Final Report

Maximising entertainment value in the vote-reveal problem

Final Year Project (CM3203) - 40 Credits

Author: Iain Johnston (1312579)

Supervisor: Richard Booth
Moderator: Xianfang Sun

Abstract

Notes - remove at end

- clean up pseudocode (remove some lines that are too specific)

- csv reader

Acknowledgements

Contents

List of Figures

List of Tables

0.1 Introduction

In many elections or competitions, a set of voters will rank a set of candidates from best to worst, or will give scores to some of the candidates, with the winner then being the candidate that gets the highest total number of points. When it comes to revealing the result after all votes have been cast, some competitions proceed by having a roll-call of all the voters in which each announces their own scores. This is often done for entertainment purposes such as in the Eurovision Song Contest.[1]

The concept of *entertainment*, especially with respect to competition, is a heavily subjective matter and as such is difficult to quantify in simple terms. There are intuitive constituent parts to an *entertaining* competition (like Eurovision) such as, if the winner is known early or late, and how many teams are in the running to win. This project will find ways to translate these slightly nebulous concepts into more concrete mathematical functions.

The two main questions that this project will aim to answer are:

1. How can we define the concept of ”entertainment” in the context of an optimisation problem, and hence try to maximise it.
2. In which order should the votes be revealed in order to maximise that entertainment value?

To try and answer these questions the Eurovision Song Contest will be used as an example as it has good quality datasets. Furthermore voting rules have gone through changes over the years as more countries joined and as it grew in popularity, giving the project a natural comparison tools throughout. Most recently since the 2016 running of the contest. The scoring rules of Eurovision are also well documented and relatively simple (see 0.2 for deeper explanation).

These changes to the voting rules show that the motivation behind this project, to maximise entertainment when revealing the votes, is a current problem and there have been attempts to try and solve this problem already.

There is no specific intended audience of this project, Eurovision is a test competition as it fits well with the more general optimisation problem that this project is trying to solve. In this regard it could be said that the beneficiaries of the project would be those who would also attempt to solve this problem, for other datasets or competitions. This project can help set out a framework for approaching these problems and many parts could even be re-used as long as the problem can be formulated in a specific way (see 0.4 for details on how)

The scope of this project is to try and solve the problem by finding a globally optimum solution, or by find the best solution possible and hence have a voting order that is the most entertaining. Also within scope is producing a way of visualising the solutions found and from this gaining better insight into why they are entertaining. This project does not try and influence other parts of the voting that could also lead to more entertainment (such as who delivers the votes or other human traits, only the order itself)

The approach to tackle this problem is a scientifically minded one. There are three main sections to the project. Firstly, designing and implementing functions that quantify the entertainment of a given voting order. Secondly, solving the optimisation problem of maximising the entertainment value given by those functions; using optimisation algorithms. Finally, analysing the solutions given and comparing and contrasting why there are entertaining and how well the algorithms did in finding them. This approach involves iteration on theoretical ideas whilst feeding back in results as the project goes along.

The important outcomes of this project are, a solution that delivers an entertaining competition when the votes are revealed, a way of describing entertainment mathematically, and a way of visualising the competition as the votes are revealed.

0.2 Background

The more general reason for this project is to try and see if there can easily be found an optimal solution to a problem when given a mathematical representation of something that humans experience.

Moreover in competitions where voting is revealed, such as in Eurovision, one of the only ways to change how entertaining the competition seems, is to change the order. Any other changes to the actual running of the competition are unethical.

Furthermore as competitions are usually televised or watched live, it is generally in the interest of those in charge of the competition to produce an entertaining show as this helps them with many facets of their business such as through advertising, but also in building a following for the competition. In this way, it is obvious that a more exciting competition leads to more engaged fans.

0.2.1 Project Context

Eurovision has been a topic of research for many years now. The main focus of that research has been in relation to the voting patterns that can be found over the course of many runnings of the competition. This research such as *"Comparison of Eurovision Song Contest Simulation with Actual Results Reveals Shifting Patterns of Collusive Voting Alliances"*[2], *"Geography, culture, and religion: Explaining the bias in Eurovision song contest voting"*[3] and *"The Eurovision Song Contest. Is voting political or cultural?"* [4] all use the Eurovision Song Contest as a basis to investigate political and cultural phenomena.

There have also been some more computational view of Eurovision such as *"Using the Clustering Coefficient to Guide a Genetic-Based Communities Finding Algorithm"*[5] which attempts to find communities within the voting patterns.

The particular problem this project is addressing has not been approached in a scientific setting as yet. It does seem to have been at least tackled by those that run Eurovision however not with exactly the same data and considerations. Moreover as they are a private company they have never published any methodology on how they pick voting orders. They usually state on their website when revealing the voting order that

```
"The voting order has been determined by the results from last night's Jury Final.
An algorithm has been created to try and make the voting as exciting as possible."
```

[6] There is no mention of how this algorithm works or what they constitute entertaining or exciting.

The biggest drawback that can be said about the current Eurovision algorithm, without seeing it and understanding its methodology, is that it only takes into account Jury votes as they can be decided on during the dress rehearsals. This is problematic as the Jury and tele-voting can diverge for many reasons and hence the voting order that purports to be entertaining, may only be such when the viewers at home agree exactly with the Jury.

This project's method takes into account both the sets of votes and hence could give a more correct picture of what will happen, entertainment wise, for any given voting order. One main deficiency with this method is that fact, as it is necessary to wait for all the votes to be cast and then the voting order can be found. In real world terms this may not be feasible for the Eurovision Song Contest as they may have commitments that require that the voting order is known in advance for various logistical reasons.

Another drawback as mentioned above, is the fact that there is no explanation as to why their algorithm believes a given order to be more exciting than any other. This project will attempt to standardise mathematically a function for entertainment, which could in future be compared through human tests (see 0.6).By describing a concept mathematically not only can it be used by algorithms but it can also form part of a proof of which a general thought put forward by a human cannot be.

This project only looks at one specific area of the competition to describe and codify entertainment. This can be seen as a drawback of the solutions found, however the way this project has been undertaken leaves the opportunity of further work in the area of defining entertainment functions as described in 0.6.

The techniques used and implemented in this project have applications outside the strict problem that is being solved here. For example even though this problem is specifically trying to produce entertaining orders for Eurovision, there is no reason why other competitions who use the same type of ordered voting system (sometimes referred to as roll-call voting) could use the algorithms and framework designed and implemented in this project to solve their problems.

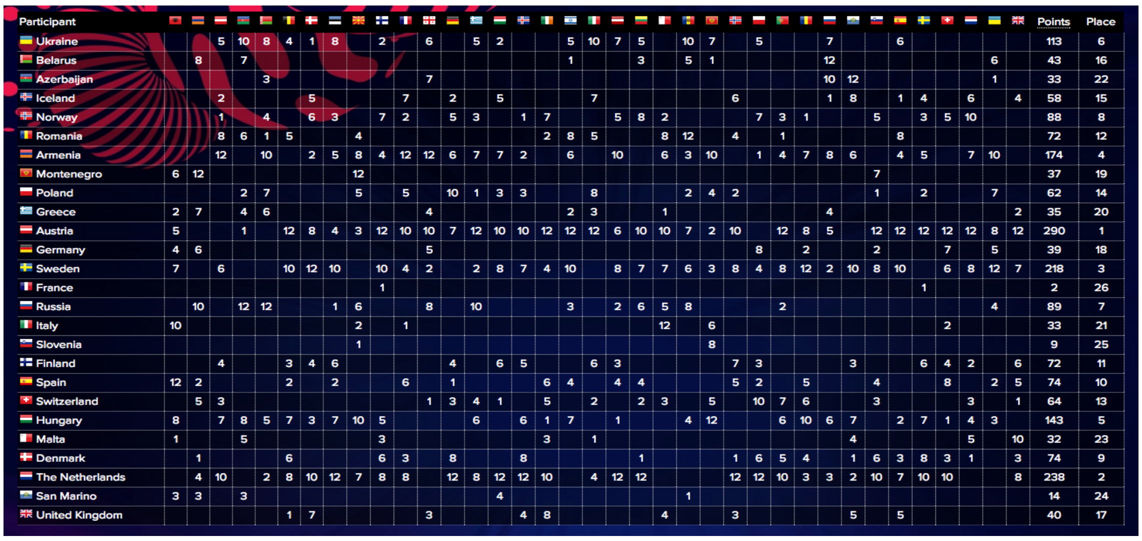
0.2.2 Background Theory

Understanding how the Eurovision Song Contest voting works is key to this project, as this affects many parts of the design and implementation of entertainment functions. Furthermore the system for voting has been changed over many years and so to try and standardise the project I will be using the system that was in use during the 2014 running of the competition, and the main dataset also comes from that year.[1]

In this system, each country awards two sets of 12, 10, 8–1 points to their 10 favourite songs: one from their professional jury and the other from tele-voting. Moreover as voting is done by both and jury and by people voting by phone, both constituent parts count 50% towards the final score.

In the 2014 contest, there were 37 countries that participated and of those 26 reached the final. All 37 countries vote in the final. From this point on in this report, the countries that vote are referred to as *voters* and those 26 competing in the final are referred to as *participants.* There is overlap between the two groups i.e the *participants* set is a subset of the *voters* set; but it should be obvious from the context to whom is being referred.

[1]2014 voting system is the system that had been in use from 2009 and changed in 2016



Participant																																						Points	Place
Ukraine			5	10	8	4	1	8		2		6		5	2			5	10	7	5		10	7		5			7			6						113	6
Belarus		8		7														1			3		5	1					12							6		43	16
Azerbaijan					3							7																	10	12						1		33	22
Iceland			2				5				7		2		5				7						6				1	8		1	4		6		4	58	15
Norway			1		4		6	3		7	2		5	3		1	7			5	8	2				7	3	1			5		3	5	10			88	8
Romania			8	6	1	5			4								2	8	5			8	12		4		1					8						72	12
Armenia			12		10		2	5	8	4	12	12	6	7	7	2		6		10		6	3	10		1	4	7	8	6		4	5		7	10		174	4
Montenegro	6	12							12																						7							37	19
Poland				2	7				5		5		10	1	3	3			8				2	4	2						1		2			7		62	14
Greece	2	7		4	6							4						2	3			1							4								2	35	20
Austria	5			1		12	8	4	3	12	10	10	7	12	10	10	12	12	12	6	10	10	7	2	10		12	8	5		12	12	12	12	12	8	12	290	1
Germany	4	6										5														8		2			2			7		5		39	18
Sweden	7		6			10	12	10		10	4	2		2	8	7	4	10		8	7	7	6	3	8	4	8	12	2	10	8	10		6	8	12	7	218	3
France										1																							1					2	26
Russia		10		12	12			1	6			8		10				3		2	6	5	8				2									4		89	7
Italy	10								2		1											12		6										2				33	21
Slovenia									1															8														9	25
Finland			4			3	4	6					4		6	5			6	3					7	3				3			6	4	2		6	72	11
Spain	12	2				2		2			6		1				6	4		4	4				5	2		5			4			8		2	5	74	10
Switzerland		5	3									1	3	4	1		5		2		2	3		5		10	7	6			3				3		1	64	13
Hungary	8		7	8	5	7	3	7	10	5				6		6	1	7		1			4	12			6	10	6	7		2	7	1	4	3		143	5
Malta	1			5						3							3		1											4					5		10	32	23
Denmark		1				6				6	3		8			8					1				1	6	5	4		1	6	3	8	3	1		3	74	9
The Netherlands		4	10		2	8	10	12	7	8	8		12	8	12	12	10		4	12	12				12	12	10	3	3	2	10	7	10	10			8	238	2
San Marino	3	3		3											4								1															14	24
United Kingdom						1	7					3				4	8					4			3					5		5						40	17

Figure 1: 2014 Scoreboard

This voting system creates a matrix of scores as seen in Figure 1. The *voters* are arranged along the top alphabetically, while the *participants* are along the left hand side. Points refers to the final total of all the scores given at the end of all voting, and rank is their final position in the competition.

This is important to look at as it can help clarify what this project's aim is. A solution to the problem is found by moving the *voters* along the top into different positions and then analysing the scores after each country has voted, to reach an *entertaining* order for those countries to reveal their votes.

