

|  |  |  |  |
| --- | --- | --- | --- |
| Course Code | : | SOF106 | |
| Course Name | : | Principles of Artificial Intelligence | |
| Lecturer | : | Xiao Han | |
| Academic Session | : | 2019/09 | |
| Assessment Title | : | Final project | |
| Submission Due Date | : | 23 December 2019 | |
| Prepared by |  |  | |
| : | Student ID | Student Name |
|  | SWE1809653 | Kong Loke Yi |
|  | SWE1809657 | Sii Yew Hei |
|  | SWE1809658 | Tan Li Ying |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
| Date Received | : |  | |
|  |  |  | |
| Feedback from Lecturer:  Mark: | | | |

**Own Work Declaration**

I/We hereby understand my/our work would be checked for plagiarism or other misconduct, and the softcopy would be saved for future comparison(s).

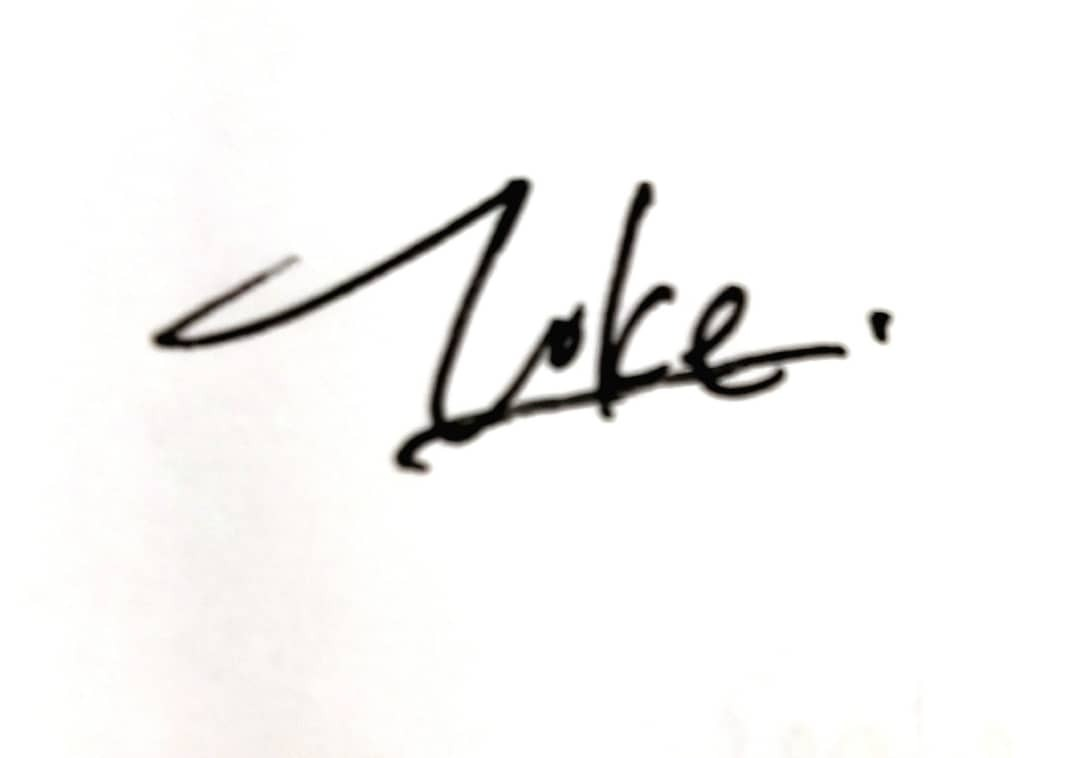
I/We hereby confirm that all the references or sources of citations have been correctly listed or presented and I/we clearly understand the serious consequence caused by any intentional or unintentional misconduct.

This work is not made on any work of other students (past or present), and it has not been submitted to any other courses or institutions before.



(Sii Yew Hei)

(Kong Loke Yi)



Signature:

(Tan Li Ying)

Date: 22 December 2019

Table of Contents

[Section I: Introduction 4](#_Toc27554896)

[Section II Literature Review 7](#_Toc27554897)

[Section III Infrastructure and methodology 16](#_Toc27554898)

[Section IV Error Analysis and Experiments 37](#_Toc27554899)

[Section V: Conclusion 39](#_Toc27554900)

[References 40](#_Toc27554901)

**Title: Alpha-beta pruning applied on the Mancala board game**

**Section I: Introduction**

Mancala is a traditional game in many countries and each country have different version. While in Malaysia, there exists a Malay traditional board game named as *Congkak* which have similar rules and required equipment. To continuously inherit the traditional culture of Malaysia, technology advancement need to be applied on this classic game. Therefore, this game is implemented and one of the players is made as AI (artificial intelligence) player to solve the problem that the need of two players for the game. The implementation the details of the Mancala board game will be discussed in this paper.

**What is Mancala?**

Mancala is as known as one of the older most board game which used to be played widely. Basically, it is a game with minimum two players who played with small stones or beans and row of holes in the earth as well as in the board. Players at least two, are required to play one round of mancala.

**How to play Mancala?**

Goal:

To collect as many stones in own house as possible. The player with the most stones in her or his house at the end of the game wins.

Set up:

Fill four stones or seeds in each of the six holes on the players’ boards. The colour of the seeds or stones does not matter.

Pits, cups, holes or hollows (in the whole proposal, namely holes)

seeds or stones (namely stones)

store, house, Mancala, or capture pit (namely houses in this paper)

**Methods of playing:**

Starting of the game

A player picks up all the stones in one hole on a turn and put the stones one by one into the successive holes. When the end of the row is reached and come to the player’s own house, then add a stone into the house and continue to. It is possible to end up putting the stones in the opponent’s pits along the way.

Basics game rules:

1. Moves around the board in a counter-clockwise circle means to the right.
2. The house on the player’s right-hand side belongs to the player. This is the place to store the stones that the player wins.
3. The holes near the player are belong to the player.
4. Only use one hand to pick up as well as put down the stones.
5. Once the player touches the stones in a hole, he or she must move those stones.
6. Only put the stones in the house belongs to the player who is now moving the stones, not the house of the opponent.

Special rules of Mancala

1. The player can take another turn when the last stone in the player’s hand land in the player’s own house.
2. The player can keep all the stones in the opponent’s hole on the opposite side when the last stone in the player’s hand lands in one of the player’s own empty hole. Then, put those captured stones and the last stone into his or her house.

Game ending

The game is over until one player’s holes are emptied completely. The player whose pits still have stones, should collect all those stones remaining in their own pits. At the end, the total number of stones in the house will be counting all the stones in the house. Eventually, the player who holds the larger amount of the stones will win game.

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

5

4

4

4

0

5

5

1

*Figure 1-First step if it is player with orange row’s turn and the 4th pit is chosen*

**Artificial Intelligence in Mancala**

There are 4 choices of AI implementation of Mancala: Minimax, Alpha-Beta Pruning Minimax, Advanced Heuristic Minimax and Monte Carlo Tree search. All of the four manners will be elucidated in details in Section II. Nevertheless, the method implemented in the program is the Alpha-Beta Pruning Minimax and heuristics.

