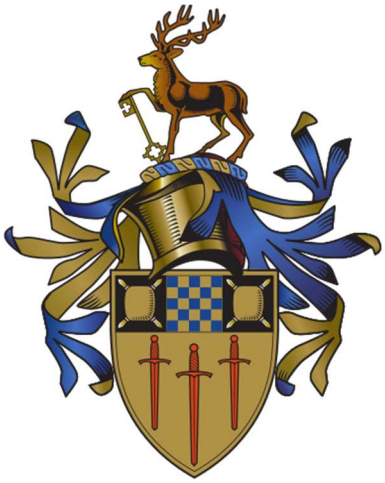
Optimising Happiness in The Student Project Allocation Problem

Adam Dowse

Mechanical Engineering BEng



Department of Mechanical Engineering Sciences
Faculty of Engineering and Physical Sciences
University of Surrey

Project Report

May 2020

Project Supervisor: Prof. Nicholas Hills

i. Personal Statement of Originality:

I confirm that the submitted work is my own work. No element has been previously submitted for assessment, or where it has, it has been correctly referenced. I have also clearly identified and fully acknowledged all material that is entitled to be attributed to others (whether published or unpublished) using the referencing system set out in the programme handbook. I agree that the University may submit my work to means of checking this, such as the plagiarism detection service Turnitin® UK. I confirm that I understand that assessed work that has been shown to have been plagiarised will be penalised.

Name of Student: Adam Dowse

ii. Abstract

Universities should strive to improve the satisfaction of students in final year by increasing the chance they are selected a project they desire for their year individual projects. In the past students have selected several choices from a list of projects and a trial and improvement method was implemented to allocate the projects. In the paper it is looked at what method is most efficient for computerising this problem. This concept is simplified to a problem called the student project allocation problem, SPAP, and is a class of combinatory problem in computer science and mathematics. 4 algorithms were studied, the brute force trivial method, the Hungarian/ Kuhn method, a match pairs function and a basic genetic algorithm. Each one was programmed with the MATLAB environment and timed. This allowed a function to be curve fit, with several error factors ensuring accuracy, so a prediction could be made as to how long each method would take for a given number of students. It was found that the match pairs algorithm performed the best with a big O notation of $O(n^{2.758})$ but could only produce perfect one score for each input that was perfect. In contrast the genetic algorithm was able to produce many possible good solutions that were better than current allocations and provided a choice. In addition, how the number of choices the students are given effects each algorithm was considered. It was found through a survey at the University of Surrey's Engineering Department, that most students were happy with their project if they were allocated one of the selected choices. To formalise this several scoring systems were developed, and algorithms compared. Finally, a workflow was suggested using the best algorithm for how a university could implement such an allocation.

UNIVERSITY OF SURREY

Table of Contents

1. Variables and Abbreviations Used

$\hat{C}$,	Cost matrix
A,	Allocation vector
N_p,	Number of projects
N_s,	Number of students
N,	Number of data points
SSE,	Sum of squares error indicator
$RMSE$,	Root mean square error indicator
R^2,	R-Squared error indicator
X_i,	Recorded data
x_i,	Predicted data
$Gamma\ or\ \Gamma$,	The gamma functions
P,	Population array
G,	Gene
c_i,	Chromosome
C,	Total cost of allocation
$\hat{C}$,	Total cost of allocation array
s_i,	Individual score
$\hat{S}$,	Score array
Pr_i,	Individual probability
$\widehat{Pr}$,	Probability array
σ,	Probability gain constant
c_{min},	Minimum costing chromosome
$\emptyset$,	Mean cost
TSP,	The travelling salesman problem
$TMax$,	Maximum time taken to produce a solution
$TAve$,	Mean time taken to produce a solution

2. Introduction

Satisfaction ratings play a major role in a Universities reputation and often how happy students are with their, yearlong, dissertation project has a huge effect on this. If a university could increase the chance of assigning each student the project they wanted, the university would hypothetically improve its ranking and produce more enthusiastic student dissertations, potentially leading to higher grades for students. The student project allocation problem, SPAP, is a generalised optimisation problem that involves allocating projects to students in accordance to each student's preference for each project. In Universities this is currently done by trial and error and has taken up to 3 days to allocate all the projects by hand (Surrey, 2020). An optimisation algorithm was designed to reduce the time taken to solve this problem and to increase the satisfaction of the students by increasing the number of students with their preferred choices. Outside of this specific setting, optimisation problems like this are commonplace in work environments when assigning jobs to staff to increase their performance or to ensure they are satisfied with their area of work. Research was conducted at the University of Surrey into how long, typically, allocations like this take and the

satisfaction of the students with their projects, considering outside factors such as the satisfaction of the project.

A survey was produced and conducted into how happy students were with the allocation of their projects in the faculty of engineering at the university of surrey in 2020 (Surrey, 2020). They were asked what ranking they gave their project that they received and how happy they were with this decision on a scale from 1 to 10, where 10 was very happy and 1 was completely unsatisfied. From a sample of 32 students it was calculated that the current methods produced a score of 0.845 using a genetic scoring method, section 4.3.3, or 3.79 when using the standard scoring system, explained in section 4.3.2. This gave an understanding of what scores needed to be produced in the algorithms studied in this paper.

The SPAP is defined as a minimisation problem in the form of equation 1, where the allocation, A, points to the cost matrix, C-hat, and the sum of these values is to be minimised.

$$\sum \hat{C}(A_i) \ minimised \tag{1}$$

This is a linear integer programming problem that is described in more detail in section 6.1. The problem in this paper is constrained to only allowing students rankings to be positive integers that are consecutive and the number of projects, Np, must equal the number of students, Ns. The SPAP exists as an umbrella term for this type of problem often used in computing science and mathematics. Other authors, such as A Ismaili, have discussed using 2 sided preferences to solve this problem (A Ismaili, 2019), the preferences could be weighted by how well a student scored in recent assignments or several different profile based solutions have been presented (M Chiarandini, 2019). In most cases the problem of stability in introduced when 2 minimisation functions are fighting against each other (A Ismaili, 2019). In this paper it was chosen to only consider one sided preference from the student's side, given a list of projects with a specified number of selections that each student can make.

In this paper 4 algorithms were evaluated when solving the SPAP by their efficiency and producing an allocation that students would theoretically be satisfied with, these included a brute force trivial method, the Hungarian method, a matched pairs algorithm and a genetic algorithm.

3. Methodology

To find the most efficient method of solving the SPAP each algorithm was programmed using MATLAB 2019b and tested by changing the number of students the algorithm had to produce an allocation for and the choices each student was allowed to select. These runs were then timed with the cputime() function multiple times in order to have enough data to produce prediction of how long it would take to assign a specific number of students. It was important to find the worst-case scenario and therefore 2 curves were fitted to the data on each graph to show the expected time taken and the slowest time possible to produce an effective allocation.

This would allow a potential user of one of the algorithms to be sure an answer could be produced in this time. In addition, the scores, section 4.3, of the programs were analysed and compared to see if they were affected.

The computer used for the simulations was an intel core i7- 6700K 4GHz processor with 16Gb of RAM and a 64bit operating system.

Several error indicators were used to evaluate the performance of each equation against the data. The sum of Squares, SSE, is calculated with equation 2 (MathWorks, 2020) and calculates the average error between the curve fitted equation and the data points providing an indication of how well the line has been fitted to the raw data. Where x_i is the curves points, X_i is the recorded data and the bar denotes the mean of that set.

$$SSE = \sum_{i=1}^{N} (x_i - \bar{X})^2 \tag{2}$$

2 other indicators were used to evaluate the prediction lines in this paper, RMSE, equation 3 (MathWorks, 2020) and R-Squared values, equation 4 (MathWorks, 2020). All 3 functions were evaluated in MATLAB's curve fit toolbox. The RMSE equation took the square root of the mean of the expected results minus the recorded data squared divided by the number of data points N. This provides a measure of goodness of fit to the data and is perfectly fit when this is equal to 0. The R squared value determines the goodness of fit to a regression line with a best score of 1. It is calculated by the ratio between the sum of squares in the regression and the total sum of squares (MathWorks, 2020).

$$RMSE = \sqrt{\frac{\sum_{i=1}^{N} \overline{(x_i - X_i)^2}}{N}} \tag{3}$$

$$R^2 = 1 - \frac{\sum_{i=1}^{N} (X_i - x_i)^2}{\sum_{i=1}^{N} (X_i - \bar{x})^2} \tag{4}$$

With predictions with a strong goodness of fit the most efficient algorithm could be selected.

4. Definitions

4.1 Big O notation

Computer programs are designed to run fast and to be efficient in their use of storage and the time taken to run so more calculations can be performed and larger analysis performed. Analysis is therefore a vital tool in ensuring functions and scripts take as little time as possible to run and how this changes when the inputs are changed, separates an efficient algorithm. Big O notation is used to show how quickly something scales. As algorithms can vary the time taken to run it was important to take a worst-case scenario, showing that the program would never take longer to run with a set input size. It can be used to measure the time taken to run an algorithm or to show how much computer memory would be needed as the input size

increases. For example, an algorithm that is said to run in linear time complexity, O(n), would take an equally increasing amount of increase in time taken to run compared to functions that increase more rapidly like, O(n^3), (Bae, 2019). Further examples of big O notation can be seen in figure 1 (Bae, 2019).

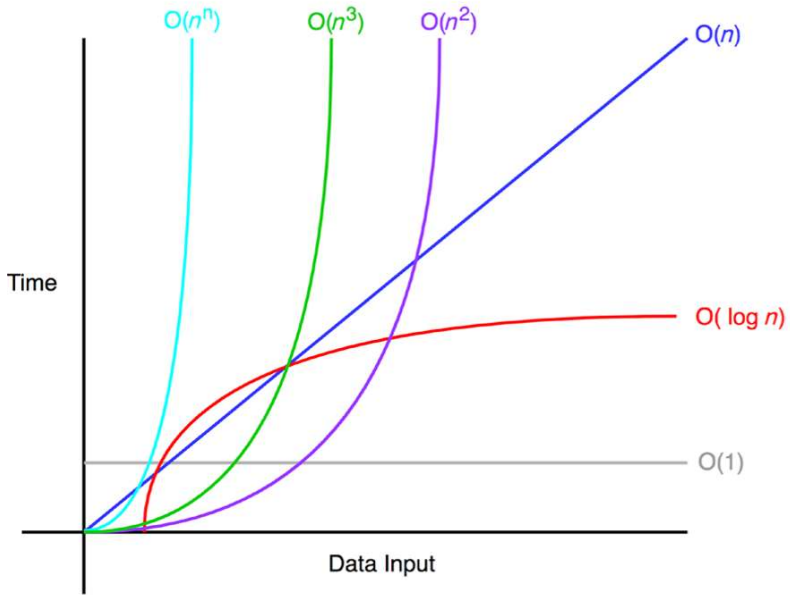


Figure 1, Time complexity explained with big O notation in various examples. Taken from: (Bae, 2019).

This is used in this paper because it ensures the longest a program could take for different numbers of students. It also allows the programs to be ranked quantitatively.

4.2 The Cost Matrix and Test Data

The cost matrix was used to input students project preferences into each algorithm in a clear way that also allowed the programs to access all the data in one variable. Cost matrices are used to compare 2 vectors and are often used in optimisation problems (Kuhn, 1955). For example, in this project each student must be compared to each project providing a ranking for each. A vector was defined of student indexes, S_{index}, and a vector of project indexes, $Proj_{index}$, each combination was evaluate as the students preference, 1 being the student's first choice and N being the student's last choice. If the choices available to the student are not equal to the number of projects, then all selections outside of the student's selection is scored a large number. More information pertaining to this is presented in section 4.3.2. This was presented as an example in table 1, where 4 students, 4 projects are selected, and each student has 3 choices.

Table 1, An example cost matrix with 4 students, 4 projects and 3 student choices.

	Students			
Projects	1	1004	3	1004
	1004	1	2	1
	2	3	1	2
	3	2	1004	3

This cost matrix can then be used throughout the algorithms to check how well a solution has been performed. How this is analysed is discussed in section 4.3. A function was created seen in appendix A to produce the best- and worst-case data sets to stress test the algorithms, ensure they did not contain bugs and were producing correct scores. The worst case would include all the students choosing all the same projects as each other and the best case is only available when each student has an independent first choice to every other student, shown in table 2. This function was also able to produce biased data where there was a high percentage to select the same projects so popular choices can be assessed.

Table 2, Example best- and worst-case cost matrices with 3 students, 3 projects and 3 choices.

		Worst Case Scenario				Best Case Scenario		
		Students				Students		
Cost Matrix	Projects	1	1	1	Projects	1	2	3
		2	2	2		2	1	2
		3	3	3		3	3	1

4.3 Scoring System

4.3.1 Complete Information Scoring

A scoring system was defined so each algorithm could be compared, and a standardised input and output format was used. It is important to note that there may not have been a perfect score possible and the chance of this is reduces as the number of students increases as more permutations of allocations are available. For example, if all the students wanted the exact same projects the score could only produce the best possible allocation and not a perfect one. Table 3 shows a worst-case and best-case scenario cost matrix when each student ranks every project in the list.

Table 3, Example best- and worst-case scores for different allocations using method 1.

		Worst Case Scenario				Best Case Scenario		
		Students				Students		
Cost Matrix	Projects	1	3	2	Projects	1	2	3
		2	1	3		2	1	2
		3	2	1		3	3	1
Allocation		3	1	2		1	2	3
Score		(3+3+3)/3 = 3				(1+1+1)/3= 1		

Once an allocation, A, had been created and populated with the project index assigned to each student then the scores could be calculated. This was done in 3 ways to assure the best method was chosen. Method 1 took each project index assigned to each student and referenced the cost value in the cost matrix adding up each score and dividing by the number of students, in order to allow the scores to be comparable to different size problems. An example of this can be seen in table 3. Method 2 added a multiplier to each score and divided the sum of the score by the number of students and method 3 squared the multiplier used. Each method had the effect of spreading out the scores as seen in figure 2.

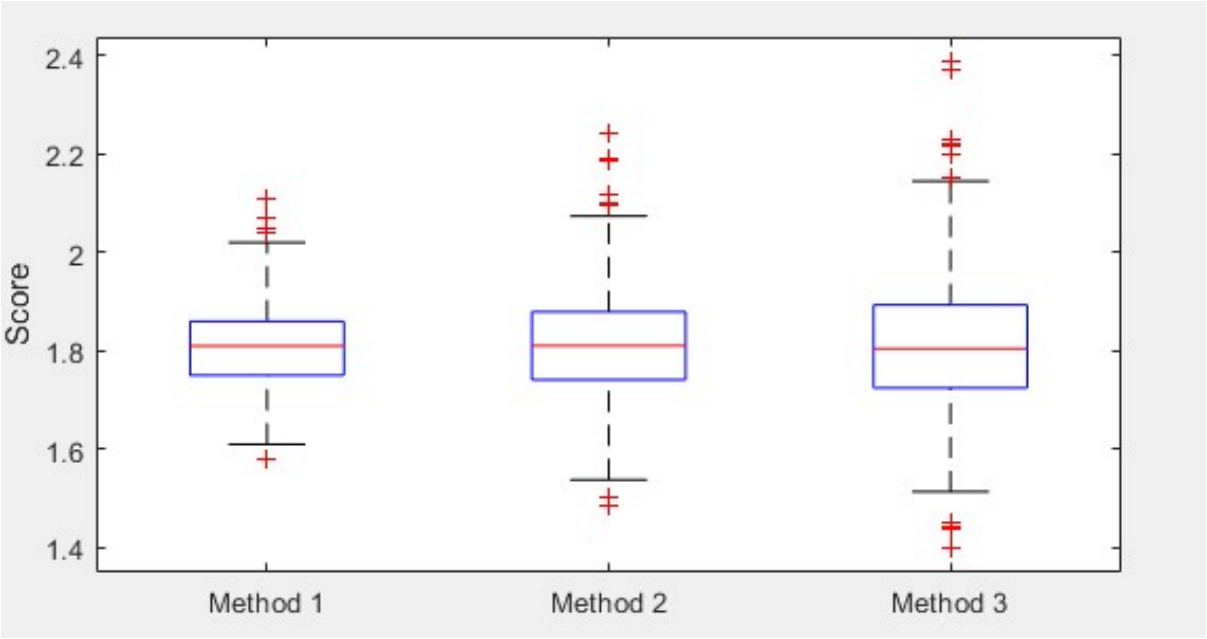


Figure 2, Hungarian method used with 100 students, 100 projects and 100 choices for the 3 different scoring methods producing box plots.

Method 1 was used throughout this paper up until the genetic algorithm because only the spread was affected, and the decision was made that added steps were not necessary to show how score changed.

4.3.2 Scores Effect When Changing Choices

The project also focused on changing the number of choices students have when selecting projects and a representation was needed to fill the cost matrix when this data was not available. It was decided to use equation 5 because it prioritised finding a solution that is always within the student's choices and harshly penalising an allocation outside of these values. If an allocation were made to a student outside of his/her choices it should be assumed that they would be unhappy with their result because it would be analogous to assigning a project randomly to the student. This was supported by the student survey conducted at the University of Surrey that showed if a student got a project in their top 10, they scored their experience 7 out of 10 when asked how happy they were with their selection on average.

$$\widehat{C_{i,j}} = 1000 + N_s \tag{5}$$

Upon the decision to use method 1, equation 6 was used to calculate the best-case scenario and equation 7 to calculate the worst-case scenario, the data was displayed in table 4.

$$Worst\ Score\ Possible\ With\ Limited\ Choices = (1000 + N_s) \tag{6}$$

$$Best\ Score\ Possible\ With\ Limited\ Choices = 1 \tag{7}$$

Table 4, Example best- and worst-case scores for different allocations with 3 students, 3 projects and 2 choices.

