Decision and Performance Analysis of an Artificial Intelligence Player

***Abstract*—In this paper, we create an artificial intelligence player and measure its performance by changing and modifying its parameters to answer how much and how alpha-beta pruning, cache, classes and depth levels influence decision performance of artificial intelligence player on a minimax-based game.**

Keywords—minimax, depth, cache, duration, analysis, alpha-beta.

I. Introduction

In this project, first, we show how to generate successors list of a state. Second, we create a random player that chooses a move from generated successors. Third, we explain our artificial intelligence player basics and build a game manager in order to provide options to users to choose which players in the list; artificial intelligence player, random player, human player and basic player, are going to play the game. Finally, we indicate our saved records to propose an optimised artificial intelligence player by analysing its performance according to changing parameters.

II. Method

A. Successor Generator and Players

According to game rules, we have two move types; Capping Move and Collatz-Ulam Move [1]. To create all the possible nodes, we created combinations of the state to sum them for a move by avoiding recap. The tricky point was that we sorted successors and used 'set' command in Python, that eliminates same elements in the list, to avoid from duplicated successors. Secondly, we checked each heap in the state and implemented Collatz-Ulam to create all the successors. That enabled us to create Basic Player, Random Player and Human Player. Random Player moves randomly choosing a state from generated successors. On the other hand, Basic Player checks '1' in the successors' list, that is a score state, to select that move, if it does not detect any score position, then plays like a random player. Human player, since it is managed by a human, we only provide successors to users for deciding a move (Fig. 1).

```
Current State is: ('max', (8, 4, 5, 3), 0, 0)
0:(5, 7, 8)
1:(3, 4, 8, 16)
2:(4, 8, 8)
3:(4, 5, 8, 10)
4:(3, 5, 10)
5:(4, 5, 10)
6:(3, 4, 4, 4, 5)
7:(2, 2, 3, 5, 8)
8:(3, 8, 9)
9:(3, 4, 10)
Choose your move number:
```

Figure 1: Human Player's Successor List

The author uses 'we' as an explanation style. The project has been conducted individually.

B. Artificial Intelligence Player

Creating an 'Artificial Player' may become a difficult process. For example, SuperNim [2] game does not allow us to compute the whole tree since it has many successors, we had to build it with some of these implementations; Minimax, Alpha-Beta Pruning and MCTS [3]. We preferred to apply Minimax first. However, we needed to use depth and heuristic method for tree cut-off. Therefore, the depth and heuristic method could enable us to estimate only the successors which correspond to depth level. That could lead to increase respond's speed exponentially. However, the heuristic method had to be logical. Otherwise, the computer answer would fail with a loss or illogical move. Another extension was that we used Alpha-Beta Pruning to cut the successors which eliminates unnecessary successors while searching depth in Minimax tree structure (Fig. 2). These both implementations contributed us to programming an artificial player. For MCTS implementation, we discuss this part at the end of paper 'Discussion' whether it is strictly necessary or not.

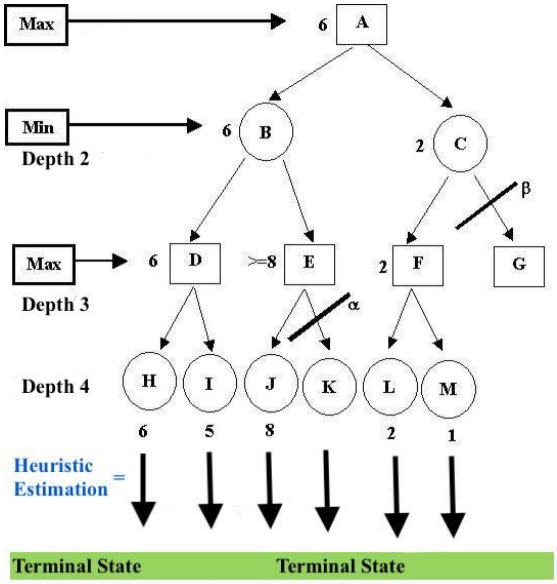

Figure 2: Minimax Tree Structure with Alpha-Beta Pruning and Heuristic Usage (*Modified from* [3])

