COMP 4106

**Final Project: Implementing AI Searches for the NHL Fantasy Lineup Generator**

Lava Tahir 100933449

Due Date: April 19 2017

**Problem Domain:**

A fantasy sports lineup is a type of online sports game where users create teams of real players of a sport. These teams will then compete with one another based on the collective statistical performance of the players’ in real-life games. For this problem domain, the statistical performance we will factor in is goals, assists, shots, penalty minutes, saves and if a win was credited to a goalie. Each factor will have an equal weight with respect to one another. A game is won whenever a team produces more than another team based on the collective statistical performance of each respective team.

Obtaining players will initially come through a fantasy draft, where users will draft real-life players that they can then use for their fantasy lineup. For this problem domain, we will assume that I have already a list of players to choose from (20 players). Depending on the sport, a lineup can consist of different types of players. For this problem domain, we will select the sport of hockey, and the National Hockey League to obtain the statistical performance of each player. For a NHL fantasy sports lineup, 3 forwards, 2 defensemen and 1 goalie must be chosen for it to be a valid lineup. For the list of 20 players I will use, I have a roster consisting of 4 goalies, 6 defensemen and 10 forwards. Overall, using statistical analysis (4 choose 1 \* 6 choose 2 \* 10 choose 3), I will have the possibility of creating 7200 possible lineups or states.

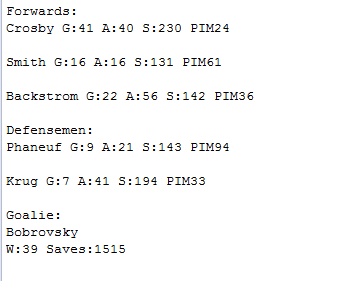
**Motivation:**

The motivation behind this final project is to create multiple AI searches that can find an optimal lineup based on each player’s statistical performance based on each player’s statistical performance history as quickly as possible. While it may seem obvious to play the players who score the most goals and assists, sometimes it is beneficial to play players who can shoot more than other players or players who get accessed the most penalties. Winning a fantasy sports league or game online can earn a highly financial award, as some players are full time fantasy sports players who earn more than they did with their past jobs. Winning a fantasy game or equivalently, finding the goal state, may be hard to accomplish, as any lineup can be used (if the lineup is valid). However, I will set up constraints that a lineup must consist of a certain number for each statistical performance.

**Design Choices Used:**

The program was designed and implemented in Java. To represent my lineup state, I needed a representation of the forwards, the defensemen and the goalie that comprise the lineup. I also required a representation of each player’s statistics, proving that the players in the lineup did pass the constraints initially set. The best way to represent the state was by using 3 lists, one containing the forwards, another containing the defensemen and the last one containing the goalie. From there I would easily be able to manipulate the lineup, which would be the three lists together. To represent the players, information containing the player’s last name, goals, assists, shots, and penalty minutes would be displayed if the player was skater, or non-goalie. For goalies, the player’s last name, wins and saves would then be displayed. For an example, see figure 1.

Figure 1: A lineup state displaying the forwards, defensemen and goalie in a lineup.



To check if a lineup was considered a goal state, the lineup must pass a set of constraints as well as be considered a legal lineup (3 forwards, 2 defensemen and a goalie are present). The constraints I created were chosen based on my experience with playing fantasy hockey in the real-world. See Table 1 for the goal state constraints.

Table 1: Cumulative constraints a lineup must pass to be considered a goal state

|  |  |
| --- | --- |
| Statistic | Constraint Value |
| Goals | 90 |
| Assists | 200 |
| Shots | 100 |
| Penalty Minutes | 100 |
| Wins | 35 |
| Saves | 1500 |

With these constraint values, any lineup that is generated by the AI searches has the potential to be considered a goal state. Some players may be chosen more often than other players, because those players are deemed more valuable to the lineup. For example, a player that has 40 assists to date may be chosen more frequently than a player that has more goals, because of how high the constraint value is set for assists in comparison to goals.

**AI techniques used:**

4 AI techniques were used to find the goal lineups. Each are listed below and will be described in detail and their results will be shown in their respective sections:

Breadth First Search, A Star (one heuristic function only), Simple Hill Climb and Steep Hill Climb.

