Final Year Project Interim Report

Alfie Jones

December 1, 2021

Contents

1 Abstract

A number of methods to calculate the relative power of players in yes-no voting systems exist, all of which use some variation of winning coalitions to determine the relative power of each player. Yet all fail to take into account the likelihood of those coalitions actually forming, resulting in simplified scenarios that lead to inaccuracies compared to the complex world of real life voting systems and political power.

The motivation of this project is to establish a new method for calculating the relative power of each player in a yes-no voting system, using probability to determine the likelihood of minimal winning coalitions existing. Building on the four major power indices already in the field, primarily the Deegan-Packel index of power. This project aims to find a way to implement elements from graph theory and historical political data to generate realistic probabilities of each minimal winning coalition in any given yes-no voting system existing. These elements include the cutting of complete weighted graphs and the conductance of the resulting partitions from those graphs.

Once a formal mathematical model is in place for determining the aforementioned probabilities the project will move to implementing the probabilities within a more conventional power index model.

The second part on this project will see the model developed into a program capable of being applied to any n-player yes-no voting system provided necessary information is given.

In this interim report three separate reports are detailed. They cover the three main areas of research and development that have been undertaken during the first stages of this project. The first contains information relating to the probability and graph theory concepts that have been implemented into the algorithm. The second is a deep dive into some power indices that were used as inspiration and the way in which the index will be calculated in the new algorithm. The final report is a walk through and description of the Python implementation written.

2 CEP Report

2.1 Introduction

The primary start point for my new power index is to build off the work done by J. Deegan and E.W. Packel in 1979. More specifically, I will use the work they did on Minimal Winning Coalitions[5] as inspiration. From there, I shall add on new calculations in order to generate a more realistic approach to generating relative power indices. In this report, I shall look at the concepts and calculations that I shall be implementing and evaluating the positives and negatives of such a system. Before doing so, however, we must first define some prerequisite concepts.

2.1.1 Coalition Existence Probability

CEP stands for Coalition Existence Probability. This is the likelihood that two parties will form a coalition with each other and is modelled as a complete weighted graph where each edge is the percentage chance the two nodes connected will join together in a coalition. All edges in a coalition must be multiplied together to find the complete CEP of the coalition.

2.1.2 Conductance

The conductance of a graph G, where $G = (V, E)$, is a measure of how strong a partition of the graph is compared to the other.[1] Typically the partitions are made by a single cut, resulting in two partitions $(S, \bar{S})$.[3] Conductance can also be thought of as a numerical measure of if a graph is bottle-necked or how many it might relatively have.[2] The conductance of a partition is calculated using two properties of the partition. The first we shall call the internal strength X of the partition. The second of which is the external pull Y of the partition. The conductance of G is calculated using the following formula:

$$\psi = \frac{Y}{X + Y}$$

Where

$$Y = \sum_{i \in S, j \in \bar{S}} a_{ij}$$

and

$$X + Y = \min(a(S), a(\bar{S}))$$

Where

$$a(S) = \sum_{i \in S} \sum_{j \in V} a_{ij}$$

$$a(\bar{S}) = \sum_{i \in \bar{S}} \sum_{j \in V} a_{ij}$$

This results in X being two times the sum of all the weights of edges solely within the partition, since by summing the weight of every edge from an element of S to an element of V you will count the edges where i and $j \in S$ twice. Y is then equal to the sum of the weights of all edges that go between S and $\bar{S}$. This leaves $a(S)$ being equal to the sum of all edges that connect the nodes in one of the partitions to any other node in G but counts those solely within the partition twice.

2.1.3 Minimal Winning Coalitions

In order to define a Minimal Winning Coalition, some other more basic concepts should be defined first. Firstly, the set we are calculating over shall be defined as the set P where $\{p_1, p_2, \dots p_n\}$ are the number of votes each party in a system received. These shall all be positive integers. Secondly a target T shall be defined as the number of votes needed for a party or coalition to form a government. This then means that a winning coalition C where $w \in C$ and $C \subseteq P$ is defined as:

$$\sum_{i \in C}^{n} w_i \geq T$$

From these definitions we can then say that an MWC C exists when the removal of any w results in the sum of the remaining elements being less than T

$$\sum_{i \in C}^{n} w_i - \min(w_i) < T$$

2.2 Modelling the System

The fundamental idea behind this system is that the political system of a region or entity can be modelled as a graph. More specifically, the chances of each party forming a coalition with the other parties can be modelled as a graph. These chances are the CEPs defined above. Since we need the CEPs for each party and every other party in the system, we end up with a complete graph. This is a graph in which all nodes are connected to all other nodes in said graph. Another definition is that the degree of every node in the graph is $n - 1$ where n is the number of nodes in the graph. An arbitrary complete weighted graph of four nodes may look something like Figure 1

It is from these graphs that we can then compute the conductance of each MWC.

2.3 The Data Required

Now we are aware of how we are modelling this problem, the next thing we should know is what data we need to create a working model and successfully

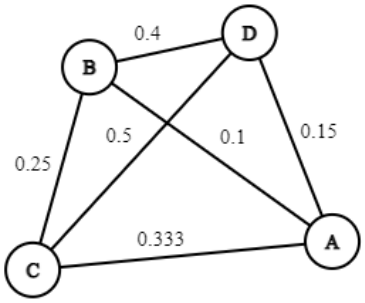

Figure 1: A complete weighted graph with four nodes.

compute the conductance of each MWC. Firstly, and most critically for the graphical model, we need to know how many parties are in a system and the likelihoods that each party will join a coalition with each of the others. For real world examples I intend to use historical trends for this. In fictional examples, these probabilities will be arbitrary. The second piece of data needed relates to the votes themselves. This is the set P discussed earlier. Once we have these two pieces of data we are then able to compute the conductance and CEPs

2.4 An Example

To begin we must first find all MWCs in the voting system, these will be our partitions we create in the graph later. Once the MWCs are found, the next step is to generate the complete weighted graph for the voting system. After the graph is generated we can compute the conductance, as defined above, for each MWC. In addition to this we must also multiply all the edges used in calculating X together to find the CEP of the corresponding MWC Lets follow these steps using an example:

$$[12|4, 6, 3, 9, 2];$$

	a	b	c	d	e
a	\	0.4	0.2	0.2	0.7
b		\	0.3	0.5	0.3
c			\	0.5	0.7
d				\	0.1
e					\

The MWCs in this example are:

$$M = \{\langle a, d\rangle, \langle b, d\rangle, \langle c, d\rangle, \langle a, b, c\rangle, \langle a, b, e, \rangle\}$$

Figure 2 displays the graph that is produced from these MWCs.