		Worst Case Scenario				Best Case Scenario		
		Students				Students		
Cost Matrix	Projects	1003	1	2	Projects	1003	1	2
		2	1003	1		2	1003	1
		1	2	1003		1	2	1003
Allocation		1	2	3		3	1	2
Score		(1003+1003+1003)/3 = 1003				(1+1+1)/3= 1		

This changed the maximum score an allocation could receive but still provides the comparison between different algorithms when in both unique cases while separating assignments with all projects inside a student's selection of choices or not.

4.3.3 Genetic Scoring

A different scoring system had to be designed for the genetic algorithm used in section 6, because the scores had to range between 0 to 1 so they could be used as probabilities during the selection stage. Equation 8 shows how this was achieved (StackOverflow, 2011).

$$Normalised\ Score = 1 - \frac{Score - N_s}{(1000 + N_s)^2 - N_s} \tag{8}$$

Once these scores were found they needed to sum to equal 1, allowing them to be used as a probability distribution. Equation 9 was used for this (Mathematics Stack Exchange, 2013).

$$Distributed\ Probability_i = \frac{Normalised\ Score_i}{\sum Normalised\ Score} \tag{9}$$

MATLAB code for all the scoring methods can be found in the appendix B under the function TestEff().

Survey

The survey conducted at the University of Surrey's Engineering department asked 32 students that had been allocated project from 132 projects where each student got to choose their top 10 choices what they ranked the current project. They were asked from a scale of 1 to 10 how happy they were with their allocation specifically, where 1 was very upset and 10 was the project choice was perfect. This data was taken, and the ratings were compared to the what project choice they got allocated.

Table 5, Analysis from the student survey.

Choice	Average Satisfaction Score	Number of Students with This Choice
1st	9	4
2nd	8.25	4
3rd	8	3
4th	7	1
5th	7	2
6th	8	1
7th	4.5	2
8th	4	1
9th	8	1
10th	-	0

At this university students are consistently less happy with projects lower on their list. Although the sample size was small this was evidence that first second and third choices should be prioritised and a good benchmark to see how many students got what choice. Additionally, the score was calculated using the genetic method of scoring and was found to be 0.845 and with method one a score of 2.11 was found. This can be used to benchmark the following solutions.

5. Brute Force (Trivial Case)

It was important to establish a baseline to compare algorithms against, that was the most trivial solution. The brute force method was chosen and provided the opportunity to ensure functions produced correct solutions later in the paper. This method finds a solution by analysing every possible permutation of projects in the allocation array and using the highest scoring allocation as the final answer.

A list of integers from 1 to the number of students, N_s, was created as 2 arrays called 'Array()' and allocation, A, and the variable **BestScore** was initialised to equal infinity(). These initial variables are set up at the start of the program before the main loop was started so the correct memory is allocated and **BestScore** can be set to be the worst it can be. This is described by equations 10 and 11.

$$Array = A = [1 \quad \dots \quad N_s] \tag{10}$$

$$BestScore = \infty \quad (11)$$

Because the program was looping through a permutation, the number of iterations needed to search through all the possible solutions was the factorial of the number of students in the allocation, N_s. This suggested that this program ran in at least O(n!) time without the addition of extra processes inside each loop. After trying to store every permutation in a variable and loop through these solutions, the computer ran out of memory above 10! and the time taken to use the program was unsuitable. This was because the matrix that the permutations were being stored in exceeded MATLAB's maximum variable memory allocation limit for a 64bit processor of 14253 MB (MathWorks, 2020). Because of this the solution of using lexicographical permutations was found.

5.1 Lexicographical Permutations

The first step in the loop is to produce a permutation lexicographically and assign it to a variable so it can be used as the allocation. The principle behind producing permutations in this way is finding a logical method of iterating through each permutation without repeats because each permutation is assigned a number, a lexicographical index. Using this method to produce each permutation ensures all the allocations are tested while only storing one allocation at a time (Mossige, 1970).

The function created was based on the principles by a contributor on an online platform and inputs the list of numbers that the permutation can take, and the lexicographical index of the permutation requested (Mathematics Stack Exchange, 2012). **Inverse_i** is set to 0 so the following code can keep track from 1 to N_s -1 as well as from N_s-1 to 1. The program then follows the following pseudo code to form the given permutation with the inputs of an array of the first permutation called ***Array*** and the index of the permutation that was desired called ***Index***. The output of this function is the permutation called ***OutArray***.

```
FUNCTION OutArray = nthPerm (Array, Index)
(1)   Length equals the length of the first permutation called Array.
(2)   Inverse_i equals 0.
(3)   FOR i equals the length minus one, to 1.
(4)     Track the index of the permutation going up from 1 to length called
        inverse_i.
(5)     X equals 0.
(6)     WHILE the index is less than 0 or the index is greater than the
        factorial of i.
(7)       Increase X by 1.
(8)       Index minus the factorial of i.
(9)     END
(10)    OutArray with an index of inverse_i equals Array with the index of X
        plus 1.
(11)    Array with an index of X plus 1 is deleted.
(12)  END
(13)  OutArray with an index of Length is equal to the last value in Array.
END
```

In more detail, the program can be found in appendix C.

Once the nthPerm function was written the brute force code simply looped through all the permutations using the function and each allocation was scored with method 1. If the new allocation's score were better than the previous best it would save that as the best score and change the best allocation to the current allocation. Once all the possible solutions were searched the program would have the best solution to the problem and produce the allocation that suited.

5.2 Analysis

In order to prove that the written code was working correctly bias data was inputted into the function and the results scored as well as 10000 random samples to produce a histogram using 8 students in figure 3. This showed that the program was functional with the worst score being 3 and the best score being 1, as expected. This also gave insight into the average score that a given allocation could be expected to get if the best possible allocation were provided. This is provided that all student-project choices are completely random and there are no other factors influencing their decisions. The test data function created also allowed bias data to be created, implying there were favourite projects or favourite project leaders. The data showed a

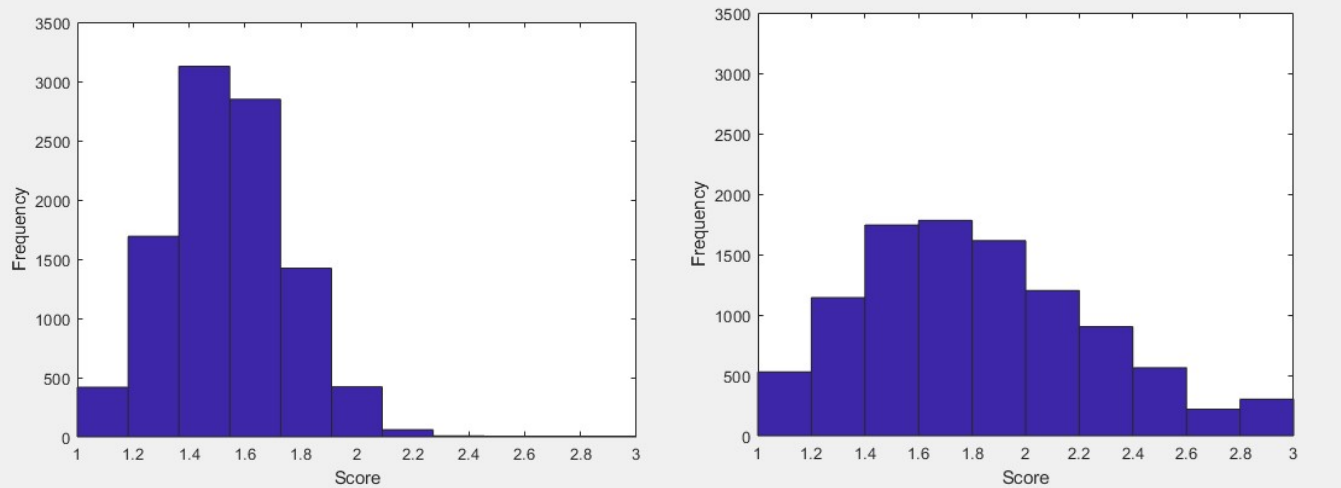


Figure 3, Histograms of the scores for 10000 runs the brute force method using 8 students, 8 projects and 8 choices. (Left: Uses unbiased input data) (Right: Uses biased input data).

reduction in the quality of score and a spreading of the data. This implies that a selection of projects that is diverse and suits the students wants, will produce more satisfied allocations.

By principle, the algorithm was expected to scale factorially because by increasing the allocation input size by 1 meant finding every permutation up to this point and for each new permutation. For example, finding the number of permutations when the allocation size was 2 verses when the allocation size is 3 is shown in the table 6 below.

Table 6, Allocation size and the corresponding permutations

Allocation Size	2	3
Permutations	1,2 2,1	1,2,3 1,3,2 2,1,3 2,3,1 3,1,2 3,2,1
Number of Permutations	2	6

It was therefore appropriate to suggest that the time space complexity of this algorithm would be O(n!). The function was timed for 10 runs and the maximum data was used in the curve fitting toolbox to find an equation that estimated the maximum possible time it would take for larger allocations to run. It was found that the gamma function shown in equation 12, can be used to approximate a factorial function shown in equation 13 and was therefore used as the custom equation for the curve fit (MathWorks, 2020). In addition, constants were added to the function in the form found in equation 14.

$$Gamma(x) = \Gamma(x) = \int_0^\infty e^{-t}\, t^{x-1} dt \tag{12}$$

$$y = Gamma(x + 1) \tag{13}$$

$$y = a \times Gamma(b \times x - 1) \tag{14}$$

The values found for a and b were 2.117e-5 and 1.066, respectively. To increase confidence in these results a comparison was made with an exponential function using the same curve fitting toolbox to ensure the curve was not a rapidly grown exponential function that approximated to a factorial. The equation base used is shown in equation 15.

$$y = a \times x^b \tag{15}$$

The values of a and b were found to be 9.13e-23 and 22.06, respectively. These curves were then plotted against the maximum values of the data and the average data points. Using the maximum was important for these equation's data because it represents the worst-case scenario in the tests and is what would need to be considered when comparing methods. Finally, the data was plotted on a graph in figure 4.

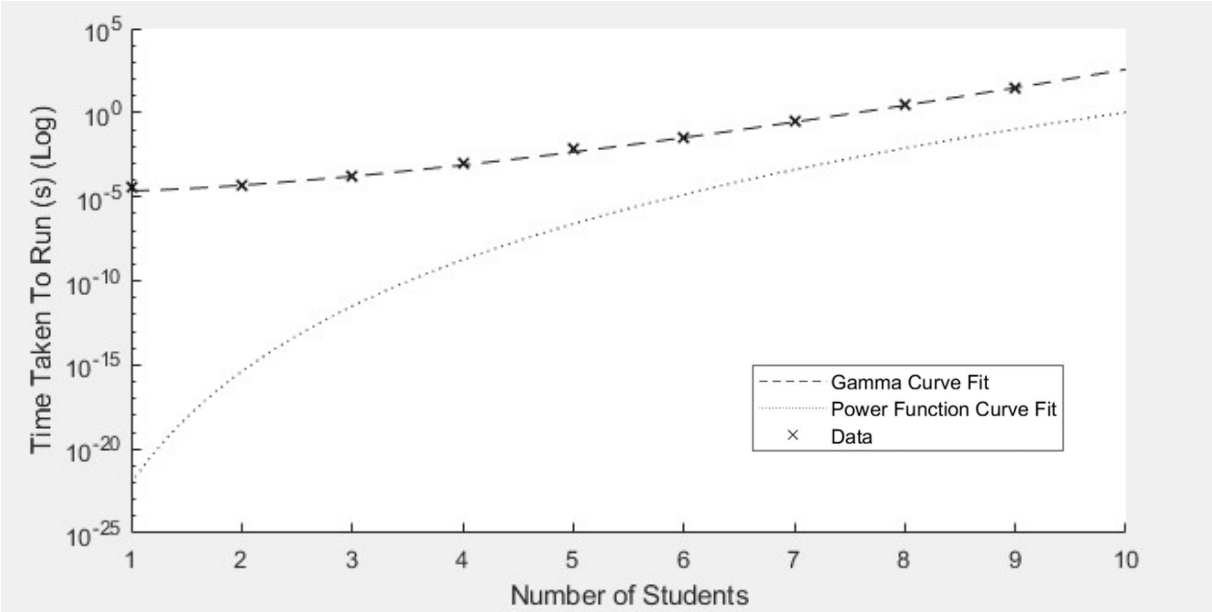


Figure 4, Time taken to run the brute force algorithm, showing an approximated factorial function and a power function curve fit.

From figure 4 and the data in table 7 the factorial gamma function fits the data much more accurately than the power function although both functions had near perfect SSE scores, very low RMSE values and a perfect R-squared score. The factorial function followed the data more strongly, thus proving the algorithm runs with an O(n!) time complexity.

Table 7, Analysis of curve fits for the brute force method.

Equation Type	Equation	SSE	R-Squared	RMSE
Gamma	$2.117 \times 10^{-5}\Gamma(1.066x - 1)$	3.41E-05	1	0.002065
Power	$9.13 \times 10^{-23}x^{22.06}$	0.008552	1	0.03495

In addition, it can also be noted that the variation in the results was considerably small with a maximum standard deviation of 0.1562. Figure 4 showed how this gamma function grows rapidly during the time taken for it to produce a solution and only 14 students could be allocated successfully in 3 days and therefore being unsuitable for a university's needs.

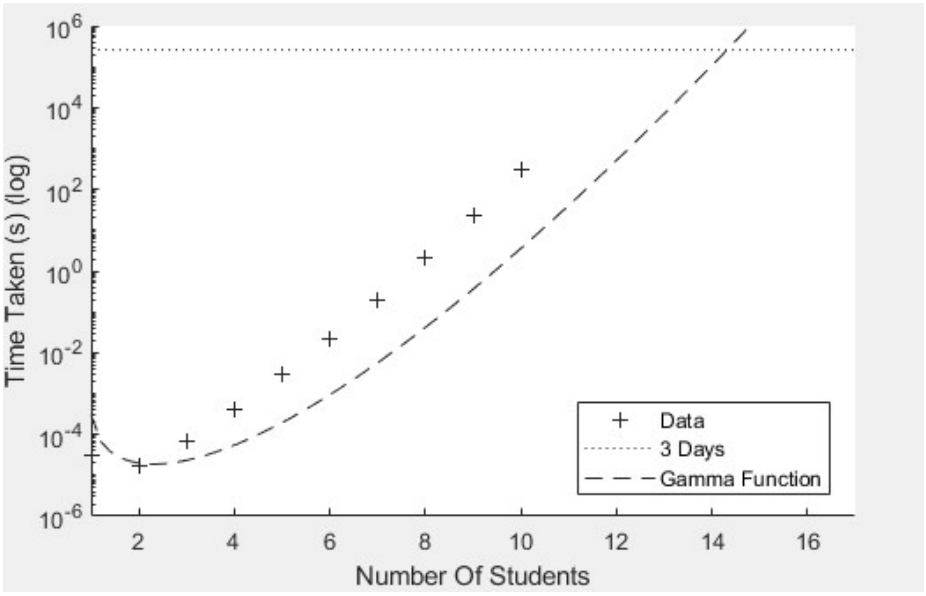


Figure 5, Time taken to run the brute force algorithm showing an intersect with 3 days.

The second criteria in this paper was to evaluate how changing the number of choices that the students have affects the scores of the solutions produced and thus their happiness. Therefore, the brute force method was run for 8 students and 8 projects and the number of choices given to the students ranged from 1 to 8 (Figure 6).

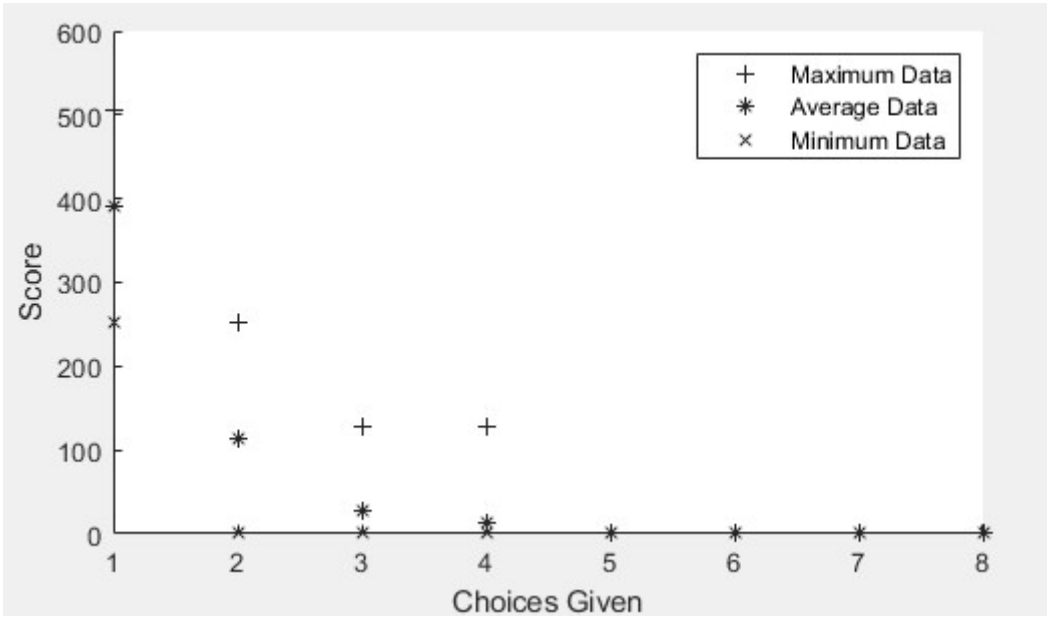


Figure 6, Score of the brute force method with 8 students, 8 projects and the choices given are varied.

