ACSE4 – Genetic Algorthim Optimization of Separation Unit Circuits

By Team Platinum:

Yusuf Falola, Xianzheng Li, Keer Mei, Sanaz Salati, Mingrui Zhang

TABLE OF CONTENTS

INTRODUCTION

The optimization of industrial separation units relies on arranging them in specific circuit configurations to maximize the yield of a valuable material and reduce the amount of waste in the concentrate stream.

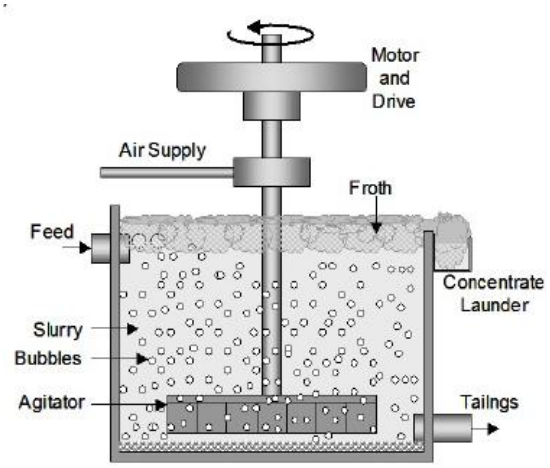


Figure 1. An example of a separation unit

As the number of units increase, the possible number of circuit configurations exponentially increases. In order to efficiently identify the optimum circuit configuration, Team Platinum has implemented a genetic algorithm that mimics the Darwinian reproduction cycle of survival of the fittest.

GENETIC ALGORITHM

The genetic algorithm is a simulation of natural evolution. Here, it is implemented to generate the optimal circuit configurations. There are 3 input parameters in our algorithm:
population(num_parents), mutation rate (gene_change_rate) and crossover rate(crossover_rate).

The pipeline of the algorithm is highlighted below:

a) Generate valid initial parents and set parameters
b) Run the circuit simulator and evaluate performance of the vectors
c) Calculate the probability distribution and push the best vector to next generation
d) Select parent pairs and decide whether to do crossover according to crossover rate or keep the pairs unchanged
e) Do mutation on the children vectors
f) Check the new children, if valid go to next step; if not, regenerate children
g) Repeat steps b->f until stop criterion (meet the expected performance or meet the maximum iteration steps)

The graph below details the overall code structure:

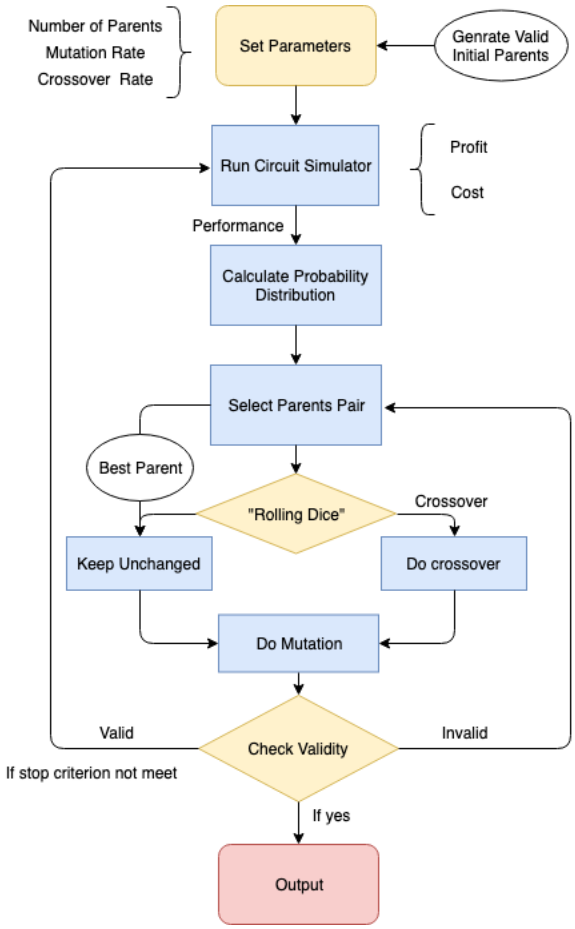


Figure 2. Genetic Algorithm process flow

DEVELOPMENT OPTIMIZATION

There were several versions of the genetic algorithm code throughout the development process. The optimizations that we have added throughout the development process include:

a) Initializing the heap memory only once and not inside a loop
b) Using resize instead of push_back on vectors. This is because the memory from resize is continous as opposed to using push_back. If the elements added to the vector using pushback exceed the reserved capacity, it will request a new larger chunk of memory and copy the old data into the new memory, an onerous and time consuming step in and of itself
c) Reduced the use of nested indices in arrays and vectors. The nested indices require mulitple iterations of address seeking which is inefficient. As much as possible, the majority of the indices throughout the codeare explicit

CIRCUIT SIMULATION

The circuit simulation evaluates a vector's (or circuit's) overall performance by calculating the amount of germanium and waste in the circuit's concentrate stream. To accomplish this calculation, Team Platinum has created a class called CUnit to represent a vector (of length 2n + 1) by an equivalent number of separation units (n + 2), with the last two being the concentrate and tailings streams, respectively.

VISUAL REPRESENTATION

Example Vector: { 0, 1, 2, 3, 0, 0, 4}

CUnit vector representation:

CUnit #0 Conc_num : 1 Tails_num: 2	CUnit #1 Conc_num : 3 Tails_num: 0	CUnit #2 Conc_num : 0 Tails_num: 4	CUnit #3 (Concentrate)	CUnit #4 (Tailings)

The CUnit class tracks the amount of germanium and waste feed in the feed, concentrate and tailings stream of each separation unit. By using a vector of this CUnit class in the evaluate_circuit function, it allows efficient iteration through the entire circuit. Through successive substitution, the evaluate_circuit function calculates the final ratio of germanium and waste in the concentrate stream.

CHALLENGES

The challenge of quantifying a circuit's performance is that randomly generated circuits by the genetic algorithm, despite being valid, may be divergent (the mass balance will never converge). To overcome this scenario, the evaluate_circuit function has set two criterion:

1. First, the amount of germanium and waste in all CUnits (except the last two) must be within a specified tolerance (1e-6 kg/s) between iterations.
2. If the first criteria is not met, then the function will stop after a max number of iterations and the performance of this diverging circuit will be returned as the worst performance (assuming the total amount of waste in the feed flows into the concetrate CUnit and no germanium exists)

Due to the difficulty of predetermining whether a circuit will converge or diverge, the second criteria ensures that diverging circuits have the lowest possible performance and therefore significantly reduce the possibility of diverging circuits surviving to the next iteration of the genetic algorithm.

