SET10117 Optimisation of a Fantasy Football Team

Calum Hamilton
40205163

Introduction:

In this coursework, I was assigned the task of creating an Evolutionary Algorithm (EA) which selects a team of 11 football (soccer) players from a pool of 523 players. Football players are each assigned a score – indicative of their skill level – and a cost.
The EA needs to generate the best team possible from the pool of players given, while remaining under the assigned budget (**£100**). This is a Maximisation problem – we're looking to Maximise the team score.
In this report, I detail my approach and implementation of this Evolutionary Algorithm.

Approach:

When approaching this task, I wanted to understand how previous people had approached similar problems. While I was unable to find papers online pertaining to football/soccer team selection, I was able to find two papers that dealt with cricket team selection.
For both papers, the representation was the same. They created a Population of Individuals, the length of the team being selected and assigned a value which corresponded to the player's position in a separate data structure. When evaluating the solution, the sum of scores and costs could be drawn from each solution, giving the solution quality.

This seemed like a solid approach for the Fantasy Football optimisation, as it restricted the search space and ensured the maximum number of players that could be selected was 11. It avoids crossover and mutation issues you may have with a binary representation (e.g. needing to ensure the string always sums to 11, that any flip-bit mutation would require a 1 to be flipped to 0 if another 0 is flipped, etc.)
I chose to use a Generational Hill Climber algorithm for the problem. It's an algorithm I understand very well and felt that it would provide options for lot of customisation. It's likely not the optimal algorithm for the task, but I was interested to see how strong the results I could generate were, using the Hill Climber.
My report and coursework focus on the customisation and optimisation of Hill Climber operators.

Algorithm/Operator Design:

Further Chromosome Design

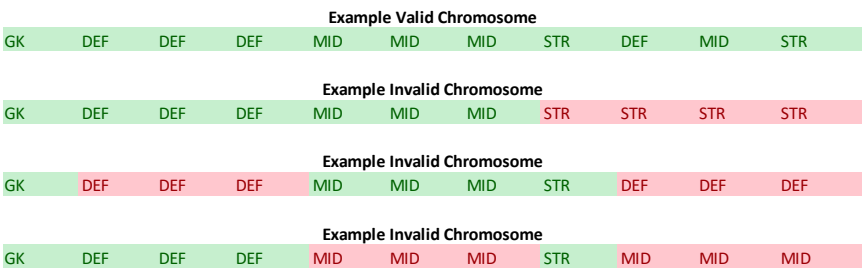
For each Football team, there are a certain number of required players in each role (e.g. you wouldn't want 11 Goalkeepers on a team). Below are the number of players required for each role:

Role	Goalkeeper	Defender	Midfielder	Striker
Team Minimum	1	3	3	1
Team Maximum	1	5	5	3

I wanted to ensure my chromosome design was as close as possible to the team's role restrictions. This would limit the search space further and help ensure that children created by different Operators solutions were more likely to be valid. My initial Chromosome design was as follows:

Position	1	2	3	4	5	6	7	8	9	10	11
Role	GK	DEF	DEF	DEF	MID	MID	MID	STR	MIX	MIX	MIX
Search Space	57	180	180	180	196	196	196	90	466	466	466

For the mixed positions (9, 10 and 11), I allowed any Defender, Midfielder or Striker. This however created a large search space for the mixed positions and allowed three invalid team composition phenotypes. I examine these invalid phenotypes below:



GK	DEF	MID	STR
1	4	4	2
1	3	3	4
1	6	3	1
1	3	6	1

This chromosome design was changed after updating the mutation operator, but I discuss it here to allow a clearer and better understand the changes made – since the design is already being discussed. I updated the design to remove invalid phenotypes that could be created and wanted to constrict the search space as much as possible. Furthermore, removing team composition errors meant the only errors thrown would be errors that could occur by the team breaking the cost restriction.
In the table below, I examine the changes made to the *mixed genes* 9 – 11, to solve this issue.