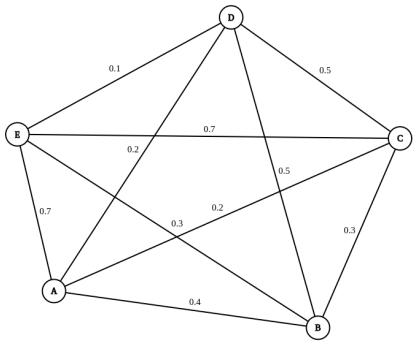

Figure 2: The graph produced from the set of MWCs M.

For each MWC, the conductance and CEP are as follows:

$$\psi_{m_1} = \frac{0.1 + 0.5 + 0.5 + 0.4 + 0.2 + 0.7}{2(0.2) + 0.1 + 0.5 + 0.5 + 0.4 + 0.2 + 0.7} = \frac{6}{7};$$

$$CEP_{m_1} = 0.2$$

$$\psi_{m_2} = \frac{9}{14};$$

$$CEP_{m_2} = 0.5$$

$$\psi_{m_3} = \frac{2}{3};$$

$$CEP_{m_3} = 0.5$$

$$\psi_{m_4} = \frac{29}{47};$$

$$CEP_{m_4} = 0.2 \times 0.3 \times 0.4 = 0.024$$

$$\psi_{m_5} = \frac{15}{43};$$

$$CEP_{m_5} = 0.084$$

Now we have all the results we need to continue on to calculate the relative power of each coalition, however that is outside the scope of this report, and will be detailed at a later time.

2.5 Evaluation

When creating a new system or idea, it is important to evaluate its merits and drawbacks compared to other systems within the field. This allows for a fair analysis of the system which can be used to determine its usability and functionality.

2.5.1 Positives

Firstly, let's look at the pros of the method that I have detailed in this report. Due to the use of MWCs the number of coalitions and therefore the number of calculations required are inherently lower than in more prominent power indices like Shapley-Shubik or Banzhaf. [7][8] For example, the running time of the Shapley-Shubik Power Index is $\Theta(n!)$ [7] while the worst case for this system is $O(2^n)$. This is due to the fact that, in a worst case example, all parties would have the same number of votes and the target would be a simple majority.

$$[30|10, 10, 10, 10, 10, 10]$$

Figure 3: An example of a worst case scenario.

In this particular example, the number of MWCs that need to be calculated over is 20. In general, in the case where all parties receive the same amount of votes, and the target is the sum of half the votes of all parties, then the number of MWCs that exist is:

$$\binom{n}{\frac{n}{2}}$$

Where n is the number of parties in the system. Since this number grows at an exponential rate, we can say that the worst case running time is as fast, if not faster, than the two most commonly used power indices.
Another benefit of this system is that it provides a more realistic approach to modelling the power of political parties. It does this by doing what no other power index does: taking the likelihood of coalitions actually existing into account. The CEPs calculated within the system provide a more detailed analysis of power within a voting system, as parties that are more willing to join with other parties are somewhat more likely to be in the final winning coalition. In other words, parties able to put aside their differences are more likely to join forces and win the vote. This is what, to some extent, my system captures.

2.5.2 Negatives

The largest issue with this system is the amount of data needed. There needs to be a sufficient amount of data in order to add weights to the edges of the graph. While the graph should, in theory, be a complete graph but the weights can still be zero. Issues will occur however if most of the weightings are zero, resulting

in an inaccurate power index. While these probabilities are not incalculable, they can be hard for the average person to find, and this should be taken into account when considering the usability of this method.

2.6 Conclusion

It is my belief that the system I intend to use has a lot of great benefits that drastically improve the realism of power indices. While the challenge of data collection is one of non-insignificant proportions, it seems to be one that is outweighed by the advantages of running time and real world accuracy. As such I will be moving forward with my new power index, with the intent to utilise the systems and ideas discussed in this report, hopefully with great success.

3 Power Index Report

3.1 Introduction

Typically we see power indices calculate the power value of a party, within a voting system, and then normalise those values to reach a final relative power ranking. This is they way almost all common power indices work[5][7][8] and makes complete sense within the context of the individual index. The primary difference between each of these indices is how they go about with the initial calculations on each parties power value. We shall look at some examples later.

This report details the advancements made since the Coalition Existence Probability Report earlier this month. The mathematics detailed in that report stand alone, however, the mathematics that I shall go through in this one are built on top of the equations of conductance and coalition existence probability from the previous report.
In this report I shall look at what the continuation of the overall algorithm looks like and give a full working run through of the new algorithm, but beforehand let's look at two notable examples in the field already that demonstrate calculating a standard power value.

3.2 Examples in the Field

3.2.1 Deegan-Packel (1978)

The Deegan-Packel power index is a lesser known power index but that does not define its utility. It uses minimal winning coalitions when computing the index of a voting system[5], which, of the major power indices, makes it unique in that regard. I will explain how it works by working through and example. Let's take the following voting system:

$$[20|12, 10, 14, 6]$$

Here, 20 is the number of votes needed by a coalition to win. Party A has 12 votes, B has 10 votes and so on. As discussed in my previous report, a minimal winning coalition is one where the removal of any party results in the winning coalition becoming a losing one. With that in mind, all MWCs are as follows;

$$\langle A, B\rangle, \langle B, C\rangle, \langle C, D\rangle, \langle A, C\rangle$$

From here we can then calculate power value of each party by summing the reciprocal of the number of parties in that coalition for all coalitions that party appears in. Taking A for example, there are two parties in both coalitions that it is in. This results in $\frac{1}{2} + \frac{1}{2} = 1$. For B this is also the case. For party D, it only appears in one coalition, made up of two parties, giving it a power value of $\frac{1}{2}$. Finally, C has a power value of $\frac{3}{2}$.

3.2.2 Shapley-Shubik (1954)

The Shapley-Shubik power index is perhaps the most popular and well known power index in use today. It is unique in that it makes use of ordered coalitions[8] to calculate the power value of each party. This means that when using this particular algorithm, you need all possible permutations of coalitions from the parties in the system. Let's look at the following example:

$$[8|3, 6, 3]$$

This is our voting system. Party A has 3 votes, party B has 6 votes and party C has 3 votes. The target each coalition needs to reach is 8. When using this index, order matters. That is to say $\langle A, B\rangle$ is not the same as $\langle B, A\rangle$. This is because Shapley-Shubik makes use of what are called *Pivotal Parties*. Every time a party is the pivotal party, (a party is the pivotal party when adding that party to a coalition results in the coalition reaching the target) it gains a point, or a tally. It is also important to remember that Shapley-Shubik only uses maximal, or grand coalitions. These are where all parties in each coalition. Given all the possible coalitions are:

$$\langle A, B, C\rangle, \langle A, C, B\rangle, \langle B, A, C\rangle, \langle B, C, A\rangle, \langle C, A, B\rangle, \langle C, B, A\rangle$$

We get the following tally chart of times each party was the pivotal party in a coalition.

