Emergent Computing For Optimisation – Report

Krzysztof Dworczyk - 40340711

November 2020

# Introduction

In this report I will be describing the approach I took to tackling the issued problem where a roster of 523 players was given and the best combination made up of 11 players was being tasked to find. This is a quiet large search space with many available combinations, for this reason I have chosen to go with an evolutionary algorithm. A list of constraints was also presented so the final team must obey by all of them as well as trying to maximise the overall points gained on top of staying under the maximum cost of £100.

# Approach

Firstly, I investigated how the teams are represented in the data file to try determining the best approach. The provide helper function extracted the essential data from the file and placed them inside lists. This allowed me to manipulate it much easier and begin creating a representation. To represent my team, I have gone with an array of zeros where each number is a gene representing if a player is on the team. This allowed me to use a lot of the off-the shelf operators for mutation and crossing. As the problem is nicely represented as such, it posed no problem continuing with it. The lists containing what player is in which position were also represented with a binary array which went hand in hand and allowed for quick checks for players costs and positions.

One parameter that will be worth looking at is the mutation probability which is discussed in the (R. N. GREENWELL, 1995) paper. It mentions that mutation probably will vary depending on the problem however it was said that if you want to use the mutation probability then it’s best change as much as possible in an individual or not at all. This will be tried as to see if any good result can be derived from it.

An interesting paper by (Wen-Yang Lin, 2003) which showcased how we can automatically adapt the crossover and mutation rates to the problem domain. This was to try and reduce the amount of time spent on manually fine tuning those paraments. They have found great success in this technique; however, it is a rather complicated features which will not be applied however still promising to see such results.

The given task also provided constrains which we need to make sure the final team obeys. I have simply chosen to ensure that every team created is feasible by creating a custom initialisation function. Furthermore, I will also have to create a custom evaluation function which will also make sure that any crossed/mutated team also fits inside all the constraints. Making sure that all out team are in the feasible region allows us to quickly convergence on the optimum as everyone has a possibility of being near it. As talked in the (Jos ́e G. M. Esgario, 2019) paper, it’s best to keep the population in the feasible regions to ensure that all team can be used as an answer. A death penalty will be applied which will throw away any team that does not fit into the constraints. Due to team being thrown out we will also need to find an optimal population that will provide enough diversity for is to keep creating new individuals.

I will run experiments on different parameters of the algorithm to try and find any more optimal numbers for example the population size and the tournament size. These all will be run 5 times to ensure enough samples have been collected to draw any meaningful conclusions.

# Algorithm & Operator Design

I have chosen to use Evolutionary Algorithm to find the maximum fitness of a team. Below I will go in detailed about some custom function I created to keep the population in the feasible range. The chosen selection, mutation and crossover also will be discussed.

## Custom Population initialisation

A custom population initialisation function was written so that every team created is feasible. Below you can see a snippet of the code which is used to ensure that every constrain is met before the players are even chosen.

Text

Description automatically generated

Text

Description automatically generatedThe first four lines satisfy the number of players in each position. There always needs to be a goalkeeper so we set that to one, team defence and midfielders get to random generate a number between a range, they’re not the same as if a maximum of 5 players for both positions were chosen then we have 11 players but no strikers. Therefore, we skew the generated range slightly so that there is always enough room for every position. The team strikers’ positions are filled with the remaining spaces left on the team.

I then map a list of ranges to the positions of players in the data documents as they appear in order. I shuffle those lists of indices to be left with a random sequence of players. I then loop through these lists filling up the positions using the previously generated team\_gk, team\_defe, team\_mid and team\_stri. If a cost constraint is broken when a new player is picked, then we move on to the next one in that lists sequence.

## Selection: Tournament

For the selection operator I have chosen the tournament. In (Yuri Lavinas, 2018) we find out that tournament size results vary largely from one problem domain to another. In this case I have found that increasing the tournament size dosed in fact help with reaching the best solution quicker as will be shown and discussed in the experiment section.

## Crossover: CxOnePoint

The crossover operator of choice was the cxOnePoint as it resulted in best performance as well as it does not mix up the populations too much. Since we are using a death penalty evaluation method, we are losing some feasible teams, to try and maintain as many feasible ones as possible I have chosen a crossover which result in least amount of mixing of random genes.

This operator still gives us plenty of crossing between individuals, however these changes will be quite large as half of each parent will be sued for the generated offspring.

## Mutation: mutShuffleIndexes

For the mutation operator I have gone with the provided mutShuffleIndexes which works by taking in an individual and applying its independent probability (indpb) on each gene to check if it should be moved. This allows for a large range of random mutation which will help in keeping the population divers.

## Evaluation: Custom Team Evaluation Function

For evaluation I have written a custom evaluation operator which ensures that no constrains have been broken once a team has been crossed over or mutated. I am using a death penalty system which assigns the fitness of 0 to any team that breaks constraints. I have chosen to do it that way as it ensures that every team in the populations is feasible which should in turn allows the solution to always remain feasible.



The if statement above checks that each constrain is satisfied before returning an individual. If the team costs too much it will be scrapped. If the final team isn’t made up of exactly 11 players, it will also be dropped. If both of those are satisfied, I then check if each position has the correct number of players playing them by giving it a range.

If all are satisfied, we pass the individual back to add to the population.

# Experiments & Analysis

The following section describes the tests and results produced.