OUTPUT

Finally, the overall performance of a circuit is calculated by the amount of profit multiplied by the amount of germanium subtracted by the cost multiplied by the amount of waste in the concentrate unit (CUnit #3 in the example above):

$$Performance\ (aka\ Revenue) = Profit_{\$/kg} \times Germanium_{\ kg/s} - Cost_{\$/kg} \times Waste_{\ kg/s}$$

In the problem specification, the initial profit is assumed to be $100/kg and the cost of waste in the concentrate is assumed to be $500/kg.

VALIDITY CHECKING

Checking circuit validity is an essential step of the genetic algorithm that ensures randomly generated parent vectors and future children vectors are feasible and performant circuits.

Validity checking relies on a recursive function called mark_units and the CUnit class mentioned previously. A vector<CUnit> of size n+2 is created to represent a circuit vector of 2n+1. In the first iteration through the circuit vector, each CUnit has their member variable mark initially set as false. The mark_units function will then go through the circuit vector, find the corresponding CUnit object, mark them as true and recursively enter that CUnit's concentrate unit and tailings unit. The mark_units function should therefore mark all of the CUnits as true, unless an invalid vector is present which indicates that not all of the available separation units are represented in the circuit vector.

Additionally, the check_validity function will also test whether a CUnit has both outlet streams going into the same unit and if both outlet streams are represented by the circuit vector.

DIVERGENCE

Despite the validity check, there are sporadic scenarios when the generated circuit vectors are valid yet diverging. This means that their mass balance will not converge into a final value. For such scenarios, the evaluate_performance function assigns the lowest possible performance value to these diverging circuits to minimize the possibility of their emergence in the next generation of parent vectors.

ALGORITHM DESIGN OPTIMIZATION AND SOLUTIONS

Given the following set of initial parameters:

Parameter	Value
Germanium in feed (kg/s)	10
Waste in feed (kg/s)	100
Profit of Germanium ($ / kg)	100.0
Cost of Germanium ($ / kg)	500.0

Team Platinum has found the optimal circuit configurations for 5, 10 and 15 separation units.

