Applying Computational Science Project 2

The Gormanium Rush

Optimal mineral recovery using Genetic Algorithms

Separation technologies are widely used to improve the purity of products. These technologies usually take the form of identical or near identical **separation units** (referred to as **units** for brevity) that are arranged in **circuits**. In minerals processing, for instance, the separation units are things like flotation cells or spirals, while in the upgrading of nuclear material the separator unit will be a centrifuge. While the separation units are different, their key property is that they will recover a proportion of the "valuable" material and will simultaneously recover a proportion of the "waste" material (Fig. 1).

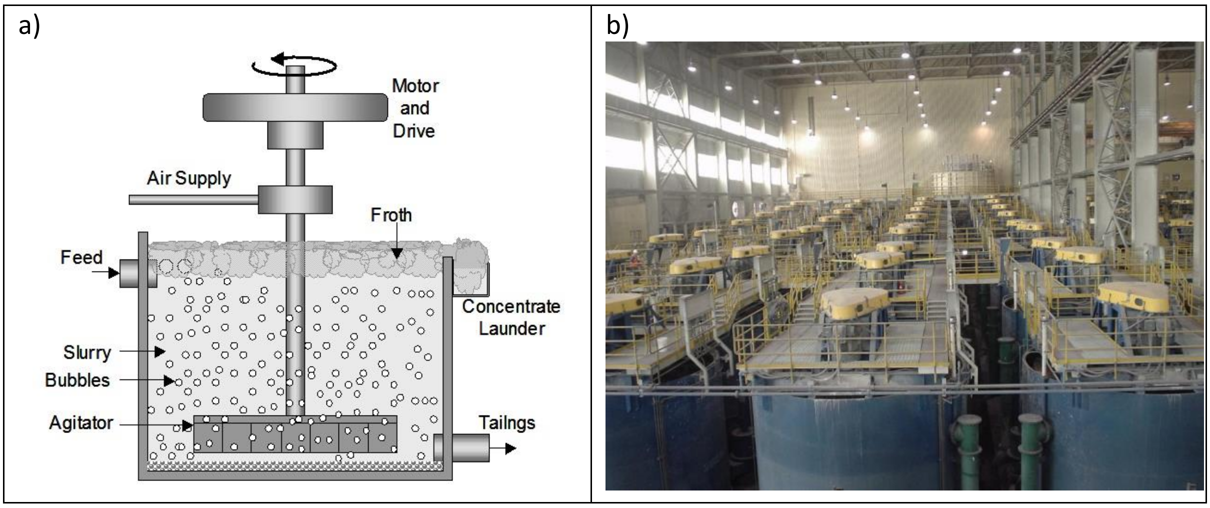


Figure 1: a) Schematic of a froth flotation cell (an example of a separation unit), which produces a concentrate stream via the froth with the rest of the material flowing out of the unit as tailings b) Picture of a large number of flotation cells arranged as a circuit

While a single separation unit in isolation will only recover a small proportion of the valuable material, multiple units can be combined together in circuits that can enhance recovery of the valuable material. The basic challenge is to design the circuit for optimal recovery. However, different circuit designs will result in different amounts of waste being recovered and thus different purities of the final product. While some circuit designs will be unambiguously better than others (i.e., produce both a better recovery and purity), for many designs there will be a compromise between the overall recovery (total mass of valuable material recovered) and purity (proportion of valuable material to the total material recovered). In such circumstances the optimum circuit will be an economic decision based on the balance between how much you are paid for the product and penalised for a lack purity.

In this project we are going to restrict ourselves to both units and an overall circuit that produces two products (**streams**), namely a **concentrate stream** and a **tailings stream**. The success of the circuit will be measured based on the first product, the concentrate stream, with a price paid per kg of the valuable material and a penalty charged per kg of waste material in the concentrate stream. The second product, the tailings stream, will be dominated by waste material and discarded at no cost. In both individual units and the overall circuit, intermediate products could be produced, but we will ignore this as it dramatically increases the complexity of any potential optimisation.

