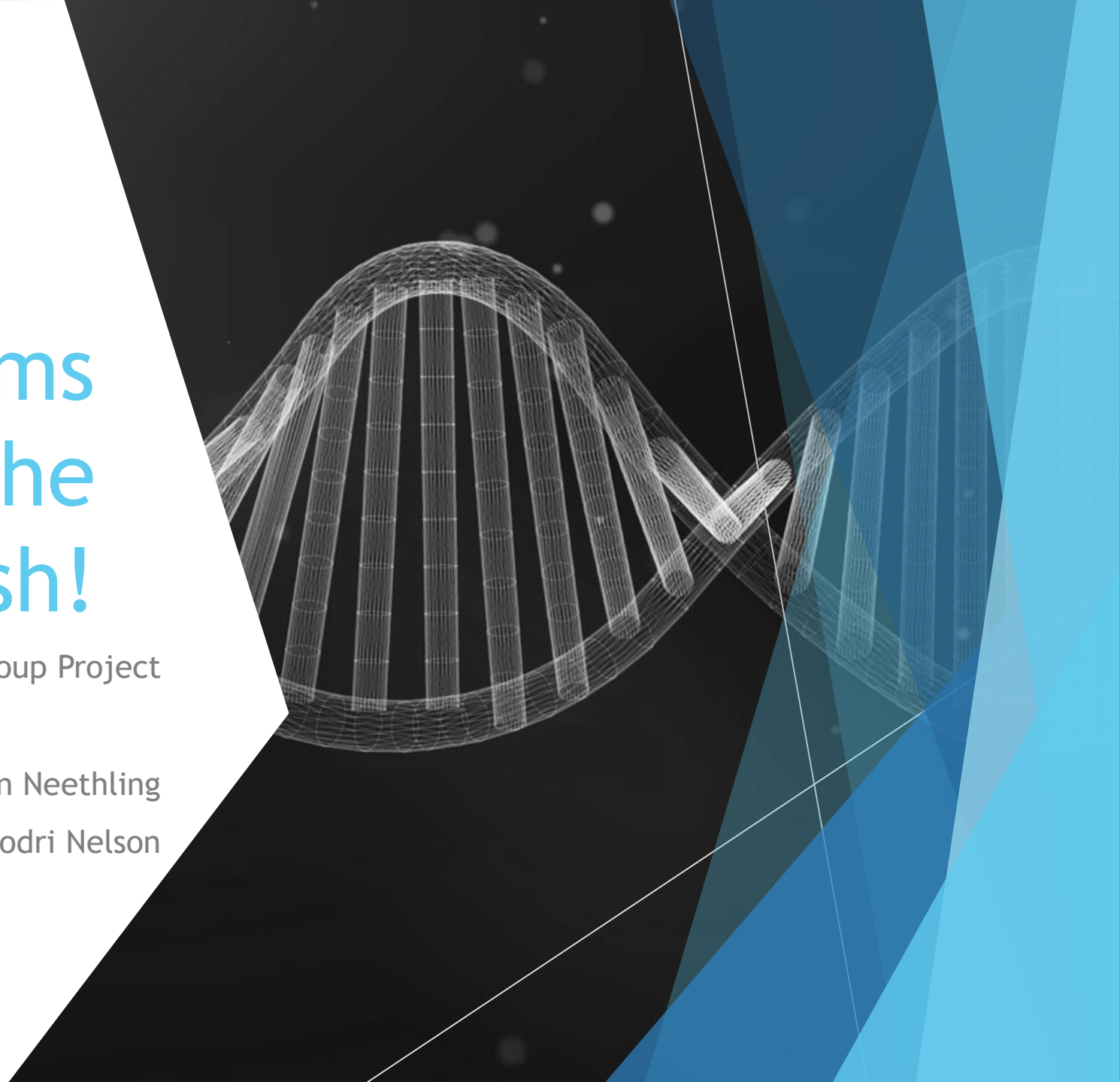
Genetic Algorithms & the Gerardium Rush!

Final ACSE Group Project

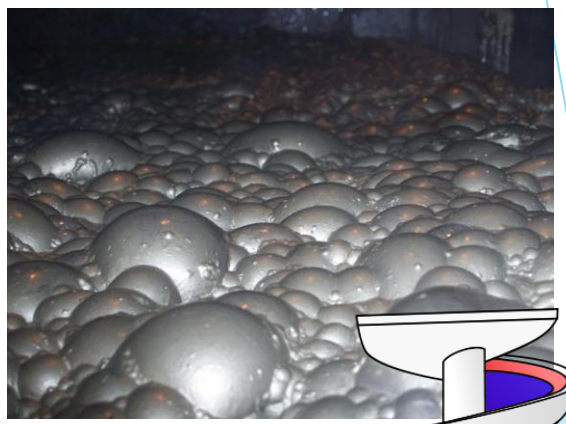
Project designed by Stephen Neethling

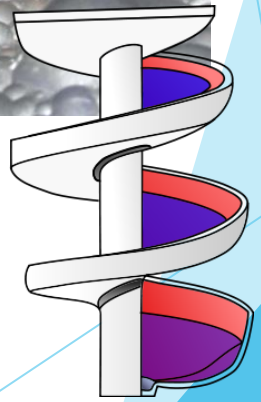
Run by James Percival, Thomas Davison & Rhodri Nelson



Our Problem

- In industry we often want to separate one material from another. E.g.
  - The mineral you want from the waste minerals after mining
  - The radioactive isotope from the non-radioactive ones in nuclear material upgrading
- These are done in separation units. E.g.
  - Froth flotation cells or spirals for minerals
  - Centrifuges for radioactive isotopes