5 Unit Circuit Solution

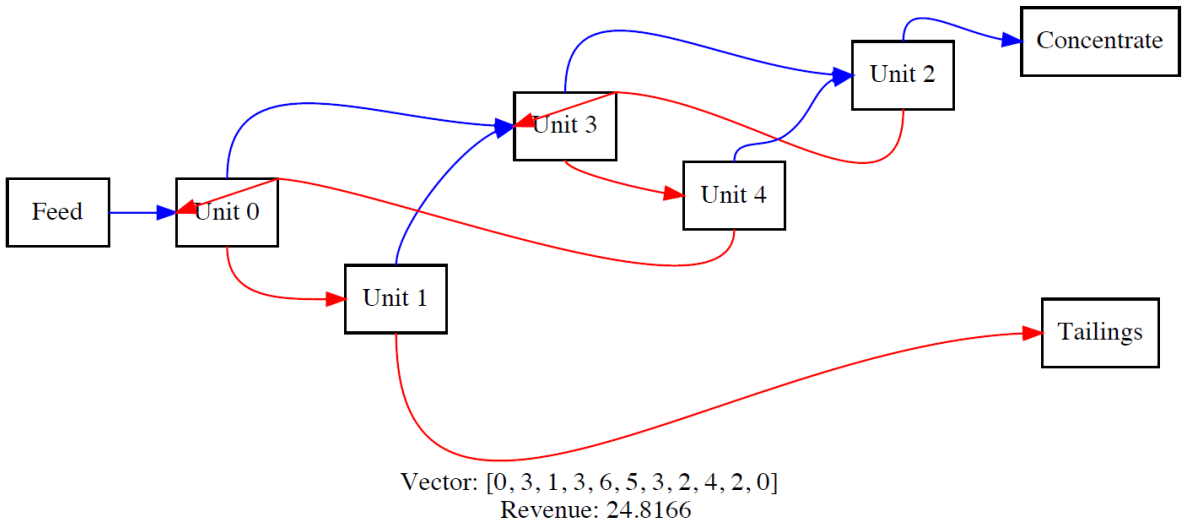


Figure 3. Solution circuit vector for 5 separation units

10 Unit Circuit Solution

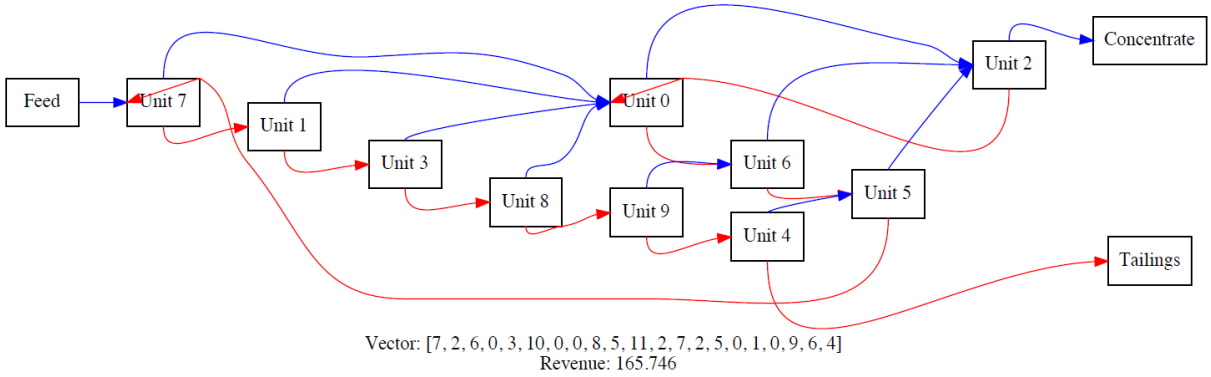


Figure 4. Solution circuit vector for 10 separation units

15 Unit Circuit Solution

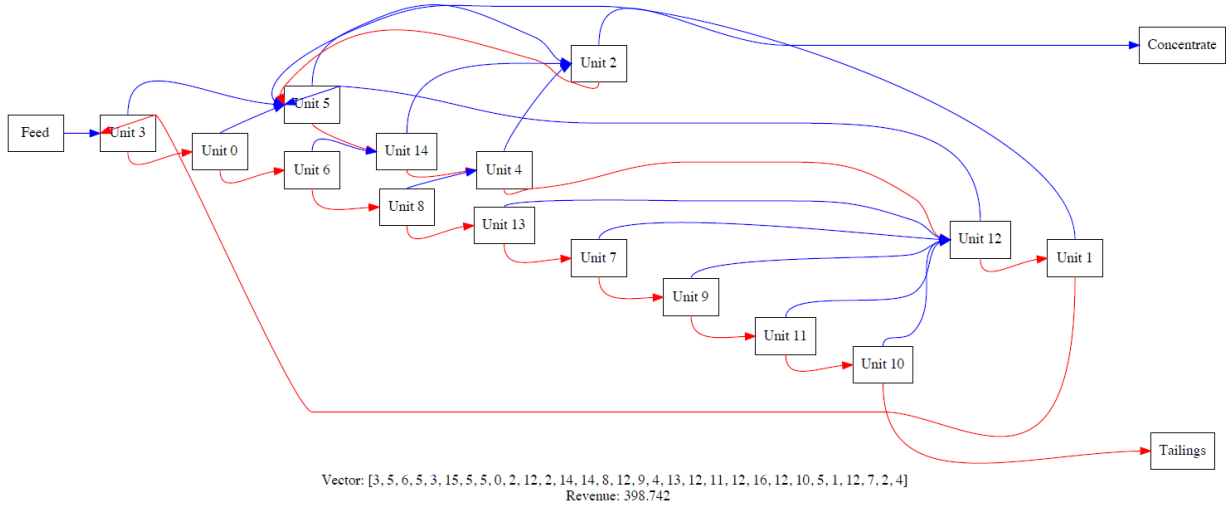


Figure 5. Solution circuit vector for 15 separation units

Economics

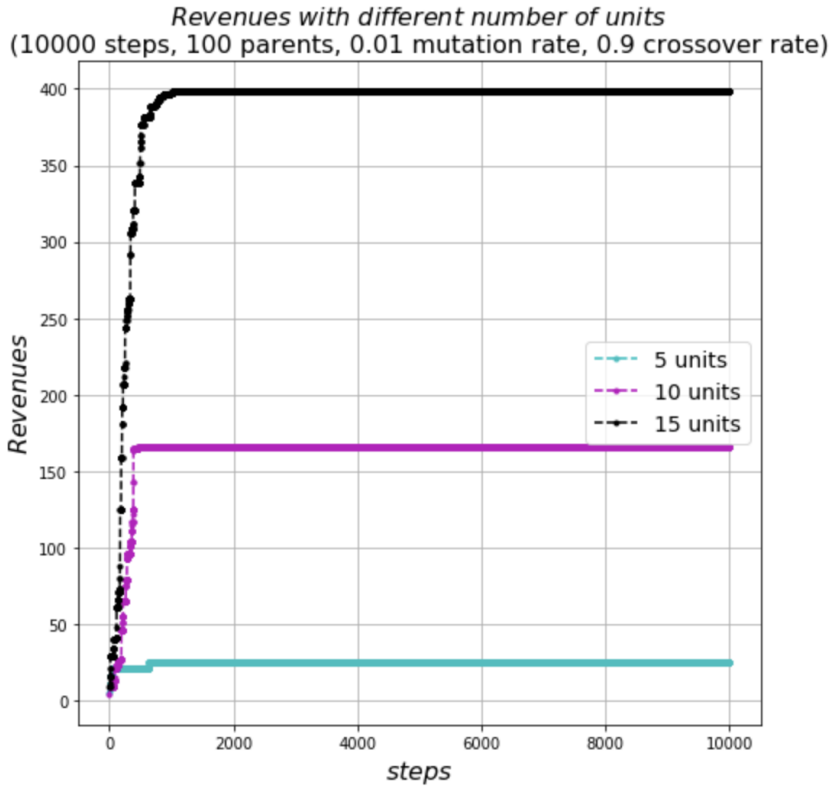


Figure 6. Revenues (performance) of different sized circuits

The above indicates that with increasing numbers of separation units, the yield of germanium is increased along with the overall revenue. From a standalone, maximizing revenue perspective, increasing the number of separation units is the correct strategy.

However, without consideration towards the initial and operating costs associated with extra separation units, it is difficult to gauge whether increasing the total unit count makes economic sense.