These circuits can have simple rows of units with the tailings or concentrate being passed to the next unit along (but not both). The circuits can also involve recycles, where a stream is passed back to a unit nearer the beginning of the circuit (but not the same unit). This means that the number of potential circuits increases factorially with the number of units in the circuits. A brute force approach to find the optimal circuit is thus only feasible for circuits consisting of relatively few units (by the time there are 60 units in the circuit there are more potential circuit configurations than there are atoms in the universe!).

The large number of possible circuit configurations necessitates an optimisation algorithm to search for a solution. As the configurations are discrete, standard gradient search algorithms won't work. There are a number of potential algorithms that can be applied to such problems and the one that we will be using is a **genetic algorithm**. The valuable material that you are trying to recover is **gormanium**.

Methodology

Genetic Algorithms

Genetic algorithms, as their name implies, work in a manner not dissimilar to how natural selection works. The heart of the algorithm is a representation of the problem as a vector of numbers (the "genetic code" of the problem). In this problem, the genetic code represents the connections in the circuit (see Fig. 2). To start, a large number of these vectors need to be randomly generated and evaluated, with a single number assigned to represent the success of each vector (i.e., the performance of the circuit). The function that takes in the vector and returns a single performance number is often referred to as the **fitness function**. The convergence of the algorithm will usually be enhanced if it can be ensured that every one of the initial vectors is both unique and valid, though this may in itself be a computationally intensive task.

The set of random initial vectors will form the **parents** for the next generation of offspring vectors via a combination of two processes:

Mutations – Random changes in the numbers in the parent vector.

Crossover – This is roughly equivalent to sexual reproduction. In this process a portion of one parent vector is swapped with a portion of another parent vector. The motivation for swapping a portion of a parent vector with another rather than swapping individual values randomly is that, over successive generations, values that work well together will end up next to one another in the vector (roughly equivalent to genes); preserving these portions of the genetic code is beneficial. In this problem, for instance, where the values in the vector represent connections in the circuit, a certain set of connections between a few units may be useful in more than one location in the overall circuit.

The steps in a basic genetic algorithm are as follows:

1) Start with the vectors representing the initial random collection of valid circuits.
2) Calculate the fitness value for each of these vectors.

You now wish to create *n* child vectors

3) Take the best vector (the one with the highest fitness value) into the child list unchanged (you want to keep the best solution).
4) Select a pair of the parent vectors with a probability that depends on the fitness value. In this case you might want to start by using a probability that either varies linearly between the minimum and maximum finesses of the current population. This should be done "with replacement," which means that parents should be able to be selected more than once.
5) Randomly decide if the parents should crossover. If they don't cross, they both go to the next step unchanged. If they are to cross, a random point in the vector is chosen and all of the values before that point are swapped with the corresponding points in the other vector.
6) Go over each of the numbers in both the vectors and decide whether to mutate them (this should be quite a small probability). If the value is to be mutated, you should move the value by a random amount (you can decide the step size). In these circuits you can avoid clustering of the results near the minimum and maximum unit numbers by "wrapping" the change (i.e., don't artificially restrict the movement, rather use a modulus to bring it back within range). In the circuit problem values are essentially completely independent of one another as there is no reason to think that a connection to a unit with a number close to that of the currently

connected unit will be better than the connection to any other unit. This means that the potential step size can be set to be the same as the size of the valid range, though in other optimisation problems it may be useful to preferentially search values close to the current one.

7) Check that each of these potential new vectors are valid and, if they are, add them to the list of child vectors.
8) Repeat this process from step 4 until there are *n* child vectors
9) Replace the parent vectors with these child vectors and repeat the process from step 2 until either a set number of iterations have been completed or a threshold has been met (e.g., the best vector has not changed for a sufficiently large number of iterations).

