Efficient Algorithms for Optimising Fantasy Football Team Selection

Daniel Travers

School of Mathematics and Statistics
University of St Andrews

I certify that this project report has been written by me, is a record of work carried out by me, and is essentially different from work undertaken for any other purpose or assessment.

Abstract

This project examines the effectiveness of three optimisation algorithms in optimising the selection of a Fantasy Premier League team over the course of a season. This specific task is similar to the 0/1 knapsack problem, a known combinatorial optimisation problem. However, there are some key differences, such as a time-varying aspect and multiple constraints on which selections are permitted.

To compare these algorithms, I first constructed a custom points-prediction model that used historical statistical data from the 2023/24 Premier League Season to produce a predicted points score for each player and every week of the 2024/25 season. Having obtained this, I discussed the theoretical background of each algorithm and then practically implemented each of them in Python.

My key finding was that the best solutions were obtained by a recursive knapsack algorithm which worked on a rolling two-week time horizon. I also found the greedy algorithm to be highly efficient in producing good solutions with little computational effort. This contrasted with the performance of the genetic algorithm, which produced relatively poor-quality solutions with a large computational effort.

Contents

Chapter One

Introduction

The fantasy sports industry has experienced dramatic growth in recent years, pitching itself as a safe and accessible alternative to traditional forms of sports betting. The United States Fantasy Sports and Gaming Association estimates that more than 60 million people play fantasy sports games in the US and Canada alone [1]. Despite the games being free-to-play, the global fantasy sports market is valued at over $37 billion as of 2025. This stems from advertising, sponsorship deals and the ability of fantasy games to boost engagement in the sports that they feature [2].

Fantasy Premier League (colloquially known as 'FPL') is one such example of a free-to-play online fantasy sports game, representing a significant proportion of this global market, with more than 10 million players signed up as of the 2024/25 season [3]. In FPL, individuals 'manage' their own football squad by selecting a certain number of Premier League footballers every week. Based on how these footballers perform in their real-life matches, the fantasy managers will earn points, with the objective of earning more points each week than their friends and the wider FPL player base.

Over the past several years the game's popularity has grown rapidly, transforming FPL from a geeky hobby into a global phenomenon. Content creators on YouTube and other social media platforms rake in hundreds of millions of views and, most relevantly to this project, paid services which rely on data analytics can charge upwards of £20 per month in exchange for giving advice to FPL managers [4] [5]. These models use historical data and advanced prediction algorithms to give an expected points score for each footballer every week. Creating accurate models for FPL has proven to be a manageable task; an analysis by one major model provider named 'FPL Review' found that the most popular models achieve more than a 95% accuracy score in player point prediction when compared with a theoretically perfect model [6]. Given that modelling seems to have been so well executed, it may appear as though the usefulness of data analytics to FPL has been fully realised - but it has not.

Even once you have a predicted points score for each player every week that you are confident in, the problem of selecting a collection of players every week is far from

straightforward. In FPL, there are several key constraints that all managers' teams must abide by; your squad must be a certain size, there is a certain budget you have to spend on your squad which cannot be exceeded, and each week you may only make one transfer without incurring a cost. These factors together transform the relatively simple modelling problem into a 38-week optimisation problem in which budget and transfer constraints must be abided by and navigated.

To demonstrate the extent of this selection problem, let us consider simply the number of possible ways a single squad of 15 could be selected. Taking budget constraints into account restricts the number of available players to roughly 400, and so the number of unique 15-player squads is approximately 400 choose 15 - more than 6×10^{26}. Once you also consider the fact there may be roughly 400 possible transfers per week, this brings up the total number of possible ways your team might develop over the season to more than 10^{121}. To put this in context, the number of atoms in the observable universe is estimated to be approximately 10^{82}, and the number of grains of sand on Earth is estimated to be around 10^{20} [7] [8]. Therefore, if we took all of the atoms in the universe and turned them into Earths, and then counted the grains of sand in all of them, we still would have a number less than a trillion million-th of the number of possible season-long team choices.

All of this is to say that the space over which an algorithm might try to pick an optimal team is far too large for a systematic, exhaustive search. Algorithms which go through large search spaces and aim to optimise an output subject to constraints can be applied to a myriad of commercial uses too, from portfolio management and stock selection to optimising the transportation of heavy cargo in shipping [9]. In this project, I will first use data from the 2023/24 Premier League season to formulate a rudimentary predicted points model, and this model will then produce a predicted points score for all players and weeks in the 2024/25 season. I will then seek to code and evaluate three different optimisation algorithms, which will be applied to this database of predicted points, with the aim of maximising the total season points of a team. To conclude, I shall compare the computational complexity and performance of the three algorithms and discuss the strengths and weaknesses of each.

Chapter Two

Creating the Point Prediction Model

In order to compare the performance of different optimisation algorithms in maximising the FPL points of a team over the course of a season, it is first necessary to create a model that predicts the points of each player each week. Using historical data from the 2023/24 season, I will use a simple model to predict the points score of each player in every week of the 2024/25 season.

In this chapter, I will outline the actions which earn football players FPL points, explain the data source for my prediction model, flesh out the rationale for my model's calculations and justify any simplifications I make.

2.1 How Players Score Points

In FPL, players can earn or lose points for a range of actions [10]. The most obvious and highest-scoring action is, of course, scoring a goal, but players also receive points for assists and how many minutes they play. In addition, players delineated as goalkeepers, defenders or midfielders are awarded points if they 'keep a clean sheet' - in other words, prevent the opposition from scoring during the entire game. Similarly, goalkeepers can earn points for the number of saves they make. Players can also lose points, for example if they receive a yellow or red card, miss a penalty or score an own goal.

The full list of point-scoring actions is slightly longer and more complicated than I have laid out, and the position of a player can affect the points they receive for certain actions. Given that my project is focused on the algorithms that seek to optimally select a team based on some database of predicted points, rather than the mechanism of predicting the points themselves, I will make several simplifications. These simplifications are made in order to keep my model practical to run for all of the players over the full 38-week season whilst also maintaining a strong predictive power. One example of a point-scoring action my model will neglect to include is goalkeeper save points, as they are extremely variable and hard to predict whilst also being low scoring. Similarly, after every match, three players earn bonus points depending on a complicated sum of their different actions

in the game. This sum would be computationally extensive to calculate for all 380 games over the season and bonus points are also relatively low scoring and infrequent, so they will also be neglected.

The following table details the actions which my model will predict and displays the number of points each type of player will earn for performing them.

Action	Goalkeeper	Defender	Midfielder	Attacker
Playing 90 Minutes	2	2	2	2
Yellow Card	-1	-1	-1	-1
Red Card	-3	-3	-3	-3
Clean Sheet	4	4	1	0
Assist	3	3	3	3
Goal	10	6	5	4

Table 2.1: The Points Scored by Each Type of Player
For Each Action Considered in the Model

2.2 Data Source

To create a model of expected points which has strong predictive power, the data source must contain extensive data regarding each of these point-scoring actions. For instance, I need to have data points for minutes, yellow and red cards, clean sheets, assists and goals. The data source I will be using in my predicted points model is an extensive online repository covering the 2023/24 FPL season. The data comes from the official FPL website and another mainstream analytics provider called Understat and has been collected and processed into a database by Vastaav Anand [11]. This database is suitable because it has statistics on every player in the league which are relevant to the key point-scoring actions. A sample of the database can be found in Appendix A.

2.3 Calculating Expected Points

Calculating the expected points scored for performing certain actions is relatively simple. For instance, the number of minutes a player gets per season tends to be fairly stable; if a player is a key member of a squad one season, he is unlikely to lose his position and be benched a lot the next season. Therefore, we can extrapolate that players will get approximately the same number of points per game for minutes as they did in the previous season. Similarly, the propensity of a player to get yellow and red cards is likely to be relatively constant when taken over the course of 38 games, and so the points from that

action can be extrapolated to the next season. The more challenging and interesting actions to predict are goals, assists and clean sheets.

2.3.1 Predicting Goals and Assists

In football analytics, it is challenging to predict the future goal-scoring behaviour of a team or individual if you just consider the previous goals they have scored. This is because football is a low-scoring game with extremely high variance; in two similar matches random factors might cause the number of goals scored to vary drastically, despite the total number of high-quality chances being the same. Therefore, statisticians developed a new metric to assess the quality of chances a team gets, rather than the end results. This metric is called 'expected goals' and it has gained prominence in mainstream football coverage in recent years [12]. Expected goals (also known as xG) is a metric which evaluates the likelihood of any single shot resulting in a goal, based on multiple factors such as the distance from the goal, the angle to the goal and the body part used. Each shot is given a rating between 0 and 1 which corresponds to its likelihood of being scored. This probability is not subjectively decided; rather it is derived from analysing thousands of shots taken in previous matches from similar positions. Thus expected goals provides a much more precise and reliable indicator of how many goals a team or player is likely to score in future when compared with the coarse and volatile measure of how many goals they have managed to score in recent games [13].

The corresponding metric to expected goals that deals with assists is unsurprisingly named expected assists (xA). A player earns some amount of expected assists when a pass they make is directly followed by a shot, and the amount of expected assists they earn for that pass is equal to the likelihood of the following shot being scored [12]. Therefore, a player who sets up a lot of high quality shooting opportunities earns a high amount of expected assists. This metric also depends on a multitude of factors such as the type and destination of the pass and is based on historical data.

Expected goals and assists are the key to accurate predictive models. They give a precise measure of both a player and a team's attacking threat, and by measuring the opposition's xG, the measure can indicate defensive capability too.

It would be easy to take the average xG and xA a player achieved per game in one season and use that for their xG and xA in every game in the next season, but that misses a key variable: the quality of an opposition team's defence. The xG and xA a player achieves varies significantly from game to game depending on the quality of the opposition. Therefore, to increase the specificity of the model, the amount of expected goals and assists a player is predicted to have in each game will be scaled up or down depending on the opposition's Defensive Weakness Multiplier.

A team's Defence Weakness Multiplier (DWM) is simply a metric I have created which

encapsulates how many high quality chances a team concedes compared to your average Premier League team. It is calculated with the following formula:

$$\text{DWM} = \frac{\text{Team's Total xG Conceded in a Season}}{\frac{1}{20} \times \text{Total xG Conceded By All Teams in the League in a Season}}$$

We will use the xG data from 2023/24 season to create these DWM values that will be used to predict points in the 2024/25 season.

We have now covered all of the point-scoring actions that a forward is eligible for. Multiplying the likelihood of different actions occurring with the number of points each action scores, we obtain the following formula for the number of points we predict a forward to score in each game:

$$\begin{aligned}\text{Forward Predicted Points} = [&2 - \text{xYC}_{90} \times 1 - \text{xRC}_{90} \times 3 \\ &+ (\text{xG}_{90} \times 4 + \text{xA}_{90} \times 3) \times \text{Opposition DWM}] \\ &\times \frac{\text{Average Minutes per Game}}{90}\end{aligned}$$

The subscript 90s within that formula represent a normalised value of how much of that action we expect the player to perform per 90 minutes. It should be noted that the xYC and xRC are not from any sophisticated statistical model, but rather from a simple calculation of how many yellow or red cards a player received per 90 in the previous season. We only choose to scale the points from goals and assists by the DWM, since we don't expect a player's minutes and yellow and red cards to be correlated with the opposition's defensive quality. You can see that the structure of the formula is:

$$(\text{Points per Action} \times \text{Expected Actions per 90}) \times \frac{\text{Average Minutes per Game}}{90}.$$

It is useful to structure the formula like this because all of the dependence on how many minutes the player is expected to play is all encapsulated in the final term. That allows us to use more generalised 'per 90' metrics within the formula for all players, regardless of how many minutes they play.

2.3.2 Predicting Clean Sheets

A coarse way to predict how many points a player should get per week from clean sheet points would be to take the number of clean sheets they achieved per 90 in the previous season. As a first-order approximation, this is fine, but just as in the case of predicting goals and assists, this neglects to take into account the quality of the opposition. In a

similar way to how we defined the Defence Weakness Multiplier of a team, we will now define the Attack Strength Multiplier (ASM). This is given by the formula:

$$\text{ASM} = \frac{\text{Team's Total xG Created in a Season}}{\frac{1}{20} \times \text{Total xG Created By All Teams in the League in a Season}}$$

It is a multiplier which essentially tells us how many more expected goals a team creates when compared with the average team in the league. By including this value within our calculations of clean sheet likelihood, we can personalise our predictions based on the opposition a player is playing against. We expect a defender to be less likely to keep a clean sheet against an opponent with a higher ASM, and so we must divide by the ASM within our calculations.

In a similar way to how xYC and xRC are not statistical models but rather a simple calculation of how many yellow or red cards a player earned last season per 90 minutes they played, we will use the metric xCS per 90 (expected clean sheets per 90) to represent the number of clean sheets a player earned last season per 90 minutes they played.

This metric is not as precise and reliable as the other metrics within the formulae as it does not take the quality of chances conceded into account. Instead, it just focuses on the number of clean sheets that were achieved in the previous season. This number is prone to variance as teams can get lucky or unlucky within games due to random factors. This variance will be minimised by the fact that this metric is built on 38 games' worth of data, but I feel it still could be improved if this project was to be extended. One idea might be to model the conceding of goals with a Poisson distribution [14]. This strikes me as valid as we have an estimation for how many goals we concede per game on average, and given this, a Poisson distribution can tell us the probability of k goals being conceded in this time frame. The assumption that goals occur independently of the time since the last goal is potentially debatable and a further examination might evaluate whether this is plausible. Nevertheless, since this project is focused more on the optimisation algorithms themselves, this discussion will be left for a possible extension elsewhere.

Now we have a full understanding of how to predict each point-scoring action, we can list the formulae for the predicted points of the goalkeepers, defenders and midfielders.

$$\begin{aligned}\text{Goalkeeper Predicted Points} = [&2 - \text{xYC}_{90} \times 1 - \text{xRC}_{90} \times 3 \\ &+ (\text{xG}_{90} \times 10 + \text{xA}_{90} \times 3) \times \text{Opposition DWM} \\ &+ \text{xCS}_{90} \times 4 \div \text{Opposition ASM}] \\ &\times \frac{\text{Average Minutes per Game}}{90}\end{aligned}$$

$$\begin{aligned}\text{Defender Predicted Points} = [&2 - \text{xYC}_{90} \times 1 - \text{xRC}_{90} \times 3 \\ &+ (\text{xG}_{90} \times 6 + \text{xA}_{90} \times 3) \times \text{Opposition DWM} \\ &+ \text{xCS}_{90} \times 4 \div \text{Opposition ASM}] \\ &\times \frac{\text{Average Minutes per Game}}{90}\end{aligned}$$

$$\begin{aligned}\text{Midfielder Predicted Points} = [&2 - \text{xYC}_{90} \times 1 - \text{xRC}_{90} \times 3 \\ &+ (\text{xG}_{90} \times 5 + \text{xA}_{90} \times 3) \times \text{Opposition DWM} \\ &+ \text{xCS}_{90} \times 1 \div \text{Opposition ASM}] \\ &\times \frac{\text{Average Minutes per Game}}{90}\end{aligned}$$

Using these formulae in conjunction with the fixture list, we are able to produce a database of predicted points for each player and each week of the 2024/25 season [15]. A sample of the model's predicted values can be found in Appendix B.

2.3.3 A Note on Promoted and Relegated Teams

At the conclusion of every Premier League season, the three teams with the fewest points are relegated into the division below, and the three teams with the most points in the division below are promoted. A problem this poses to our model is that our database does not contain statistics on these promoted teams or on their players. Consequently, this model will not evaluate the suitability of any promoted players, and they will be ignored in our discussions. Although this may seem drastic, in the context of FPL, the players of promoted teams tend to rarely be chosen by FPL managers. This is because promoted teams often struggle significantly with the quality of the Premier League, and consequently, their players score few goals and keep few clean sheets. There are exceptions to this rule, but generally, promoted players tend not to be good FPL choices, and as such, this omission is not a significant problem for our model.

That being said, it is still necessary to consider the Attacking Strength and Defensive Weakness Multipliers of the promoted teams for our calculations of other players' predicted points. For a placeholder value, we will assume the promoted teams have the DWM and the ASM corresponding to the lowest values of a team in the previous season. This reflects the fact that the promoted teams tend to be low quality and also gives us relatively accurate estimated values for them.

Chapter Three

The Greedy Algorithm

Before diving into the specifics of the three algorithms this project will discuss, I would first like to lay out the constraints that any valid FPL teams must abide by. This is a critical consideration as it informs the implementation of all three algorithms. In order to make the game a little more challenging, FPL imposes rules which dictate the composition of your team in certain ways. In typical FPL, you must own two goalkeepers, five defenders, five midfielders and three forwards. During any one week, you must bench four players, meaning the points they score during that week don't add to your team's total. In addition, each player costs a certain pre-defined amount of money and the cost of your squad must not exceed your budget of £100m. At the end of every week, you get one transfer which you can use to swap one player in your squad for one not in your squad, so long as it fits your budget constraints. Lastly, you may not own more than three players from any one Premier League team.

