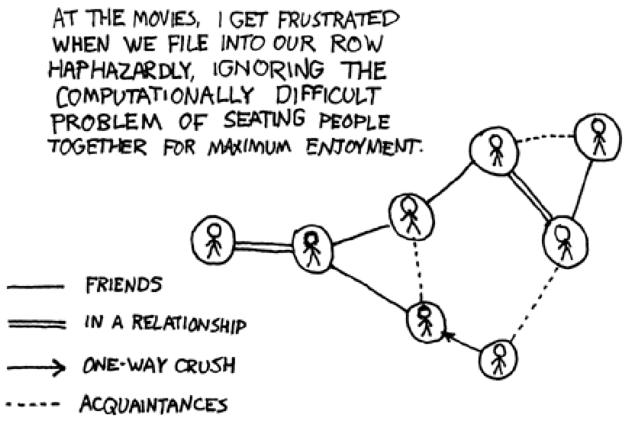
Optimising linear seating arrangements with a first-principles genetic algorithm

Robert Martin

ABSTRACT. We discuss the problem of finding an optimum linear seating arrangement for a small social network, *i.e.* approaching the problem put forth in XKCD comic 173 [Munroe(2006)], as shown here:


AT THE MOVIES, I GET FRUSTRATED WHEN WE FILE INTO OUR ROW HAPHAZARDLY, IGNORING THE COMPUTATIONALLY DIFFICULT PROBLEM OF SEATING PEOPLE TOGETHER FOR MAXIMUM ENJOYMENT.
FRIENDS
IN A RELATIONSHIP
ONE-WAY CRUSH
ACQUAINTANCES

We begin by improving the graphical notation of the network, and then propose a method through which the total enjoyment for a particular seating arrangement can be quantified. We then discuss genetic programming, and implement a first-principles genetic algorithm in python, in order to find an optimal arrangement.

While the method did produce acceptable results, outputting an optimal arrangement for the XKCD network, it was noted that genetic algorithms may not be the best way to find such an arrangement. The results of this investigation may have tangible applications in the organising of social functions such as weddings.

Key words and phrases. social network, optimisation, genetic algorithm.

Contents

1. Introduction

Social groups are known to be remarkably complex affairs; one reason being that there exist different types of relationship between individuals. There are certain situations in which these difficulties become apparent, for example, when deciding on seating arrangements. This has has been comically summarised in one of Randall Munroe's XKCD comics, shown in Figure 1 [Munroe(2006)]:

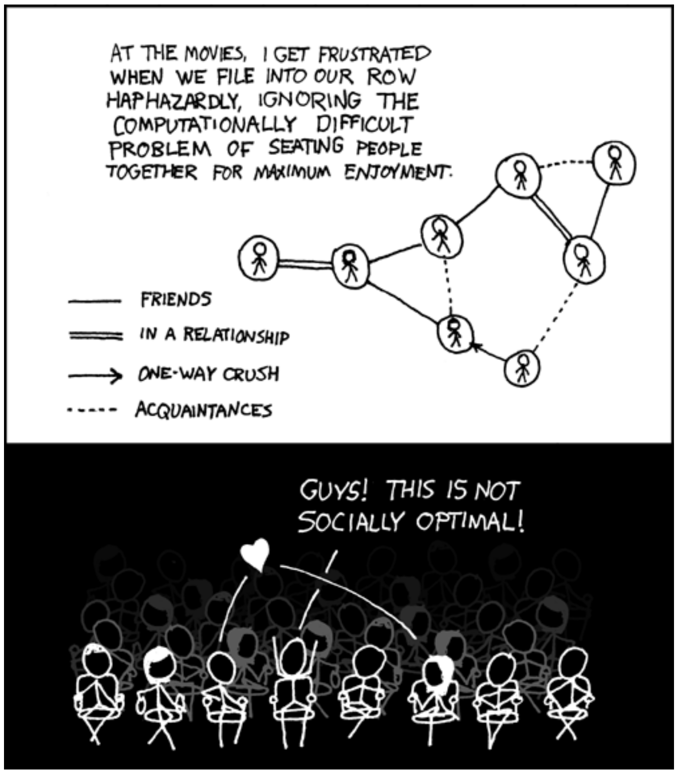

AT THE MOVIES, I GET FRUSTRATED WHEN WE FILE INTO OUR ROW HAPHAZARDLY, IGNORING THE COMPUTATIONALLY DIFFICULT PROBLEM OF SEATING PEOPLE TOGETHER FOR MAXIMUM ENJOYMENT.
FRIENDS
IN A RELATIONSHIP
ONE-WAY CRUSH
ACQUAINTANCES
GUYS! THIS IS NOT SOCIALLY OPTIMAL!