The score increased rapidly when the students were only given 1 to 4 choices out of 8 and shows how student satisfaction will drop when they are given less projects that they select to

choose. However, the score is adjusted drastically by an allocation outside of the students preferred choices and from 5 choices upwards every student was allocated their choice. This does not take into account how happy each student is with this allocation it just shows it is likely that students start to be allocated projects outside of their chosen list when they can choose a limited number of projects. Additionally, the time taken to run this algorithm, when the choices were changed, did not alter as the same computation is done no matter what occupies the cost matrix, it is simply the number of students that effects this. Varying the number of projects was outside the scope for this project but it could be estimated that reducing the size of the cost matrix would reduce the time it would take for the algorithm to run. Further research should be conducted on this matter.

The solution produced here was not suitable for the SPAP and especially for use in university setting with student numbers around 150 so further solutions were explored.

6. Hungarian

The Hungarian method was first discussed as a form of solution to the assignment problem from 1949 to 1951 by an array of authors studying linear programming and solutions linked to the travelling salesperson problem, TSP. However, Kuhn, was the first to formalise these findings and realise their potential in computational systems (Kuhn, 1955) and thus the algorithm is often named after him. In his paper he started by showing how the simple assignment problem, where the cost values were either 1 or 0 to represent whether a job can be performed or not in binary. From this simplified problem and alongside 7 mathematical theorem Kuhn provided proof that an optimal solution can be found if the binary job representations are converted into a positive number representing how well the job will be performed and thus creating the basis for the SPAP.

6.1 Linear Integer programming

The idea of linear integer programming was the basis that the Hungarian method was founded on, and so it was key to understanding the algorithm's origins. The premise of linear integer programming was that an objective or cost function of a minimisation or maximisation of a sum uses only a linear variable (Karloff, 1991). For example, the objective of the SPAP was to minimise the sum of the costs associated with the allocation vector, equation 1, and thus because the problem contains just one variable, it can be solved using linear programming. As the problem is also an integer problem, where the costs are constrained to positive integers, linear integer programming can be used to solve problem. It is summarised by Karloff in his quote, 'Linear Programming is the process of minimizing a linear objective function subject to a finite number of linear equality and inequality constraints.'. He also goes into further detail into how linear programming is used specifically in the SPAP (Karloff, 1991).

6.2 Implemented method

The solution used in this paper was adapted from (Karloff, 1991) (Arif Anwar, 2003) (Kuhn, 1955). The outline of the code allowed the iteration of the 2 functions described below and is limited to 100000 repetitions to account for a solution where an optimal allocation cannot be found, implying an error is present. The 2 functions are then run one after the other and the second function returns a variable called completed, allowing the program to be terminated once a complete solution has been found.

Function 1 was to perform row and column reductions by subtracting the minimum value in each row or column from each other row or column. This was simply achieved by for loops, MATLAB's min() function and simple manipulation of matrices and is shown in appendix D.

Function 2 completed the row and column scanning that was needed to look for a solution each iteration. The cost matrix was inputted and the allocation, modified cost matrix and the completed variable are passed out of the function. To start variables were initialised for the holding the number of 'drawn' lines on cost the matrix.

Defining the following terms can be seen in table 8 and then be used in the following pseudo code:

Table 8, Variables used in the Hungarian method.

Variable	Type / Size	Description
Allocated	Row Vector	The allocation of projects to students.
TData	Matrix	The cost matrix with students and their project choices.
Completed	Binary	Has the solution been found? 1 for yes 0 for no.
LinesNum	Integer	The number of lines drawn.
Lines	Binary Matrix	The cover matrix that shows what cells are covered by a line.
ZeroSumR	Column Vector	The sum of each row of the cost matrix.
ZeroSumC	Row Vector	The sum of each column of the cost matrix.
Row	Row Vector	Holds the row index of values that correspond to the cover matrix in the cost matrix.
Col	Row Vector	Holds the column index of values that correspond to the cover matrix in the cost matrix.
ModData	Row or column Vector	Holds the data of a specific line in the cost matrix that correspond to the data not covered by the cover matrix.
ModIndex	Row Vector	Holds the indexes of where Lines is equal to 0.

ZeroIndex	Integer	Stores the index of the single zero in the row or column so it can be allocated.

```
FUNCTION [Allocated,TData,Completed] = RCScanning(TData)
(1)   LinesNum equals 0.
(2)   Lines equal to a matrix of size [Np,Ns] full of zeros. Representing
      no lines drawn.
(3)   Allocated = Row vector of size [1,Ns] filled with zeros.
(4)   WHILE the number of zero values in TData that correspond to where the
      lines matrix has a zero value, does not equal 0.
(5)      ZeroSumR equals an array of the sum of values where TData
         and Lines equal 0 in the rows.
(6)      Where ZeroSumR equals 0 set it equal to NaN.
(7)      ZeroSumC equals an array of the sum of values where TData
         and lines equal 0 in the columns.
(8)      Where ZeroSumC equals 0 set it equal to NaN.
(9)      IF the smallest value in ZeroSumR and ZeroSumC are less
            than 1.
(10)        Set the corresponding indexes of where Lines and TData
            is equal to zero to the row and col arrays.
(11)        Set the column corresponding to the selected row index
            to NaN in Lines.
(12)        Increase lineNum by 1.
(13)        Increases Lines in the column corresponding to the row
            index by one.
(14)        The allocated project is the first columns value and
            the row is the student index set to allocated.
(15)     ELSE
(16)        FOR i equals 1 to the number of students. This scanned
               the rows for zeros.
(17)           ModData is equal to a single row of TData where
               Lines is equal to 0.
(18)           IF the number of zeros in ModData is equal to 1
                  THEN.
(19)              Set ZeroIndex to index of where Lines and TData
                  equals to 0.
(20)              Add 1 to the column in Lines where 0 was found
                  in the row.
(21)              Increase LineNum by 1.
(22)              Add project selection to the Allocation array.
(23)           END
(24)        END
(25)        REPEAT Lines 16 to 24 but swap rows and columns.
(26)     END
(27)  END
(28)  IF the size of the array is equal to the number of lines drawn,
      LineNum, the solution is found.
(29)     Completed equals 1.
(30)  ELSE
```

```
(31)      Completed equals 0.
(32)      Set ModData to the values of TData that correspond to
          where Lines is equal to 0.
(33)      Set ModIndex to the indexes of where Lines is equal to 0.
(34)      FOR each term of ModIndex
(35)         Minus the minimum value from each value in ModData.
(36)      END
(37)      All cells with a value of 2 in lines add min value to
          Them in TData.
(38)      FOR each term of ModIndex
(39)         Add the minimum value in ModData to the other ModData
             values.
(40)      END
(41)   END
END
```

The code works to reduce the cost matrix until a row or column contains a zero and then produces a solution, reducing the potential selection choices available with a cover matrix (Kuhn, 1955). A clear example of how this is achieved can be found in Kuhn's original paper on the process.

The program was expected to run in polynomial time with O(n^4) as its time complexity (Tang, 2017)

6.3 Analysis

The algorithm was run 10000 times, scored and timed to allow the curve fitting process to be as accurate as possible. With the average time taken of the data calculated, MATLAB's curve fitting toolbox was used on several equations to find ones that were appropriate and expected form the research (Tang, 2017). Table 9 holds this information.

Table 9. Analysis of curve fits for the Hungarian method.

Type	Equation	SSE	R-Squared	RMSE
Power	$1.19 \times 10^{-6}x^{2.319}$	0.01263	0.6897	0.003557
Exponential	$0.0008714e^{0.03092x}$	0.0132	0.6757	0.003636
Simple Power	$1.737 \times 10^{-6}x^{2}$	0.01264	0.6892	0.003558

It can be seen from the R squared values that the closest fitting curve to the data was the power equation and thus it could be assumed that the program, on average, ran with a growth rate of approximately n^2. The SSE and RMSE values were also exceptionally low and therefore provided a good fit for the data. On top of this the maximum data was taken from each input size and a line was also fitted using the same base function but allowing the variables to be recalculated. Equation 15 showed that the maximum increase rate of this function, and thus big O notation is O($x^{2.336}$) contradicting Tang's findings of O(n^4). It must be noted that this

curve fit did allow for some outliers above the best fit line meaning that it was possible for this algorithm to produce solutions slower than the predicted maximum time, but for this project that error is equal amongst all the experiments allowing them to be compared fairly and could be why this function is producing a lower than expected big O notation..

$$TMax_{Hungarian} = 5.598E - 6x^{2.366} \quad (15)$$

Figure 7 could be used in addition to equation 15 found the maximum number of students that this algorithm could produce in 3 days to be 270317 and using an average time, 501743 students. These programs are more than adequate for large data sets and only produce the best answer available. However, when using systems involving big data can matrices can be used over a few hundred megabytes, the point where 'normal' systems like Microsoft Excel fails (Gordon, 2005).

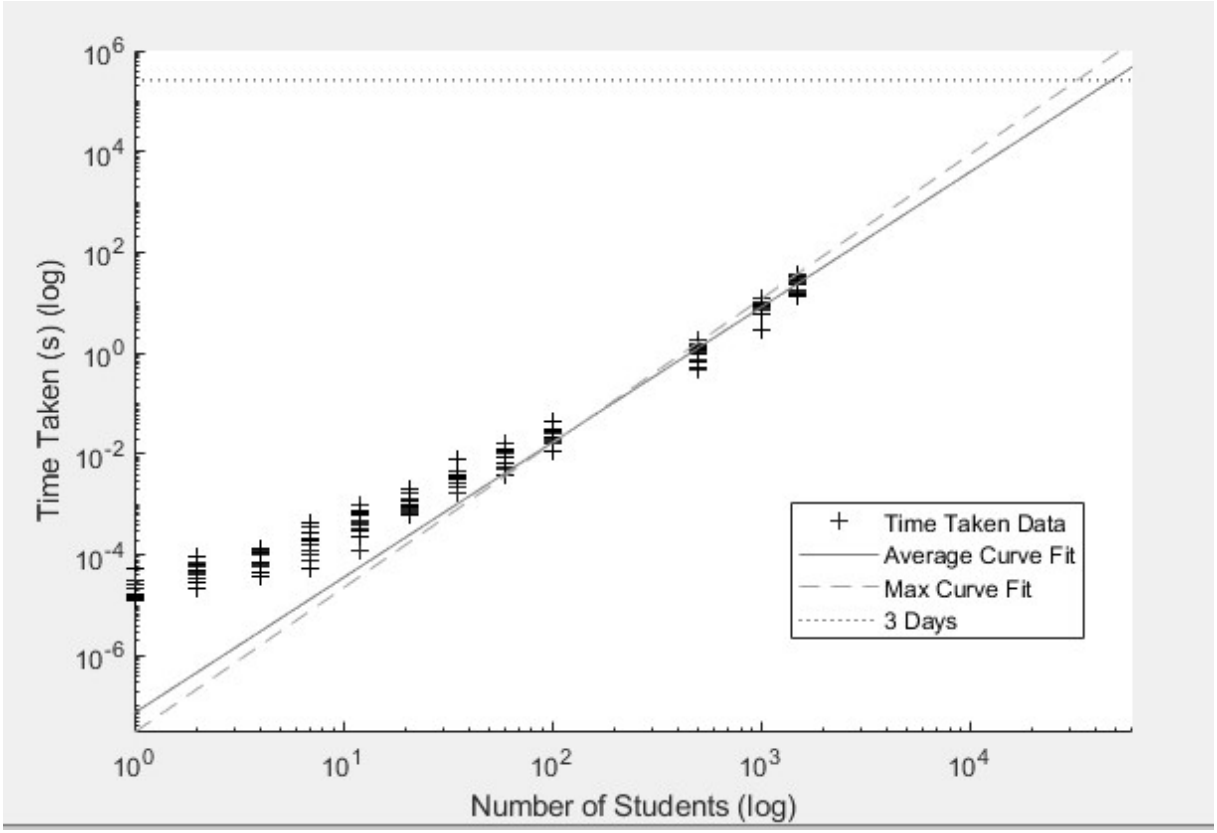


Figure 7, Changing student numbers when measuring time taken to complete an allocation up to 3 day.

The algorithm only finds the best solutions possible for the data it is given (Kuhn, 1955) and therefore it was important to prove that the code was functioning without any errors and the produced scores were correct. The program was therefore run 10000 times with 8 students, alike to the brute force method, and the histogram in figure 8 was created. The average solution was 1.668 very similar to the average solution of the brute force methods, proving that the method was only producing best possible allocations. The graph had a dip around the score

of 1.7 but this is because the cost matrixes were created randomly by the function described in section 4.2 and when smoothed out the data showed how both the Hungarian method and the brute force method produce almost identical results, but with the Hungarian being considerably faster.

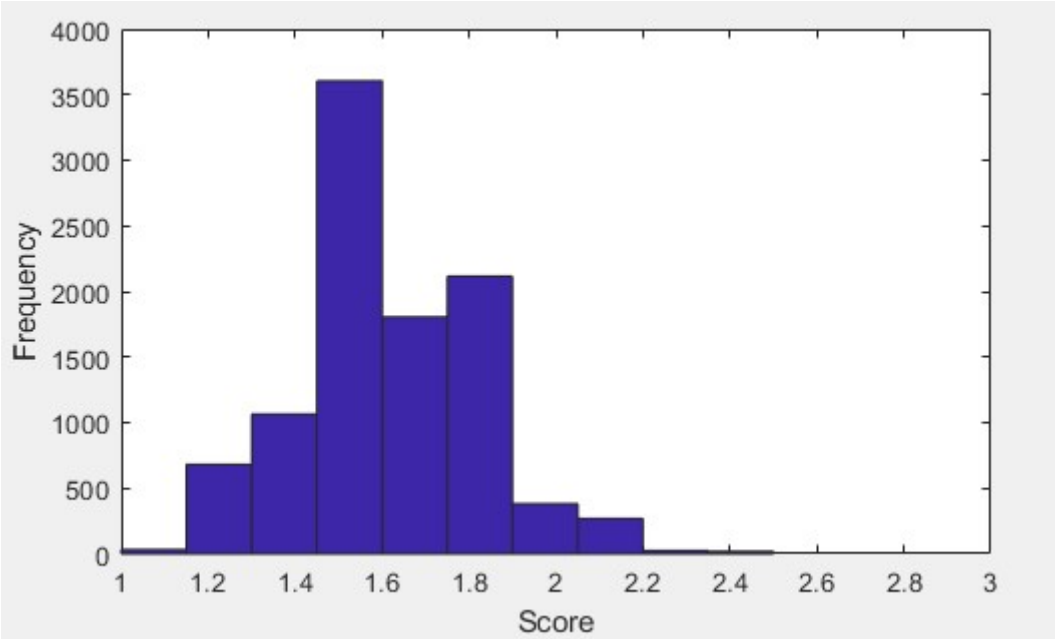


Figure 8, Histogram of the Hungarian method scores run 10000 times with an input size of 8 students.

The second goal of this project was to see how varying the number of choices would affect the algorithms. By increasing the number of choices each student could choose from 1 to 100, or from 1% to 100% choice capability it showed how these scores progressed. Figure 9 shows how from choices 1 to 9 the scores are dramatically increased As was discussed in section 5.2, the chance of the best solution allocating a project that is out of a student's chosen selection is high, thus increasing the scores at these low values.

In the same test the time taken for these allocations to be produced was also measured. As figure 10 shows, the time taken increased when the project choices were lower but only by approximately 0.35 seconds and was an inconsiderable difference when the base allocation took a maximum of 0.2 seconds. This could have been because more iterations of the method needed to be completed as less information is available to rank each solution due to the values in the random cost matrix. Interestingly with only 1 choice the time taken to produce these solutions were considerably lower than 2 choices because the data used is random and it may have therefore been easier to produce a solution. Therefor it was concluded that the choices have little effect on the time taken to run this algorithm.