Hence the problem that this project is trying to solve is about ordering who votes when so as to maximise some mathematical concept of entertainment. Should the order put Spain and France next to each other in the order, or would it be better if they were split up. Should Ukraine's vote be near the start or near the end so that the competition is exciting. This is what the project is trying to decide.

0.3 Algorithm Designs/Approach

This section should give an understanding of the main computational parts that are needed to solve this problem, before implementation can be explained. This will include some pseudocode of the algorithms used, explanations of the neighbourhoods tried and the entertainment functions designed. It is intended to give an introduction to the why and how these parts of the project exists, before Section 0.4 goes into implementation details with code and explanations using Python.

Firstly to help clarify the problem it is easier to view a much smaller competition and then run through the main points of theory, that then are used in the full Eurovision system.

Firstly we design a competition that involves 3 teams total, of which 2 are *participants*. The scoring system is even more simple than the one used by Eurovision. Here each voting team gives 1 point the team they prefer and 0 to the other. A team cannot give itself 1 point. This produces a lot of scores similar to the one in Figure 1. It is shown in Table 1.

Table 1: Scores for simple example game

	Austria	Spain	France
Austria	0	1	1
France	1	0	0

From Table 1 we can see how the scores will be calculated in this type of competition. For simplicity, the order that the teams vote is the order from left to right in Table 1. This means that after Austria has voted the scores are $Austria : 0, France : 1$, the rest of the competition is shown in Table 2

Table 2: Scores per round

	After Austria's vote	After Spain	After France
Austria	0	1	2
France	1	1	1

We can see that France wins this competition, with 2 points. This cumulative scoring extends to the full Eurovision competition in the exact same way, except for in that case there are 30+ voters.

All that is needed to turn this simple example into Eurovision is to add all the *voters* and *participants* and change the scores given out to be $12, 10, 8 - 1$ instead of just 1 and 0.

0.3.1 Entertainment Functions

From the small example, it is quite easy to think of some theoretical reasons why a given order is entertaining. This leads straight into the design of entertainment functions and how they model entertainment.

It is important to start with the design of the entertainment functions as they are the main workhorse part of finding solutions to the problem. The approach was to analyse the competition and try and identify things in a competition that lead to excitement. From those ideas, it was a case of trying to formalise that theory into a concrete piece of maths that could then be programmed. Furthermore the entertainment functions are quite problem specific and may not transfer well into other problems, whereas the algorithms are entirely agnostic to the problem at hand. This means the entertainment functions follow much more closely from the problem itself than any other part of the project.

Any entertainment functions talked about in this section share some characteristics that need to be explained. The first is that they take a **solution (Φ)** to the problem (see Glossary-3) as input and they return a single **entertainment value** (ε) (see Glossary-4). Moreover they calculate the **scores**(S) (see Glossary-6) for each *participant* country every **round** (R)(see Glossary-5).

MaxMin

The first entertainment function that was designed is named **MaxMin** and as it's name suggests it works by finding the difference between the highest score (max) and the lowest score (min) after each voting round. It then sums the value which is the ε value for that solution. This function follows quite naturally from watching and analysing the Eurovision competition as the way roll-call style voting works, everything is building towards the later end of the voting order. Hence it seems to be a innate part of the competition that you would want every country to be in with a chance of winning for as many rounds as possible. Moreover as there is only one prize, for first place, it is even less important to worry about positioning other than the top.

So each round the distance (Λ) is calculated as such:

$$\Lambda = \max(S) - \min(S) \tag{1}$$

Then keeping the distance per round i as Λ_i, the entertainment value (ε) is found by:

$$\varepsilon = \sum_{i=0}^{n_R} \Lambda_i \tag{2}$$

This is a relatively simple equation and hence transfers simply to code. However more importantly, it encodes an intuitive part of entertainment mathematically. These functions encode the fact that an entertaining competition is one in which the distance between first and last place is as low as possible as often as possible. In this case instead of maximising ε, we want to minimise it.

Over the course of a whole competition i.e: n_R voting rounds, it is intuitive to want that distance to stay as low as possible. Hence finding the sum of the distance (Λ) in each round and then using optimisation algorithms to try and find a solution (Φ) that minimises this value.

However there is one major drawback of the basic MaxMin method, which is that in some rounds, especially later in the competition, the last place country is actually mathematically eliminated from the race for first place.

RefinedMaxMin

This leads to a second version of the entertainment function for this problem. As it is essentially a refinement of the first function and not a brand new method it is called **RefinedMaxMin**.

Equation 2 is the same across both functions, however where **RefinedMaxMin** and **MaxMin** differ is how the min part of equation 1 is calculated. As some of the countries cannot win the competition after some number of rounds, they should not be taken into account when seeing if a solution is entertaining or not.

To find whether a team can still win, the upper bound of points left available is found. This means that we assume for each team, and for the remaining rounds left to be revealed, that they gain the maximum (12) points and the hence we find the highest score they could ever attain. This method does not take into account whether the country has already given itself a vote (in which case they would not be allowed to get 12 points in that round). This was done to simplify the method and make testing it's improvements against MaxMin easier. Furthermore finding the upper bound is generally safe, especially when removing the low scoring teams in

Eurovision as voting is generally quite uniform past a certain point in the voting; i.e countries that are given high points already generally get more, and countries that have received few points continue to get few.

The method for finding if a team can still win is to first sort the scores for every country in round i in ascending order. Then iterating through that list of scores from the start and for each score checking whether equation 3 is true.

$$countriesScore + (maxScorePerRound * roundsRemaining) < currentTopScore \tag{3}$$

If equation 3 hold true then that country cannot win even if it received the maximum number of points (12 in Eurovision's case) and the leaders received the worst score possible (0), for the remaining voting rounds. As the scores are in ascending order the last minScore found for which Equation 3 held true is the minimum score to be returned.

It is quite simple to justify this refinement when looking at the competition. Those teams who cannot win should will not be making a mark on the entertainment of the voting order as most viewers will not be paying attention to their scores. Moreover this refinement only works in competitions where there is only interest at the top of the table as opposed to competitions with relegations or play-offs, where more than just who is the overall winner is important.

An important part of the design of these two methods is taking into account the Big-O complexity. The complexities are shown in Table 3. How these values where reached is explained more in Section 0.4.2 as they are effected more by the implementation details discussed therein. The main point to recognise is that the methods are different in terms of complexity which can help in evaluation such as in Section 0.5.

Table 3: Big-O of ε methods

	Big-O time complexity
MaxMin*	O(n)
refinedMaxMin	O(n log n)

* Complexity of Python's *min*() function from[26]

0.3.2 Neighbourhoods

An important part of solving optimisation problems is designing and implementing neighbourhoods for given solutions. As a solution (Φ) is a list of *voters* in a certain order, then we define a neighbourhood a solution as any other solution that has two members swapped. This means that there are only two changes between a solution and any of it's neighbours.

In this regard an order is a *permutation* of the *voters*. Using Cauchy's two line notation for permutations[7], we can show the basic theory for neighbourhoods.

$$\begin{pmatrix} x_1 & x_2 & x_3 & \cdots & x_n \\ \sigma(x_1) & \sigma(x_2) & \sigma(x_3) & \cdots & \sigma(x_n) \end{pmatrix} \tag{4}$$

Equation 4 shows the first solution on the top row and a neighbour of that solution on the bottom row. The functions that map the elements of a solution to a neighbour (σ) are explained below.

These methods are examples of cyclic permutations i.e a permutation of a set which maps a subset of elements to each other in a cyclic way, while mapping to themselves all other elements of the set. The cyclic parts of a permutation are called *cycles*. A cycle with only two elements is called a *transposition*[8]. As both the methods following only contains two indexes, they are transpositions.

Random neighbour

The first idea and most simple for this problem, was to swap two random elements of the ordering. This means that the function σ swaps the two elements at those indexes. This method can be described as such in equation 5.

$$\begin{aligned} \sigma_r = x_i &\mapsto x_j, \\ x_j &\mapsto x_i, \\ \forall x \neq i \vee \forall x \neq j : x &\mapsto x \end{aligned} \tag{5}$$

for two random indexes i and j

This means that we swap the elements at indexes i and j and swap the rest with themselves. The code for this will be explored in Section 0.4. This method give a large possible neighbourhood as theoretically for any single solution there are {length of solution $\times$ length of solution - 1} possible neighbours. In the 2014 Eurovision competition this would mean $\{37 \times 36 = 1332\}$ neighbours.

This method may be simple however, when a closer look is taken at the ε values of solutions that are very similar to each other i.e they have 3 or 4 elements swapped, it is clear that their ε values only differ by a little. This calls into question if the method detailed above is actually going to help the optimisation algorithm reach an optimal solution.

Adjacent neighbour

Those questions about the efficacy of the random method leads to another method for finding a neighbour of a solution. As solutions that are close together are usually quite similar, it is intuitive that swapping adjacent elements may improve the performance.

This method works by finding a random value between 0 and length of solution - 1. Then swapping the element at that index with its immediate neighbour to the right i.e. index + 1. It can be expressed in the same way as in equation 5 in equation 6, except that in this case index i is a random number and index $j = i + 1$.

$$\begin{aligned} \sigma_a = x_i &\mapsto x_j, \\ x_j &\mapsto x_i, \\ \forall x \neq i \vee \forall x \neq j : x &\mapsto x \end{aligned} \tag{6}$$

Another difference between this method and the random method is how it can actually be calculated. After finding a neighbour of a given solution, the ε value of that solution must be found. By swapping 2 adjacent countries in the solution a *partial re-calculation* of the entertainment value can be done using the entertainment value of the previous solution.

As the previous solution and the new solution only differ by one position it is possible to only partially re-calculate the ε value. Re-calculation involves removing the distances for the two rounds of the swapped countries in the old solution and adding them back in their new positions in the new solution. The equation for this is shown in 7

$$\varepsilon_{new} = \varepsilon_{old} - \Lambda_{ij_{old}} + \Lambda_{ij_{new}} \tag{7}$$

The ε_{new} and $\Lambda_{ij_{old}}$ values can be found by keeping them from the old solution and passing them into the ε calculation function. The $\Lambda_{ij_{new}}$ must still be re-calculated however only up to the index of the higher of the two indexes.

The efficacy of this neighbourhood can be see when calculating the ε values. A full calculation, such as is used in the random swap method involves {number of voters $\times$ number of participants} or $V \times P$.

However due the adjacent neighbour swaps Table 4 shows what this method needs to calculate.

Table 4: Score calculations needed with adjacent neighbour

	Number of calculations
Max	$V \times P$
Min	$2 \times P$
Average	$\approx 18 \times P$

Table 5 show that what this would mean in the 2014 competition.

Table 5: 2014 Eurovision specific calculations

	Number of calculations
Max	962
Min	52
Average	468

A deeper comparison and evaluation between the two methods can be found in Section 0.5

0.3.3 Optimisation Algorithms

In the previous section the necessary steps to solve the problem were outlined. Namely calculate an entertainment value (ε) which can be optimised. To do this it is necessary to use a set of optimisation algorithms which maximise or minimise a value in order to settle on an optimal solution.

Greedy Search

The first algorithm that was posited was a simple greedy search algorithm. Greedy search works on the idea that by selecting the optimal choice at every possible stage, it will lead to a globally optimal solution.

At first glance it is not obvious to see if the problem we are attempting is simple or complex. This lead to greedy search as a good first try as it is sufficiently simple to allow for easy writing of the code and will likely find the optimal solution if the solution space is relatively simple. If the solution space *is* in fact too complex then greedy may instead find a good approximation for the problem.

The pseudocode for greedy is shown in Algorithm 1 in the appendix.

As can be seen, it is a very simple algorithm that takes possible solutions and compares them to the already found best solution. It's simplicity is, unfortunately, also it's downfall in this problem. Although it can find a good approximate solution it is not able to find optimal solutions that other algorithms do find.

Looking at the pseudocode, and with some simplifying assumptions the time complexity for this greedy search implementation can be found.

This is done by giving all the instructions that make a difference to complexity, a value T_i from their line number. This means line 5 is T_5, then working out how many times that instruction will need to be run given an input. The input is the *voters* and *participants*, V and P. Some lines can be ignored as they do very little to complexity. Table 6 shows the complexity added by some important lines.