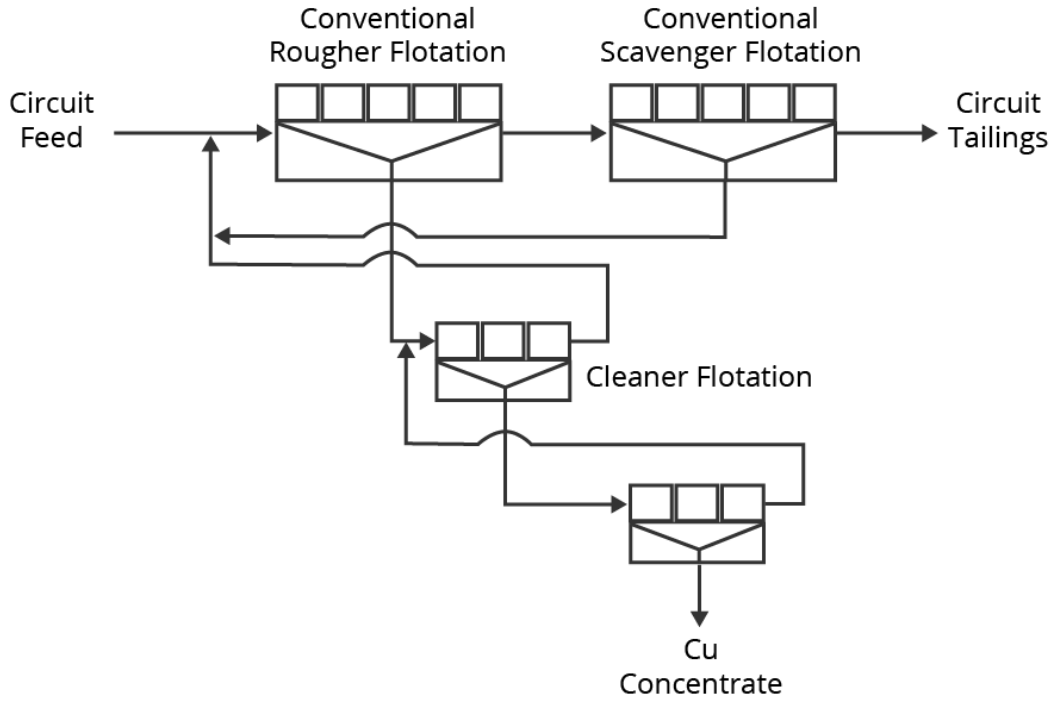
- Most separation units produce two or three products:
  - A concentrate stream in which there is a higher proportion of the valuable material than in the feed
  - A tailings stream with a lower proportion of the valuable material than in the feed
  - Some separation equipment produce intermediate products of better grade than the tailings. We'll be considering the case with 3 product units (such as spiral separators)

The Performance of a Unit

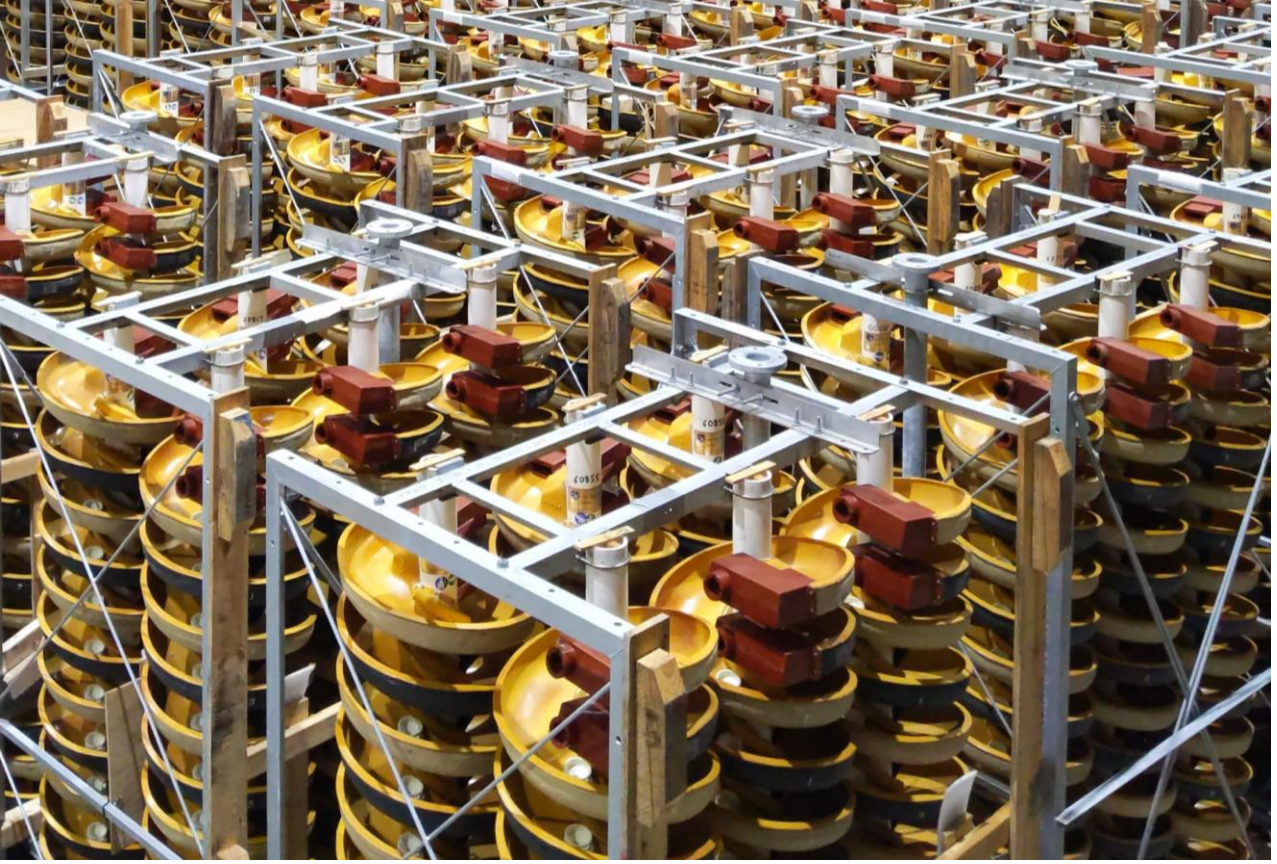
- Performance of a unit will depend on both its operating conditions, as well as the composition and rate of the feed
  - We'll be using a simplified model for the individual units – discussed later
- Problem is that individual units often relatively inefficient
  - Waste remains in the product/concentrate stream
  - Valuable material remains in the tailings stream
- Therefore we usually need many of them in circuits to produce acceptable overall performance