VARIATION IN THE COST VS. PROFIT PARAMETERS

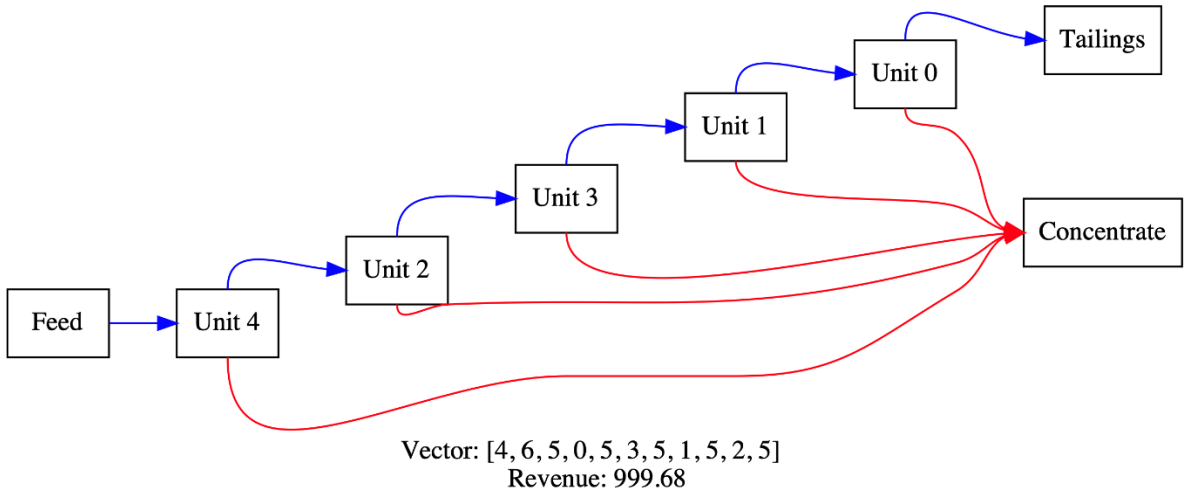


Figure 7. No costs given to waste in the concentrate stream

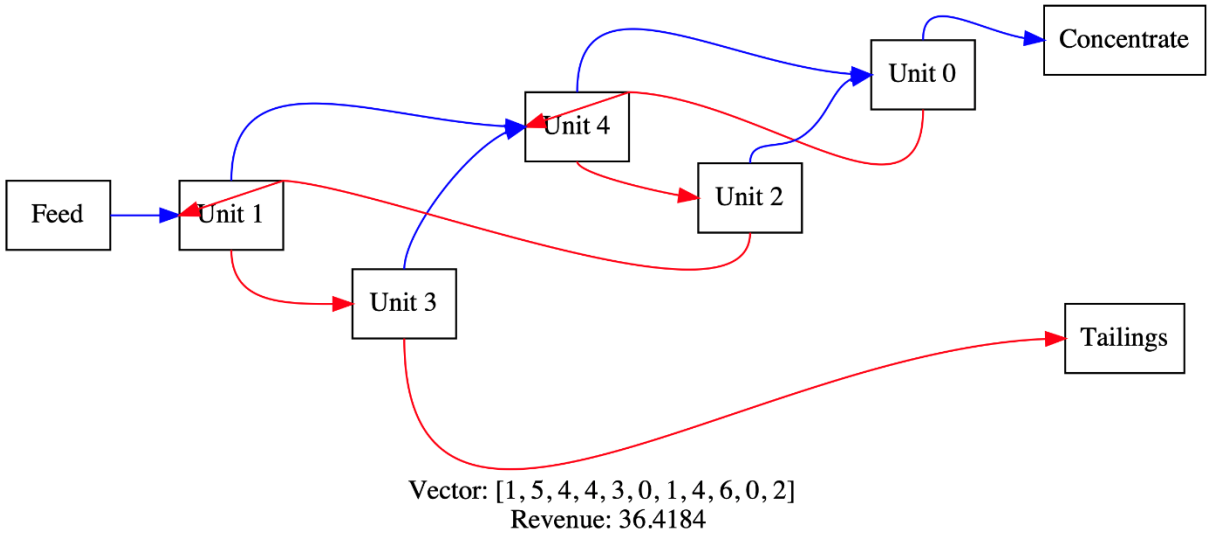


Figure 8. Cost = $300 /kg and Profit = $100/kg

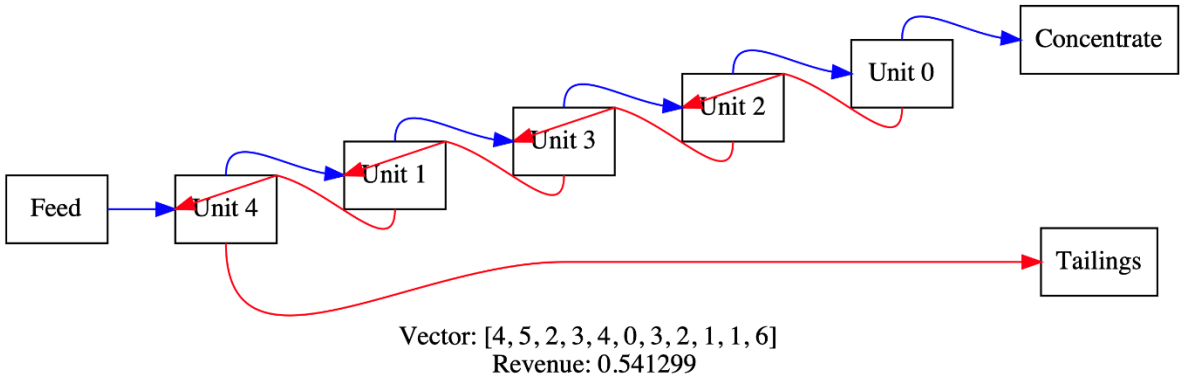


Figure 9. Cost = $5000 /kg and Profit = $100/kg

Varying the cost vs. profit ratio has a direct impact on the optimal circuit configuration. In essence, by increasing the penalty of waste in the concentrate stream, the circuit will arrange itself so that more and more recycles between the units take place. What this achieves is enhancing the purity (or ratio of germanium vs waste) in the concentrate stream.

On the other extreme, when there is no penalty associated with having waste in the concentrate stream, the circuit will arrange itself so that it maximizes the yield of germanium in the concentrate stream. It does this by arranging itself in a continuous series with no recycle, as it no longer matters if waste appears in the concentrate so as long as the maximum possible amount of germanium is being retrieved.

Algorithm Design Parameters

To better understand the optimal tuning parameters for efficient convergence of the genetic algorithm, several experiments were conducted. These tests include varying the pool size of the parent and children vectors and the mutation rate.