In this project, I will be including all of these constraints, just with some minor alterations to aid in the practicality of my code whilst still keeping the core challenges of FPL. The main alteration is that I will exclude considering the 4 bench players. The reason is that the bench players are typically low-cost, low-scoring and they feature in the 'starting eleven' rarely. As such, their contribution is minimal. Additionally, adding the benching and squad selection mechanism of FPL to my code would be computationally extensive whilst also negligibly impacting the evaluation and comparison of the optimisation algorithms. Consequently, I have chosen the following constraints for the optimisation algorithms to follow:

- The squad must have 11 players: one goalkeeper, four defenders, four midfielders and two forwards
- The total budget must not exceed £80m. This is slightly reduced from £100m to reflect the reduction in squad size.
- One transfer may be made at the end of each week

- You must not have more than three players from any one Premier League team

3.1 Introduction to the Greedy Algorithm

An algorithm is called 'greedy' if it aims to achieve its global objective by making the locally optimal choice at every decision point [16]. The only factor that greedy algorithms take into account when evaluating their choices is the immediate benefit that the choice contributes towards their overall goal. Greedy algorithms neglect the longer-term consequences of their actions, and as such, they are unable to plan ahead and manage constraints in a manner which might lead to more optimal solutions. Typically, greedy algorithms are prevented from back-tracking and so these locally optimal decisions are final.

Despite the apparent short-sightedness that this approach entails, in certain situations, greedy algorithms can not only provide good solutions, but they can actually provide the optimal solution. Problems whose optimal solution emerges by simply taking the best locally optimal choice at every step are said to have the 'greedy-choice property' [17]. A common example of a problem which can be optimally solved by using a greedy algorithm is Dijkstra's algorithm [18]. This algorithm is used for finding the shortest path between two nodes in a weighted graph, a highly suitable problem for representing transportation networks for instance. Dijkstra's algorithm takes a greedy approach because at every decision point, it selects the next closest unvisited node in order to minimise the distance. It is a suitable problem for the greedy algorithm because it also has an 'optimal substructure' [19]. A problem has an 'optimal substructure' if it can be broken down into sub-problems and if the global optimal solution is simply the combination of the optimal solutions to all of the sub-problems. We can see that this shortest path problem has this structure by considering how to find the shortest distance from node A to C via B; the shortest way to go from node A to C via B necessarily involves going along the shortest path from A to B and then the shortest path from B to C, and hence the optimal global solution is the combination of the optimal solutions to the sub-problems.

A greedy algorithm has both advantages and disadvantages when applied to optimising an FPL team over the course of a season. The main advantages this algorithm has are its simplicity of implementation and its speed of execution. By virtue of the fact it does not consider the multitude of consequences its decisions might have in the future, the time complexity of greedy algorithms tends to be low, typically O(n) or O(n log n), where n is the number of objects under consideration [20].

However, there exist clear disadvantages for this approach. Optimising an FPL team for short-term gains necessarily means that the long-term consequences of the choices are overlooked. Players which have difficult fixtures coming up and hence reduced points potential might still be transferred in just because they have a single high-scoring week next. Similarly, the algorithm will fail to plan its budget and may be unable to transfer in a

certain player in a week where they score a lot of points. This overlooking of consequences and lack of long-term strategy makes this algorithm unsuitable for finding the optimal solution in a game with so many constraints.

3.2 Implementing the Algorithm

Applying the greedy algorithm to any problem involves making the locally optimal decision towards the objective at each stage. In the context of FPL, the global objective is to maximise the total points scored over the course of the season. Hence, our greedy algorithm will aim to add the most points to the team at each possible stage.

Throughout the season, an FPL manager must make two key sets of decisions. First is the selection of the initial team for the first round of fixtures, known as Gameweek 1 (GW1). This initial squad selection is crucial because after GW1, only one player can be transferred out per week, and consequently, many players in the initial team will remain for weeks. The second key set of decisions is the weekly transfers. There are 37 possible transfers over the course of the season, and they will dictate how the team evolves over time.

The pseudocode for the greedy algorithm's approach to these two key sets of decisions is outlined below:

Selecting the Initial Team:

1. **Create Constraints**
    - Start with a budget of £80m (represented as 800 units in the database)
    - Make sure there is only one slot for a goalkeeper, four for defenders, four for midfielders and two for forwards
    - Define a limit of three players per Premier League team
2. **Choose Players**
    - Sort the whole database of players by their GW1 predicted points score
    - While the squad still has empty slots, add the highest non-chosen GW1 point scorer to the team
    - Only allow this choice if it abides by the team, budget and position constraints
    - Update a counter of how many slots remain in each position, how many players from each team have been chosen and how much budget is left
    - Ensure that every selection leaves enough budget for the team to fill up with 11 players

Making Weekly Transfers:

1. **Selecting the Transfer**

    - Choose a player already in the team
    - Find the player with the highest predicted points for next week which could be afforded by transferring out this player, whilst meeting the constraints
    - Calculate the points differential between the player already in the team and the potential replacement
    - Do this for all players currently in the team and make the transfer with the greatest positive points differential

The code for this greedy algorithm can be found in Appendix C.

Player	Position	Gameweek 1 Points
Caoimhin Kelleher	GKP	0.85
Gabriel Magalhães	DEF	4.64
William Saliba	DEF	4.61
Illia Zabarnyi	DEF	3.01
Jamaal Lascelles	DEF	1.50
Mohamed Salah	MID	6.18
Bukayo Saka	MID	5.44
Son Heung-min	MID	4.84
Anthony Gordon	MID	4.15
Erling Haaland	FWD	4.72
Divin Mubama	FWD	0.06
Total		**40.00**

Table 3.1: The Greedy Algorithm's Initial Squad Selection

Metric	Value
Total Season Points	1683.20
Initial Squad Value	£80m
Final Squad Value	£77.5m
Average Squad Value	£78.3m
Total Function Calls	50.8 million
Total Execution Time	12.9 seconds

Table 3.2: A Summary of the Greedy Algorithm's Season

A function call is a discrete bit of computation performed by the code and the number of function calls can be easily tracked within Python. This gives a reliable proxy for the computational effort used in the running of some code [21].

A full analysis of these results will be done in Chapter 6 so that they can be contrasted with the results from the other two algorithms.

Chapter Four

The Knapsack Algorithm

The key drawback of the greedy algorithm is its short-sightedness. Its focus on optimising the points score of the next round necessarily means that it neglects combinations of players which might be more optimal over the longer-term. It is this tendency which motivates us to look for an algorithm which takes more than one week into account. The first such algorithm we will construct is based on an algorithm that was produced to solve the knapsack problem.

4.1 Introduction to the Knapsack Problem

Imagine having a knapsack (a hiking bag, in other words) and a range of different objects that you might want to take with you somewhere. The objects each have a certain weight and a certain value, and your knapsack can only carry so much weight. The optimisation problem of trying to fit the maximum amount of value within your knapsack without exceeding the weight limit is known as the 0/1 knapsack problem [22]. The 0/1 in the name stems from the fact that each object can either be chosen once or not at all; no multiple or fractional copies of the same object are allowed.

This problem is clearly analogous to our problem of selecting the optimally scoring GW1 team in FPL because we have a range of different objects with different values and costs associated, we have a limit on the total cost of our selections and we may only pick a player once at most.

The algorithm I will use for solving the knapsack problem will take a dynamic programming approach. Dynamic programming is the name given to any program which takes a problem and splits it up into sub-problems. Once it has solved the sub-problems, these solutions are used recursively to build up a solution to the main problem [23].

In this chapter, we will first use a recursive, dynamic programming approach to find the optimal GW1 team. Then, we will extend our knapsack algorithm to incorporate transfers and seek to optimise the total season points score. In the process of optimising our total season score, we will determine the optimal number of weeks for the knapsack

algorithm to consider in its time horizon. It should be noted that the knapsack algorithm can provably solve the GW1 problem, but we cannot be sure whether its output for the extended season problem will be optimal or just a relatively high-scoring solution. This uncertainty is due to the time-varying aspect introduced by the transfer process.

4.2 Implementing the Algorithm for the Initial Team

The general idea of the recursive knapsack algorithm works by defining a function $m[i, w]$, which gives the maximum value we can get by considering the first i objects that are available and having a maximum weight constraint of w. If we had x objects and a maximum weight of y, our aim would be to find $m[x, y]$. The way we do this is by building up a 2D table of all of the $m[i, w]$ values recursively [22].

Let's consider the following example where we have 4 objects, each given an identity number i, each with a value of v and a weight of w. If our knapsack has a capacity for 6 units of weight, the solution to our problem is $m[4, 6]$ - the maximum value when we consider all 4 objects with maximum weight capacity of 6 units.

i	v	w
1	5	4
2	4	3
3	3	2
4	2	1

Capacity=6

		w						
		0	1	2	3	4	5	6
i	0							
	1							
	2							
	3							
	4							

Figure 4.1: The Unfilled Table for the Knapsack Algorithm [24]

To find $m[4, 6]$, we can fill up this table recursively with the following formula:

$$m[i, w] = \begin{cases} 0 & \text{if } i = 0 \text{ or } w = 0 \\ m[i-1, w] & \text{if } w_i > w \\ \max\left(m[i-1, w],\ m[i-1, w-w_i] + v_i\right) & \text{if } w_i \leq w \end{cases}$$

We can intuitively interpret what each of these three elements means as follows:

1. Set an entry in the table to 0 if it corresponds to a total weight limit of 0 or if the number of items being considered is 0.

2. If the weight of the i-th object is greater than the current weight constraint w, then our maximum value is the same as if we didn't include the i-th object at all:

$$m[i, w] = m[i-1, w] \quad \text{if } w_i > w.$$

3. If the weight of the i-th object is less than or equal to the current weight constraint, we consider whether including it improves the maximum value. We compare the maximum value if we exclude the item with the maximum value if we include it. This is a comparison of $m[i-1, w]$ versus the sum of v_i and the best value achievable with the remaining weight, $w - w_i$:

$$m[i, w] = \max\left(m[i-1, w],\ m[i-1, w-w_i] + v_i\right) \quad \text{if } w_i \leq w.$$

Filling the table up row by row using this formula eventually yields the following filled table:

i	v	w
1	5	4
2	4	3
3	3	2
4	2	1
Capacity=6		

i \ w	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	0	0	0	5	5	5
2	0	0	0	4	5	5	5
3	0	0	3	4	5	7	8
4	0	2	3	5	6	7	9

Figure 4.2: The Filled Table for the Knapsack Algorithm [24]

From the table, we can see that our maximum value for the weight limit of 6 units is 9. We can work out that this corresponds to selecting items 2, 3 and 4.

Applying this to finding our optimal GW1 team, we might try and fill up a similar table with the number of players under consideration as our i value and the total budget constraint of our team as the w value. The only problem is that we also need to incorporate our other constraints, such as the number of players in each position, the total squad size and the number of players from each Premier League team. Including these additional constraints provides a considerable challenge to our dynamic programming approach.

The way to overcome this challenge is to introduce new parameters to our objects, turning our problem into a multidimensional knapsack problem [9]. The term 'multidimensional' in that title simply describes the fact that each object we are examining now not only has a value and a cost, but also some other characteristics which must satisfy a constraint. For our situation, we want our solution to satisfy the constraints that there is 1 goalkeeper, 4 defenders, 4 midfielders and 2 forwards. In addition, we want no more than 3 players per Premier League team.

If we were to add a constraint for the number of goalkeepers, defenders, midfielders and forwards, along with a constraint for the number of players in every premier league team, this would correspond to adding 24 new dimensions to the system. If we consider turning an originally 2D grid into a 26D grid, we will have added 22 dimensions that have 4 possible values, 1 dimension with 2 possible values and another dimension with 1 possible value. A back-of-the-napkin calculation suggests that filling out this grid might

take around 10^{13} times longer than it would without these constraints [25]. Clearly this is unfeasible.

To address this, we aim to reduce this number of dimensions massively by considering redundancy. We obviously can't get rid of the position counters, but it does seem likely that many of the team counters will end up being unnecessary considerations; how many players from Crystal Palace or Bournemouth or Fulham do we really expect to be in our GW1 team, considering the fact their teams score few goals and allow the opposition to score many goals? Based on this, we first run the dynamic programming knapsack algorithm without any constraints on how many players from each team we take. Then we see which teams, if any, show up more than three times in our squad. Then we include those teams in our constraints and add them as dimensions, and run it again. We do this until our initial GW1 squad doesn't break our rule of 3 per Premier League team, and then we have successfully found the optimal team without introducing unnecessary dimensions.

So, we set up our recursive function of all of these parameters (except for number of players per team) and run the Python code. We find that the number of operations required is far too many to practically compute. We can intuitively understand this by considering the number of possible states in this multidimensional space. If we consider the budget ranging from £51.5m (the lowest possible) up to £80m in £0.1m steps, that's 286 possible values. If we look at all 475 players plus the case of examining none of them, we have 476 possible values. Additionally, we can have 0 or 1 goalkeepers, between 0 and 4 defenders and so on. Multiplying all of these dimensions together, we have state space that has more than 20,000,000 entries. This is too computationally extensive to run, so one again we consider how to make some simplifications.

Since we are only trying to find the optimal GW1 team here, we can confidently cut down the number of players we consider in our recursive programme. This idea is called pruning, and allows us to reduce our search space by removing unnecessary elements [26]. To have a high level of confidence that the players we are getting rid of won't be important, we choose to remove players with low GW1 predicted points scores and also with bad points to cost ratios. This ensures that we don't remove any players who could be big point scorers or useful bargains for our GW1 team.

We prune our dataset to only include the top 5 goalkeepers in GW1 points and top 5 goalkeepers in GW1 points to cost ratio, top 20 in defenders and midfielders and top 10 in forwards. The ratio of these numbers reflect the ratio of these positions in our 4-4-2 team formation.

When we run our knapsack algorithm on this pruned dataset, we see that the only team which is represented more than 3 times is Arsenal. Thus, we include Arsenal within our constraints and run it again. Once we've done that, we see our optimal GW1 team according to the recursive knapsack algorithm is the following:

Player	Position	Gameweek 1 Points
Jordan Pickford	GKP	3.31
Gabriel Magalhães	DEF	4.64
William Saliba	DEF	4.61
Pedro Porro	DEF	4.13
James Tarkowski	DEF	3.72
Mohamed Salah	MID	6.18
Bukayo Saka	MID	5.44
Son Heung-min	MID	4.84
Brennan Johnson	MID	3.74
Alexander Isak	FWD	4.42
Dominic Solanke	FWD	3.88
Total		**48.91**

Table 4.1: The Recursive Knapsack Algorithm's Initial Squad Selection

Metric	Value
Total Function Calls	439 million
Total Execution Time	91.7 seconds

Table 4.2: The Recursive Knapsack Algorithm's GW1 Selection Performance

We can be confident that this is not only a good estimation but actually the exact optimal GW1 team, so long as we trust that none of the players who were pruned out of consideration might have been used. We can take confidence that this team is indeed the exact optimal GW1 team by decreasing the amount of pruning we did.

If we decrease the amount of pruning we do so that we actually consider the top 10, 40, 40 and 20 players in each position and in points and points per cost ratio respectively, we notice we still get the same optimal GW1 team. Therefore, we can have high confidence that the team is exactly optimal.

The code for this GW1-focused implementation can be found in Appendix D. There is one key difference between the code's process and the idea of filling up a multi-dimensional array from the bottom up, as shown in Figure 4.1. The key difference is that the code implements a process known as memoization [27]. Rather than filling up the table from the bottom up, the code fills up the table from the top down. In addition, it doesn't actually fill up all of the squares in the grid either. It only fills up the squares which are relevant to the branching decision tree. This approach of only filling grid entries which are necessary for our final answer reduces the amount of calculations we need to do whilst maintaining the integrity of our answer.

4.3 Implementing the Algorithm for the Whole Season

The recursive knapsack algorithm is particularly suited to finding the highest scoring GW1 team, but when we extend our problem to a whole season, the time-varying aspect presents a challenge. We need to consider a way of choosing transfers which is more long-term focused than the greedy algorithm.

To do this, I consider a new approach. Instead of just optimising the GW1 score and then finding transfers afterwards like the greedy algorithm does, the approach will be to optimise the team's score on a rolling k-week basis. What I mean by this is the following: to pick the initial team, find the team which has the highest total score for weeks 1 to k. This will be solved by a recursive knapsack algorithm. Then, for the transfer before GW2, we iterate through our players and search for the transfer which would most increase our team's predicted points score in weeks 2 to k+1. Then, before GW3, we loop through our new team and transfer in the player which will most increase our total predicted points score in weeks 3 to k+2. This transfer strategy is quasi-greedy, though it has a longer-term focus, so this season-long approach is a sort of knapsack-greedy synthesis.