FIGURE 1. XKCD 173, 'Movie Seating'.

The precise problem here is that each individual would prefer to sit with some people rather than others, and that the nonlinear graph has to be 'compressed' onto a linear arrangement. Notwithstanding these facts, we will discuss the ways in which one can optimise linear seating arrangements.

2. Developing the initial model

The key issue in developing this model concerns the manner in which 'maximum enjoyment' will be measured – what does it mean for a seating arrangement to be optimal? We will adopt a utilitarian approach, simply calculating the enjoyment per individual, then summing over all individuals for a particular arrangement. However, we must first decide on a way of describing social networks, which will be amenable to our proposed analysis.

2.1. Examining a social network. Let each individual be denoted by a circular node. We will consider four types of relationships: friends, one-way 'crushes', romantic relationships, and acquaintances. These relationship types will be differentiated pictorially by the style of the connecting line between two individuals, as shown in Figure 2.

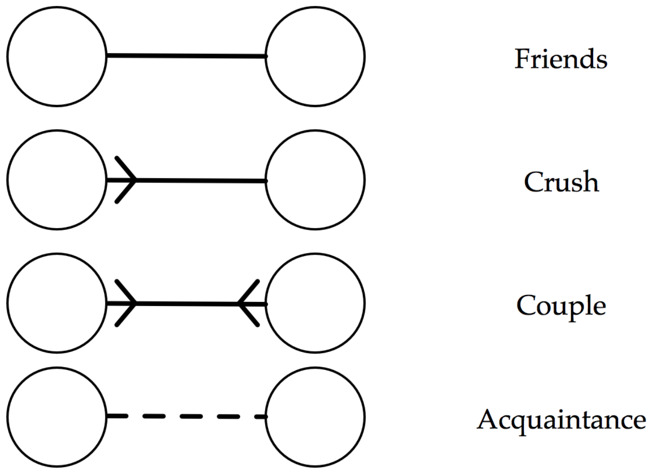


FIGURE 2. The four types of relationship between individuals

This notation was chosen, instead of the system adopted in Figure 1, because it is more simple and elegant. For example, it allows a romantic relationship to be thought of as a 'mutual crush'. Though in real life this may not be the case, it will serve for our purposes. An added benefit of this notation is that we only have to consider the *outgoing* connections from a particular individual. It is clear that there are only four types of outgoing connections: a solid line, an arrowed line, a dotted line, and no line.

In fact, this feature is what will allow us to conduct a quantitative analysis. We assign each type of outgoing arrow a weighting, corresponding to how much enjoyment an individual experiences when sitting with a person of that relation. Here is one proposed weighting:

- Crush (arrowed line): weight = 2
- Friend (solid line): weight = 1
- Acquaintance (dotted line): weight = 0.5
- Stranger (no line): weight = 0

Of course, the actual values are quite arbitrary – what matters is their relative ordering. We assert that any individual's seating preferences follow the above list in descending order. That is to say, given free choice, an individual would enjoy most to sit next to their crush, and least enjoy sitting next to a stranger. Choosing precise weightings would require one to quantify exactly how much more one would prefer to sit with a crush than with a friend, which is no doubt a difficult task.

2.2. An example application. Having discussed how we might describe a social network, we will consider how to measure the total enjoyment for a seating arrangement. We will use the example given in the XKCD comic. Figure 3 shows the graph, reproduced with our new notation. We have labelled each individual with a letter of the alphabet.

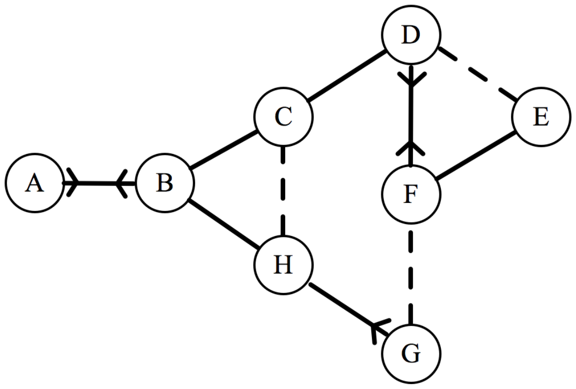


FIGURE 3. The XKCD social network

If this group has to sit down in a line, they will form an arrangement that will be denoted by a sequence of letters. For example, the arrangement $BFGHECDA$ represents the arrangement wherein F sits to the right of B, G to the right of F, and so on. For each of the 8! possible arrangements, we can calculate a total enjoyment, denoted by ϵ_{total}, which is done as follows.

Consider the arrangement $ABCDEFGH$, which is depicted properly in Figure 4. Each individual has a maximum of two connections (we assume that an individual can only be affected by the individuals immediately to their left or right). Each of these connections can assume a value of 2, 1, 0.5, or 0 depending on the type of outgoing connection. For each individual, we add up these values, then we sum over all the individuals in order to get our value of ϵ_{total}.