As a heuristic method, we used 'Manhattan Distance' [4] that estimates how many steps are remaining to reach score position '1' for each stick by using the Collatz-Ulam move and ignoring the Capping move. This simple method, even though it does not give us a global optimum solution, it works logically to collect score and decreases process's duration extremely. In Fig. 2, after computer estimates first six level, that is our default depth value, it starts to calculate heuristic values of the last level heaps that we are going to call 'static values'.

Manhattan Distance was returning with increasing values. For example, four sticks need two steps to reach terminal state '1' with Collatz-Ulam that equals two points for heuristic, but nine sticks need 19 steps (19 points). We preferred to convert this values with decreasing values using "(1)" for 'Max' player. As for 'Min' player, we simply multiply equation with '-1' since its purpose to decrease profitability by choosing a minimum value of successors' utilities. That equation provides us 81 points for 9 sticks instead of 19 as a utility (heuristic) value. Therefore, the algorithm can try to maximise its utility value for each successor among these stick's heuristic values to reach terminal position. As a result, we can assign terminal states (win or loss states) '100' and '-100' to have a sensible tree. Therefore, when Artificial Intelligence player detects a score position or anticipates score position, it tends to play this move to get a score.

$$Heuristic\ List = [100 - Remaining\ Collatz\ Step\ for\ Each\ Stick\ in\ the\ Heap\ ] \quad (1)$$

We determined depth level as 6 for a starting position as we discussed earlier. In addition, (2, 5, 6, 7), (3, 8, 8, 11, 21) and (5, 9, 8, 97) have been used during experiments as initial states respectively.

The artificial intelligence player's algorithm essentially consists of these steps:

- Creating first successors of input state which equals depth level 1 just as Fig. 1 successors.
- Implementing minimax, together with alpha-beta and heuristic, for each successor. The computer evaluates static values and then returns with their heuristic (utility) values.
- Comparing utility values with bottom-up approach considering its player's turn, those are min and max. E.g. Let's assume we have ('Max', (8, 4, 5, 3), 0, 0) as an initial state. The algorithm compares utilities to detect min value of depth level 6, then a max value for level 5, min value for level 4, max value for level 3 and min value for level 2 respectively. This process can resemble a depth-first search in some way. At the end of this algorithm, we obtain a utility array of moves such as '[97, 81, 93, 75, 100, 90, 94, 88, 90]'.
- Merging utilities with their equivalent successors which are in Fig. 1 and choosing maximum value of the utility array as a move (output) since the initial state started with a max player.

C. Game Manager

As we have many players, we have created a game manager to provide an ability to users to determine players. The game manager function asks users to determine first and second players. After this step, it creates a player array which includes chosen players list. Therefore, the main function could manage turns by switching it. While observing the game, we realised that 'Artificial Player vs Random Player' matches usually end when a game score reached approximately 800-900. As a result, we decided to limit it 200 moves, 100 per each player. It is important to note that all the time-duration experiments have been conducted with 200 moves. After this arrangement, the games resulted in approximately 50-60 score for artificial intelligence player with a shorter time that alleviated our experiments. We can assume that Random Player's move duration was small enough to be ignored, approximately 0.05 second for a move. Another reason to ignore Random Player's effects is that moves have been included in all the experiments which will be compared. Therefore, we can accept it as a bias (noise) value for all the tests.

D. Performance and Duration Analysis

Many experiments later, we achieved to record our results with Python codes. Pandas, Matplotlib and NumPy [5] contributed us to obtain experiment results, such as duration, random player's score, artificial intelligence player's score together with visualising data efficiently. As a measurement, we determined several parameters to compare; these are cache, classes in part of object-oriented programming, alpha-beta pruning and depth levels. Mainly, we decided to start experiments with our default parameters; with cache, classes, alpha-beta and depth level 6. As a result, all the graphs will be compared with this initial default choice.