Table 6: Greedy complexity per line

Line	Effect on complexity
T_1	run once with V steps
T_2	run once
T_3	run once with $V \times P$ steps
T_5	m times with constant steps
T_6	m times with $V \times P$ steps; as worst case, assume that full re-calculation is needed
$T\_7 - T\_11$	m times as is worst case, so assume this block always run
T_{12}	m
T_{14}	1

N.B The main loop (lines 4 - 13) is constant, given it comes from *maxIterations*. It adds m times to the instructions inside it.

Factoring this down into equation 8 gives:

$$\begin{aligned} f(n) = \ & (V \times T_1) + T_2 + (V \cdot P \times T_3) + (m \times T_5) + (m \times V \cdot P \times T_6) \\ & + (m \times T_{7-11}) + (m \times T_{12}) + T_{14} \end{aligned} \tag{8}$$

From this we can safely ignore anything that is less than the highest order as it will dominate the complexity growth. Moreover we can see that the highest order is T_6, so we are left with equation 9

$$f(n) = \ (m \times V \cdot P \times T_6) \approx m \times V \cdot P \tag{9}$$

where $P <= V \ll m$. This can be further simplified in terms of *voters* (V), as P is always less than or equal to V and m is always larger or equal to V, they can be equated to V. The coefficients of m and P compared to V don't matter to the complexity, just that they are near the same size. This leaves us with a fair approximation of the complexity in equation 10.

$$O(n) \approx \; m \times V \cdot P \approx V \times V \times V \approx V^3 \tag{10}$$

A discussion and comparison with the other algorithms can be found in Section 0.5.3.

Brute force

After running the greedy algorithm and collecting some initial results it became clear that it would be difficult to know by just looking at the problem, what a good ε value would be. This lead to the discussion about using a brute force algorithm and if it was possible.

The implementation is extremely simple, even more so than greedy. The reasons why are to do with Python and are explained in Section 0.4. Even though the pseudocode and the real code will differ, the pseudocode help understand the algorithm so is shown in Algorithm 2

When some simple calculations are made about the solution space it is clear as to why the brute force method is not ever going to feasible for this problem.

Searching all solutions, is not a simple task. As it must look at all permutations of solutions which are of length V, the complexity of brute force follows the number of permutations for n distinct objects.[9].

The Big-O complexity is shown in equation 11

$$O(n) \approx \; V! \tag{11}$$

This means that in all cases brute force will run in $\approx V!$ time. Comparison to the other algorithms used can be found in Section 0.5.3.[2]

For example in the 2014 running of Eurovision this would be 37! which, after approximating the time to look at one single solution, would take about 1.2×10^{40} seconds, which is quite obviously computationally infeasible with the time and resources for this project.

Simulated annealing

These results reveal the fact that this problem is not a simple one at all and hence another, more capable algorithm is needed to find more optimal solutions.

The pseudocode for Simulate Annealing can be found in Algorithm 4. It is quite a lot more complex than either brute force or greedy search, but that is for good reason.

The skeleton of the algorithm is shared with both the previous algorithms; that is, lines 22 - 25 in Algorithm 20 are the same as lines 5 - 7 in Algorithm 2 (brute force) and lines 9 -12 in Algorithm 1 (greedy).

The important part where they differ is how they pick a solution to try. Brute force tries every solution one-by-one by permuting through the possible solutions. Greedy search picks a

[2]A full table of Big-O complexity for all algorithms can be found in the appendix in table 8

neighbour of the current solution and sees if that is better than the best it has seen. Where simulated annealing differs is that is also finds a neighbour of the current solution, however where greedy always takes that solution and compares it to the best, Simulated Annealing does two things before that comparison.

Firstly as can be seen on lines 12- 14 of Algorithm 4, it checks whether the new solution is better than the current one. If it is better, then it takes that solution for comparison to the best. However, if it is not better there is still a chance Simulated Annealing will take it for comparison.

In lines 16 - 19, a check is performed that allows worse solutions than the current to be checked against the best. This is usually named an *"Uphill move"*. The reason for accepting a worse solution is because it allows for a more extensive search for the optimal solution. Line 17 is a feature that is designed to prevent the algorithm from becoming stuck at a local minimum that is worse than the global one.[10]. Moreover as it varies with the value of t, it allows the algorithm to take worse solutions at the start, but not take the worse solution nearer the end, meaning the *"Uphill moves"* occur so that the algorithm does not get stuck in a local optima.

An approximation of the worst case Big-O of Simulated Annealing is shown in Table 7.

Table 7: Simulated Annealing complexity per line

Line	Effect on complexity
T_1	run once with V steps
T_2	run once with $V \times P$ steps
T_3	run once with constant steps
T_4	run once with $V \times P$ steps
T_7	$m \times l$ times with constant steps
T_8	$m \times l$ times with $V \times P$ steps
T_9	$m \times l$ times with constant steps
$T_{10} - T_{19}$*	$m \times l$ times with constant steps
$T_{20} - T_{23}$	$m \times l$ as it is the worst case, the assumption is that this block is always run
T_{26}	m times with constant steps
T_{27}	m times with constant steps
T_{29}	run once

* This if-else block can be treated as one because one of the two will always be run

N.B The main loop (lines 5 - 28) is constant, given it comes from *maxIterations*. It adds m times to the instructions inside it. The inside loop (lines 6 - 25) is also constant, given it comes from the temperature length, *tl*. It adds l times to the instructions inside it.

Factoring this down into equation 12 gives:

$$\begin{aligned} f(n) = \; & (V \times T_1) + (V \cdot P \times T_2) + T_3 + (m \times V \cdot P \times T_4) \\ & + (m \times l \times T_7) + (m \times l \times V \cdot P \times T_8) + (m \times l \times T_9) \\ & + (m \times l \times t_{10-19}) \\ & + (m \times l \times T_{20-23}) \\ & + (m \times T_{26}) + (m \times T_{27}) + T_{29} \end{aligned} \tag{12}$$

From this we can safely ignore anything that is less than the highest order as it will dominate the complexity growth. Moreover we can see that the highest order is T_8, so we are left with equation 13

$$f(n) = \; (m \times l \times V \cdot P \times T_8) \approx m \times l \times V \cdot P \tag{13}$$

where $P <= V <= l <= m$. As before this can be further simplified in terms of *voters* (V). In this case, there is an extra term with will be greater than or equal to V. This leaves us with a fair approximation of the complexity in equation 14.[3]

$$O(n) \approx \; m \times l \times V \cdot P \approx V \times V \times V \times V \approx V^4 \tag{14}$$

A full comparison between the algorithms and their complexities is undertaken in Section 0.5.3.

Piecemeal

Part way through the project a slightly different approach to building a solution was suggested. The three algorithms discussed above all have a major thing in common, that they move from a **full** solution to another.

It was suggested that another methodology for finding a solution could be used. This method moves slightly more towards population-based search methods such as Genetic Algorithms, but is actual really a more simplified version of them, that also uses a greedy search algorithm as it's base.

This method begins with just one *voter* instead of a full solution. This *voter* can be found randomly or just be the first in the list. Then to pick the next *voter* to add to the solution, the algorithm looks at all the available *voters* it could add and it picks the one that gives the lowest overall Λ value. The Λ value is the distance between the best and worst who can still win, which at the end is summed to get the solution's ε value.

It does this over and over until it finds a full solution. Where population-based search methods keep track of a *population* of possible solutions, this method named the *piecemeal* method, builds a solution by being doing a greedy search of the available next *voters*.

This method hopes that by picking the best possible next choice this will lead to a good approximation or globally optimal solution. It acts very much like Greedy Search that has been discussed, and the pseudocode supports that, however by working on partial solutions instead of full solutions, the idea is that it will at least approximate a good solution without

[3]A full table of Big-O complexity for all algorithms can be found in the appendix in table 8

having the need to check many full solutions, hence increasing performance, especially for large solution problems.

The pseudocode for it is shown in Algorithm 3 and the implementation is discussed in more detail in Section 0.4.

The main points of interest are that even though lines 7 - 9 look very similar to the three algorithms discussed before, there is a subtle difference. In this algorithm, the value that is being checked against the best is not entertainment (ε) but the distances (Λ).

Another point of interest is a more academic one. It is how do you choose what voter is put at the start. The two most obvious choices are randomly choosing one, which complicates the code slightly, or picking the first from the given list of voters, making the code simpler. The choices and their effect on the ε returned is explored more in Section 0.5

To come to a solution, the algorithm must look at a descending number of possible next solutions from the length of the solution down to 1. In the 2014 Eurovision, this would be 37 choices in the first round, 36 in the second, 35 in the third etc. until all *voters* are accounted for.

The formula in equation 15 describes the series that this algorithm will take, as it is a geometric series from the length of the *voters* down to 1.

$$\sum_{k=0}^{V} \frac{V(V+1)}{2} \tag{15}$$

which can then be simplified for Big-O notation into equation 16. This is because for looking at the complexity of algorithms it is most interesting to look at the term that does the most to the complexity, so in this algorithms case it is the V^2 term.[4]

$$\frac{V(V+1)}{2} = \frac{V^2+V}{2} \approx V^2 + V \approx O(V^2) \tag{16}$$

Looking at the Big-O complexity in equation 16 and comparing it to the others is done it Section 0.5.3.

[4]A full table of Big-O complexity for all algorithms can be found in the appendix in table 8

0.4 Implementation of System

In this section, the main important parts of the projects code will be explored. This will take the theory and pseudocode described in Section 0.3 and discuss the actual code implemented as part of the project. It will both discuss some specific code as well as a discussion as to why parts of the system were implemented in certain ways. All code referenced in this section can be found in the attached zip file, submitted along with this report.

0.4.1 Entertainment Functions wrappers

The ε functions form one of the most important parts of this project, however for them to be implemented they need some contextual data. The functions described below are larger functions that wrap around and allow the ε functions to do their work. The ε functions actually form just the $min(S)$ part of equation 1.

These wrapper functions share an important part, which is shown in 2.[5]

Figure 2: Calculating the cumulative scores per participant

```
1 for j in range(len(solution)):
2   for i in range(len(countries)):
3     v = voters.index(solution[j])
4     scores[i] = scores[i] + score_board[i][v]
5     ...
```

The main point of interest is the two nested *for* loops and specifically lines 3 and 4.

As the distance (Λ) is found as the difference between max and min scores in each round, it is necessary to go through and cumulatively add the scores given to each *participant* in every round. This is stored in the array *scores* at the index for that *participant*: $scores[i]$. The *score_board* is the matrix of all scores given, as discussed in Section 0.2.

These lines are interesting to highlight as they are likely the most critical lines in all the system. Every algorithm must use them at some point and all ε functions will need them to calculate the ε value.

As the score lookup is done against a matrix of scores, it is necessary to find the correct column to look in. The row is the current *participant*: i, however finding the column is a little more complex.

The columns in the matrix are in a certain order, normally alphabetical in Eurovision datasets. This, along with the fact that each solution is a permutation of that order, means that the currently voting country in the solution, $solution[j]$, must be found in the original order to correctly retrieve the score it gives. By keeping track of the original order as it corresponds to the scoreboard, the system only ever has to keep one matrix in memory, along with 2 list of *voters* and *participants*. Any other method would include shuffling the scoreboard into the order of the current solution. As the scoreboard grows i.e in other, larger, problems, this method

[5]Section of code from **getEntertainment** in *support.py*. Full function can be found in attached source code

would become untenable and possibly begin adversely affecting the algorithm's runtimes and complexities.

The method *array.index(element)* returns the index in *array* of the given *element*. In this case *solution*$[j]$, would be a country name such as "Germany", and the value of v will be an integer corresponding to the column.

Looking at the full code listings there are actually two methods that wrap around ε calculation functions to provide them with context and data. These are called **offsetGetEntertainment** and **getEntertainment**, and they both share the code shown in Figure 2. They are also fully independent of the *maxMin* and *refinedMaxMin* functions which are the actual ε functions. These methods differ because of when and why they can be used.

The main reason for these differences is shown in Section 0.3.2 as using different neighbourhood methods leads to different ways to calculate the entertainment.

offsetGetEntertainment

Firstly the more complex **offsetGetEntertainment** is used in conjunction with adjacent neighbour swaps as it relies on using the previous ε value. The theory behind why this works is explained in Section 0.3.2, specifically the Adjacent neighbours subsection and equations 6 and 7.