Party	Tally
A	1
B	4
C	1

Now, we turn these tallies into fraction by dividing each tally by the number of coalitions used to get the power value of each party.

Party	Power Value
A	0.16666
B	0.66666
C	0.16666

3.2.3 Normalising the Power Values

We have seen how two power indices work but have not touched on the final part of the Deegan-Packel power index. Although, sneakily, we did it in the Shapley-Shubik index. This final part is the normalisation of the power values. This is what gives us the relative power of a relative power index. Normalising is as follows:

$$\frac{P_i}{\sum_{k=1}^{n} P_k}$$

Where P_i is the parties power value you want normalised and n is the total number of parties in the voting system. Effectively, take the party you want to normalise and divide its power value by the sum of all power values in the system. I will leave it as an exercise for the reader to normalise the example given in the Deegan-Packel section, and also, to explain why normalisation has already occurred in the Shapley-Shubik example.

3.3 How the Power Index Works

In continuation of the previous report, discussing the conductance ψ and the CEP of a graph, we now move forward with using those two values to calculate the power value and then the power index of each party. First, a reminder of what the conductance of a graph means. The conductance ψ of a graph is the proportion the external pull of a partition is of the sum of the external and internal pull of said partition. That is to say, the higher ψ is, the more external weight there is pulling the coalition apart. With that in mind, the first calculation to make once you have ψ and CEP for each minimal winning coalition is to divide the latter by the former.

$$\Lambda = \frac{CEP}{\psi}$$

Once Λ has been calculated for each MWC, a tally must be made of each time a party appears in an MWC. Second, you must take each tally and find the fraction of all tallies each party has. The Λ of each party is the largest Λ from an MWC that contains that party.

$$\Lambda_i = \frac{Tally_i}{\sum_{k=1}^{n} Tally_k}$$

After the power value of each party has been found, normalise each of them to find the final power index.

3.4 A Complete Example

Let's walk through, from start to end, an example. Let's use the voting system:

$$[15|3, 4, 7, 9, 6]$$

The likelihood of each party joining another in a coalition is:

	a	b	c	d	e
a		0.7	0.4	0.1	0.2
b			0.6	0.2	0.1
c				0.6	0.8
d					0.7
e					

The first step is to find all the minimal winning coalitions. In this case they are as follows:

$$\langle D, E\rangle, \langle C, D\rangle, \langle B, C, E\rangle, \langle A, B, D\rangle, \langle A, C, E\rangle$$

The graph from the matrix is presented in Figure 4

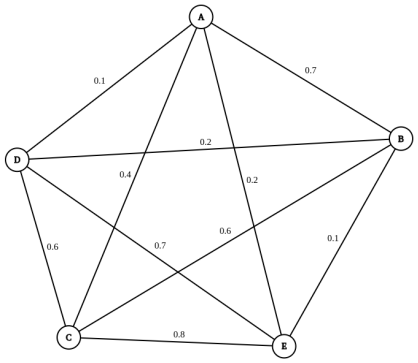

Figure 4: The likelihood matrix modelled as a complete weighted graph.

From the graph in Figure 4 we can calculate the conductance ψ and the CEP of each MWC.

$$\psi_1 = \frac{2}{3.4}, CEP_1 = 0.7$$
$$\psi_2 = \frac{2.8}{4}, CEP_2 = 0.6$$
$$\psi_3 = \frac{2.8}{5.8}, CEP_3 = 0.048$$

$$\psi_4 = \frac{2.6}{4.6}, CEP_4 = 0.014$$

$$\psi_5 = \frac{2.8}{5.6}, CEP_5 = 0.064$$

Now we must calculate Λ by dividing CEP by ψ

$$\Lambda_1 = 1.1900$$

$$\Lambda_2 = 0.8571$$

$$\Lambda_3 = 0.0994$$

$$\Lambda_4 = 0.2477$$

$$\Lambda_5 = 0.1280$$

Once Λ has been calculated for each MWC, we need to make a tally of each time a party appears in an MWC. That tally chart looks like this:

Party	Tally	Fraction
A	2	$\frac{2}{13}$
B	2	$\frac{2}{13}$
C	3	$\frac{3}{13}$
D	3	$\frac{3}{13}$
E	3	$\frac{3}{13}$

We now have everything we need to find the final power index of each party, we just need to select the correct Λ for each of them. The correct Λ will be the largest Λ from an MWC that a party was a part of. For A, that will be Λ_4. For B, it will be Λ_4. C will use Λ_2, and D and E will use Λ_1. Adding this on to our table we get:

Party	Tally	Fraction	Λ
A	2	$\frac{2}{13}$	0.2477
B	2	$\frac{2}{13}$	0.2477
C	3	$\frac{3}{13}$	0.8571
D	3	$\frac{3}{13}$	1.1900
E	3	$\frac{3}{13}$	1.1900

Multiplying the fractions with the respective Λ we get power values of

$0.0381, 0.0381, 0.1978, 0.2746$ and 0.2746 for A, B, C, D and E respectively

Normalising these values gives us our final power index which is displayed in figure 5

Party	Power Index
A	0.0463
B	0.0463
C	0.2403
D	0.3336
E	0.3336

Figure 5: The final power index of the example.

3.5 Conclusion

To conclude, the complete algorithm takes the work done by Deegan and Packel and adds the idea that coalitions should be given some form of probability of all the parties in them working together. It is this probability in combination with the alternatives available to each coalition that provides a weighting to be applied to the Deegan-Packel index. Once the weighting is applied and the values normalised, it provides the final power index for the voting system.

4 FYP Report 3: Revenge of the Code

4.1 Introduction

In this report is a detailed explanation of how the program I wrote runs, the decisions made and an overall evaluation of the program in its state ahead of the interim presentations. I will go through each of the non-testing files in turn in order to, hopefully, fully document the functionality of the program.

4.2 I hate running

4.2.1 main.py

Firstly, we shall discuss the main.py file. This is the file to be run in order to start the program. It contains calls to each of the necessary files and functions in order for the program to run. The full file is displayed below:

```
from algo_work.program_files import all_the_lambda,
find_mwc, graph_theory_section, tallying_parties

statement = "Enter number of parties: "
number_of_parties = int(input(statement))

parties = []


def run():
    for i in range(number_of_parties):
        statement = "Enter the name of party:", i + 1
        name_of_party = str(input(statement))
        statement = "Enter number of votes the party has: "
        number_of_votes = int(input(statement))
        parties.append((name_of_party, number_of_votes))
    print(parties)
    statement = "Enter the number of votes needed to win: "
    target = int(input(statement))
    prob_matrix = graph_theory_section.set_up_prob_matrix(parties)
    list_of_mwcs = find_mwc.generate_coalitions(parties, target)
    conductances = graph_theory_section.conductance(parties, list_of_mwcs, prob_matrix)
    ceps = graph_theory_section.cep(parties, list_of_mwcs, prob_matrix)
    tally = tallying_parties.tally(list_of_mwcs)
    list_of_lambdas = all_the_lambda.assign_Lambda(parties, conductances, ceps)
    unnormalised_scores = (all_the_lambda.score_lambda_fun_times(tally, list_of_lambdas))
    all_the_lambda.normalise(unnormalised_scores)


run()
```