First experiments have been implemented between alpha-beta pruning and without alpha-beta pruning. To plot results, we used matplotlib library. The results include only 15 experiment duration which has been attained from (3, 8, 8, 11, 21) initial state (Fig. 3). Experiments have been conducted on a personal computer MacBook Pro 2013 early model by using nearly same memory and CPU ratios in order to observe parameters effects.

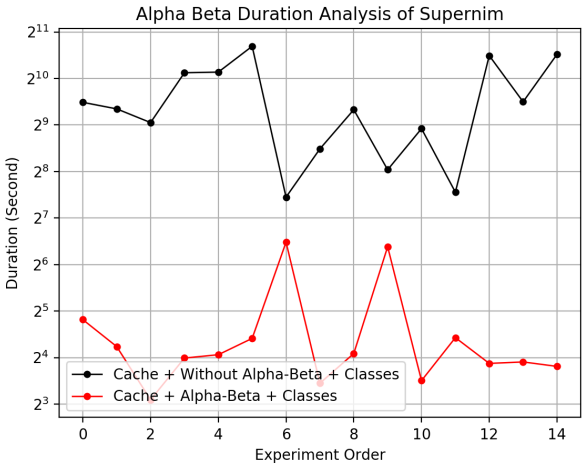

Figure 3: Alpha-Beta Pruning Effects on Performance

Second experiments have been conducted between classes in object-oriented programming and without classes. Firstly, we used functions to process our algorithm. Then, we used object-

oriented programming style and created two extra classes to run it. The game has been run 45 times for this experiment with three different initial states those are (2, 5, 6, 7), (3, 8, 8, 11, 21) and (5, 9, 8, 97) respectively to observe algorithm reactions according to changing initial states (Fig. 4)

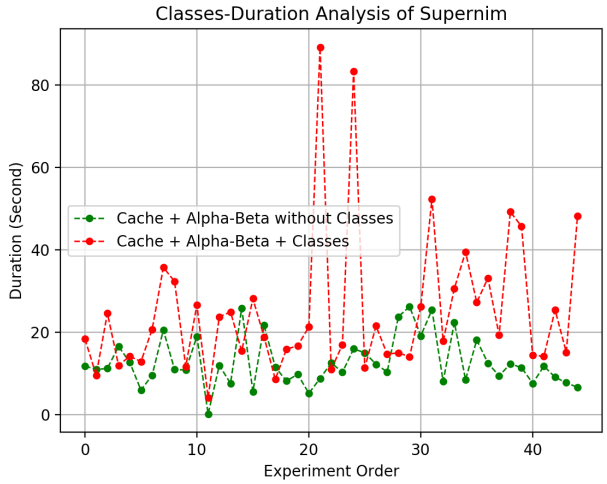

Figure 4: Object-Oriented Programming Effects on Performance

Third experiments have been conducted between storing values in the cache and without storing in the cache. To create a cache, we had two options. One of them was programming a Python dictionary and removes scores from the state. For example, if we remove scores from ('Max', (8, 4, 5, 3), 0, 0), we only obtain ('Max', (8, 4, 5, 3)) that would alleviate values to store and recall. Since scores are not strictly necessary in the state, we could have removed it. However, after planning some ideas that we will discuss this in 'Discussion' section, we realised that a ready cache tool would be better to use. Therefore, the second idea was that Last Recent Used (LRU) cache programmed by Google-Android was implemented for these experiments (Fig. 5) [5]. While doing this experiment, we

