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

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.

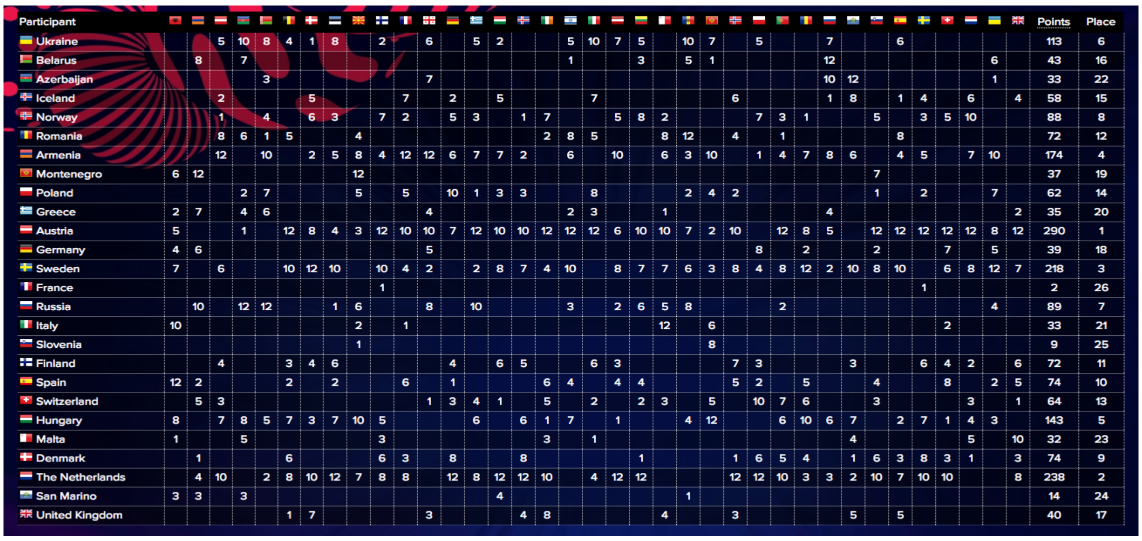
0.2.2 Background Theory

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)

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.

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

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.

Firstly,

T_1 = run once with V steps

T_2 = run once

T_3 = run once with $V \times P$ steps

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
```

The main loop (lines 6 - 15) is constant, given it comes from *maxIterations*. It adds m times to the instructions inside it. So:

$T_5 = m$ 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

$T_{12} = m$

$T_{14} = 1$

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.[2]

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

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

A discussion and comparison with the other algorithms can be found in Section 0.5.

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

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
```

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]

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.

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

The pseudocode for Simulate Annealing can be found in Alorithm 3. It is quite a lot more complex than either brute force or greedy search, but that is for good reason.

Algorithm 3 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
```

The skeleton of the algorithm is shared with both the previous algorithms; that is, lines 22 - 25 in Algorithm 3 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 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 3, 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.

To find an approximation for the worst case Big-O of Simulated Annealing:

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

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.

$T_7 = m \times l$ times with constant steps

$T_8 = m \times l$ times with $V \times P$ steps

$T_8 = m \times l$ times with constant steps

$T_{10} - T_{19} = m \times l$ times with constant steps, this if-else block can be treated as one because one of the two will always be run.

$T_{20} - T_{23} = m \times l$ 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

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.[4]

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

A full comparison between the algorithms and their complexities is undertaken in Section 0.5.

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

The pseudocode for it is shown in Algorithm 4 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

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

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

Algorithm 4 Piecemeal Search

```
Require: list of voters
Require: scoreBoard
Require: list of performers
 1: $[solution] \leftarrow chooseFirstVoter()$
 2: $best \leftarrow chooseFirstVoter()$
 3: $[distances] \leftarrow 12$
 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:    $[solution] \leftarrow best$
13:    $[distances] \leftarrow bestDistance$
14: end for
15: $entertainmentSolution \leftarrow sum(distances)$
16: return solution, entertainmentSolution
```

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.

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

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 n^2 term.[5]

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

Looking at the Big-O complexity in equation 16 and comparing it to the others is done it Section 0.5.

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

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.[6]

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

[6]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 4

Figure 3: getEntertainment method