For each position type, the minimum required players for each role were fixed on the left of the chromosome (positions 1-8). For the remaining positions (Defender, Midfielder, Striker), there were a maximum of two players in a single role that could be taken, as taking three of players of a single role result in an invalid phenome.
I updated the last three genes, to restrict the maximum players that could be selected in a single role to two, while keeping the minimum number of players in a role that could be selected as zero.

Position	1..8	9	10	11
Role	Required	DEF & MID	MID & STR	STR & DEF
Search Space	Multiple	376	286	270

Comparing Operators

I compare various operators in the section below. If one operator has been shown to have a statistically significant advantage over another, I update the algorithm to use this operator. This naturally means that as the report progresses, the operators being used for testing will change – the last operator being examined isn't running on the exact same algorithmic variation as the first operator. However, where changes have been made to the algorithm, this has been clearly stated. **Every operator has been tested for 50 runs on a population size of 200.** This ensures that evaluation is fair and allows for more accurate statistics to be gathered and better conclusions to be drawn. All standard deviation values have been calculated as a sample of the population, as opposed to the entire population.

There have been two different colours used in tables comparing operators below.
Operator Highlighted Orange: This previous default operator for the EA and the operator that new possible operators are compared to.
Operator Highlighted Green: The operator selected for use going forward.

Algorithm Design:

Algorithm type:	Generational	**Selection Method:**	Tournament
Representation:	Integer-based	**Tournament Size**	3
Crossover:	Two-Point	**Mutation rate:**	0.091
Population:	200	**Replacement:**	Replace Generation
Generations:	200	**Initialisation:**	Custom Initialisation

Mutation Operator Design

I tried using the inbuilt mutation operators, however I found that none of them suited the problem that I was trying to solve. This was due to the specific gene ranges (and later, I needed a custom mutation function to wrap the mutated value of the 11th gene between the Defender and Striker ranges).

In the table below:

- The mutateUniformInt operator refers to the built-in deap operator originally tested
- (1/length) or (3/length) are the mutation chances I was testing for each gene in the chromosome. I have simplified this to the according percentage chance (0.091%) and (0.273%)
- (± 5) or (± 10) is the uniform distribution that the new mutation value is selected from

Operator	Max	Mean	Std Deviation (sample)	p-value (normal)	t-test (p-value)	t-test (statistic)
(0.091%) ± 5	2056	1986.0	40.2	0.006	-	-
(0.273%) ± 5	2065	1989.8	41.3	0.014	0.640	0.469
mutUniformInt	2015	1921.8	56.7	0.336	2.91 e-09	-6.532
(0.091%) ± 10	2057	1988.3	41.0	0.384	0.777	0.283
(0.273%) ± 10	2069	2002.1	34.6	0.171	0.033	2.164

The t-test allows us to dismiss the possibility that the best performing operator (number 5) is drawn from the same distribution as the original operator used. There is statistically significant evidence to suggest that operator 5 is the best performing operator – it has found the single and average highest values of the mutation operators. It also has the lowest standard deviation – suggesting it's the most consistent operator.
This was the point I updated my chromosome design from the mixed ranges to specific ranges.

Crossover Design

The difference in operator performance between the best performing mutation operator and the two-point crossover operator can be attributed entirely to the new chromosome design.
I have understood from previous testing and discussion, that two-point crossover operator tends to outperform the one-point crossover operator. For this reason, I started off using two-point and later tested to see if one-point outperformed it. Testing suggests that the two-point crossover has a better max, mean and standard deviation, giving better results on average. There is also enough statistical

significance to conclude that the operators don't lie on the same distribution. For this reason, I decided to keep using the two-point crossover.