This is very easy to do with just one arrangement, but bear in mind that we eventually want to find the optimal arrangement, for which we will need a robust algorithm that can be easily applied to any possible arrangement.

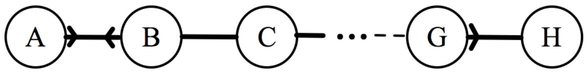


FIGURE 4. Seating arrangement ABCDEFGH

3. Refining the model

As previously stated, we need to develop a method that allows us to efficiently calculate ϵ_{total} for any arrangement.

Let the number of individuals in a network be denoted by n. Each individual will be arbitrarily labelled $1, 2, 3, \ldots, n$, rather than using letters of the alphabet. In order to define an individual's relationships with other members of the group, we will associate a vector x_i for the ith individual in the group, where $x_i \in \mathbb{R}^n$. That is, each vector contains one element for each member in the network (including itself). The mth entry of the vector x_i encodes the enjoyment produced for individual i by individual m. We will denote the mth row of the vector x_i as $[i, m]$.

This notation may perhaps become clearer with an example. We will return to the XKCD network, reproduced in Figure 5 with new notation. For the vector representing the sixth individual's relationships, we have:

$$x_6 = \langle 0, 0, 0, 2, 1, 0, 0.5, 0 \rangle$$

The explanations for the entries are as follows:

- The first three entries (and the final one) are equal to zero, because individual 6 has no relationship with individuals 1, 2, 3, and 8.
- The fourth entry is equal to two, because individual 6 has a crush on individual 4.
- The fifth entry is equal to one, because individual 6 is friends with individual 5.
- The seventh entry is equal to a half, because individual 6 is an acquaintance of individual 7.

The reader may notice that we have neglected to mention the sixth entry. x_6 represents the sixth individual's relationships, so $[6, 6]$ represents the enjoyment produced for the sixth individual by the sixth individual. This entry is fairly useless, so we will set it to zero. That is to say, if $i = m$, $[i, m] = 0$.

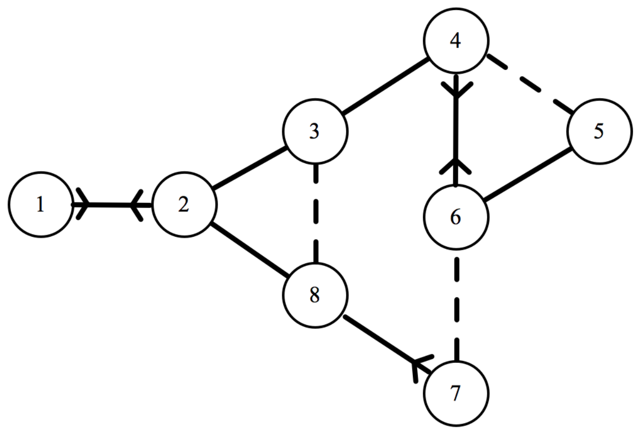


Figure 5. The XKCD social network, labelled with numbers

A vector can be written for all n individuals in our network; incidentally, these vectors are all we need to define the network – the graphical representations are not required. We will now proceed to develop an algorithm that calculates the total enjoyment for a general arrangement.

3.1. The total enjoyment for a general arrangement.

We will denote a general linear arrangement of individuals as

$$u_1, u_2, u_3, \ldots, u_n.$$

In order to calculate ϵ_{total}, we need to calculate the enjoyment for each u term, then sum these up. To clarify, each u term represents one of the individuals in our network (individuals which are labelled $1, 2, 3, \ldots, n$).

Because of our simplification that an individual is only affected by the people immediately adjacent to them, it actually makes sense to group the arrangement into pairs. So we will think of the arrangement as:

$$(u_1, u_2), (u_2, u_3), (u_3, u_4), \ldots, (u_{n-1}, u_n),$$

where each pair of parentheses encloses the relationship between two individuals. It is at this stage that we need to recall the previously-defined vector notation.

The enjoyment produced in individual u_1 by individual u_2 can be found by looking in the u_2th entry of the vector x_{u_1}. The subscript-within-subscript notation is quite confusing, so it is fortunate that we have previously defined an alternative notation, at the end of Section 3. Thus, the u_2th entry of the vector x_{u_1} is simply denoted by $[u_1, u_2]$.

However, note that in the pair (u_1, u_2), individual u_1 also produces some enjoyment in u_2. Thus the total enjoyment produced within the pair (u_1, u_2) is equal to:

$$[u_1, u_2] + [u_2, u_1] \tag{1}$$

Now all we need to do is sum over all the $n-1$ pairs. Thus we see that

$$\epsilon_{\text{total}} = \sum_{r=1}^{n-1} \left([u_r, u_{r+1}] + [u_{r+1}, u_r] \right), \tag{2}$$

where r is a dummy variable.