4.2.2 A Line By Line Look

Starting with the first line

```
from algo_work.program_files import all_the_lambda,
find_mwc, graph_theory_section, tallying_parties
```

These are all the other files in the program being imported to the main.py file so that they can be run. We will see what each of these file do later. For now, we will move on to lines 3 - 6. Lines 3 - 6 are as follows:

```
statement = "Enter number of parties: "
number_of_parties = int(input(statement))

parties = []
```

This is the first thing the user will see. The console will print out statement and expect the user to input an integer to be used as the number of parties. The reason for this is so that when it comes to naming and assigning the number of votes to a party, the for loop that is used will know how many times to loop. The array parties is also initialised to be empty and this will fill up as we append the parties to it using user inputed data from the next few lines. Speaking of which:

```
def run():
    for i in range(number_of_parties):
        statement = "Enter the name of party:", i + 1
        name_of_party = str(input(statement))
        statement = "Enter number of votes the party has: "
        number_of_votes = int(input(statement))
        parties.append((name_of_party, number_of_votes))
    print(parties)
```

Lines 9 - 16 is where the user is to input the name of each party and the votes it received. The string of the name and the integer of the votes are then appended to the parties array as a tuple. The reason for using a tuple is that tuples are immutable and the data the program is given should also be immutable. Any change to the names or the votes of each party during the running of the program will result in the final results being incorrect. And that, dear reader, would be a travesty. In line 10, we can see the reason for asking the user for the number of parties at the start of the program. It is so the for loop repeats that many times. Once the for loop has been completed then the list of parties is printed out on to the console.

I would argue that lines 17 and 18 are the most important in the function; perhaps even the whole program. This is because, as seen below, they acquire the target number of votes needed in order for a coalition to be winning. Without this number it would not be possible to even begin the algorithm. Again, this number is naturally a user input.

```
statement = "Enter the number of votes needed to win: "
target = int(input(statement))
```

It is at this point in the file that all the crazy computer magic really starts to happen.

```
    prob_matrix = graph_theory_section.set_up_prob_matrix(parties)
    list_of_mwcs = find_mwc.generate_coalitions(parties, target)
    conductances = graph_theory_section.conductance(parties, list_of_mwcs, prob_matrix)
    ceps = graph_theory_section.cep(parties, list_of_mwcs, prob_matrix)
    tally = tallying_parties.tally(list_of_mwcs)
    list_of_lambdas = all_the_lambda.assign_Lambda(parties, conductances, ceps)
    unnormalised_scores = (all_the_lambda.score_lambda_fun_times(tally, list_of_lambdas))
    all_the_lambda.normalise(unnormalised_scores)


run()
```

Line 19 calls the set_up_prob_matrix function that we will explore in depth later. All we need to know for now is that it is vital for calculating the conductance and Coalition Existence Probability (CEP) of each Minimal Winning Coalition (MWC). Line 20 uses the parties array we collected and the target to find each of the MWCs that are in the voting system given by the users inputs. Lines 21 and 22 then calculate the conductance and the CEPs of each MWC which are then used in line 24 to calculate the lambda value for each MWC. Line 23 uses the list of MWCs that was found in line 20 to find the tally score of each party. Lines 24 and 25 are then used to compute the final power index. Line 30 calls the run function to begin the whole process in the first place. Magic!

4.3 MWCs are good at hide and seek

4.3.1 find_mwc.py

In this section we will discuss the workings of the file find _mwc.py. This piece of code goes through every combination of votes and finds all those that are both winning and minimal. Firstly, let's take a quick look at it in its entirety.

```
from itertools import combinations


def generate_coalitions(lst, target):
    """
    Finds all combinations needed from parties

    Takes the list of parties and cycles through,
    creating a list of all winning combinations.

    """
```

```
    list_of_mwc = []
    index = 0
    for i in range(1, len(lst)+1):
        combination = list(combinations(lst, i))
        # Finds all combinations of size i from lst
        # list function coverts into list format,
        # result is a list of all combinations of size i
        for j in range(len(combination)):
            coalition_to_test = list(combination[j])
            # Each combination in list is extracted to then be tested
            total = 0
            for k in range(len(coalition_to_test)):
                total += coalition_to_test[k][-1]
                # Total is the sum of the votes in each coalition
            if minimal_test(coalition_to_test, total, target):
                list_of_mwc.insert(index, coalition_to_test)
    return list_of_mwc


def minimal_test(combination, total, target):
    """
    Takes combination and checks if it is minimal

    The each party's votes are removed from the total one at a time
    and if for all parties, each time that happens the total is
    smaller than the target, the combination is minimal.
    """
    count = 0
    for i in range(len(combination)):
        check = total - combination[i][-1]
        if check < target:
            count += 1
    if count == len(combination) and total >= target:
        return True
    return False
```

There's quite a bit to unpack here so we'll jump straight into things.

4.3.2 A Line by Line Look 2: The Electric Boogaloo

Here, we start with the importing of combinations from the itertools library. This is because it is a very useful library for finding all the combinations of a group, given a certain size. We will use this specifically in line 15. The rest of the code is the function generate_coalitions, which takes an array (the list of parties) and an integer (the target needed to win). What happens first is we initialise an empty array to which we shall add our MWCs and an integer called

index which is originally set to zero.
Next are the three nested for loops. Let's take these one at a time. Firstly the first for loop:

```
for i in range(1, len(lst)+1):
        combination = list(combinations(lst, i))
```

There are three things going on here. The first is the loop itself. The variable i starts at one and finishes at the length of the list. The reason for the end of the range being one greater than the length of the list is that the range function in python goes from between the start value (in this case, one) to one less than the end value, and since we need to include the full length of the list as that is a valid coalition to check, we need to add one in order to ensure it is included. The second thing we should focus our attention on is the combinations function. This is what we imported from itertools earlier. The combinations function takes a list of, in our case, MWCs and finds all non-repeating combinations of a given size. This is why we need the for loop, because we need to check through every possible combination of parties to find the MWCs that we need to complete the algorithm. This function was chosen over the permutations function because it only take into account non-repeating combinations. I.e: C, D, F is the same as F, C, D so the repeating combination should be ignored.
Since combinations returns the combinations in a tuple of tuples the list function is required to turn it into a list so it can be manipulated. Specifically so we can use its length in the next for loop. Which, funnily enough, looks like this

```
for j in range(len(combination)):
            coalition_to_test = list(combination[j])
            # Each combination in list is extracted to then be tested
            total = 0
```

This for loop is very simple and doesn't do anything we haven't seen already, so I'll be quick. coalition_to_test is the name we give to a single index in the combinations array. For each iteration of the outer most for loop (the one that uses i) the length of coalition_to_test increases by one. Hopefully this is fairly clear. We take that index, set the total to zero and in the next for loop, find the sum of all the votes of that coalition. The reason for not just using the sum function is that the parties are a tuple and you'll get all sorts of errors if you try. Trust me, I'm a doctor.