## **Section II Literature Review**

**Greedy Algorithm**

According to Rovaris (2017), the greedy algorithm in Mancala which is deterministic applies the rules that it captures the most stones in one move. The current state of the board and all the available moves are considered and the consequences of the moves are evaluated. Subsequently, the move which creates the greatest difference between the player’ s and the opponent’ s house is selected. This greedy algorithm is applied each time the player gets an extra move. However, this foresight has the limitation that it evaluates only to one extra turn.

The greedy strategy initializes the best value of the move to negative infinity and update the value when the value is returned from the evaluation function. The list of available moves is retrieved and the evaluation function is called which returns the difference between the player’s and the opponent’s house after the move. The current best value is updated with the greater value of the difference each time. The chosen move is returned as all the available moves are evaluated (Rovaris, 2017).

|  |
| --- |
| Greedy Algorithm  1: **function** Greedy(Board, Player, Opponent)  2: *bestValue* 🡨 - INFINITY  3: *chosenMove* 🡨 0  4: *M* 🡨availableMoves(board)  5: **fo**r all *m* in *M* **do**  6: **if** Evaluate(*m, board, player, opponent*) >= *bestValue* **then**  7: *bestValue*🡨Evaluate(*m, board, player, opponent*)  8: *chosenMove*🡨*m*  9: **end if**  10: **end for**  11: **return** *chosenMove*  12: **end function**  13:  14: **function** Evaluate(*move, board, player, opponent)*  *15: evalBoard* 🡨 DoMove*(board, move)*  16: **return** Store(*evalBoard, player*) - Store(*evalBoard, opponent*)  17: **end function** |

**Minimax**

This Minimax algorithm is one of the AI method that can solve the mancala problem (Gehlot, 2016). This algorithm is implemented by Neelam Gehlot from University of Southern California. The root node of the tree is the current state of the board. Start form the root node, this algorithm will check all the available moves in the CHILDREN function, then it we move them one by one to obtain a new state of board and act as a child of the root node. From this child node, it will process the same algorithm again to get its children nodes until it reaches a leaf that in terminal state or a depth that already predefined. At the leaf, the algorithm will call the EVALUATE function which will calculate the difference between the score in the player store and the score in the opponent store. Therefore, in MINIMAX function, if the node is the leaf node, it will return the value of the EVALUATE function.

From Line 5 to Line 12, if the algorithm is at the turn of player, it will maximize it. First, initialize the best value to -∞. After that, all the children of this node are computed using the CHILDREN function, for each child the MINIMAX is called recursively to calculate the value of each child. Then, return the maximum best value. While, from Line 13 to Line 20, if the opponent is playing now, the algorithm will start minimize. The best value will initialize to +∞ and the minimum value is chosen as best value.

|  |
| --- |
| Algorithm Basic minimax pseudo-code   1. **function** MINIMAX (*node, player, opponent, depth*) 2. **if** *node* is terminal OR *depth*=0 **then** // leaf node 3. **return** EVALUATE (*board, player, opponent*) 4. **end if** 5. **if** TURNPLAYER (*board*) = *player* **then** // player playing: maximize 6. *bestValue* 🡨 -∞ 7. *children* 🡨 CHILDREN (*board*) 8. **for** **all** *child* in *children* **do** 9. *val* 🡨 MINIMAX (*child, player, opponent, depth-1*) 10. *bestValue* 🡨 MAX (*bestValue, val*) 11. **end for** 12. **return** *bestValue* 13. **else** //opponent playing: minimize 14. *bestValue* 🡨 +∞ 15. *children* 🡨 CHILDREN *(board*) 16. **for all** *child* in *children* **do** 17. *val*🡨 MINIMAX (*child, player, opponent, depth-1*) 18. *bestValue* 🡨 MIN (*bestValue, val*) 19. **end for** 20. **return** *bestValue* 21. **end if** 22. **end function** 24. **function** EVALUATE (*board, player, opponent*) 25. **return** Store (*board, player*) – Store (*board, opponent*) 26. **end function** 28. **function** CHILDREN *(board*) 29. *M* 🡨 AvailableMoves (*board*) 30. **for all** *m* in *M* **do** 31. *child* 🡨 DoMove (*board, m*) 32. Add *child* to *Children* 33. **end for** 34. **return** *Children* 35. **end function** |

**Alpha-Beta Pruning Minimax**

The alpha-beta pruning minimax is the improved minimax algorithm which reduced the computational time and memory consumption during the evaluation. The values of alpha and beta are initialized to negative infinity and positive infinity respectively and the values are updated during the plays. The alpha value is updated when the evaluation value is higher than it while the beta value is updated when the evaluation value is smaller than it. The pruning takes place when the value of alpha is greater than beta and the evaluation function is stopped in which the remaining branches are skipped since the values do not contribute towards the result of the evaluation. The pseudocode of this method for Mancala is shown as below.

|  |
| --- |
| Algorithm AlphaBeta function  1: **function** AlphaBeta*(node, player, opponent, depth)*  2: **if** *node* is terminal OR *depth* = 0 **then** // leaf node  3: **return** Evaluate*(board, player, opponent)*  4: **end if**  5: **if** TurnPlayer(board) = player **then** // player turn: max  6: *bestValue* 🡨1  7: *children* 🡨 Children(**board**)  8: **for all** *child* in children **do**  9: *val*  AlphaBeta (*child, player, opponent, depth -* 1)  10: *bestValue* max(*bestValue, val*)  11: **end for**  12: **return** *bestValue*  13: **else**// opponent turn: min  14: *bestValue* 🡨 + INFINITY  15: *children* 🡨 Children(*board*)  16: **for all** child in children **do**  17: *val* 🡨 AlphaBeta (*child, player,opponent, depth - 1*)  18: *bestValue* 🡨 min(*bestValue, val*)  19: **end for**  20: **return** *bestValue*  21: **end if**  22: **end function**  23:  24: **function** Evaluate(*board, player, opponent*)  25: **return** Store(*board, player*) - Store(*board, opponent*)  26: **end function**  27:  28: **function** Children(*board*)  29: M 🡨 availableMoves(*board*)  30: **for all** *m* in M do  31: *child* 🡨 DoMove(*board,m*)  32: Add *child* to **Children**  33: **end for**  34: **return** **Children**  35: **end function** |

**Advanced Heuristic Minimax**

Based on the research of Divilly.C (2013), a set of heuristic is developed that can be applied in mancala game. The heuristics are shown as the following:

* H1: *Keep as many stones as possible in one hole*. This heuristic is look one move ahead, in order to keep as many as possible stones in the left-most hole on the board. Thus, at the end of the game, all the stones on a side of the board can be awarded to that player’s score.
* H2: *Keep as many stones on the player own side*. This heuristic is based on the principle of H1.
* H3: *Have as many moves as possible when choose which hole to move*. This heuristic function will choose the hole that can move more steps. It has a look ahead of one.
* H4: *Maximize the amount of stones in the player own store*. This heuristic tries to choose a move that can add more stones to its own store. It looks one move ahead.
* H5: *Move the stones from the hole that closest to the opponent side*. This heuristic look one move ahead, try to make a move from the hole that closest to the opponent’s side.
* H6: *Keep opponent score to a minimum*. This heuristic with a look of two move ahead. It tries to minimize the stones the opponent can score on their next move.

