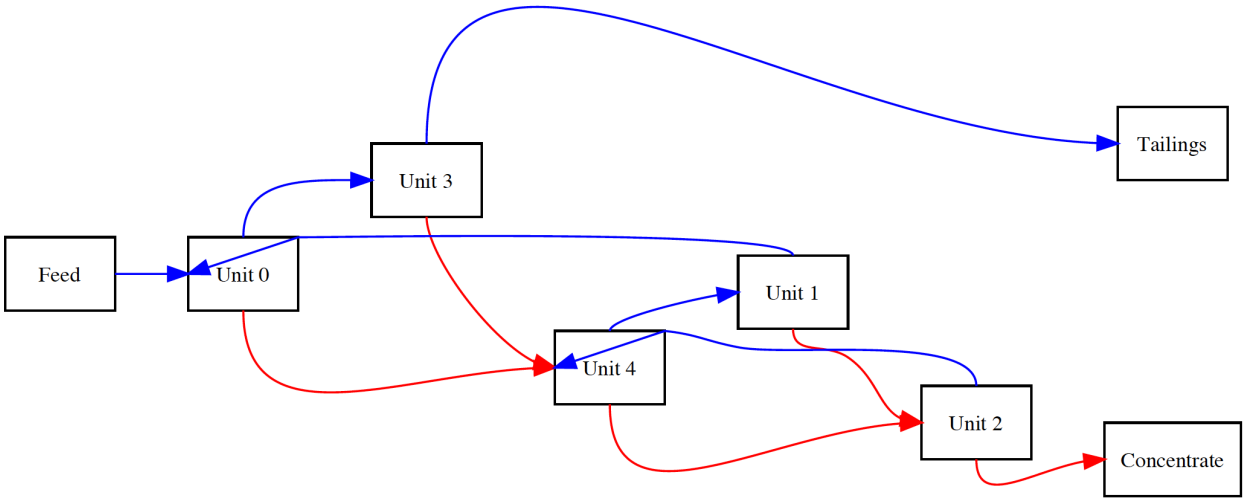
If the max difference between last time feed and current feed is smaller than a tolerance value, then we

consider it as a convergence. The calculation of convergence condition includes two steps. First, the function find out the max value of difference between the amount of this time valuable and next time valuable, and similarly repeat it for the waste difference. Second, find the max value between max valuable difference and waste difference. In order to calculate the convergence parameters, the CUnit class need to have two feed value–the old one and the current one.

In the case of divergence, its clear that if even after the maximum number of iterations (200, which is quite generous), if the maximum difference in any feed component in the circuit is still large (lets say 10) then we can assume it has either diverged or failed to converge. We then simply return -50000 (worst possible fitness score) as a flag for the genetic algorithm to check for divergence and consider the circuit invalid.

2.3.1 Testing

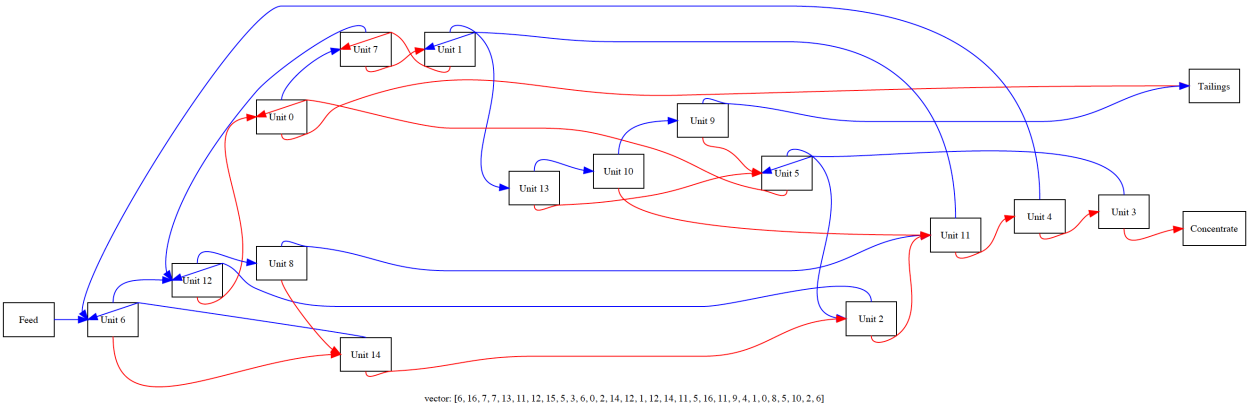
We subjected this to two tests, both provided in the project introduction slides:



vector: [0, 4, 3, 2, 0, 5, 4, 4, 6, 2, 1]

$$[0, 4, 3, 2, 0, 5, 4, 4, 6, 2, 1]$$

with 5 units and a fitness score of 24.82, and the much larger one

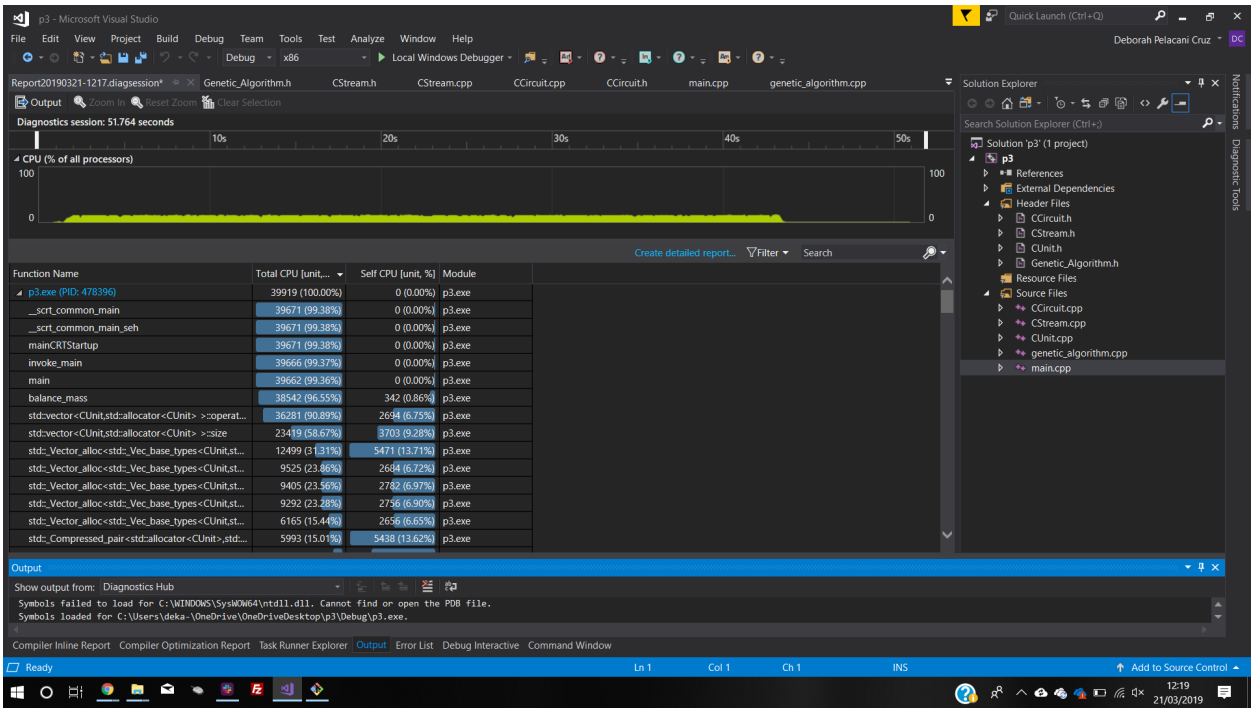


$$[6, 16, 7, 7, 13, 11, 12, 15, 5, 3, 6, 0, 2, 14, 12, 1, 12, 14, 11, 5, 16, 11, 9, 4, 1, 0, 8, 5, 10, 2, 6]$$

