Global Optimisation Methods

Minimisation and Maximisation Problems

- A generic maximisation (or minimisation) problem involves having an objective function that is dependent on a number of input variables for which a maximum (or minimum) value is required
- For objective functions that are continuous functions of the input variables, gradient search methods are often employed
    - E.g. conjugate gradient methods
- For problems where the input variables are either discrete and/or the objective function is discontinuous, gradient search methods are not appropriate and global methods can be used instead

Global methods

These methods can be classified into *exact methods* and *heuristics*.

An exact (or deterministic) method guarantees to find and verify global solutions. Otherwise, it is called a heuristic.

Heuristics try to find global solutions without verifying global optimality. They can be used as stand-alone solvers or as an acceleration tool in deterministic methods.

Examples of heuristic search strategies:

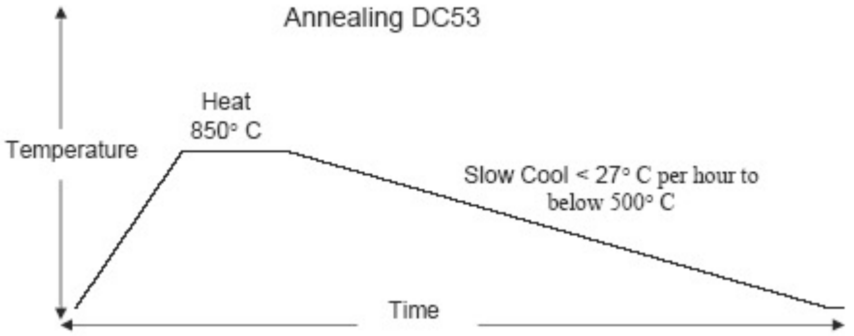
- Simulating annealing
- Memetic algorithms
- Particle swarm optimisation
- Genetic algorithms

Simulated annealing

Annealing is a metallurgical process in which a material is heated above its recrystallization temperature for a certain amount of time, and then is slowly cooled.

The slow cooling brings the metal to a crystalline state, altering their physical/chemical properties to increase its ductility and reduce its hardness.

In *simulated annealing* points of a sample set are modified by applying a descent step with a random search direction and an initially large step-size that is gradually decreased during the optimisation process.



Typically used in discrete, but very large, configuration spaces.

Simulated annealing

At each iteration, a new point is randomly generated. The extent of the search is based on a probability distribution (with a scale proportional to the "temperature").

Importantly, the algorithm accepts new points that e.g. maximise the objective function but also, with a certain probability, points that do not, in order to avoid being trapped in local maxima.

An *annealing schedule* is selected to systematically decrease the temperature as the algorithm proceeds. As the temperature decreases, the algorithm reduces the extent of its search to converge to a maximum.

Simulated annealing

1. Initialize the system configuration.
   Randomize $\boldsymbol{x}(0)$.
2. Initialize T with a large value.
3. **Repeat**:
   a. **Repeat**:
      i. Apply random perturbations to the state $\boldsymbol{x} = \boldsymbol{x} + \Delta\boldsymbol{x}$.
      ii. Evaluate $\Delta E(\boldsymbol{x}) = E(\boldsymbol{x} + \Delta\boldsymbol{x}) - E(\boldsymbol{x})$:
         if $\Delta E(\boldsymbol{x}) < 0$, keep the new state;
         otherwise, accept the new state with probability $P = e{-}\Delta E/T$.
      until the number of accepted transitions is below a threshold level.
   b. Set $T = T - \Delta T$.
until T is small enough.

Note that a simulated annealing algorithm run multiple can result in different local minima

→ a problem needs to be run several times before the solution can be accepted as the global optimum.

Memetic algorithms

These are evolutionary algorithms capable of performing local refinements. Based on the concept of evolution (similar to Genetic Algorithms, which we'll discuss later) but rather than biological evolution, it is the evolution of a concept that is applied.

Memes, building blocks of meaningful information, are the social analogue of genes and can be thought of as schemata.

The typical memetic algorithm uses an additional mechanism to modify schemata and that refinement can be passed on to an individual's offspring.