Separation Circuits

- To overcome inefficiency of individual separation units, usually arranged in circuits
  - Reduce amount of valuable material lost to final tailings (i.e dumped).
  - Restrict amount of waste collected to final concentrate, thus improving its purity



Typical flotation circuit layout (note extensive use of recycles)



Spiral Separators

Optimum Separation Circuits

- **Question:** what is best circuit configuration for given number of individual separation units?
  - Can usually increase performance with more units, but comes at an increased capital cost
- Some circuits are unambiguously better than others:
  - Produce a higher purity product at a higher overall recovery rate of the valuable material
- Often there will be compromise between purity and recovery.
  - I.e. circuit might produce a higher recovery, but at a lower purity than another
- Optimum circuit not purely a technical question, but will also depend on the economics
  - How much are you paid for the product vs how much you are penalised for a lack of purity?
  - We'll again work with simplified contracts

How to find an optimum circuit?

Minimisation and Maximisation Problems

- A generic maximisation (or minimisation) problem involves a scalar objective function depending on a (often large) number of input variables for which a maximum/minimum value is needed.
- For objective functions that are continuous functions of the input variables, gradient search methods are often employed
  - E.g. (stochastic) conjugate gradient methods, Newton methods
- For problems where input variables are discrete (e.g. integers) and/or where the objective function is discontinuous, gradient search methods aren't often appropriate (or possible).

Genetic Algorithms

- Our problem is discrete one, in that the variables are the connections between the units
  - Units are either connected to one another or not, thus discrete
- One popular approach to problems with discrete variables are genetic algorithms
  - Note the method can be used with continuous variables (where either some or all of the variables continuous)
  - If all input variables and the objective function are continuous then gradient search methods are likely to perform better (i.e faster).

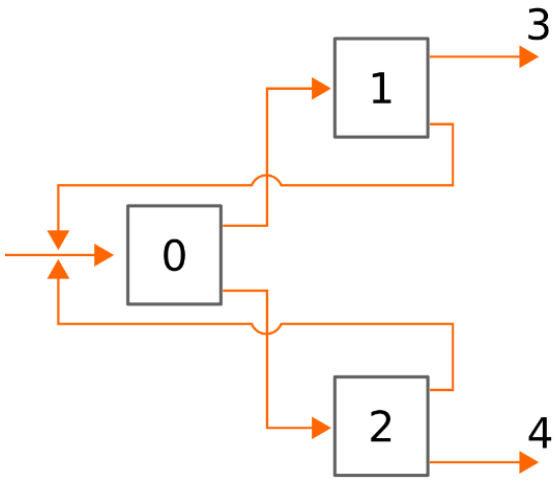
Genetic Algorithms – Representing the Problem

- Genetic algorithms rely on problems represented as vectors of values – “genetic code”

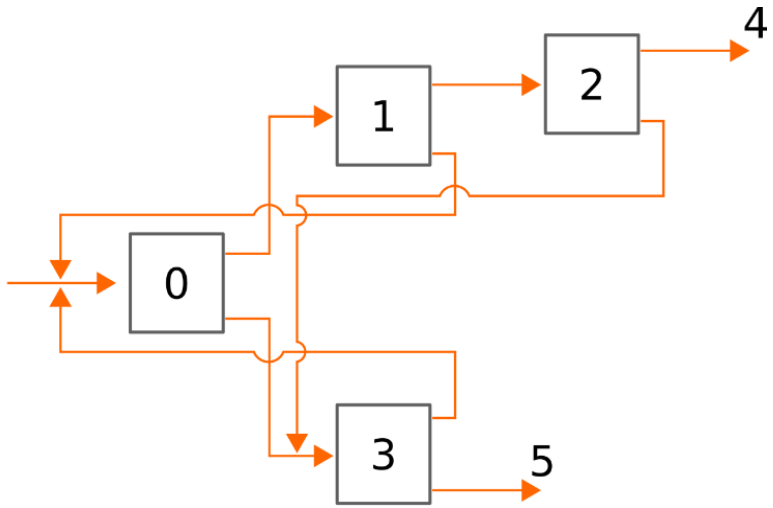
- We can represent a generic circuit by the destination of units’ output streams
  - First item in vector is the destination unit of the input circuit feed
  - Subsequent numbers come in tuple based on number of outputs
    - Destination unit of 1st output
    - Destination unit of 2nd output
    - Destination unit of 3rd, 4th output etc.
- Representation is generic
  - Any circuit can be represented