3.2. Application to an example arrangement. We will examine the XKCD network in Figure 5. For the sake of variety, we will not use the arrangement $1, 2, 3, 4, 5, 6, 7, 8$. Generating a random sequence using an online tool [Random.org(2016)], we will use the sequence $3, 2, 8, 5, 4, 6, 1, 7$.

To begin with, we need to know the vectors x_i, which need to be hardcoded. In Section 3, we showed the full form of x_6. We repeat the same process to find $x_i, 1 \leq i \leq 8 \mid i \in \mathbb{N}$ (because $n = 8$).

$$\begin{aligned}
x_1 &= \langle 0, 2, 0, 0, 0, 0, 0, 0 \rangle \\
x_2 &= \langle 2, 0, 1, 0, 0, 0, 0, 1 \rangle \\
x_3 &= \langle 0, 1, 0, 1, 0, 0, 0, 0.5 \rangle \\
x_4 &= \langle 0, 0, 1, 0, 0.5, 2, 0, 0 \rangle \\
x_5 &= \langle 0, 0, 0, 0.5, 0, 1, 0, 0 \rangle \\
x_6 &= \langle 0, 0, 0, 2, 1, 0, 0.5, 0 \rangle \\
x_7 &= \langle 0, 0, 0, 0, 0, 0.5, 0, 2 \rangle \\
x_8 &= \langle 0, 1, 0.5, 0, 0, 0, 1, 0 \rangle
\end{aligned}$$

In order to calculate ϵ_{total}, we just need to apply Formula 2. Doing so,

$$\begin{aligned}
\epsilon_{\text{total}} &= \sum_{r=1}^{7} \left([u_r, u_{r+1}] + [u_{r+1}, u_r] \right) \\
&= [3, 2] + [2, 3] + [2, 8] + [8, 2] + \ldots + [1, 7] + [7, 1] \\
&= 1 + 1 + 1 + 1 + 0 + 0 + 0.5 + 0.5 + 2 + 2 + 0 + 0 + 0 + 0 \\
&= 9
\end{aligned}$$

It is worth commenting on the fact that all of the above values come in identical pairs (that is, $[u_r, u_{r+1}] = [u_{r+1}, u_r]$). This is not true generally. However, for this particular network, there is only one asymmetrical relationship, the one-way crush between individuals 7 and 8 – and in our particular random arrangement, 7 is never next to 8.

We could have found ϵ_{total} without our vector notation, and simply with arithmetic by looking at the linkages on the graph (as described in Section 2.2). However, the process that we have elucidated in this section does not require actually looking at the networks, meaning that we can easily employ a computer program to do the calculations for us.

4. Programming

Equation 2 allows us to effectively calculate ϵ_{total} for a particular arrangement. We will now discuss the process of writing a computer program which finds the optimal arrangement.

4.1. Overview of the program. Here is a summary of what we will need our program to do:

(1) Accept an input of n vectors, which represents the social network.
(2) Somehow, find the arrangement which has the highest value of ϵ_{total}.
(3) Output the optimal arrangement, with it's value of ϵ_{total}.

Steps 1 and 3 just require a knowledge of the programming language's syntax. However, the actual optimisation which occurs in Step 2 is not so simple. A brute force search (i.e, calculating ϵ_{total} for every possible arrangement and returning the maximum), is not feasible due to the the computational complexity, which is $\Theta(n!)$ [Weisstein(2016)]. We therefore need to implement an optimisation algorithm. The majority of these rely on calculus, which in turn requires a continuous independent variable. Our independent variable is the arrangement, which is neither continuous nor ordered, meaning that we cannot use many of these algorithms. How, then, do we approach this problem?

4.2. Genetic algorithms. A genetic algorithm is a "search heuristic that mimics the process of natural selection" [Mitchell(1996)]. Loosely speaking, it is a 'shortcut' method of optimisation based on Darwinian natural selection. The relevant features of natural selection are stated here:

(1) There is a population of phenotypes, that is, individual members with different genetics and hence different characteristics.
(2) This variety means that certain individuals are better suited to their environment.
(3) The 'fitter' individuals survive and reproduce, passing on their genetic information to the next generation.
(4) During the process of reproduction, random mutations may occur, resulting in slight changes to the child phenotype.
(5) The process repeats, and over time, the species gets increasingly better adapted to the environment.