By using a genetic algorithm, a formula of strong strategy was found with combination of different heuristics. The formula of the combination of different heuristics is computed as:

V is the value of a move, H1 to H6 are heuristics, W1 to W6 are the weights. The weight are range from 0 to 1. The higher the heuristics weight, the higher the potential contribution that the heuristic can make in evaluating a position in the game. H6 and its weight is subtract in the function because it is aim to estimate the most stones an opponent can score after a player has move their stones.

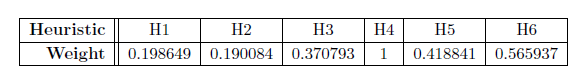


Table 1.0 Weight of the evolved player

|  |
| --- |
| Algorithm Advanced heuristic function for minimax   1. **function** EVALUATE (*board, player, opponen*t) 2. H1 🡨 Stones\_In\_Left\_Most\_Holes (*board, player*) 3. H2 🡨 Stones\_In\_All\_Holes (*board, player*) 4. H3 🡨 Nummber\_of\_Non\_Empty\_Holes (*board, player*) 5. H4 🡨 Store *(board, player*) 6. **if** Previous\_Move (*board, player*) was the rightmost **then** 7. H5 🡨 1 8. **else** 9. H5🡨 0 10. **end if** 11. H6 🡨 Store (*board, opponent*) 13. **return** *H1\*W1 + H2\*W2 + H3\*W3 + H4\*W4 + H5\*W5 – H6\*W6* 14. **end function** |

These advanced heuristic function has improved the pruning of the tree in order to select the best move. Moreover, the heuristic algorithm also uses beam search that explores a graph by expanding the most promising node in a limited set and randomly pick one of them. For example, the root node has a set of possible moves , their values are , and the best value is . The return value is pick randomly between the moves that have a value if the value is positive, or if the value is negative. By using this way, a move with a slightly lower value than the best value, will has the same probability of being played of the move with the best value. Thus, it can eliminate the effect that small differences of values between two or more move may have.

**Monte Carlo Tree Search**

Monte Carlo Tree Search also a method that can solve mancala game (Rovaris, 2017). It does not need a heuristic function. The root node act as the current state of mancala game, while rest of each node act as state of available move of mancala game, terminal state which a player won or tie state. MCTS will play simulation game by choosing all the possible move from the root until the game is terminated.

According to the above algorithm for MCTS, after creating a root node with the current state of board, we will set a computational budget which is the number of nodes visited by the algorithm. In the exploration step, the TREEPOLICY tree policy will find the most urgent node of current tree. While in the expansion step, it will use the DEFAULTPOLICY randomly pick the available move. After the simulation game, then it will backpropagated to the root node, each vector containing the score of each player. The reward term for a victory is 1 while defeat is 0. Lastly, the best move will be chosen.

|  |
| --- |
| Algorithm Monte Carlo Tree Search   1. **function** MCTS (*)* 2. create root node with state 3. **while** within computational budget **do** 4. 🡨 TREEPOLICY () 5. ∆ 🡨 DEFAULTOLICY (s ()) 6. BACKPROPAGATE ( , ∆) 7. **end while** 8. **return** a (BESTCHILD ()) 9. **end function** |

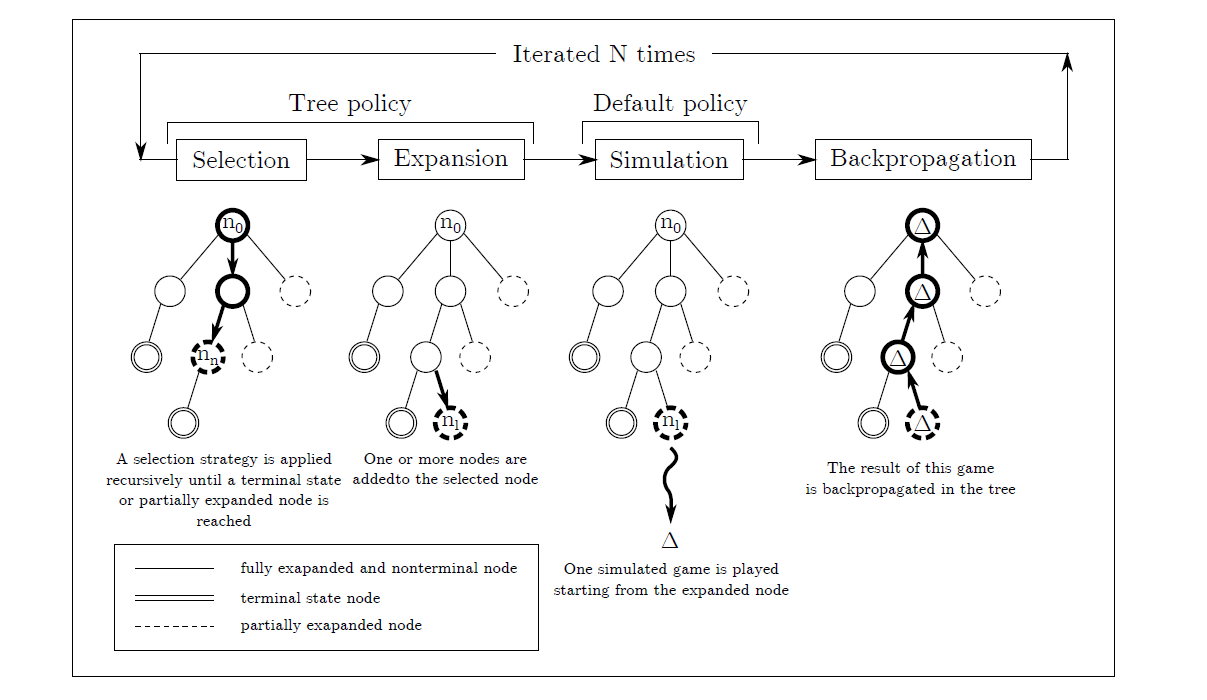


Table 2.0 Steps for MCTS algorithm (Rovaris, 2017)

**Large End-Game Database**

According to (Donkers, Jos & de Voogt, 2003), building large end-game databases which contain for a huge number of board positions how many counters can be captured and the best move to play is one of the strategies in solving Awari, which is a difficult variant of Mancala. Even though it is impossible to create a complete database with all possible positions, as stated in the article, the expectation is that soon the game-tree-search from the starting position and the end-game database will meet in the middle.