Tuning the algorithm – You may notice that there are a number of hyper-parameters that you can tune when running these algorithms. These include:

- the number of offspring *n* that are evaluated in each generation;
- the probability of crossing selected parents rather than passing them into the mutation step unchanged (a recommended range is between 0.8 and 1);
- the rate at which mutations are introduced (recommended probabilities of 1% or lower).

You will need to investigate how these hyper-parameters change the rate of convergence achieved. The optimum parameters will depend on both the type of problem being investigated and the size of the problem (they will be different between your test problem used to develop the genetic algorithm and the actual circuit simulation, and they will vary with the number of units in the circuit).

Before the optimum circuit can be determined, though, we need to be able to evaluate the performance of a given circuit.

Modelling a Circuit

There are a number of aspects that need to be covered in terms of calculating circuit performance. The first is representing the circuit as a vector:

Representing the Circuit – In these circuits the separation units can take in as many input streams as desired, but they must have only two output streams each. One output stream is the "concentrate" containing more of the valuable material than the waste; the other output stream is the "tailings" containing more of the waste than the valuable material. We can therefore represent the circuit in terms of the stream destinations, with each unit represented by two numbers. The first number is the unit number of the destination of the concentrate stream; the second number is the unit number of the destination of the tailings stream. In addition to the *n* units, the streams can also be directed to the final tailings stream or the final concentrate stream, which is the product to be evaluated. As well as recording the destination unit of each output stream, the genetic code for the circuit must also include the destination unit of the circuit **feed** or input. This means that if there are *n* units in the circuit, it will be represented by vector $2n+1$ long, with each value being in the range of zero to less than $n+2$ (0 to n-1 being the identifiers for the destination units, while a value of *n* represents the destination of the final concentrate and $n+1$ the destination of the final tailings), with the exception of the input feed's value, which should be between zero and $n-1$ (the input feed should not be fed directly into one of the final product streams).

The following are a couple of examples of circuit vectors and the corresponding circuit layouts:

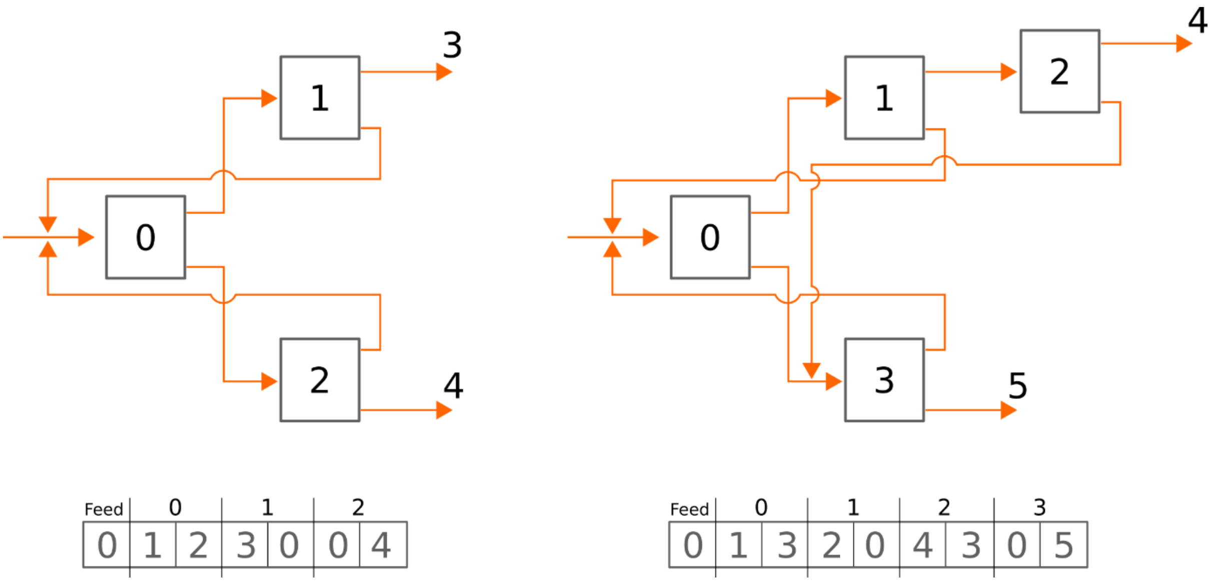


Figure 2: Two simple circuits and their corresponding circuit vector ("genetic code")

Modelling the performance of a Unit – The performance of a separation unit operation will depend on a number factors, including the feed composition and feed rate. We will assume that there are only two components in the feed, namely a waste and a valuable component (in many systems there will actually be a range of valuable and waste components with different separation performances).