The functioning of a genetic algorithm corresponds very well to the above description of natural selection. For our particular problem, the genetic algorithm needs to find the optimal arrangement. An overview of the possible functioning of a genetic algorithm for our purposes is as follows:

(1) There is a population of different possible arrangements.
(2) Some of these arrangements have a higher ϵ_{total}.
(3) These phenotypes survive, and some of the survivors reproduce, that is, their arrangements are 'combined'. In computer science, this step is known as the *crossover* [Whitley(1994)].
(4) Random mutations occur, *i.e.* some arrangements may be changed slightly.
(5) The process repeats, and over time, the mean ϵ_{total} of the population increases. Eventually, it may reach the true optimum.

We will now attempt to implement the ideas presented in this section in python. Python was chosen for its simple syntax and relative ease of use.

5. Implementation in python

As discussed in Section 4.1, we can split the program into three sections: input, optimisation, and output. In Section 4.2, we further subdivided optimisation into five steps. We will explain the key concepts behind the code, and later apply it to the XKCD network.

While there exist python packages that implement genetic programming, we will code our basic genetic algorithm from first principles.

5.1. Input. Though we are not going to fully calculate the optimal arrangement for the XKCD network just yet, we will use it to demonstrate how we will input data into our program. Of the many possible data structures, a python list will be used, for its simplicity. However, we will need to use a 'list of a list', that is, a list wherein each entry is also a list. For clarity, we will refer to the outer list as a 'superlist'. Our network is then represented as:

```
network = [[0, 2, 0, 0, 0, 0, 0, 0],
           [2, 0, 1, 0, 0, 0, 0, 1],
           [0, 1, 0, 1, 0, 0, 0, 0.5],
           [0, 0, 1, 0, 0.5, 2, 0, 0],
           [0, 0, 0, 0.5, 0, 1, 0, 0],
           [0, 0, 0, 2, 1, 0, 0.5, 0],
           [0, 0, 0, 0, 0, 0.5, 0, 2],
           [0, 1, 0.5, 0, 0, 0, 1, 0]
```

The parallels between this format and that in Section 3.2 are evident.

It may be worth noting now that there may be an added confusion due to the way python indexes lists. For example, in order to obtain the third entry of the fifth vector, one would be forgiven for thinking that `network[5][3]` would return the needed result. This code would return the second entry of the fourth vector. The reason for this that python lists/arrays begin their indexing with zero, meaning that to return the first entry of the list, you would need to use `yourlist[0]`.

5.2. Generating the population. In order to generate a population, we need to know the network size (the number of individuals) and the desired population size.

```
1   import random
2
3   population = []
4   # Empty superlist, which will contain the arrangements
5   # Superlist because each arrangement is itself a list
6
7   def generate_population(network_size, population_size):
8   # input the network size and the population size
9
10      test_phenotype = []
11      for i in range(1, network_size + 1):
12          test_phenotype.append(i)
13      # The above code produces the simplest arrangement,
14      # which is just the integers in order.
15      # e.g for the XKCD network,
16      # test_phenotype = [1,2,3,4,5,6,7,8].
17
18      for i in range(1, population_size + 1):
19          random.shuffle(test_phenotype)
20          population.append(list(test_phenotype))
```

Lines 18-20 are the important part of this process. Within the for-loop, we shuffle the test phenotype, then append the result to the population superlist. The result is that the superlist `population` contains a list of random possible phenotypes.

5.3. Evaluating a phenotype's fitness. We now need to write a function which will accept an input of a particular phenotype (that is, a list representing a possible arrangement), and then return ϵ_{total}, *i.e.* implementing Formula 2. We do this by initially setting $\epsilon_{\text{total}} = 0$, then cumulatively adding the contributions from each pair.

```
def evaluate_etotal(arrangement):
    etotal = 0
    # Calculating $\sum_{r=1}^{n-1}[u_r, u_{r+1}] + [u_{r+1}, u_r]$
    # using a for-loop.

    for r in range(0, network_size - 1):
        u_r = arrangement[r] - 1
        u_r1 = arrangement[r + 1] - 1
        # u_r1 is meant to represent $u_{r+1}$
        # When defining u_r and u_r1, we subtracted 1,
        # because indices in python start at zero.

        epair = network[u_r][u_r1] + network[u_r1][u_r]
        # network[u_r][u_r1] is the same as $[u_r, u_{r+1}]$
        # epair represents the contribution by the pair

        etotal += epair
        # We add the contribution from the pair
        # to the running total.

    return(etotal)
```

In Section 3.2, we worked out ϵ_{total} with Formula 2. Let us instead use the function that we have just defined, by running:

```
evaluate_etotal([3, 2, 8, 5, 4, 6, 1, 7])
```

This returns a value of `9.0`, which concurs with our previous calculation.