POOL SIZE OF PARENT AND CHILDREN VECTORS

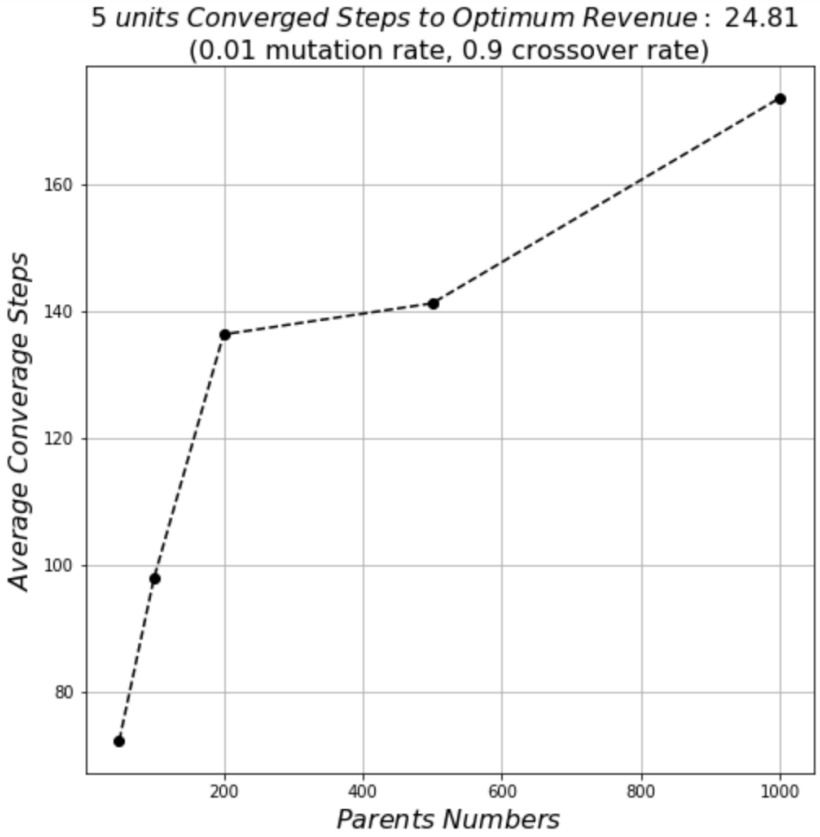


Figure 10. Convergence steps vs parent vectors pool size

Intuitively, based on Darwinian evolution, increasing the number of parent vectors should improve the convergence rate by decreasing the number of steps required. However, the data suggests otherwise.

A possible explanation can be derived considering the process of the genetic algorithm: the best performing children are less likely to undergo a beneficial crossover with each other, due to the sheer number of worse-performing candidates that can be randomly generated in a larger pool size.

Therefore, the algorithm requires additional steps to sort out inefficient candidates. This leads to a stagnation of the performance values in successive iterations.

It should be noted that the parent pool size will vary with the circuit size. In the above experiment, the convergence rate is tested on a circuit of 5 units. With increasing numbers of units, the total possible number of valid vectors will also increase. Therefore, the pool size should be scaled in according to the size of the circuit.

MUTATION RATE

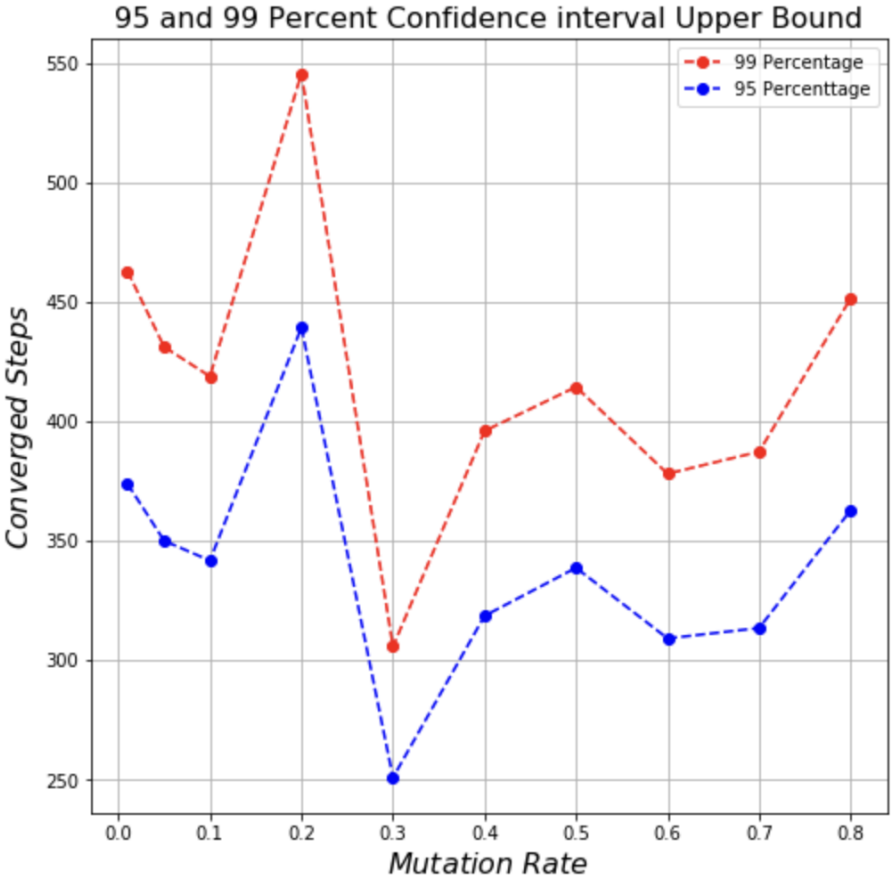


Figure 11. Number of Steps for convergence vs. Mutation Rate

There is no linear relationship between the mutation rate and its effect on the convergence rate of the genetic algorithm. Rather, the above graph indicates that there is an optimum that lies somewhere around 0.3 (30%). This is because on either extremes, mutation can cause improved children vectors to deviate from away from the desirable traits that they have inherited from the parent vectors.

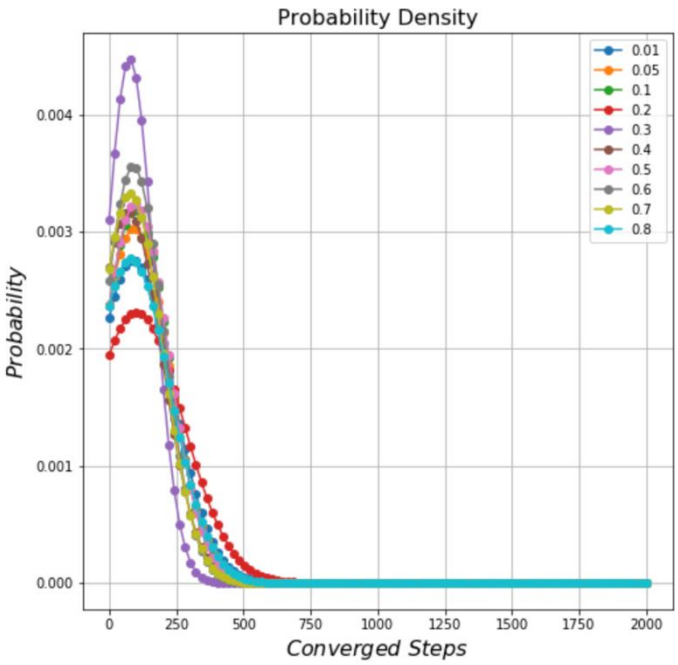


Figure 12. Probability distribution of convergence vs. the number of steps using different mutation rates