**Breadth First Search**

The Breadth First Search is a search algorithm that performs a breadth search on a searching tree. Starting at the root, the search explores the neighbors first, checking to see if the goal state is found, and continues to explore the next level of neighbors if a goal state is not found. As for all the algorithms, a state will be deemed as a goal state if it passes the threshold values initially set. The results for 10 rounds of the Breadth First Search algorithm on the problem domain are shown in table 2.

Table 2: Results of 10 Breadth First Searches

|  |  |
| --- | --- |
| Trial | Number of Moves Made until Optimal Solution |
| 1 | 7 |
| 2 | 7 |
| 3 | 8 |
| 4 | 8 |
| 5 | 7 |
| 6 | 7 |
| 7 | 7 |
| 8 | 7 |
| 9 | 8 |
| 10 | 7 |
| Average | 7.3 |

Because the Breadth First Search algorithm finds the shortest path to the goal state, it is no surprise that the algorithm can find a goal state in 7 or 8 moves. The reason why some trials take 8 moves while other trials take 7 is because any state can be considered the goal state. Therefore, depending on who is initially chosen in the lineup, or the root player chosen, the total number of moves that are made can differ.

When implementing the breadth first search for the fantasy lineup generator, one of the benefits that was observed was that the algorithm would find the shortest path to the goal state, so a shorter run time was achieved. Another benefit was that due to the completeness of the algorithm as well as the finite number of states, the algorithm would always find the goal state. One of the disadvantages of the algorithm was during the checking of neighbor states if the current state was not deemed a goal state. The algorithm was not smart enough to determine that the reason why the current state is not considered the goal state was that it was currently incomplete, i.e., the lineup currently was not a legal state with the legal number of forwards, defensemen and goalie. For example, if the current lineup only had 1 defensemen, the algorithm would correctly deem that it was not considered to be a goal state and check the neighbor. However, the neighbor state would not always add in a defenseman to make it a legal lineup, as it would sometimes add or remove a forward or goalie. Therefore, useless nodes were searched which added time and space to the algorithm. Overall, the breadth first search algorithm was one of the best algorithms to use for this problem domain. One possible addition to the algorithm for this problem domain would be to filter out nodes that did not contain a legal lineup, which would reduce the space and time complexity to the algorithm.

**A Star Search:**

A-Star search is a modification of the Best-first search algorithm, and it is used to find the optimal path. The A-Star search requires the sum of two functions. One of the functions is the estimator for the cost from the current state to the goal state, using a heuristic function. Another function is which calculates the cost so far to reach the current state. The heuristic function used returned the total score of the current lineup. An illegal lineup would be given the score of 0, a goal state lineup would be given the score of 10,000 and lastly, any legal lineup would be given a score that summed up all the player’s statistics in the lineup. Table 3 displays the results of 10 trials running the A-Star search algorithm with the cumulative player stats heuristic.

Table 3: Results of 10 A-Star Searches

|  |  |
| --- | --- |
| Trial | Number of Moves Made until Optimal Solution |
| 1 | 7 |
| 2 | 8 |
| 3 | 8 |
| 4 | 7 |
| 5 | 8 |
| 6 | 8 |
| 7 | 7 |
| 8 | 7 |
| 9 | 7 |
| 10 | 7 |
| Average | 7.4 |

As the A-Star algorithm uses an admissible heuristic, which is to state that the heuristic never overestimates the cost to the goal state, then the algorithm is optimal, as shown with the results. Along with the results from the Breadth First Search algorithm, the A-Star results also show that it may take 7 or 8 moves to find the goal state, due to the dependency on the root of the node.

When implementing A-Star search for the fantasy lineup generator, one of the advantages was that because I was using a heuristic which never overestimates the cost to the goal state, A-Star finds the optimal path to the solution. One disadvantage of the algorithm was finding a heuristic which will always underestimate the cost. Finding a heuristic like that was time consuming, as I had experimented with different heuristics, such as the number of players in place heuristic. This heuristic function would return the total number of players that were in the lineup. This heuristic was not useful as it did not take into consideration how far the current state was in respect to the statistics. Overall, the A-Star search algorithm was equivalent to the Breadth First Search algorithm in respect to the number of moves being made to find the goal state. One possible addition to the algorithm would be to only generate nodes that contained legal game states, i.e., game states with the legal number of players, as that would remove searching unnecessary nodes.