We will be assume that the recovery of the material to the concentrate follows a first-order kinetic relationship. This means that the recovery of material to the concentrate is proportional to the amount of material in the cell. If the material in the cell is well mixed, then the recovery, R, of material i to the concentrate obeys the following relationship:

$$R_i = \frac{k_i \tau}{1 + k_i \tau}$$

Where k_i is the rate constant associated with component i and τ is the average residence time in the cell. We will assume that $k_{gormanium} = 0.005\ s^{-1}$ and $k_{waste} = 0.0005\ s^{-1}$.

The residence time is the volume of the cell divided by the combined volumetric feed rate of solids and water into the cell. We will assume that both solids have a density, ρ, of 3000 kg/m^3 and that the total solids (gormanium + waste) content of the feed by volume, φ, is 10% in all cells. We will also assume that all the cells have a volume $V = 10\ \mathrm{m}^3$. In this case,

$$\tau = \varphi \frac{V}{\sum \frac{F_i}{\rho}}$$

where F_i is the mass flow rate of solid component i in the feed to the cell.

Once the recovery has been calculated, it can be used to calculate the mass flow rates of both components in both the tails and concentrate streams from the cell.

$$C_i = F_i\, R_i$$

$$T_i = F_i - C_i$$

Where C_i is the mass flowrate of material i to the concentrate and T_i is the mass flowrate of material i to the tails.

As an example calculation, if $F_{waste} = 80\ kg/s$ and $F_{gormanium} = 20\ kg/s$ then $\tau = 0.1\ \frac{10}{\frac{80+20}{3000}} = 30\ s$. $R_{waste} = \frac{30 * 0.0005}{30 * 0.0005 + 1} \approx 0.0148$. $R_{gormanium} = \frac{30 * 0.005}{30 * 0.005 + 1} \approx 0.130$. This means that the concentrate and tails have the following flowrates: $C_{waste} \approx 1.18\ kg/s$, $C_{gormanium} \approx 2.61\ kg/s$, $T_{waste} \approx 78.82\ kg/s$, $T_{gormanium} \approx 17.39\ kg/s$.

There is one note of warning: In some invalid circuits the feed rate in some cells will iterate towards zero, which will cause issues in the calculation of the residence time. To stop overflow errors in the calculation, set a minimum flowrate/maximum residence time to stop these errors (a minimum flowrate of 1.e-10 m^3/s would be appropriate).

Modelling the Circuit – To combine the behaviour of each of the individual units into an overall circuit simulation we must calculate the mass flow rate of each component in each stream. To do this we will assume that the circuit is at steady state, which implies that the flows in each stream do not change with time and that there is no accumulation (i.e., the total feed into a unit is equal to the sum of the flow out of the unit through the two output streams).

As these circuits can (and will typically) involve recycles—that is, one or more of the output streams can feed back into an earlier unit in the circuit—the circuit performance will need to be solved iteratively. (Note that, as this particular problem has a linear relationship between feed and product flows, you could solve the circuit directly by matrix inversion, though this would not be possible for more complex unit performance models). As successive substitution of the component mass flows in the streams is guaranteed to converge in these types of problems (you can look up the proof if you wish) we recommend this approach to start with. It is possible to achieve quicker convergence using a more complex convergence algorithm, though you do still need to ensure stability of convergence.