The rest of the code in this function is as follows:

```
            for k in range(len(coalition_to_test)):
                total += coalition_to_test[k][-1]
                # Total is the sum of the votes in each coalition
            if minimal_test(coalition_to_test, total, target):
                list_of_mwc.insert(index, coalition_to_test)
    return list_of_mwc
```

The if statement in this snippet of code checks to see if the coalition currently being tested is a MWC. If it is then the coalition is added to the list of MWCs,

if not then the outer most for loop increments by one and the process starts again.

```
def minimal_test(combination, total, target):
    """
    Takes combination and checks if it is minimal

    The each party's votes are removed from the total one at a time
    and if for all parties, each time that happens the total is
    smaller than the target, the combination is minimal.
    """
    count = 0
    for i in range(len(combination)):
        check = total - combination[i][-1]
        if check < target:
            count += 1
    if count == len(combination) and total >= target:
        return True
    return False
```

Now that is covered, let's talk about how we actually test a coalition to see if it's winning and minimal.

As we can see, this function requires a coalition of parties, the coalitions total number of votes and the target needed in order to be considered winning. Firstly, the integer count is initialised to zero. Count will be the determining factor in finding out if a coalition is minimal or not. The for loop behaves as follows: Each party, more specifically, its assigned number of votes is subtracted from the total in turn and then the new total, check, is compared against the target. If check is smaller than the target, count increases by 1. Once the for loop is complete, if count is the same as the number of parties in the coalition and the original total is greater than the target, then the coalition is an MWC. This is because if the sum of the votes is greater than the target the coalition is winning. For a coalition to be minimal no party can be removed from the coalition and the coalition still be winning. If the coalition achieves both these goals the function returns true, otherwise it returns false.

4.4 Drawing? This isn't art school!

Run through of the file How the code relates to the 1st report

4.4.1 graph_theory_section.py

To begin, recall the first report I wrote on this algorithm. It detailed the mathematics behind the concepts of conductance and CEP. In this section I will go through how this maths and graph theory work was translated into working code. As always, we will start with the code in its entirety.

```
import numpy as np
import itertools as it


def set_up_prob_matrix(parties):
    """
    Method to create the matrix of probabilities of parties joining
    coalitions with others.
    """
    prob_matrix = np.zeros((len(parties), len(parties)))
    # Creates an nxn matrix of all 0
    for i in range(len(parties)):
        for j in range(len(parties)):
            if i < j:
                statement = "Enter probability for (", parties[i][0], parties[j][0], ")"
                prob = input(statement)
                # User input to give probability to appropriate indices in matrix
                prob_matrix[i][j] = prob
                prob_matrix[j][i] = prob
                # Makes matrix symmetrical
    return prob_matrix


def conductance(parties, mwcs, prob_matrix):
    """
    Function calculates the Conductance of the probability matrix.

    Calculates the internal strength and external pull of the mwc
    then divides the latter by the sum of both.
    """
    conductances = []
    for mwc in mwcs:

        # Internal Strength
        edges = list(it.combinations(mwc, 2))
        # Creates all edge pairings of the mwc in the graph
        everything_in = 0
        everything_out = 0
        for edge in edges:
            row = parties.index(edge[0])
            # Since prob matrix and parties have the same ordering of the parties...
            # ...this lines up the rows and columns.
            column = parties.index(edge[1])
            everything_in += prob_matrix[row][column]

        # External Pull
```

```
        for party in mwc:
            row = parties.index(party)
            for i in range(len(parties)):
                if prob_matrix[row][i] != 0 and parties[i] not in mwc:
                # Checks if the column currently being looked at
                # is associated with a party not in the mwc and is > 0
                    everything_out += prob_matrix[row][i]

        # Final conductance
        psi = (everything_out / (everything_out + (2 * everything_in)))
        conductances.append((mwc, psi))
    return conductances


def cep(parties, mwcs, prob_matrix):
    """
    Finds the Coalition Existence Probability of each mwc.

    Does this in the same way as internal strength in conductance
    function but multiplies instead of adds.
    """
    ceps = []
    for mwc in mwcs:
        edges = list(it.combinations(mwc, 2))
        # Creates all edge pairings of the mwc in the graph
        cep = 1
        for edge in edges:
            row = parties.index(edge[0])
            # Since prob matrix and parties have the same ordering of the parties...
             # ...this lines up the rows and columns.
            column = parties.index(edge[1])
            cep *= prob_matrix[row][column]  # Multiplies instead of adds
        ceps.append((mwc, cep))
    return ceps
```

4.4.2 A Line By Line Look 3: This Time It's Personal

At the start of the file we see the two libraries we've imported. These are itertools and numpy. Itertools, we shall use in a very similar way to the last file. Numpy, we shall use primarily to create the probability matrix which is used to build the completed graph.

```
import numpy as np
import itertools as it
```

```
def set_up_prob_matrix(parties):
    """
    Method to create the matrix of probabilities of parties joining
    coalitions with others.
    """
    prob_matrix = np.zeros((len(parties), len(parties)))
    # Creates an nxn matrix of all 0
    for i in range(len(parties)):
        for j in range(len(parties)):
            if i < j:
                statement = "Enter probability for (", parties[i][0], parties[j][0], ")"
                prob = input(statement)
                # User input to give probability to appropriate indices in matrix
                prob_matrix[i][j] = prob
                prob_matrix[j][i] = prob
                # Makes matrix symmetrical
    return prob_matrix
```

Next we move on to the function set_up_prob_matrix. This takes an argument of the list of parties given to the program by the user. The first thing this function does is generate a square matrix of zeros. This means that every value in it is initialised to zero to begin with. The size of the matrix is, as we can see, a square with sides of the length of the parties array (I.e. The number of parties in the system).
The nested for loops then ask the user for every non-repeating edge weight. This is the probability of those two parties joining in a coalition together. This value is then entered into the matrix twice, so the matrix is symmetrical. This is for the reason that the graph is a completed graph but is undirected. By making the graph symmetrical both those attributes are applied.