**Game-Tree Search**

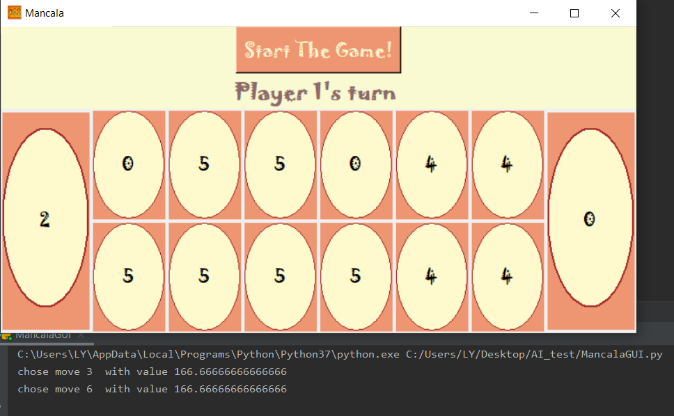
Meanwhile, the easy variant of Mancala known as Kalah is solved by implementing the game-tree search and for the instances of the game, only a winning strategy from the opening position is known explicitly. During the game, every possible positon is considered to solve the small instance of Kalah play and the game-theoretic value of the larger instance of the game can be stored in databases while implementing the program to find the optimal move (Donkers, Jos & de Voogt,2003).

## **Section III Infrastructure and methodology**

**Part 1 Method towards Mancala**

**First player to win strategy**

The method towards Mancala implemented is to apply the first player to win strategy. The first move in Mancala can be very strong and hence the AI player is set as the first player to start the game. In the first move, starting with the third hole is considered to be the best opening move (Brown, 2019). This is because the last stone will be landed in the player’s house and subsequently the player can play again. This move benefits the player to get the score at the first move and enables the second play which offers the opportunity to beat the opponent’s first strategic move. The second move after the first move is generally choosing the hole which closest or second closest to the player’s house (Brown, 2019). This move will drop a stone in the opponent’s third hole and prevent the opponent to perform the best opening move as taking the third hole. The hole closest to the house is the preferable move because the empty hole may benefit the play afterwards. Because the rightmost hole is directly next to the players’ own home, whenever a single stone is picked up from that hole as the move, the player who is moving will immediately score a point and get another move. For this reason, emptying that hole early is a powerful strategy. After the hole has been emptied, whenever a stone lands there, the immediate next move of the moving player should be to drop that stone into his or her own home for a free point and another moving change.

In the evaluation of AI player towards the move at each hole, the highest evaluation will be chosen as the next move. In the evaluation function, as the player can play again since the stone landed in the house, the value assigned to that particular move is relatively higher than the other move. After evaluating all the moves at each hole, the third hole will be chosen as it will land in the house. The second move to be chosen is based on the evaluation and the Alpha-Beta pruning algorithm which takes the opponent’s moves into consideration. After performing the alpha-beta pruning with depth of 9, the move is chosen based on the greater evaluation returned by the function.

*Figure 2-The decision of the AI player to choose the first move at the third hole and the sixth hole for the second move.*

**Alpha -beta Pruning**

Current Board

Player 2 moves

(depth =1)

For each node, branch out 6 child nodes

Player 1 moves

(or Player 2 moves again)

(depth = 2)

For each node, branch out 6 child nodes

Player 2 moves

(or Player 1 moves again)

(depth =3)

*Figure 3-The alpha-beta pruning tree*

……………

Until depth = 9

Note:

Value of Alpha and Beta = evaluation score at each hole

Pruning happen when Alpha >= Beta

The Alpha-Beta Pruning with Minimax is implemented to decide on the move to be chosen by the AI player. Alpha-Beta Pruning is a game theory and search technique to prune the search tree based on the *min* and *max* values of the states. This technique prevents suboptimal moves from being searched, which increases the efficiency of the program to find the optimal moves. Minimax algorithm with Alpha-Beta Pruning implementation are almost tantamount to the standard Minimax Algorithm. There are only two differences between them: the parameters of Minimax algorithm with Alpha-Beta Pruning as the upper and lower value bound and their function as boundary.

As the search begins for a movement in Mancala, the Alpha-Beta Pruning tree takes Player 2 (AI player) as the Maximize player and Player 1 (Human player) as the Minimize player. During the process of searching, the tree is not restricted to allow the alternate player to take the move. If the current player can make a move again (get into own house), the current player keeps on the searching in the next depth. Therefore, for a particular node which represent the movement at a particular hole, the next move is probably the current player or the opponent player. The nodes of the trees are the movement (only if the movement is legal) to each hole which needed to be evaluated. Combining with the heuristics which evaluate the score for each hole, the value of alpha and beta are updated with the evaluation score for the resulting board caused by a particular movement at each hole. The value of Alpha is expected to grow and the value of Beta is expected to decrease. With the recursive evaluation until the certain depth of 9, the best movement and its associated final evaluation score can be obtained and further be used in the pruning of the trees to speed up the searching. Using the evaluation score calculated, the AI player will choose the highest evaluation score which will lead to its winning of the game.

**Heuristics**

In the implementation of Mancala programme, the heuristics are applied in function *score (board)* for the advanced searching for the best move. The evaluation score obtained is used in updating the values of alpha and beta in the Alpha-Beta Pruning algorithm. The heuristics are the following:

* H1: *Has AI won the game.* This heuristic evaluates has the AI won the game. If the AI has won, the game it will return 500 score.
* H2: *Has opponent won the game.* This heuristic evaluate has the opponent won the game. If the opponent has won the game, it will return -500 score.
* H3: *How close the AI to win.* The heuristic evaluates whether the stones in the AI house has more than half, 24. If it is more than half, it will return 250 score.
* H4: *How close the opponent to win.* The heuristic evaluates whether the stones in the opponent house has more than half, 24. If it is more than half, it will return -250 score.
* H5: *Choose a move that can lead AI has one more turn in the game.* When the last stone goes into the player own house, it has additional turn. This heuristic evaluates the value of *additional* by using the (50.0 \* (i + 1))/6 which gives 50 points to the current hole as average of 6 holes.
* H6: *How many stones can opponent take from AI rows of holes.* If the last stone of opponent lands on an empty hole, it can take away all the stones from the opposite holes which belong to AI. This heuristic evaluates the value of *capturing* as 50 scores if the opponent takes no stone, 40 scores for 1 stone and 25 scores for 2 stones. The higher the scores, when the opponent take less stones from AI player.
* H7: *Number of stones of the chosen move are less.* If the number of stones is less than 4, the heuristic will add the value of *stones* with low scores 5.
* H8: *Keep the last stones get into its own house.* When, the last stones get into AI own house, increase the value of *stones* with 30 scores.
* H9: *Stones land at other holes.* When, the stones land at other holes, add the value of *stones* with lower scores 10.