Memetic algorithms

In Canonical Memetic Algorithms (CMA), the notion of memes is limited to mathematical procedures that serve as local search schemes which are subsequently hybridized with some population-based stochastic global optimiser.

1. **Initialize**: Generate initial population
2. **repeat**
3.   **for** each individual $\boldsymbol{x}_i \in$
4. $fi \leftarrow$ Evaluate $\boldsymbol{x}_i$
5.   **if** $\boldsymbol{x}_i$ is chosen for local search
6. $\boldsymbol{x}_{mod}, f(\boldsymbol{x}_{mod}) \leftarrow \textit{Local Search}(\boldsymbol{x}_i)$
7. $\boldsymbol{x}_i, f_i \leftarrow \textit{Update}(\boldsymbol{x}_i, f_i, \boldsymbol{x}_{mod}, f(\boldsymbol{x}_{mod}))$
8.   **end if**
9.   **end for**
10. Apply global optimiser operators, *eg generational selection*, *crossover*, and *mutation*, on current population to create the next generation
11. **until** *termination criterion of CMA met*

Particle swarm optimisation

It uses a simple mechanism that mimics swarm behaviour (eg birds flocking, fish schooling) to guide the particles to search for global optimal solutions.

- Useful for the optimisation of irregular problems that are noisy and change over time.
- It evolves populations or swarms of individuals → particles.

It finds the global best solution by adjusting the moving vector of each particle according to its personal best and the global best positions of particles in the entire swarm at each iteration.

Relevant parameters:

- number of particles
- position of agent in the solution space
- velocity of agents
- neighbourhood of agents

Particle swarm optimisation

Basic algorithm:

1. Set $t = 1$.
   Initialise each of the N_P particles in the population by randomly selecting values for its position x_i and velocity $\boldsymbol{v}_i$, $i = 1, \ldots, N_P$.
2. **Repeat**:
   a. Calculate the fitness value of each particle i .
      If the fitness value for each particle i is greater than its best fitness value found so far, $\boldsymbol{x}^*_i$ then revise $\boldsymbol{x}^*_i(t)$.
   b. Determine the location of the particle with the highest fitness and revise the global best, $\boldsymbol{x}^g(t)$, if necessary.
   c. **for each** particle i, calculate its velocity
   d. Update the location of each particle i
   e. Set $t = t + 1$.
   until stopping criteria are met.

$$\begin{cases} \boldsymbol{v}_i(t+1) = \boldsymbol{v}_i(t) + cr_1\left[\boldsymbol{x}_i^*(t) - \boldsymbol{x}_i(t)\right] + cr_2\left[\boldsymbol{x}^g(t) - \boldsymbol{x}_i(t)\right], \\ \boldsymbol{x}_i(t+1) = \boldsymbol{x}_i(t) + \boldsymbol{v}_i(t+1), \quad i = 1, \ldots, N_P, \end{cases}$$

Particle swarm optimisation

Computationally inexpensive in terms of both memory requirements and speed.

It can locate the region of the optimum fast, but once in this region the progress is slowly (due to the fixed velocity step size), which is an aspect that different variants of the algorithm try to address.