Examples

- Units with two output



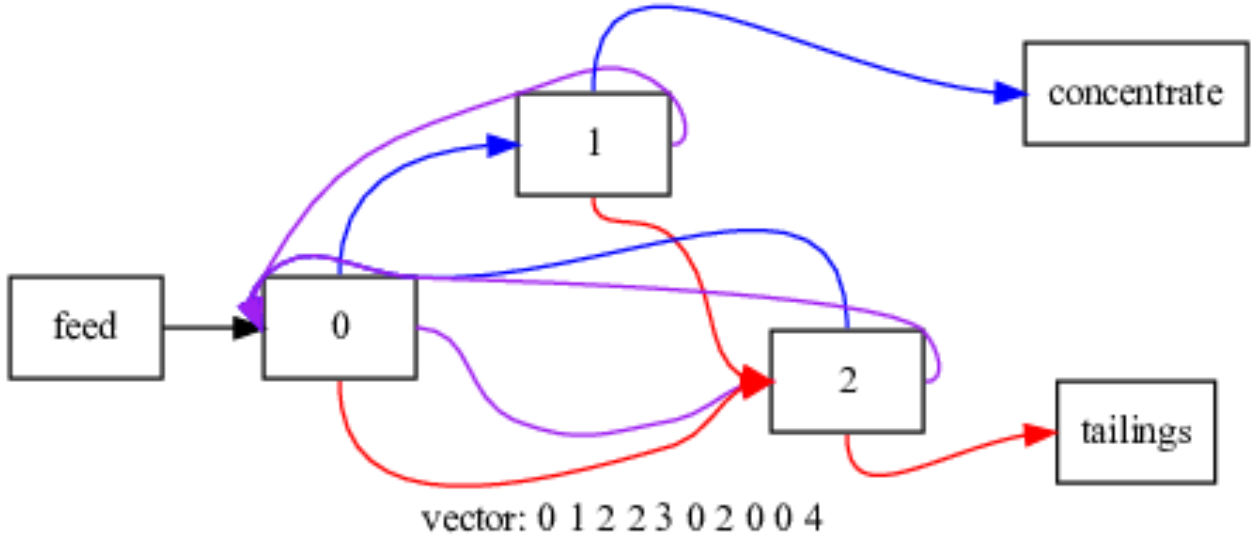
Feed	0		1		2	
0	1	2	3	0	0	4



Feed	0		1		2		3	
0	1	3	2	0	4	3	0	5

Examples

- Units with 3 outputs (you'll be working with units like this)



Genetic Algorithms – A Fitness Value

- Can use circuit vector to calculate performance of a circuit (i.e. flow rates & composition of every stream)
  - Need to know flowrate and composition of circuit feed
  - Need a model for the individual units
  - …We'll discuss how to do this later
- Once we have the circuit performance, we can use it together with information on the economics to produce a single fitness value for a given circuit vector
  - We 'll use a simple fitness function where we're paid for valuable material in the final product stream and penalised for all waste material remaining in this stream

Evaluating Circuits

Need to evaluate any given circuit as a crucial input into the genetic algorithm:

- Check that the circuit is valid
- Calculate flows in all the circuit streams
  - Requires the modelling of the individual units
- Use the product flows to calculate a single performance metric

Note that we will be assuming **steady state** behaviour – Flows don't change with time & no accumulation of material

  - I.e. flows in and out of any section of the circuit must match for **all** components

Modelling the Units

Only two components in the system

- A valuable material (gerardium) and a waste material

We are going to use a first order kinetic model

- What goes in must go out (steady state)
- The amount of material of species i in concentrate/intermediate output stream is proportional to the total amount in the cell
- If the cell is well mixed, this results in the following equations:

$$R_i^1 = \frac{k_i^1 \tau}{1 + \left(k_i^1 + k_i^2\right)\tau} \qquad R_i^2 = \frac{k_i^2 \tau}{1 + \left(k_i^1 + k_i^2\right)\tau}$$

- The residence time is a function of the feed rate:

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

F_i is the mass flow rate of component i in the feed, ρ is the solid density (3000 kg/m^3), φ is the volume fraction solids (0.1) and V is the cell volume (10 m^3)
k_i^n is a rate constant

Modelling the Units

- Once you know the recovery rates for each material you can work out the mass flow rates in the concentrate (C), intermediate (I) and tails (T) streams:

$$C_i = F_i \, R_i^1$$

$$I_i = F_i \, R_i^2$$

$$T_i = F_i - C_i - I_i$$

$$= F_{\mathrm{i}} \left(1 - \sum_n R_i^n \right)$$

Circuit Feed

- To model circuit, need to know the overall input feed rate.
- For your base case we will be using:
  - 10 kg/s valuable material (gerardium)
  - 90 kg/s waste material

Modelling the Circuit

- Because we have recycles the easiest way to calculate the circuit performance is iteratively
  - Successive substitution is guaranteed to work if the circuit is valid – this could be speeded up, but you still need to ensure convergence (for valid circuits)
  - Non-convergence is an indication that the circuit is invalid!
- The algorithm on next few slides will thus work for valid circuits, but could potentially be made quicker.

Modelling the Circuit – An Algorithm

1) Give an initial guess for the feed rate (mass per second) of both components (gerardium and waste) to every cell in the circuit
   - You can guess the same feed rate everywhere (10 kg/s gerardium; 90 kg/s waste)
   - Alternatively recursively fill starting from the feed – Gives instant correct answers if there is no recycle
2) For each unit, use the current guess of the feed (input) flowrate of each component to calculate the output flowrate of each component via the concentrate stream, intermediate stream and the tailings streams
3) Store current value of the feed of each component into each cell as “old” feed values and then set the current value of the feeds for each component to zero.
4) For cell receiving the circuit feed, set the feed of each component equal to the flowrate of the circuit feed: i.e., 10 kg/s for the gerardium feed and 90 kg/s for the waste feed

Modelling the Circuit – An Algorithm (cont.)

5) For current unit:

   Start with concentrate stream. Add flowrates of components in this stream (calculated in step 2) to the flowrate of the relevant component in the feed going into the destination unit (or final concentrate stream) at the end of the concentrate stream, based link in the circuit vector.

   Repeat this procedure for the tailings stream, which will increment the feeds of gerardium and waste to a different unit in the circuit (or the final tailings stream).

6) Move to the next unit and repeat step (5) for each unit in the circuit.

   - Don't need to do this in any particular order as long as each unit is visited once and once only and thus you can simply loop through the list of units in unit number order. After visiting all units, a new estimate for the feed of gerardium and waste into each unit is determined.

7) For each component, 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

8) Based on the flowrates of gerardium and waste through the final concentrate stream calculate a performance value for the circuit.