```
 1 def offsetGetEntertainment(solution, countries, score_board, voters, key1,
       oldEntertainment, oldDistances, maxScorePerRound):
 2
 3     entertainmentValue = 0
 4     performing_countries = countries[:]
 5     current_solution = solution[:]
 6     distances = oldDistances[:]
 7     key2 = key1 + 1
 8
 9     oldDistance1, oldDistance2 = oldDistances[key1], oldDistances[key2]
10
11     l_dist = []
12     scores = [0] * 26
13     for j in range(key2 + 1):
14         for i in range(len(countries)):
15             v = voters.index(solution[j])
16             scores[i] = scores[i] + score_board[i][v]
17         otherMin = refinedMaxMin(scores, solution, j, maxScorePerRound)
18         l_dist.append(max(scores) - otherMin)
19
20     newDistance1, newDistance2 = l_dist[-2], l_dist[-1]
21     entertainmentValue = oldEntertainment - (oldDistance1 + oldDistance2) +
           (newDistance1 + newDistance2)
22
23     distances[key1] = newDistance2
24     distances[key2] = newDistance1
25
26     return entertainmentValue, distances
```

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 3 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 0.5.2.) 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 3. **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**

Figure 4: getEntertainment method

```
1  def getEntertainment(solution, countries, score_board, voters, maxScorePerRound):
2    entertainmentValue = 0
3    performing_countries = countries[:]
4    current_solution = solution[:]
5
6    distances = []
7    iters = 37
8    scores = [0] * 26
9    for j in range(len(solution)):
10     for i in range(len(countries)):
11       v = voters.index(solution[j])
12       scores[i] = scores[i] + score_board[i][v]
13     otherMin = refinedMaxMin(scores, solution, j, maxScorePerRound)
14     distance = max(scores) - otherMin
15     distances.append(distance)
16   entertainmentValue = sum(distances)
17
18 return entertainmentValue, distances
```

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 4 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 5

Figure 5: 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 6: 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 7: 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 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 8. 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, $order1$ and $order2$.

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 8: Finding the difference between two orders

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

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*[16], which provides a set of common methods such as *permutations*[17].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 10.

Figure 9: 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 10, 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. The theoretical differences were discussed in detail in Section 0.3.3. Moreover it is the most novel approach to solving this problem in this project.

Figure 10: Piecemeal algorithm