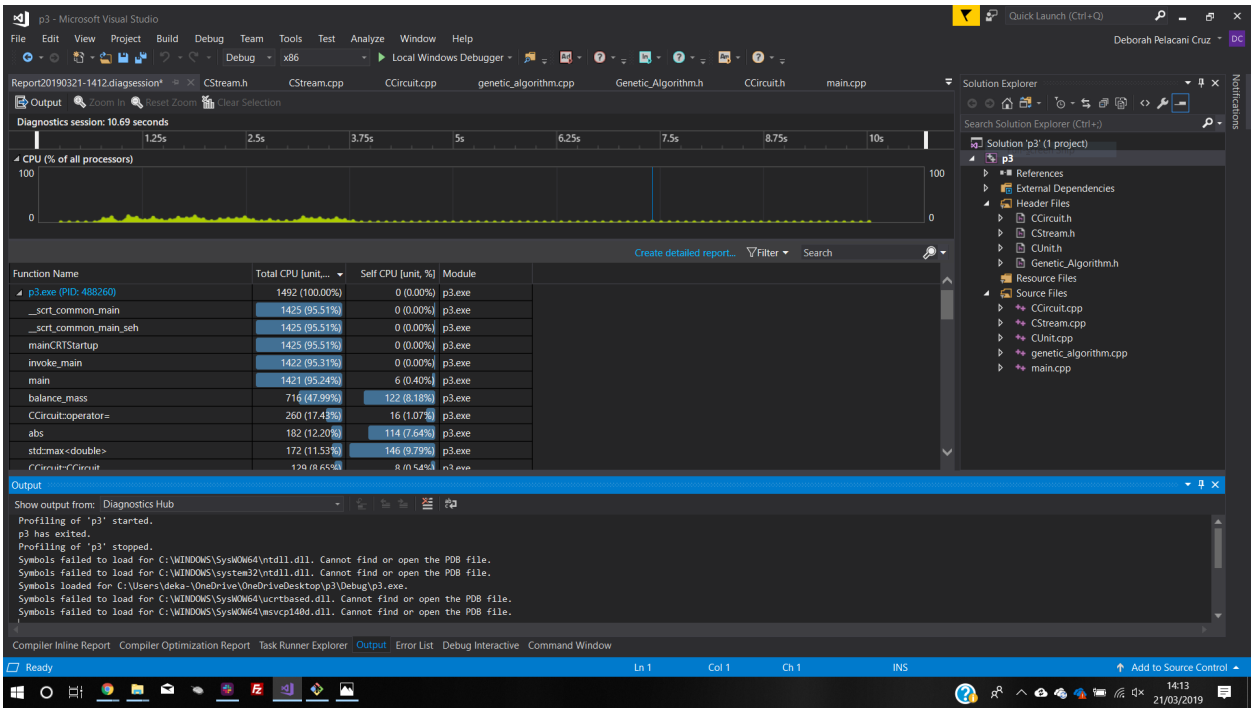
with 15 units and a score of 10.47, and both test cases passed. These are included in test3.cpp.

2.3.2 Optimisation

After profiling, we realised that around 96% of the code's run time was spent in the balance mass function, and an embarrassingly large amount of time was spent only on converting the unit array in the circuit object to a vector.