Circuit Performance Metric

- You will be paid for your valuable gerardium
  - £100 per kg in the final concentrate
- You will be penalised for waste in the product
  - £750 per kg in the concentrate (i.e. a negative value)
- You are not charged for disposal of the tailings

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 a negative number)

Deciding Circuit Validity

- Lack of validity could be assessed using lack of convergence of the circuit flows
  - Computationally very expensive
  - Lack of convergence should still be checked for as there will be some pathological cases not considered in the explicit checks
- Should explicitly check for lack of validity
  - Allows circuits to be rejected before being considered as a child
  - Having only valid parents as the initial set results in much quicker convergence

What is required for circuit validity?

- Every unit must be accessible from the feed
  - I.e. there must be a route that goes forward from one unit to the next from the feed to every unit
- 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 should 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
  - No unit should have itself as the destination for any of its product streams.
- The destination for all products from a unit should not be the same unit.

- Look at the circuits that you decide are invalid through non-convergence and see if you can find additional validity checks

How to traverse the circuit

- For validity checking traversing the circuit via the connections is important
- The circuit takes the form of a directed graph
- The best way to traverse such a graph is recursively

The code on the following slide is a generic function for visiting every unit that can be visited going forward from a given unit

- This can be modified for many of the validity checking requirements

How to traverse the circuit – The stub of a unit class

- Note that the actual class will also need to store the stream information for the feed, concentrate and tails
  - recommend having a stream class that contains the flows of each component
    - This class's operators could then be overloaded to allow streams to be easily added to one another

```
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 concentrate 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

};
```