The motivation behind this approach is as follows: we want to ensure our team is focused on more than just the next week. By looking ahead to the next k weeks, we can have a more long-term view.

A key question which emerges from this approach is which k-value is optimal? In other words, how many weeks ahead should we consider a player's performance in the process of choosing transfers? Since the team can make one transfer per week and there are 11 players in the team, an entire team could be changed every 11 weeks if we so wanted, and as such, a suitable suggestion might be that we should look 11 weeks ahead. In counterpoint to this, some extremely valuable players such as Cole Palmer or Mohamed Salah are unlikely to ever leave the team, and as such, there may only be a few members of our team who ever get transferred out and so we only need to look a short time ahead.

These arguments suggest the right time horizon lies somewhere between 1 week and 11 weeks ahead. To analyse this and ensure our approach scores the maximum season-long score, we will run this approach for all k values from 1 to 11 and analyse the results.

Interestingly, as a side note, a time horizon of 1 corresponds exactly to the transfer process of the greedy algorithm. In fact, in the case where we choose k=1, we recreate the greedy algorithm except that it starts with the optimal GW1 team. This is fortunate as it also allows us to evaluate how much of a problem the initial squad selection of the greedy algorithm was to its full season performance.

Having run the code with time horizons of 1 to 11 weeks, we obtain the following results for the total season score:

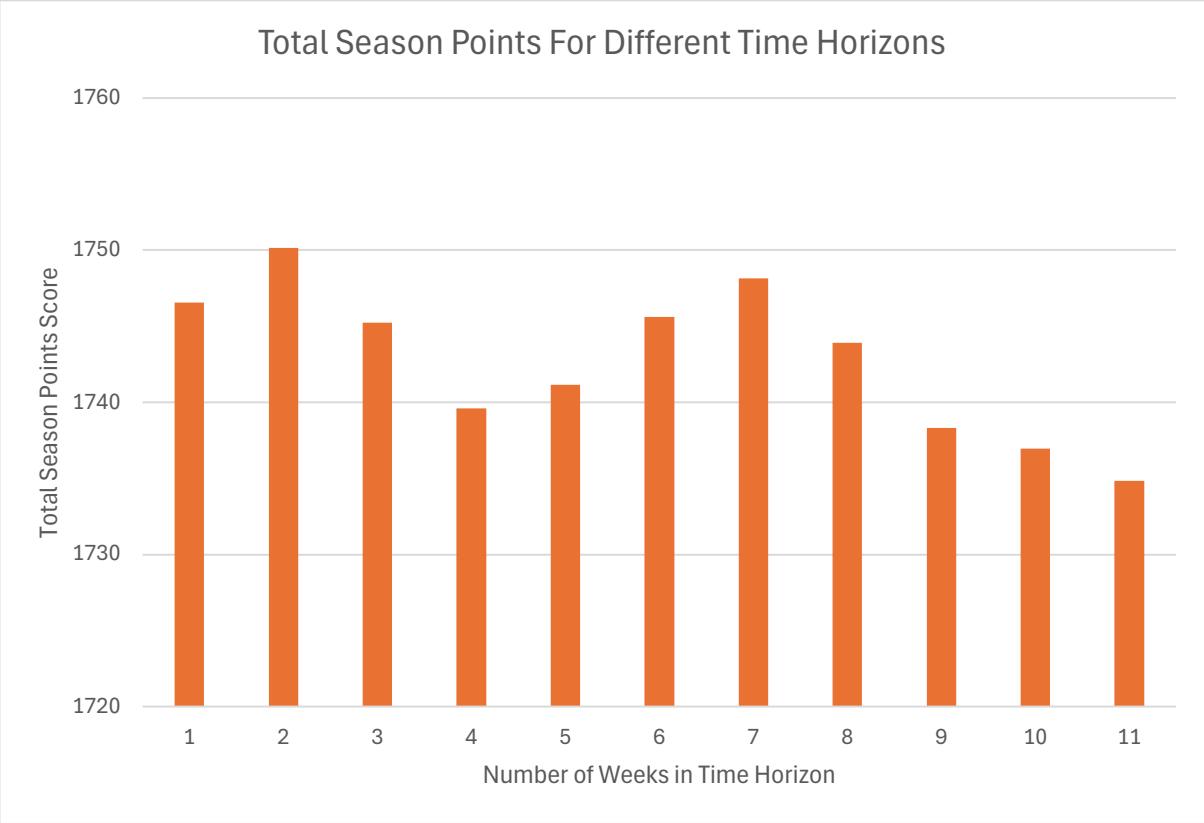

Figure 4.3: The Points Score of the Knapsack Algorithm For Different Time Horizons

Looking at this graph, we can see that the the total season points scores fell in a rough range of 1730 to just over 1750. The peak total season points score was 1750.14, corresponding to a time horizon of 2 weeks. Close behind was the time horizon of 7 weeks with a season score of 1748.13 points. The lowest point score was 1734.86, corresponding to the largest time horizon of 11 weeks. In fact, the three lowest scoring seasons were for the three largest time horizons.

After running this code, I was struck by just how flat the distribution is; from the highest to lowest point scores, the season score only varies by around a percent. I would have guessed that there would be a unimodal distribution, approximately around $k = 6$, with quite a high drop off at the edges. I expected this peak to be around 6 because I expect some players to remain in the team the whole season due to their high points to cost ratio, and so we don't need to be thinking very far ahead to consider how to swap out all of the players we may want to change.

The relatively flat distribution of total season points score against time horizon suggests that the knapsack algorithm is not overly sensitive to the value of k, provided it takes a reasonable value of approximately $2 - 7$. In this range, the algorithm is putting together a team of very strong players whilst striking a balance between short-term adaptability and long-term preparation. Those teams with smaller k values may be capitalising on particularly high-scoring moments for individual players while neglecting to plan ahead enough. Those with larger k values are likely to be having more consistent point scores,

but failing to adequately realise the value of a short-term player pick. I hypothesis that these effects are potentially balancing out, leading to the flat distribution.

With regards to why the points scores drops off for very large k values, I suspect that at these large k values we are becoming insensitive to important fixture changes in the near future. A long time horizon of $k = 10$ is enough time for difficult fixtures and easy fixtures to roughly cancel out and so our transfer strategy basically becomes fixture insensitive. Consequently, we end up constantly swapping high-quality players at essentially random times based on whether or not they are in the team at that time. This loses the benefit of having transfers at all.

I believe that the simplified version of Fantasy Premier League which I am discussing in this project makes it such that the gains from planning ahead are diminished. Supporting this argument is the fact that in typical FPL, your squad is made of 15 players, and you can choose any 4 of them to be on the bench in any week, meaning the points they score aren't counted. This allows you to pick different combinations of players which can navigate the fixture schedule more successfully, and introduces even more complexity in planning ahead. This partly helps to explain the flatness in this distribution of total season points.

If we take the k value which strikes this balance best, the results for this algorithm is as follows:

Metric	**Value**
Total Season Points	1750.15
Total Function Calls	1.31 billion
Total Execution Time	281 seconds

Table 4.3: The Recursive Knapsack Algorithm's Total Season Performance for k = 2

Chapter Five

The Genetic Algorithm

5.1 Introduction to the Genetic Algorithm

This chapter draws upon the introduction to genetic algorithms laid out in the textbook 'Genetic Algorithms - Principles and Perspectives' by C. Reeves and J. Rowe [28].

In 1859, Charles Darwin published his book 'On the Origin of Species', and introduced his concept of evolution by natural selection to the world [29]. He presented natural selection as the mechanism by which traits that aid an organism's survival become increasingly prevalent within a population over time.

There are three key processes which enable natural selection to produce increasingly well adapted organisms: mutation, crossover and selection.

A mutation is an occurrence where a small part of an individual's genetic code is randomly altered, introducing a new trait to a population.

In crossover, instead of random changes creating new genetic code, individuals within the population combine some of their existing genetic code in their production of a new offspring. As our understanding of DNA and genetics has improved over time, so too has the importance with which biologists regard crossover to the evolutionary process. One leading evolutionary biologist Ernst Mayr has called it 'the primary source of genetic variation', surpassing that of mutation [28].

Finally, in selection, we see that individuals less suited to the environment are less likely to survive and reproduce. This means their genetic information is prevented from being passed to the next generation. Consequently, we observe an increase in the concentration of well-adapted genes within the population over time [30].

Inspired by this process, genetic algorithms use concepts of mutation, crossover and selection to obtain highly-satisfactory solutions to complex optimisation problems [31]. Translating the language of evolutionary biology into our mathematical language, we consider any valid solution to an optimisation problem as an organism within a population. (Since individuals are characterised by their chromosomes, in genetic algorithms, the

word chromosome is often used interchangeably with the word organism. I will use the word organism here for ease of understanding). How good the solution is at solving the optimisation problem is akin to how well adapted an organism is to survive in its environment. A genetic algorithm takes a large number of trial solutions as an initial population and then exerts the processes of mutation, crossover and selection on those solutions over many generations. At the end of the process, the best performing solution is extracted and used as a high-quality solution to the problem.

A template for a genetic algorithm is shown below in Figure 5.1 [28]. The termination condition might correspond to reaching a target score you want your optimisation solution to achieve or a number of generations you want to run. The crossover and mutation conditions are typically probabilistic factors which control how many organisms to combine or mutate per generation.

```
Choose an initial population of chromosomes;
while termination condition not satisfied do
    repeat
        if crossover condition satisfied then
        {select parent chromosomes;
        choose crossover parameters;
        perform crossover};
        if mutation condition satisfied then
        {select chromosome(s) for mutation;
        {choose mutation points;
        perform mutation};
        evaluate fitness of offspring
    until sufficient offspring created;
    select new population;
endwhile
```

Figure 5.1: A General Template of a Genetic Algorithm [28]

Before putting this theory into practice and running this type of algorithm on an FPL team selection problem, some discussion of the general strengths and weaknesses of genetic algorithms might inform our implementation.

Starting with the strengths of genetic algorithms, the predominant advantage is that they can cover many different options in the search space [32]. By virtue of the vast number of possible player combinations, the number of valid FPL teams is vast. Consequently, the more player combinations that an algorithm can cover, the more likely it is that it will reach some high-scoring outcome. In contrast to the greedy and knapsack approaches, a

genetic algorithm will start with many different valid teams, and generate many more, giving it a unique advantage in this aspect.

Another advantage of genetic algorithms is that they can produce more unique and creative solutions to problems than other approaches. This is because they avoid getting stuck in locally optimal solutions like the greedy algorithm does by adding in random mutations.

This advantage supports the idea that the genetic algorithm might be able to build on the work of the knapsack approach in the previous chapter. In the knapsack approach, we had to choose some fixed time horizon of weeks over which our team would try and optimise its team's score. One drawback of this was that this time horizon was fixed, and so the team was not able to think short term when the long-term state of the team was good, nor could it think long-term when the short-term state of the team was good. We would expect that the optimal season-long score would come from the approach that dynamically adjusts its time horizon depending on whether the upcoming fixture difficulties are imminent or further down the line. In this vein, the genetic algorithm doesn't have a fixed future time horizon, and so the best performing solutions it outputs are likely to dynamically think on short-term and long-term time horizons at the right points of the season.

One last advantage of the genetic algorithm is that it enables parallel processing [32]. Parallel processing is a practical implementation technique in computing where one large compute problem can be broken down into separate problems which do not depend on each other. Consequently, these problems can be distributed to different processors and executed in parallel, vastly speeding up the execution time [33]. In the genetic algorithm, there is no sequential order in which different solutions in the population must be dealt with, and thus parallel processing can be done here. This separates this algorithm from the greedy and recursive methods which necessarily require sequential processing of either a single solution or a single iterative function.

The key weakness of the genetic algorithm is that it is computationally extensive to run. Even if different solutions are evaluated in parallel, the algorithm may require a large intial population and many generations to start obtaining high-quality solutions.

Another weakness to consider is that the genetic algorithm gives no guarantee of optimality. Unlike the knapsack algorithm which could find an optimally scoring team for GW1, the genetic algorithm cannot for certain know when it has found an optimal outcome. Tied to this is the dependence of the performance of the genetic algorithm on the parameters we use for how often mutations and crossovers occur. Finding values for these parameters which give high scoring solutions might take a lot of experimentation.

5.2 Adapting the Algorithm for FPL

Let's now apply this algorithm to the problem of optimising the total points score of a Fantasy Premier League team over the course of a season.

The organisms which we will be mutating and breeding to produce new generations will be valid solutions to the FPL problem; in other words, an organism here is an initial team along with a transfer strategy for the season.

The first thing we must do is create an initial population of solutions. Instantly, we are confronted with the problem of how large to make our initial population. A larger population would span a larger number of possibilities, but also would be more computationally extensive whilst also potentially being redundant if we have lots of similar teams within it. It turns out that a range of studies have suggested that population sizes of around 30 are adequate in most cases [28]. For our purposes, we will adjust our initial population size in the ballpark of this figure, and examine how this number affects our final result.

A related question is whether it is optimal to seed our initial population with high-quality solutions that we have obtained from previous methods; in our case, that would be by using our known solutions from the knapsack algorithm over the time horizons of 1 up to 11 weeks, which all score very highly. Views differ in regards to whether seeding high-quality solutions into the initial population is a good idea. This is because seeding good-quality solutions might cause the algorithm to converge too quickly on a specific form of solution which might not be optimal. That being said, others have found that seeding good solutions helps genetic algorithms find better solutions much faster than from a random start [28]. For the sake of completeness, we will run one simulation with a random starting initial population, and another with a large number of good quality solutions seeded in.

Having built our initial population, we now want to apply mutations and crossover to it.

The mutation aspect of a genetic algorithm aims to introduce variation into the population. Applying this to FPL, this corresponds to taking an existing solution and randomly swapping out one player for a valid replacement. It should be noted that, just like in real evolution, it is not necessary that this mutation always needs to be beneficial; so long as the mutation is beneficial sometimes, the dynamics of natural selection will ensure that good mutations are often propagated forward through the generations. Applying this to FPL is challenging because mutating one player selection has knock on effects for future transfers. In writing my code, I produced a lengthy pseudocode structure for this mutation process to overcome this problem. The pseudocode is available in Appendix E.

The crossover aspect takes two parent organisms and combines them somehow to produce new unique offspring. In an FPL context, this corresponds to taking two valid

solutions and taking elements from each to produce a new solution. Implementing crossover in this FPL context might be challenging as it requires finding a point within a season where two different solutions have the exact same team. Once two solutions have been found to have the same team one week, we can switch their subsequent transfer strategies, producing two new offspring that are a recombination of their parents. I anticipate that for a randomly generated population, finding these crossover points will be vanishingly rare, but when they are found, they may produce a drastically improved offspring solution.

Finally, we come to the selection aspect. At the end of one iteration, we now have a larger number of solutions than we started with, because we've created new ones with mutations and crossover. We thus want to cut down our population again by selecting solutions based on their performance. The selection aspect of our genetic algorithm will act with respect to the total season points score; this is our measure of how well our organism solves the optimisation problem, and hence is our analogue to the suitability of the organism to its environment. After each generation, we want to ensure that the best solutions propagate to the next generation, but we also don't want to be so harsh that we kill off all the diversity which might become fruitful later on.

One common way to do this is known as the roulette-wheel method [28]. In this method, each solution within the population is given a probability of being selected proportional to its fitness (in our case, its total points score). During the selection stage, we select solutions without replacement until we have the number of solutions we want to have in our next generation. This method ensures that better solutions are more likely to get through, whilst still maintaining diversity throughout generations.

One thing to add to the roulette wheel selection process is a concept known as elitism [28]. Elitism is where a very small number of the absolutely highest scoring organisms within the population are guaranteed to go through into the next generation, and are shielded from the roulette wheel. This can stop the population from undesirably reducing its maximum fitness overtime, and it provides a fail-safe to prevent the roulette wheel's randomness being too damaging to progress. In our algorithm, I have chosen to keep the top 5% of the solutions immune from the roulette wheel selection.

The final termination condition of this whole process is the number of generations we wish to run.

5.3 Implementing the Algorithm

5.3.1 Starting With a Random Population of Solutions

To evaluate the effectiveness of the genetic algorithm and to observe the significance of seeding the starting population with strong solutions, we first use a randomly generated starting population.