Eventually, all these heuristics combine together to compute the total evaluation score of the AI player’s move in the Alpha-Beta Pruning algorithm.

**Part 2 System design and implementation**

**Mancala Board**

|  |
| --- |
| **class MancalaBoard:** |

The *MancalaBoard* class includes the functions which deal with the operations on the Mancala board.

|  |
| --- |
| **Initialize algorithm**  Function: \_\_init\_\_()  reset() |

The initialization function for the *MancalaBoard* class which initialized the *reset()* function as the attribute of the class.

|  |
| --- |
| **Reset algorithm**  Function: reset()  *NHOLES*🡨6  *scoreHoles*🡨 [0,0]  *P1Holes*🡨 [4]\* *NHOLES*  *P2Holes*🡨 [4]\* *NHOLES* |

The *reset()* function resets the mancala board for a new game. The number of holes for each row is 6. The *scoreHoles* which is the house of the two players is initialized to 0 for both. The row of holes for each player is set as 6 holes and there are 4 stones in each hole ([4], [4], [4], [4], [4], [4]).

|  |
| --- |
| **Legal move algorithm**  Function: legalMove (*player,hole*)  //Input: the player’s turn (Player 1 or Player 2) and the next hole to be moved  //Output: return true or false for a move  **if** (*player*.*num*==1)  *holes*🡨*P1Holes*  **else**  *holes*🡨*P2Holes*  **End if**  **return** *hole* > 0 && *hole* <= *len*(*holes*) && *holes*[*hole*-1] > 0 |

The *legalMove()* function returns whether a move is legal or not. The function gets the row of holes for the current player and check the legal move. A legal move should follow that the hole is in the range of the number of holes for one row. The current hole should also not to be empty to allow move to the next hole.

|  |
| --- |
| **Legal Moves algorithm**  Function: legalMoves (*player*)  //Input: the player’s turn (Player 1 or Player 2)  //Output: a list of legal moves  **if** (*player.num*==1)  *holes*🡨*P1Holes*  **else**  *holes*🡨*P2Holes*  **end if**  *moves*= [ ]  **for** *m* in range of length of holes  **if** (*holes*[*m*]!=0)  *moves*🡨*moves* + [*m*+1]  **end for**  **return** *moves* |

The *legalMoves()* function returns a list of legal moves for the player. The function gets the row of holes for the current player and initializes a list for moves. For each hole which in the row, the index of the hole is stored into moves and the function eventually returns a list of legal moves, *moves*[ ] .

|  |
| --- |
| **Make move algorithm**  Function: makeMove (*player, hole*)  //Input: the player’s turn (Player 1 or Player 2) and the next hole to be moved  //Output: return whether the player can make a move again or not  *again* = makeMoveHelp(*player, hole*)  **if** (gameOver())  **for** *i* in range of length of P1Holes **do**  s*coreHoles*[0] 🡨 *scoreHoles*[0] + *P1Holes*[*i*]  *P1Holes*[*i*] 🡨 0  **end for**  **for** *i* in range of length of *P2Holes* **do**  *scoreHoles*[1] 🡨 *scoreHoles*[1] *+ P2Holes*[*i*]  *P2Holes*[*i*] 🡨 0  **return** False  **else**  **return** again  **end for** |

The function *makeMove()* clears out stones in the holes for both the players if the game is over. The stones in every hole belonging to the player’s row are put into the player’s house. If the game can continue to play, the function returns variable again to call the *makeMoveHelp()* function to make a move for the player.

|  |
| --- |
| **Make move help algorithm**  Function: makeMoveHelp(*player, hole*)  //Input: the player’s turn (Player 1 or Player 2) and the next hole to be moved  //Output: return whether the player gets another turn or not  **if** (*player.num* == 1)  *holes* 🡨 *P1Holes*  *oppHoles* 🡨 *P2Holes*  **else**:  *holes* 🡨 *P2Holes*  oppHoles 🡨 *P1Holes*  **end if**  *initHoles* 🡨 *holes*  *nstones* 🡨 *holes*[*hole* - 1]  *holes*[*hole* - 1] 🡨 0  *hole* 🡨 *hole* + 1  *playAgain* 🡨 False  **while** (*nstones* > 0) **do**  *playAgain* 🡨 False  **while** (*hole* <= *len*(*holes*) && *nstones* > 0) **do**  *holes*[*hole* - 1] 🡨 *holes*[hole - 1] + 1  nstones 🡨 nstones - 1  *hole* 🡨 *hole* + 1  **end while**  **if** (*nstones* == 0)  **break**  **if** (*holes* == *initHoles*)  *scoreHoles*[*player.num* - 1] 🡨 *scoreHoles*[*player.num* - 1] + 1  *nstones* 🡨 *nstones* - 1  *playAgain* 🡨 True  **end if**  *tempHoles* 🡨 *holes*  *holes* 🡨 *oppHoles*  *oppHoles* 🡨 tempHoles  *hole* 🡨 1  **end while**  **if** (*playAgain*)  **return** True  **if** (holes == initHoles && *holes*[*hole* - 2] == 1) *scoreHoles*[*player.num*-1]🡨 *scoreHoles*[*player.num* - 1] + *oppHoles*[(*NHOLES*-  *hole*) + 1]  oppHoles[(*NHOLES* - *hole*) + 1] 🡨 0  *scoreHoles*[*player.num* - 1] 🡨 *scoreHoles*[*player.num* - 1] + 1  *holes*[*hole* - 2] 🡨 0  **end if**  **return** False |

The function *makeMoveHelp()* make a move for the player and return true if the player gets another turn and false if not by assuming a legal move. The function first gets the row of holes of the current player, holes and the row of holes of the opponent, *oppHoles*. The function then gets the holes as *initHoles* and *nstones* as the number of stones in the current hole, *hole*[*hole* – 1] since the hole in the argument refers to the next hole. The current hole is now empty and the hole argument is now referring to the next two hole from the current hole. The variable *playAgain* is initialized as false. The function starts to loop to make move for the player while the player still has stones left to move. One stone is dropped into the hole and moving to drop the stones to the next hole until there are no stones left and break from the function. The side can be switched to opponent row for the function to make the move at the opponent holes and it continues the move until the stones are all dropped. If the player is at own row after some moves and reaches its own house, the stone is dropped into the house and *playAgain* is set to true. After the player complete one play, if *playAgain* is true, the function returns true which indicates that the player gets another turn to play. Meanwhile, after the moves, the function also checks that if the player ended the move in a blank space at own side of row. When the player drops the last stone in an empty hole at own row, the player can take the stones in the current hole and the stones in the opposite hole into the player’s house. This makes the current hole to be empty. After this check, the function returns false which indicates the player can only perform this move since *playAgain* is false.