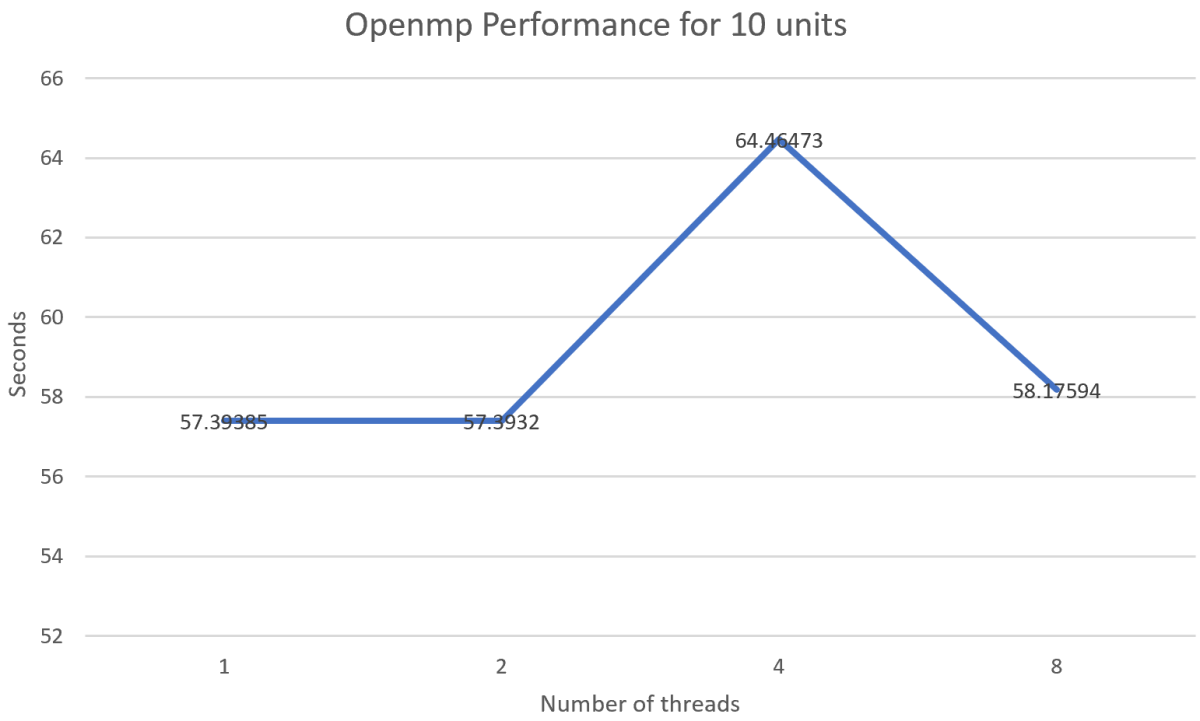


Once we removed this and made the function compatible with a simple array of units, we observed a massive speed up in the circuit modelling function, almost two orders of magnitude! (going from taking up 96% of the run time to only 47%).



2.3.3 Parallelisation

Within the assessment function, we decided to use a multi-threaded approach, given that this function was called on individual circuits themselves and required a significant amount of shared memory (which otherwise would have required send/receive operations). After testing, it was clear that parallelising every for loop would not be ideal, and so we decided to only parallelise the loop responsible for calculating the concentrate and tail streams from the feed stream for each unit. The OpenMP parallelisation was tested on 100 circuits each containing 10 units. The resulting performance plot is shown below:

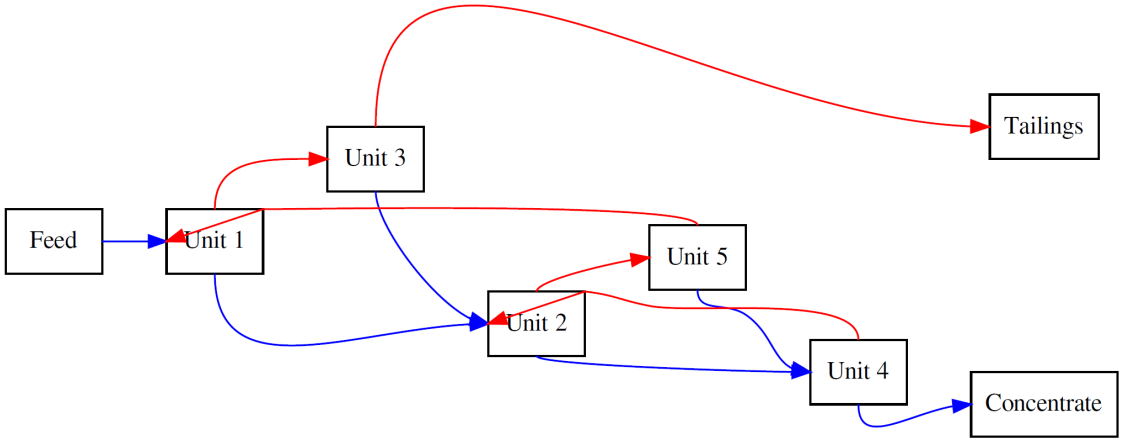


Overall we believe our implementation of OpenMP did not produce significant changes to this particular problem. The overhead created by the parallelisation does not compensate the parallelisation itself. However, it is worth investigating how the performance changes with a greater number of circuit units. Due to the timeframe we have been unable to produce these additional tests.

3 Results

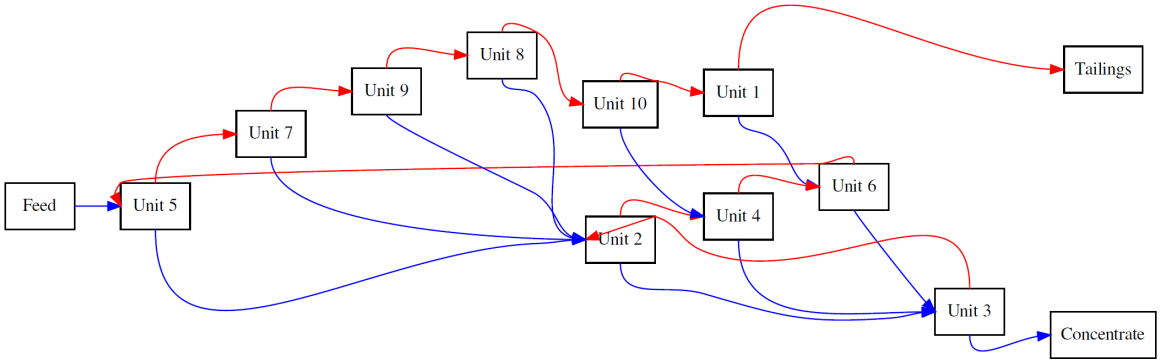
3.1 Optimum circuit

We have implemented our program on 5 units, 10 units and 15 units. Iterations presented are recorded for when the software first identifies the best circuit. The optimum circuit of each situation is as following:



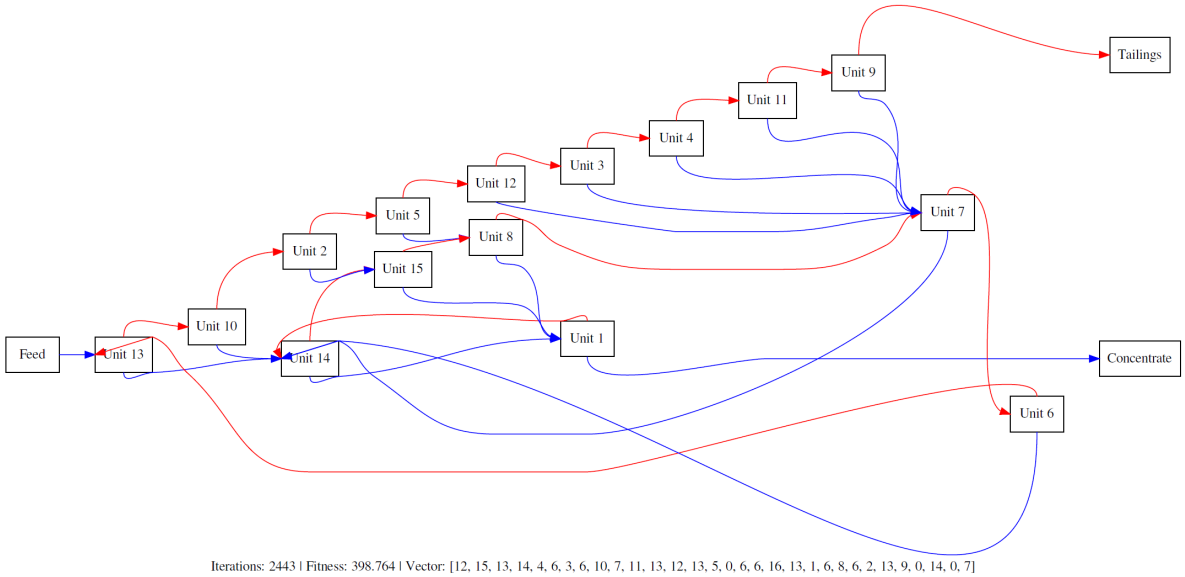
Iterations: 47 | Fitness: 24.8163 | Vector: [0, 1, 2, 3, 4, 1, 6, 5, 1, 3, 0]