This function is shown in Figure 19.

The important parts to understand about this implementation are that for it to work it must be given two keys that correspond to the indexes of the *voters* that were swapped to get this solution; the ε value of the old solution, and the array of the distances used to calculate that solution's ε value.

One obvious difference is between the *for* loops in Figure 18 and Figure 2. In the **offsetGetEntertainment** method the *for* loops only need to calculate the scores up the higher of the two keys as they scores after there are unchanged.

There are some further parts that need explanations. The actual recalculation described in Equation 7 is implemented in line 21. On lines 3 and 4, a copy of *countries* and *solution* are taken using Pythons array copy syntax. This is most likely not really necessary but during implementation there was a couple of times where after amending one of the lists locally inside a method, the global version of that list that should have stayed the same, was also changed. This is a little hack that means that any local version of the lists that are amended will not override the original.

Line 20 gets the distances for the two keys, using Python's reverse array accessing. As the l_dist variable is often a partial list of distances, the final element being the second key's distance, the new distances can be returned by accessing the last two elements.

Lines 23 and 24 are necessary so that the new distance values can be passed to this function again, but this time as oldDistances and their distances will be in the correct positions. Essentially what this does is constantly pass around a list of distances, amending it by swapping values before passing it on.

offsetGetEntertainment is certainly faster and more performant for most solutions. This does depend on where in the solution the swaps have taken place (explored in Section 0.5.3.) The main drawback of this method is that is can only be used with adjacent neighbour swaps.

getEntertainment

The second entertainment function wrapper is a simpler version than the one shown in Figure 18. **getEntertainment** is the general purpose wrapper for calculating the entertainment of a given solution. It can work with either of the two neighbourhoods described. This makes it more general purpose, however it also make it slower especially when the solution size is large.

The points of interest here are found in lines 14, 15 and 16. These lines implement that which can be found in Equations1 and 2 exactly as described. Line 14 finds the difference between the max of the scores and the whatever min was returned by the current ε function. Line 15 adds the Λ value found to an array and then line 16 sums all the Λ values found to finally get a ε value. To allow it to interact with the **offsetGetEntertainment** method it also keeps track of the array of Λ values and returns them so they can be passed to **offsetGetEntertainment**

0.4.2 ε functions

The **refinedMaxMin** method is the more interesting of the two ε functions to look at in depth. The original **maxMin** method is very simple and uses the Python in-built *min()* [11] and *max()* [12] methods.

As can be seen in Figure 19 on line 13, the **refinedMaxMin** method is invoked, with the current list of scores for all *participants*, the current solution being tested and what round of voting the competition is at j. The $maxScorePerRound$ is used so that different competitions can define what it is instead of keeping it locally in the function.

The function is shown in Figure 3

Figure 3: refinedMaxMin method

```
1  def refinedMaxMin(scores, solution, j, maxScorePerRound):
2    sorted_scores = sorted(scores[:])
3    currentTop = sorted_scores[-1]
4    minScore = sorted_scores[0]
5    del sorted_scores[-1] # remove highest score
6    roundsRemaining = (len(solution) - 1 - j)
7
8    for score in sorted_scores:
9      if score + (maxScorePerRound * roundsRemaining) < currentTop:
10       minScore = score
11   return minScore
```

The theory behind this method was described in Section 0.3.1. Some of the implementation points to pick up on are line 1 where the in-built Python function *sorted*()[13] is used to take the list of scores and sort them in ascending order. This does slow the method slightly, taking the method's complexity to $n \log n$ but it also simplifies the logic later on as the method does not need to keep track of anything other than the minimum found and only updates it.

This implementation is likely not the most efficient way of finding the refined version of the minimum, however it is quite a simple method. More in depth comparisons between the two methods can be found in Section 0.5, which take into account the complexity and the results that the methods produce.

0.4.3 Random and Adjacent Neighbourhood Functions

The random and adjacent neighbourhood methods are very similar as can be understood from the theory in Section 0.3.2. This does mean they could be written as one single method which are supplied with the keys needed to perform the swaps. This was a thought during their implementation however in the end it was decided to favour duplication over complexity in the system. This means that there *are* two methods that are for all intents and purposes exactly the same, but are named differently.

The only difference occurs on lines 3 and 4 in both methods. Firstly in the adjacent neighbour method the first key is found by getting a random integer between 0 and 2 elements from the end of the list. It must be 2 elements from the end as the second key is taken as the first key + 1. In terms of a list this means the element directly to key1's right. The $len(neighbour)$ part is using Python's in-built length function to calculate the length of the list and hence the last index it can return for correct swapping.[14].

The random neighbour method uses the Python *random* library in line 3 to sample the given order and get two of the elements back.[15]. As this method actually returns the elements and not indexes of those elements, like in the adjacent method, it is necessary to change them back into indexes (line 4) for efficient swapping in line 5.

Figure 4: getAdjacentNeighbour method

```
1 def getAdjacentNeighbour(xNow):
2   neighbour = xNow[:]
3   key1 = random.randint(0, len(neighbour) - 2)
4   key2 = key1 + 1
5   neighbour[key2], neighbour[key1] = neighbour[key1], neighbour[key2]
6 
7   return neighbour, key1
```

Figure 5: getRandomNeighbour method

```
1 def getNeighbour(xNow):
2   neighbour = xNow[:]
3   key1, key2 = random.sample(list(neighbour), 2)
4   index1, index2 = neighbour.index(key1), neighbour.index(key2)
5   neighbour[index2], neighbour[index1] = neighbour[index1], neighbour[index2]
6 
7   return neighbour
```

0.4.4 Simulated Annealing and Greedy Search

The implementation of the search algorithms is not the most interesting part of this project and hence only a little time will be spent explaining their implementation. Both Simulated

Annealing and Greedy Search can be found in the appendices, or completeness, and in the attached source code to be run.

The main reason for not going into depth about their implementations is that both algorithms are essentially general search algorithms that receive the problem specific input from the other function such as **getEntertainment** and **getInitialSolution()**.

A point that is shared across both implementations is the increasing while loop that gets reset to 0 when a new solution has been found. Using Greedy Search as an example, in Figure 21, the while loop start at line 11 with i equal to 0. It is incremented on line 23 after on iteration and check of a new solution. On line 19, the value of i is reset to 0. This allows the *num_iterations* loop to be the number of iterations since a new best solution was found. This minor implementation detail actually has a very measurable impact on the speed of the algorithms. Due to the nature of the problem space, it is likely that both Greedy search and Simulated Annealing reach local (possibly global) optima. When this happens, they are not going to choose a new solution. Furthermore having the $num_iterations$ variable set as a hard number of iterations would require a very deep knowledge of the problem space and how the algorithms are traversing it; so that there is little overhead after they have found the best solution they are going to find. This is not really possible so this method also allows for more wiggle-room.

Another implementation point that is shared is the fact that both algorithms must pass the *oldEntertainment* and *oldDistances* into the **offsetGetEntertainment** method. The reason for this is explained in Section 0.4.1. This also explains the need for the variable $key1$ to be returned from the **getAdjacentNeighbour** method.

A minor point that is related to Python as a language is the ability to return multiple values from a function in a tuple by default. This hugely simplifies the code as methods can then return, both a neighbour and the first key used to find it (**getAdjacentNeighbour**) or a possible solution and the individual distances that make up that solution (**offsetGetEntertainment**). In other languages there are ways to work around this problem, but this solution is one simple way in Python.

Specifically looking at the Simulated Annealing code in Figure 20, the main lines of interest in terms of implementation are line 32 and 34. Line 32 uses the Python *random*[16] library to get a random number between 0 and 1. It then uses the Python *math*[17] library to do the $e^{-deltaC/t}$ part that is necessary for Simulated Annealing to work. These again, are two examples of the Python libraries making implementation very simple for this project.

The general cooling ration values which were explained in Section 0.3.3 on Simulated Annealing, were experimented with throughout the project. The final values are shown in Section 0.5.2 as it forms part of the experimental methodology and the implementation details are quite self explanatory.

0.4.5 Brute force

The implementation for the brute force algorithm was one of the most simple, and this has much to do with the libraries that Python provides. The library in use for the brute force implementation is *itertools*[18], which provides a set of common methods such as *permutations*[19].This makes this implementation extremely simple as I did not have to write any code to correctly permute through the possible solutions without repeating, as the library handled it for me. It can be seen on line 5 of Figure 6.

Figure 6: Brute Force algorithm implementation in Python

```
1  def bruteForce(score_board, countries, voters):
2    best = voters[:]
3    bestCost = support.getEntertainment(voters, countries, score_board, voters)
4
5    for solution in itertools.permutations(voters):
6      currentCost, dist = support.getEntertainment(solution, countries, score_board,
           voters)
7
8      if currentCost < bestCost:
9        best = solution[:]
10       bestCost = currentCost
11
12   return best, bestCost
```

It of course shares many similarities with the other algorithms described so far. Namely, the if block in line 8 - 10 that checks if the current solution is better than the best found so far. Furthermore it uses the same **getEntertainment** method that is used by both the other search algorithms described so far. One small but important point is that, in contrast to Greedy and Simulated Annealing, during the main for loop (lines 5 - 10) in Figure 6, brute force cannot use the **offsetGetEntertainment** method. This is the one drawback of using the *itertools* library as it controls getting a new solution to try, the key needed for **offsetGetEntertainment** to work cannot be passed around. This further slows this brute force implementation however it is not highest order value in terms of the complexity so doesn't make a large difference overall.

0.4.6 Piecemeal algorithm

The **piecemeal** algorithm is the only break from the traditional style of search algorithms that have so far been discussed. Moreover it is the most novel approach to solving this problem in this project.

The theoretical differences were discussed in detail in Section 0.3.3. The code for this algorithm is shown in Figure 22 in the appendix.

Firstly looking at the similarities between this algorithm and the full-solution searches, piecemeal follows the method of checking if a value is less than the best value already seen. At first glance the code at lines 29 - 32 are exactly the same as before. The difference only becomes clear when looking at the larger for loop in which it sits (lines 19 - 35). Whereas the other search algorithms are comparing the ε value of Φ, the piecemeal method is comparing the distance that the current *voter* would add to the current solutions ε value.

Secondly, the concept of *ignoreIndexes* in lines 19, 22 and 23 is necessary so that the correct score is taken from the scoreboard. As the value of j is used as an index in the scoreboard matrix, it is necessary to ignore those indexes of countries that have already been added to the solution. As was discussed in Section 0.4.5, in the brute force method Python took care of not having repeats, so in this method as the algorithm is slightly more complex and Python in0built methods cannot be used, this must be done manually.

0.4.7 Re-Use of Code for future work

One of the more general points that relates to how the project was implemented is the possibility of re-use of the work. Quite early in the project, when discussing other problems that are similar in type to Eurovision, it became clear that the code that was being written would be quite general to all problems. This lead to all the attempt to make all the implementations as re-usable as possible.

This means that the loops have no knowledge of the problem baked into them. This can mainly be seen in that all loops have an upper bound that depends on the length of the lists given. For example Figure 19 on lines 9 and 10.

This is further obvious when looking at the controlling file *order.py* in the attached source code, and what each algorithm needs to run. There are 4 inputs that all the algorithms need to run. These are a a list of *participants*, a list of *voters*, a scoreboard which is a matrix of scores that *voters* give to *participants* and finally the max score that can be given (12 in Eurovision). If a problem can be formalised into these 4 inputs then these methods and algorithms can quite easily be used for other problems, no matter their complexity or size.

0.4.8 Visualisation

A part of this project that was within scope, but not directly part of solving the problem, was creating a way to visualise the solutions that the algorithms produced. The main reason for doing this was to have a way to compare solutions that more closely correlates with how the real solutions would be viewed.

My idea for this was to produce a simple web application that could be given a competition and the solution found by one of the algorithms and then "play out" the competition for a viewer. A screenshot of the application is shown in Figure 7.