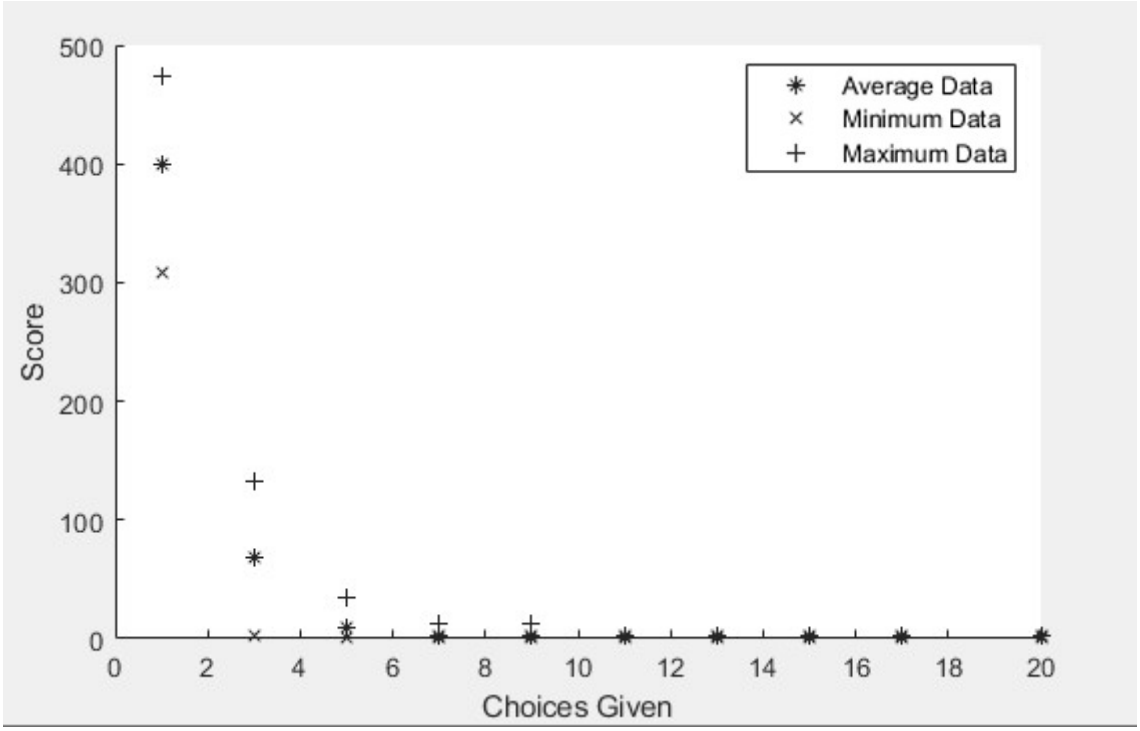


Figure 9, Graph showing how score varies with the number of choices each student can make out of 100 projects with 100 students.

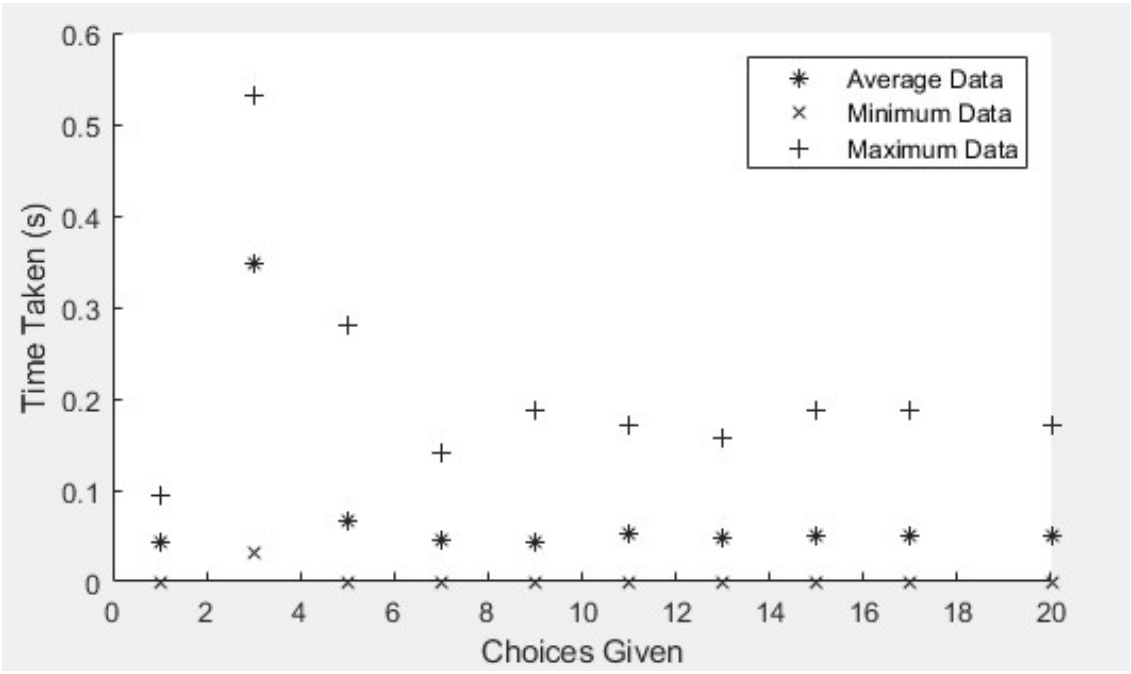


Figure 10, Graph showing how time taken to run the program varies with the number of choices each student can make out of 100 projects with 100 students.

7. Matched Pairs

While researching the linear assignment problem a solution was found as a MATLAB function called matchpairs(). The description of this function was stated as:

"M = matchpairs(Cost, costUnmatched) solves the linear assignment problem for the rows and columns of the matrix Cost. Each row is assigned to a column in such a way that the total cost is minimized. costUnmatched specifies the cost per row of not assigning each row, and also the cost per column of not having a row assigned to each column." (MathWorks, 2020)

The website uses the example of minimising the cost of assigning salespeople to fights and this is a direct comparison to the SPAP. Simply a cost matrix is inputted into the function as well as a cost penalty for any assignments that cannot be found. The larger this value is the deeper the algorithm searches for a solution and this cost is used as a penalty for not finding an allocation for a student (MathWorks, 2020).

The method is based on calculating the distances between nodes that are plotted on a graph and uses a function called equilibrate() that permutes and rescales so diagonal entities have a value of 1 and off diagonal enteritis have a value of between 0 and 1. According to (MathWorks, 2020) this greatly improves the efficiency of the algorithm. However, these concepts were not explored further in this paper and only the overall algorithm was evaluated.

This program was used as a confirmation for an algorithm for this problem that has been produced by a large company and so this reduces the chance of there being errors in the code and it is done in a more efficient way. It was expected that this would produce an answer the quickest.

7.1 Analysis

Once the algorithm was run for 10000 iterations a histogram of the scores produced was drawn, figure 11. With a mean value of 1.634 and a skewed score distribution identical to that of the brute force method and the Hungarian so can be presumed that this program only finds the best allocation for a given cost matrix. The slight difference in mean score and score histogram is because of the random input data that was used to prove overall effectiveness. In order to assure that the algorithm was producing the same results as the brute force and the Hungarian methods the worst- and best-case scenario data were used, and the scores were compared. All 3 algorithms produced the same results, solidifying the perfect answers being produced.

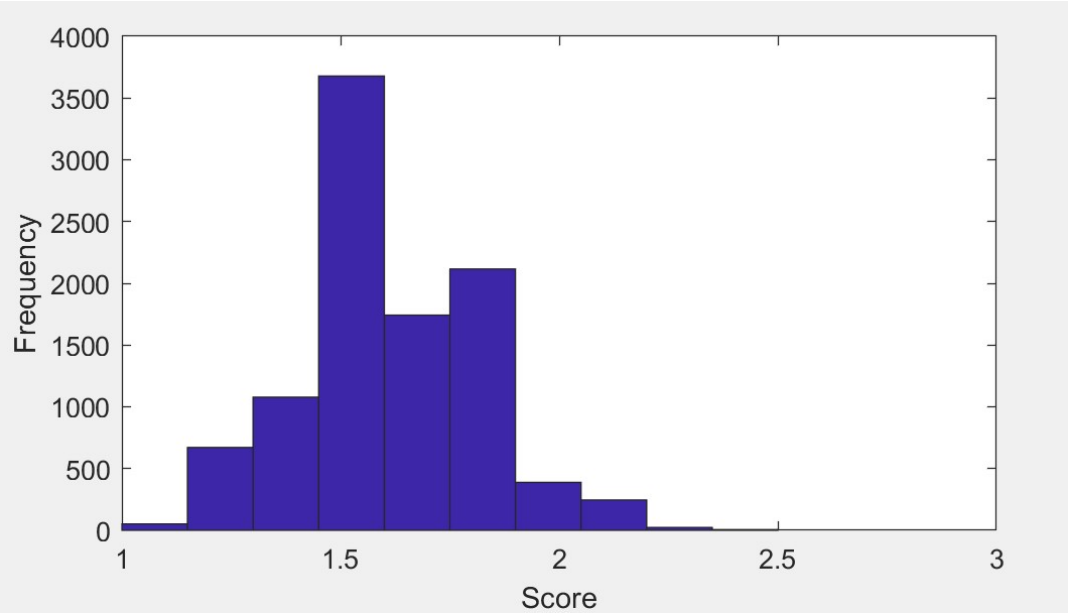


Figure 11, Histogram of the Match Pairs algorithm's scores run 10000 times with an input size of 8 students and unmatched cost of 1000.

With this data MATLAB's curve fitting toolbox was used to fit 2 curves, the average and the maximum so the time taken could be evaluated for large input sizes. The equations derived can be seen in equation 16 and equation 17. From this it can be stipulated that this algorithm increases the time taken to run, on average, to the power of 2.486 when the input size is increase by 1. This allows a potential user to estimate the time that a specific solution size may take but does not provide information on the big O notation for this algorithm and one of the criteria that is compared when a conclusion is formed. For this to be analysed the maximum curve fit line must be studied. With the stated equation it is suggested that the big O notation for this algorithm is O(n^2.491).

$$TAve_{Matched\ Pairs} = 1.994 \times 10^{-9} x^{2.486} \quad (16)$$

$$TMax_{Matched\ Pairs} = 2.154 \times 10^{-9} x^{2.491} \quad (17)$$

From figure 12 and equations above, the number of students that could be optimised in 3 days was 449267 but it could be expected, but not guaranteed, to solve for up to 475726 students. This is more than adequate for a typical university department.

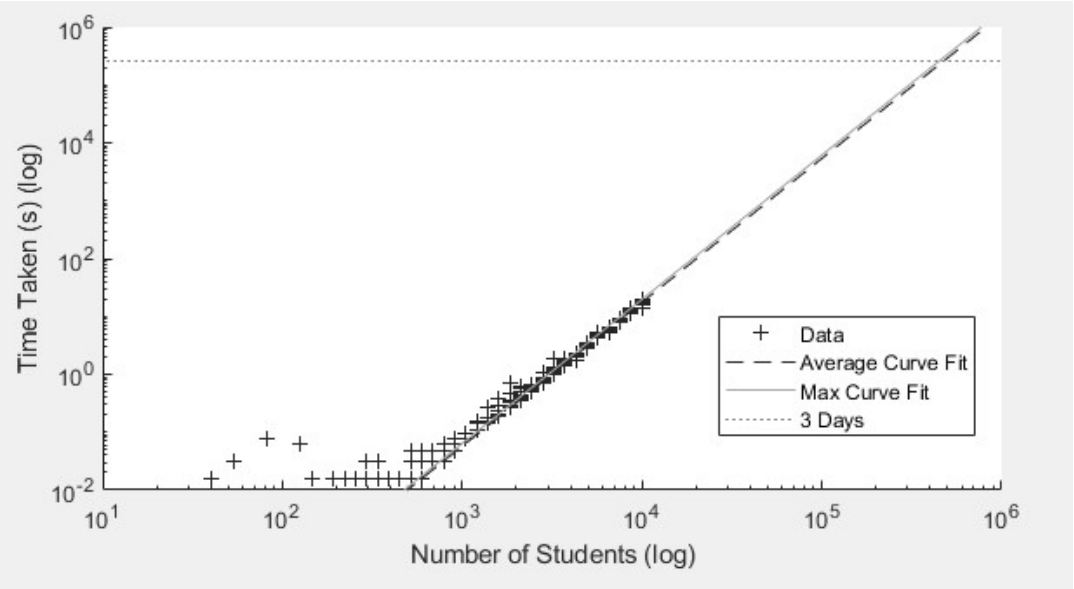


Figure 12, Changing student numbers in the Match Pairs algorithm when measuring time taken to complete an allocation up to 3 day.

Changing the choices available to each student showed that there was no effect on the time to run unlike other algorithms, figure 13.

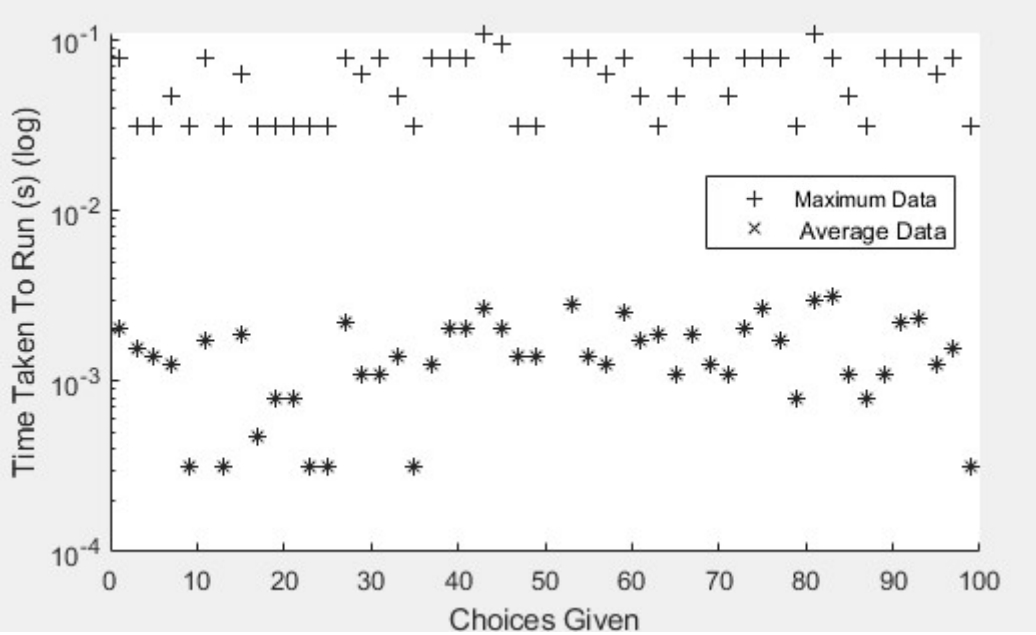


Figure 13, Changing the number of student choices in the Match Pairs algorithm when measuring time taken to complete an allocation with 100 students and 100 projects.

When the score was plotted against the choices given to the students, figure 14 it showed a sharp increase when the was between 1 and 8 choices and again this yet clarifies further the

effect allocating a student to a project that was not in their selection list has on the overall score. As discussed, the probability of the best solution only including projects that are chosen is higher when the percentage of projects to choices is low. Additionally, the difference between the maximum and minimum increases drastically between 3 and 8 choices. Showing that it the random nature of the input has a huge effect on the allocation score. Finally, it needs to be understood that this drastic increase in score could be unrepresentative of realistic student happiness and was designed to shown when students were being assigned projects outside of their choices. Once past 10% of the number of projects the score returns to allocating within all student choices.

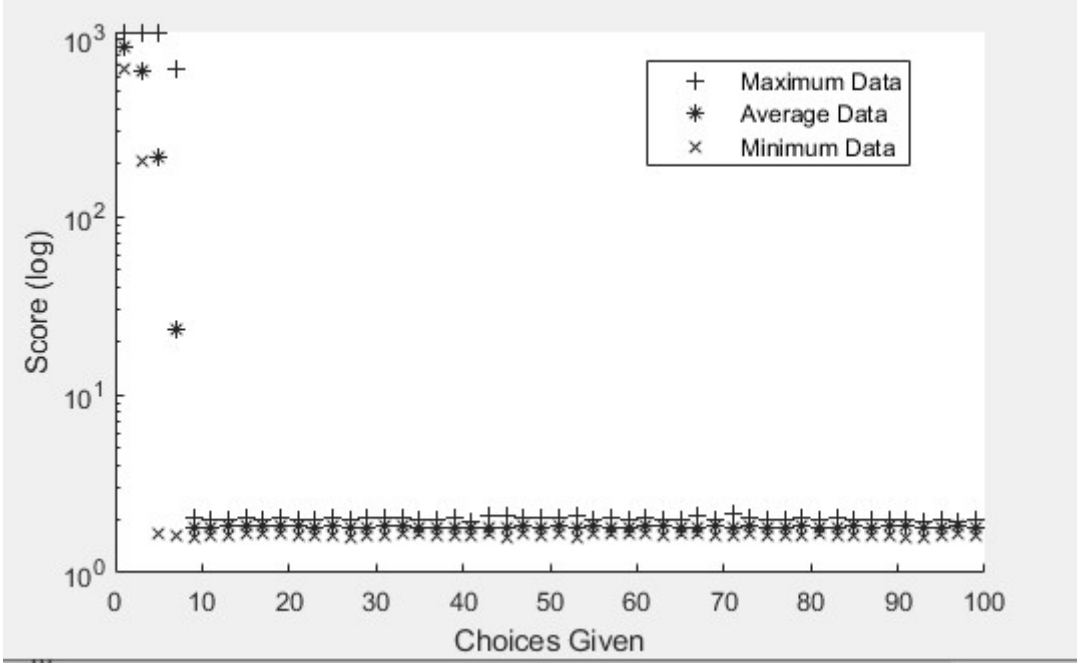


Figure 14, A graph to show how score of the Match Pairs algorithm is affected when the number of choices given to a student is varied.

8. Genetic Algorithm

A genetic algorithm is a computer program based upon Darwinian theory of evolution by survival of the fittest and has been developed by many authors revolving around 5 main principles, population, fitness, crossover, mutation, and solution finding (Simon, 2013) (Reeves, 2002). Each section has a large impact on the solution accuracy, the time taken to find that solution and the storage space needed to run the algorithm. An outline of the algorithm used in this paper is as follows:

Take a large population pool array, **P**, that contains genes, **G**, that each contain chromosomes, **c**. Over each iteration, often called a generation, score each gene between 1 and 0, being good or bad results respectively, and create a new population based on these scores. This is done by selecting 2 genes from the old population by a random distribution where it is more likely to select higher scoring genes than lower scoring one. These 2 parent genes are then combined in some way to create a new child gene and a possible mutation was applied. In this step there is a small chance that the genes chromosomes could be affected randomly to assist in providing diversity to the population is an idea that is discussed later. Finally, the gene is added to a new population pool and the process is continued until the new pool is full. This process completes the generation and is repeated to attain improved results (Simon, 2013).