## Parameters Used:

MUTPB = 0.1

CXPB = 0.7

POPSIZE = \*Will change in this experiment\*

NGEN=300

TNSIZE=2

## Population Size Optimization

Population size was one of the first experiments I ran as I expected it to have a big effect due to the diversity factors. All the teams I create are feasible and the ones that are not get dropped. This means that I might start losing a lot of diversity fast once the teams start crossing and mutating. To try and mitigate this I tested on population size and presented the findings below.

I will be trying to reject the following hypothesis: “changing the population size from 500 to 1500 has no effect on performance”

|  |  |  |
| --- | --- | --- |
| **PopSize** | **Fitness** | **gensMaxFound** |
| 500 | 1893 | 275 |
| 500 | 1824 | 275 |
| 500 | 1759 | 91 |
| 500 | 1817 | 258 |
| 500 | 1871 | 62.0 |
| 1000 | 1887.0 | 277.0 |
| 1000 | 1921.0 | 205.0 |
| 1000 | 1879.0 | 136 |
| 1000 | 1855.0 | 221 |
| 1000 | 1842 | 171 |
| 1500 | 1889 | 116 |
| 1500 | 1966 | 202 |
| 1500 | 1958 | 294 |
| 1500 | 2003 | 190 |
| 1500 | 1921 | 289 |

|  |  |  |
| --- | --- | --- |
| **500’s average** = 1,832.8 | **1000’s** average = 1876 | **1500s** average = 1947 |

The table above shows 5 runs of 3 different populations sizes starting at 500 and incrementing by 500 each time top try and highlight if there is any noticeable difference.

### Statistical test results

Probability that the 500 populations and 1500 population samples are taken from the same distribution: 0.22%. This allows us to reject the hypotheses as changing the population dose in fact increase the performance of the algorithm.

## Tournament Size Optimization

## Parameters Used:

MUTPB = 0.1

CXPB = 0.7

POPSIZE = 500

NGEN=300

TNSIZE=\*Will change in this experiment\*

Another promising experiment was the tournament size. We increase the size to try and put more pressure on selecting the better fitness scores for each team. Below is a table breaking down each tournament size and the best fint4edsss and what generation they were found at.

I will be trying to reject the following hypothesis: “changing the tournament size from 2 to 10 has no effect on performance”

|  |  |  |
| --- | --- | --- |
| **TournSize** | **Fitness** | **gensMaxFound** |
| 2 | 1868 | 249 |
| 2 | 1702 | 292 |
| 2 | 1832 | 299 |
| 2 | 1688 | 295 |
| 2 | 1579 | 253 |
| 6 | 1578 | 295 |
| 6 | 1782 | 293 |
| 6 | 1764 | 272 |
| 6 | 1759 | 226 |
| 6 | 1721 | 225 |
| 10 | 1661 | 299 |
| 10 | 1728 | 253 |
| 10 | 1704 | 258 |
| 10 | 1703 | 296 |
| 10 | 1777 | 229 |

|  |  |  |
| --- | --- | --- |
| **2 average** = 1733.8 | **6** average = 1,720.8 | **10** average = 1,714 |

The table above shows 5 runs of 3 different tournament sizes starting at 2 and incrementing by 4 each time top try and highlight if there is any noticeable difference.

### Statistical test results

Probability that the 2 tournament size and 10 tournament size samples are taken from the same distribution: 1.2%. This allows us to reject the hypotheses as changing the population dose in fact increase the performance of the algorithm.

# Conclusion

For the experiments we have found out that population size greatly increases the diversity which in turn gives more chance for the best team to emerge. The best team was found accumulating 2003 points, this was found when the population was set to 1500 as to allow for large diversity. I achieved this score using the Tournament Selection size 6, cxOnePoint, mutShuffleIndexes mutation. With the experiment fine tuning the parameters I believe is the reason I was able to achieve such a score.

I found it interesting how much each run may be different in terms of luck or being unlucky. Algorithm ran with he same paraments may not always give the same answer and it all depends on chance sometimes just like real evolution. With larger population however we increase the chance of finding the best individual.

Overall, I am happy with the results achieved however I do believe there is room for improvements. Sometimes the algorithm may get stuck on a local optima fort a few generations, time could be saved by including an annealing algorithm which allows for worse teams to be allowed so the algorithm can find its way out of the local optimal.

To conclude the final team and parameters to arrive at the permutation are as follows:

## EA Parameters

Population size: **1500**

Tournament size: **10**

Mutation Probability: **0.15**

Crossover Probability: **0.7**

Max Generations: **300**

The screenshot of the best found team can be found in Appendix A.

# Future Work

In retrospective, further improvements could have been made to the mutation operator. A custom operator could increase the efficiency of the algorithm when mutating the player teams. I used an included operator which worked fine however it was mostly a random mutation which may have resulted in a team becoming not feasible if for example an extra goalkeeper get added to the team making the total 2. Instead it could have picked a payer from the selected team and if a player is chosen to be mutated then the function should random choose another player from the document but from the same position. This means that the team would always be feasible increasing your chances of creating the optimal team.

# Works Cited

Jos ́e G. M. Esgario, I. E. (2019). *Application of Genetic Algorithms to the MultipleTeam Formation Problem.*

R. N. GREENWELL, J. E. (1995). *Optimal Mutation Probability for Genetic Algorithms .* ScienceDirect.

Wen-Yang Lin, T.-P. H. (2003). *Adapting Crossover and Mutation Rates in Genetic Algorithms.* ResearchGate.

Yuri Lavinas, C. A. (2018). Experimental Analysis of the Tournament Size on Genetic Algorithms. *International Conference on Systems, Man, and Cybernetics*, (p. 7).

# Appendix A

Text

Description automatically generated