with 5 units. The fitness value of 24.8163 corresponds to our given reference value.



Iterations: 1076 | Fitness: 165.753 | Vector: [4, 5, 11, 2, 3, 10, 1, 2, 5, 1, 6, 2, 4, 1, 8, 1, 9, 1, 7, 3, 0]

with 10 units. The fitness value is of 165.753.

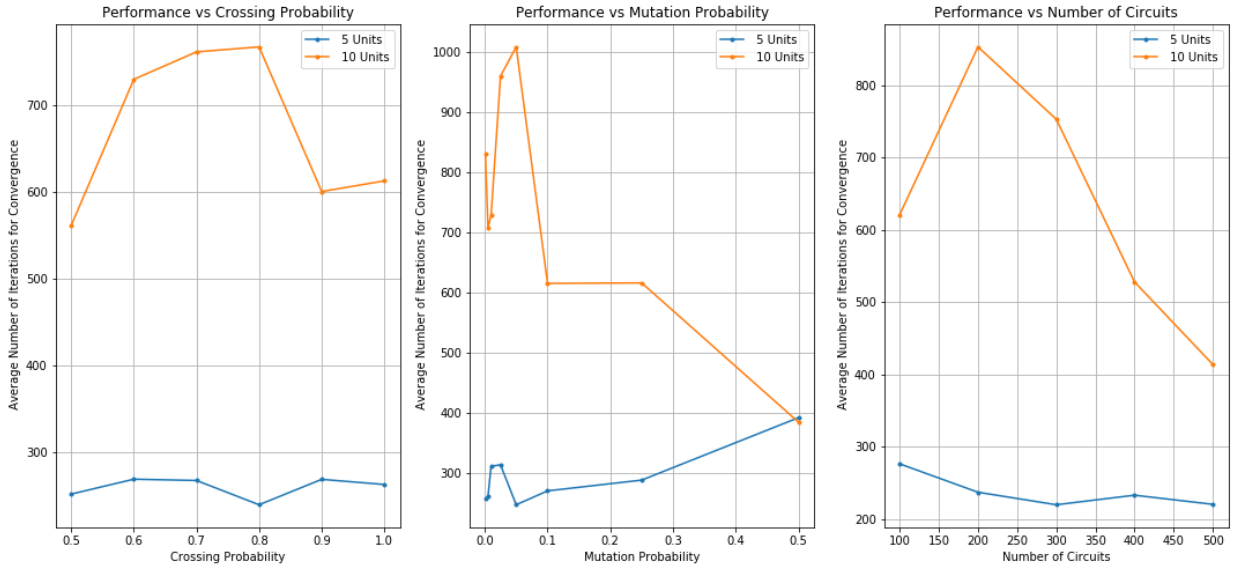


with 15 units. The fitness value of 398.764 also corresponds to our reference value, which implies our program can successfully find the best circuit for different numbers of units. This validation makes us confident that the circuit we found for 10 units is correct.