In doing this, the best solution in each successive population pool should improve, or evolve with natural selection, much like finches on the Galapagos islands as documented by Darwin in 1858 in his book 'Origin Of Species' (Darwin, 1859) (Reeves, 2002). One of the most important notes from transferring natural evolution into computational evolution was that it took a long time for specialist adaptations to form in species to improve their fitness. This section will focus on how to improve the natural process when its components are converted to run on a computer to solve combination problems like the SPAP.

Genetic algorithms are a subcategory of evolutionary algorithms, EAs, that represent the wide range of disciplines derived from the theory of evolution and the idea of natural selection and genetics. Some of these include genetic programming and artificial life and are fields within themselves (Reeves, 2002). From its inception genetic algorithms were used to solve optimisation problems pioneered by Holland from 1975 in his book 'Adaptations in Natural and Artificial Systems' (Holland, 1975) but further progress in the field was made in the 1960s by several other authors (Reeves, 2002).

More recently algorithms like these have been used in design software and played a key role in the development of modern components that rely on an underlying optimisation problem. For example the 2006 NASA ST5 spacecraft used a generative optimisation algorithm to design an antenna for the shuttle in order to produce the best radiation patterns possible for the mission (Gregory Hornby, 2006).

Although complex and advanced genetic algorithms have been studied (Simon, 2013) a simple algorithm was used in this paper to evaluate the use of 'survival of the fittest' type iterative algorithms in solving permutation optimisation problems like the SPAP.

8.1 Populations

A population was used to group genes in a pool and is the basis of any genetic algorithm. Mathematically, **P**, is described as an array of genes, **G**, taking an overall size of $[N_p \times N_s]$ for this project (Equation 13). In terms of the problem, rows represented a project, columns represented each student and the values of each chromosome was the preference.

$$P = \begin{bmatrix} G_1 \\ \vdots \\ G_N \end{bmatrix} = \begin{bmatrix} c_{1,1} & \cdots & c_{1,N_s} \\ \vdots & \ddots & \vdots \\ c_{N_p,1} & \cdots & c_{N_p,N_s} \end{bmatrix} \tag{13}$$

The group must be selected randomly to gain as much search space accessibility as possible. In the book Genetic Algorithms - Principles and Perspectives : A Guide to GA Theory (Reeves, 2002) it is suggested to produce an initial population where the size of the population is large enough to access every possible chromosome in each gene within one crossover. Because of the random selection of the initial population and stochastic nature of the crossover process this size needed can only be predicted within confidence bounds, but is suggested to be 30 for most problems (Reeves, 2002). The method aims to produce at least one chromosome type in each part of the gene. To achieve this distributed population simply in this project a standard pseudo random generator was used. There is contradictory literature on how population sizes should scale with the size of the gene. Reeve's suggested that this relationship increases exponentially however 'a linear dependency was adequate' (Reeves, 2002).

The population that was created consists of permutations of choices, where 2 projects cannot be picked twice, providing additional complexity to the analysis of the problem. Reeves discusses the permutation flow shop sequencing problem, PFSP, similar to the SPAP that is being considered in this paper and notes that the crossover algorithms would be effected, discussed in 8.3 (Reeves, 2002). However, initial populations must obey the permutation restrictions to create an accurate initial search space. The method used adds each random chromosome into the gene only if it has not already been used, creating a random permutation or alternatively a random lexicographic number could have been selected.

The size of the population, thus the number of genes used, effects solutions scores dramatically. For example, if a given population was too small there may not be the diversity required to produce effective final results because there are no successful chromosomes in the initial set and thus, without a high mutation chance, the solution will converge to a local minima and not produce optimal results. Comparatively, using a population size that was too large increases the computational memory required to run the program and was not able to find an optimum solution because it could not perform as many iterations as a small population size could (Reeves, 2002). A population can be seeded with genes that are known

to be a good starting point for a solution, but this requires external knowledge of the process being studied or the use of other optimisation methods to create the population (Michael Mutingi, 2017). This was found to reduce the resulting quality of solutions when used and required external processing power reducing efficiency. It was also said to lead to premature convergence (Reeves, 2002). Therefore, it was chosen to randomly select the population over an equally random distribution.

Using the methods described above, the algorithm was run several times with different population sizes for 900 seconds to see how well it performed under each situation. The theory of only requiring a low population was confirmed when the population size was relatively small at 10, when using 100 students to optimise (often referred as the word size) for each gene, failed to produce an optimal solution and converged to a local minima with a score of 0.8. The search space was not being explored thoroughly thus leading to a poor solution. The target score was said to be above 0.9 as the aim was to improve the current score of 0.845, showing that a population size needed to above this for a suitable solution to be found. However, because this method is stochastic, there is a probability that it may or may not reach this solution in a single run of set time. Once the population size is increased to 50 an appropriate solution was found with a score of 0.96 in 900 seconds. It must be noted that the time taken to reach a score of 0.9 was much faster, taking 91 seconds to reach an appropriate solution. Therefore, it would be possible to provide the user with a suitable allocation much faster and could be stopped earlier, but this will be explained in 8.5. However, when compared with a population size of 400, between 0 seconds and 400 seconds the larger population produces a lower score, as each generation took longer to run. After this point the solution produced had a higher score, suggesting that a larger population size, although taking longer to run, would ultimately produce a higher quality allocation. This suggested that there exists an optimum population size for a given time limit and number of students.

Comparatively a relatively large population size of 6400 can be seen to reach an equally poor solution in 900 seconds in 15 because of the large processing power that is needed to complete each iteration and therefore the number of iterations is reduced drastically. Therefore, there is an optimum population size for a given running time that the program can achieve, this population size would have the highest probability of producing the best score.

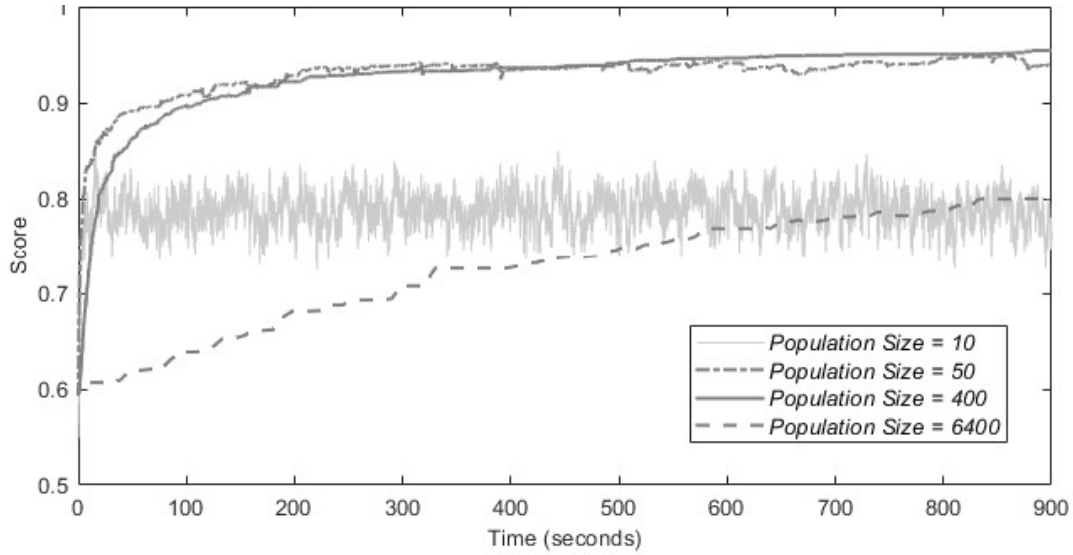


Figure 15, Development of score over time while varying population size with 100 students, 100 projects and a mutation rate of 0.005.

However, if the program was run for a longer period of time it was hypothesised that eventually it would produce an allocation with a higher score than lower population sizes, thus the program was extended and run for a longer period of time to produce figure 16 and a solution just as good as lower populations sizes.

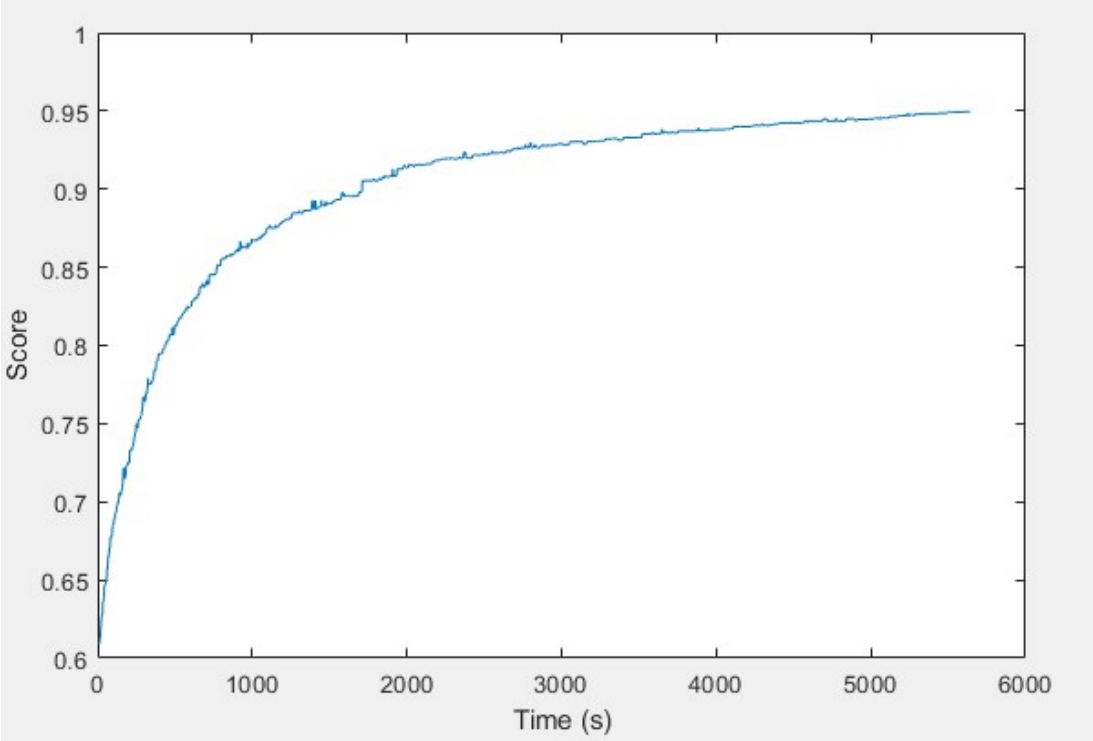


Figure 16, Score of a genetic algorithm over time with a population of 6400, a mutation rate of 0.005 and 100 students and 100 projects.

Figure 16 shows the score of a genetic algorithm with population of 6400 over a longer period, showing that large population sizes can reach high scores if given enough time. The program was set to run until it hit a score of 0.95 and took 5639 seconds, roughly 1 and a half hours. Additionally, the gradient of the line towards the end of the iterations is steeper than that of smaller population sizes in 15. This was because with a larger population there was higher diversity between the genes, allowing the algorithm to have more possible allocations and dramatically reducing the chance of converging to a local minimum. Considering the goal of this project was to find an allocation in under 3 days large population sizes could be considered however smaller populations still produced the target score of 0.95 more quickly. The population size required depends on the number of students that are needed to allocate, and this relationship is not explored further here. However, for around 100 students using populations between 50 and 400 is recommended. Increasing the size will increase the time taken to reach an appropriate solution but will be more likely to find a fully optimised solution if required and reducing the population size reduced the time taken to converge but increases the likelihood of finding local minima.

Alternatively, after continued research, it was suggested that using a Latin Hypercube to populate the initial population reduced the processing times of stochastic methods like genetic algorithms by improving their efficiency (A. Olsson, 2002). This is a principle that is used to build the initial population and provides a statistically more distributed population, increasing the accessible search space. The method requires splitting the search space into N sections and then selecting points randomly in each section (A. Olsson, 2002). This ensures the search space is more evenly covered and reduces the possibility of converging prematurely to a local minimum.

8.2 Fitness

Calculating the fitness of each gene in a population is done so a probability can be assigned to each gene, used in the crossover 8.3, letting higher scores be chosen more often and carrying on the better genes to the next iteration. The total cost, **C,** is calculated here by summing the costs of each chromosome that corresponds with the allocation, c_i, equation 14. This was then done for each gene and stored in an array, $\hat{C}$ vector 15 with the size $[N_s \times 1]$. These values range from N_s, the size of allocation array, to N_s^2.

$$C = \sum_{i=1}^{N} c_i \tag{14}$$

$$\hat{C} = \begin{bmatrix} C_1 \\ \vdots \\ C_{N_s} \end{bmatrix} \tag{15}$$

It was then normalised, so the minimum score was zero, by subtracting N_s and then scaling the values in the array to between 0 and 1. Finally, the scores were reversed so a perfect score is 1 and the worst possible score is 0, this made it easier to visualise and allocate probabilities.

Equation 16 shows how to do this operation. Let $\hat{S}$ be a vector of size [Nsx1] shown in equation 17.

$$S_i = \frac{-C_i}{N_s - {N_s}^2} + N_s + 1 \tag{16}$$

$$\hat{S} = \begin{bmatrix} S_1 \\ \vdots \\ S_{N_s} \end{bmatrix} \tag{17}$$

This value was used to display how well a population was doing, but to assign a score the array $\hat{S}$ needed to sum to 1 so it could be used as a probability distribution in MATLAB. The method used was a static linear assignment from the sum of the allocation 19 to probabilities, Pr, meaning that there is no bias to giving higher scored assignments some extra probability of being selected. The basic method used is summarised by equation 18 (Mahmood, 2018), and is a common practice in genetic algorithms in data science applications.

$$Pr_i = \frac{S_i}{\sum_{i=1}^{N_s} S_i} \tag{18}$$

$$\widehat{Pr} = \begin{bmatrix} Pr_1 \\ \vdots \\ Pr_{N_s} \end{bmatrix} \tag{19}$$

Using this math, a function was constructed to evaluate each gene in the population and assign a probability to it by inputting the current population and the cost matrix, for the scores to be analysed, and outputting the scores.

The fitness function used plays a large role in the evolution of the program and is based on the objective function. It is stated as an example in Reeves 'Genetic Algorithms – Principles and Perspectives' that it is important to consider the scale on which the fitness is being evaluated, '…values of 10 and 20 are much more clearly distinguished than 1010 and 1020' and goes on to suggest that scaling can be an important tool in normalising the results. Additionally converting an objective function into a fitness function may require it to be transformed from a maximisation to a minimisation function (Reeves, 2002). Thus, the mathematics described in this section were used to create a suitable fitness function for the program. Reeves mentions other forms of fitness allocation, such as ranking, by simply sorting into a list and not needing to scale values or tournament selection, by comparing randomly selected genes at the selection stage.

On conducting further research, it was found that there are several ways to improve a fitness function in a genetic algorithm to reduce the time taken to find a solution. These methods can be classed as A, static selectivity scaling and B, dynamic scaling selectivity (J.A. Lima, 1996). Static selectivity scaling applies a scaling function, to the costs of the genes increasing the probability of selecting higher scoring genes is increased. From the literature it can be seen that often an exponential function is used for this process, drastically increasing the probability

of selecting 2 higher scoring genes for crossover. An example of such an equation can be seen in 20 where σ is a constant and c is the cost of the element. Where "the higher the σ is, the higher the probability gain" (J.A. Lima, 1996).

$$f(c) = e^{-\sigma} \tag{20}$$

Additionally, Lima uses a dynamically scaling function to apply a relationship between the minimum cost element, c_{min}, and the mean cost of the gene, φ. Conversely to static selectivity, dynamic selectivity function is modified based on the values of the cost array, C. Sown in equation 21 the scaling function changes based on the distribution of the mean and minimum score values in c.

$$f(c) = e^{\ln(\phi)\frac{c-c_{min}}{\bar{c}-c_{min}}} \tag{21}$$

The literature states that this method is comparatively more resource intensive than static scaling method and by comparison to linear scaling both static and dynamic scaling drastically improved the scores more quickly in a given number of generations.

8.3 Selection and Crossover

A genetic algorithm needs to have a good mix of solution finding capabilities and exploration to search as much of the search space as possible while still converging to a solution. Selection and crossover play a key role in providing these characteristics to the function. Once all genes in a given population, P, have been assigned a probability of selection, Pr, over a distribution, often 2 genes are selected over this distribution randomly by methods such as Roulette Wheel selection (Reeves, 2002). Selection relies on picking better solutions more frequently than the lower scoring solutions in order to carry on the data held in the genes to the new population.

Roulette wheel selection, RWS, probabilities are assigned to each section of a circle, summing to 1, and uses a randomly generated number from 0 to 1 to simulate an angle on this circle. Because higher probability sections have the chance of being selected more often this is a good method of selecting genes. 17 shows from Reeves' book what this may look like.