5.4. Survival of the fittest. We have defined a function to measure the fitness of an arrangement, which should be applied to all of the arrangements in the population. We then need to find a way to remove the 'weak' individuals.

```
import statistics

population_fitness = []

for phenotype in population:
    phenotype_fitness = evaluate_etotal(phenotype)
    population_fitness.append(phenotype_fitness)
# population_fitness is a sibling list of population.
# Each element in population_fitness is the etotal
# of the corresponding arrangement in the population list.


```

```python
elements_to_remove = []
median = statistics.median(population_fitness)

for i in range(0, len(population)):
    if population_fitness[i] < median:
        elements_to_remove.append(i)
# If a phenotype's etotal is lower than the median,
# we schedule it for execution.

survivors = [i for j, i in enumerate(population)
                 if j not in elements_to_remove]
# A survivor list is created, made of 'fitter' phenotypes
```

Clearly, the survivor list is smaller than the population list, because some phenotypes have died. Thus, in our next section, we will devise a way for the survivors to reproduce such that the population size is maintained.

5.5. Crossover. The essential characteristic of reproduction, when it comes to genetic algorithms, is that the offspring receives features from both parents (who are themselves 'fit') – there is a crossover of genetic material.

With regard to our problem wherein the phenotypes correspond to different arrangements, a naïve method of crossover could be to take the first half of one arrangement and combine it with the second half of another arrangement. This approach was initially adopted by the author, who soon realised his mistake: such a method allows for duplicates within the arrangement. Consider two potential parent phenotypes, `[1,2,3,4]` and `[3,2,4,1]`. The offspring phenotype would be `[1,2,4,1]`. This is clearly not an acceptable phenotype – how can someone be seated on both ends of a row?

We thus require a more ingenious way of combining two arrangements. However, it was beyond the author to devise such a method. Thus, we will have to forsake one of the main aspects of crossover: that the offspring receive genetic material from two parents. Instead, our offspring will be made by modifying a parent.

```python
import random

next_generation = survivors
# We define the new generation, which begins with the survivors.

for i in range(0, number_removed):
    # This for-loop exactly maintains the population size.
    parent = random.choice(survivors)
    # Pick a parent from the survivors
    estranged_father = parent[int(network_size/2):network_size]
    random.shuffle(estranged_father)
    child = parent[0:int(network_size/2)] + estranged_father
    # The above code takes the first half of a parent,
    # then shuffles the rest to make a child

    next_generation.append(child)
    # The child is welcomed to the new generation.
```

The drawback of this method is reasonably obvious. We are not properly 'crossing over' genetic material; in fact, what we have done here is merely a glorified form of mutation. It will be left to the reader to invent any more suitable means of reproducing two arrangements without forming duplicates.

5.6. Mutation. Mutations are random changes to the genetic material. With regard to seating arrangements, we will consider a mutation to be the swapping of any two members within the arrangement.

A required parameter is the *mutation rate*, which encodes how often mutations should occur. A value of 0.05 (which is what we shall use) means that, on average, five in every 100 phenotypes will contain a swap.

```
mutation_rate = 0.05

for i in range(0, population_size):
    if random.random() < mutation_rate:
    # random.random() returns a float between 0 and 1,
    # so this if-statement results in a probability
    # equal to the mutation rate

        to_swap = random.sample(range(network_size), 2)
        a, b = to_swap[0], to_swap[1]
        population[i][b], population[i][a] =
                population[i][a], population[i][b]
        # Picks two elements at random from an arrangement
        # and swaps them.
```

5.7. Iteration. The iteration step is the most simple. We simply repeat the process of survival, crossover, and mutation over a number of generations. This can be quite easily done with a for-loop in python. However, a comment must be made regarding the code.

In Sections 5.2 and 5.3, we wrote python functions, which need to be called when running the program. However, to aid understanding, we did not do this for the remaining blocks of code. For example, in Section 5.4, no function was defined. The final program has been written slightly differently, though the content is practically the same. An example of the difference is that survival and crossover have been merged into one function named `survive_and_reproduce()`. The full code has been relegated to Appendix A.

5.8. Application to the XKCD network. In Section 5.1, we discussed how the network needs to be entered into the program (we even used the data for the XKCD network). All that remains to be done is choosing suitable values for the parameters. In the past, population sizes of around 100 have been chosen [Sarmady(2007)], but a more recent analysis has suggested that a population size of 16 is optimal [Haupt(2000)]. This is the value we shall use.