Alongside our analysis of the usefulness of seeding, I also felt interested to see how the number of generations and size of the initial population affected the score of our best-performing solution. As such, in our analysis we will run the genetic algorithm for a large number of generations so we can see how the output of the algorithm changes through the generations. We will also vary the size of our initial populations, using sizes of 20, 30 and 40 as well. These values balance computational complexity whilst also being in the range that should give adequate genetic diversity.

The code for running the genetic algorithm with randomly generated teams can be found in Appendix F. The code for the seeded version of the genetic algorithm is almost exactly the same, so it will be omitted in the interest of brevity.

To try and guide our mutations and initial squad selections to be high quality, I will implement a small level of pruning. We will only consider the top 10 goalkeepers, 40 defenders, 40 midfielders and 20 forwards in the categories of total season points and total season points per cost ratio.

To figure out how many generations are needed to get a representative view of how our algorithm performs, I ran the genetic algorithm on a population size of 20 for 150 generations - a very extensive run. This yielded the following data:

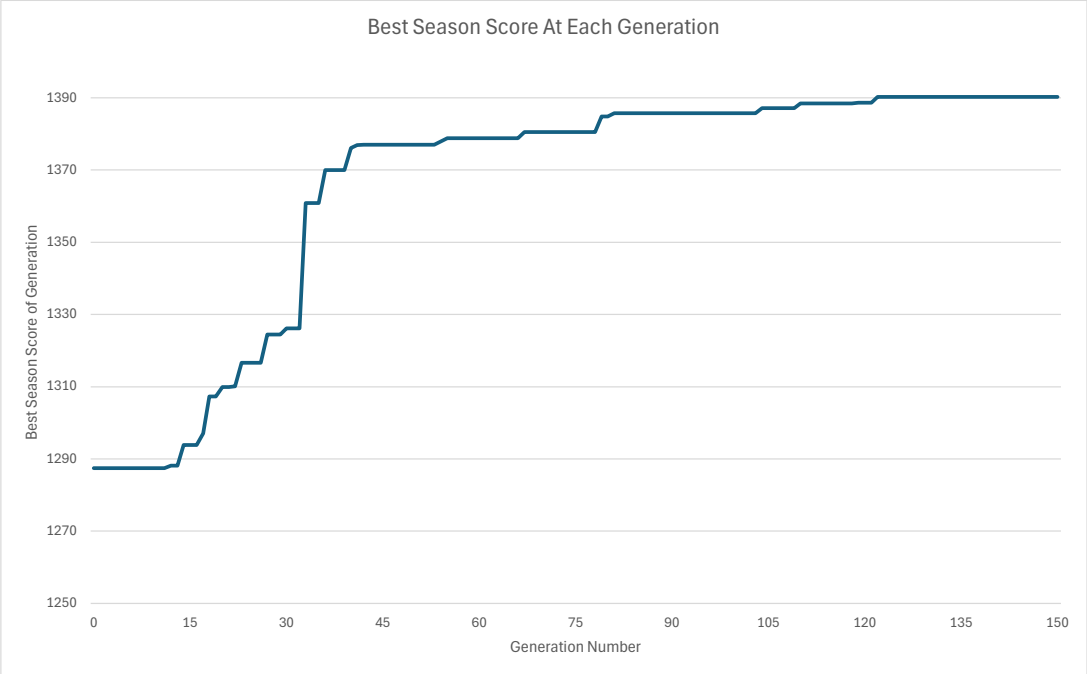

Figure 5.2: A Line Graph of Best Season Score Against Generation for a Random 20 Solution Population

Seeing how the best season score virtually plateaued around the 100 generation mark, and acknowledging the fact that although this might incrementally improve forever, the rate of improvement was slowed drastically at that point, I made the decision to do all my analysis up to generation 100.

After running the code a few times, I noticed that there was some random variation between the results produced by different runs. I ran four simulations with a population of 20 solutions, and this gave the following results:

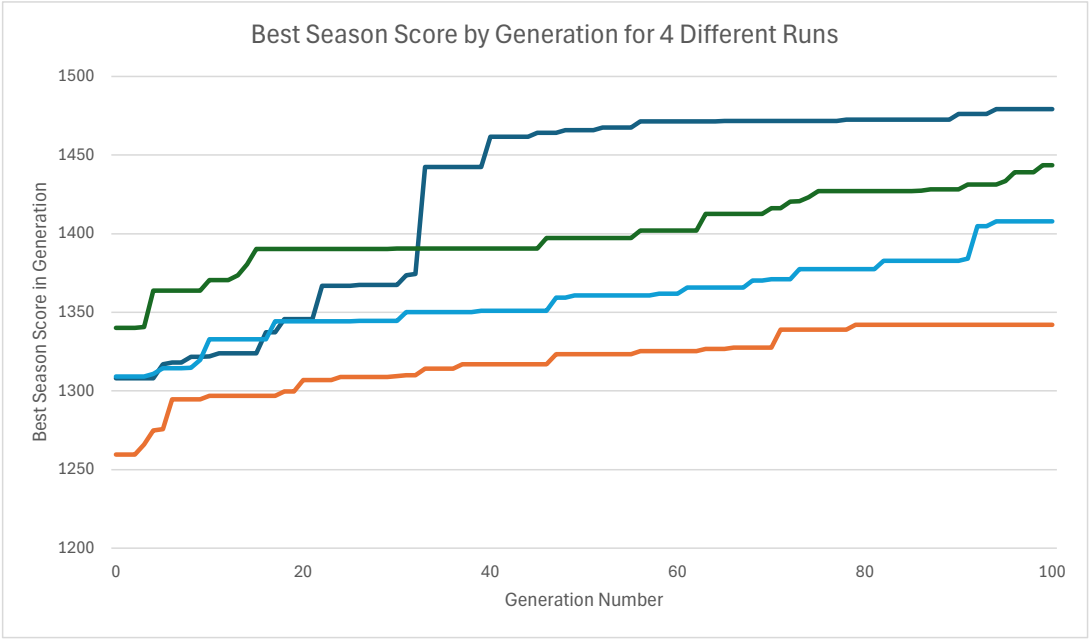

Figure 5.3: A Line Graph of Best Season Score Against Generation for Four Runs with Populations of 20 (Random, No Seeding)

We can see that there is a very large variation in the progression of the algorithm in these four runs, owing to the random aspects of the initial population selection, mutation function and crossover function.

This makes comparing the effect of different population sizes challenging. Even so, I ran four simulations for populations of 30 and of 40 each as well. If we take the average season score at each generation for these different population sizes to try and reduce the effect of randomness, we get the following plot:

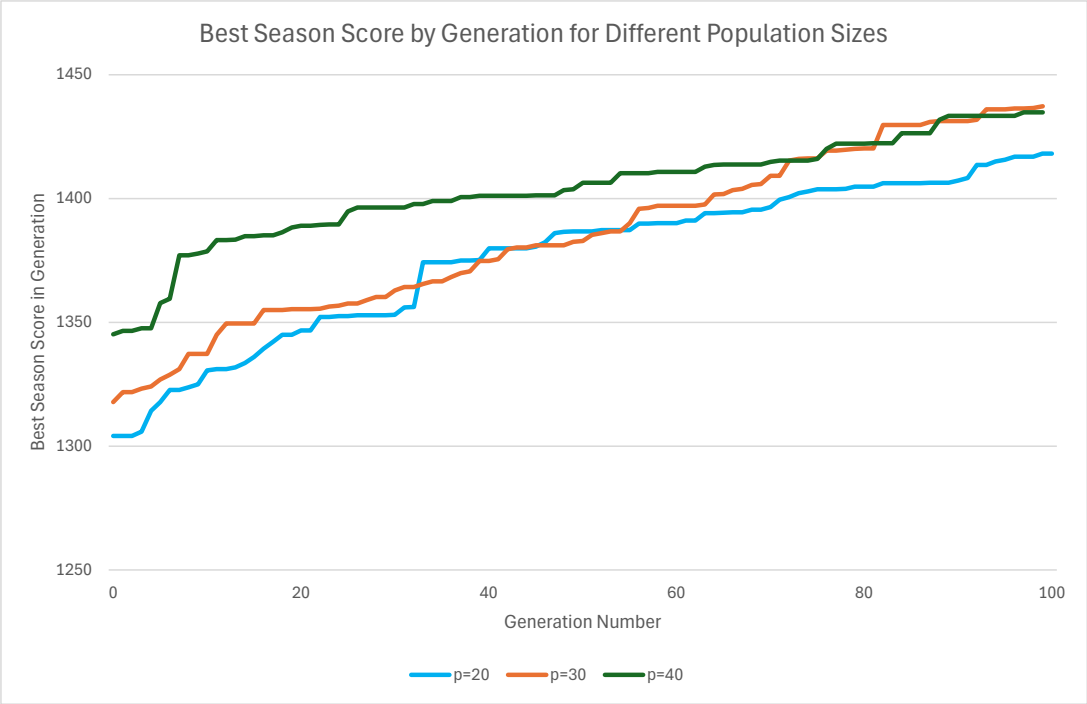

Figure 5.4: A Line Graph of Best Season Score Against Generation for Three Different Population Sizes (Random, No Seeding)

Looking at the early generations, there is a correlation between the size of the population and the best score of the initial population. This makes sense as the larger population simply has more opportunities to randomly create a high-quality solution.

By the end of the 100 generations however, the correlation is less strong, as the four runs with a population size of 30 actually had a better best season score than the population of 40 did. I believe this demonstrates the very large effect of randomness within the mutation, crossover and selection processes that the four repeats have only marginally mitigated.

I would've taken a larger sample of code runs and obtained more accurate averages but each run wasn't quick; each single one took at least 5 and up to 10 minutes. It seems that intrinsically the genetic algorithm's results are very variable for a randomly selected population, and the benefit of a larger population is marginal in comparison to the size of that variation.

I would say that based on this data and an understanding of the process at play, I believe a larger population size on average does lead to better results, but the significance is marginal above a population size of around 20 and the random variability inherent in the process reduces the significance of this effect.

The average time taken and number of function calls made for each of these runs are displayed below too.

Metric	Pop. = 20	Pop. = 30	Pop. = 40
Highest Total Season Points	1418.09	1437.34	1434.74
Total Function Calls	1.21 billion	1.87 billion	2.54 billion
Total Execution Time	291 seconds	445 seconds	597 seconds

Table 5.1: The Genetic Algorithm's Average Performance for Differently Sized Random Initial Populations

We can see that the total season score was relatively similar for all three runs, differing by approximately 2%. In terms of how computationally extensive each run was, we notice that the total function calls and time taken scale roughly linearly with the population size. Consequently, we see that using an initial population of 20 achieved the best ratio of computational effort to the score of the best solution.

Interestingly, we an see that the population of 30 managed the best result of them all. This fits with the idea mentioned earlier that a population size of around 30 can be more than adequate for having enough genetic diversity to obtain a strong solution [28].

5.3.2 Starting With a Seeded Population of Solutions

Naturally building on this, we seek a way to seed our initial populations with high quality solutions.

To seed our initial population with high quality solutions, we will conveniently take the 11 solutions we obtained in Chapter 4, each corresponding to a time horizon value of 1 to 11 weeks. These solutions were all found to have high total season scores, and so are perfect for our purpose.

The first way I aimed to implement this was by taking our 11 high-scoring solutions and combining them with 29 randomly generated solutions, forming an initial population of 40 solutions. I ran this process for 100 generations and yet found no improvement whatsoever in the best-scoring solution in the initial population, which was the $k = 2$ solution from the earlier knapsack method.

Puzzling over this, I hypothesised that the random populations were so far below the seeded populations that they had little impact in the process.

Taking this into account, I ran a simulation with an initial population of 44 solutions. 11 of those solutions were the previously found 11 high-scoring solutions, and the remaining 33 solutions were made by copying these 11 solutions 3 times, and mutating each copy once. This allowed our process to search a larger space whilst having very strong starting solutions.

Once again, after running these 44 solutions through 100 generations of crossover and mutation, still no improvements were found. This surprised me. To see why this might be the case, I tracked not only the best fitness each generation but also the average fitness. The results are shown below.

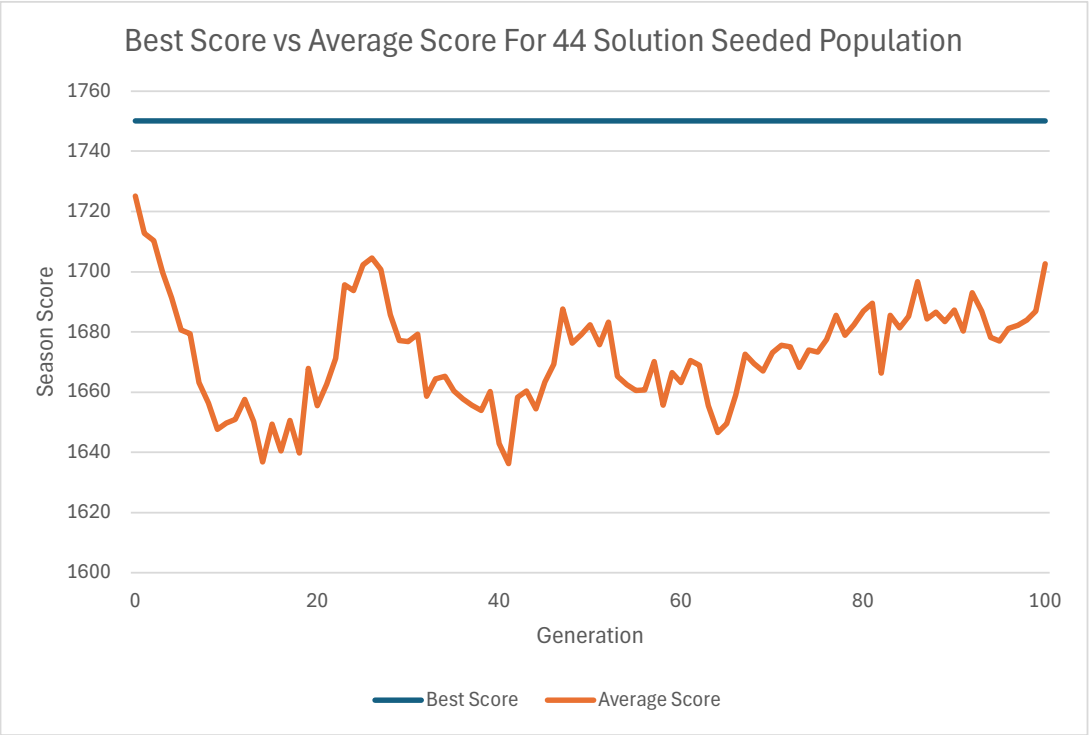

Figure 5.5: A Line Graph of Best Season Score and Average Season Score for a 44 Solution Seeded Population

We can see that the average fitness dropped from approximately 1720 at the start, before bouncing around unpredictably in the range of 1640 to 1700 points for the remainder of the run. That said, once again we failed to beat the knapsack $k = 2$ score we had in the initial population.

I believe we can understand why this is the case by considering the impact of the average generational change. Our starting teams are very high scoring, each with a finely calibrated combination of players over the season that were chosen to navigate upcoming fixture difficulties. In our genetic algorithm, the mutations are random, and therefore, the average mutation will bring in a less well-suited player to the team. It is precisely because our initial teams are so good that our mutations are negatively affecting our average total season score. This is in direct contrast to the effect of the genetic algorithm's mutations in the case where the population was randomly selected.

To improve our algorithm, we want to increase the benefit of the average generational change so that we reach a best score of greater than 1750 before plateauing.

To try and achieve this, I made three changes. First, I increased the level of pruning. Previously in our code, we had pruned the available players for selection to be the top 10 in total points and points to cost ratio for goalkeepers, top 40 for defenders, top 40 for midfielders and top 20 for forwards. To improve this, we pruned much more aggressively, only allowing the top 5, 20, 20, and 10 players in each position and category to be available for selection.

Secondly, I increased the amount of mutations drastically. Instead of each generation performing our crossovers and then mutating half of our population and performing selection, we will take our population, perform crossover and then mutate all organisms in 5 different ways. Then we run the roulette wheel selection. This gives us more chance to hit a successful mutation each generation.

Lastly, I only used the $k = 2$ solution for our seeding. Essentially, this corresponds to creating a population of mutants of the best solution found in the knapsack process to see if we can find any improvement whatsoever.