|  |
| --- |
| **Has won algorithm**  function: hasWon(*playerNum)*  //Input: the player number, player 1 is human player and player 2 is AI player  //Output: return whether or not the player has won  **if** (gameOver())  *opp* 🡨 2 - *playerNum* + 1  **return** *scoreHoles*[*playerNum* - 1] > *scoreHoles*[*opp* - 1]  **else**  **return** False  **end if** |

The function *hasWon()* performs check determine whether a given player has won the game. The function calls the function *gameOver()* which returns true or false to indicates the game is over or not. If the game is over, the function checks the number of stones in the houses of both players. The function returns true if the number of stones in the current player’s house is more than the opponent to win the game, else the function returns false which indicates the current player does not win.

|  |
| --- |
| **Get player holes algorithm**  function: getPlayersHoles(*playerNum*)  //Input: the player number, player 1 is human player and player 2 is AI player  //Output: return the row of holes for given player  **if** (*playerNum* == 1)  **return** *P1Holes*  **else**  **return** P2Holes  **end if** |

The function *getsPlayersHoles()* returns the row of holes for the given player using if/else statement. The function returns the row of holes for player 1 and player 2 by checking the current player number.

|  |
| --- |
| **Game Over algorithm**  Function gameOver ( )  *over* 🡨 True  **for** *elem* in the range of *P1Holes* **do**  if *elem* not equal to 0  *over* 🡨 False  **end for**  **if** *over* is True  return True  **for** *elem* in range of *P2Holes* **do**  **if** *elem* not equal to 0  *over* 🡨 False  **end for**  **return** True |

This *gameOver( )* function is to check the mancala game end or not. If the stones, *elem* in the *P1Holes* and *P2Holes* not equal to 0, it means the game haven’t over yet so it will return False. On the other hand, if the stones, *elem in P1Holes* and *P2Holes* equal to 0, it means the game is over, it will return True.

|  |
| --- |
| **class Player:**  *HUMAN* =1  *ABPRUNE* =2 |

**AI**

**Player**

This is the class for the player such as the human is value 1 and alpha beta pruning method is value 2.

|  |
| --- |
| **Initialize algorithm**  function \_init\_ (*playerNum, playerType, ply=0*)  *num* 🡨 *playerNum*  *opp* 🡨 2- *playerNum*+1  *type* 🡨 *playerType*  *ply* 🡨 *ply* |

This function initialize function. The *num* is the *playerNum* 1 or 2. Then the opponent, *opp* can be computed by 2-*playerNum* (1 or 2) +1. Besides that, the type equals to the *playerType* in the class player such as *HUMAN* and *ABPRUNE*. Last, the *ply* is the depth of the Alpha-Beta Pruning tree, it is set default to 0.

|  |
| --- |
| function \_repr\_ ( )  **return** str (*num*) |

This function returns a string representation of player number.

Minimax algorithm with Alpha-Beta pruning

1. alphaBetaMove (board, depth (ply))
2. alphaMaxValue (board, depth (ply), turn, alpha, beta)
3. alphaMinValue (board, depth (ply), turn, alpha, beta)

|  |
| --- |
| **Alpha Beta Move algorithm**  Function alphaBetaMove(*board, depth (ply))*  Let the *alpha* = negative infinity  Let the *beta* = positive infinity  **Return** maximum value of *alpha* by calling the function alphaMaxValue |

Above retrieve about the function to choose a move with alpha beta pruning.

|  |
| --- |
| **Alpha max value algorithm**  function alphaMaxValue (*board, ply, turn, alpha, bet*a)  #input: *board,* depth of search, *ply*, *alpha* and *beta* value  #output: score of the player, *score* and which move to choose, *move*  **if** *board.gameOver ( )* or *ply* == 0 or (*board.scoreHoles*[0] – *board. scoreHoles*[1]) >  24  **return** *turn.score (board)* , -1  *score* 🡨-∞  *move* 🡨-1  **for** *m* in *board.legalMoves ( )* **do**  *opponent* 🡨*Player (opp, type, ply)*  *nextBoard* 🡨 *deepcopy (board)*  *again*= *nextBoard.makeMove (m)*  **end for**    **if** *again* == True **then**  *s, m* 🡨 alphaMaxValue (*nextBoard, ply-1, turn, alpha, beta*)  **else**  *s, m* 🡨 opponent.alphaMinValue (*nextBoard, ply-1, turn, alpha, beta*)  **end if**    **if** *s> score* **then**  *score* 🡨 *s*  *move* 🡨*m*  **if** *score* >= *beta* **then**  **return** *score, move*  **if** *score*> *alpha* **then**  *alpha*= *score*  **return** *score, move* |

This *alphaMaxValue* is a function to compute the maximum alpha value for the player to choose its next move. It takes the board configuration, *board*, depth of search, *ply*, which player is playing, turn and the *alpha* and *beta* values as input. First, it checks whether the game is in game over state or the depth of search is 0 or the difference of the score between the player is greater than 24 which means one of the players get the most half score in their house, it almost wins. If the one of these conditions is true, it will directly return the score of the player now and stop its move. if none of these conditions are true, it assign the score to negative infinity like the alpha value and assign the move to -1.

In the for loop in a range of list of legal moves, make a new player, opponent and copy the board, *nextBoard* in order not to ruin the previous board. Then, make sure the board still have legal move. If the board still have legal move, the s and m is equal to the recursive function of *alphaMaxValue* with the next depth, ply-1. If there are no legal move, now is opponent turn and calculate the s and m with *alphaMinValue* with next depth, *ply*-1.

After that, if the updates score, *s* is greater than the score, assign the score to *s*, move to *m*. Moreover, if the score is greater than the *beta* value, so we can prune it and then return directly return the *score* and *move*. Otherwise, if the *score* greater than the *alpha*, then the *alpha* replaces by the new *score* value. Lastly, the *alpaMaxvalue* function will return the largest score and best move.

|  |
| --- |
| **Alpha min value algorithm**  Function alphaMinValue (*board, depth (ply), turn, alpha, beta*)  If *board*.gameOver() or *ply* == 0 or abs(*board.scoreHoles*[0] - *board.scoreHoles*[1]) > 24:  **return** the score of the *score* and *move*  Let the *score* = infinity  Let the *move* = -1  **for** *m* to the legalMoves() **do**  Let *opponent* = Player (*opp, type, ply*)  *nextBoard* = deepcopy(*board*)  *again* = nextBoard.makeMove(*m*)  **If** *again*  *s, mv* = alphaMinValue(nextBoard, *ply* - 1, *turn*, *alpha, beta*)  **Else**  *s, mv* = *opponent*.alphaMaxValue(nextBoard, *ply*-1*, turn*, *alpha, beta*)  **end if**  **if** *s* < *score*  *score* = *s* value  *move* = the *m* value  **if** *score* <= *alpha*  **return** *score* and *move*  **if** *score* < *beta*  *beta* = the *score* value  **end for**  **return** *score* and *move* |

