COMP 6721 Project 1 Report:

Heuristic Minimax search with Alpha-Beta pruning

for the Double Card game

Jun Shao[1] and Zhiyi Ding[2]

1 ID-40077592 shao12jun@gmail.com

2 ID-26963013 dingzhiyi22@gmail.com

1 Introduction

Game playing is an import domain of heuristic search, and its procedure can be represented by a special tree. Minimax search with α-β pruning has been the predominant algorithm for game-tree search for three decades [1, 2].

The purpose of this work is to establish an AI with informed Minimax search and alpha-beta pruning algorithm to play the Double Card game. Besides, we will analyze the effectiveness of alpha-beta pruning, compare the strengths and weakness of different heuristic functions.

1.1 Game Description

Double Card is a 2-player adversarial game with maximum 24 identical cards on an 8×12 board. Each card is made of two segments with different combination of color (red and white) and shape (circle and dot) and has two faces. Each card can be placed on the board on either side and can be rotated by 0°,90°,180 or 270°, for a total of 8 possible placements.

At first, players should decide what he/she will play: color or dot. If all 24 cards have been played and neither player won the game, then players proceed to use recycling moves rather than the regular moves. The player who first gets four consecutive segments (in a row, column or a diagonal) in their desired type (color or dot) on the board wins. If after 40 moves, no player has won, then the game ends in a draw [3].

1.2 Technical Details

Game_Frame() is a main method used to control the game procedure. The first part of the method is to achieve inputs from user to set up the gameplay such as playing mode (human vs human or human vs AI), red playing condition (win as color or shape) and machine play first or human playing first.

The second part is the main game loop which produces an optimal next move based on the different heuristic functions if it's in the AI mode.

At last, if the steps exceed 40, the program will enter the recycling move mode which retrieves a card first on the board before placing the new card.

Terminal_checker() will check if the game is terminated for each game stage. It is not necessary to search the whole board brutally to find if there are four-consecutive-segments existing with the same color (red or white) or shape (dot or circle) on the board. Actually, we only need to check if the latest placement generated four-consecutive -segments in four directions (horizontal, vertical, left diagonal and right diagonal), because the stage before last move was not a terminal, possible terminal can only be brought in by the lasted card on the board.

Rule_checker() will check the legality of each move. Cards can only be placed at row 1 of the board or on top of a card already on the board. There should be no segment hang over an empty cell. A card can't overlap with an existing card on the board. For recycling moves, A player cannot move the card that the other player

just moved/placed. A player cannot place a new card at the place (two cells) where an old card was just moved but can occupy part of the place by pacing a new card with the different direction [3].

Minimax_ab() is the Minimax search algorithm with alpha-beta pruning function. We implemented different Minimax_ab functions according to some special needs of heuristics.

h_n() is different versions of heuristic functions which evaluate a stage of the game and returns a score represents how favorable the stage is for Max player. If h_n(s)>0, the node s is favorable to Max, and vice versa.

2 Algorithm

2.1 Minimax

Theorem 1. *Minimax is a decision rule to maximize or minimize the possible case scenario[4].*

In the implementation of Minimax algorithm, the goal is to find the optimal next move based on scores of each possible stage status. Some necessary inputs are required to calculate the scores: current game stage, the depth of searching desired, indicator for the player's turn. The proper output of Minimax algorithm would be the combination of the position and orientation (rotation) of the cards.

2.2 Alpha-Beta Pruning

Alpha-Beta pruning improves the efficiency of the Minimax algorithm. It gets rid of the undesired branches (nodes) of the Minimax will possibly go through. The simple mechanism of Alpha-beta pruning is to assign two variables alpha and beta to each search node alternatively. If the Alpha<=Beta, the node and its branch will be pruned.

The pseudo code of Minimax algorithm [3] and Alpha-Beta Pruning [3] are as follows:

```
Result: Optimal next move
function minimax(node, depth, maximizingPlayer) :;
if depth = 0 or node is a terminal node then
    return the heuristic value of node;
else
end
if maximizingPlayer then
    value := -Infinity;
    for each child in children do:;
    value := max(value, minimax(child, depth  1, FALSE));
    return value;
else
    value := +Infinity;
    for each child in children do:;
    value := min(value, minimax(child, depth  1, TRUE));
    return value;
end
```

Algorithm 1: Minimax search

```
Result: Optimal next move
function aplhabeta(node, depth,a,b, maximizingPlayer) :;
if depth = 0 or node is a terminal node then
    return the heuristic value of node;
else
end
if maximizingPlayer then
    value := -∞
    for each child in children do:;
    value := max(value, minimax(child, depth  1, FALSE));
    a := max(a,value);
    if b ≤ a then break;
    return value;
else
    value := +Infinity;
    for each child in children do:;
    value := min(value, minimax(child, depth  1, TRUE));
    b := min(b,value);
    if b ≤ a then break;
    return value;
end
```

Algorithm 2: Alpha-beta Pruning

2.3 Heuristic

Before designing our heuristic functions, we implemented a naive heuristic function, which aims to raise the level of cards on the board rather than get four-consecutive-segments as soon as possible. In consequence, although this naive heuristic function can see far ahead, it gives us bad guidance in the game. Therefore, we decide to design well-informed heuristics. In this project, two heuristics were designed. Although the main idea and features used are the same, they differ in realization to reduce computing time.