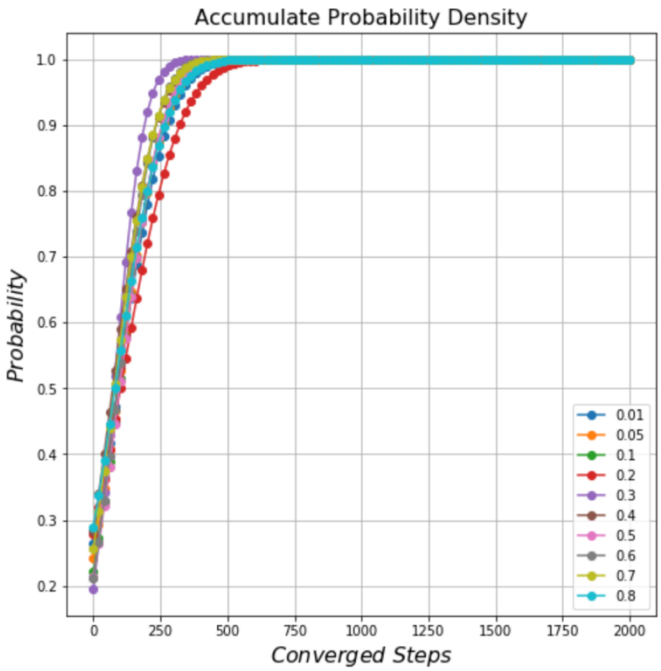


Figure 13. Cumulative probability of convergence vs. the number of steps using different mutation rates

The probability distribution graph above suggests the same conclusion, with the mutation rate of 30% having the highest likelihood to converge given the smallest total number of steps.

CONCLUSION

When trying to optimize the genetic algorithm, Team Platinum suggests setting the mutation rate to 30% based on experimental data.

With regards to the pool size of parent vectors, the end user should proportionally scale the pool size in accordance with the circuit size, with small circuits using a smaller total number of parent vectors and large circuits using larger numbers of parent vectors.

If run time is a consideration, the parallel implementation of the genetic algorithm should be utilized.

PARALLEL GENETIC ALGORITHM

Team Platinum's implementation of the parallel genetic algorithm uses MPI's distributed memory structure. By nature, the genetic algorithm is a series of dependent steps:

1. generating valid parent vectors,
2. evaluating vector performance,
3. randomizing cross over and mutation,
4. creating children vector,

with the most onerous steps being the validity checking and performance evaluation. However, these tasks can be divided among processors as each parent vector is completely random and unassociated with the other parent vectors.

MPI_Datatype and Cmatrix class

Before discussing the implementation of MPI functions, it is important to first highlight a class called Cmatrix implemented in the parallel code.

The Cmatrix class contains a double **pointer called data, which represents a total pool of parent vectors. Within the Cmatrix class are two important MPI_Datatype variables called row_send_type and row_recv_type. Using these two data types, it is possible to send and receive entire rows of vectors (of column length 2n+1) between processors using Cmatrix's member functions send_matrix_rows and receive_matrix_rows.

Domain Decomposition

One important aspect of parallelization is to decompose the problem domain so that no single processor is bottlenecking. To accomplish this goal, a function called split_group evenly divides the total number of parent and children vectors each processor is responsible for evenly (or as close as possible using integer division).

Therefore, throughout the entire genetic algorithm, each processor is only responsible for generating its own assigned number of parent and children vectors and calculating their performance. Each processor will then generate its set of children vectors and send them to all the other processors in preparation for the next iteration of the algorithm.

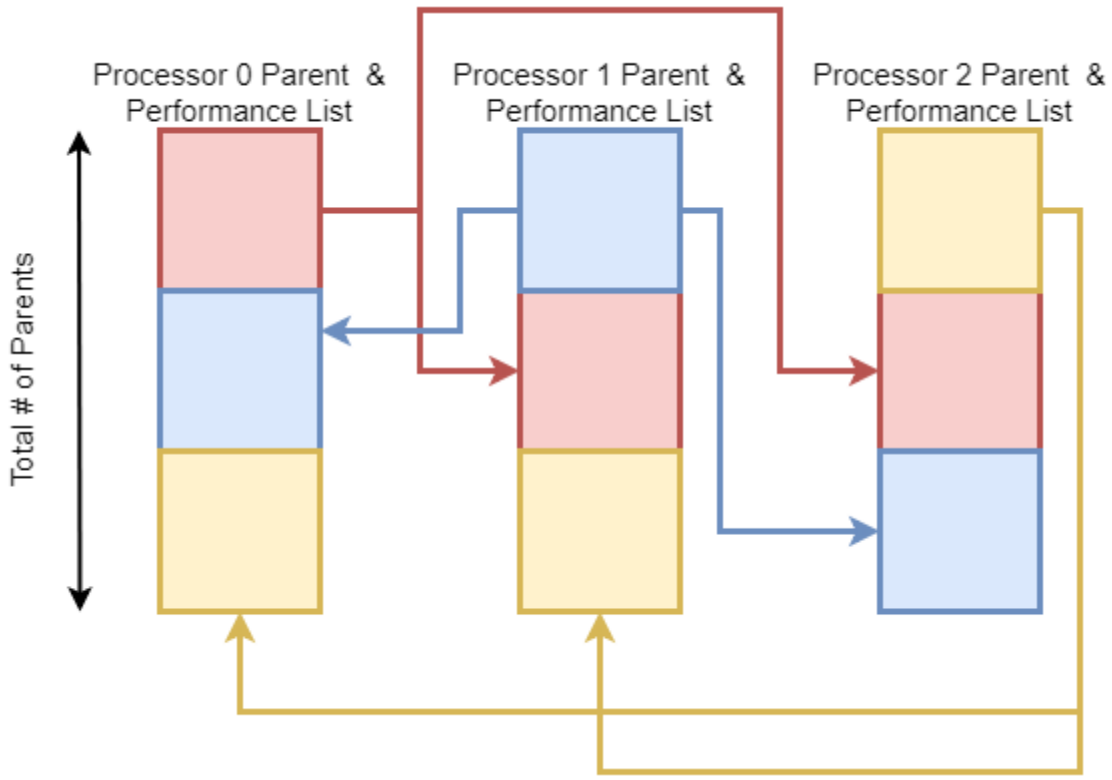


Figure 14. Example of 3 parallel processors communicating their portions of the parent vectors between each other

Limitations

A current limitation of the parallel code is that it assumes an even distribution of the total parent vectors among each of the processors. If the number of parent vectors specified cannot be evenly distributed, then the processor with handling the largest portions may become a bottleneck for the parallel process.