In this function, the conductance of each MWC is calculated. This is done by calculating two other values first. The internal strength of the MWC and the external pull on it. Let's start by having a look at the internal strength.

```
def conductance(parties, mwcs, prob_matrix):
    """
    Function calculates the Conductance of the probability matrix.

    Calculates the internal strength and external pull of the mwc
    then divides the latter by the sum of both.
    """
    conductances = []
    for mwc in mwcs:

        # Internal Strength
        edges = list(it.combinations(mwc, 2))
        # Creates all edge pairings of the mwc in the graph
        everything_in = 0
```

```
        everything_out = 0
        for edge in edges:
            row = parties.index(edge[0])
            # Since prob matrix and parties have the same ordering of the parties...
            # ...this lines up the rows and columns.
            column = parties.index(edge[1])
            everything_in += prob_matrix[row][column]
```

The first thing to note is that the function takes 3 arguments. The list of parties we have seen a number of times already; the list of MWCs that will need to have their conductances computed and the probability matrix. This is because we need the values in that matrix to find the conductance and, later on, the CEPs of each MWCs as well. The second thing you will see is that, like in a lot of other functions presented so far, an empty array is initialised. Then the for loops begin.
Naturally the conductance must be calculated for each MWC in the voting system which is why the first for loop is necessary. Moving on to inside the for loop we see the first use of the itertools library in this file. Again it is the combinations function but this time always finding combinations of size two. This is because an edge can only occur between two nodes. Once all these combinations are found, the weighting of each edge solely within the partition is added on to the everything_in value. The row and column variables are there realistically for readability, but I think they deserve a quick explanation. Each edge is made up of two nodes, which in this case, because of the use of the combinations function, are tuples. The first party in the tuple is found within the list of parties and its index becomes the row number to find the correct weighting in the probability matrix. The same can be said for the column variable, but it uses the other party in the edge. Where the row and column meet is the weighting used to add on to the everything_in variable.

The next section is about calculating the external pull on the partition of the graph and finding the final conductance of the MWC. The code is as follows:

```
        # External Pull
        for party in mwc:
            row = parties.index(party)
            for i in range(len(parties)):
                if prob_matrix[row][i] != 0 and parties[i] not in mwc:
                # Checks if the column currently being looked at
                # is associated with a party not in the mwc and is > 0
                    everything_out += prob_matrix[row][i]

        # Final conductance
        psi = (everything_out / (everything_out + (2 * everything_in)))
        conductances.append((mwc, psi))
    return conductances
```

The main thing in this section of code to point out is the if statement. The first for loop just iterates through each party in the MWC. The second is tied

to the if statement, which checks if the party is part of the MWC and has an edge that goes to a party not in the MWC. If that is the case then the weight of that edge is added to the everything_out variable.
In the final few lines the conductance is calculated and then a tuple of the MWC and its conductance is appended to the list initialised at the start of the function to then be returned.

There isn't much more to say that hasn't been covered with regards to the next section of code. This is becasue, as you can see, the cep function is almost virtually identical to the internal strength calculation already discussed

```
def cep(parties, mwcs, prob_matrix):
    """
    Finds the Coalition Existence Probability of each mwc.

    Does this in the same way as internal strength in conductance
    function but multiplies instead of adds.
    """
    ceps = []
    for mwc in mwcs:
        edges = list(it.combinations(mwc, 2))
        # Creates all edge pairings of the mwc in the graph
        cep = 1
        for edge in edges:
            row = parties.index(edge[0])
            # Since prob matrix and parties have the same ordering of the parties...
             # ...this lines up the rows and columns.
            column = parties.index(edge[1])
            cep *= prob_matrix[row][column]  # Multiplies instead of adds
        ceps.append((mwc, cep))
    return ceps
```

The only notable difference here is that instead of adding the weights of the edges together, they are multiplied instead.

4.5 Lines, but not the fun ones

4.5.1 tallying_parties.py

In this section I shall give a walk through of the file that handles all the functionality of computing the tally scores of each party in the voting system. The code in question is below:

```
def tally(mwcs):
    """
    Takes a list of MWCs and counts up how many times a
    party appears across all of them.
    """
```

```
    list_of_lists = [item for t in mwcs for item in t]  # Takes mwcs and
    # converts from a list of list of tuples to a
    # list of tuples
    storage = []
    denominator = len(list_of_lists)
    scores = []

    for i in range(len(list_of_lists)):
        if storage.count(list_of_lists[i]) == 0:  # Checks if the party
        # has already been counted
            storage.append(list_of_lists[i])
            count = list_of_lists.count(list_of_lists[i])
            score_tuple = (list_of_lists[i][0], count/denominator)  # Creates tuple of
            # name of party and percentage of
            # times each party appeared across all mwcs
            scores.append(score_tuple)
    return scores
```

This is the smallest file in terms of number of lines but it is unquestionably vital as the tally score is the first stage to creating the final power index values of each party.

4.5.2 A Line By Line Look 4: A New Code

While the lists storage and and scores, along with the integer denominator are trivial, I would like to direct your attention to the list comprehension that occurs before the initialisation of the aforementioned arrays and variables.

```
list_of_lists = [item for t in mwcs for item in t]
```

This converts the list of lists of tuples that is the list of MWCs into a list of tuples. The reasoning behind this decision is that the tally score for each party is just the percentage each party makes up of the number of parties (or tuples) in the list of MWCs. As such, a list of lists isn't necessary and just a list of tuples will suffice. In addition to this, it also reduces the complexity of the code as fewer indices are needed to access the correct data.

Line 13 and onward is the for loop in which these tally scores are calculated. While iterating through the list of parties that are included in the MWCs (this list contains repetitions of parties, it should be noted), we first check if that party has already been counted. Naturally this will not be the case in the first MWC. The party is then appended into the storage array and then the number of times it appears in the list is counted. That count is then divided by the length of the list of parties to find the percentage. This percentage and the corresponding party are added to the scores list as a tuple.

4.6 This is Sparta!

4.6.1 all_the_lambda.py

Λ is defined as $\frac{CEP}{Conductance}$. This can be seen from the first function in the file.

```
def Lambda(conductance, cep):
    return cep/conductance


def assign_Lambda(parties, list_of_conductances, list_of_ceps):
    """
    Generates and assigns lambda value to each party.

    Finds the conductance and cep of each mwc and uses that to generate
    a value for lambda. From there it applies to a party,
    the largest lambda from a mwc that the party in question is a part of.
    """
    list_of_Lambda = []
    party_lambdas = []
    for i in range(len(list_of_conductances)):
        Lambda_to_use = Lambda(list_of_conductances[i][-1], list_of_ceps[i][-1])
        # Finds the corresponding conductance
        # and cep for each mwc and divides the latter by the former
        list_of_Lambda.append((list_of_conductances[i][0], Lambda_to_use))
        # Adds the mwc and its lambda value as a
        # tuple to the list
    for party in parties:
        best_lambda = 0
        for i in range(len(list_of_Lambda)):
            if party in list_of_Lambda[i][0] and list_of_Lambda[i][-1] > best_lambda:
                # Checks if the party is in the
                # mwc and that the current value of lambda is
                # larger than the previous largest
                best_lambda = list_of_Lambda[i][-1]
                party_lambda = list_of_Lambda[i][-1]
        party_lambdas.append((party, party_lambda))
    return party_lambdas


def score_lambda_fun_times(scores, lambdas):
    """
    Function to multiply lambda values and tally of each party together.
    """
    final_scores = []
    for i in range(len(scores)):
        final_scores.append((scores[i][0], scores[i][-1]*lambdas[i][-1]))
```

```
        # Multiplies the tally from
        # tallying_parties.py by the lambda for each party
    return final_scores


def normalise(scores):
    """
    Normalises the scores so that they are relative to each other.
    """
    final_scores = []
    denominator = 0
    for party in scores:
        denominator += party[-1]  # denominator is the sum of all scores from parties
    for i in range(len(scores)):
        final_scores.append((scores[i][0], scores[i][-1]/denominator))
        # Normalisation step
    print("The final results are:")
    print(final_scores)
    return final_scores
```