Because of the high increase in the mutation rate, I reduced the population to 10 solutions to ensure the code could be run in reasonable time. This run took approximately 40 minutes, and the results are shown below:

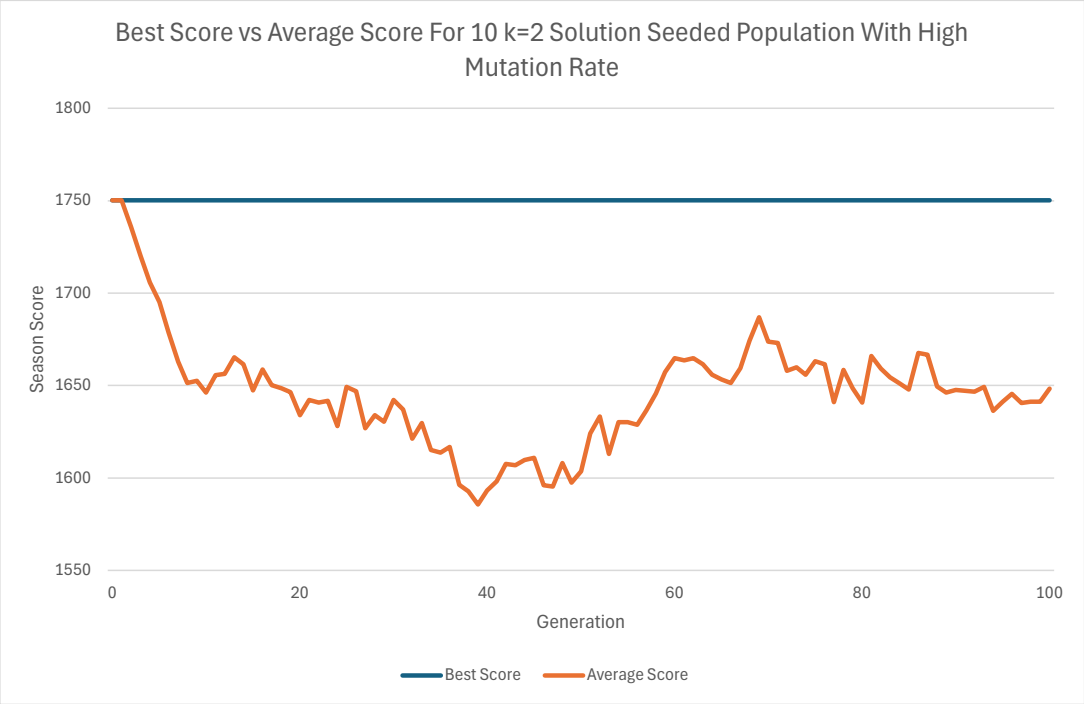

Figure 5.6: A Line Graph of Best Season Score Against Average Season Score for a 10 k=2 Solution Seeded Population With High Mutation Rate

Once again, we see the familiar pattern of the average season score of the solutions in the population languishing a significant distance from the best season points score of the solution in our initial population. In fact, in this set up, we are bouncing around the 1650 mark and below for most of our generations, below the average of the 44 population seeded simulation.

Just to see if I could ever achieve a better solution than the solution obtained for the recursive knapsack with a time horizon of $k = 2$, I coded a very simple Python script, adapted from our original genetic algorithm code in Appendix F. In the process, I took a single $k = 2$ solution and just mutated it up to 15 times, scoring it each time, and then

discarding it and trying again. I ran 50 cycles of this and in every single case, no solution better than the $k = 2$ solution was found.

To summarise our results from all of the different seeding processes, we can make a few key observations.

When looking at the random population process, there seemed to be a weak correlation between population size and the best score produced, but the variability within the results made the significance of this correlation small. With regards to population size, increasing the population size above 30 seemed to have no positive effect on the performance of the algorithm. The genetic algorithm was able to improve a random set of initial populations substantially over 100 generations, increasing its best score by around 100 points on average.

Considering population seeding, seeding our initial population with high-scoring solutions drastically improved the performance of the algorithm. Comparing the best solution that the random and seeded populations achieved is slightly unfair since the best solution the seeded population achieved came from the seeding. However, if we compare the average score the populations achieved over the generations, the seeded populations settled at a steady score of around 1680 points, much larger than the plateaus the random populations seemed to reach at around 1425 points.

Finally, we found the genetic algorithm doesn't always guarantee improvement. Particularly in the case where we seed with very high-scoring solutions, the genetic algorithm's process of mutation and crossover has a net negative effect. I hypothesis this comes from the mutations disturbing the delicate balance and future planning that the high quality-solutions necessarily have. That said, the genetic algorithm was effective in increasing the average and best scores achieved by a random population over time.

Chapter Six

Comparison of Algorithms

In this chapter, I will provide a comparison of the outcomes achieved by the three algorithms. I will discuss key aspects including the quality of the solutions they obtained, their computational efficiency and the difficulty in coding and implementing them.

For comparison purposes, I will be using the $k = 2$ time horizon recursive solution, and the solution produced by the random genetic algorithm for a population of 30 solutions. This is because both of these were the best performing results for the various iterations in their chapters. I will not be including the results from the seeded genetic algorithm as it failed to improve on the recursive knapsack solution, whilst also only obtaining its high score as a result of it.

Below is a table and some plots summarising the key findings of these three algorithms:

Metric	**Greedy**	**Recursive (k=2)**	**Random Genetic**
Highest Total Season Points	1683.20	1750.15	1437.34
Total Function Calls	50.8 million	1.31 billion	1.87 billion
Total Execution Time	12.9 seconds	281 seconds	445 seconds

Table 6.1: A Comparison of the Key Results of the Three Algorithms

Firstly, let's compare the key measure of success of any optimisation algorithm: how well the best solution they obtained scored. The recursive knapsack algorithm for a time horizon of two weeks achieved the best total season points score of 1750.15. Approximately 4% below this, the greedy algorithm scored 1680.20 and then much worse than either of these scores, the random population genetic algorithm scored a lacklustre 1437.34 points.

Taking the total function calls made by each algorithm as a proxy measure of the computational effort that each algorithm took to run, we can add some useful context. The greedy algorithm was by far the least computationally intensive, only making 50.8 million function calls. Taking around 20 times longer to run, the next most intensive was the recursive knapsack algorithm, making 1.31 billion function calls. Lastly, by far and

away the most intensive algorithm of our three was the genetic algorithm, making 1.84 billion function calls.

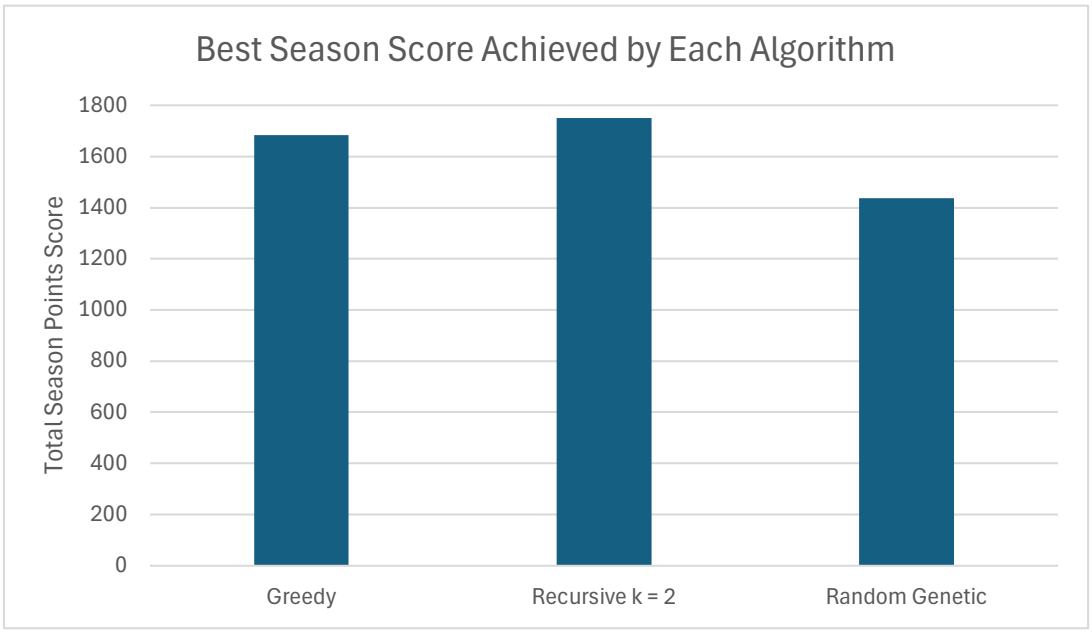

Figure 6.1: A Bar Graph of Best Total Season Score For Each Algorithm

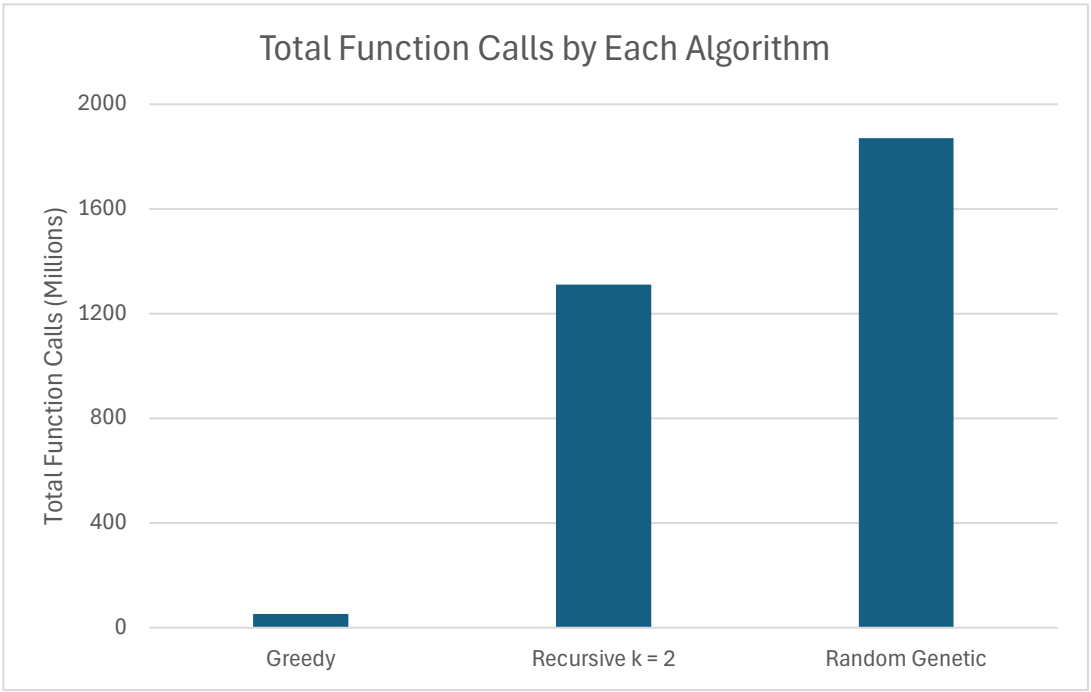

Figure 6.2: A Bar Graph of Total Function Calls For Each Algorithm

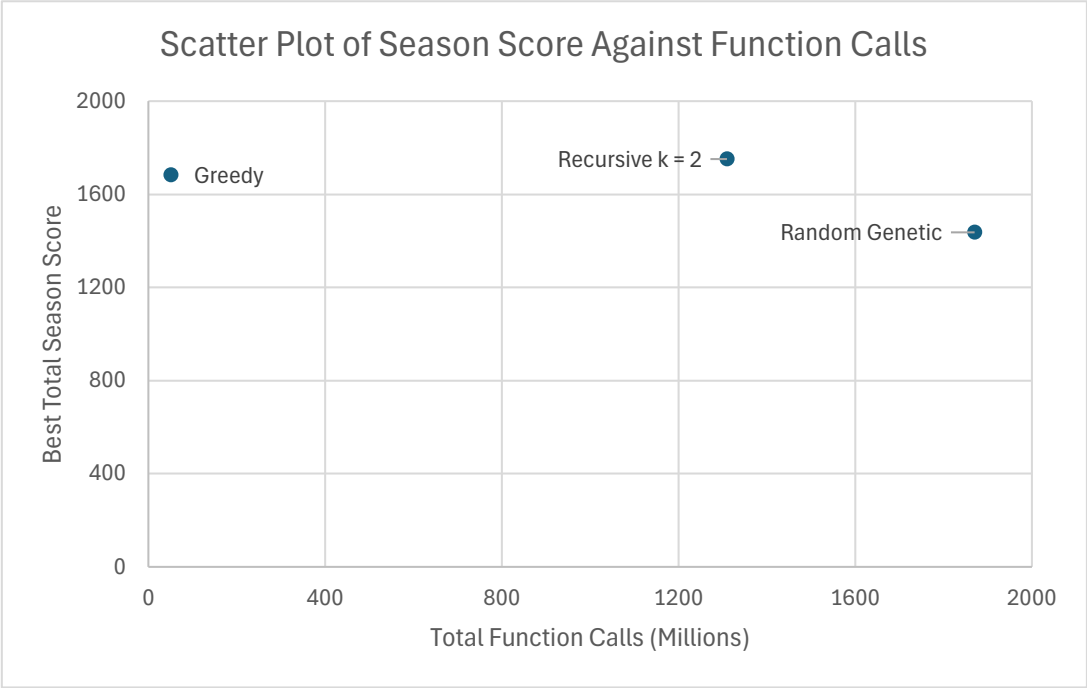

Figure 6.3: A Scatter Plot of Best Season Score Against Total Function Calls For Each Algorithm

A key quantity for the comparison of these algorithms is not only these quantities in isolation, but also their ratio; more function calls might not necessarily be a bad thing if it comes with a payoff of very high-quality solutions being obtained. Looking at Figure 6.3, we can see a scatter plot of the best total season score of each algorithm against the total function calls it made. If we consider a line from the origin to each point, the gradient of this line is proportional to the ratio of the best total season points score to the total function calls. Consequently, this gradient might be said to represent the efficiency of each algorithm in balancing the quality of the solutions to the computation required. Thus, we can see that the greedy algorithm was by far the most effective, with the recursive knapsack algorithm in second place and the random genetic algorithm in third. Taking this ratio explicitly, the greedy algorithm obtained 33.13 points in its total season score per million function calls, the recursive knapsack algorithm obtained 1.34 points per million function calls and the random genetic algorithm obtained 0.77 points per million function calls.

Alongside these concrete measures of performance, it is also worth considering how challenging each algorithm was to practically implement. The easiest code to write by far was the greedy algorithm, as each step in the pseudocode was simple and logically followed naturally. The initial team and transfers were picked by simply sorting our database and choosing the highest scoring valid replacement, meaning the hardest part was simply checking the constraints.

The recursive knapsack algorithm was more challenging as it required setting up memoization by assigning each state in the process a unique ID and coding the recursion

process. However, once that recursion was coded, the process was complete.

By far the most challenging to implement was the genetic algorithm, as reflected in the length of the final code, and the length of the pseudocode of just one section, as shown in Appendix E. Implementing the mutation and crossover process whilst trying to ensure valid teams were still produced took extensive writing and debugging, and many edge cases had to be covered. These factors should be taken into account when assessing the practicability of each algorithm for running in real-world contexts.

Chapter Seven

Conclusion

In summary, this project found that the recursive knapsack method was the most effective at obtaining high-quality solutions to the problem of selecting optimal FPL teams over the course of a season. Its extremely strong initial squad selections from the recursion combined with its well-chosen time horizon for considering transfers made it the best choice of the three.

The greedy algorithm's short-term focus led to poor GW1 selections, reducing its total season score dramatically. Interestingly, when we compare the $k = 1$ knapsack solution with the greedy algorithm, they share the same transfer philosophy but simply differ in their starting teams. The resultant effect of having a poor starting team is reflected in their different scores: 1683.20 points for the greedy and 1746.54 points for the $k = 1$ knapsack solution. Despite this shortfall, the greedy algorithm was by far the most efficient in how good its solutions were, relative to the computational effort required. In a real-life context, if the search space was simply too large for the recursive knapsack algorithm to run in suitable time, the findings of this project suggest the greedy algorithm is a suitable approach for finding a good-quality solution.

The genetic algorithm's approach was shown to be highly ineffectual in this specific context. Despite the fact it was able to slowly improve the total season score of a random population over time, its solutions plateaued far below the solutions of the other two algorithms. I hypothesise that this comes from the fact that there are simply far too many available players and transfer choices than can be adequately investigated with a purely random approach. In addition to its poor output, the genetic algorithm was also by far the most computationally effortful algorithm to run. I anticipated that this computational complexity would be beneficial as it would allow the algorithm to effectively plan on a dynamically changing time-horizon, given enough generations to make this organically happen. I was proved wrong by the findings, and I feel that the specific context of this problem with its multiple constraints and highly-dependent transfer strategy led the randomness of the genetic algorithm to ultimately be a significant weakness.

There are many ways that the scope of this project could be extended that occurred

to me during the process.

The first is an adaptation to the mutation strategy of the genetic algorithm which might help it produce higher-quality solutions. If instead of the purely random mutation strategy, the mutation process had a weighted probabilistic element, such that higher quality mutations were favoured, the outcomes might be considerably better. Implementing this could be done in conjunction with the tuning of other parameters, such as the amount of elitism and the rates of crossover and mutation. These values in this project were chosen relatively arbitrarily, so a further examination in another project could be fruitful.