Genetic Algorithms

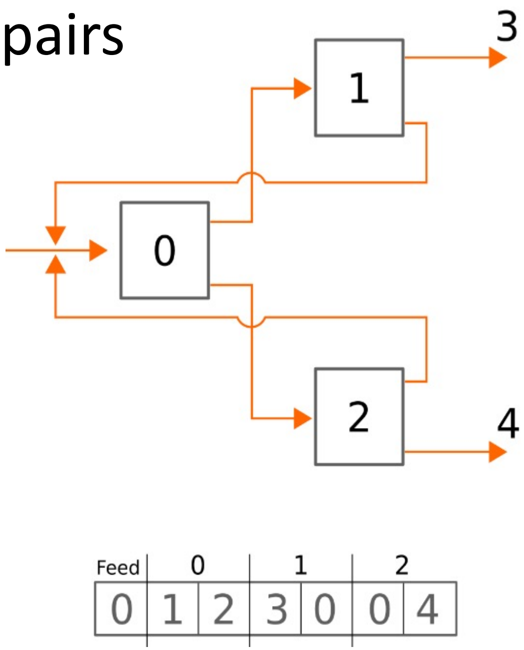
Genetic Algorithms

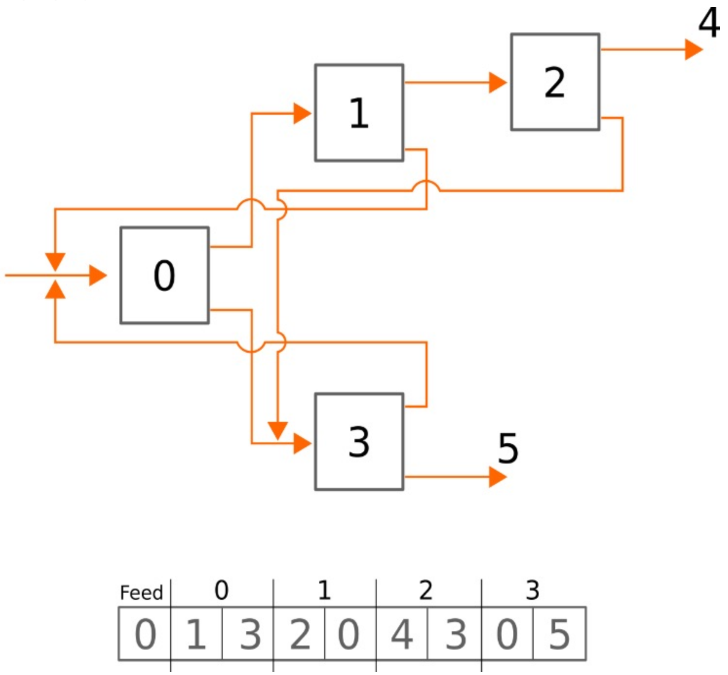
A popular approach to problems with discrete variables are genetic algorithms

- Note that the method can be used with continuous variables (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

Genetic Algorithms – Representing the Problem

- Genetic algorithms rely on problems being represented as a vector of values – a “genetic code”
- We can represent a circuit by the destination of its two product streams
  - First item in the vector is the destination of the circuit feed
  - Subsequent numbers come in pairs
    - Destination of concentrate
    - Destination of tailings





- Representation is generic
  - Any circuit can be represented

Genetic Algorithms – A Fitness Value

- We can use a circuit vector to calculate the performance of a circuit (flowrates and composition of every stream)
  - Need to know flowrate and composition of circuit feed
  - Need a model for the individual units
  - …We will 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 will use a simple fitness function in which we are paid for the valuable material in the final product stream and penalised for the waste material in this stream

Genetic Algorithms – Producing Children

- The heart of a Genetic Algorithm is the production of “child” vectors from a list of “parent” vectors
- Done using two operations:
  - ***Crossover***
    - This is roughly equivalent to sexual reproduction.
    - A portion of one parent vector is swapped with a portion of another parent vector to produce two child vectors
    - 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)
  - ***Mutations***
    - Random changes in the numbers in the vector.

Genetic Algorithms – Setting Up

Starting the calculations –

- We need to start with a list of random parent vectors
  - How many is best is a tuning parameter – I recommend trying it with around 100 parents, but test for what is best
    - It will depend on both the type and size of problem being optimised
- These vectors should all be for valid circuits
  - Many vectors will result in circuits that are invalid for various reasons
    - We will discuss this in detail later

Genetic Algorithms – The Calculation Steps

The main calculation loop is 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
   - Probability of selection 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.

Genetic Algorithms – The Calculation Steps (cont.)

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
   - As there is no reason why a connection to a unit with a similar number should be favoured you should choose a completely random new number in this step for the actual simulation
7) Check the validity of each of these potential new vectors and, if they are valid, 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).