Feature design: When playing the Double Card game with each other, we found that each time when the opponent placed a new card and got two three-consecutive-segments which cannot be blocked at one time, then I would lose the game unavoidably in his next step. This experience gave us the idea to design a heuristic which assigns a high score (5 scores) for a stage with one desired three-consecutive-segments. The more three-consecutive-segments the state has, the greater scores will be given. By contrast, a very low negative score (-10 scores) will be given to the state if a three-consecutive-segment in mini players' favor is detected.

Two-consecutive-segment also makes sense, however, it does not contribute as much as three-consecutive-segment, so we assign a lower weight 0.2 and -0.2 to two-consecutive-segment when it is favorable or unfavorable to the Max player, respectively. Considering that one-segment alone are common, to save the computational resource, we do not count one segment as a feature.

Moreover, as we have designed an effect **Terminal_checker()** function, which can check if a state is a terminal, we do not take four-consecutive-segments into account in our heuristics.

In conclusion, we take the number of three-consecutive-segments, two-consecutive-segments as the features of our heuristics and use a 4×2 array to store the value of these features. The general equation of our heuristic function is:

$$h(s) = w_1 *(nb\_three\_red+nb\_three\_white) +w_2* (nb\_two\_red+nb\_two\_white) +w_3*(nb\_three\_dot+nb\_three\_circle)+w_4*(nb\_two\_dot+nb\_two\_circle) \quad (1)$$

In our project, when Max player plays color, then [w_1, w_2, w_3, w_4] will be assigned [5,0.2,-10,-0.2]. Otherwise, [-10,-0.2,5,0.2] will be assigned to the weight tuple.

The first heuristic (h_1): To balance the speed and performance of heuristics, we find out the surface spaces on the board, at first. Surface space is an occupied unit with the highest row in each column, which means that there are no cards on top of surface space.

After finding out surface spaces, we can find out the filled lateral spaces that are open to the open space as well. Then, we check each of the surface spaces and lateral spaces in five directions (left, right, vertical, left diagonal and right diagonal) and count how many two-consecutive-segments and three-consecutive-segments there are in different color and shape, respectively.

The normal depth of searching in a horizontal and diagonal direction is 5, including 2 spaces in left and 2 spaces in right. But vertically, we at most search for 3 spaces (surface space plus 2 spaces below). All consecutive-segments we get are "alive" as they are linked with empty space, and it is possible for them to form a four-consecutive-segment at last.

As we only search for consecutive-segments starting from surface space or lateral space, we reduce the computation for "died" consecutive-segments that are blocked or surrounded by spaces with different color or shape. In this project, the heuristic function h_1 search brutally from up to down and from left to right to achieve all surface and lateral spaces. We also implemented a program to trace the surface of occupied space by a serial of conditional clauses. Consider the difference in computational time can be ignored while using nodes pruning. So, we count them as the same heuristic.

The second heuristic(h_2): Inspired by the designing of terminal checking function, we design a heuristic that only calculates the changes brought in by the latest move, rather than search all the surface and lateral spaces. Thereby, reducing the time for computation.

However, before obtaining the evaluation value of a certain state, we have to evaluate all the nodes in the trajectory from root to the desired node, which will increase the number of notes that need evaluation.

Concretely, we calculated, in five directions, the number of consecutive-segments in different colors and shapes brought in by the latest card (two spaces), then subtract the number of "dead" consecutive-segments blocked by the same card. Then compute the evaluation value e'(s) with equation 1. Then the final evaluation value of the state s is:

$$e(s)= e'(s)+e(s\_parent) \quad (2)$$

Where s_parent is the parent node of s. It is easy to realize the evaluation by searching the game tree recursively.

In the beginning, we established a visited tree with each node in the tree represented by an action (move). In the tree, a state can be generated from the root state after a sequence of moves in the trajectory starts from the root to the node. In the procedure of Minimax search, all nodes above the target level will be evaluated and stored, so that the result can be reused in the next time. However, maintaining the gaming tree had been proved to be costly and error-prone.

We prefer the second heuristic h_2 without a visited tree to h_1, not only because it is easier to realize the algorithm of h_2 recursively but also because h_2 calculates faster than h_1.

3 Experiments

3.1 Minimax search compared with alpha-beta Pruning

To study the effect of alpha-beta pruning in Minimax searching, we compared the number of nodes evaluated in Minimax searching both with and without alpha-beta pruning. To get a good comparison, we use the same heuristic function (h_1) for both the first and the second players. We change the depth of our heuristic from 2 to 4, then evaluate the effect of alpha-beta pruning.