Another extension to this project would be to look at the full, unsimplified version of FPL. This involves selecting 15 players rather than 11, and allowing any 4 of them to be on the bench each week, meaning their scores aren't counted. This aspect was neglected due to the increasing complexity it would add. However, I believe the algorithms coded here could serve as a suitable base for future work which includes the benching component.

Lastly, perhaps the most significant extension would be to consider other optimisation algorithms that might exceed the scores achieved in this project. The one that struck me as potentially being the most useful given the results we've obtained in this project is the beam search algorithm. The beam search algorithm is a way of exploring a decision tree that is far too large to exhaustively search [34]. In our context, since we already have very good starting teams from the knapsack algorithm, I would suggest using the beam search algorithm in conjunction with it to navigate the transfer strategy process. In the beam search algorithm, a fixed number of strong-looking decisions are evaluated at each step, and only a small number of those are then chosen. Then, we consider all of the strong-looking decisions that we could make from these new positions, and then we choose a fixed number of those. This process can be conceptualised as following the most fruitful-looking branches down a decision tree, without our exploration growing exponentially more vast. Because we know that our initial GW1 squads from the knapsack algorithms are provably optimal for scoring the most total points in the first 1 to 11 weeks, I feel that the best way to improve the solutions obtained in this project is by improving our transfer strategy following that, and hence the beam search algorithm is a great choice.

In conclusion, the aims of this project - to create a predicted points model and code and evaluate three optimisation algorithms - have been achieved. Despite having extensively examined these three optimisation algorithms, the extensions that I have laid out above show that there are many ways in which work on this topic could be developed even further, ultimately leading to an improvement in our ability to tackle problems of this kind.

Appendix A

Sample of Database

	web_name	element_type	team	team_name	now_cost	minutes	total_points	points_per_game	goals_scored	expected_goals	expected_goals_per_90
378	Gibbs-White	3	16	Nott'm Forest	57	3156	142	3.8	5	6.65	0.19
379	Kouyaté	3	16	Nott'm Forest	48	206	12	1	0	0.25	0.11
380	Mangala	3	16	Nott'm Forest	50	1562	40	2	1	0.77	0.04
381	Yates	3	16	Nott'm Forest	48	1979	59	1.7	1	2.78	0.13
382	Dominguez	3	16	Nott'm Forest	50	1489	67	2.6	2	1.3	0.08
383	Sangaré	3	16	Nott'm Forest	49	1031	25	1.5	0	0.67	0.06
384	Reyna	3	16	Nott'm Forest	46	228	13	1.4	0	0.36	0.14
385	Johnson	3	18	Spurs	58	2317	131	3.7	5	10.36	0.4
386	Bentancur	3	18	Spurs	54	997	32	1.4	1	2.01	0.18
387	Bissouma	3	18	Spurs	50	2070	44	1.6	0	1.18	0.05
388	Bryan	3	18	Spurs	49	200	12	1.1	0	0.67	0.3
389	Højbjerg	3	18	Spurs	51	1289	39	1.1	0	0.68	0.05
390	Kulusevski	3	18	Spurs	67	2759	130	3.6	8	4.95	0.16
391	Lo Celso	3	18	Spurs	48	491	47	2.1	2	0.54	0.1
392	Maddison	3	18	Spurs	78	2137	117	4.2	4	6.53	0.28
393	Richarlison	3	18	Spurs	68	1481	122	4.4	11	9.49	0.58
394	Sarr	3	18	Spurs	44	2063	86	2.5	3	3.17	0.14
395	Skipp	3	18	Spurs	45	691	22	1	0	0.77	0.1
396	Son	3	18	Spurs	100	2934	213	6.1	17	11.49	0.35
397	Solomon	3	18	Spurs	51	196	14	2.8	0	0.41	0.19
398	Moore	3	18	Spurs	45	3	2	1	0	0	0
399	Phillips	3	19	West Ham	47	394	10	0.8	0	0.22	0.05
400	Benrahma	3	19	West Ham	56	611	20	1.5	0	1.91	0.28
401	Bowen	3	19	West Ham	76	3020	182	5.4	16	11.96	0.36
402	Cornet	3	19	West Ham	52	109	13	1.9	1	0.4	0.33
403	P.Fornals	3	19	West Ham	46	418	19	1.3	0	0.31	0.07
404	L.Paquetá	3	19	West Ham	60	2627	100	3.2	4	5.19	0.18
405	Souček	3	19	West Ham	49	2867	116	3.1	7	6.69	0.21
406	Álvarez	3	19	West Ham	50	2375	62	2	1	1.37	0.05
407	Ward-Prowse	3	19	West Ham	57	3000	146	3.9	7	6.23	0.19
408	Kudus	3	19	West Ham	67	2485	137	4.2	8	5.06	0.18
409	Earthy	3	19	West Ham	45	33	8	2.7	1	0.35	0.95
410	Doyle	3	20	Wolves	44	1209	38	1.5	0	0.42	0.03
411	Hee Chan	3	20	Wolves	54	2116	125	4.3	12	7.53	0.32
412	João Gomes	3	20	Wolves	49	2646	75	2.2	2	1.82	0.06

Figure A.1: A Screenshot of the Database Used by the Model [11]

Appendix B

Model Predictions

	A	B	C	D	E	F	G	H	I	J	K
1	id	first_name	second_name	web_name	element_type	team	team_name	now_cost	GW1 Points	GW2 Points	GW3 Points
2	17	Aaron	Ramsdale	Ramsdale	1	1	Arsenal	45	0.58	0.51	0.53
3	113	David	Raya Martin	Raya	1	1	Arsenal	53	3.75	3.19	3.38
4	49	Emiliano	Martínez Romero	Martinez	1	2	Aston Villa	52	2.61	2.31	2.96
5	53	Robin	Olsen	Olsen	1	2	Aston Villa	39	0.24	0.24	0.24
6	77	Norberto	Murara Neto	Neto	1	3	Bournemouth	46	2.43	2.13	2.36
7	88	Mark	Travers	Travers	1	3	Bournemouth	40	0.46	0.37	0.44
8	607	Ionuț	Radu	Radu	1	3	Bournemouth	43	0.11	0.11	0.11
9	101	Mark	Flekken	Flekken	1	4	Brentford	47	2.73	2.33	2.96
10	116	Thomas	Strakosha	Strakosha	1	4	Brentford	39	0.08	0.08	0.08
11	148	Jason	Steele	Steele	1	5	Brighton	40	1.13	1.12	1.06
12	152	Bart	Verbruggen	Verbruggen	1	5	Brighton	44	1.48	1.46	1.35
13	145	Robert	Sánchez	Sanchez	1	7	Chelsea	45	1.02	1.19	1.17
14	687	Đorđe	Petrović	Petrović	1	7	Chelsea	47	1.50	1.78	1.76
15	230	Sam	Johnstone	Johnstone	1	8	Crystal Palace	43	1.66	1.73	1.52
16	233	Remi	Matthews	Matthews	1	8	Crystal Palace	39	0.00	0.00	0.00
17	385	Dean	Henderson	Henderson	1	8	Crystal Palace	44	1.32	1.36	1.22
18	263	Jordan	Pickford	Pickford	1	9	Everton	48	3.31	3.08	3.33
19	275	Bernd	Leno	Leno	1	10	Fulham	48	3.00	3.52	3.52
20	291	Alisson	Ramses Becker	A.Becker	1	11	Liverpool	57	2.76	2.31	2.34
21	301	Caoimhin	Kelleher	Kelleher	1	11	Liverpool	36	0.85	0.74	0.75
22	352	Ederson	Santana de Moraes	Ederson M.	1	13	Man City	55	2.42	3.30	2.81
23	361	Stefan	Ortega Moreno	Ortega Moreno	1	13	Man City	36	0.45	0.53	0.49
24	597	André	Onana	Onana	1	14	Man Utd	50	2.98	2.87	2.51
25	409	Martin	Dubravka	Dubravka	1	15	Newcastle	43	1.77	1.57	1.50
26	424	Nick	Pope	Pope	1	15	Newcastle	53	1.73	1.42	1.30
27	580	Loris	Karius	Karius	1	15	Newcastle	39	0.05	0.05	0.05
28	28	Matt	Turner	Turner	1	16	Nott'm Forest	37	1.07	1.17	1.11
29	710	Odysseas	Vlachodimos	Odysseas	1	16	Nott'm Forest	45	0.37	0.42	0.40
30	808	Matz	Sels	Sels	1	16	Nott'm Forest	45	0.93	0.99	0.96
31	520	Guglielmo	Vicario	Vicario	1	18	Spurs	53	3.05	2.73	2.51

Figure B.1: A Screenshot of the Excel Sheet Containing the Model's Predicted Points

Appendix C

Greedy Algorithm Code

```
0
import pandas as pd
import cProfile
import pstats
import time

def get_player_data():
    """Get our FPL database from the csv file that has all of our predicted
  points data and puts it in a dataframe."""
    player_data = pd.read_csv('Player Points Database.csv')
    # The database we used had numbers to represent the positions. We now map
  from the number to the positions and add this information to a new column.
10  position_map = {
        1: 'GKP',
        2: 'DEF',
        3: 'MID',
        4: 'FWD'}
    player_data['position'] = player_data['element_type'].map(position_map)
    return player_data

def set_up_constraints():
    """Here, we put our budget, position and team constraints into a dictionary
  ."""
20  return {
        'budget': 800,
        # 1 million pounds corresponds to 10 budget units in the database.
        'positions': {
            'GKP': 1,
            'DEF': 4,
            'MID': 4,
            'FWD': 2},
        'max_players_per_team': 3,
        'min_costs': {
30          # These come from just looking at our database.
            'GKP': 36,
            'DEF': 37,
            'MID': 42,
            'FWD': 41}}

def choose_starting_team(player_data, constraints):
    """This functions takes in our player data and constraints and selects our
  initial GW1 team by using the greedy algorithm."""
    # We sort our dataset by highest GW1 points to lowest and put it in a new
  dataframe.
    sorted_players = player_data.sort_values('GW1 Points', ascending=False)
40  # We start with an empty list.
    selected_team = []
    spent_so_far = 0
```

```
    # We create a dictionary to track how many players we've picked so far in each position.
    position_counter = {pos: 0 for pos in constraints['positions']}
    # We also create a dictionary to track how many players we've picked from each team in the league.
    team_counter = {}

    # We loop through each row of our dataframe which corresponds to each player.
    for _, player in sorted_players.iterrows():
        # This breaks the loop if our team has been fully picked.
        if len(selected_team) == 11:
            break

        # This skips this row if this player's position has been filled up in the team already.
        if position_counter[player['position']] == constraints['positions'][player['position']]:
            continue

        # This skips this row if the player's club already has three other players in the team.
        if team_counter.get(player['team'], 0) == constraints['max_players_per_team']:
            continue

        # We need to make sure that we have enough money left after this choice to fill up the rest of our team.
        positions_still_needed = {
            pos: constraints['positions'][pos] - position_counter[pos]
            for pos in constraints['positions']}

        money_needed_for_remaining = sum(
            constraints['min_costs'][pos] * (count - (1 if pos == player['position'] else 0))
            for pos, count in positions_still_needed.items())

        # This skips the row if we don't have enough money left to put them in the squad.
        money_available = constraints['budget'] - spent_so_far - money_needed_for_remaining
        if player['now_cost'] > money_available:
            continue

        # Finally, if we haven't skipped by now, we add the player to our initial team.
        selected_team.append(player)
        spent_so_far += player['now_cost']
        position_counter[player['position']] += 1
        team_counter[player['team']] = team_counter.get(player['team'], 0) + 1

    # We convert the list of player rows into a dataframe at the end.
    return pd.DataFrame(selected_team)

def show_team_info(team, is_starting_eleven=False):
    """This function displays the points of the initial and final team we get."""
    squad_value = team['now_cost'].sum()

    if is_starting_eleven:
        points_column = 'GW1 Points'
    else:
        points_column = 'GW38 Points'

    total_points = team[points_column].sum()

    if is_starting_eleven:
```

```
        print("\nStarting Team:")

    print("\nSquad List:")
    for _, player in team.iterrows():
        if is_starting_eleven:
            points = player['GW1 Points']
            points_label = "GW1 Points"
        else:
            points = player['GW38 Points']
            points_label = "GW38 Points"
        print(f"{player['first_name']} {player['second_name']} ({player['
position']}) - "
              f"Value:   {player['now_cost']/10:.1f}m, {points_label}: {points
:.2f}")

    print(f"\nTotal Squad Value:   {squad_value/10:.1f}m")
    print(f"Expected {points_label}: {total_points:.2f}")

def make_transfer(team, player_data, current_gw):
    """This function looks at our team and finds the best transfer according to
 the greedy algorithm."""
    print(f"\nTransfer before GW{current_gw + 1}:")
    next_gw = f'GW{current_gw + 1} Points'
    best_transfer = {
        'out': None,
        'in': None,
        'points_gain': 0,
        'budget_impact': 0}

    # Look through  all the players in our team and find which player would be
best to transfer in for them.
    for _, current_player in team.iterrows():
        current_points = current_player[next_gw]

        # Make a list of all the players of this player's position and sort
them by their next week points.
        potential_replacements = player_data[
            (player_data['position'] == current_player['position']) &
            (player_data['id'] != current_player['id'])
        ].sort_values(next_gw, ascending=False)

        # Loop through all the possible replacements to find the most
beneficial one.
        for _, replacement in potential_replacements.iterrows():
            # Check if the replacement would respect our constraints.
            budget_after_transfer = team['now_cost'].sum() - current_player['
now_cost'] + replacement['now_cost']
            team_count = len(team[team['team'] == replacement['team']])

            # This skips this replacement if they'd disobey our constraints.
            if (budget_after_transfer > 800 or
                (team_count == 3 and replacement['team'] != current_player['
team']) or
                replacement['id'] in team['id'].values):
                continue

            # Here we calculate how many points this potential transfer would
make
            points_gain = replacement[next_gw] - current_points
            if points_gain > best_transfer['points_gain']:
                best_transfer = {
                    'out': current_player,
                    'in': replacement,
                    'points_gain': points_gain,
                    'budget_impact': budget_after_transfer}
```

```
    # Once we've evaluated all transfers, we make the one which is most beneficial.
    if best_transfer['points_gain'] > 0:
        print(f"Out: {best_transfer['out']['first_name']} {best_transfer['out']['second_name']} ({best_transfer['out']['position']})")
        print(f"In: {best_transfer['in']['first_name']} {best_transfer['in']['second_name']} ({best_transfer['in']['position']})")
        print(f"Points gain: {best_transfer['points_gain']:.2f}")

        # This removes the player being transferred out.
        new_team = team[team['id'] != best_transfer['out']['id']]
        # This adds the new player.
        new_team = pd.concat([new_team, pd.DataFrame([best_transfer['in']])], ignore_index=True)
    else:
        print("No beneficial transfers found")
        new_team = team

    remaining_budget = 800 - new_team['now_cost'].sum()
    print(f"Money Left in Bank:   {remaining_budget/10:.1f}m")
    return new_team

def main():
    """This is the function which executes all of the other functions to actually run the full greedy algorithm for the season."""

    # This sets up our player data and constraints.
    player_data = get_player_data()
    constraints = set_up_constraints()

    # This chooses our initial team and prints it.
    initial_team = choose_starting_team(player_data, constraints)
    show_team_info(initial_team, is_starting_eleven=True)

    # This sets up the variables which track our season.
    current_team = initial_team.copy()
    total_points = initial_team['GW1 Points'].sum()
    squad_values = [initial_team['now_cost'].sum()]

    # This loop does all the transfers and tracks our points and squad value.
    for gw in range(1, 38):
        current_team = make_transfer(current_team, player_data, gw)

        total_points += current_team[f'GW{gw + 1} Points'].sum()

        squad_values.append(current_team['now_cost'].sum())

    # Calculate average squad value
    avg_squad_value = sum(squad_values) / len(squad_values)

    # Print our final team.
    print("\nFinal Team:")
    show_team_info(current_team)

    # Print a season summary.
    print("\n Season Summary:")
    print(f"Total Season Points: {total_points:.2f}")
    print(f"Average squad value:   {avg_squad_value/10:.1f}m")

# This function times our code and also counts the total number of functions that are called.
def run_with_profile(func):
    start_time = time.time()
    profiler = cProfile.Profile()
    profiler.enable()
    func()
    profiler.disable()
```

```
    stats = pstats.Stats(profiler)
    total_calls = stats.total_calls
    end_time = time.time()
    execution_time = end_time - start_time

    # This displays the time taken and number of function calls.
    print(f"\n Performance Summary:")
    print(f"Total function calls: {total_calls}")
    print(f"Total execution time: {execution_time:.3f} seconds")

if __name__ == "__main__":
    run_with_profile(main)
```