Tuneable Parameters

There are a few parameters you can tune in a genetic algorithm (the recommended parameter are for the main problem):

- The number of offspring n that are evaluated in each generation
  - Try values of order 100
- The probability of crossing selected parents
  - Try values between 0.8 and 1.0
- The rate at which mutations are introduced
  - Values should be less than 1% per item in the vector

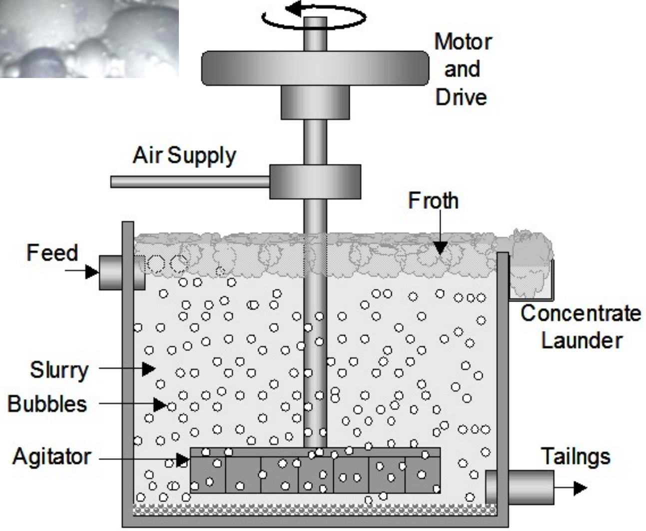
Genetic Algorithms

Mini Project

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 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 can produce intermediate products, but we will not be considering these

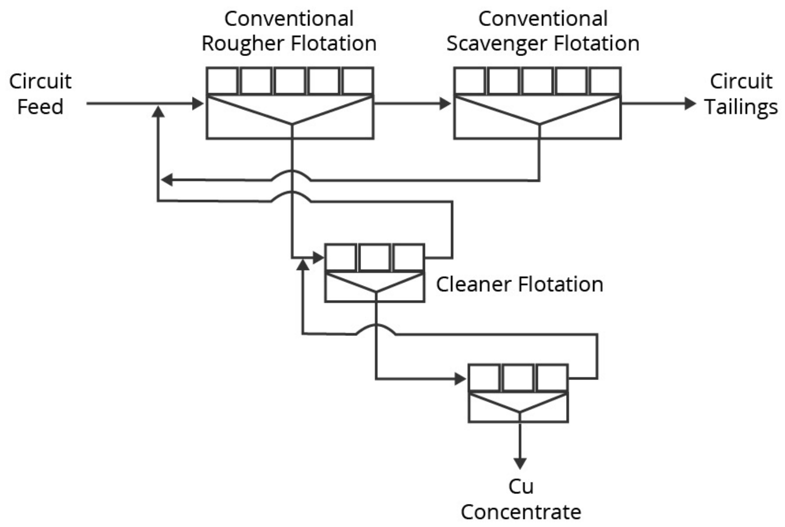


The Performance of a Unit

- The performance of a unit will generally depend on both its operating conditions, as well as the composition and rate of the feed
  - We will be using a very simplified model for the individual units – discussed later
- The problem is that individual units are often relatively inefficient
  - Waste in the product stream
  - Valuable material in the tailings stream
- We therefore need to use many of them in circuits to produce an acceptable overall performance

Separation Circuits

- To overcome the inefficiency of individual separation units, they are usually arranged in circuits
  - Reduce the amount of valuable material lost to the final tailings
  - Restrict the amount of waste collected to the final concentrate and thus improving its purity



Typical flotation circuit layout (note the extensive use of recycles)

Optimum Separation Circuits

- The question we want to ask is what is the best circuit configuration for a given number of individual separation units?
  - We can usually increase performance by the appropriate use of more units, but this will come at an increased capital cost

- There are some circuits that will be unambiguously better than others
  - Produce a higher purity product at a higher overall recovery of valuable material

- Often though there will be a compromise between purity and recovery
  - I.e. one circuit might produce a higher recovery than another, but at a lower purity

- The optimum circuit will thus not be 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?

How to find an optimum circuit?