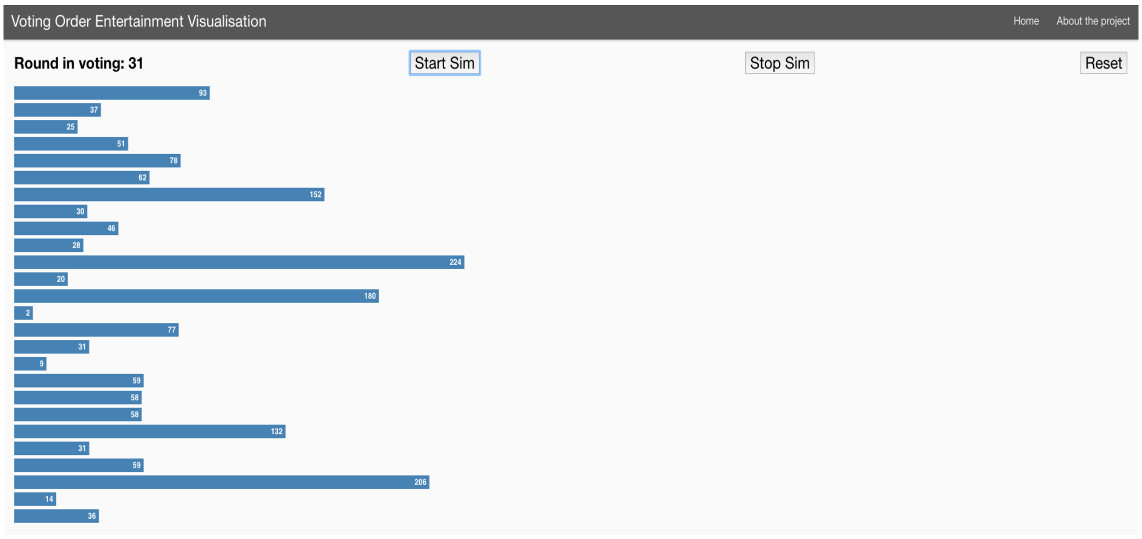


Figure 7: Visualisation of solution to 2014 competition

The application is a Node.js application that uses two main libraries for rendering the chart seen in Figure 7. The first is *d3.js* which is needed to scale the bars to the correct proportions as the scores are calculated. This uses the *d3-scale* library[20] instead of using the whole of *d3*. *d3* could have theoretically taken care of updating of scores each round, however I was not extremely familiar with the way in which it updated state. Moreover for this project I wanted to produce something rather than getting stuck in the implementation details of learning a new library, and the more important part of this project was to produce solutions.

For this reason I turned to *Preact.js*[21], which is a smaller and faster version of the currently very popular library *React*[22] from Facebook. I chose *Preact*, as I have experience with React however I did not need the large number of functions that React provides and I needed the state changing to be fast, as there a calculations that need to be done from round to round. Moreover to speed up starting up the visualisation, the code is based of a *Preact* example application.[23].

The visualisation is very simple, the user clicks a button to begin the competition voting and from there the scores are updated and shown on screen. A timer is used so that only one button press is needed to start the voting and from then on around every 1.5 seconds the new scores are shown, until the end of the competition. This provides anyone viewing the voting a simple way to see how close certain teams are to each other and the current leader, and hence why the algorithms may have settled on this order. A further goal of producing this visualisation, which will be explained in Section 0.6, is using this visualisation as a tool for discovering new entertainment functions for this and other problems.

As with the other code in this project I have attempted to make this as re-usable as possible. It takes the same types of inputs as the algorithms need, and any work to modify the code should only be purely cosmetic, such as changing names of *voters* or *participants*.

The visualisation code is attached in the source code, along with explanations on how to run it to view it locally.

0.5 Results and Evaluation

0.5.1 Overview of goals

The main barrier to saying that this project achieved all it's stated goals is the fact that the optimal solution is not know. The goals set out in Section 0.1 were:

1. a solution that delivers an entertaining competition when the votes are revealed
2. a way of describing entertainment mathematically
3. a way of visualising the competition as the votes are revealed.

As it is the most important of the three goals, the goal of finding entertaining solutions is explored below in Section 0.5.2 in more depth.

Describing the entertainment of a given voting order mathematically was explored theoretically in Section 0.3.1 and their implementations in Section 0.4.2. In terms of success in achieving the goal, I think this project has been successful. The main entertainment function (**refinedMaxMin**) provides a simple and elegant way of enumerating a concept that can be quite difficult to define. The functions not only define an entertaining solution mathematically, but they also allow them to be maximised or minimised easily, so in this regard they are also very accomplished in helping to achieve the first goal. A comparison between the two functions defined during the project and an explanation as to how they were tested against each other and the project can be found in Section **??**. Moreover this will also provide more detail as to why I believe this goal has been successfully reached.

It is more difficult to quantify if the goal of creating a way to visualise solutions has been met or not. In my opinion, the visualisation web application does a very good job of allowing another dimension of this project to be shown. The side of the project that the visualisation helps to shed light on, is essentially ignored by the algorithms and only generalised heavily in the entertainment functions. That is how real people would feel when viewing an "entertaining" solution. The visualisation provides that more visual and tactile feedback to the viewer. Furthermore it helps to justify the other goals without having to go into as much detail about the correctness of the algorithms or the entertainment functions, such as in this report.

0.5.2 Solutions

This section will discuss both the general solutions found by all the algorithms and compare how they performed in finding an optimal solution. Moreover the methodology for collecting and analysing these solutions will be explored. A table of the best and average solutions along with average times for each of the four algorithms can be found in table 9 in the appendix.

Methodology

To find a set of solutions for each algorithm in order to analyse them, the first step was to do some repeated runs. To facilitate some code was added to the main controller *order.py* so that an optional command line parameter could be given which would repeat the running of the given algorithm how ever many times the user input.

The general method was to run each of the algorithms lots of times and record the Φ and ε values for that solution. From this we can be more confident about the performance of each algorithm and make some mathematical comparison between them. To try and keep the experiment fair all the algorithms that take an initial solution were given the same one, that is the one given by the method *getInitialSolution* in *support.py*.

An important point to recognise, before explaining the basic methodology, is that only testing of Greedy search and Simulated Annealing can be done in the exact same way, due to their similarities. Piecemeal and brute force cannot be tested in the same way however the same idea of running them many times to collect many Φ still stands. Their methods are explained later on.

The experiments were designed to give each of the algorithms as much chance as possible to perform at their best. This meant that I attempted to keep the surrounding computation at a minimum so as not to add to the load on the CPU.

The experimental method for both Greedy search and Simulated Annealing are the same save changing filenames. It begins with running a command in the terminal similar to that in Figure 8.

Figure 8: Running algorithms

```
>> python order.py simAnnealing 140 >> results/simAnnealingResults.txt
```

This command needs to be run in the same directory as the controller *order.py* and the algorithm that is being run. In this example this command will run the Simulated Annealing algorithm

140 times and then send the output to a file called *simulatedAnnealingResults.txt* in a *results* subdirectory. The >> part is a **bash** command that means append the output of the preceding command into the file after it. We append as it was necessary to run this code in smaller chunks to reach 1000 runs, so to not create a new file each time, we append from wherever we left off. This will be used multiple times going forward when running algorithms to save their outputs to files. Saving the output from could have be achieved quite easily from within Python as the algorithms themselves are running in Python. However I felt that it was a sound idea to take the saving of the output away from Python as the terminal commands are usually faster and, in my opinion safer to use. They are also much more simple to write and have very little room for error. Moreover, when doing the timing experiments I wanted the least amount of overhead so that as little external time was spent that could accidentally leak into the times for the algorithms.

After doing this I was left with a set of files that contained both a Φ and an ε value per line. These can be seen in the attached results files such as *greedyResults.txt*. A snippet of this file is shown in Figure 9.

Figure 9: Snippet of Greedy results

```
...
(['Russia', 'Portugal', 'Ukraine', 'Poland', 'Sweden', 'Switzerland', 'Montenegro',
'Romania', 'San Marino', 'Italy', 'Latvia', 'Malta', 'The Netherlands', 'Slovenia',
'FYR Macedonia', 'Ireland', 'Hungary', 'Iceland', 'United Kingdom', 'Lithuania',
'Georgia', 'Norway', 'Germany', 'Greece', 'Israel', 'Spain', 'Armenia', 'Estonia',
'Austria', 'France', 'Finland', 'Belgium', 'Denmark', 'Albania', 'Belarus', 'Moldova',
'Azerbaijan'], 3409)
(['Italy', 'Russia', 'Ukraine', 'San Marino', 'Latvia', 'Norway', 'Israel', 'Belarus',
'Slovenia', 'United Kingdom', 'Poland', 'Portugal', 'Romania', 'Georgia', 'Albania',
'Denmark', 'Montenegro', 'Lithuania', 'Switzerland', 'Austria', 'The Netherlands',
'Germany', 'Azerbaijan', 'Armenia', 'Ireland', 'Spain', 'Greece', 'Moldova',
'Hungary', 'Finland', 'Belgium', 'Iceland', 'FYR Macedonia', 'Sweden', 'France',
'Malta', 'Estonia'], 3298)
...
```

Although the Φ are very interesting in their own right, for them to make more sense and the find the best, it was necessary to extract the ε values from this file so that some analysis can be done. To do this I used *ack*[24], which "...is written purely in portable Perl 5 and takes advantage of the power of Perl's regular expressions." This allowed me to extract just the scores from the files like *greedyResults.txt* and then send them to a file such as *greedyScores.txt*[6]. The command for this is shown in Figure 10.

Figure 10: Extracting scores from results file

```
>> ack -o "(?<=,\s)\d{4}(?=\))" algorithmResults.txt > algorithmScores.txt
```

The *ack* tool uses Perl regular expressions for matching the given expression. In this case the regular expression is very simple but looks complex. The d{4} part matches 4 of any digit. The parts before in brackets is a *positive lookbehind* and after in brackets is a *positive lookahead*. These match something before or after the digits but do not keep them in the output; and are used to find a comma and a space before, and the final bracket after the score. The *-o* part is

[6] *greedyScores.txt* can be found in the attached code and results.

a flag that is used to only print whatever is matched, otherwise the whole line where a match was found would be printed and there would be no difference between the results and score files. Once again we use the **bash** commands to send the output to a file, however this time we do not append, but create the file and write to it immediately.

Now we have files that contain 1000 ε values, which we can easily copy into Microsoft Excel for analysis.

When running the two search algorithms there are some setup values that are needed. The main one is *num_iterations*, which both Simulated Annealing and Greedy Search use. To try and keep the tests as close to each other the same value was used $num\_iterations = 500$.

Specific to Simulated Annealing's case there are some more input values that are needed, and they were discussed in Section 0.3.3 in the pseudocode. The final values are shown in Table **??** in the appendix. To reach these values I took into account more general wisdom for each one as well as trying to apply more problem specific knowledge to reach the values that give the best solutions. Furthermore, the time taken to find a solution could be heavily effected by the values so a balance was struck so that the algorithm did not take too long, but also produced consistently good ε values.

The methodology for the Piecemeal algorithm was slightly different. This is due to the fact that given the same first choice the ε value will always be the same. To try and find the best solution means having to try all the possible first choices and saving them.

To do this it was necessary to wrap the piecemeal code as shown in Figure 22 inside a for loop and instead of hard-coding the first element on lines 7, 9 and 10, using the index from the loop.

Once again to extract the scores I used the same *ack* command from Figure 10 to find the scores and send them to a file.

As the brute force algorithm can not be run in the same way as the other three, a different method was devised to test it. The method was to leave it running for 1 hour. This is a fair time to run it for as as will be discussed in the timing section of this section, the other algorithms produce solutions consistently in under 30 seconds. As before the experimental conditions were as similar as possible to the other tests.

Solutions found

The best Φ for the 2014 Eurovision was found by Simulated Annealing with a ε value of 2426 (using **refinedMaxMin**). This is an ordering of the *voters* that keeps the distance between countries that can still win as low as possible for as long as possible.

Figure 11: Best solution

Table 9 in the appendix shows some of the interesting ε values found for all the algorithms used. The Φ corresponding to those values is not shown as they are very long, however finding them is a case of searching the results file for the ε value as they are saved together.

Figure 12 shows the 3 algorithms so far discussed with the minimum, average and maximum ε values so they can be compared.