4.6.2 A Line By Line Look 5: When Aliens Attack

The function assign_Lambda is as follows:

```
def assign_Lambda(parties, list_of_conductances, list_of_ceps):
    """
    Generates and assigns lambda value to each party.

    Finds the conductance and cep of each mwc and uses that to generate
    a value for lambda. From there it applies to a party,
    the largest lambda from a mwc that the party in question is a part of.
    """
    list_of_Lambda = []
    party_lambdas = []
    for i in range(len(list_of_conductances)):
        Lambda_to_use = Lambda(list_of_conductances[i][-1], list_of_ceps[i][-1])
        # Finds the corresponding conductance
        # and cep for each mwc and divides the latter by the former
        list_of_Lambda.append((list_of_conductances[i][0], Lambda_to_use))
        # Adds the mwc and its lambda value as a
        # tuple to the list
    for party in parties:
        best_lambda = 0
        for i in range(len(list_of_Lambda)):
            if party in list_of_Lambda[i][0] and list_of_Lambda[i][-1] > best_lambda:
                # Checks if the party is in the
```

```
                # mwc and that the current value of lambda is
                # larger than the previous largest
                best_lambda = list_of_Lambda[i][-1]
                party_lambda = list_of_Lambda[i][-1]
        party_lambdas.append((party, party_lambda))
    return party_lambdas
```

As you can see, the first for loop calculates the Λ value for each MWC and appends it to the list. The second for loop then goes through each party in the voting system and finds the MWC the party is in that contains the largest Λ. The party and its relevant Λ value are then, as a tuple once again, appended to the party_lambdas list to then be returned.

This is the second function in the file and is very simple. all it does is it multiplies the score of each party by the Λ value given to the parties by the previous function. These final scores are then returned and sent as arguments to the next function.

In this function the final operation of the entire algorithm takes place. The normalisation. The denominator is the sum of all the scores in the voting system, which is evident by the way it is calculated. Then, for each party in the system, their final score is divided by the sum of the scores and that, along with the party name, is placed in the final list that contains the normalised, relative power index of each party in the given system.

4.7 Never Quite Good Enough

4.7.1 What Was Good

I think that there is a lot to be pleased with about my code. It is simple for a start. There are no incredibly complex pieces of code that need to be heavily documented in order to understand what they do. Only a very basic understanding of the concepts I have used needs to be applied in order to understand the code and this makes it very user friendly. If someone else were to take my code and improve upon it, the first step in doing so would be to understand what I have written and I believe that I have made that as easy as possible.
In addition to that I have made and completed unit tests for each of the functions in the code so I know that they work exactly as intended. While I did not mention the test files in this report because they do not add to the users understanding of the code, which is the aim of this report, I will mention them here as I believe that the program has benefited from being unit tested. Naturally it has meant that the code know works in the way I want it to work. That is, it follows the exact course of the algorithm as if it were done by hand. This allows for a greater understanding of what part each function plays within the program and, as a developer, it allowed for an easier time when writing the code as I could be sure what I was writing in practice was what I had created in theory.

4.7.2 Where Can Improvements Be Made

The most notable areas of improvement in the current state of the program are that of time and space complexity. The inherent nature of the problem means, as mentioned in a previous report the worst case running time of the algorithm is $O(2^n)$, however this is a specific, and realistically improbable case. While I have not done any true testing or further research into the matter, there are points in the program where for loops are nested up to three times. While the length of the for loops are not all the same, I think it would be safe to say that the current implementations running time is roughly around, if not greater than $O(n^3)$.
In a similar vein, the space complexity could be greatly improved since nearly every function requires creating a new list to then be filled. This however is a symptom of using tuples as much as this program does. Tuples are immutable, that is to say the data they contain cannot be changed in anyway and as such, any and all calculations in which the storing of new numbers is required needs a new list to do so. I am happy with the way the use of tuples has allowed me to create this program but I remain interested to see if using other data structures would provide a more space efficient program.

4.7.3 What Is There Still To Do

Currently the program contains no error handling or edge cases. This will be the first thing I set out to rectify next term. Some errors that are in the forefront of my mind and at the top of my priority list include TypeErrors, for example, not using integers for the number of votes a party has. DivideByZero errors are also another. Some edge cases that need to be included are when one party already has enough votes to win. This should return a final score of one for that party and zero for the rest, however, currently this is not the case.
Another thing I would like to implement next term is a rudimentary GUI to improve the user experience although this is currently not a critical priority of mine.

5 References

[1] en.wikipedia.org. 2021. Conductance_(graph) - Wikipedia. [online] Available at:
https://en.wikipedia.org/wiki/Conductance_(graph) [Accessed 6 October 2021].

[2] en.m.wikipedia.org. 2021. Cheeger constant (graph theory) - Wikipedia. [online] Available at: https://en.m.wikipedia.org/wiki/Cheeger_constant_theory(graph_theory) [Accessed 6 October 2021].

[3] en.m.wikipedia.org. 2021. Cut (graph theory) - Wikipedia. [online] Available at:
https://en.m.wikipedia.org/wiki/Cut_(graph_theory) [Accessed 7 October 2021].

[4] Chierichetti, F., Giakkoupis, G., Lattanzi, S. and Panconesi, A., 2018. Rumor Spreading and Conductance. Journal of the ACM, 65(4), pp.1-21.

[5] Deegan, J. and Packel, E., 1978. A new index of power for simple n-person games. International Journal of Game Theory, 7(2).

[6] All graphs generated by: https://csacademy.com/app/graph_editor/.

[7] Banzhaf, J., 1965. Weighted Voting Doesn't Work: A Mathematical Analysis. Rutgers Law Review.

[8] Shapley, L. and Shubik, M., 1954. A Method for Evaluating the Distribution of Power in a Committee System. American Political Science Review, 48(3), pp.787-792.