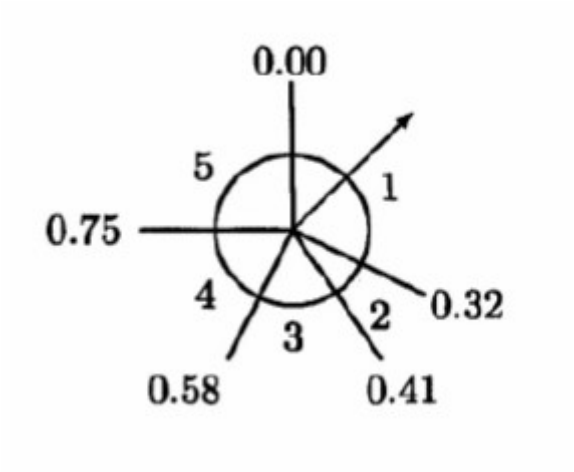


Figure 17, Roulette wheel selection, considering the genes to select are 1,2,3,4 and 5 and the angle of the section represents its probability, Taken from (Reeves, 2002)

Reeves suggests using a more advanced version of RWS in his book, but this was not implemented here, as a MATLAB function, randsrc(), exists that was convenient to use. Once RWS selected 2 genes the process of crossover began.

Crossover is the process of creating a new gene, for the next population, by mixing the chromosomes of 2 selected parent genes. This can be done in many ways but it was most common in the literature to use 1 (Reeves, 2002) or 2 point (Naima Hamid, 2018) crossover methods. The principle of both is splitting the 2 parent genes and combining them to form a new gene known as the child that will be inserted into the new population in one or 2 places. In this paper the focus was on one-point crossover and can be found in an illustration in figure 18.

Parent Gene A				
A_1	A_2	A_3	A_4	A_5

Parent Gene B				
B_1	B_2	B_3	B_4	B_5

New Child Gene				
A_1	A_2	B_3	B_4	B_5

Figure 18, The creation of a child gene by 1-point crossover from 2 parents.

However, the solutions to the SPAP is a permutation and must not contain repeats. This makes the problem of crossover considerably harder, as standard 1- or 2-point crossover methods do

not preserve this property when the genes are combined. It was suggested to use a permutation specific crossover method that involves crossing over the genes in segments and removing duplicate integers and replacing them with 'legal' alternatives (Reeves, 2002). The term 'legal' is defined as a problem specific characteristic and in the SPAP this would be any unused number from 1 to the number of students. An improved version of permutation crossover was presented by Harper and involved looping through the new child gene, one chromosome at a time. Below this is described in a pseudo code adaptation of Harper's work (Paul R. Harper, 2005).

```
(1)   FirstGeneIndex = distributed random selection using
      probabilities
(2)   SecondGeneIndex = distributed random selection using
      probabilities
(3)   GeneU = Select the gene from the population with FirstGeneIndex
(4)   GeneV = Select the gene from the population with
      SecondGeneIndex
(5)   ProbabilityU = The corresponding probability of selection of
      GeneU
(6)   ProbabilityV = The corresponding probability of selection of
      GeneV
(7)   Initialise Child vector with zeros
(8)   Initialise Available vector with integers 1 to the number of
      students
(9)   FOR j = 1 to the number of students
(10)     IF GeneU(j) and GeneV(j) is not already in Child
(11)        IF GeneU(j) is equal to GeneV(j)
(12)           Child(j) is set to GeneU(j)
(13)           Remove GeneU(j) from Available
(14)        ELSE
(15)           IF ProbabilityU is greater than ProbabilityV
(16)              Child(j) is set to GeneU(j)
(17)              Remove GeneU(j) from Available
(18)           ELSE
(19)              Child(j) is set to GeneV(j)
(20)              Remove GeneV(j) from Available
(21)           END
(22)        END
(23)     ELSEIF GeneU(j) is not in Child but GeneV(j) is in Child
(24)        Child(j) is set to GeneU(j)
(25)        Remove GeneU(j) from Available
(26)     ELSEIF GeneU(j) is in Child but GeneV(j) is not in Child
(27)        Child(j) is set to GeneV(j)
(28)        Remove GeneV(j) from Available
(29)     ELSE
(30)        Child(j) is set to a random value in Available
(31)        This assigned value is removed from Available
(32)     END
```

```
(33)  END
```

This is the method that was used in this paper's genetic algorithm. It should be noted that there are a few drawbacks to this method. For example, the chance of using the probability of a gene to allocate a chromosome is reduced when the code reaches the end implying that students at the end of the allocation will have reduced satisfaction. Although the gene is chosen by its fitness the satisfaction along the allocation would not be linear.

Upon further research it was discovered that there are other ways of representing crossover, such as on a graph and using nodes to represent solutions and paths between these nodes as indicators for how probable this solution would be. Although abstract, ideas like this are being used in permutation problems today (Alberto Moraglio, 2007). Additionally, advanced concepts like inbreeding, increasing the chance of a local minima being found and a reduced diversity while the program is run. However, these ideas are not explored in this paper.

8.4 Mutation

Mutation plays a key role in maintaining diversity in a population throughout the generations. In biology, the processes that these algorithms are based upon, mutation is the process of slightly altering a segment of DNA in a gene. This is a relatively rare occurrence and can lead to positive and negative effects (Cambridge Dictionary, 2020) but within a computational algorithm this process keeps the diversity in the population as high as possible, allowing the population to access a wider range of solutions without converging prematurely (Simon, 2013) (Naima Hamid, 2018).

The process is often done in conjunction with the crossover stage and is defined as the small random chance of changing one or more chromosomes in a gene randomly. Because of the close relationship between crossover and mutation Reeves argues that the simplicity of a mutation function compared to a crossover function could increase overall performance (Reeves, 2002). However, other sources suggest that mutation and crossover are important (Simon, 2013). Reeves also suggests that mutation can also be adjusted to only mutate chromosomes to values close to the current value to preserve as much data as possible. However, this was more suited to problems like the traveling salesman problem, TSP, where close points would be more likely to improve performance rather than a random selection. In the SPAP values numerically close to current chromosomes hold no relevance in information.

The mutation rate could be set to achieve different goals, with an increased rate, converging to a random search algorithm, where information was lost between generations when random chromosomes were created but the search space is explored more thoroughly. A rate too low mitigated the effects the mutation was trying to achieve, leading to convergence on local minima and an increase level of 'in-breeding' (Simon, 2013). 'In-breeding is when 2 genes that are identical crossover, thus producing the same gene for the next population. Although this

shows a possible solution has been found, the program ceases to improve upon the previous generations, potentially leading to convergence at a local minimum.

Mutation provided a problem for a permutation problem, such as the SPAP, because when a random chromosome was replaced it would lead to duplicates in the allocation, breaking the conditions of the solution. Therefore, it was suggested by Harper to incorporate this into the crossover function. Section 8.3 shows the crossover pseudo code without the addition of the mutation and is a good reference to use when discussing this adaptation, the full annotated code can be found in the appendix E. An AND statement was added testing if a random number between 1 and 0 was above the mutation rate. If this random number fell below the mutation rate then the program would act like both parent chromosomes were in the child already, and thus select a random integer from the variable **Available**. This allowed the allocation to incorporate mutation while still satisfying the constraints of a permutation allocation problem (Paul R. Harper, 2005).

There is no correct mutation rate, although considerably low (Simon, 2013), but there has been guideline equations for finding rough rate for a given population and the length of the chromosomes in the algorithm (the allocation size). Equation 22 (Paul R. Harper, 2005) shows that the mutation rate is inversely proportional to the population size and inversely square rooted to the length of the genes.

$$Mutation\ Rate = \frac{1}{Population\ Size\ \times \sqrt{N_p}} \tag{22}$$

A graph was produced to prove that equation 22 for the mutation rate produced suitable values for the SPAP and to also see how the rates effected solutions when they were changed. Figure 19 shows how varying the mutation rates from 0.1931 to 0 effects the scores of the algorithm over 180 seconds. It was chosen to run the programs for a shorter time because most of the phenomena took place during the rapid start of the genetic algorithms.

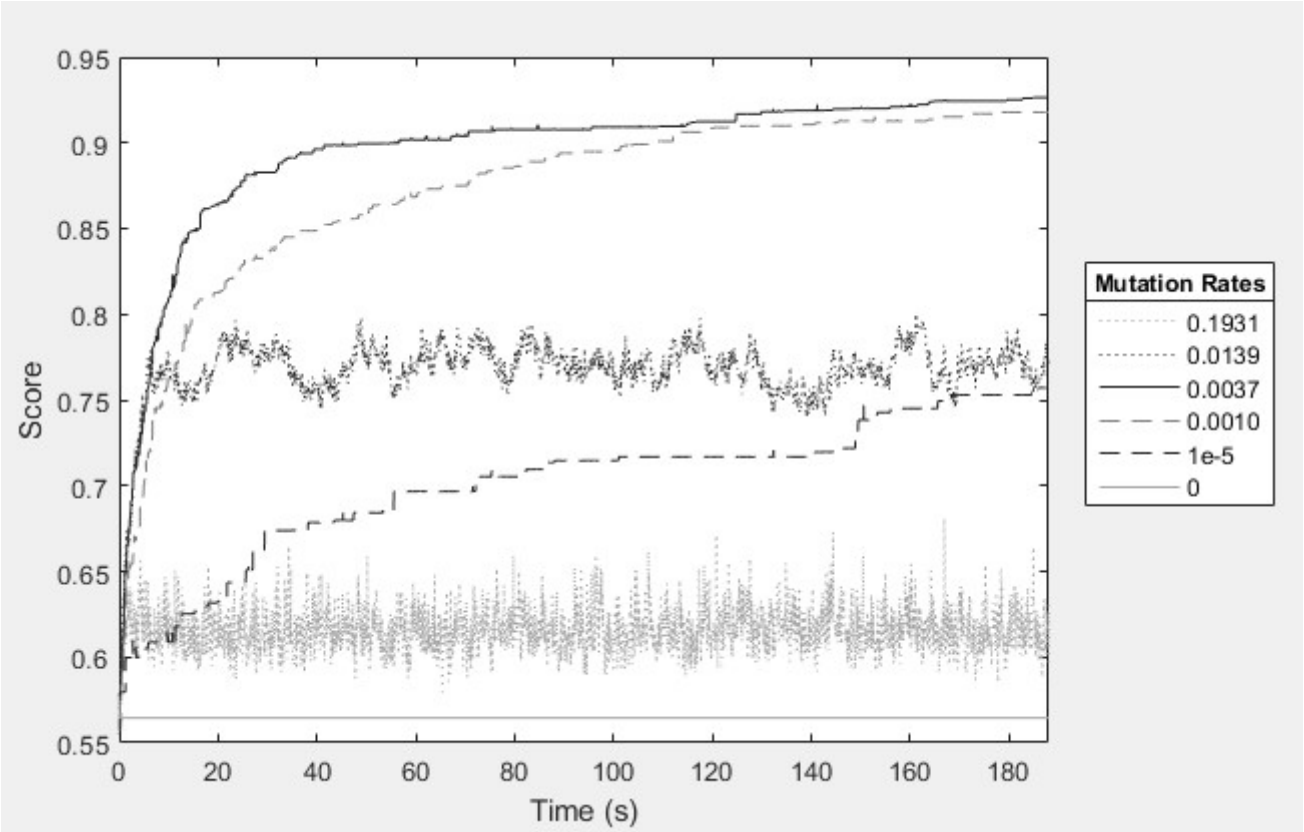


Figure 19, A graph showing the improvement of score over time for varying mutation rate. (100 students, 100 projects and population of 100)

The higher the mutation rate was, the lower the overall score became because too much mutation was happening in the population and thus the important data was lost between generations. Additionally, rates that were too low and tended towards 0, started to plateau increasing the time taken to produce adequate solutions often getting caught on local minima for long periods of time, backing up what Simon discussed in his book (Simon, 2013). This graph suggested that there would be an optimal mutation rate for a given population and gene length, but as discussed, because of the stochastic nature this does not exist. Equation 22 was shown to produce a high scoring solution, however not the best, and reduces the variables a user needs to input into the function.

Further research suggested by Reeves showed that additional literature had mixed opinions on the importance of mutation, some calling it a 'secondary function' but he concluded by saying it was problem specific. Additionally 'The Elements Of A Genetic Algorithm' showed evidence of several more advance mutation functions being developed by other authors that change the mutation rate to vary the diversity in a population as a program runs (Reeves, 2002).

8.5 Solution Finding

Genetic algorithms are iterative by nature and therefore it was important for them to have some stopping criteria if used in an application to produce a solution. Some ways that this has been done is by a set number of generations (Paul R. Harper, 2005), a select run time or a threshold value for the best fitness score (Simon, 2013). The use of set generation caps was not particularly useful in the context of the SPAP as the number of students and projects varied when used by different universities or schools. This meant that without being able to find a predicted time to run the user would have no idea how long the program would take to produce an allocation or if the allocations they received were good enough because of the random nature. Similarly using a threshold fitness score, eliminates the possibility of finding a poor answer as the program would run until an appropriate allocation was found. However, there is no guarantee that the algorithm will produce an answer above the threshold, especially with large data sets, and in addition of a user not knowing what scores would be acceptable. For example, a user with a large data set of 5000 students may only be able to expect a score of 0.7 as the complexity increases the chance of student choices being alike. Therefore, it was chosen to use the run time of the algorithm to terminate the program. This method allowed the user to specify the maximum time they must produce an allocation and thus makes the code more applicable to a wide range of situations. Some universities may have 3 days to run an algorithm whereas some may only have an hour.

In the program, this was achieved by using the cputime() function in MATLAB, recording the time when the program is initiated and the time difference is then checked on each generation until the time limit was reached (MathWorks, 2020).

The entire code for the genetic algorithm can be found in appendix E.

8.6 Analysis

To analyse the efficiency of the program it was important to predict how long this algorithm would take to run an increasing number of students thus showing how efficient the program was. The complex nature of genetic algorithms meant that there was a lot a variability in the performance in these measures based on factors like population size, mutation rates and stopping criteria. To analyse this program alongside other algorithms a standard set of these variables had to be used, potentially hiding the maximum performance that could be achieved. The population size was set to 100 because increasing the input size, the number of students that need to be allocated projects, increased the number of possible permutations and it was expected that more search space would be required for a good solution to be found. As it was found using the equation to modify the mutation rate with a changing number of students allowed the rate to be close to the optimum without needing to specify a value for each run of the program. Each data point was produced 5 times and an average was taken to increase the accuracy of the results. Once this data had been collected MATLAB's curve fitting toolbox was used to predict how long the algorithm would take with larger numbers of students. 4 types

of equation were used, shown in table 10, and the 3 error detection methods were used to evaluate how well each line fitted the data.

Table 10, Analysis of curve fits for the Genetic algorithm.

Equation Type	Equation	SSE	R-Squared	RMSE
Exponential	175.6*exp(0.004061*x)	1.04E+06	0.9899	247.1
Polynomial Degree 2	0.009114*x^2-1.383*x+44.73	2.24E+05	0.9978	118.4
Polynomial Degree 3	8.354e-06*x^3+0.002339*x^2-0.02792*x+2.433	2.92E+04	0.9997	44.09
Power	0.00005331*x^2.703	1.14E+05	0.9989	82.03

From the table the 3rd degree polynomial produced the best results in terms of SSE, R-Squared and RMSE values. Because of the random nature of genetic algorithms, the expected results varied considerably on each run, producing data that fluctuated. This favoured the polynomial curve fitting because it could change direction to accommodate these changes, fitting the data more closely and producing the best error values. However, this behaviour was not desirable and led to overfitting of the curve. Therefore, it was chosen to use a power to approximate the line. Seen in figure 20, a polynomial fits the data more accurately with low student numbers but both lines estimate a similar number of students that can be allocated in 3 days, 3252, to be expected. This number represents the expected values and it can take longer to produce the solution. Therefore, the maximum of each point was taken, and a curve of best fit was applied to produce an equation for the worst-case time taken, 28. From this the time complexity of this genetic algorithm with these settings is O($n^{2.758}$) when the input size is adjusted to the program. The equation was used to calculate the number of students that could be assigned whereby there was no chance of the solution taking longer than the allotted time in 3 days as 2972.

$$TMax_{Genetic} = 0.00006833x^{2.758} \quad (28)$$

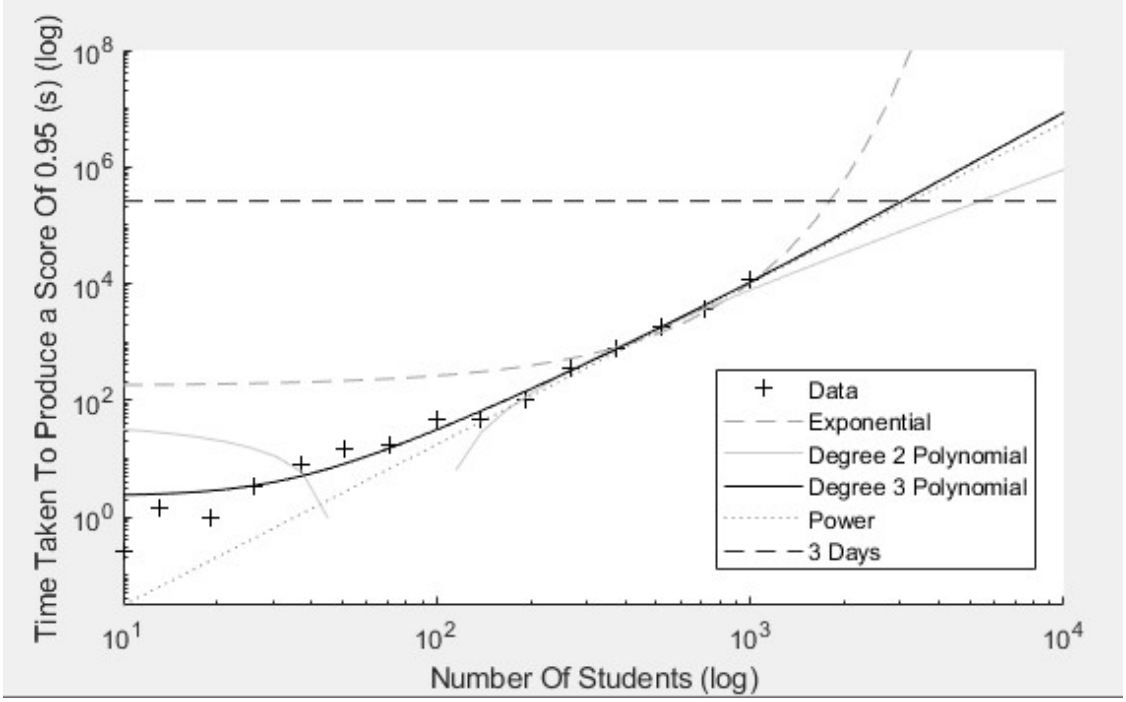


Figure 20, a graph showing time taken to reach a score of 0.90 for a varying number of students.