Appendix D

Knapsack Algorithm GW1 Code

```
0
import pandas as pd
import cProfile
import pstats
import time

# This lets us print the team names of the final squad to check our constraints
   are met
TEAM_NAMES = {
    1: "Arsenal",
    2: "Aston Villa",
10  3: "Bournemouth",
    4: "Brentford",
    5: "Brighton",
    6: "Burnley",
    7: "Chelsea",
    8: "Crystal Palace",
    9: "Everton",
    10: "Fulham",
    11: "Liverpool",
    12: "Luton",
20  13: "Man City",
    14: "Man Utd",
    15: "Newcastle",
    16: "Nott'm Forest",
    17: "Sheff Utd",
    18: "Spurs",
    19: "West Ham",
    20: "Wolves"
}

30 def get_player_data():
    """Get our FPL database from the csv file that has all of our predicted
  points data and puts it in a dataframe. Then prune it."""
    player_data = pd.read_csv('../Player Points Database.csv')
    position_map = {
        1: 'GKP',
        2: 'DEF',
        3: 'MID',
        4: 'FWD'}
    player_data['position'] = player_data['element_type'].map(position_map)

40  # Add a column for the GW1 points per cost ratio of each player
    player_data['points_per_cost'] = player_data['GW1 Points'] / player_data['
  now_cost']

    # Create a new empty dataframe to fill up with the pruned data
    pruned_data = pd.DataFrame()
```

```
    # Iterate through each position, getting the X number of players in points and points per cost ratio
    for pos, (top_points, top_ratio) in {
        'GKP': (10, 10),
        'DEF': (40, 40),
        'MID': (40, 40),
        'FWD': (20, 20)
    }.items():
        pos_data = player_data[player_data['position'] == pos]

        # Select the players with the highest GW1 points
        top_by_points = pos_data.nlargest(top_points, 'GW1 Points')
        # Select the players with the highest GW1 points to cost ratio
        top_by_ratio = pos_data.nlargest(top_ratio, 'points_per_cost')

        # Add these players to the initially blank dataframe and remove duplicates if there are any
        pruned_data = pd.concat([pruned_data, top_by_points, top_by_ratio]).drop_duplicates()

    return pruned_data

def set_up_constraints():
    """Here, we put our budget , position and team constraints into a dictionary."""
    return {
        'budget': 800,
        'positions': {
            'GKP': 1,
            'DEF': 4,
            'MID': 4,
            'FWD': 2},
        'max_arsenal_players': 3}

# We create a cache which is a dictionary. It stroes previously computed results so we don't have to keep doing them again.
cache = {}

def get_state_key(i, budget, pos_counts, arsenal_count):
    """This functions creates a unique key to keep track of our state in the iteration. It includes the index of the player under consideration, the budget remaining, the number of players in each position and the number of Arsenal players. Our iteration starts with the player with the highest index and works down. The index number here represents the player's index we are at as we go down the decision tree."""
    pos_tuple = tuple(pos_counts.items())
    return (i, budget, pos_tuple, arsenal_count)

def is_valid_addition(player, pos_counts, arsenal_count, constraints):
    """This function checks if adding a certain player would violate constraints."""
    # Checking position constraints
    if pos_counts[player['position']] == constraints['positions'][player['position']]:
        return False

    # Checking Arsenal team constraint
    if player['team'] == 1:
        if arsenal_count == constraints['max_arsenal_players']:
            return False

    return True

def solve_recursive(players, constraints, i, budget, pos_counts, arsenal_count):
```

```
    """This function carries out the recursive knapsack algorithm. It takes in
    our player database, constraints, player index of where we are in the
    decision tree, number of each position filled and numebr of Arsenal players
    picked."""

    # If we have no players under consideration or or no budget to consider,
    return 0 for points and no selections.
    if i < 0 or budget <= 0:
        return 0, []

    # Here, we check if we've been in this situation before. If we have, take
    the result that's stored in the cache for this state key.
    state_key = get_state_key(i, budget, pos_counts, arsenal_count)
    if state_key in cache:
        return cache[state_key]

    # This exracts the specific player under consideration from the database
    player = players.iloc[i]

    # Consider not taking the player
    dont_take_value, dont_take_players = solve_recursive(
        players, constraints, i - 1, budget, pos_counts.copy(), arsenal_count
    )

    # Consider taking the player, if it's valid
    take_value, take_players = 0, []
    if (player['now_cost'] <= budget and
        is_valid_addition(player, pos_counts, arsenal_count, constraints)):

        # Update the position counter and Arsenal counter
        new_pos_counts = pos_counts.copy()
        new_pos_counts[player['position']] += 1
        new_arsenal_count = arsenal_count + (1 if player['team'] == 1 else 0)

        # Call the function for the next player to continue the recursion
        value, players = solve_recursive(
            players,
            constraints,
            i - 1,
            budget - player['now_cost'],
            new_pos_counts,
            new_arsenal_count)
        take_value = value + player['GW1 Points']
        take_players = players + [i]

    # Decide whether picking the player or not is worth it and store the result
    if take_value > dont_take_value:
        result = (take_value, take_players)
    else:
        result = (dont_take_value, dont_take_players)

    cache[state_key] = result
    return result

def find_optimal_team(player_data, constraints):
    """This function implements the recursive function that we defined above.
    """
    # We set the position counters to 0.
    initial_pos_counts = {pos: 0 for pos in constraints['positions']}

    # Make sure the cache is empty
    cache.clear()

    # Start the iteration process from the highest index player and store the
    indices of the players of the optimal team
    _, selected_indices = solve_recursive(
        player_data,
```

```
        constraints,
        len(player_data) - 1,
        constraints['budget'],
        initial_pos_counts,
        0)

    return player_data.iloc[selected_indices]

def show_team_info(team):
    """This function shows us information about the optimal team."""
    squad_value = team['now_cost'].sum()
    total_points = team['GW1 Points'].sum()

    print("\nSelected Team:")
    for _, player in team.iterrows():
        team_name = TEAM_NAMES.get(player['team'], f"Team {player['team']}")
        print(f"{player['first_name']} {player['second_name']} ({player['
  position']}, {team_name}) - "
              f"Value:   {player['now_cost']/10:.1f}m, GW1 Points: {player['GW1
   Points']:.2f}")

    print(f"\nTotal Squad Value:   {squad_value/10:.1f}m")
    print(f"Expected GW1 Points: {total_points:.2f}")

def main():
    """This is the main function which executes our other functions."""
    player_data = get_player_data()
    constraints = set_up_constraints()

    print(f"\nPruned dataset contains {len(player_data)} players")

    print("\nFinding optimal team...")
    optimal_team = find_optimal_team(player_data, constraints)

    show_team_info(optimal_team)

def run_with_profile(func):
    """This function times our code and also counts the total number of
  function calls."""
    start_time = time.time()
    profiler = cProfile.Profile()
    profiler.enable()
    func()
    profiler.disable()

    stats = pstats.Stats(profiler)
    total_calls = stats.total_calls
    end_time = time.time()
    execution_time = end_time - start_time

    # This diplays the time taken and number of function calls.
    print(f"\n Performance Summary:")
    print(f"Total function calls: {total_calls}")
    print(f"Total execution time: {execution_time:.3f} seconds")

if __name__ == "__main__":
    run_with_profile(main)
```

Appendix E

Mutation Process Pseudocode

```
When we mutate, we pick a random week from 1 to 38. There are two cases:

Case 1: We pick GW1.

In this case, we have to mutate the starting team.
We pick one player randomly in the starting team and get the data about the
player we want to replace.
We call this player 'player_to_replace'.
We then search the database for all players we could swap in which fit all the
constraints.
We randomly pick one and call this player 'replacement'.
We make the swap.

After this, we have to update the future transfer strategy so it doesn't give
an invalid solution.
To do this, first we need to edit the lowest gameweek transfer which transfers
out 'player_to_replace' (if there is one).
If this transfer exists, we have to switch this transfer from transferring out
'player_to_replace' to transferring out 'replacement'.
Second, we need to edit the lowest gameweek transfer which transfers in
'replacement' (if there is one).
If this transfer exists, we have to switch this transfer from transferring in
'replacement' to transferring in 'player_to_replace'.

Once this is done, we validate the final solution we have obtained to check it
fits all of the constraints each week.
If it's not, we restart this process and try again.
```

Case 2: We pick GWX, where X is between 2 and 38 inclusive.

In this case, we mutate the transfer strategy. We will be changing the transfer which occurs in GWX.
To do this, we first delete the transfer planned for this week.
We call the player who would've been transferred out 'deleted_out' and the player who would've been transferred in 'deleted_in'.

Next, we choose a random player currently in the team (i.e. the team obtained in GW(X-1)).
We call this player 'player_to_replace'.
We then search the database for all players we could swap in which fit all the constraints.
We randomly pick one and call this player 'replacement'.
We add this transfer to our transfer strategy.

After this, we have to update the future transfer strategy so it doesn't give an invalid solution.
First, we will not have 'deleted_in' in our team anymore.
Therefore, we need to edit the lowest gameweek transfer (above GWX) which transfers out 'deleted_in' (if there is one).
If this transfer exists, we have to switch this transfer from transferring out 'deleted_in' to transferring out 'deleted_out'.

Second, we will not be able to transfer in 'deleted_out' in the future.
Therefore, we need to edit the lowest gameweek transfer (above GWX) which transfers in 'deleted_out' (if there is one).
If this transfer exists, we have to switch this transfer from transferring in 'deleted_out' to transferring in 'deleted_in'.

Once this is done, we validate the final solution we have obtained to check it fits all of the constraints each week.
If it's not, we restart this process and try again.