Master & Slave vs. Peer to Peer architectures

The majority of the communications taking place inside the code can be said to be peer-to-peer as each processor is responsible for:

1. creating its own parent vectors,
2. evaluating its own parent vectors' performance,
3. generating its own children vector

and these communications are efficiently achieved using the send_matrix_rows and receive_matrix_rows member functions mentioned previously. The ordering of the parent vectors among the different processors are irrelevant as the performance_list indices are always kept in the corresponding locations through the sends and receives.

One caveat lies in determining when the genetic algorithm has converged. In the main processor (id == 0), the best performing vector is always kept in data[0], or the first vector of the Cmatrix class. Once both of the following criteria are met:

1. The performance of the best vector is within a tolerance of the expected performance and,
2. The number of steps exceeds a maximum number of steps

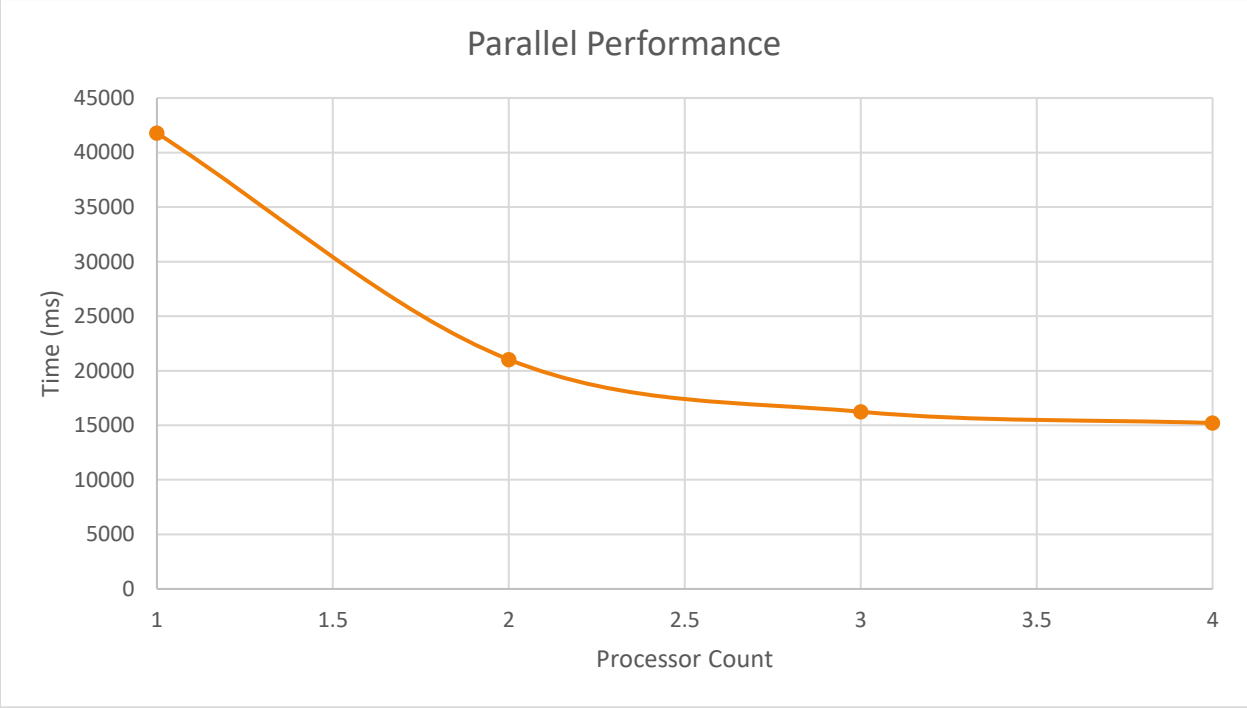
the main processor acts as a master and sends a nullptr to all of the other processors and exits itself. The other processors, using the MPI_Probe function, exits once they receive a nullptr. In this way, the code behaves in a master and slave fashion when checking for convergence.

PERFORMANCE

The following performance benchmark was conducted on:

- a circuit of 5 separation units
- a maximum convergence iteration of 500 for the performance evaluation
- a pool of 100 vectors
- a maximum step count of 500

All of the runs arrived at the expected ideal performance value of 24.82.



Despite a small sample size, the time data above validates the conclusion that the code has achieved a high parallel efficiency as the convergence time is almost directly proportional to the number of processors being used. Without a larger sample size, it is difficult to conclude the asymptotic trajectory of the curve. However, for larger numbers of separation units, the parallel code should be more efficient than serial due to the ability to handle larger pools of parent vectors.

CONTINUOUS INTEGRATION

Team Platinum has implemented Travis for continuous integration. A source file called *test1.cpp* is written for the genetic algorithm simulator which considers the following:

- Whether the simulator is properly linked, all functions are available and the solution vector can be obtained
- Whether the solution vector obtained is valid by calling the check validity function

Inside GitHub, the file *.travis.yml* will run a script called *run_tests.py* automatically, which tests all available tests in the project directory.

It should be noted however, another file named *Makefile* needs to be modified in advance for this process to work. **The existing *Makefile* inside the project directory should satisfy all of these requirements. These instructions are for users who wish to modify the code for their own purposes.**

The contents of the *Makefile* should satisfy the following:

- All new directories' name should match the exact folders. For example, the *SOURCE_DIR* should be the *src* folder : SOURCE_DIR = src
- Create the files which match the name of the header files with extension *.o*. Make sure all *.o* files are included behind the *$(BIN_DIR)/Genetic Algorithm:* .
- Define the number of tests and put the test file's name in the *TESTS* line. Meanwhile, make sure all *.o* files are included behind the *$(TEST_BIN_DIR)/test#:* .
- Make sure all names after the *make* command match the names in the *Makefile*.

Once the *Makefile* is properly created and all of the tests are available inside their directories, Travis will take over automatically for continuous integration.