It was important to note that the time taken to find a solution could change drastically if the population size is changed for a given input and more research needs to be completed to find optimum values for each input size.

Students are not required to rank all their choices for every project available to them, so a graph was produced using 100 students, 100 projects and using a population size of 100 while changing the choices available to the students to see how this would effect time and scores. For this experiment the genetic algorithm was modified to stop after the score did not change for 10 iterations, showing that the evolution had slowed down sufficiently and would not alter much from the proposed solution. In this way the time taken to find a solution and the score of this solution could be measured.

Looking at figure 21 the score steadily increased when the choices are increased showing that a better solution can be found when the students get to choose a larger number of projects. However, because this algorithm does not necessarily find the best possible solution the chance of allocating a project outside of a student's choices was relatively high for many of the allocations.

When the time taken was plotted against the choices available there was a drastic decrease in time taken for the algorithm to complete, figure 21. The high number of unselected choices meant that the diversity in the cost matrix was very low leading to local minima being found quickly and because of this the algorithm was stopped prematurely. This can be seen more clearly in figure 21 where the time taken to produce the allocation is plotted against the score of the allocation. It clearly shows that the chance of producing a low scoring allocation is increased when the time taken is reduced. Therefore, the longer the algorithm is run the better solution it produces. Conversely, producing a high score, because of the stochastic nature of this program, can take a wide range of times.

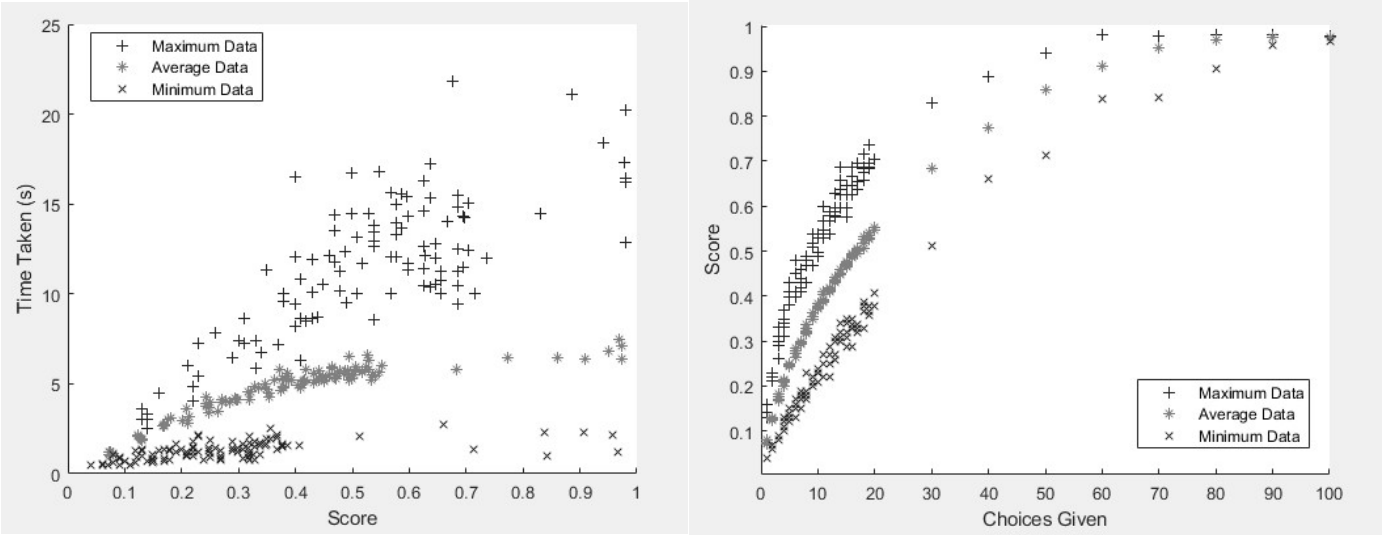


Figure 21, Varying the choices given to students with 100 students. (Left: A comparison of the score to how long the solution took) (Right: A comparison of how the score is affected when the choices given to students are changed).

9. Comparison Analysis

Finally, the time taken to run each algorithm was compared against one other in figure 22, where the maximum time taken to run each algorithm was taken and the corresponding line of best fit drawn. This allowed the scaling of each function to be seen easily and the most suited algorithm could be selected. The data plotted for the brute force, Hungarian and matched pairs algorithms was the time taken to produce a solution as they only produce the best possible solution. Controversially, the genetic algorithm does not find one best solution and therefore a cap of 0.9 was used to portray a suitable allocation and was decided because of the current score produced by the University of Surrey's of 0.845 and wanting an algorithm that at least beats this.

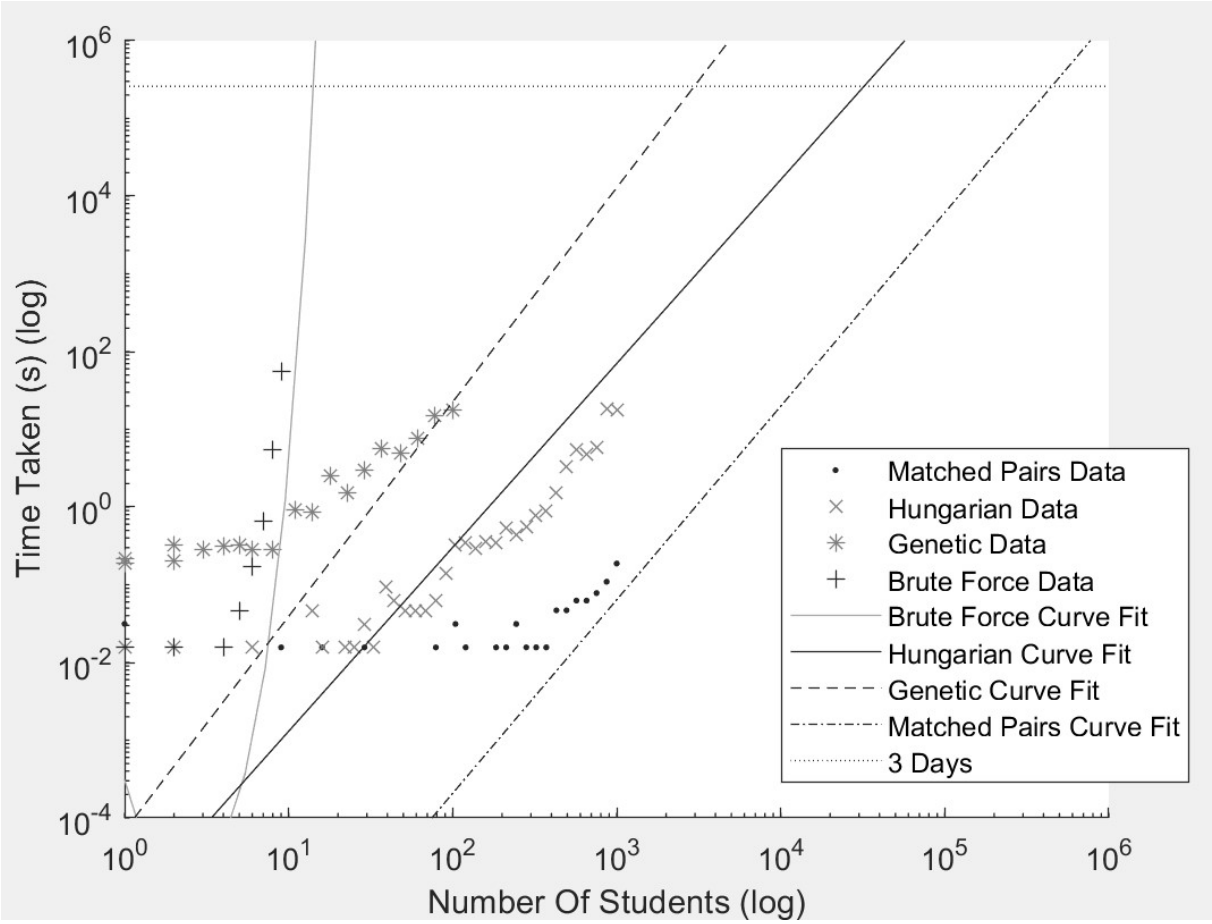


Figure 22, Comparison of algorithms used to show predicted time taken up to 3 days for varying student numbers.

To assist in the comparisons table 11 was produced to compare key statistics of each algorithm. This included the big O notation found in this paper, a comparison to suggested big O notation from a variety of literature sources and the number of projects that could be allocated to students in 3 days.

Table 11, Comparison of the algorithm's used statistics.

Algorithm	Big O Notation	Suggested Big O Notation	Projects Assigned In 3 Days
Brute Force	$O(n!)$	$O(n!)$	14
Genetic	$O(n^{2.758})$	-	2972
Match Pairs	$O(n^{2.758})$	-	449267
Hungarian	$O(n^{2.366})$	$O(n^4)$ or $O(n^3)$	270317

The hierarchy between the best performing algorithms seen on figure 22 and table 11 indicates clearly that the matched pairs algorithm performs the allocations in the fastest time. The program has the lowest line on the graph and therefore is the fastest. However, the data in the table suggested that the Hungarian method scales better when the student sizes are increased. This does mean the Hungarian is more efficient in terms of big O notation, but the scaling applied in order to create these equations means that the matched pairs function produced solutions much faster. This could have been because the match pairs function was provided by MATLAB (MathWorks, 2020) and therefore would have been produced to be efficient for their users, compared to the code written for the other functions were produced without knowledge of what computer procedures are faster than others to satisfy a goal. The genetic algorithm was considerably slower than the other algorithms because of its stochastic nature with the slowest of the non-trivial algorithms and a big O notation of O(n^2.758). This was expected but they have the advantage of producing alternative options that are not optimal but still satisfactory. Genetic algorithms can also be very complex and deep research into how to increase solution finding time could be done to improve the run time. The brute force method was used as a comparison and, as expected, performed exceedingly poorly with one of the worst big O notations possible (Bae, 2019). The current time for a typical university to complete the SPAP by hand is 3 days (Surrey, 2020) and comparing how many students could have been allocated successfully in this time showed how viable each solution was. The Hungarian and matched pairs algorithms both had student numbers above 10^5 and as the largest university, has 178735 students enrolled (Higher Education Statistics Agency, 2020) they would have no problem producing solutions in this time. The genetic algorithm could produce a largest allocation of 2972 in 3 days. The brute force method is not able to assign above 14 students and is therefore unsuitable in this situation and is very limited in its uses.

The difference between the big O notation produced in this paper compared to the literature can be seen in the Hungarian method, where there is a difference of 0.634 in the scaling power. This is a large difference and could suggest that the results used are erroneous.

10. Conclusion

To conclude, the most suitable algorithm to use for the SPAP is the matched pairs function in MATLAB (MathWorks, 2020) with a big O notation of O(n^2.491) producing best solutions. This algorithm produced a best-case allocation with 449267 projects and students in 3 days. The algorithm produced the best allocation score possible only and thus there was no way for the user to select an allocation that they wanted to use. This is the advantage of using a genetic algorithm, where a perfect solution is not reached but a 'good' score can be. What a 'good' score is depends on the problem and can be adjusted in algorithms like these to increase or decrease the run time making it very flexible. In this paper it was found to be suitable at 0.9. Conversely, the algorithm is stochastic and is therefore not guaranteed to produce a suitable solution.

The choices a student has can be different to the number of projects available to them and it was found that all students are likely to be allocated within their selection list if they are allowed to choose 10% of the total projects. Algorithms that find best allocations, brute force, Hungarian and matched pairs, have very little effect on student satisfaction when the number of choices is increase above 10% but the genetic algorithm discussed in this paper struggled to find acceptable solutions when students selected below 80% of the projects available.

The effect of biased data on the brute force algorithm showed that the scores deteriorated when a high percentage of students chose the same projects. Therefore, universities should attempt to provide a wide array of interesting projects.

A survey was produced for students at the University of Surrey Mechanical Engineering Department and it was found that the current allocations to students final year individual projects was 84.5% equating to 3.79 (a score that can be compared to the standard scoring method). The brute force, Hungarian and match pairs algorithms all produced a distribution with a mean score around 1.6 showing an improvement from the method done by hand.

Further research should investigate reducing the run time of each algorithm and re comparing to assure the functions are as efficient as possible. Also advanced fields like genetic algorithms have a huge amount of resources available to them and thus providing a better genetic algorithm could reduce the run time and increase the score of programs.

Finally, a program was developed with the match pairs function in MATLAB in combination with a google sheet to create an easy workflow for universities struggling with this problem and this can be found in appendix F.

References

A Ismaili, K. Y. M. Y., 2019. Student Project Resource Matching Allocation Problems: Two-Sided Matching Meets Resource Allocation. *AAMAS,* Volume Extended Abstract, pp. 2033-2035.

A. Olsson, G. S. O. D., 2002. On Latin hypercube sampling for structural reliability analysis. In: *Structural Safety.* Lund: Elsevier, pp. 47-68.

Alberto Moraglio, R. P., 2007. Inbreeding Properties of Geometric Crossover and Non-geometric Recombinations. *Foundations of Genetic Algorithms,* Issue 9, pp. 1-14.

Arif Anwar, A. B., 2003. Student project allocation using integer programming. *IEEE Transactions on Education,* 46(3), pp. 359 - 367.

Bae, S., 2019. *JavaScript Data Structures and Algorithms.* Hamilton: Apress.

BBC, n.d. *Charles Darwin Evolution and the Story of Our Species.* [Online]
Available at: https://www.bbc.co.uk/teach/charles-darwin-evolution-and-the-story-of-our-species/z7rvxyc
[Accessed 9 May 2020].

Cambridge Dictionary, 2020. *mutation.* [Online]
Available at: https://dictionary.cambridge.org/dictionary/english/mutation
[Accessed 20 March 2020].

Darwin, C., 1859. *On the Origin of Species by Means of Natural Selection, or the Preservation of Favoured Races in the Struggle for Life.* London: s.n.

Gordon, M., 2005. Big Data: Its's Not the Size That Matters. *Journal of National Security Law and Policy ,* Volume 7, pp. 311-323.

Gregory Hornby, A. G. D. S. L. J. D. L., 2006. *Automated Antenna Design with Evolutionary,* Ames Reasearch Center: NASA.

Higher Education Statistics Agency, 2020. *Where do HE students study.* [Online]
Available at: https://www.hesa.ac.uk/data-and-analysis/students/where-study
[Accessed 11 March 2020].

Holland, 1975. *Adaptation in Natural and Artificial Systems.* Michigan: University of Michigan Press.

J.A. Lima, N. G. H. P. A. R., 1996. Fitness Function Design for Genetic Algorithms. *Proceedings of IEEE International Conference on Evolutionary Computation,* pp. 207-212.

Jose Allen Lima, N. G. H. P. A. R., 1996. Fitness Function Design for Genetic Algorithms in Cost Evaluation Based Problems. *Proceedings of IEEE International Conference,* Volume 10, pp. 207-212.

Karloff, 1991. *Linear Programming.* s.l.:s.n.

Kuhn, H. W., 1955. The Hungarian Method For The Assignment Problem. *Naval Research Logistics Quarterly,* 2(2), pp. 83-97.

M Chiarandini, R. F. S. G., 2019. Handling preferences in student-project allocation. *Annals of Operations Research,* 275(1), pp. 39-78.

Mahmood, H., 2018. *Towards Data Science.* [Online]
Available at: https://towardsdatascience.com/softmax-function-simplified-714068bf8156
[Accessed 7 May 2020].

Mathematics Stack Exchange, 2012. *Finding the n-th lexicographic permutaton of a string.* [Online]
Available at: https://math.stackexchange.com/questions/60742/finding-the-n-th-lexicographic-permutation-of-a-string
[Accessed 1 May 2020].

Mathematics Stack Exchange, 2013. *"Normalise" values to sum to 1 but keeping their weights.* [Online]
Available at: https://math.stackexchange.com/questions/278418/normalize-values-to-sum-1-but-keeping-their-weights
[Accessed 5 May 2020].

MathWorks, 2020. *Equilibrate.* [Online]
Available at: https://uk.mathworks.com/help/matlab/ref/equilibrate.html
[Accessed 1 May 2020].

MathWorks, 2020. *Evaluating Goodness of Fit.* [Online]
Available at: https://uk.mathworks.com/help/curvefit/evaluating-goodness-of-fit.html
[Accessed 18 May 2020].