Evaluating Circuits

We need to be able to evaluate a given circuit as a crucial input into the genetic algorithm:

- Check that the circuit is valid
- Calculate the flows in all the circuit streams
    - Requires 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 and no accumulation of material

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

Modelling the Units

We are going to use a very simple unit model

- Only two components in the system
  - A valuable material and a waste material
- Assume no impact of feed rate on performance
- Fraction of each component reporting to the concentrate is the same in each unit
  - 20% of valuable material (Gormanium) in the feed reports to the concentrate
  - 5% of waste material in the feed reports to the concentrate
  - The rest of the feed reports to the tailings stream

Circuit Feed

- To model a circuit we need to know what the overall feed rate is
- We will be using:
  - 10 kg/s valuable material (Gormanium)
  - 100 kg/s waste material
- Note that because our unit performance does not depend on the feed rate, the optimum circuit will only depend on the feed purity, not the total feed rate
  - This is not always the case as unit performance will more usually depend on the feed rate

Modelling the Circuit

- Because we have recycles the easiest way to calculate the circuit performance is iteratively
  - Our very simple unit models are linear and thus we could use matrix inversion for this particular problem – not applicable to more complex unit models
  - Successive substitution is guaranteed to work if the circuit is valid – this could be speeded up, but still need to ensure convergence

- The algorithm on the coming slides will thus work for valid circuits, but could potentially be made quicker
  - You have been given a simple implementation of this algorithm

Modelling the Circuit – An Algorithm

1) Give an initial guess for the feed rate (mass per second) of both components (Gormanium and waste) to every cell in the circuit
   - You can guess the same feed rate everywhere (10 kg/s Gormanium; 100 kg/s waste)
   - Alternatively recursively fill starting from the feed – Gives instantly 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 both the concentrate and the tailings streams
3) Store the 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 the 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 Gormanium feed and 100 kg/s for the waste feed

Modelling the Circuit – An Algorithm (cont.)

5) For the current unit, consider first the concentrate stream. Add the flowrates of the 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 on the linkages in the circuit vector. Repeat this procedure for the tailings stream, which will increment the feeds of Gormanium 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.
   - You do not 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 Gormanium 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 Gormanium and waste through the final concentrate stream calculate a performance value for the circuit.

Circuit Performance Metric

- You will be paid for your valuable Gormanium
  - £100 per kg in the final concentrate
- You will be penalised for waste in the product
  - £500 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 either of the two product streams.
- The destination for both products from a unit should not be the same (different) unit.

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 - 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
}
```

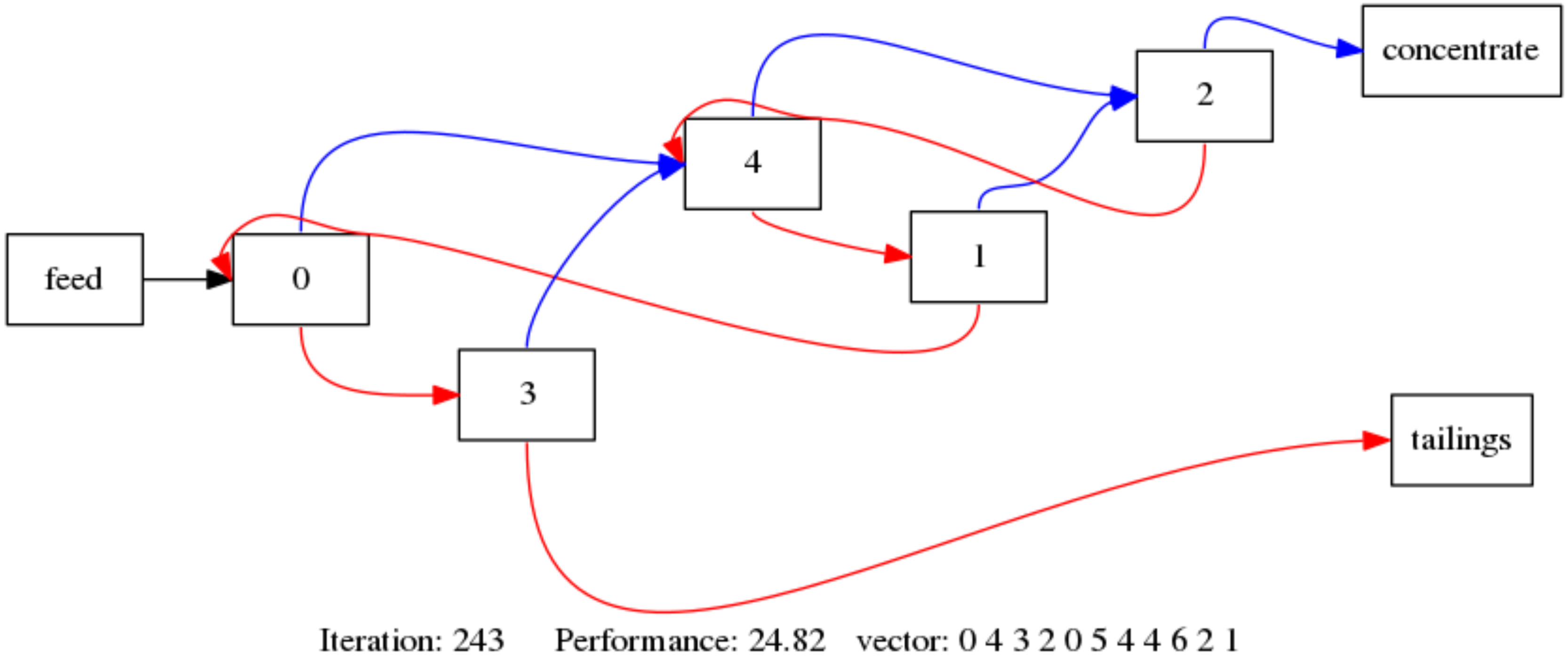
```
for (int i=0;i<num_units;i++)
    units[i].mark = false;