INSTALLATION INSTRUCTION

For the end user, the following software and modules are required to be installed for the genetic algorithm simulator and post-processing functions:

- Visual Studio Community 2017
- Jupyter Notebook (Included in Anaconda)
  - Matplotlib.pyplot, matplotlib >= 2.2.2
  - Numpy >= 1.14.3
  - Graphviz >= 1.0.0
- IPython Kernel (Included in Anaconda)

To run the genetic algorithm simulator, download the package from Team Platinum's GitHub repository found at: https://github.com/msc-acse/acse-4-project-3-platinum.

Before compiling and running, the user can choose one of two ways to allocate the source and header files:

- Gather all source files in the *src* folder and header files in the *includes* folder from the cloned local repository and then put them inside the visual studio project folder which contains the *.vcxproj* file.
- Gather all source files in the *src* folder and put them in the *includes* folder from the cloned local repository.
    1. Enter the visual studio project.
    2. Right click on the project in the *Solution Explorer* and go to *Properties*, then click on the C++ Directories and choose *General*
    3. Edit the *Additional Include Directories,* add the path of the modified *includes* folder, then click *Apply* in the bottom right corner.
- Choose either method listed above, then right click the Source Files folder inside visual studio, choose *add -> existing item* to add all source(.cpp) files
- Right click the Header Files folder inside visual studio project, choose *add -> existing item* and add all header(.h) files.

To support the parallel version of the genetic algorithm simulator, the following steps are required:

- Install Microsoft MPI: the required files can be found at https://docs.microsoft.com/en-us/message-passing-interface/microsoft-mpi. Click on Downloads to download and install both files.
- Add the location of the downloaded files to the system PATH. Open the Window's System Properties dialog box and click on Environment Variables and click edit for the Path variable and add "C:\MS_mpi\Bin" (assuming the default install location).
- Open Visual Studio to create a new project, right click on the project in the Solution Explorer and go to Properties, then click on VC++ Directories and (again assuming the MPI SDK is installed in the default directory) add:
    - "C\MS_mpi\Include" to the Included Directories
    - "C\MS_mpi\Lib\x64" or "C\MS_mpi\Lib\x86" to the Library Directories (depending on whether to compile the file as 32 or 64 bit, respectively)

HTML DOCUMENTATION

HTML versions of Team Platinum's source file documentation generated by doxygen can be found on the github page: https://github.com/msc-acse/acse-4-project-3-platinum.

APPENDIX: HEADER FILES, CLASSES AND FUNCTIONS

GENETIC_ALGORITHM.H

Functions	Description
`void find_max_min(double *performance_list, int num_parents, int &max_index, int &min_index);`	This function returns the maximum and minimum index values from a list of parent vectors

`void select_parent(int &tgt_parent_index_1, int &tgt_parent_index_2, double *distribution, int num_parents);`	This function randomly selects two parents from a set of parent vectors, with higher likelihood given to parent vectors with a higher distribution (performance
`bool RollingDice(double crossover_rate = 0.9);`	This function is a simulation of a rolling dice. It helps randomly decide whether parents will crossover and/or mutate
`void mutation(int *circuit_vector, double gene_change_rate = 0.01);`	This function goes through a vector and randomly (based on the gene change rate) if each number inside the vector mutate to a different number
`void swapping_parent(int **all_parents, int **new_all_parents, int num_parents, int length);`	This function swaps the components of parent and children vectors based
`bool Check_Convergence(double expected_performance, double performance, double tol, int steps, int max_steps);`	This function is used to evaluate whether the genetic algorithm has achieved the expected performance and ensures that a maximum number of steps is executed
`void run_genetic_algorithm(int **all_parents, int **new_all_parents, double *performance_list, double *distribution, int num_parents, int num_units, int max_steps, double gene_change_rate = 0.01, double crossover_rate = 0.9);`	This is the main genetic algorithm function that executes and produces the solution vector

CIRCUIT_SIMULATOR.H

Functions	Description
`double Evaluate_Circuit(int *circuit_vector, double tolerance, int max_iterations, int num_units, double conc, double waste, double profit, double cost);`	This is the evaluate circuit function. It returns the performance (revenue) of a circuit vector

CCIRCUIT.H

Functions	Description
`bool Check_Validity(int *circuit_vector);`	This function evaluates if a circuit is valid and meets prerequisites
`void mark_units(int unit_num);`	This is the recursive function that enters a circuit vector and marks every separation unit

CUNIT.H

class CUnit	Description
`int conc_num, tails_num;`	These represent the concentrate and tailings units downstream of the current unit
`double top[2];`	Top is an array that contains the amount of germanium and waste in the concentrate stream
`double bottom[2];`	Bottom is an array that contains the amount of germanium and waste in the tailings stream
`double feed[2];`	This is the array that contains the amount of germanium and waste in the feed of the unit
`double old_feed[2];`	This array stores the previous iteration's feed
Member Functions	**Description**
`void calc_stream();`	This member function calculates the quantity of germanium and waste in the concentrate and tailings given a feed
`void store();`	This member function stores the current feed into the old feed
`void wipe_feed();`	This member function resets the feed of a unit to zero

CMATRIX.H

class Cmatrix	Description
`int ** data;`	This double pointer is the container used to store a pool of parent vectors
`int rows;`	Rows represent the total amount of parent vectors
`int columns;`	Columns represent the size of the circuit vectors (2n+1)
`MPI_Datatype *row_send_type;`	This is a custom datatype to send entire rows of data between processors
`MPI_Datatype *row_recv_type;`	This is a custom datatype to receive entire rows of data between processors
Member Functions	**Description**
`void Set_Size(int nrows, int ncolumns);`	This member function sets the size of **data to rows x columns

`void setup_MPI_row_send(int min_row, int max_row);`	This member function creates and commits the row_send_type datatype
`void setup_MPI_row_recv(int min_row, int max_row);`	This member function creates and commits the row_recv_type datatype
`void send_matrix_rows(int destination, int tag_num, MPI_Request *request);`	This member function sends entire rows of data
`void receive_matrix_rows(int source, int tag_num, int min_row, int max_row, MPI_Request *request);`	This member function receives entire rows of data
`void Clear_send_Type();`	This member function frees and resets the row_send_type datatype to a nullptr
`void Clear_recv_Type();`	This member function frees and resets the row_recv_type datatype to a nullptr
`void write_solution_vector();`	This member function writes the first solution vector to a local ".txt" file