MathWorks, 2020. *Gamma Function.* [Online]
Available at: https://uk.mathworks.com/help/matlab/ref/gamma.html
[Accessed 3 May 2020].

MathWorks, 2020. *Match Pairs.* [Online]
Available at: https://uk.mathworks.com/help/matlab/ref/matchpairs.html
[Accessed 18 May 2020].

MathWorks, 2020. *Measure the performace of your code.* [Online]
Available at: https://uk.mathworks.com/help/matlab/matlab_prog/measure-performance-of-

your-program.html
[Accessed 28 April 2020].

MathWorks, 2020. *memory.* [Online]
Available at: https://uk.mathworks.com/help/matlab/ref/memory.html#brl1pef-1
[Accessed 18 May 2020].

Michael Mutingi, C. M., 2017. *Grouping Genetic Algorithms.* 666 ed. Cham: Springer.

Naima Hamid, R. A. M. I. O. M. F. D. A., 2018. Parameters identification of photovoltaic solar cells and module using the genetic algorithm with convex combination crossove. *International Journal of Ambient Energy,* pp. 517-524.

Paul R. Harper, V. d. S. I. T. V. A. K. S., 2005. A genetic algorithm for the project assignment problem. *Computers & Operations Research* , Volume 32, pp. 1255-1265.

Reeves, C. R., 2002. The Elements of a Genetic Algorithm. In: *Principles and Perspectives : A Guide to GA Theory.* Secaucus: Kluwer Academic Publishers, pp. 15-48.

Simon, D., 2013. *Evolutionary Optimization Algorithms.* Somerset: John Wiley & Sons.

StackOverflow, 2011. *How to scale down a range of numbers with a known min and max value..* [Online]
Available at: https://stackoverflow.com/questions/5294955/how-to-scale-down-a-range-of-numbers-with-a-known-min-and-max-value
[Accessed 18 May 2020].

Stanford University, n.d. *Error Sum of Squares (SSE).* [Online]
Available at: https://hlab.stanford.edu/brian/error_sum_of_squares.html
[Accessed 9 May 2020].

Surrey, U. o., 2020. *Mechanical Engineering Department.* Surrey: s.n.

Tang, G., 2017. *Hungarian.* s.l.:Stevens Institute Of Technology.

Appendix

Appendix A, The test data function written in MATLAB.

```
function TData =
TestDataMk3(NumStudents,NumProjects,NumChoices,Type,Bias)
%TestData Produces a set of random test data with students and their
%preferred projects.
%   Type = 0: The Projects are completely random permutations.
%             The bias must also be set to 0.
%   Type = 1: Half the projects are assigned within the bias region
%   Type = 2: The best-case scenario where everyone receives there
first
%             choice.
%   Type = 3: The worst-case scenario where everyone picks the same
choices
%   Example:
%   TestDataMk3(4,4,3,Type,Bias)
%         Students
%Projects 1 2 1 +
%         3 3 2 1
%         2 1 + 3
%         + + 3 2
%
%   TestDataMk3(3,4,3,Type,Bias)
%         Students
%Projects 1 2 1
%         3 3 2
%         2 1 +
%         + + 3
%The numbers in the matrix are the preference for that choice
%
%If the matrix has a limited number of choices
%LOOK AT HOW The matrix looks when choices are limited
%Initialise the size of the matrix to save time
%Error Check
%if (Bias < NumProjects) && (Bias ~= 0)                          %If
Bias is less than the number of projects and the bias is not zeros
%    error('Bias must be at least the number of choices');
%Present an error
%end
%Loop through all the columns and create a random permutation.
Select
%NumChoices from the set of 1 to NumStudents
if Type == 0
    for i = 1:NumStudents
        Perms = [1:NumChoices,ones(1,abs(NumProjects-
NumChoices))*(NumStudents+1000)];
```

```
        TData(:,i) = Perms(randperm(NumProjects))';
    end
elseif Type == 1
    for i = 1:NumStudents
        if randi(Bias) == 1
            TData(:,i) = 1:NumProjects;
        else
            TData(:,i) = (randperm(NumStudents,NumProjects));
        end
    end
elseif Type == 2
    TData(:,1) = [1:NumStudents]';
    for i = 2:NumStudents
        TData(1:(NumStudents+1-i),i) = [i:NumStudents]';
        TData(NumStudents+2-i:end,i) = [1:i-1]';
    end
elseif Type == 3
    for i = 1:NumStudents
        TData(:,i)= [1:NumStudents]';
    end
end
end
```

Appendix B, Test efficiency function and the scoring methods used.

```
function [Score] = TestEffMk3(Allocated,OriginalData,Type)
%TestEff calculates how happy students are with a certain system
%   The allocated projects are checked with the students wish list.2
%   metrics are then returned, the percentage of 'happy' students,
in which the student has received a project in their top x places.
%Initial Vars
Score = 0;
iTot =0;
%Standard Sum Mean
if Type == 0
    for i = 1:length(Allocated)
        %
        Score = Score + OriginalData(Allocated(i),i);
    end
    Score = Score / i;
%Weighted Mean
elseif Type == 1
    for i = 1:length(Allocated)
        Score = Score + OriginalData(Allocated(i),i)*i;
    end
    Score = Score / i;
%Squared Mean
elseif Type == 2
```

```
    for i = 1:length(Allocated)
        Score = Score + OriginalData(Allocated(i),i)*i^2;
        iTot = iTot +i^2;
    end
    Score = Score / iTot;
%Genetic Scoring
elseif Type == 3
    for i = 1:length(Allocated)
        Score = Score + OriginalData(Allocated(i),i);
    end
    %Y = (X - min(X)) / (max(X) - min(X)) changes from 0 to 1
    Score = Score - length(Allocated);
    Score = Score./(((1000+length(Allocated)).*length(Allocated)) -
length(Allocated));
    Score = 1 - Score;
%1 is good zero is bad
end
end
```

Appendix C, function to create a lexicographical permutation.

```
function [OutArray] = nthPerm(Array,Index)
%nthPerm Calculates the nth permutation in lexographical order
%   Uses factoral remainders to find the next possible number in
each index
%   that could give the desierded index.
%   Code based on the following artical:
%   https://math.stackexchange.com/questions/60742/finding-the-n-th-
lexicographic-permutation-of-a-string
%   Array is the Numbers to be permuted
%   Index is the Lexographical index of the permutation we are
trying to
%   achieve

%-------Variable Initiation-------%
Len = length(Array);                            %The number of
numbers that needs to be manipulated.
Inversei = 0;                                   %Start counting up
from 0.

%-------Main Function-------%
for i = linspace(Len-1,1,Len-1)                 %Work backwards
across the permutation with i from the length minus 1
    Inversei = Inversei + 1;                    %Track the index
going up from one
    x = 0;                                      %x is reset to 0.
```

```
    while (Index < 0) || (Index > factorial(i)) %While the Index is
less than 0 or greater than the factorial of the position in the
number
        x = x + 1;                              %Increase the digit
by 1
        Index = Index - factorial(i);           %Minus the factorial
of the digit position from the Index
    end
    OutArray(Inversei) = Array(x+1);            %The selected Digit
is added to the final output array
    Array(x+1) = [];                            %The same element is
removed from array
end
OutArray(Len) = Array;                          %The last element
left in array is added to the end of the output array
End
```

Appendix D, The Hungarian Method

```
function [Allocated] = HungarianMk2(TData)
% A function that returns the allocated projects using the Hungarian
method
% The input is the cost matrix of the projects and the students
%%
%----------Row and Column Reductions Function----------%
function [TData] = RCReductions(TData)
%TData is the cost matrix of projects and students
%Basically, manipulates the cost matrix, reducing it
    %-----Row Reductions-----%
    for i = 1:length(TData)                                 %Loop
through the rows
        TData(i,:) = TData(i,:) - min(TData(i,:));          %Each
row - the lowest value in that row
    end
    %-----Column Reductions-----%
    for i = 1:length(TData)                                 %loop
through the columns
        TData(:,i) = TData(:,i) - min(TData(:,i));          %Each
column - the lowest value in each column
    end
end
%%
%----------Row and Column Scanning Function----------%
function [Allocated,TData,Completed] = RCScanning(TData)
    %Inputs :  the cost matrix
    %Outputs: the Allocated array of projects
    %         the TData cost modified matrix
    %         weather the final answer has been reached
```

```
    %-----Initialise variables-----%
    LinesNum = 0;                                   %The number of lines
that have been drawn onto the matrix
    Lines = zeros(length(TData));                   %An array showing
where the lines cover once drawn
    Allocated = zeros(1,length(TData));             %Initialised array
for the allocated projects with zeros
    %-----
    while nnz(~TData(find(Lines == 0))) ~= 0            %while the
number of zero values in TData that correspond to where the lines
matrix has a zero value does not equal 0
        ZeroSumR = sum((TData == 0 & Lines == 0),2);    %an array of
the sum of values where TData and Lines equal 0 in the rows.
        ZeroSumR(ZeroSumR == 0) = NaN;                  %Where
ZeroSumR equals 0 set it to NaN
        ZeroSumC = sum((TData == 0 & Lines == 0),1);    %an array of
the sum of values where TData and Lines equal 0 in the columns.
        ZeroSumC(ZeroSumC == 0) = NaN;                  %Where
ZeroSumC equals 0 set it to NaN
        if min(ZeroSumR) > 1 && min(ZeroSumC) > 1       %if the
smallest value in ZeroSumR and ZeroSumC are less than 1
            [row,col] = find(Lines == 0 & TData == 0);      %Gives
the index of where Lines and TData equals zero to the row and column
arrays
            Lines(row(1),:) = NaN;                          %Changes
the column corresponding to selected row index to NaN
            LinesNum = LinesNum + 1;
%Increases lineNum by 1
            Lines(:,col(1)) = Lines(:,col(1)) + 1;
%Increases the column corresponding to the row index by one
            Allocated(col(1)) = row(1);                     %The
allocated project is the first column value and the row is the
student
        else
            for i = 1:length(TData)
%Scan rows for zeros
                ModData = TData(i,find(Lines(i,:)==0));
%The parts of the row that is not covered by a line (SINGLE ROW)
                if nnz(~ModData) == 1
%If the number of zeros not covered by a line = 1
                    ZeroIndex = find(TData(i,:) == 0 & Lines(i,:) ==
0);    %The index of the zeros in Lines and TData
                    Lines(:,ZeroIndex) = Lines(:,ZeroIndex) +1;
%Add 1 to the column where the 0 was found in the row
                    LinesNum = LinesNum + 1;
%Increase the number of lines used
                    Allocated(ZeroIndex)= i;
%Add project selection to the allocation array (i is the student)
```

```
                end
            end
            for i = 1:length(TData)
%Scan Columns for zeros
                ModData = TData(find(Lines(:,i)==0),i);
%The parts of the column that is not covered by a line (SINGLE
COLUMN)
                if nnz(~ModData) == 1
%If the number of zeros not covered by a line = 1
                    ZeroIndex = find(TData(:,i) == 0 & Lines(:,i) ==
0);    %The index of the zero
                    Lines(ZeroIndex,:) = Lines(ZeroIndex,:) +1;
%Add 1 to the row where the zero was found in the column
                    LinesNum = LinesNum + 1;
%Increase the number of lines used
                    Allocated(i)= ZeroIndex;
%Add selection to the allocation array (i is the project index)
                end
            end
        end
    end
    if length(TData) == LinesNum                %If the size of the
array is equal to the number of lines drawn the solution is found
        Completed = 1;                          %The solution is
found
    else
        Completed = 0;                          %The solution is not
found
        ModData = TData(find(Lines == 0));      %Find the values in
TData that correspond to where Lines equal 0
        ModIndex = find(Lines == 0);            %Find the indexes
where Lines equal 0
        for i = ModIndex                        %Loop through
ModIndex as i becomes each value
            TData(i) = TData(i) - min(ModData); %Minus the minimum
value in ModData from the indexes shown above
        end
        ModIndex = find(Lines == 2);            %All cells with a
value of 2 in lines add min value to them in
        for i = ModIndex
            TData(i) = TData(i) + min(ModData); %Add the minimum
value in ModData to the other ModData values
        end                                     %leave all cells
with Line equal to 1
    end
end
%%
%----------------MAIN PROGRAM----------------%
```

```
if size(TData,2) > size(TData,1)
    error('Not possible to find a complete solution with more
students than projects');
else
    for j = 1: 100000
%Repeat an arbitrary large amount of time
        TData = RCReductions(TData);
%Run the function RCReductions on TData
        [Allocated,TData,Completed] = RCScanning(TData);
%Run the function RCScanning on TData
        if Completed == 1
%if the function returns completed as 1 break
            FoundSolution = 1;
            break
        end
        if j == 100000
%if the repetitions are up the solution has not been found
            FoundSolution = 0;
        end
    end
end
end
```

Appendix E, The Genetic Algorithm

```
function [NewBestScore,t] =
GeneticAllocation(NumberOfStudents,NumberOfProjects,CostMat,PopSize)

MutationChance = 1/(PopSize*sqrt(NumberOfStudents));   %Mutation
chance is said to be good at 1/(PopSize*sqrt(AllocationLength))
%MutationChance = MutationRate;
%Create random population
%   Try making the allocations permutations, would need to be
incorporated
%   in rest of code too
Pop = randi(NumberOfProjects,PopSize,NumberOfStudents);

%Iterations
NumLevel = 0;
t = cputime();
NewBestScore =0;
while (NumLevel < 10)
    OldBestScore = NewBestScore;
    [Score,EzScore] = fitness(Pop,CostMat);         %Calculate the
score of each creature
    NewBestScore = max(EzScore);                        %Find the max
score
    if NewBestScore == OldBestScore
```

```
        NumLevel = NumLevel +1;
    else
        NumLevel = 0;
    end
    Pop =
CrossoverAndMutation(Pop,Score,NumberOfProjects,MutationChance);
    %disp(NewBestScore);
end
t = cputime() - t;
end

%----Functions----%
function [Score,EzScore] = fitness(Pop,CostMat)
%Calculate the scores of each creature
Score = zeros(1,size(Pop,1));                          %Pre-allocate
var size
for i = 1:size(Pop,1)                                  %Begin Loop
through the population
    Score(i) = TestEffMk3(Pop(i,:),CostMat,3);         %Score using
function
end
EzScore = Score;
Score = Score./sum(Score);                             %This makes it
sum to 1
end

function [NewPop] =
CrossoverAndMutation(Pop,Score,NumberOfProj,MutationChance)
%Select 2 creatures and combine the genes
%New crossover function to stop multiple projects being allocated to
a
%student

for i = 1:size(Pop,1)
    Index1 = randsrc(1,1,[1:length(Score);Score]);
    Index2 = randsrc(1,1,[1:length(Score);Score]);
    U = Pop(Index1,:);
    V = Pop(Index2,:);
    UScore = Score(Index1);
    VScore = Score(Index2);
    C = zeros(1,size(Pop,2));
    Available = 1:NumberOfProj;


    %Main Algo
    for j = 1:size(Pop,2)
        RandChance = rand();
```

```
        if (~ismember(U(j),C)) && (~ismember(V(j),C)) && (RandChance
> MutationChance)
            if U(j) == V(j)
                C(j) = U(j);
                Available (Available == C(j)) = [];
            else
                if UScore >= VScore
                    C(j) = U(j);
                    Available (Available == C(j)) = [];
                else
                    C(j) = V(j);
                    Available (Available == C(j)) = [];
                end
            end
        elseif (~ismember(U(j),C)) && (ismember(V(j),C))
            C(j) = U(j);
            Available (Available == C(j)) = [];
        elseif (ismember(U(j),C)) && (~ismember(V(j),C))
            C(j) = V(j);
            Available (Available == C(j)) = [];
        else
            %Select value not in C
            C(j) = Available (randi(size(Available)));
%Select a number from Available
            Available (Available == C(j)) = [];
%Remove this number from available
        end
    end
    NewPop(i,:) = C;

end
end
```

Appendix F, The suggested method for completing the SPAP

```
% Student Project Allocator.
% """""""""""""""""""""""""""""
% Finds the optimum solution from a csv file allocating students to
% projects. Please use the google doc template given by:
% https://forms.gle/G215nbwpbBCCsrKD7
%
% Steps to use:
% 1: Create a google form and have the students enter the project
number of
%    their top n choices. Ensure all responses are set to mandatory.
% 2: Download the responses as a csv file
% 3: Adjust variables
% 4: Run this code
```

```
% 5: Convert the .csv file to an excel sheet and upload to surrey
learn
%
% By Adam Dowse
% University Of Surrey

% --------------------------------
% CHANGE THESE VARIABLES:
FileLoc = "C:\Users\adamd\OneDrive\Documents\Uni\Individual Project
- Student Project Assignment
Problem\Code\Project_Allocations_Test1.csv";
NumOfProj = 20;
% --------------------------------

% Variables
OData = readmatrix(FileLoc);
OData = OData';
StudentIndex = OData(2,:);
OData(1:2,:) = [];
NumOfStud = size(OData,2);
NumOfChoices = size(OData,1);
Data = zeros(NumOfProj,NumOfStud);

% Convert into cost matrix
for i = 1:NumOfStud
    for j = 1:NumOfChoices
        if OData(j,i) > 0
            Data(OData(j,i),i) = j;
        end
    end
end

% Replace zeros with arbitrary large value
[x,y] = find(Data == 0);
for i = 1:length(x)
    Data(x(i),y(i)) = NumOfChoices + 10000;
end

%Produce allocation
Allocation = matchpairs(Data,1000);
Allocation(:,2) = StudentIndex';

%Output Data
writematrix(Allocation);
```