//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
```

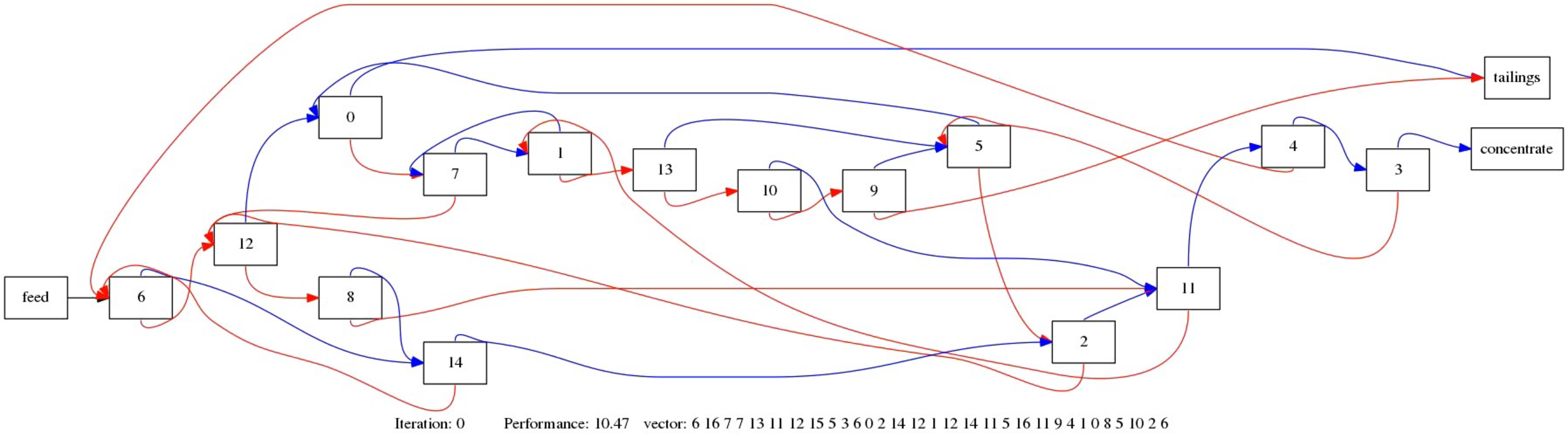
An Optimum to Test Against

- A circuit with 5 cells
  - Note that there are different vectors that can represent this circuit (swapping cell numbers does not change the circuit)



Examples of my Solutions

- Evolution of the best circuit as the Genetic Algorithm converges



What is required for the coursework?

- 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.
- You have been supplied with a simple circuit simulator code. You can improve and expand this code.
- There is similarly a simple validation implementation which can be expanded upon

Write these in a modular fashion

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

What is required of for the coursework?

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.
- 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