The following is a simple successive substitution algorithm that is guaranteed to converge *if a valid solution exists*:

1) Give an initial guess for the feed rate of both components to every cell in the circuit
2) For each unit use the current guess of the feed flowrates to calculate the flowrate of each component in both the concentrate and tailings streams
3) Store the current value of the feed to each cell as an old feed value and set the current value for all components to zero
4) Set the feed to the cell receiving the circuit feed equal to the flowrates of the circuit feed
5) Go over each unit and add the concentrate and tailings flows to the appropriate unit's feed based on the linkages in the circuit vector. This will also result in an updated estimate for the overall circuit concentrate and tailings streams' flows.
6) Check the difference between the newly calculated feed rate and the old feed rate for each cell. If any of them have a relative change that is above a given threshold (1.0e-6 might be appropriate) then repeat from step 2. You should also leave this loop if a given number of iterations has been exceeded or if there is another indication of lack of convergence as this will generally indicate an invalid circuit configuration (be aware that as you test this on larger circuits the number of iterations required for convergence of valid circuits will increase).
7) Based on the flowrates of the overall circuit concentrate stream, calculate a performance value for the circuit. If there is no convergence you may wish to use the worst possible performance as the performance value (the flowrate of waste in the feed times the value of the waste, which is usually a negative number).

Checking Circuit Validity – A key step before running a simulation is to ensure that the circuit is a valid one. Recall that the algorithm described above for evaluating circuit performance will not converge if the circuit is invalid. Lack of convergence is thus a test of circuit validity. However, as this is a computationally intensive way to evaluate circuit validity, we recommend that pre-requisite validity checks are implemented, based on the following considerations.

For the circuit to be valid a few conditions must be met:

- Every unit must be accessible from the feed. I.e., there must be routes that go forward from one unit to the next starting at the feed and ending at each of the units in the circuit
- Every unit must have a route forward to both of the outlet streams. A circuit with no route to any of the outlet streams will result in accumulation and therefore no valid steady state mass balance. If there is a route to only one outlet then the circuit will be able to converge, but there will be one or more units that are not contributing to the separation and could therefore be replaced with a pipe.
- There should be no self-recycle. In other words, no unit should have itself as the destination for either of the two product streams.
- The destination for both products from a unit should not be the same unit.

Note that this is not an exhaustive list of how circuits can be invalid or obviously sub-optimal. You should think about other ways in which circuits can be invalid and do tests for these. Even once you have implemented validity checking based on the above criteria, you should still check for lack of convergence as there are some pathological circuit configurations that you might not think of in the validity testing. Hence, you should still check if the circuit simulation is diverging as this will indicate an invalid circuit (have a maximum number of iterations allowed in the circuit mass balance convergence). It may be insightful to look at some of the circuits that are identified as invalid through non-convergence and see if you can identify why they are invalid.

When doing these checks you should note that the circuit takes the form of a directed graph. It is often easiest to write recursive functions to traverse the graph, though you should note that, because recycle is allowed, you do need to ensure that you don’t get stuck going around a recycle loop. The easiest way to do this is to mark the units that you have already visited. A simple generic function for using recursion to traverse the circuit is outlined in the section below.

Traversing the Circuit using Recursion

Recursive functions are the easiest way to traverse a tree or graph. This short code snippet demonstrates a function which marks every unit which is accessible from a given unit (i.e. every unit that product from a given unit can potentially reach). It assumes that the data for each individual unit is stored in a class:

```
class CUnit
{
public:
        // index of the unit to which this unit's concentrate stream is connected
        int conc_num;
        // index of the unit to which this unit's tailings stream is connected
        int tails_num;
        // A Boolean that is changed to true if the unit has been seen
        bool mark;
        …other member functions and variables of CUnit
};
```

And it assumes that there is an array of these units:

```
int num_units;
…set a value to num_units
vector<CUnit> units(num_units);
```