Running the program a few times with a population size of 16 and a generation number of 100 did not yield a consistent optimal arrangement. Final values of ϵ_{total} ranged between 15 and 17. Figure 6 shows a graph of one possible convergence, which was accompanied by the following output:

```
The optimal arrangement is:  [5, 6, 4, 3, 7, 1, 2, 8],
with an etotal of:  17.
```

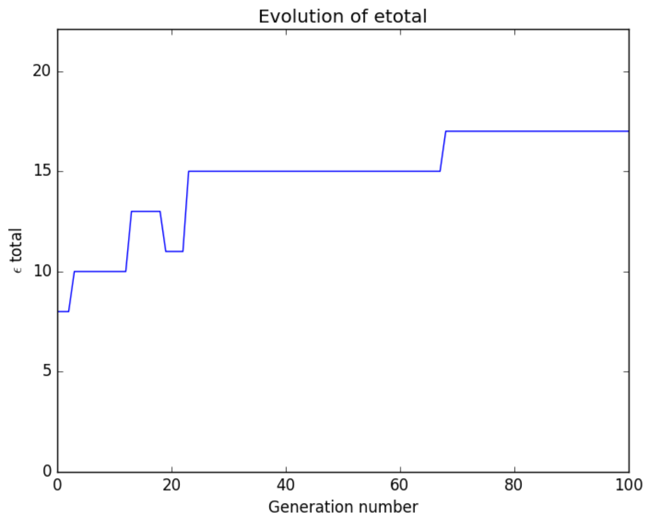


FIGURE 6. population size = 16, generation number = 100

We can improve this by increasing the generation number, which probabilistically ensures that the optimum will be reached. It is an interesting exercise to make changes to the parameters and observe the results. From a cursory glance, decreasing the population size tends to increase the volatility of the ϵ_{total} values. Increasing the population size seems to cause the values to converge faster. This can be seen in Figure 7, which converges on $\epsilon_{\text{total}} = 17$ at around the 30th generation.

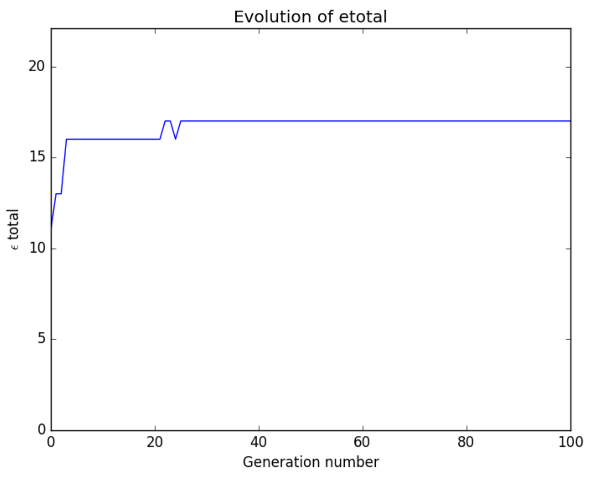


FIGURE 7. population size = 100, generation number = 100

6. Conclusion

It is of the author's belief that the methods outlined in this investigation are sufficiently robust as to deal with larger networks. One of the reasons for this is that the parameters can be adjusted as required, so for larger networks, perhaps a larger population size and a greater generation number would be required.

This said, genetic algorithms may not be the best tool for finding the optimal arrangement. In fact, it is not certain that the method implemented here may even be considered a true genetic algorithm, because of the problems with implementing the crossover (see Section 5.5).

Despite this, we saw that acceptable results were produced, at least for the XKCD network. It would be interesting, if unscientific, to make further observations concerning the optimal solution. For example, we see that in the optimal arrangement, the mutual crushes are grouped together as couples – which makes intuitive sense. Facts like these could, in future, be used to improve the optimisation method.

6.1. On the possibility of an analytical solution. It is not inconceivable that, for the problem of optimising linear seating arrangements, there exists an analytical solution, such that by incorporating the network vectors $x_1, x_2, \ldots x_n$ into a suitable deterministic formula, the result would be the arrangement maximising the ϵ_{total}.

The author suspects that this problem would fall under the realm of graph theory, as indeed the social networks presented such as in Figure 3 are, mathematically speaking, termed *weighted graphs* [Peter Fletcher(1991)]. This may well be a cousin of the Travelling Salesman Problem, as we could interpret our investigation as trying to find the optimal route through the social network.

6.2. Extensions. It is true that the problem considered by this investigation was rather narrow: there are only a few situations wherein a social group has to sit in a line. That being said, one possible extension could be to deal with circular arrangements, which are far more common. This could have a tangible application when organising seating plans for meal times or large functions such as weddings. Obviously the problem would not be identical, but the key elements are the same. The total enjoyment would be a function of the arrangement, so we would just have to optimise this.

Another potential criticism is that we classified all forms of social interactions into four types (five including strangers), as seen in Figure 2. This is, in fact, one of the easier things to extend. We can have as many friendship types as required, or even make a continuous spectrum of it. All that would change is the composition of the network vectors. Instead of saying that a friend causes an enjoyment of 1, you could be more specific and assign a value of 1.2 to friend A, 0.9 to friend B, etc. This would only affect the input of the network into the program; the program itself would run just as before. Again, this does not ameliorate the issue of quantifying different social relationships. But let this be something for sociologists to discuss.