In the function: *alphaMinValue* with parameter board, depth (*ply*), *turn*, *alpha* and *beta*. If the board is game over or reach the root of tree or the absolute value of subtraction of both players homes’ stone is more than 24, return the score of the score and move. Then, let the score assigns with infinity value. Let the move assigns with value of -1. For loop is applied from 0 to the legal move of board. In the for loop, let opponent be the player1. Make a complete copy of the board to not ruin the previous one. Then, make sure that there is still having available movements. If still have available moves, so update s and mv be the current player’s *alphaMinValue*. Else then update s and mv be the opponent’s *alphaMaxValue.* If updated s is smaller than the score, score is assigned with the s value and move is assigned with the m value. If the updated score is smaller or equal to the alpha value, return score and move. One more condition, if the updated score is smaller than beta value beta is assigned with the score value. Eventually, return score and move.

|  |
| --- |
| **Choose move algorithm**  Function chooseMove(*board*)  //Input: Mancala board  //Output: returns the next move that the AI player makes  **if** (*type* == *ABPRUNE*)  *val, move* 🡨 a*lphaBetaMove*(*board, ply*)  print the chosen move and the value  **return** move |

The function *chooseMove()* returns the next move that the AI player makes. The function *alphaBetaMove()* is called to get the move and evaluation value that obtained by performing alpha-beta pruning and heuristic evaluation. The function prints the chosen move and value on the terminal and return the move of the AI player.

|  |
| --- |
| **Score algorithm**  function: score(*board*)  //Input: Mancala board  //Output: evaluation of the Mancala Board for AI player  **if** (*board*.hasWon(*num*))  **return** 500.0  **else if** (*board*.hasWon(*opp*))  **return** -500.0  **else if** (*board.scoreHoles*[*num* - 1] > 24)  **return** 250  **else if** (*board.scoreHoles*[*opp* - 1] > 24)  **return** -250  **else**  **if** (*self.num* == 1):  *holes* 🡨 *board*.*P1Holes*  *oppHoles* 🡨 *board.P2Holes*  *scoreDiff* 🡨 *board.scoreHoles*[0] *- board.scoreHoles*[1]  **else**  *holes* 🡨 *board.P2Holes*  *oppHoles* 🡨 *board.P1Holes*  *scoreDiff* 🡨 *board.scoreHoles*[1] - *board.scoreHoles*[0]  **score** 🡨 (*scoreDiff* + 30) \* 10 / 6    *additional* 🡨 0.0  *capturing* 🡨 0.0  *stones* 🡨 0.0  **for** *i* in range of length of holes **do**  *numStones* 🡨 *holes*[*i*]  **if** (*numStones* > 0 || *oppHoles*[*i*] > 0)  **if** (n*umStones* == (6 - i))  additional🡨 (50.0 \* (i + 1))/6  **end if**  *temp* 🡨 o*ppHoles*[*i*] % 13  *temp\_capturing* 🡨 0.0  *ownStones* 🡨 -1  **if** (*temp* <= (5 - i))  **if** (t*emp* == 0)  *ownStones* 🡨 *holes*[5 - *i*]  **else**  *ownStones* 🡨 *holes*[5 - *i* - *temp*]  **end if**    **else if** (*temp* >= (13 - *i*))  *ownStones* 🡨 *holes*[5 - *i* + *temp* - 13]  **end if**  **if** (*ownStones* == 0)  *temp\_capturing* 🡨 50.0  **else if** (*ownStones* == 1)  *temp\_capturing* 🡨 40.0  **else if** (*ownStones* == 2)  temp\_capturing 🡨 25  **end if**    **if** (temp\_capturing > capturing)  capturing 🡨 temp\_capturing  **if** (*numStones* < 4)  *stones* 🡨 *stones* + 5  **else**  **if** (*i* == 5)  *stones* 🡨 *stones* + 30  **else**  *stones* 🡨 *stones* + 10  *total* = *capturing + score + additional + stones*  **return** *total* |

The function *score()* evaluates the Mancala board for AI. The function performs several evaluations based on self-determined values to calculate the evaluation value for the move.

The function assigns the value ±500 whether if the AI player can win the game and the value ±250 if the number of stones in the house is greater than 24 (half of the total number of the stones) in which the player may win the game. If neither of these conditions is true, the function continues to evaluate the move.

Then, the function gets the row of holes of the player, the row of holes of the opponent and the difference of score between the houses of the players. The function evaluates the total of evaluation score using four values, which are *score, additional, capturing and stones*. A self-determined equation, (*scoreDiff* + 30) \* 10 / 6 is used to check for a better score for the move and to evaluate the score.

Subsequently, the function then evaluates the value for additional, capturing and stones in every hole, i in the row of holes. The value of additional is evaluated if the number of stones in a particular hole is sufficient to get the last stone to be dropped in the house. As the last stone goes into the house, the player will have an additional turn and hence the additional value is evaluated as *(50.0 \* (i + 1))/6* which gives 50 points to the current hole as average of 6 holes.

The function then evaluates the capturing value from the opponent’s perspective in the hope that the opponent captures lesser stone from the AI player’s row of holes. Variable *temp = oppHoles[i] % 13* is used to know which hole the last stone will land. There are total of 12 holes and using %13 can know whether the stone lands at the origin hole (13%13=0). The number of stones that taken away by the opponent is stored as *ownStones*. If the opponent performs moves which will not get into the opponent’s house and land back at the same hole, which is the empty hole, the opponent can take away the stones in the opposite hole, which is the AI player’s stones. Meanwhile, if the opponent lands at own row of holes or else lands at the hole beside the current hole, the opponent can also take away the stone opposite from player AI’s row if the last stone is dropped at an empty hole. After these checks, the value for capturing is evaluated as 50 points if the opponent takes no stone, 40 points for 1 stone and 25 points for 2 stones. The higher points are given if the opponent takes lesser stone in which the score for that move towards winning is higher. The value for *capturing* is also updated if the temporary capturing value is higher.

Since the number of stones in a particular hole is also evaluated, the number of stones which smaller than 4 can increase the value of *stones* for 5 as moves are made. The move that get the last stone into the player’s house is evaluated with an increase of 30 points for the stones value and an increase of 10 points if the stone lands at the other hole.

Eventually, the function returns the total of evaluation as the sum of the value of *score, additional, capturing and stones*.

## **Section IV Error Analysis and Experiments**

There are some flaws in the program that the calculation for the first step of the AI player takes long time to begin the search compared to the time for the remaining steps to be taken. The Alpha-Beta Pruning is carried out with the depth of 9 and the AI player will take a second turn after the first move. The calculation process begins again to perform the next move and this has doubled the computation time for the first move.

A simple experiment had been carried out to test the accuracy of the AI towards Mancala Board game. Two categories of players which are three advance players included the group members and two beginner players, were invited to engage in this experiment.