Appendix F

Genetic Algorithm Random Population Code

```
0
import pandas as pd
import random
import time
import cProfile
import pstats
import copy
import numpy as np

# Setting up constraints
10 budget = 800
positions = {
    1: 1,
    2: 4,
    3: 4,
    4: 2}

max_players_per_team = 3
population_size = 20
generations = 100
20
def get_player_data(filename):
    """Get our FPL database from the csv file that has all of our predicted
  points data and put it in a dataframe."""
    return pd.read_csv(filename)

def prune_player_data(player_data):
    """This function prunes our player data to only consider some players."""

    # We calculate total points for each player
    gw_columns = [f'GW{i} Points' for i in range(1, 39)]
30  player_data['total_points'] = player_data[gw_columns].sum(axis=1)

    # We also calculate points per cost
    player_data['points_per_cost'] = player_data['total_points'] / player_data[
  'now_cost']

    selected_players = []

    # We loop through each position, picking the top in each category
    for pos in [1, 2, 3, 4]:
        pos_data = player_data[player_data['element_type'] == pos]
40
```

```
        top_by_points = pos_data.nlargest(10 if pos == 1 else (40 if pos in [2, 3] else 20), 'total_points')

        top_by_ppc = pos_data.nlargest(10 if pos == 1 else (40 if pos in [2, 3] else 20), 'points_per_cost')

        # Here we combine the picks
        selected_players.extend(top_by_points.index.tolist())
        selected_players.extend(top_by_ppc.index.tolist())

    # Here we remove the duplicates
    selected_players = list(set(selected_players))
    return player_data.loc[selected_players].reset_index(drop=True)

def validate_team(team_ids, player_data):
    """This checks if a team at a specific GW meets the constraints."""

    # This gets the IDs of the players in the team
    team = player_data[player_data['id'].isin(team_ids)]

    # Checking we have 11 players
    if len(team) != 11:
        return False

    # Checking we have the right number of players in each position
    if team['element_type'].value_counts().to_dict() != positions:
        return False

    # Check budget
    if team['now_cost'].sum() > budget:
        return False

    # Check Premier League Club limits
    team_counts = team['team'].value_counts()
    if (team_counts > max_players_per_team).any():
        return False

    return True

def generate_random_team(player_data):
    """This makes a random valid initial FPL team"""

    while True:
        team_ids = []
        position_counts = {pos: 0 for pos in positions}
        team_counts = {}
        total_cost = 0

        # We start with goalkeeper and fill up
        for pos, limit in positions.items():
            pos_players = player_data[player_data['element_type'] == pos]

            # Pick a random player until we hit the position limit
            while position_counts[pos] < limit:
                # To choose the next player, find the players not in the team
                players_not_in_team = pos_players[~pos_players['id'].isin(team_ids)]

                # Filter players out who break the constraints
                valid_players = pos_players[
                    (~pos_players['id'].isin(team_ids)) &
                    (pos_players['team'].map(lambda x: team_counts.get(x, 0) < max_players_per_team)) &
                    (pos_players['now_cost'] <= budget - total_cost)
                ]

                if valid_players.empty:
```

```
                    break

                # Pick a random player
                player = valid_players.sample(1).iloc[0]

                # Check the player fits the budget
                if total_cost + player['now_cost'] > budget:
                    break

                # Add player to team
                team_ids.append(player['id'])
                position_counts[pos] += 1
                team_counts[player['team']] = team_counts.get(player['team'],0)
 + 1
                total_cost += player['now_cost']

        # Once we have the team, just double check it's valid
        if len(team_ids) == 11 and validate_team(team_ids, player_data):
            return team_ids

def generate_random_transfers(starting_team, player_data):
    """This function takes our random starting team and generates a subsequent
 random transfer strategy."""

    transfers = []
    current_team = starting_team.copy()

    # Our transfers start for GW2 and end in GW38
    for gw in range(2, 39):
        # We don't always have to transfer so add a small chance of skipping
        if random.random() < 0.2:
            continue

        current_team_data = player_data[player_data['id'].isin(current_team)]

        # Randomly pick a player to transfer out
        player_out = random.choice(current_team)
        player_out_data = player_data[player_data['id'] == player_out].iloc[0]

        # Get a list of players not in the current team in this position
        same_pos_players = player_data[
            (player_data['element_type'] == player_out_data['element_type']) &
            (~player_data['id'].isin(current_team))]

        # Calculate current team cost and team counts
        current_cost = current_team_data['now_cost'].sum()
        team_counts = current_team_data['team'].value_counts().to_dict()

        # Decrease the team counter for the player we're transferring out
        team_counts[player_out_data['team']] = team_counts.get(player_out_data[
'team'], 0) - 1

        # Calculate remaining budget after transferring out
        remaining_budget = budget - (current_cost - player_out_data['now_cost'
])

        # Filter out players that would break constraints
        valid_players = same_pos_players[
            (same_pos_players['now_cost'] <= remaining_budget) &
            (same_pos_players['team'].map(lambda x: team_counts.get(x, 0) <
 max_players_per_team))]

        if valid_players.empty:
            continue

        # Randomly pick a player to transfer in
        player_in = valid_players.sample(1).iloc[0]
```

```
            # Make the transfer
            current_team.remove(player_out)
            current_team.append(player_in['id'])

            # Add transfer to list
            transfers.append({
                'gameweek': gw,
                'transfer_out': player_out,
                'transfer_in': player_in['id']
            })

    return transfers

def generate_random_solution(player_data):
    """This function just combines our 'make initial team' and 'make transfer strategy' functions."""
    starting_team = generate_random_team(player_data)
    transfers = generate_random_transfers(starting_team, player_data)

    return {
        'starting_team': starting_team,
        'transfers': transfers
    }

def get_team_at_gw(solution, gw, player_data):
    """This function gets a team at specific GW by running the transfers."""
    current_team = [int(player_id) for player_id in solution['starting_team']]

    # Sort transfers by gameweek to ensure they're applied in order
    sorted_transfers = sorted(solution['transfers'], key=lambda x: x['gameweek'])

    # Apply the weekly transfers
    for transfer in sorted_transfers:
        if transfer['gameweek'] <= gw:
            current_team.remove(int(transfer['transfer_out']))
            current_team.append(int(transfer['transfer_in']))

    return current_team

def mutate(solution, player_data):
    """This function implements the mutation strategy outlined in Mutation Strategy.txt."""
    # Create a copy of the solution to try and mutate
    original_solution = copy.deepcopy(solution)

    # Keep trying to mutate until we get a valid one
    while True:
        sol_to_mutate = copy.deepcopy(original_solution)
        target_gw = random.randint(1, 38)

        if target_gw == 1:
            # Need to mutate starting team
            current_team = sol_to_mutate['starting_team'].copy()
            player_to_replace = random.choice(current_team)

            # Get the data of the player to replace
            player_data_row = player_data[player_data['id'] == player_to_replace].iloc[0]
            pos = player_data_row['element_type']

            # Calculate budget for new player
            current_cost = sum(player_data[player_data['id'] == player]['now_cost'].iloc[0] for player in current_team)
            player_out_cost = player_data_row['now_cost']
            budget_for_new = budget - (current_cost - player_out_cost)
```

```
            # Find valid replacements
            players_of_pos = player_data[player_data['element_type'] == pos]
            valid_replacements = players_of_pos[
                (~players_of_pos['id'].isin(current_team)) &
                (players_of_pos['now_cost'] <= budget_for_new)
            ]['id'].tolist()

            if not valid_replacements:
                continue

            # Pick a replacement randomly
            replacement = random.choice(valid_replacements)

            # Update the starting team
            sol_to_mutate['starting_team'].remove(player_to_replace)
            sol_to_mutate['starting_team'].append(replacement)

            # Update future transfers
            # Find lowest gameweek transfer that transfers out
player_to_replace
            transfer_out = next((t for t in sol_to_mutate['transfers'] if t['
transfer_out'] == player_to_replace), None)
            if transfer_out:
                transfer_out['transfer_out'] = replacement

            # Find lowest gameweek transfer that transfers in replacement
            transfer_in = next((t for t in sol_to_mutate['transfers'] if t['
transfer_in'] == replacement), None)
            if transfer_in:
                transfer_in['transfer_in'] = player_to_replace

        else:
            # Need to mutate transfer strategy
            team_before = get_team_at_gw(sol_to_mutate, target_gw - 1,
player_data)

            # See if there's an existign transfer planned for this week
            existing_transfer = next((t for t in sol_to_mutate['transfers'] if
t['gameweek'] == target_gw), None)

            if existing_transfer:
                # Store the players involved in the transfer we are deleting
                deleted_out = existing_transfer['transfer_out']
                deleted_in = existing_transfer['transfer_in']

                # Remove the existing transfer
                sol_to_mutate['transfers'].remove(existing_transfer)
            else:
                deleted_out = None
                deleted_in = None

            # Pick a random player to replace
            player_to_replace = random.choice(team_before)

            # Get the data of the player to replace
            player_data_row = player_data[player_data['id'] ==
player_to_replace].iloc[0]
            pos = player_data_row['element_type']

            # Calculate budget for new player
            current_cost = sum(player_data[player_data['id'] == player]['
now_cost'].iloc[0] for player in team_before)
            player_out_cost = player_data_row['now_cost']
            budget_for_new = budget - (current_cost - player_out_cost)

            # Find valid replacements
```

```
            players_of_pos = player_data[player_data['element_type'] == pos]
            valid_replacements = players_of_pos[
                (~players_of_pos['id'].isin(team_before)) &
                (players_of_pos['now_cost'] <= budget_for_new)
            ]['id'].tolist()

            if not valid_replacements:
                continue

            # Pick a replacement randomly
            replacement = random.choice(valid_replacements)

            # Add the new transfer
            sol_to_mutate['transfers'].append({
                'gameweek': target_gw,
                'transfer_out': player_to_replace,
                'transfer_in': replacement
            })

            # Id we had an existsing transfer, we need to change the future strategy so it stays valid
            if deleted_out is not None and deleted_in is not None:
                # Find lowest gameweek transfer that transfers out deleted_in
                transfer_out = next((t for t in sol_to_mutate['transfers'] if t['gameweek'] > target_gw and t['transfer_out'] == deleted_in), None)
                if transfer_out:
                    transfer_out['transfer_out'] = deleted_out

                # Find lowest gameweek transfer that transfers in deleted_out
                transfer_in = next((t for t in sol_to_mutate['transfers'] if t['gameweek'] > target_gw and t['transfer_in'] == deleted_out), None)
                if transfer_in:
                    transfer_in['transfer_in'] = deleted_in

        # Sort transfers by gameweek to ensure they're in order
        sol_to_mutate['transfers'].sort(key=lambda x: x['gameweek'])

        # Validate solution just to double check
        is_valid = validate_solution(sol_to_mutate, player_data)
        if is_valid:
            return sol_to_mutate

def calculate_season_score(solution, player_data):
    """Calculate the total season points score for a solution."""
    total_points = 0

    # Loop through each week
    for gw in range(1, 39):
        team_at_gw = get_team_at_gw(solution, gw, player_data)

        # Calculate points for this gameweek
        gw_points = sum(
            player_data[player_data['id'] == player_id][f'GW{gw} Points'].iloc[0]
            for player_id in team_at_gw)

        total_points += gw_points

    return total_points

def validate_solution(solution, player_data):
    """This function checks a solution is valid."""

    # First, we check if the transfers are valid i.e if the players mentioned are in the team
    current_team = set(solution['starting_team'])
```

```
    sorted_transfers = sorted(solution['transfers'], key=lambda x: x['gameweek'
])

    for transfer in sorted_transfers:
        if int(transfer['transfer_out']) not in current_team:
            return False
        current_team.remove(int(transfer['transfer_out']))
        current_team.add(int(transfer['transfer_in']))

    # Check each gameweek's team is valid
    for gw in range(1, 39):
        team_at_gw = get_team_at_gw(solution, gw, player_data)
        if not validate_team(list(team_at_gw), player_data):
            return False

    return True

def find_crossover_point(parents, player_data):
    """This function finds if there are any points where two parents are
identical. It only returns if there is only one crossover point. This
prevents repeated crossover with similar solutions."""

    valid_points = []

    # Check weeks 2 to 37. Crossover at GW1 or GW38 doesn't make sense.
    for gw in range(2, 38):
        team1 = get_team_at_gw(parents[0], gw, player_data)
        team2 = get_team_at_gw(parents[1], gw, player_data)

        # Check if teams are identical. We sort the lists to compare
        if sorted(team1) == sorted(team2):
            valid_points.append(gw)

    # Only return a crossover point if there is only one valid one
    return valid_points if len(valid_points) == 1 else []

def crossover(parent1, parent2, player_data):
    """This function actually carries out the crossover. It returns the parents
 if the crossover fails, and returns two offspring if it works."""

    crossover_point = find_crossover_point([parent1, parent2], player_data)

    if not crossover_point:
        return parent1, parent2

    # We extract the point from the list
    crossover_gw = crossover_point[0]

    # See what our teams look like at the crossover
    team1 = get_team_at_gw(parent1, crossover_gw, player_data)
    team2 = get_team_at_gw(parent2, crossover_gw, player_data)

    # Create two offspring by copying the parents
    child1 = copy.deepcopy(parent1)
    child2 = copy.deepcopy(parent2)

    # Swap the children's transfers after the point to perform crossover
    child1_transfers = [t for t in child1['transfers'] if t['gameweek'] <=
crossover_gw]
    child1_transfers.extend([t for t in parent2['transfers'] if t['gameweek'] >
 crossover_gw])
    child1['transfers'] = child1_transfers

    child2_transfers = [t for t in child2['transfers'] if t['gameweek'] <=
crossover_gw]
    child2_transfers.extend([t for t in parent1['transfers'] if t['gameweek'] >
 crossover_gw])
```

```
    child2['transfers'] = child2_transfers

    return child1, child2

def run_genetic_algorithm(player_data):
    """This function puts all of our functions together and runs the genetic
  algorithm.
    1. Report the best score from the generation
    2. Make a copy of the top 5% of solutions (elitism) to preserve them
    3. Look through pairs of parents, doing crossover.
    4. Mutate half of the population
    5. Perform selection: First add the elite solutions to final population,
  then use roulette wheel.
    """

    # We start timing
    algorithm_start_time = time.time()

    # Make our initial population
    population = []
    while len(population) < population_size:
        solution = generate_random_solution(player_data)
        is_valid = validate_solution(solution, player_data)
        if is_valid:
            population.append(solution)

    # We keep track of stats here
    generation_stats = []

    # Score our initial population (named Generation 0)
    initial_scores = [calculate_season_score(sol, player_data) for sol in
  population]
    best_initial_solution = population[initial_scores.index(max(initial_scores)
  )]

    # Here, we start our tracking of the best solution
    best_solution_overall = best_initial_solution
    best_score_overall = max(initial_scores)
    generation_stats.append({
        'generation': 0,
        'best_score': best_score_overall,
        'time_taken': 0})

    print(f"Generation 0 (Initial): Best Score = {best_score_overall:.2f}")

    # We run through the generations
    for generation in range(1, generations + 1):
        # We score the generation and update our best solution
        current_scores = [calculate_season_score(sol, player_data) for sol in
  population]
        current_best_score = max(current_scores)
        current_best_solution = population[current_scores.index(
  current_best_score)]

        if current_best_score > best_score_overall:
            best_score_overall = current_best_score
            best_solution_overall = current_best_solution

        # We store the generation stats
        generation_stats.append({
            'generation': generation,
            'best_score': current_best_score})

        # Here, get population for next generation
        # Make a copy of the elite solutions
        num_elite = max(1, round(len(population) * 0.05))
```

```
        # Sort the indices of the solutions by score from most to least
        sorted_indices = sorted(range(len(current_scores)), key=lambda i:
current_scores[i], reverse=True)
        elite_solutions = [population[i] for i in sorted_indices[:num_elite]]

        # Here, we run the crossover section
        new_population = []
        available_parents = list(range(len(population)))  # List of indices of
available parents

        while available_parents:
            # Pick a random parent
            parent1_idx = random.choice(available_parents)
            parent1 = population[parent1_idx]
            available_parents.remove(parent1_idx)

            # Try to crossover with each remaining parent in random order
            random.shuffle(available_parents)
            found_crossover = False

            for parent2_idx in available_parents:
                parent2 = population[parent2_idx]
                child1, child2 = crossover(parent1, parent2, player_data)

                # If we got new children, it was a successful crossover
                if child1 != parent1 or child2 != parent2:
                    new_population.extend([child1, child2])
                    available_parents.remove(parent2_idx)
                    found_crossover = True
                    break

            # If no crossover was found with any parent, we add parent1 to new
population
            if not found_crossover:
                new_population.append(parent1)

        # After crossover, we mutate half the new population
        num_to_mutate = len(new_population) // 2
        indices_to_mutate = random.sample(range(len(new_population)),
num_to_mutate)

        for i in indices_to_mutate:
            new_population[i] = mutate(new_population[i], player_data)

        # Calculate scores for the new population so we can do roulette wheel
        scores = [calculate_season_score(sol, player_data) for sol in
new_population]

        # Create next generation out of elite solutions and roulette wheel
picks
        remaining_slots = population_size - num_elite

        # Calculate total score of new population
        total_score = sum(scores)

        # Calculate selection probabilities
        selection_probs = [score/total_score for score in scores]

        # Select remaining solutions using roulette wheel
        selected_solutions = []
        for _ in range(remaining_slots):
            # Use a numpy function that does weighted random choices
            selected_index = np.random.choice(len(new_population), p=
selection_probs)
            selected_solutions.append(new_population[selected_index])

        population = elite_solutions + selected_solutions
```

```
        # Print our progress
        print(f"Generation {generation}: Best Score = {current_best_score:.2f}"
    )

    # Calculate the total time taken
    total_time = time.time() - algorithm_start_time

    # Print final stats
    print("\nFinal Results:")
    print(f"Total Generations: {generations}")
    print(f"Final Best Score: {best_score_overall:.2f}")
    print(f"Total Time Taken: {total_time:.2f} seconds")

    return best_solution_overall, generation_stats

def main():
    """This function runs the genetic algorithm and loads the player data."""
    # Start timing and counting function calls
    profiler = cProfile.Profile()
    profiler.enable()
    start_time = time.time()

    player_data = get_player_data("Player Points Database.csv")
    player_data = prune_player_data(player_data)

    # Run the genetic algorithm
    best_solution, generation_stats = run_genetic_algorithm(player_data)

    # End the timing and counting function calls
    profiler.disable()
    total_time = time.time() - start_time

    program_stats = pstats.Stats(profiler)
    print(f"\nProgram Statistics:")
    print(f"Total function calls: {program_stats.total_calls}")
    print(f"Total execution time: {total_time:.3f} seconds")

if __name__ == "__main__":
    main()
```

Bibliography

[1] Fantasy Sports and Gaming Association, "Industry Demographics," 2025. Available at: thefsga.org. Accessed: 30-Jan-2025.

[2] Mordor Intelligence, "Fantasy Sports Market Size", 2025. Available at: mordorintelligence.com. Accessed: 30-Jan-2025.

[3] E. Darby and D. Ayers, "Student, 15, tops Fantasy Premier League worldwide", 2025. Available at: bbc.co.uk. Accessed: 30-Jan-2025.

[4] YouTube, "Let's Talk FPL", 2025. Available at: youtube.com. Accessed: 30-Jan-2025.

[5] FantasyFootballHub, "Member Upgrade", 2025. Available at: fantasyfootballhub.co.uk. Accessed: 30-Jan-2025.

[6] FPLReview,"Ultimate Truth: How FPL Models Perform Relative to a "Perfect" Model", 2025. Available at: fplreview.com. Accessed: 30-Jan-2025.

[7] H. Baker, "How many atoms are in the observable universe?", 2025. Available at: livescience.com. Accessed: 30-Jan-2025.

[8] P. Plait, "Do Stars Outnumber the Sands of Earth's Beaches?", 2025. Available at: scientificamerican.com. Accessed: 30-Jan-2025.

[9] H. Kellerer, U. Pferschy and D. Pisinger, *Knapsack Problems*, Springer, 2004.

[10] The Scout, "FPL basics explained: Scoring points", 2024. Available at: premierleague.com. Accessed: 12-Feb-2025.

[11] Vastaav Anand, "FPL Historical Dataset", 2022. Available at: github.com. Accessed: 5-Mar-2025.

[12] Sky Sports, "Advanced Stats Explained", 2024. Available at: skysports.com. Accessed: 5-Mar-2025.

[13] StatsBomb, "What Are Expected Goals (xG)?", 2025. Available at: statsbomb.com. Accessed: 5-Mar-2025.

[14] Richard Routledge, “Poisson Distribution”, 2025. Available at: britannica.com. Accessed: 5-Mar-2025.

[15] Premier League, “All 380 fixtures for 2024/25 Premier League season”, 2024. Available at: premierleague.com. Accessed: 5-Mar-2025.

[16] Geeks For Geeks, “Greedy Algorithms”, 2024. Available at: geeksforgeeks.org. Accessed: 7-Mar-2025.

[17] T. Cormen, C. Leiserson, R. Rivest and C. Stein, *Introduction to Algorithms*, MIT Press, 1990.

[18] Brilliant, “Dijkstra’s Shortest Path Algorithm”, 2025. Available at: brilliant.org. Accessed: 7-Mar-2025.

[19] Brilliant, “Greedy Algorithms”, 2025. Available at: brilliant.org. Accessed: 7-Mar-2025.

[20] WsCube Tech, “Greedy Algorithms: Examples, Types, Complexity”, 2025. Available at: wscubetech.com. Accessed: 7-Mar-2025.

[21] Python, “The Python Profilers”, 2025. Available at: docs.python.org. Accessed: 20-Apr-2025

[22] Geeks For Geeks, “0/1 Knapsack Problem”, 2025. Available at: geeksforgeeks.org. Accessed: 23-Mar-2025.

[23] Chiradeep BasuMallick, “What is Dynamic Programming? Working, Algorithms, and Examples”, 2022. Available at: spiceworks.com. Accessed: 23-Mar-2025.

[24] RDSEED, “Knapsack Problem Dynamic Programming Algorithm Demonstration”, 2025. Available at: wikimedia.org. Accessed: 23-Mar-2025.

[25] Eiríkur Fannar Torfason, “Picking An Optimal Fantasy Team”, 2024. Available at: eirikur.dev. Accessed: 23-Mar-2025.

[26] Geeks For Geeks, “Prune-and-Search | A Complexity Analysis Overview”, 2024. Available at: geeksforgeeks.org. Accessed: 23-Mar-2025.

[27] German Cocca, “What is Memoization? How and When to Memoize in JavaScript and React”, 2022. Available at: freecodecamp.org. Accessed: 23-Mar-2025.

[28] C. Reeves and J. Rowe, *Genetic Algorithms - Principles and Perspectives*, Kluwer Academic Publishers, 2003.

[29] C. Darwin, *On the Origin of Species by Means of Natural Selection*, J. Murray, 1859.

[30] Emily Osterloff, “What is natural selection?”. Available at nhm.ac.uk. Accessed: 11-Apr-2025.

[31] MATLAB Help Centre, “What Is the Genetic Algorithm?”, 2024. Available at uk.mathworks.com. Accessed: 11-Apr-2025.

[32] Restack, “Genetic Algorithm Pros And Cons”, 2025. Available at: restack.io. Accessed: 11-Apr-2025.

[33] M. Flinders and I. Smalley, “What is parallel computing?”, 2024. Available at: ibm.com. Accessed: 11-Apr-2025.

[34] GeeksForGeeks, “Introduction to Beam Search Algorithm”, 2025. Available at: geeksforgeeks.org. Accessed: 15-Apr-2025.