```
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
```

0.4.7 Re-Use of Code for future work

0.5 Results and Evaluation

0.5.1 Solutions

talk about solutions found

0.5.2 Partial and Full ε calculation

A full calculation of the entertainment value, such as in the random swap method involves {number of voters $\times$ number of participants}; in the 2014 version of the problem this always involves: $37 * 26 = 962$ score calculations (37 = number of voters, 26 = number of participants).

This means that the partial entertainment calculation will involve: -

- Max = 37 * 26 = 962 score calculations (same as full)
- Min = 2 * 26 = 52 score calculations
- Average = 18 * 26 = 468 score calculations

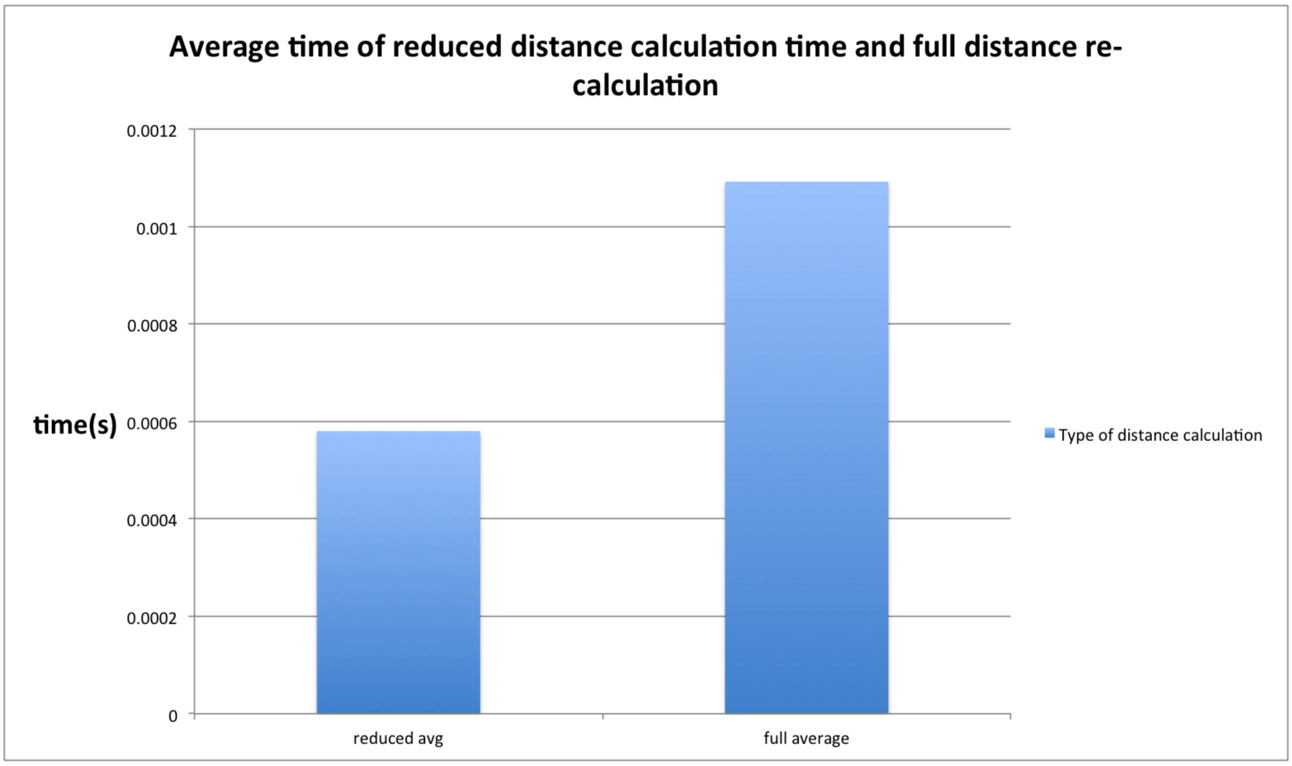


Figure 11: Average time taken to perform full and partial ε calculation on the same solution

0.5.3 maxMin and refinedMaxMin

The effect this has on the minimum value that is used in Equation 1, is shown in Figure 12. 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 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.

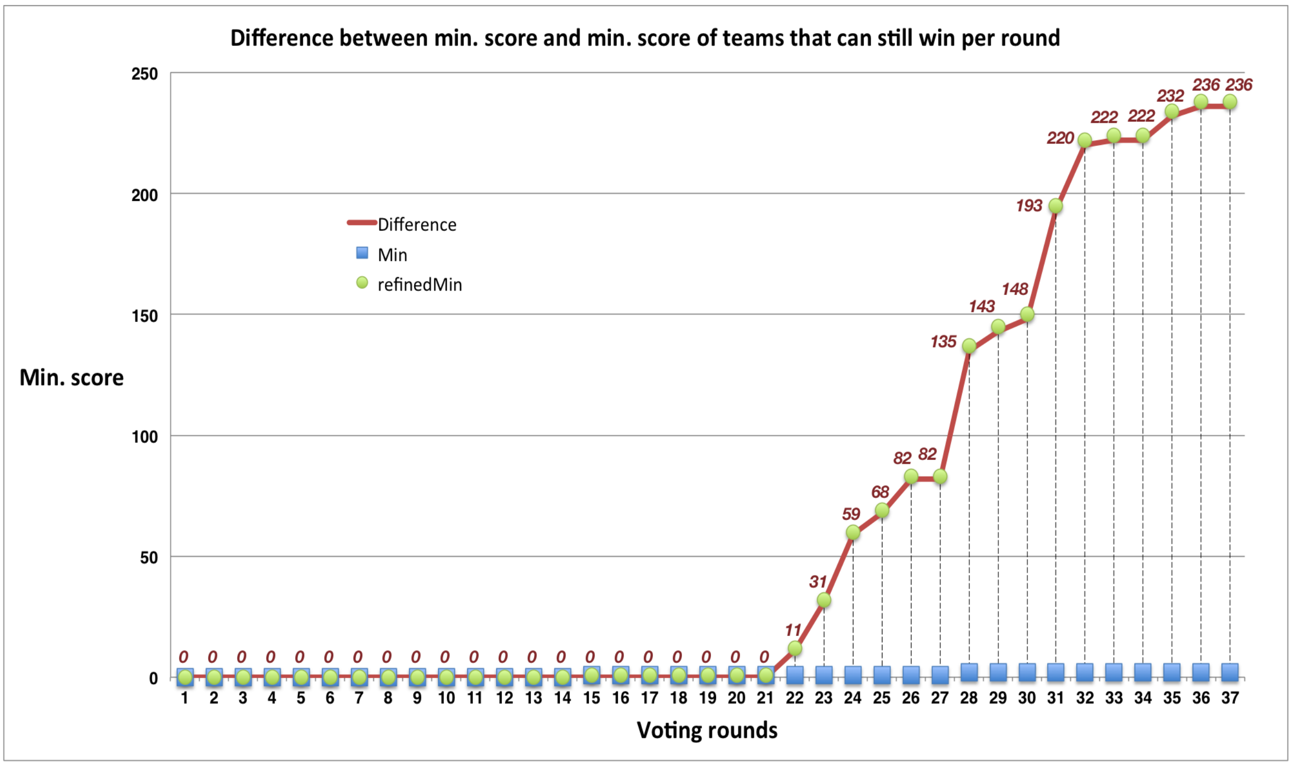


Figure 12: Difference between $min(scores)$ value found for MaxMin method and Refined-MaxMin

0.5.4 Comparison of Big-O Complexities

Comparing Simulated Annealing to Greedy Search in equation 9, we see that Simulated Annealing and Greedy Search are broadly similar. They are both polynomial time algorithms which when compared to Brute force explains why they can find solutions in a much shorter time.

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 $n_R = lengthofV$.