**Advance Player (Group members)**

|  |  |  |  |
| --- | --- | --- | --- |
| **No.** | **AI score** | **Human score** | **Winner** |
| 1 | 41 | 7 | AI |
| 2 | 25 | 23 | AI |
| 3 | 31 | 17 | AI |
| 4 | 38 | 10 | AI |
| 5 | 25 | 23 | AI |
| 6 | 27 | 21 | AI |
| 7 | 41 | 7 | AI |
| 8 | 36 | 12 | AI |
| 9 | 29 | 19 | AI |
| 10 | 41 | 7 | AI |
| 11 | 35 | 13 | AI |
| 12 | 41 | 7 | AI |
| 13 | 41 | 7 | AI |
| 14 | 27 | 21 | AI |
| 15 | 28 | 20 | AI |
| 16 | 28 | 20 | AI |
| 17 | 41 | 7 | AI |
| 18 | 42 | 6 | AI |
| 19 | 29 | 19 | AI |
| 20 | 41 | 7 | AI |
| 21 | 38 | 10 | AI |
| 22 | 37 | 11 | AI |
| 23 | 42 | 6 | AI |
| 24 | 35 | 13 | AI |
| 25 | 25 | 23 | AI |
| 26 | 36 | 12 | AI |
| 27 | 27 | 21 | AI |
| 28 | 32 | 16 | AI |
| 29 | 26 | 22 | AI |
| 30 | 41 | 7 | AI |

**Beginner Player (Ms. Tay & Ms. Chien)**

|  |  |  |  |
| --- | --- | --- | --- |
| **No.** | **AI score** | **Human score** | **Winner** |
| 1 | 41 | 7 | AI |
| 2 | 38 | 10 | AI |
| 3 | 29 | 19 | AI |
| 4 | 40 | 8 | AI |
| 5 | 34 | 14 | AI |
| 6 | 35 | 13 | AI |
| 7 | 36 | 12 | AI |
| 8 | 35 | 13 | AI |
| 9 | 41 | 7 | AI |
| 10 | 30 | 18 | AI |
| 11 | 40 | 8 | AI |
| 12 | 31 | 17 | AI |
| 13 | 41 | 7 | AI |
| 14 | 42 | 6 | AI |
| 15 | 33 | 15 | AI |
| 16 | 36 | 12 | AI |
| 17 | 41 | 7 | AI |
| 18 | 40 | 8 | AI |
| 19 | 31 | 17 | AI |
| 20 | 31 | 17 | AI |

Through this experiment, both advance and beginner players lost the game to AI player. The advance players tend to get higher score however still beaten by the AI player. The beginner players lost by getting relative lower score. Overall, 50 samples are obtained to calculate the accuracy.

Accuracy = x 100%

= 100%

There is no data obtained that the AI player has lost the game, this may due to the quantity of testing data is insufficient.

## **Section V: Conclusion**

The artificial intelligence algorithms for Mancala games are designed and analysed in this report. The Alpha-Beta Pruning Minimax algorithm uses Minimax with the alpha-beta pruning technique to reduce memory consumption and computational time. Meanwhile, the mentioned algorithm is implemented with the heuristics which evaluate each movement of the AI player with evaluation score in order to choose the best strategy for the current play and lead the AI to the winning of the game. Therefore, Alpha-Beta Pruning Minimax is applied instead of Minimax. Moreover, the programming language applied is Python.

Through a series of experiments with 50 samples, the AI player won with accuracy of ­100%. It is reasonable and valid as the AI player will chose the best strategy for its movement after the calculation is done.

## **References**

Brown, S. (2019, November 11). Basic Strategy for Mancala. Retrieved from The Spruce Crafts: <https://www.thesprucecrafts.com/how-to-win-at-mancala-basic-strategy-411832>

Divilly, C. (2013). Exploration and analysis of the evolution of strategies for Mancala variants. 2013 IEEE Conference On Computational Inteligence In Games (CIG). doi: 10.1109/CIG.2013.6633628

Donkers, Jeroen & Uiterwijk, Jos & de Voogt, Alex. (2003). Mancala games: Topics in Mathemathics and Artificial Intelligence. Journal of Board Game Studies. 6. 135-148.

Gehlot, N. (2016). neelamgehlot/Mancala-Game-Playing-Agent. [online] GitHub. Available at: https://github.com/neelamgehlot/Mancala-Game-Playing-Agent [Accessed 10 Dec. 2019].

LangenBorg, E. (2017, May 15). The Ancient Game and the AI. Retrieved from Towards Data Science: <https://towardsdatascience.com/the-ancient-game-and-the-ai-d7704bea280d>

Mancala World. (2019). Pie rule. Retrieved from Fandom: <https://mancala.fandom.com/wiki/Pie_rule>

Rovaris, G. (2017). Design of Artificial Intelligence for Mancala Games (Unpublished dissertation). Politecnico Di Milano, Milan, Italy. Retrieved from <http://hdl.handle.net/10589/134455>

Sumali, B., & Michael Siregar, I. (2016). Implementation of Minimax with Alpha-Beta Pruning

as Computer Player in Congklak. Retrieved 13 December 2019, from https://media.neliti.com/media/publications/133348-EN-implementation-of-minimax-with-alpha-bet.pdf

**APPENDIX 1**

**Marking Rubrics**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Component Title** | **Project** | | | | | **Weight** | **50** | |
| **Criteria** | **Score and Descriptors** | | | | | | **Weight** | **Marks** |
| **Excellent (directly relevant)**  **80-100** | **Good**  **(Somewhat relevant)**  **70-80** | **Average**  **(Remotely related)**  **50-70** | **Need Improvement**  **(Not related)**  **40-50** | **Poor (not related / not present)**  **0-40** | |
| Presentation & Report | Information is presented in effective order. Excellent structure of paragraphs and transitions enhances readability and comprehension. | Information is logically ordered with paragraphs and transitions. | Include vague information. | Details and examples are not organized, and hard to follow and understand. | Details are not related and badly written and hard to follow. | | 20 |  |
| **Criteria** | **Excellent (directly relevant)**  **80-100** | **Good**  **(Somewhat relevant)**  **70-80** | **Average**  **(Remotely related) 50-70** | **Need Improvement**  **(Not related)**  **40-50** | **Poor (not related / not present)**  **0-40** | |  |  |
| Demonstration | Functions are fully implemented relative to extra demands and extra analysis | Functions are fully implemented without any extra demands and extra analysis. | Functions are implemented in the manner of conventional methods. | Functions are not fully implemented but the core part is finished. | Nearly no implementation about any functionalities. | | 20 |  |
| **Criteria** | **Excellent (directly relevant)**  **80-100** | **Good**  **(Somewhat relevant)**  **70-80** | **Average**  **(Remotely related) 50-70** | **Need Improvement**  **(Not related)**  **40-50** | **Poor (not related / not present)**  **0-40** | |  |  |
| Manner | Good manner with proper voice speed and proper voice volume. | Good manner with proper voice volume. | Good manner | Normal Manner | Poor Manner | | 10 |  |
| **TOTAL** | | | | | | | **50** |  |

Note to students: Please print out and attach this appendix together with the submission of coursework