The following function is recursive, which means that it calls other instances of itself within the function.

```
void mark_units(int unit_num)
{
        if (units[unit_num].mark) // Exit if we have seen this unit already
                return;
        units[unit_num].mark = true; // Mark that we have now seen the unit

//If conc_num does not point at a circuit outlet, recursively call the function
        if (units[unit_num].conc_num < num_units)
                mark_units(units[unit_num].conc_num);
        else
                …Potentially do something to indicate that you have seen an exit

//If tails_num does not point at a circuit outlet, recursively call the function
        if (units[unit_num].tails_num < num_units)
                mark_units(units[unit_num].tails_num);
        else
                …Potentially do something to indicate that you have seen an exit

}
```

To use this function in the code you need to use the specification vector to set the `conc_num` and `tails_num` values for every unit in the units array:

```
//Set all the cells to unseen
for (int i=0; i<num_units; i++)
        units[i].mark = false;

//Mark every cell that start_unit can see
mark_units(start_unit);

for (int i=0;i<num_units;i++)
        if (units[i].mark)
                …You have seen unit i
        else
                …You have not seen unit i
```

Base Case Circuit Specification

Your challenge is to determine the optimum circuit configuration and performance for a circuit that contains 10 units. It should have a total circuit feed of 10 kg/s of gormanium, the valuable material, and 100 kg/s of the waste material You will be paid £100 per kg of gormanium in the product and charged £500 per kg of the waste material.

Your task is to provide advice on optimum circuit configuration. Therefore, additional credit will be given to teams that demonstrate a deeper understanding of how optimum circuit configuration depends on the number of units as well as the economic factors through application of their tool. This might include identification of common configuration patterns and how these change with different economic constraints.

What is required of each team?

Software development

Your software tool should comprise four parts, which can be developed independently and should be written in a modular fashion.

1) Create software capable of using a genetic algorithm to optimise a system represented by a specification vector where the performance is based on the evaluation of a fitness function that takes in the vector and returns a single number to be maximised.
2) Write a mass balance simulator which can take in a specification vector for the circuit and calculate the mass flows of all components in every stream and ultimately returns a single number to represent the monetary value of the final concentrate.
3) Write a validity checking function that takes in the circuit vector and returns true or false based on its assessment of the circuit validity.
4) Write a postprocessing code that can convert any circuit vector into an image of the circuit that this vector represents. There are Python libraries that can assist in this task (note that the circuit is a directed graph). You could also expand the code to able to animate the convergence path towards the optimum circuit.

By developing the software in this modular fashion, you could either use the genetic algorithm on a different problem or apply a different optimisation algorithm to the circuit simulator.

Model analysis and performance enhancement

When you have a working tool, you should use it to address the following:

5) Obtain the optimum circuit configuration for the base case specifications. How quickly and reliably does the algorithm converge on the optimum and how does this change with the genetic algorithm parameters such as the number of child vectors used and the mutation rate? Note that the reliability with which the algorithm finds the global solution will often be in conflict with the number of iterations required to find the optimum. Remember to run the code a number of times to see if an improved optimum solution can be found (the more cells in the circuit the more likely that the code will find a local rather than the global optimum).
6) Investigate how the optimum circuit changes as the various problem-specification parameters change, including the number of units, the price paid for gormanium relative to the cost of disposing of the waste material, and the purity of the input feed. Note that you

will usually have to make large changes in these parameters to drive significant changes in the optimum circuit configuration. Are there any circuit design heuristics that you might recommend based on the observed trends in the optimum configuration? Note that different economic parameters will require different balances between the recovery and purity of the product in the optimum circuit.

7) The fact that the genetic algorithm requires the evaluation of a large number of independent circuit configurations at each iteration means that it is readily parallelised. It is up to you whether you do openMP or MPI based parallelisation (openMP will be easier to implement, but your code will be restricted to working only on a single node). You could also write a script to test different parameter settings by running different instances of your serial code in parallel, though this would get less credit than the implementation of a truly parallel code.

Note that you are not expected to complete all of these tasks, though you should complete tasks 1-5 as a minimum. These tasks also carry the greatest weighting in terms of the marking scheme.