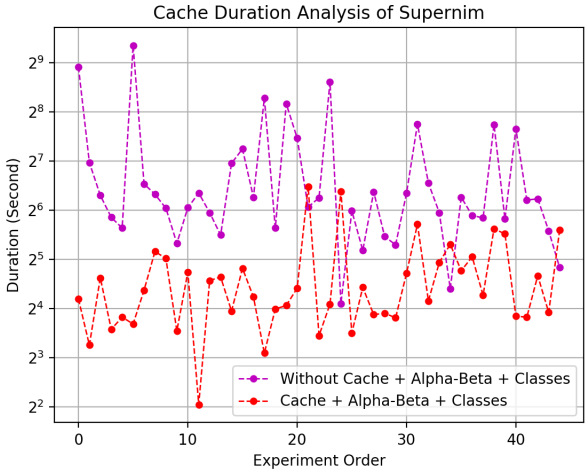

Figure 5: LRU Cache Effects on Performance

followed a black box approach and limited the cache as maximum 128 value for storing that is a default value written as 'None' in the code. We run it 45 times to record with same starting positions in the second experiment.

The fourth experiments have been conducted between depth levels to observe how much it will increase the speed and decrease the score of artificial intelligence player (Fig. 6). We also recorded game score with their means, standard deviations, max values and min values. Since we had many parameters during analysing, we preferred to show them with notations (Table I).

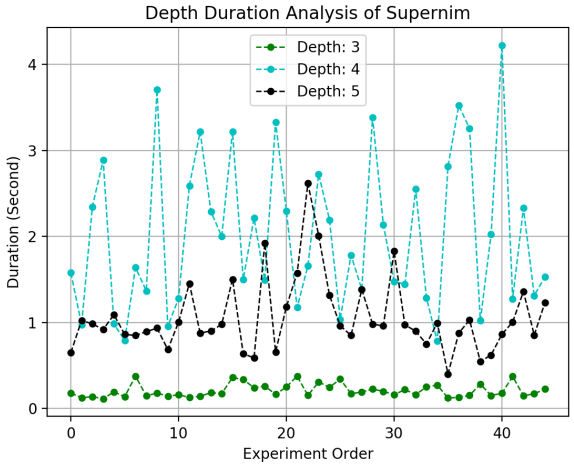

Figure 6: Depth Levels' Effects on Duration

All the tables and figures in this project were obtained from Python libraries; Pandas, Numpy and Matplotlib. As a result, to run 'statistis_nim.py' file, the users need to download these libraries to see these pictures as an output.

Table II: Experiment Features and Their Notations

Features	d1	d2	d3	d4	d5	d6	d7	d8
Cache	+	+	+	+	-	+	+	+
Classes	-	-	+	+	+	+	+	+
Alpha-Beta	+	-	+	-	+	+	+	+
Depth	6	6	6	6	6	3	4	5

Plusses (+) show which features included, minuses (-) show which features are not included. For example, if we look at d2, we can say that 'd2' named experiments have been conducted with cache programming and six depth level; however, classes and alpha-beta have not been used for these experiments. By applying those notations, we are able to show the statistical values of 8 experiments in one table (Table III)

Table IV: Statistical Data of AI Player vs Random Player Experiments

major	minor	d1	d2	d3	d4	d5	d6	d7	d8
	count	45	30	45	15	45	45	45	45
max	Time	26	3029	89.2	1653	650	0.37	4.2	2.6
	Random Score	14	12	14	12	20	10	18	8
	AI Score	66	72	76	62	76	30	76	28
min	Time	0.11	11.1	4.12	174	17.1	0.11	0.78	0.40
	Random Score	0	0	0	0	0	0	0	0
	AI Score	6	16	8	20	12	4	12	0
mean	Time	12	466	25	768	118	0.2	2.0	1.0
	Random Score	4.1	4.46	3.37	3.73	4.35	1.82	5.64	2.35
	AI Score	34.2	36.7	32.0	34	31.6	15.3	35.2	12.4
std	Time	6	512	17	481	126	0.07	0.88	0.42
	Random Score	3.8	3.1	3.0	2.8	4.2	2.1	4.3	2.5
	AI Score	13.3	11.2	12.4	12.7	13.7	6.99	12.7	7.22

III. Conclusion

First experiments illustrated that alpha-beta pruning has a great impact when we compare with the other parameters. Using alpha-beta pruning contributed to experiments being almost 30 times faster. While alpha-beta implementation's mean (d3), is 25 second, without alpha-beta (d4), the mean of duration increased to 768 seconds. Even, it was observed to be up to 1653 seconds for a match between Random vs Artificial Intelligence Player.