3.2 Genetic algorithm convergence

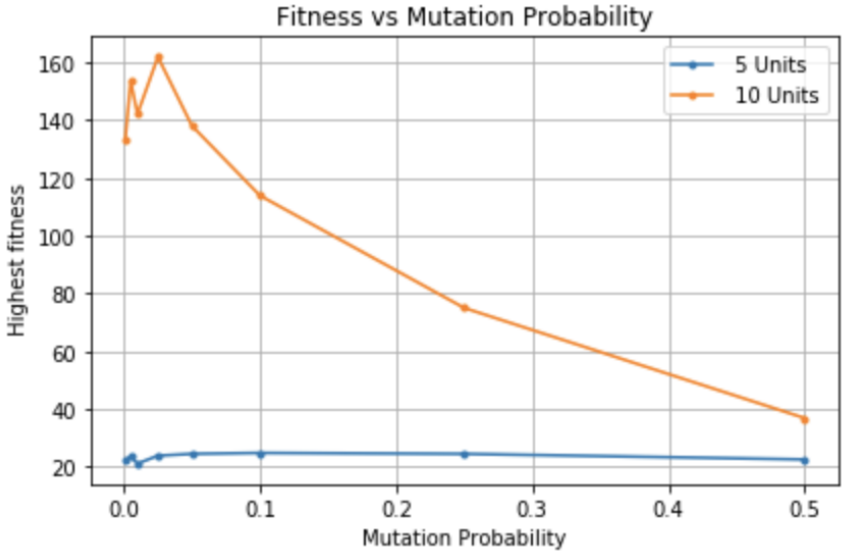
3.2.1 Changing algorithm parameters

We measured the performance of our software by recording the amount of iterations required to obtain the best circuit by varying the following parameters: probability of circuits crossing over, probability of each element of the circuit mutating and the number of circuits per generation. The results are shown in the following plots:



The number of iterations are averaged over 10 executions for a maximum iteration number of 5000 and a convergence requirement of 200 iterations (i.e. the number of iterations by which the best circuit remains

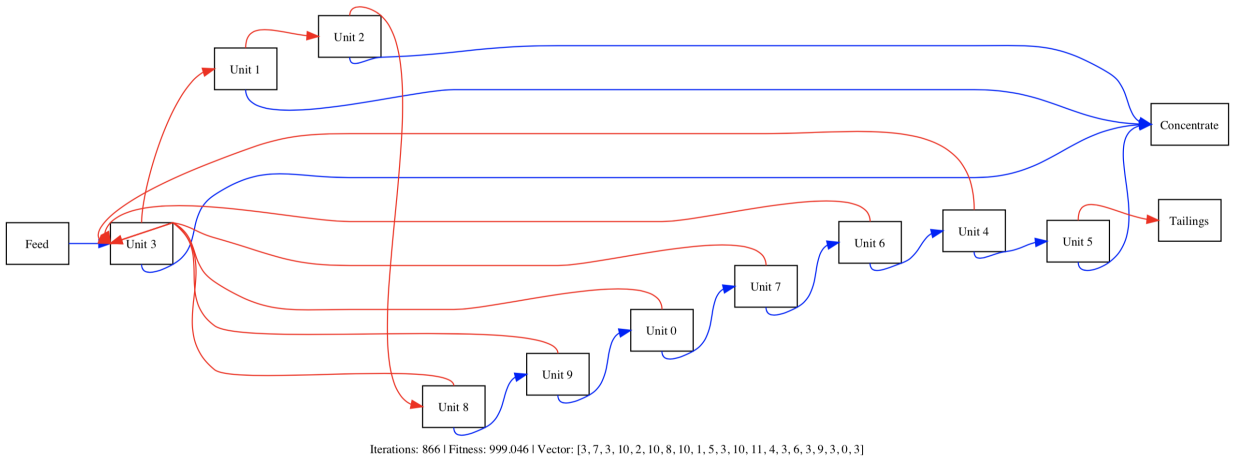
unchanged). The plots were obtained by varying each parameter individually. Standard values are: 90% probability of crossing over, 1% probability of mutation, and 100 circuits per generation. From the plots we can infer that the optimal probability of crossing over is problem-dependent, but should lie between 80-100%. The general trend of decreasing iterations with increasing number of circuits is as expected. For a wider range of circuits the likelihood of a generation producing the optimal circuit increases. The probability of mutation, on the other hand, shows different trends depending on the number of units in the circuit. Increasing mutation for 5 circuits decreases the likelihood of obtaining the best circuit whereas for 10 units this likelihood is increased. We have observed, however, that for 10 units an increasing probability of mutation is associated with a lower fitness value. This seems to indicate the best circuit is stuck in a local minimum and any conclusions from this plot are invalid. Further tests would include the performance with mutation probability for wider range of circuit units and increasing the convergence requirement to at least 500 iterations. These tests were not performed due to the time frame.