**Simple Hill Climbing Search:**

The Simple Hill Climbing Search is a type of a local search where it starts with a certain state, then incrementally changes the state to find a better solution. The incremental changes keep occurring until no further improvements can be found. This solution is best for finding a local optimum, a solution that can’t be improved by a neighboring state. The results for the Simple Hill Climbing Search are shown in Table 4.

Table 4: Results of 10 Simple Hill Climbing Searches

|  |  |
| --- | --- |
| Trial | Number of Moves Made until Optimal Solution |
| 1 | 9 |
| 2 | 9 |
| 3 | 9 |
| 4 | Too many/Stopped early (20) |
| 5 | 9 |
| 6 | 9 |
| 7 | Too many/Stopped early (20) |
| 8 | 9 |
| 9 | 9 |
| 10 | 9 |
| Average | 11.2 |

As shown from the Simple Hill Climb, the average for the number of moves made until the solution is higher than the Breadth First Search and A-Star algorithm. This is because the simple hill climb algorithm is not optimal, as it only checks if the neighbors are considered to be goal states or are in a better position to be a goal state for the next level of checking. Also, the algorithm would sometimes take too many moves to find the optimal solution. This is because the algorithm would get stuck at local maximums when searching. In this case, the algorithm would think that a neighbor state is in a better position than the current state, and change accordingly. Later however, it is found that the change resulted in a state in which no goal state can be found, so it would be stuck. Overall, the Simple Hill Climbing algorithm was worse in comparison to the Breadth First Search and the A-Star algorithm as it took more moves to find the goal state lineup. While implementing the Simple Hill Climbing algorithm, one advantage to the algorithm in comparison to the other algorithms was that it was easier to code. In Java, it only took 10 lines of code to implement the algorithm. One way to further improve the algorithm would be to allow a way to backtrack to a crucial node to solve the issue of getting stuck at local maximums.

**Steep Hill Climbing:**

The Steep Hill Climbing algorithm is a variation of the Simple Hill Climbing algorithm where when observing the successors of a node, each successor is compared and the closest to the goal state is chosen. The Steepest Hill Climbing as well as the Simple Hill Climbing both do not provide the optimal solution as they are both local searches. Table 5 displays the results of 10 trials using the Steepest Hill Climbing Search algorithm.

Table 5: Results of 10 Steepest Hill Climbing Searches

|  |  |
| --- | --- |
| Trial | Number of Moves Made until Optimal Solution |
| 1 | 9 |
| 2 | 9 |
| 3 | 9 |
| 4 | Too many/Stopped early (20) |
| 5 | 8 |
| 6 | 9 |
| 7 | Too many/Stopped early (20) |
| 8 | 9 |
| 9 | 9 |
| 10 | 8 |
| Average | 11 |

The Steepest Hill Climb performs better than the Simple Hill Climbing algorithm yet it still suffers from the same problems the Simple Hill Climbing algorithm has. The Steepest Hill Climb suffered from the local maximum problem, due to the process of taking the neighbor that was initially thought to be the best, while realizing later in the searching process that the neighbor could only lead to a state was could not ever be a goal state. One advantage to the steepest hill climb was that when choosing a successor, it would evaluate all the neighbors and choose the best one. This would then be able to find a goal state in a lesser number of moves, as shown in Trial 5 and Trial 10. One improvement to the algorithm that could be added is to be able to pick a random node somewhere else in the search to avoid the local maximum problem.

**Conclusion:**

Out of the four search algorithms that were implemented, the A-Star search as well as the Breadth First Search both provided solutions to the problem domain every time as well as providing the solution in the least possible moves. The Simple Hill Climbing Search was the worst algorithm, as it took the longest time and would get stuck at local maximums while searching.

References:

Cherry. “The Computer That Beat Two Million Humans at Fantasy Football” (2012).0

Oommen, Tou Ng. COMP 4106 Lecture Notes (2017).

Appendix:

To run the program, go to the link <https://github.com/lavatahir/NHLFantasyAI> and download the repository or to clone the link. Then open Eclipse and import the project. Run the project and input the desired search:

A-Star: for A-Star search

BFS: for Breadth First search

SHC: For Simple Hill Climb search

SSHC: For Steep Hill Climb search.

For the SHC and SSHC, the algorithm will run indefinitely if a solution is taking too long due to local maximums that were discussed in their respective sections. Sometimes it will stop early as the program catches the local maximum while other times it will not catch it. If this occurs, just stop the program.