Second experiments showed object-oriented programming making the process slower nearly two times (Fig. 2) when compare d1 vs d3. Classes implemented games averagely finished it 25 seconds, while without classes was finishing it 12 seconds.

Third experiments indicated the mean of duration of d5 is 118 that means applying LRU based cache for storing values (d3) might speed up 4.8 times when compared with not using the cache (d5).

Fourth experiments posed that depth levels' time average can be ordered as 'd7 > d8 > d6' that means depth level 4 lasted more than depth level 5 and depth level 3. While depth level was taking 4 seconds for a battle, depth level 3 and 5 took 0.2 and 1 second respectively in average. The fastest level has been observed as depth level 3. Depth level 5 indicated 'draw' sometimes instead of full win. Depth level 4 collected the most AI score that equals 35.2 while level 3, level 5 and level 6 were achieving 15.3, 2.4 and 32 respectively as average.

The longest games have been observed in d4 experiments considering average that was 768 second. However, d2 has the winner of the slowest game competition by reaching 3029 second that equals 50 minutes nearly.

If we try to answer our research question, we highly recommend using alpha-beta, cache and without classes for fast performance. Also, Depth level 4 should be chosen to create faster and genuine artificial player for this game and algorithm.

IV. Discussion

Before Starting the project, we aimed to create an artificial player which can win against all the human players and algorithms. However, even though the AI player defeats the random player, it only focuses to gather scores, that is not exactly a winning algorithm essentially if we consider '100 move rule'. To create a winning algorithm, we were planning to use 'move counter' to implement a better heuristic that might be MCTS. In sum, when the move counter reaches 100 moves, AI has to move last score position to win. Then, the game should be ended. Otherwise, our Basic Player vs AI Player battle always results in a draw or maximum two scores difference depending on the initial states.

Some experiments were limited since they take too much time. Therefore, we had to limit them, especially, those without alpha-beta pruning implementations. Also, game rules limits creating a complex move. So, deeper analysis or complex algorithms may not change anything remarkably. Even our Basic player cannot be defeated by any Human or AI player if we ignore 100 moves rule.

Some results were interesting. We were expecting that depth level 4 should take less time than depth level 5; however, that did not happen. Even depth levels were much slower than odd depth levels. Nevertheless, they did not result in any 'draw' unlike depth level 5 and collected highest scores for AI player. When we consider those parameters; AI scores and 200 moves for a game, 4 second duration time is an extremely short time in comparison the other levels, that was not expected.

For future extensions, endgame scenarios and MCTS algorithm can be implemented to defeat Basic Player or Human Player. In this way, the algorithm can consider remaining moves all the time with score analysis. Let's assume; it plays against a Random Player, and when the list of sticks is extended a lot, AI player can explore score positions to increase its score while Random Player's score probability is low. Another extension can be changing cache method to increase the speed of the program and decrease RAM usage.

References

[1] S. O'Connor, J.J.; Robertson, E.F. (2006). "Lothar Collatz". St Andrews University School of Mathematics and Statistics, "No Title."

[2] P. Daniel, "Super Nim," *Univ. Hertfordshire, Theory Pract. Artif. Intell.*, 2018.

[3] "Minimax, Alpha-Beta." [Online]. Available: http://www.cs.nott.ac.uk/~pszgxk/courses/g5aiai/005gameplaying/game-playing.htm. [Accessed: 05-Apr-2018].

[4] BLACK and P. E., "'Manhattan distance' 'Dictionary of algorithms and data structures,'" *http://xlinux.nist.gov/dads//*.

[5] "LruCache | Android Developers." [Online]. Available: https://developer.android.com/reference/android/util/LruCache.html. [Accessed: 05-Apr-2018].