6 Literature Survey

[1] - Provided useful description and formulae of conductance.
[2] - Provided alternate definition of conductance.
[3] - Was used to define the concept of a cut within graph theory and the partitions created from it.
[4] - Abstract and introduction was used to help formulate understanding of conductance and the application of external pull and internal strength.
[5] - Original paper on Deegan-Packel power index, used as inspiration for the project and provided enormously useful information regarding how the method works and definition of minimal winning coalitions.
[6] - Graphing tool used to make graphs in figures across the report.
[7] - Original paper on Banzhaf power index, used for all information regarding said index.
[8] - Original paper on Shapley-Shubik power index, used for description and process behind the algorithm.

7 Diary Entries

7.1 Week 1

7.1.1 What Have I Done This Week?

Met with supervisor and have come up with idea for project

Further research into the maths behind the PIs, their advantages and disadvantages

Researched the different power indices

Taken some time to consider options in how, at a high level, the power index might work

Started work on the project plan. Completed the abstract and bibliography

7.1.2 What Were The Issues I Faced And How Were They Solved?

Struggled for a while on what exactly to do the project on

Marco suggested implementing the likelihood of coalitions forming into my own PI to create a new, hopefully more realistic/useful PI

Some of the papers and other sources used for research were very in depth and not very useful for introductions to the topics being researched

Further searching with some recommended tools provided more usable sources which can be found in the project plan bibliography.

7.1.3 What Is The Plan For Next Week?

Primary goals include completing the timeline and risk assessment for the project plan, as well as submitting the plan before Friday

Continuing research for the project. This will most likely take the form of graph theory and thinking about how to apply concepts from it to establish a robust method for calculating coalition existence probability

7.2 Week 2

7.2.1 What Have I Done This Week?

Completed project plan. This was the timeline and risk assessment. Both are done and the plan was submitted on Friday. The timeline was especially useful as now I have a clear schedule to work to.

I've been considering how to implement conductance and probabiltites into my power index. Currently the plan is, in real word application, to use historical data to create a simple probability of voalitions forming. This obviously has a number of drawbacks, most notably, incredibly small sample sizes. Once the probabilities are in place the idea is to simply calculate the conductance of each minimal winning coalition when modelled as a complete weighted graph.

7.2.2 What Were The Issues I Faced And How Were They Solved?

At first I found it very difficult to start the timeline as I didnt have much of an idea as to where I wanted the project to head, nor could I visualise any major deliverables. After a long brainstorm however, I was able to break things down into clear managable steps that are hopefully clear in the timeline in the plan

7.2.3 What Is The Plan For Next Week?

Next week my major goal is to write a short report describing and possibly evaluating how I plan to implement the CEP into my project, since this is the major part of the PI.

7.3 Week 3

7.3.1 What Have I Done This Week?

This week I focused my attention to the first deliverable of my project, outside of the project plan. This was a report on how I intend to utilise the concepts of graph conductance and probabilities in my project. In it I described some major concepts of the project and gave an example of how these concepts would work together, then evaluated the pros and cons of the algorithm developed.

7.3.2 What Were The Issues I Faced And How Were They Solved?

My research and understanding of conductance took a while to get a grasp on but after a conversation with my supervisor I got to much better grips with it and now have a fully realised understanding of how the system I have developed works.

7.3.3 What Is The Plan For Next Week?

Next week I will sit down and start work on the second part of the power index. This part will be focusing on taking the calculations realised in this weeks work and developing them into a relative power ranking. I imagine that I will have to normalise the results at some point, as this is how you get the relative part of the relative power rankings.

7.4 Week 4

7.4.1 What Have I Done This Week?

I have developed the second part of the algorithm, the power index part. Essentially this is the part where the parties are tallied up in some way and some form of number is given to them to then be normialsed, giving the final relative power.

What makes mine different to others in the field is the inclusion of a multiplier, which is divised from the CEP and conductance score of each MWC.

7.4.2 What Were The Issues I Faced And How Were They Solved?

Not really any issues this week, just some ideas that didn't work very well.

For example, using the average multiplier for each party. This resulted in some strage behaviours that seemed to be counter intuitive tothe data given

7.4.3 What Is The Plan For Next Week?

Finalise the power index

Write a report about the whole algorithm

7.5 Week 5

7.5.1 What Have I Done This Week?

Written up a report about the second half of the algorithm. It includes background research examples of two other indices. It also has a complete example from start through to the final power index

7.5.2 What Were The Issues I Faced And How Were They Solved?

No issues this week

7.5.3 What Is The Plan For Next Week?

Next week I'm planning to write a piece of code that implements the Deegan-Packel power index

7.6 Week 6

7.6.1 What Have I Done This Week?

I have attempted to create a program that modelled the Deegan-Packel power index

7.6.2 What Were The Issues I Faced And How Were They Solved?

Issues faced included being bad at programming.

While the theory behind it all was sound my implementation had a few flaws in it due to the way I was finding the combinations of parties. I'll come up with a solution soon, I hope.

7.6.3 What Is The Plan For Next Week?

For the next three to four weeks I will start on my main program. I'll need to research TDD in Python and set up a Git repo as well.

7.7 Week 7 and 8

7.7.1 What Have I Done This Week?

I have started work on developing my algorithm into a functioning program. So far it has been relatively straightforward, but I did have a small breaktrhough in how to get the previous weeks program to work correctly and have since implemented that into this new program.

I have also learnt how to use the unittest library in python to help with the TDD of the program

7.7.2 What Were The Issues I Faced And How Were They Solved?

Linking my github repo to Pycharm took a little more effort that I was expecting but it works perfectly so I can't complain too much

7.7.3 What Is The Plan For Next Week?

Continue on writing code and testing it. I'm looking forward to see how future me handles all of the graph theory aspects and how he codes it. It'll probably suck but as long as it works who really cares?

7.8 Week 9

7.8.1 What Have I Done This Week?

Finished off the code to calculate the tallies for each party and completed the testing for that class

Started work on implementing the graph theory. Already have a way to set up the probablity matrix using user input

7.8.2 What Were The Issues I Faced And How Were They Solved?

Noticed a slight flaw in the logic of a bit of code that generated coalitions incorrectly. Fixed this using the itertools library and everything runs smoothly with it

7.8.3 What Is The Plan For Next Week?

Have a fully functioning algorithm

Calculating Conductance might be a little bit challenging but since the rest is pretty trivial I'm optimistic this will be the only challenge

7.9 Week 10

7.9.1 What Have I Done This Week?

I have finished off the code for the algorithm. There is still more to be done but in its current state, is functional.
I have also written up a third report that describes how the program works

7.9.2 What Were The Issues I Faced And How Were They Solved?

This week went fairly smoothly all things considered. There were no real issues to solve, more just small problems that required a bit of thought. Nothing of note.

7.9.3 What Is The Plan For Next Week?

Next week and the week after will focus on finalising the interim reports and presentations.