Recommendation for how to initially proceed

There are four aspects to this project that can be worked on independently before they are brought together in the final code. Before any of these tasks can be attempted, though, a data structure for the circuit vector needs to be agreed. Additionally, for testing of the circuit validity and carrying out circuit simulation it would be useful to agree data structures for the individual units, streams etc.

The following are four tasks that can be independently tackled:

Circuit Simulation – For a given circuit vector you need to be able to calculate the mass balance over all the units and thereby the composition of both of the outlet streams as this is required to produce a single fitness value. You can develop this simulator without validity checks or the genetic algorithm by manually creating a few test vectors. This can be done by drawing a circuit that you can see is valid and then writing out the corresponding circuit vector. In drawing the test circuits, you should design complex circuits with lots of recycles rather than circuits you think may be efficient in order to have a good test of the circuit simulator's convergence behaviour.

You can use the following form for the circuit performance/evaluation function definition. This form assumes that you are representing your individual vectors as a simple array of integers:

```
double Evaluate_Circuit(int *circuit_vector, double tolerance, int max_iterations)
```

Circuit Validity – Having valid circuits is important for convergence of the mass balance. While you could use non-convergence of the mass balance as an indicator of an invalid circuit, it is much better to explicitly check circuit validity first. Checking circuit validity can be done without the ability to calculate the mass balance and can therefore be done as an independent task (how these circuits might be invalid is discussed in an earlier section).

The following function definition can be can be used for this function.

```
bool Check_Validity(int *circuit_vector)
```

Alternatively, you could agree an internal circuit representation akin to that used in the section on traversing the circuit and have a shared function which converts the circuit vector into this format for use in both the circuit evaluation and validity checking.

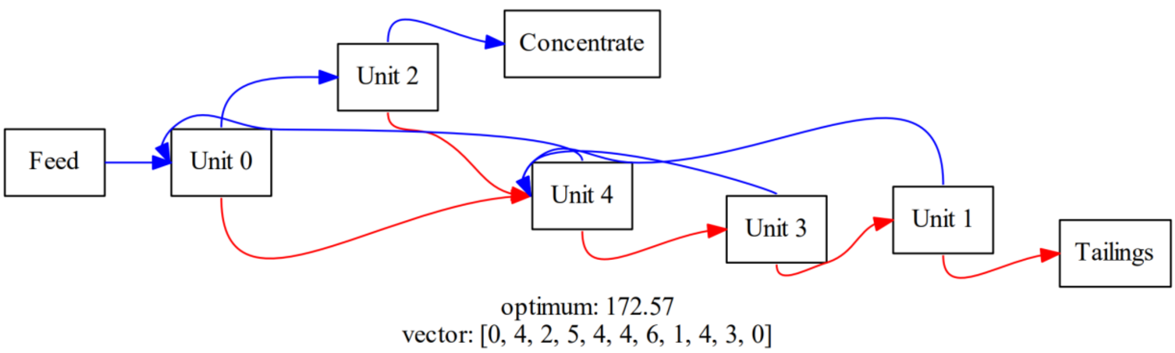
The Genetic Algorithm – This can be developed without the circuit simulator as you simply need a fitness function that gives back a value based on a given number vector. While this will ultimately be the value of the concentrate stream as calculated in the circuit simulator, in order to develop the algorithm a simple test case is to choose a random vector as the answer and have the fitness function return a number that represents the difference between this answer vector and test vector. The advantage of this approach is that you know if the answer returned is the correct one. Don't rely on this test problem to tune for the correct parameters to use in the genetic algorithm as this is a dramatically simpler optimisation than the actual one and which should converge much more quickly.