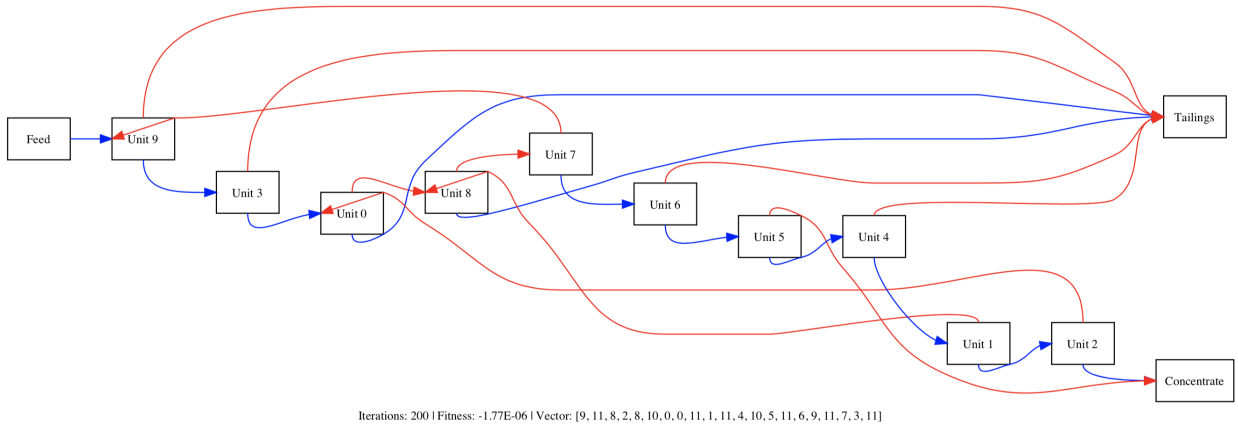
3.2.2 Changing value-waste weightings

Another two variables that we can try to alter are the value-waste weighting. The default value of one kilogram of Gormanium is $100, while the penalty of each kilogram of waste is $500. By altering these values, we can visualize the optimal circuit in cases of when there is no penalty as well as the no reward case.

Ideally, the program should return a circuit that tries to sends everything to the concentrated exit when there is no penalty. On the other hand, no reward case should avoid sending anything to the concentrated exit. The figures below are the results that were obtained from running our simulation.



No penalty case



No reward case

It can be seen that both case act as we predicted, thus confirmed that our program can simulate these two extreme cases correctly.

We have also ran some more tests by varying the cost and penalty parameters but since the program was designed to maximize Gormanium and minimize waste in the concentrate exit, the resulting circuit are all similar. The only difference would be the fitness number since it was calculated from those two parameters.

4 References

[1]: Aaron Michaelove - 1992 - *Amdahl's law* - https://home.wlu.edu/ whaleyt/classes/parallel/topics/amdahl.html

[2]: Xiaohui Zhu - 2006 - *Particle Swarm Optimisation* - http://www.swarmintelligence.org/tutorials.php