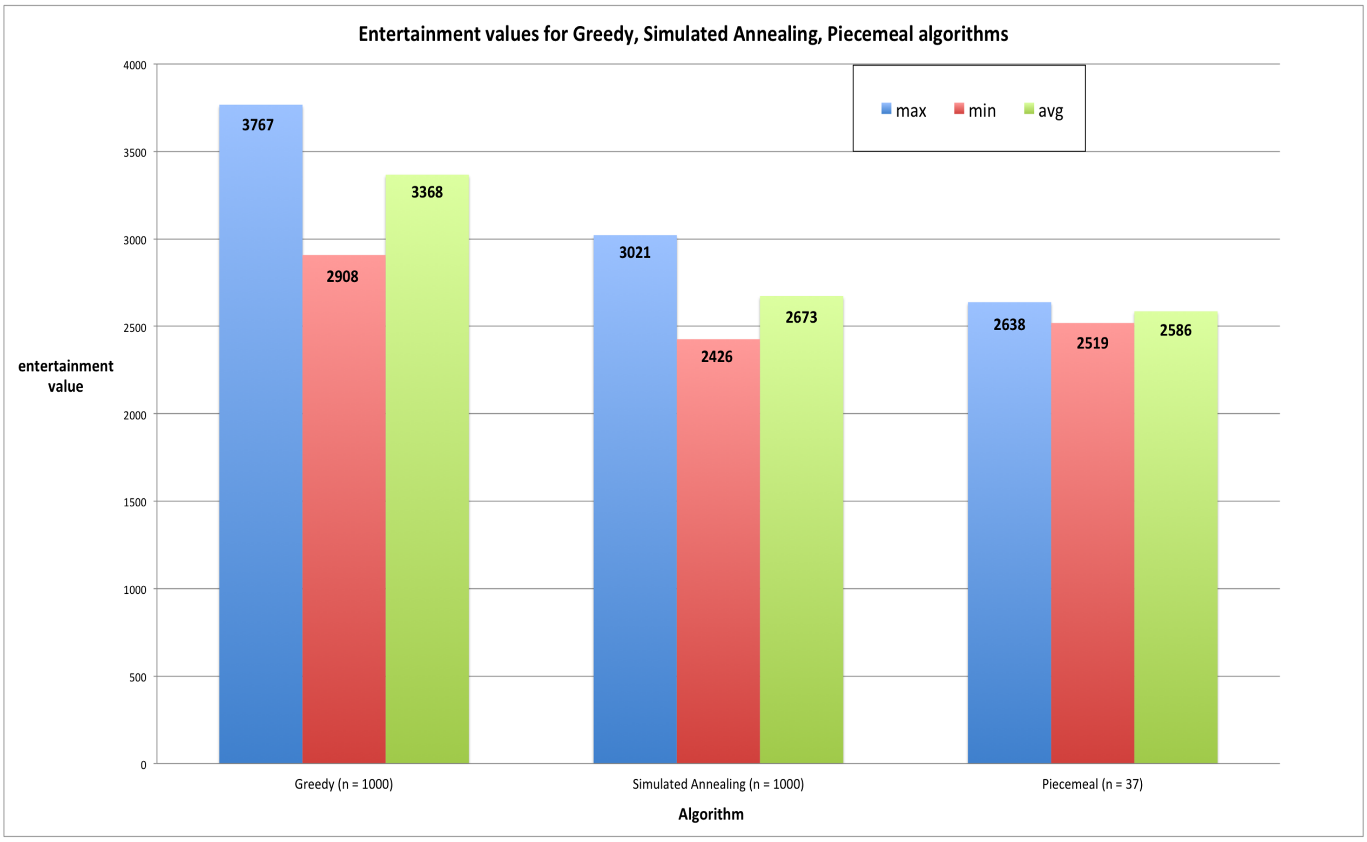


Figure 12: Chart of ε values found by Greedy, Simulated Annealing and Piecemeal

In terms of comparing the algorithms the first point is that Simulated Annealing is clearly the best algorithm as is has found the best ε value and hence solution as discussed above. This is in terms of ε value found, and not with respect to any other measure of quality such as speed. How they compare in other regards is discussed in Section 0.5.3.

The Greedy algorithm does not perform particularly well as it's average ε is much higher than the other two. Moreover the best Φ it found has a ε of 2908, which is only about 100 better than the worst ε that Simulated Annealing found. The large range of values as seen from the large difference between the max and min means that Greedy is not a consistent algorithm when given consistent input, as in this problem. This backs up the idea that lead to the development and use of Simulated Annealing and Piecemeal for this project. If the results produced by Greedy were better then there would have been little gained from the rest of the project.

One interesting point that is not shown in Figure 12 is that if we use a solution given by the Piecemeal algorithm as the initial solution, then Greedy can perform as well as Simulated Annealing (if it is given the same initial solution as before). This increase in quality when given an already good Φ is shared by Simulated Annealing. This leads to the conclusion that Greedy is getting stuck in local optima and as it cannot escape, returns the best Φ that it has.

The piecemeal method was a major success for this problem, especially as it is not a general algorithm which have been proven and tested on many problems. Referring to Figure 12 the chart shows that the piecemeal method held it's own against a much more complex and established algorithm in Simulated Annealing. The lowest ε value that it found was 2519 which less than 100 away from the globally best solution. Moreover as can be seen from the range of values, this method is consistent in returning good solutions. This may be because the only difference between the attempts is the first voter, however that is the only part that is changeable.

0.5.3 Other Testing

The entertainment of a solution is not the only thing that gives it value. This is why some other testing was undertaken to try and give a more well rounded picture of how well the algorithms performed on this problem. Taking all the tests executed together gives a higher level of confidence in the results as there is more experimental data that can then be leaned on when evaluating the project as a whole and when making future decision about algorithm choice for other, similar, problems.

Timing and Big-O Complexity

As well as producing results that maximise entertainment, it is necessary for the algorithm to do so in a reasonable time. Table 10 contains some results for timing of the algorithms.

The methodology for analysing the timing was to run each of the algorithms 100 times each and have timing set to show how long it took for the function that runs the algorithm to finish. The timing is done using the *timeit*[25] library in Python, which purports to take care of many common pitfalls when timing code. It should be advised that these timing values are not necessarily the best, which is why they are averaged over 100 runs.

Figure 13: How to time the algorithms using Python

```
from timeit import default_timer as timer
...
start = timer()
print(gs.greedySearch(SCOREBOARD, PERFORMING_COUNTRIES, VOTING_COUNTRIES, 12))
end = timer()
print('algorithm took: ', end - start)
```

After sending those results into file using the same method as for the real results. I again used *ack* to extract the times and send them to a timing file. For example the command shown in Figure 14

Figure 14: Extracting times from a results file

```
ack -o "(?<=\('algorithm took: ',\s)\d+.\d+" greedyTiming.txt >> greedyTiming.txt
```

From there I copied the values in a new sheet in the attached Results excel file. Direct comparison is quite difficult to visualise as there is a huge difference in the running times of the main algorithms. Due to brute force's special situation no results are collected for it.

The first thing to notice is that Simulated Annealing is the slowest, taking an average of almost 13 seconds to return a solution. Greedy on the other hand is very quick taking on average half a second to return a solution. Finally by far the fastest is Piecemeal which on average takes around 8 milliseconds to return a solution.

So comparing Greedy and Piecemeal to Simulated Annealing we find that Greedy is ≈ 25 times faster $(4s.f)$ and Piecemeal is ≈ 1606 times faster $(4s.f)$. Moreover Piecemeal is ≈ 63 times faster than Greedy.

When we take these timing results with the ε values that each algorithm found, we can make judgments as to why some algorithms are better than others.

Another inherent part of the algorithms is their Big-O complexity. This describes how the algorithms will scale to different size inputs. Explanation of each algorithms complexity in turn can be found in Section 0.3.3 and Table 8 summarises them. The complexity has a large effect of the runtime of the algorithms so it is useful to look at the theorised complexities and see if they compare to experimental runtimes. Although the algorithms have not been run against various sizes of input it can still be interesting to compare between the algorithms.

For example, following the complexities in Table 8, Piecemeal should be the fastest followed by Greedy and then Simulated Annealing, with Brute force taking a computationally infeasible time. This theory is backed up by the timing results that suggest the same. So in this way running the timing experiments is useful for proving the hypotheses explained in Section 0.3, as well as helping with decision making.

Partial and Full ε calculation

It was theorised that, as well as producing more realistic and exciting solutions, the partial calculation method described in Section 0.4.1 also reduces computation time for any algorithm that uses it.

To find out if that is true first it was necessary to run both of the functions **getEntertainment** (which uses full calculation each time) and **offsetGetEntertainment** (which uses partial calculation) side by side. When doing this the actual solutions that were found were discarded as the extra overhead of doing 2 calculations of the same entertainment value could have adversely effected the results. The timing method was the same as used for the general algorithms (see Figure 13).

To get the results, the Greedy algorithm was run and its output saved to a file as before. To distinguish the times, the full calculation was output with "reduced" before and the partial time with "partial" before it. Once again *ack* was used to extract the times and save them to another file specific for either reduced or full times. this produced 10,000 values for each of the methods which were then analysed in Excel.

Figure 15 shows the average time over those 10,000 iterations. The actual time taken is not the interesting part, more that the reduced method takes around half the time. This result supports the hypothesis above in that is shows that the computation time for the reduced calculation is indeed better. This also means that the computation time for any algorithm using it is also going to be lower. A further point that speaks to why the reduced method is better is that it is used many, many times during the solving of this problem, so any saving in terms of computation time will be heavily felt by the overall time to reach a solution.

One drawback of this method is that it does not always guarantee the same reduction in time over the full method. As was described in Section 0.3.2, the amount of calculations is dependent on where in the order the swaps take place. The number of calculations range from being at worst the same as the full method to being only around 50 calculations. Furthermore if the reduced method must calculate the same or nearly the same number as the full method, it is likely that the time reductions will have been lost due to setup overhead in the function.

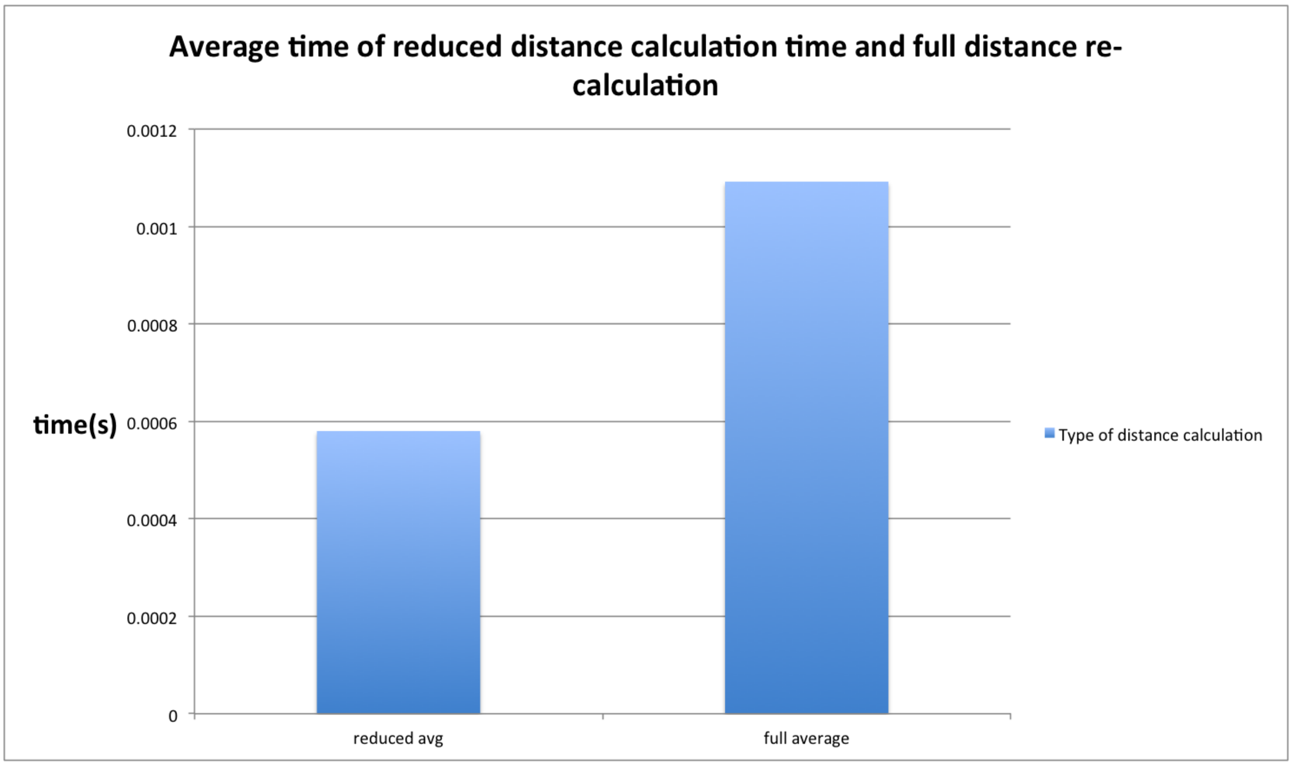


Figure 15: Average time taken to perform full and partial ε calculation on the same solution

To make sure that the reduced method was indeed correctly calculating the ε value it was decided to run them side by side and compare their results. This could quite easily be done as the testing setup necessary to log the times required they both be run every time. This added to the confidence in the ability as it was quite easy to return to test that at any time after any major changes to the algorithms or main support methods.

maxMin and refinedMaxMin

The theoretical reasons why the **refinedMaxMin** method is better have been discussed, such as giving a more real picture of entertainment. To try and find some evidence to see why intuitive idea may be, I tested the minimum values that each method returned. To do this I followed the same method as for the full and partial ε calculations test, that is to run both methods side by side and return the minimum values that were found in two lists which could then be analysed.