When developing the genetic algorithm (i.e., before the other sub-groups have completed their tasks) you might want to use the following test versions of the above functions (note that the optimum settings for the genetic algorithm will be very different for this test problem compared to the actual problem):

```
bool Check_Validity(int *circuit_vector)
{
        return true;
}

double Evaluate_Circuit(int *circuit_vector, double tolerance, int max_iterations)
{
        double Performance =0.0;
        for (int i=0;i<vector_size;i++)
        {
                //answer_vector is a predetermined answer vector (same size as
circuit_vector)
                Performance+=(20-abs(circuit_vector[i]-answer_vector[i])*100.0;
        }
        return Performance;
}
```

Post Processing – The output from the simulator will be a vector of circuit connections and a performance score. This vector can be represented as a directed graph of the nodes in the circuit:



Your group can choose to use Python or C++ in this postprocessing task and there are various libraries that can assist with this task. One that we recommend is graphviz (https://graphviz.org) and, in particular, its Digraph functionality. You are free to develop your own tools as needed, using *matplotlib* or a similar library in Python, or any widely distributed C++ library.

Assessment

Your group project will be assessed in four ways:

Software (70 marks)

Software will be assessed based on functionality (50/70 marks) and sustainability (20/70 marks).

Functionality and performance (50 marks): Your software will be assessed on its ability to perform the required tasks. 10 marks will be assigned to each of the main sub-tasks (1-4, above), namely the genetic algorithm implementation, the mass balance solver, the circuit validity checking and the post-processing.

A further 10 marks will be assessed based on additional features and optimisations beyond the direct implementation of the algorithms specified in the description. This could include, but is not limited to, things like parallelisation and modifications and improvements to the algorithms used. Part of the assessment for these marks will be based on the final speed and efficiency of the code, but documentation of these additional features is also important so that they are not missed during the assessment.

Marks will be deducted for (a) inaccuracy in the solution; (b) bugs or mistakes in implementation; (c) computational inefficiency.

Sustainability (20 marks): As with all software projects, you should employ all the elements of best practice in software development that you have learned so far. A GitHub repository will be created for your project to host your software. The quality and sustainability of your software and its documentation will be assessed based on your final repository and how it evolves during the week. Please refer to the ACS handbook for more information about the assessment of software quality. You should have test cases for the different aspects of the code, including the genetic algorithm, the circuit simulator and the circuit validity checking. Other important aspect of the sustainability of the project include sufficient and clear documentation of the code, clear descriptions of the code's usage, as well as examples and test cases.

Presentation (20 marks)

Your project will also be assessed on the basis of a 15-minute video presentation that you must upload to MS Stream before the deadline of Friday 25th March, 4:00 pm UK time.

You can record the presentation in any software that you like, but we recommend recording in MS Teams as this allows for simple uploading to MS Stream.

You should view the presentation as your report to the project client. They have tasked you with determining the optimum circuit configuration, with the specified problem constraints. To be awarded the contract for further work, they will want to know how you arrived at your answer and see evidence that your tool can be trusted and performs well. However, they will also be interested in any insight you have gleaned into the sensitivity of the optimum circuit configuration to changes to the problem constraints, particularly the economic factors.

Your presentation should provide the following information:

- A discussion of the solution method and, in particular, any improvements and optimisations made beyond the given algorithms

- Your solution for the base case with 10 units based on the specified process and economic variables.
- An investigation into the program's performance, including both speed of convergence and the robustness of the final solution, and how this is influenced by the genetic algorithm's parameters.
- An investigation into how and why the optimum circuit configuration changes with the input variables, in particular the economic factors.

Teamwork (peer assessment; 10 marks)

After the presentations, you will complete a self-evaluation of your group's performance. This will inform the teamwork component of your mark. Please refer to the ACS guidelines for more information about the assessment of teamwork.

Technical requirements

You should use the assigned GitHub repository exclusively for your project

Your software should be predominantly written in C++, though you can write the post-processing modules in Python if you wish.

Your program should be written in ANSI standard C++ so that it is able to compile under both Windows and Linux.

Your program should be able to be run from the command line without any additional user input so that it can be submitted to run on an HPC system. Feel free to include example submission scripts in your submission.

You should use Github Actions for any automated testing that you implement, but are free to modify the existing testing workflow in the template repository you are supplied.