Author's address
martin.robertandrew@gmail.com

References

[Haupt(2000)] Haupt, R. L. (2000). Optimum population size and mutation rate for a simple real genetic algorithm that optimizes array factors. *Antennas and Propagation Society International Symposium, 2000. IEEE*, *2*, 1034–1037 vol.2.

[Mitchell(1996)] Mitchell, M. (1996). *An Introduction to Genetic Algorithms*. 9780585030944. Cambridge, MA: MIT Press.

[Munroe(2006)] Munroe, R. (2006). Movie seating. https://xkcd.com/173/.

[Peter Fletcher(1991)] Peter Fletcher, C. W. P., Hughes Hoyle (1991). *Foundations of Discrete Mathematics*. 0-53492-373-9. PWS-KENT Pub. Co., international student ed ed.

[Random.org(2016)] Random.org (2016). Random sequence. https://www.random.org/sequences/.

[Sarmady(2007)] Sarmady, S. (2007). An investigation on genetic algorithm parameters.

[Weisstein(2016)] Weisstein, E. W. (2016). Big-theta notation. MathWorld – A Wolfram Web Resource.
URL http://mathworld.wolfram.com/Big-ThetaNotation.html

[Whitley(1994)] Whitley, D. (1994). A genetic algorithm tutorial. *Statistics and Computing*, *4*(2), 65–85.

Appendix A. The complete program

```
import random
import statistics
import matplotlib.pyplot as plt

# Required parameters
network_size = 20
population_size = 100
gen_number = 1000  # the number of iterations
network = []  # Input formatted as in Section 5.1

population = []
progress_list = []
# Used to track the iterations
hall_of_fame = []
#contains the best arrangements from each generation


def generate_population(network_size, population_size):
    test_phenotype = []

    for i in range(1, network_size + 1):
        test_phenotype.append(i)
    for i in range(1, population_size + 1):
        random.shuffle(test_phenotype)
        population.append(list(test_phenotype))


def evaluate_etotal(arrangement):
    etotal = 0

    for r in range(0, network_size - 1):
        u_r = arrangement[r] - 1
        u_r1 = arrangement[r + 1] - 1
        epair = network[u_r][u_r1] + network[u_r1][u_r]
        etotal += epair

    return (etotal)


def survive_and_reproduce(end=False):
    population_fitness = []

    for phenotype in population:
            phenotype_fitness = evaluate_etotal(phenotype)
            population_fitness.append(phenotype_fitness)

        progress_list.append(max(population_fitness))
        max_location = population_fitness.index(
                    max(population_fitness))
        hall_of_fame.append(
                    population[max_location])
        # Add the generation's champion to the progress_list

        if end:
            champion_location = progress_list.index(
                            max(progress_list))

            print("The optimal arrangement is: ",
                hall_of_fame[champion_location],
                ", with an etotal of: ",
                max(progress_list))
        # We print the overall best arrangement, and its etotal.

        median = statistics.median(population_fitness)

        elements_to_remove = []

        for i in range(0, len(population)):
            if population_fitness[i] < median:
                elements_to_remove.append(i)

        survivors = [i for j, i in enumerate(population)
                     if j not in elements_to_remove]
        number_removed = len(elements_to_remove)

        next_generation = survivors

        for i in range(0, number_removed):
            parent = random.choice(survivors)
            estranged_father = parent[
                                int(network_size / 2):network_size]
            random.shuffle(estranged_father)
            child = parent[0:int(network_size / 2)] \
                    + estranged_father
            next_generation.append(child)

        return next_generation
```

```
def mutate(generation):
    mutation_rate = 0.05

    for i in range(0, len(generation)):
        if random.random() < mutation_rate:
            to_swap = random.sample(range(network_size), 2)
            a, b = to_swap[0], to_swap[1]
            generation[i][b], generation[i][a] = \
            generation[i][a], generation[i][b]


generate_population(network_size, population_size)

for i in range(0, gen_number + 1):
    if i != gen_number:
        next_generation = survive_and_reproduce()
        mutate(next_generation)
        population = next_generation
    else:
        survive_and_reproduce(True)
        # On the last iteration, we print the overall winner.


# Optional code to graph the change in etotal
# as a function of generation number
plt.plot(progress_list)
plt.title('Evolution of etotal')
plt.xlabel('Generation number')
plt.ylabel(r'$\epsilon$ total')
plt.axis([0, gen_number, 0, 1.3*max(progress_list)])
plt.savefig('etotal_graph.png')
```

Author's Address
martin.robertandrew@gmail.com