The effect that the refined method has on the minimum value that is used in Equation 1, is shown in Figure 16. The graph shows the minimum scores that are returned by the two methods as green and blue points, as well as the difference between the two values (red line and red numbers) over a single typical competition. From this graph it can be seen that the **maxMin** method stays relatively low as in this competition the lowest score at the end was only 2 points. On the other hand, the **refinedMaxMin** method diverges at around round 22. This is when it can start ruling out *performers* from winning the overall crown. By the last few round the difference is quite large as there are only a few countries that could still win so only their scores should be taken into account for entertainment.

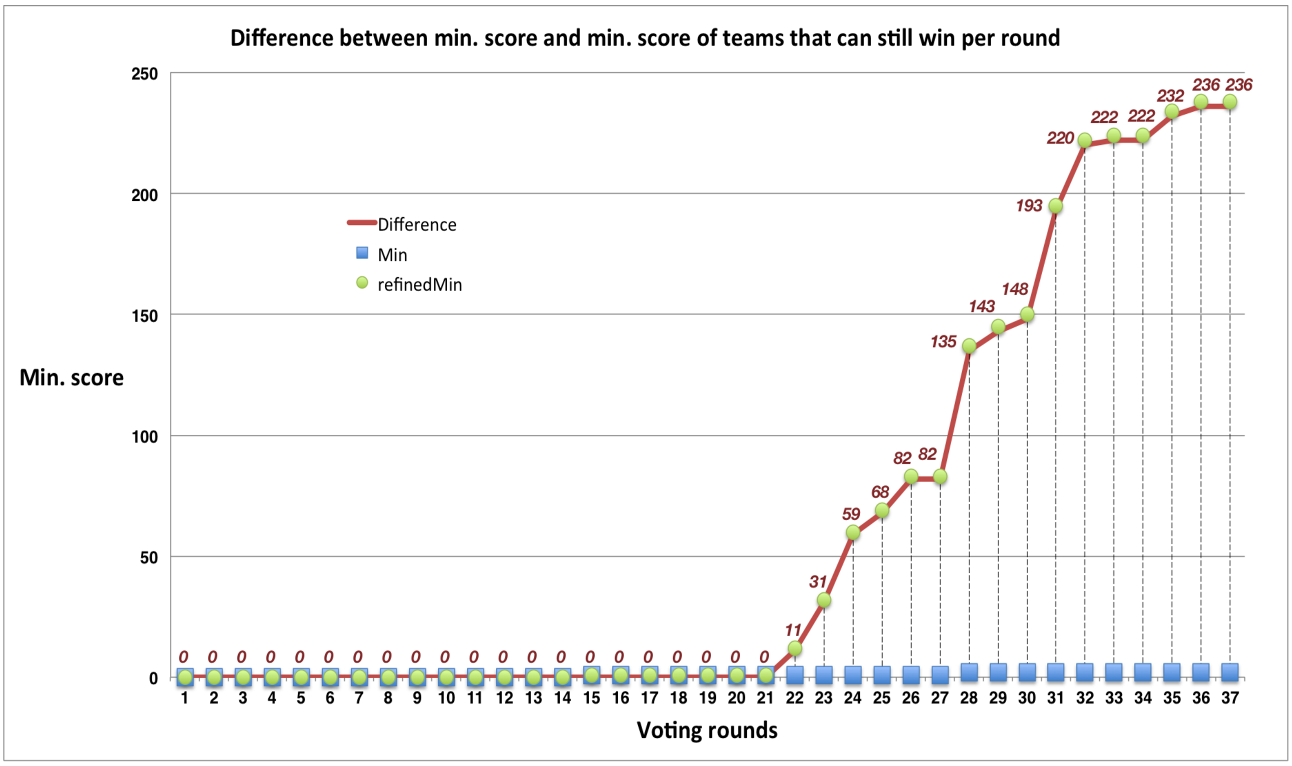


Figure 16: Difference between min value found for MaxMin method and RefinedMaxMin

This chart helps to explain mathematically why the idea of removing teams that can no longer win from the calculation can aid in achieving better solutions. The **maxMin** method only ever really returns a value less than 5, which corresponds to the country currently in last place. It is easy to see nearer the end of the competition those countries are not adding anything to the entertainment of who will win as their scores are near 0. On the other hand, the **refinedMaxMin** method begins to prune the losing countries, and by the end is only looking at a small group of countries that can still win.

Another point to take into account is the complexities that these methods have and how they can affect the choice to use them. Table 3 in Section 0.3.1 shows the complexities of the two methods. Taking this with the results from Figure 16, even though the complexity of the refined method is slightly higher, the value that it adds means that the trade-off is worth it in my opinion.

Comparing solutions

Throughout the project the same problem kept appearing while trying to compare the intermediate or final Φ. For my piece of mind I wanted to know if similar Φ were giving similar ε values and have a quick way to compare two Φ without having to manually look at country names and compare in what position they were in. Moreover having a way to almost immediately know if two orders were the same was very helpful as separating orders based solely on their ε values was not good enough as they could be shared by vastly different orders. It was also helpful to find a way to see if there were common groupings or positioning of *voters* in given Φ.

To solve this problem a quick and simple Python script was written which can be see in Figure 17. In the full source code that is attached, the code differs slightly as this code is not meant for use during the running of any algorithms so the file has manually hard-coded values for the orders, *order*1 and *order*2.

The idea behind it is very simple, if there are two orders that are different, sum up by how many places the same *voters* are separated in each. This occurs on line 4, where the *dist* variable is the total sum of the difference between the indexes of the same *voter*. By using the *abs*() method from the Python standard library it was possible to not worry about which index was higher and hence keep the code simple.

Figure 17: Finding the difference between two orders

```
1 dist = 0
2 for country in (order1):
3   index1, index2 = order1.index(country), order2.index(country)
4   dist = dist + abs(index1 - index2)
5 print(dist)
```

Unit tests

As the main algorithms do not have repeatable behaviour it is not possible to unit test them without defeating the purpose of unit testing. There are however some functions that can be unit tested. The file *orderTest.py* contains these tests. The methods that are tested are **getAdjacentNeighbour**, **getNeighbour**, and **refinedMaxMin**.

The tests themselves should be fairly self explanatory they call the function with set inputs and assert that the output from that function matches some value or set of values.

Although these tests are not strictly necessary for the completeness of the project, they help to provide another level of confidence along with the in-place testing described in this section. The tests are not the most comprehensive tests and only really test the simple working of the methods. The main reason for the lack of unit testing is that this project does not have an already known result, which is something that unit tests require.

0.5.4 Critical Evaluation

0.6 Future Work

0.7 Conclusions

0.8 Reflection on Learning

Glossary of Terms

1. **Voters** (V): All countries that are in the Eurovision song contest who vote in the final. The voters is a list of countries. It looks like [”Ukraine”, ”Austria”, ”France”,....]
2. **Participants** (P): A subset of the voters that perform songs in the final and receive points from the other voters.
3. **Solution** (Φ): A solution to the optimisation problem this project is attempting to solve. A solution consists of two things:
    - An ordering of the voting countries as a list
    - An entertainment value found by an entertainment function
4. **Entertainment Value** (ε): A value given to a solution that describes how entertaining it is. Calculated using an entertainment function given a solution.
5. **Round** (R): One round is when one *voter* has given all the *participants* a score. The competition is made up of n_R rounds where $n_R = length of V$.
6. **Scores** (S): How many points each country has received per round. The scores are an array of the same length as the number of *participants*. It looks like [0,0,2,12,7,4,0,0,3,2,10,....]. When referring to it in the report along with **rounds** it is most likely cumulative so that after the final round the scores are the total scores for each country.

Table of Abbreviations

Appendices

Table 8: Big-O complexities of algorithms used

	Big-O time complexity
Greedy Search	$O(V^3)$
Brute Force	$O(V!)$
Simulated Annealing	$O(V^4)$
Piecemeal	$O(V^2)$

All algorithms are expressed in terms of V: *voters* and P: *participants*. More explanation and calculations can be found in Section 0.3.

Algorithm 1 Greedy Search

```
Require: list of voters
Require: scoreBoard
Require: list of performers
1: $xNow \leftarrow getInitialSolution()$
2: $xBest \leftarrow getInitialSolution()$
3: $entertainmentXBest \leftarrow getEntertainment(xBest)$
4: while $i < maxIterations$ do
5:     $xNow = getNeighbour(xNow)$
6:     $entertainmentXNow = getEntertainment(xNow)$
7:     if entertainmentXNow < entertainmentXBest then
8:         $xBest = xNow$
9:         $entertainmentXBest = entertainmentXNow$
10:        $i = 0$
11:    end if
12:    $i = i + 1$
13: end while
14: return XBest, EntertainmentXBest
```

Algorithm 2 Brute Force

```
Require: list of voters
Require: scoreBoard
Require: list of performers
1: $xBest \leftarrow getInitialSolution()$
2: $entertainmentXBest \leftarrow getEntertainment(xBest)$
3: for all $S \in Solutions$ do
4:     $entertainmentXNow = getEntertainment(S)$
5:     if entertainmentXNow < entertainmentXBest then
6:         $xBest = xNow$
7:         $entertainmentXBest = entertainmentXNow$
8:     end if
9: end for
10: return XBest, EntertainmentXBest
```

Algorithm 3 Piecemeal Search

```
Require: list of voters
Require: scoreBoard
Require: list of performers
 1: [$solution$] ← $chooseFirstVoter()$
 2: $best$ ← $chooseFirstVoter()$
 3: [$distances$] ← 12
 4: for $k = 0$ to length(voters) -1 do
 5:     for all $Voter \in possibleNextVoters$ do
 6:         calculate $\Lambda$ with that voter as next in solution
 7:         if $\Lambda < bestDistance$ then
 8:             $bestDistance = \Lambda$
 9:             $best = Voter$
10:         end if
11:     end for
12:     [$solution$] ← $best$
13:     [$distances$] ← $bestDistance$
14: end for
15: $entertainmentSolution$ ← $sum(distances)$
16: return solution, entertainmentSolution
```

Figure 18: offsetGetEntertainment method

```python
def offsetGetEntertainment(solution, countries, score_board, voters, key1,
    oldEntertainment, oldDistances, maxScorePerRound):

    entertainmentValue = 0
    performing_countries = countries[:]
    current_solution = solution[:]
    distances = oldDistances[:]
    key2 = key1 + 1

    oldDistance1, oldDistance2 = oldDistances[key1], oldDistances[key2]

    l_dist = []
    scores = [0] * 26
    for j in range(key2 + 1):
        for i in range(len(countries)):
            v = voters.index(solution[j])
            scores[i] = scores[i] + score_board[i][v]
        otherMin = refinedMaxMin(scores, solution, j, maxScorePerRound)
        l_dist.append(max(scores) - otherMin)

    newDistance1, newDistance2 = l_dist[-2], l_dist[-1]
    entertainmentValue = oldEntertainment - (oldDistance1 + oldDistance2) +
        (newDistance1 + newDistance2)

    distances[key1] = newDistance2
    distances[key2] = newDistance1

    return entertainmentValue, distances
```