Operator	Max	Mean	Std Deviation (sample)	p-value (normal)	t-test (p-value)	t-test (statistic)
Two-Point	2086	2015.8	37.8	0.551	-	-
One-Point	2082	1986.9	46.5	0.405	0.00097	-3.401

Penalty Design

I had been multiplying the total returned score by a factor of 0.6 for every broken constraint. I wanted to ensure that solutions could be close or possibly slightly outside the penalty boundary, but also ensure that the best solution returned wasn't breaking constraints. As shown below, as the penalty multiplier started to reach a factor of 1, a much higher proportion of solutions were breaking constraints.
Note: the only constraint being broken was cost, so there wasn't ever more than 1 constraint broken.

Constraints Broken	0	1
0.60	50	0
0.70	50	0
0.80	17	33
0.85	5	45
0.90	0	50

I found that 0.7 was a good multiplication factor, as it prevented broken constraints being selected, while also not penalising good genetic material that broke constraints too harshly.

Selection Design

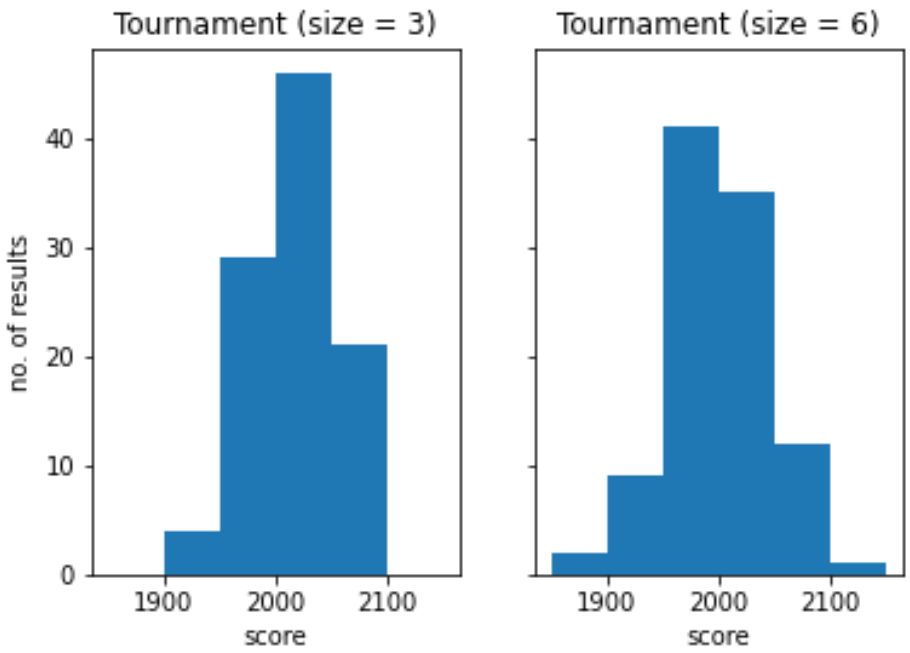
I had been using Tournament Selection (size = 3) and found this performed well. When choosing a selection operator, I wanted to test different tournament sizes and a range of other operators.

Operator	Max	Mean	Std Deviation (sample)	p-value (normal)	t-test (p-value)	t-test (statistic)
Tournament = 3	2086	2015.8	37.8	0.551	-	-
Tournament = 5	2082	2008.9	39.3	0.183	0.378	-0.886
Tournament = 6	2104	2002.7	45.9	0.903	0.124	-1.550
Tournament = 8	2069	1997.4	34.8	0.0006	0.013	-2.535
Roulette	2033	1958.9	34.9	0.606	6.41 e-12	-7.811
NSGA-II	1931	1866.7	24.7	0.953	8.49 e-42	-23.312
Random	1773	1485.4	111.4	0.119	1.59 e-53	-31.871

I found that tournament (size = 3) scored the highest mean and lowest standard deviation, while tournament (size = 6) was able to score a maximum value above 2100.
I wanted to compare the solutions head-to-head to see which solution would be better and to see if, given additional runs, the smaller tournament size could generate a higher max value. I decided to run each for a further 50 runs and compare the total of 100 runs.