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 4: 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.

References

[1] Wikipedia, “Voting at the eurovision song contest — wikipedia, the free encyclopedia,” 2016. [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.

[3] L. Spierdijk and M. Vellekoop, “Geography, culture, and religion: Explaining the bias in eurovision song contest voting,” February 2006. [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.

[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.

[6] EBU, “’good evening copenhagen’ - voting order revealed,” 10 May 2014. [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. [Online; accessed 18-April-2017].

[9] Wikipedia, “Permutation — wikipedia, the free encyclopedia,” 2017. [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. [Online; accessed 19-April-2017].

[11] Python-Software-Foundation, “Python built-in functions - min,” 2017. [Online; accessed 22-April-2017].

[12] Python-Software-Foundation, “Python built-in functions - max,” 2017. [Online; accessed 22-April-2017].

[13] Python-Software-Foundation, “Python built-in functions - sorted,” 2017. [Online; accessed 22-April-2017].

[14] Python-Software-Foundation, “Python built-in functions - len,” 2017. [Online; accessed 22-April-2017].

[15] Python-Software-Foundation, “Python random library - sample,” 2017. [Online; accessed 22-April-2017].

[16] Python-Software-Foundation, “Python itertools library,” 2017. [Online; accessed 22-April-2017].

[17] Python-Software-Foundation, “Python itertools library,” 2017. [Online; accessed 22-April-2017].