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

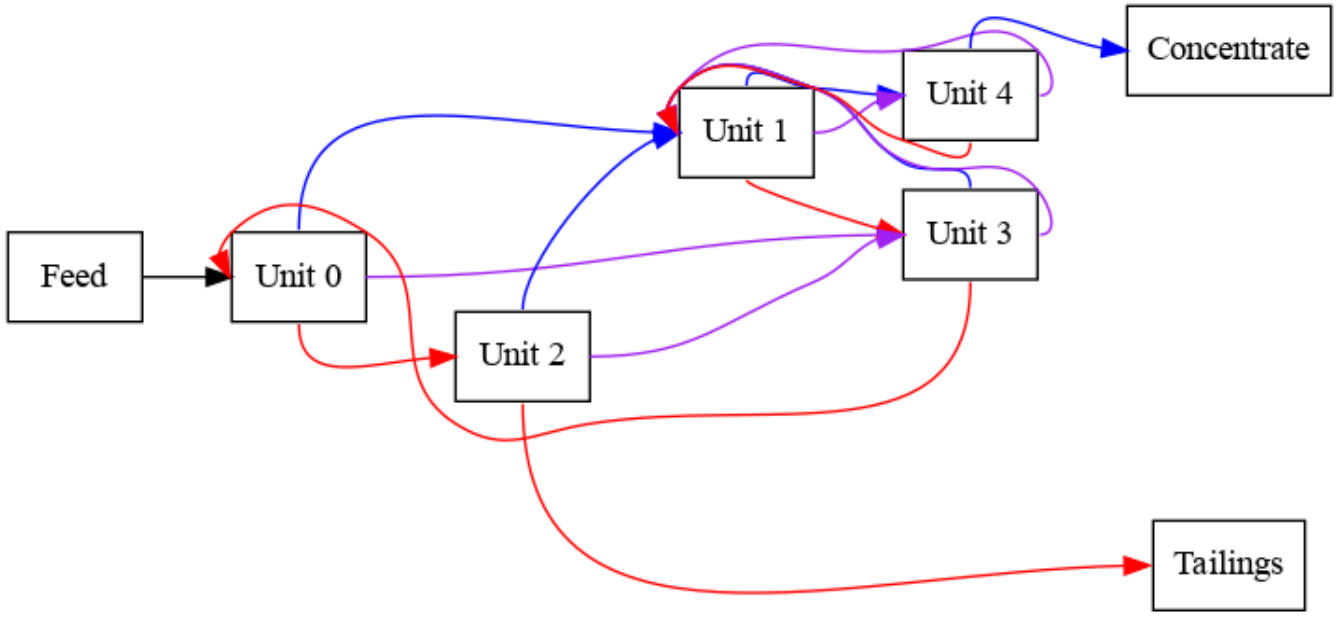
How to traverse the circuit - Recursion

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

    //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
}
```

```
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
```

An Optimum to Test Against

- A circuit with 5 cells, base configuration
  - Note that there are several different vectors that can represent the same circuit (swapping cell numbers does not change the circuit)



Vector: 0 1 3 2 4 4 3 1 3 6 1 1 0 5 1 1
Performance 167.38
Recovery 20.28%
Grade 97.72%

What is required of each team?

- Create software capable of using a genetic algorithm to optimise a system represented by a specification (circuit) 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.
- Write a mass balance simulator which can take a circuit vector and calculate mass flows of each component (gerardium and waste) in every stream.
- Write a validity checking function that takes in a circuit vector and returns true or false based on its assessment of the circuit validity.
- Write post-processing code to visualise the circuit
  - You can use Python for this task and/or use libraries such as graphviz (and, in particular, Digraph). Note that the solution takes the form of a directed graph.

Write all these in a **modular** fashion.

- Obtain the optimum circuit configuration for the base case specifications.

What is required of each team?

Additional tasks – do as many as you are able to

- Investigate genetic algorithm performance
  - How quickly does the algorithm converge on the optimum and how does this change with the genetic algorithm parameters? Note that, as the algorithm is stochastic, the performance can't be evaluated based on a single run – Use the average of a few runs. Also note that the number of runs required to find the global optimum will also depend upon the parameters
- Investigate how the optimum circuit changes as the various model parameters change.
  - Note that you will usually have to do quite 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?
- Parallelise the algorithm
  - The genetic algorithm requires the evaluation of a large number of independent circuit configurations at each iteration which makes it readily parallelised.
  - Shared memory (openMP) parallelisation or distributed memory (MPI) parallelisation

Splitting the Problem

You need to agree your shared data structures first

- All tasks share the circuit specification vector – you have a mathematical form for this
- Circuit modelling and validity checking need to share data structures for the units and their connections – Can base this on the stub in the repos, or make it from new.

Divide into 3 or 4 sub-groups

- Genetic Algorithm development
- Circuit modelling development
- Validity checking development
- Post-processing (i.e visualisation of circuits and data)

Given the number of initially independent, but ultimately interacting tasks, this project will strongly benefit from good code sustainability practices.

Genetic Algorithm team

- Work on optimization algorithm.
- Need to think/research about probability, parameter tuning.
- Iterative refinement process.
- Will have to work closely with the circuit simulator team once that starts to work.

- Work can be hard to test/QA. Consider some easy/deterministic cases.
- Good algorithm implementation is:
  - Fast to run (independent of the simulation/fitness code + validity)
  - Robust (Almost always gets optimum for easy problems, usually gets close for hard ones)

Circuit modelling team

- Needs to build code to calculate on directed graphs
- Balance speed versus accuracy versus robustness.
- Pass product on to GA team
- Work with circuit validity team to implement validity checks

- Lots of opportunities to test/QA strongly.
- Good simulation is:
  - Fast to run (independent of Genetic algorithm code or validity checker
  - Accurate, even for “stiff” problems
  - Modular to changes in physics (with appropriate parameter tuning)
  - Deals appropriately with bad circuits

Circuit validity team

- Need to think about physical realities of problem + heuristics.
- Strong overlap with circuit simulator team:
  - for coding
  - for examples of bad circuits

  & visualisation team:
  - To identify patterns in bad circuits
- Provide functionality to Genetic Algorithm team

- Good code identifies invalid/bad circuits as quickly as possible.
- Make sure to document any novel approaches.

Post-processing team

- More choice of programming language (C++/Python, or even something else)
- Need to provide circuit visualisation tools to the circuit validity team
- Otherwise, most opportunity to work independently (if you choose)
- Alternatively can couple more strongly to the other teams and show additional data.

- Good outputs explain the connectivity of the circuit and the flow patterns inside it in easy to parse ways.
- Similarly demonstrate other data arising from your code in ways which will be useful in your final video presentation.

Developing sections independently

Link the sections by having, agreed function forms, e.g:

- Circuit simulation
  - Take in a circuit and give back a single performance value

```
double Evaluate_Circuit(int vector_size,
                        int *circuit_vector, …)
```

- Circuit validity
  - Take in a circuit and return true or false based on validity

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

Test these functions using a few different valid and invalid vectors

- Draw a few circuits and write out the corresponding vectors by hand

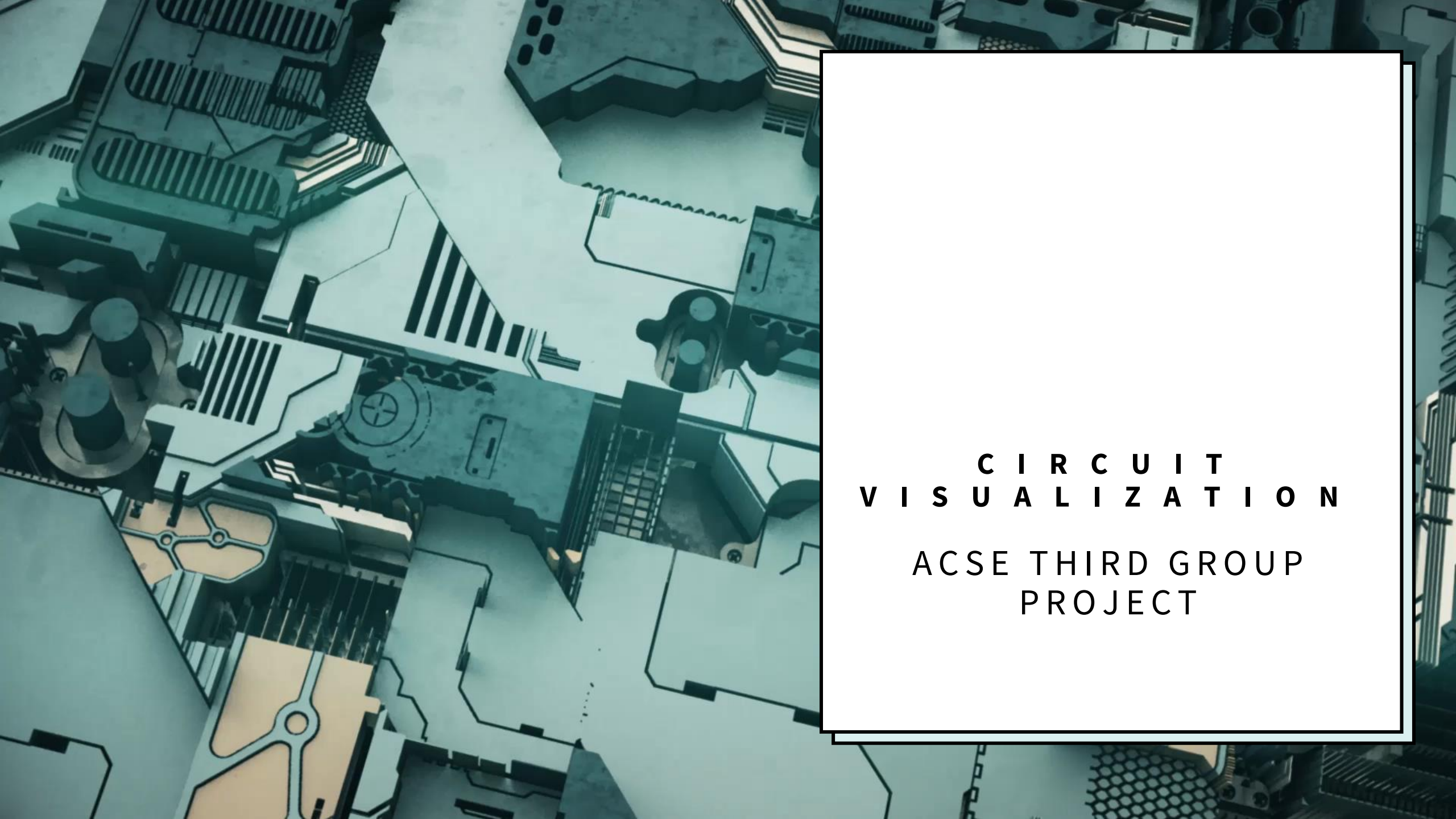
Form of full genetic algorithm functions determined by the main loop of your executable

- Remember code modularity!

Developing the Genetic Algorithm

- For the development of the genetic algorithm you need simple test versions of the other sub-group's problems
  - You can use something like the following – Note that this is a dramatically simpler problem than the real one and so only use it to test that the algorithm is working, or to measure raw GA speed, not to optimise/tune hyperparameters/operations

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


double Evaluate_Circuit(int vector_size, int *circuit_vector, …)
{
        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;
}
```