100 Run Comparison						
Operator	Max	Mean	Std Deviation (sample)	p-value (normal)	t-test (p-value)	t-test (statistic)
Tournament = 3	2097	2016.5	36.4	0.874	-	-
Tournament = 6	2104	1999.5	43.5	0.377	0.003	-2.997

I graphed the distributions of these two tournament sizes through the matplotlib library.



Conclusion:

The results of the lower sized tournament tend to be more closely grouped and have higher mean values than the results of the higher sized tournament. The histogram gives a good visual representation of the difference between the maximum, mean and standard deviation values.

To conclude which algorithm should be selected, I present the original task's mission statement:
The goal is to select a team that maximises the total points score of the team, remains within budget, and satisfies all of the constraints.

The task is therefore to find the maximal solution satisfying all constraints.
Therefore, even though the original tournament is more consistent and has a higher average score, the single best solution from 100 runs came from the tournament with the larger selection size. I therefore present this as the 'best' solution and will therefore use the algorithm with the larger tournament size. I have included the screenshot of the 2104-score solution in the Appendix.

Evaluation:

An area where I could have explored and tested more operators was the crossover operator. I struggled with choosing an appropriate selection operator due to the chromosome design. Operators tested outside of one-point, two-point and uniform crossover needed to be custom written to deal with the range of accepted values for each gene, and the 11^{th} gene wrap from the Defender to Striker range. I tried using several other standard Deap operators, but the performance was poor, or they simply didn't work.

Additionally, I could have tested other Algorithm types than the Hill Climber. I know that Tabu search and Simulated Annealing tend to perform better than Hill Climbing, and likely these algorithms could have achieved better results. I felt however, that I didn't have time to fully investigate these different Algorithms, on top of the operator selection and exploration for each algorithm. I would recommend this as future work.

One strength of the task was the chromosome design that I created for the task. The initial idea and design, based on research of similar projects, gave me strong initial results. It highly simplified the expansive search space and generated solid initial results with the simple Hill Climber algorithm.
The updated version then further restricted the search space and eliminated further invalid permutations, making the problem like that of a knapsack problem.

Another strength was the scientific approach that I took. Firstly, I ensured my experiments were all conducted fairly and equally. I also ran experiments them enough runs to ensure the statistical significance between operators could be proven.
I also ensured that after fully testing each operator, I only updated one operator at a time. For example, I completed testing and experimentation on the mutation operator and selected the best operator, before moving on to looking at selection operators. This ensured that individual increases in performance could be specifically attributed to the operator.
Finally, I feel that I drew the correct conclusion from the two tournament sizes available. I feel this was because I fully understood the problem specification and viewed the solution with respect to the original brief.

Overall, I feel I approached this task in a solid manner and feel that the report and approach is appropriate for the task at hand.
For future work I suggest testing other algorithm types than the Hill Climber and custom writing additional crossover operators for testing.

References

1. S N, Omkar. (2003). Cricket team selection using genetic algorithm.
2. Ahmed, Faez & Deb, Kalyanmoy & Jindal, Abhilash. (2013). Multi-objective optimization and decision making approaches to cricket team selection. Applied Soft Computing. 13. 402–414. 10.1016/j.asoc.2012.07.031.

Appendix:

Single Best Solution – Output of the check_constraints function

```
Highest_Value_Chromosome = [470, 163, 15, 1, 180, 187, 196, 376, 29, 185, 0]
Binary_Highest_Value = convert_to_bin(Highest_Value_Chromosome)
print(check_constraints(Binary_Highest_Value))
```

```
total broken constraints: 0
total points: 2104.0
total cost is 100.0
selected players are [0, 1, 15, 29, 163, 180, 185, 187, 196, 376, 470]
(0, 2104.0)
```