Algorithm 4 Simulated Annealing

```
Require: list of voters
Require: scoreBoard
Require: list of performers
Require: ti: Initial Temperature
Require: tl: Temperature length
Require: cr_coefficient: cooling coefficient
1: $xNow \leftarrow getInitialSolution()$
2: $entertainmentXNow \leftarrow getEntertainment(xNow)$
3: $xBest \leftarrow getInitialSolution()$
4: $entertainmentXBest \leftarrow getEntertainment(xBest)$
5: while $i < maxIterations$ do
6:     for $j$ to $tl$ do
7:         $xPrime = getNeighbour(xNow)$
8:         $entertainmentXPrime = getEntertainment(xPrime)$
9:         $deltaC = entertainmentXPrime - entertainmentXNow$
10:        if deltaC <= 0 then
11:            $xNow = xPrime$
12:            $entertainmentXNow = entertainmentXPrime$
13:        else
14:            $q$ = random int between 0 and 1
15:            if $q < e^{-deltaC/t}$ then
16:                $xNow = xPrime$
17:                $entertainmentXNow = entertainmentXPrime$
18:            end if
19:        end if
20:        if entertainmentXNow < entertainmentXBest then
21:            $xBest = xNow$
22:            $entertainmentXBest = entertainmentXNow$
23:            $i = 0$
24:        end if
25:    end for
26:    $t = t \times$cr_coefficient
27:    $i = i + 1$
28: end while
29: return XBest, EntertainmentXBest
```

Figure 19: getEntertainment method

```python
def getEntertainment(solution, countries, score_board, voters, maxScorePerRound):
  entertainmentValue = 0
  performing_countries = countries[:]
  current_solution = solution[:]

  distances = []
  iters = 37
  scores = [0] * 26
  for j in range(len(solution)):
    for i in range(len(countries)):
      v = voters.index(solution[j])
      scores[i] = scores[i] + score_board[i][v]
    otherMin = refinedMaxMin(scores, solution, j, maxScorePerRound)
    distance = max(scores) - otherMin
    distances.append(distance)
  entertainmentValue = sum(distances)

return entertainmentValue, distances
```

Figure 20: Simulated Annealing code

```python
def simulatedAnnealing(score_board, performers, voters, maxScorePerRound):
  num_iterations = 500
  ti = 4000
  tl = 40
  cr_coefficient = 0.92
  i = 0

  t = ti

  xNow = support.getInitialSolution(performers, score_board, voters,
      maxScorePerRound)
  entertainmentXNow, distxNow = support.getEntertainment(xNow, performers,
      score_board, voters, maxScorePerRound)

  xBest = xNow[:]
  entertainmentXBest = entertainmentXNow
  oldEntertainment = entertainmentXNow
  oldDistances = distxNow

  while i < num_iterations:
    for j in range(tl):
      xPrime, key1 = support.getAdjacentNeighbour(xNow)

      entertainmentXPrime, distXPrime = support.offsetGetEntertainment(xPrime,
          performers, score_board, voters, key1, oldEntertainment, oldDistances,
          maxScorePerRound)

      deltaC = entertainmentXPrime - entertainmentXNow

      if deltaC <= 0:
        xNow = xPrime[:]
        entertainmentXNow = entertainmentXPrime
        oldEntertainment = entertainmentXPrime
        oldDistances = distXPrime
      else:
        q = random.randint(0, 1)

        if q < math.exp(-(deltaC)/t):
          xNow = xPrime[:]
          entertainmentXNow = entertainmentXPrime
          oldEntertainment = entertainmentXPrime
          oldDistances = distXPrime

        if entertainmentXNow < entertainmentXBest:
          xBest = xNow[:]
          entertainmentXBest = entertainmentXNow
          i = 0

    t = t * cr_coefficient
    i = i + 1
  return xBest, entertainmentXBest
```

Figure 21: Greedy Search code

```
def greedySearch(score_board, performers, voters, maxScorePerRound):
  num_iterations = 500
  iters = 0
  xNow = support.getInitialSolution(performers, score_board, voters,
      maxScorePerRound)
  xBest = xNow[:]
  entertainmentXBest, distances = support.getEntertainment(xNow, performers,
      score_board, voters, maxScorePerRound)
  oldEntertainment = entertainmentXBest
  oldDistances = distances
  i = 0

  while i < num_iterations:
    xNow, key1 = support.getAdjacentNeighbour(xNow)

    entertainmentXNow, distancesXNow = support.offsetGetEntertainment(xNow,
        performers, score_board, voters, key1, oldEntertainment, oldDistances,
        maxScorePerRound)

    if entertainmentXNow < entertainmentXBest:
      xBest = xNow[:]
      entertainmentXBest = entertainmentXNow
      i = 0

    oldEntertainment = entertainmentXNow
    oldDistances = distancesXNow
    i = i+1

  return xBest, entertainmentXBest
```

Figure 22: Piecemeal algorithm

```python
def stepByStepSolution(score_board, countries, voters, maxScorePerRound):
    entertainmentValue = 0
    performing_countries = countries[:]
    solution = []
    voting_countries = voters[:]
    distances = []
    best = voting_countries[0]
    ignoredIndexes = []
    bestScores = [row[0] for row in score_board]
    solution.append(voting_countries[0])
    distances.append(12)

    for k in range(len(voting_countries) - 1):
      scores = bestScores[:]
      countries_tried = []
      bestDistance = -1
      ignoredIndexes.append(voting_countries.index(best))

      for j in range(len(voting_countries)):
        local_scores = scores[:]
        current_country = voting_countries[j]
        if j in ignoredIndexes:
          continue
        for i in range(len(performing_countries)):
          local_scores[i] = score_board[i][j] + local_scores[i]
        otherMin = support.refinedMaxMin(local_scores, voting_countries, k,
            maxScorePerRound)
        distance = max(local_scores) - otherMin

        if distance < bestDistance or bestDistance < 0:
          bestDistance = distance
          best = current_country
          bestScores = local_scores[:]
        countries_tried.append(current_country)
      distances.append(bestDistance)
      solution.append(best)
    entertainmentValue = sum(distances)

  return solution, entertainmentValue
```

Table 9: Best and average solutions found by algorithms

Algorithm	Type of solution	ε value
Brute force*	Best	3125
Greedy Search	Best	2908
	Average	3368
Simulated Annealing	Best	2426
	Average	2673
Piecemeal	Best	2519
	Average**	2586

Average taken over 1000 solutions unless otherwise stated

* Brute force was only run once so only has 1 solution, it's best.

** Piecemeal was run 37 different times, one for each different starting country.

Table 10: Timing results for main algorithms (all times in seconds to 4 s.f)

	Greedy	Simulated Annealing	Piecemeal
Min	0.2377	11.8288	0.0063
Max	1.4933	17.4252	0.0130
Average	0.5030	12.8072	0.0080
Median	0.4074	12.5793	0.0075

References

[1] Wikipedia, "Voting at the eurovision song contest — wikipedia, the free encyclopedia," 2016. `https://en.wikipedia.org/w/index.php?title=Voting_at_the_Eurovision_Song_Contest&oldid=756839176`; [Online; accessed 28-January-2017].

[2] D. Gatherer, "Comparison of eurovision song contest simulation with actual results reveals shifting patterns of collusive voting alliances.," *Journal of Artificial Societies and Social Simulation*, vol. 9, no. 2, p. 1, 2006. `http://jasss.soc.surrey.ac.uk/9/2/1.html`.

[3] L. Spierdijk and M. Vellekoop, "Geography, culture, and religion: Explaining the bias in eurovision song contest voting," February 2006. `http://doc.utwente.nl/66198/`; [Online; accessed 28-January-2017].

[4] V. Ginsburgh and A. G. Noury, "The eurovision song contest. is voting political or cultural?," *European Journal of Political Economy*, vol. 24, no. 1, pp. 41 – 52, 2008. urlhttp://www.sciencedirect.com/science/article/pii/S0176268007000547.

[5] G. Bello, H. Menéndez, and D. Camacho, "Using the clustering coefficient to guide a genetic-based communities finding algorithm," in *Intelligent Data Engineering and Automated Learning - IDEAL 2011: 12th International Conference, Norwich, UK, September 7-9, 2011. Proceedings* (H. Yin, W. Wang, and V. Rayward-Smith, eds.), pp. 160–169, Berlin, Heidelberg: Springer Berlin Heidelberg, 2011. `http://dx.doi.org/10.1007/978-3-642-23878-9_20`.

[6] EBU, "'good evening copenhagen' - voting order revealed," 10 May 2014. `http://www.eurovision.tv/page/news?id=good_evening_copenhagen_-_voting_order_revealed`; [Online; accessed 11-April-2017].

[7] H. Wussing, *The Genesis of the Abstract Group Concept: A Contribution to the History of the Origin of Abstract Group Theory*, ch. 2, p. 94. Courier Dover Publications, 2007. "Cauchy used his permutation notation—in which the arrangements are written one below the other and both are enclosed in parentheses—for the first time in 1815.".

[8] Wikipedia, "Cyclic permutation — wikipedia, the free encyclopedia," 2016. `https://en.wikipedia.org/w/index.php?title=Cyclic_permutation&oldid=755419947`; [Online; accessed 18-April-2017].

[9] Wikipedia, "Permutation — wikipedia, the free encyclopedia," 2017. `https://en.wikipedia.org/w/index.php?title=Permutation&oldid=772638699`; [Online; accessed 19-April-2017]; 'The number of permutations of n distinct objects is n factorial, usually written as n!,...'.

[10] Wikipedia, "Simulated annealing — wikipedia, the free encyclopedia," 2017. `https://en.wikipedia.org/w/index.php?title=Simulated_annealing&oldid=775987067`; [Online; accessed 19-April-2017].

[11] Python-Software-Foundation, "Python built-in functions - min," 2017. `https://docs.python.org/2/library/functions.html#min`; [Online; accessed 22-April-2017].

[12] Python-Software-Foundation, "Python built-in functions - max," 2017. `https://docs.python.org/2/library/functions.html#max`; [Online; accessed 22-April-2017].

[13] Python-Software-Foundation, "Python built-in functions - sorted," 2017. `https://docs.python.org/2/library/functions.html#sorted`; [Online; accessed 22-April-2017].

[14] Python-Software-Foundation, "Python built-in functions - len," 2017. `https://docs.python.org/2/library/functions.html#len`; [Online; accessed 22-April-2017].

[15] Python-Software-Foundation, "Python random library - sample," 2017. `https://docs.python.org/2/library/random.html#random.sample`; [Online; accessed 22-April-2017].

[16] Python-Software-Foundation, "Python random library," 2017. `https://docs.python.org/2/library/random.html`; [Online; accessed 22-April-2017].

[17] Python-Software-Foundation, "Python math library," 2017. `https://docs.python.org/2/library/math.html`; [Online; accessed 22-April-2017].

[18] Python-Software-Foundation, "Python itertools library," 2017. `https://docs.python.org/2/library/itertools.html`; [Online; accessed 22-April-2017].

[19] Python-Software-Foundation, "Python itertools library - permutation," 2017. `https://docs.python.org/2/library/itertools.html#itertools.permutations`; [Online; accessed 22-April-2017].

[20] d3, "D3 - scalelinear," 2016. `https://github.com/d3/d3-scale#scaleLinear`; [Online; accessed 17-February-2017].

[21] Preact, "Preact.js," 2016. `https://preactjs.com/`; [Online; accessed 11-February-2017].

[22] Facebook, "React," 2016. `https://facebook.github.io/react/`; [Online; accessed 11-February-2017].

[23] developit, "Preact-boilerplate," 2017. `https://github.com/developit/preact-boilerplate`; [Online; accessed 11-February-2017].

[24] "ack," 2017. `https://beyondgrep.com/`; [Online; accessed 10-April-2017].

[25] Python-Software-Foundation, "Python timeit library," 2017. `https://docs.python.org/2/library/timeit.html`; [Online; accessed 23-April-2017].

[26] Python-Software-Foundation, "Common python method's complexity," 2017. `https://wiki.python.org/moin/TimeComplexity`; [Online; accessed 25-April-2017].