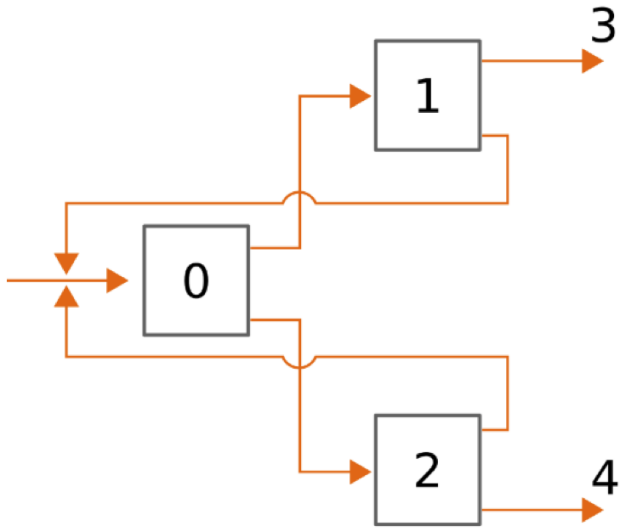


CIRCUIT VISUALIZATION

ACSE THIRD GROUP PROJECT

Circuits are vectors, circuits are graphs

- Circuits are encoded as integer vectors/arrays
- Circuits can be visualized as directed graphs
- Units are nodes, input/output streams are edges



Feed	0		1		2	
0	1	2	3	0	0	4

Directed graphs

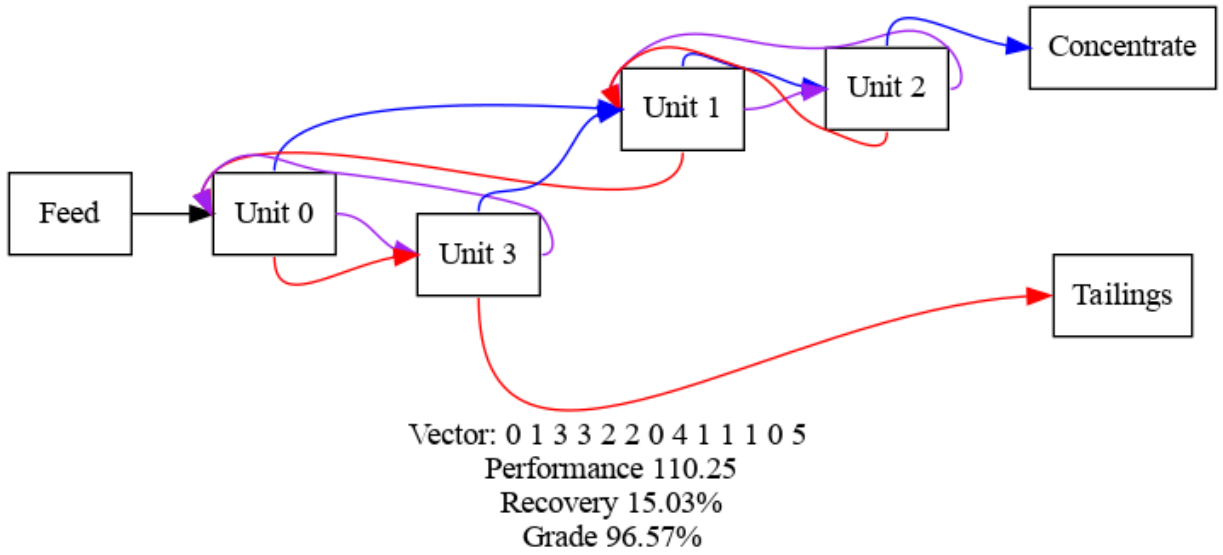
Examples of use:

- Flow charts
- Electrical circuit diagrams
- Dependency graphs

Possible free programs:

- dia - draws structured diagrams
- inkscape - vector graphics package
- graphviz - directed graph library

Plenty of others out there.



Graphviz

Graphviz is low effort to install

- Via conda:

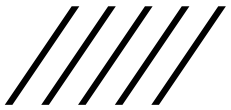
```
conda install graphviz python-graphviz
```

- via homebrew (mac only)

```
brew install graphviz
```

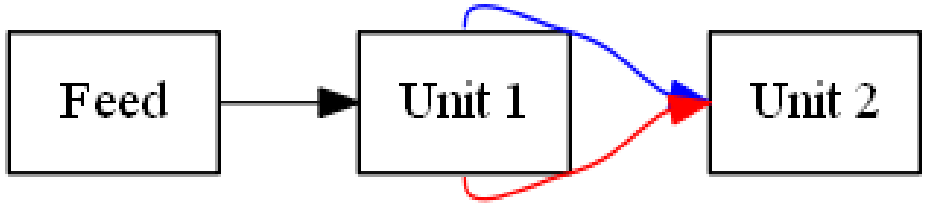
```
pip install graphviz
```

- via apt (Ubuntu/Debian linux & WSL)
  - `apt install graphviz libgraphviz-dev python-pygraphviz`



Using Graphviz

Graphviz builds graph by describing the edges (and optionally the nodes)

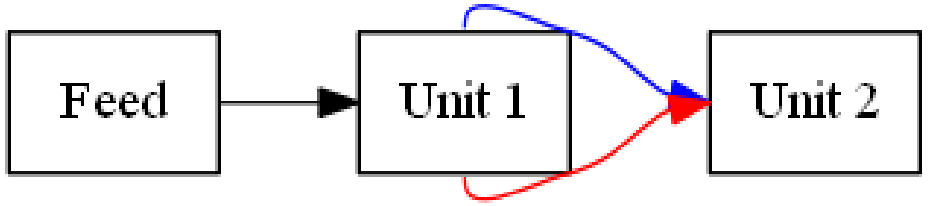


See the Python wrapper documentation for more options and examples.

```
import graphviz
graph = graphviz.Digraph()
graph.attr(rankdir='LR')
graph.attr('node', shape='rectangle')
graph.edge('Feed', 'Unit 1',
           color='black',
           headport='w', tailport='e',
           arrowhead='normal')
graph.edge('Unit 1', 'Unit 2',
           color='blue',
           headport='w', tailport='n',
           arrowhead='normal')
graph.edge('Unit 1', 'Unit 2',
           color='red',
           headport='w', tailport='s',
           arrowhead='normal')
graph.render('example', cleanup=True,
             format='png')
```

Using Graphviz

Graphviz builds graph by describing the edges (and optionally the nodes)



See the C documentation for more options and examples.

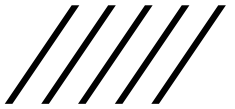
```c
#include <graphviz/gvc.h>

int main(int argc, char** argv) {
    Agraph_t* g;
    GVC_t* gvc;
    Agedge_t *edge1, *edge2, *edge3;
    Agnode_t *node1, *node2, *node3;

    char* args[4];
    int i;


    args[0] = "dot";
    args[1] = "-Tpng";
    args[2] = "-o";
    args[3] = "example.png";

    gvc = gvContext(); /* library function */

    gvParseArgs(gvc, 4, args);
    g = agopen("g", Agdirected, 0);

    agsafeset(g, "rankdir", "LR", "");
    agattr(g, AGNODE, "shape", "rectangle");

    node1 = agnode(g, "Feed", 1);
    node2 = agnode(g, "Unit 1", 1);
    node3 = agnode(g, "Unit 2", 1);

    edge1 = agedge(g, node1, node2, 0, 1);

    edge2 = agedge(g, node2, node3, 0, 1);
    agsafeset(edge2, "color", "blue", "");
    agsafeset(edge2, "tailport", "n", "");
    agsafeset(edge2, "headport", "w", "");

    edge3 = agedge(g, node2, node3, 0, 1);
    agsafeset(edge3, "color", "red", "");
    agsafeset(edge3, "tailport", "s", "");
    agsafeset(edge3, "headport", "w", "");

    gvLayoutJobs(gvc, g);
    /* Write the graph according to -T and -o options */
    gvRenderJobs(gvc, g);

    gvFreeLayout(gvc, g); /* library function */
    agclose (g); /* library function */
    gvFreeContext(gvc);
}
```

Using Graphviz inside C++

- Relatively simple on mac/linux
  - Install development package with headers/library using brew/apt
  - Point CMake at the right location.
  - Write your code, #including the relevant headers
- On Windows, easiest to download and build from Graphviz source

```
git clone –recurse-submodules https://gitlab.com/graphviz/graphviz.git
```

  - Add graphviz\windows\dependencies\graphviz-build-utilities\ to your path
  - Open graphviz.sln in Visual Studio and build X86/Win32 version.
  - Then set up Cmake appropriately



Good Circuit Visualisations

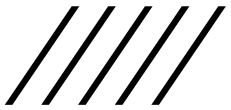
In order of importance, good circuit visualizations:

- Show the layout of the circuit clearly
- Includes information on optimality/performance.
- Includes information on circuit flow data.

Can have more than one plot.

Other data visualisations

- Lots of other interesting things to plot:
  - Change in optimum performance with economic conditions
  - Circuit heuristics (what do your individual units do? Clean/scavenger etc.)
  - Convergence plots for your genetic algorithm
  - Key changes in your circuit design as parameters vary
- **Don’t** just animate your whole route to convergence
  - it’s not all that interesting and often hard to follow if you don’t relabel units
- **Do** aim towards your final presentation



Assessment

- Breakdown as with previous group projects –

  See full problem statement/module handbook for more details

Software	(70 marks)
- Functionality and performance	(50 marks)
- Genetic algorithm	(10 marks)
- Mass balance calculator	(10 marks)
- Validity checking	(10 marks)
- Post processing	(10 marks)
- Additional features and optimisations	(10 marks)
- Sustainability (tests/documentation, etc)	(20 marks)
Video Presentation	(20 marks)
Individual contribution mark	(10 marks)

Your Presentation

Presentation due 4pm on Friday along with documentation/repo work

- Code (i.e. .cpp/.h files) due by 12 noon

Your group must produce a 15 minute video on your project

Presentation should be aimed at the client who has asked you to find the best design for their plant

Your presentation should provide the following information:

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

No need to rehash this briefing

List both positives and negatives

Randoms in parallel

- Modern best practice:
  - *cstdlib*'s *rand*() function can't be assumed thread safe, and is device dependent
    - Visual Studio version in particular is quite bad
    - Modern gnu version is much better, but has locks
  - Best to use a distribution in STL <*random*>, no need to roll your own
  - Initialise generators/engine with a *random_device* call
    - Very expensive, so ideally only do it once (per thread)
  - Fastest safe way is to set up (once, i.e. statically/globally) a generator per thread. Try to make many calls to the same generator.
  - May want a deterministic debug mode!
  - Let's go to the repo

Questions…