3.2 Improving the effectiveness of alpha-beta pruning

Although the state space of gaming is referred to as a tree, it is a directed acyclic graph. Recognizing previously visited nodes allows one to eliminate large portions of the tree traversed by alpha-beta pruning algorithm [1]. To improve the effect of alpha-beta pruning, we use a transposition table to save the visited nodes. Then we analyze the effect of adding a transposition table by comparing the number of nodes visited and the time used for each step.

3.3 The impact of weights in heuristics

Empirically, the weight of different features differs in heuristics. For example, the AI player should take the action to block a three-consecutive-segment as the priority when it is in favor of the opponent rather than generate a newly desired three-consecutive-segment. In this experiment, four sets of parameters (w1, w2, w3, w4) are used to test the performance of heuristics.

3.4 Comparing the performance of two heuristics in terms of time and depth

In this experiment, we compared the performance of two heuristics (h_1 and h_2) in terms of time expenditure when looking in different depths ahead. We also compared the rate of winning and steps needed when competed with a baseline (h_1 with a depth of 2).

3.5 Heuristic function for the tournament

We deployed the first heuristic function h_1 to the tournament. To improve the performance of our AI, we dynamically changed the depth of the heuristic function evaluated, so that it can look as deep as possible within 6 seconds. The depth strategy deployed for the heuristic was as follows:

```
If step<=2:    depth=2;
elif step<=10:        depth=3;
elif step<=21:        depth=4;
elif step<=24:        depth=5;
elif step>=25:        depth=2;
```

4 Results and analysis

4.1 Effectiveness of α-β pruning

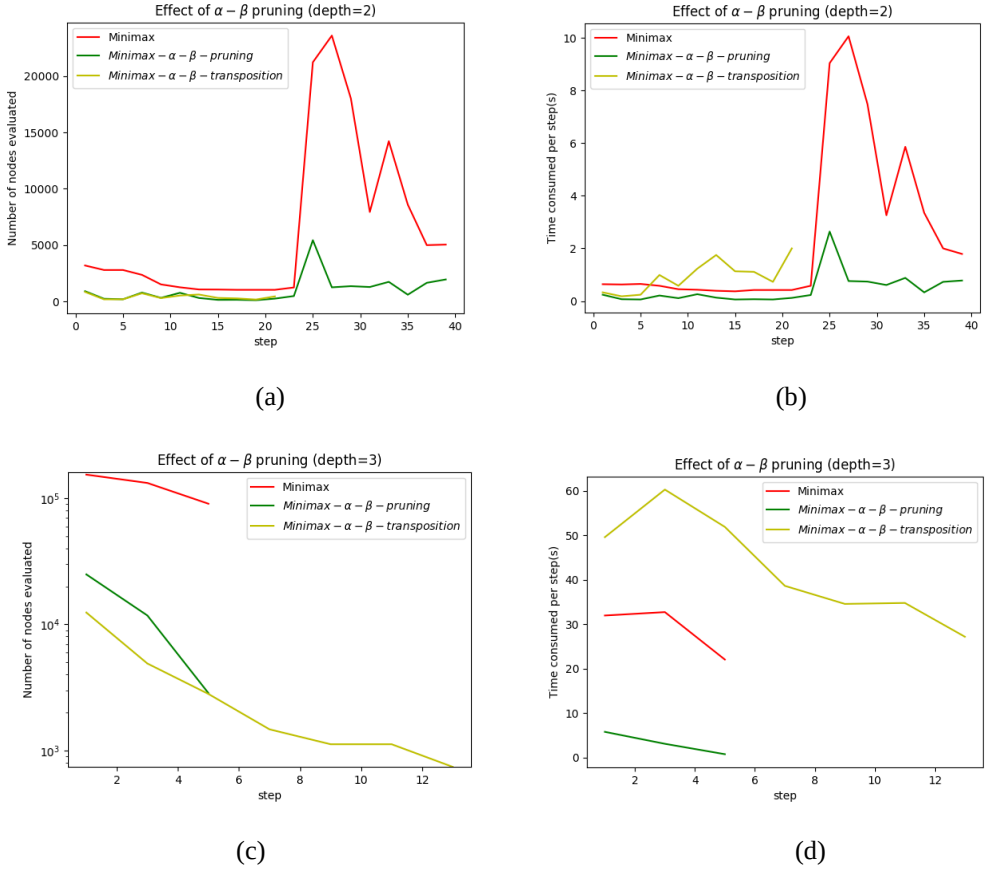

Fig. 1. Effect of α-β pruning compared with minimax only and α-β pruning algorithm with a visited table in terms of time and number of nodes evaluated.

As shown in the figure above, compared with Minimax algorithm only, α-β pruning algorithm significantly reduces the number of nodes needed to be evaluated in the whole gaming process.

When the depth of the algorithm is 2, the number of nodes evaluated each time in Minimax ranged from 3184 to 1024 in the first 24 steps (Fig. 1(a)). While In α-β pruning algorithm, the number of nodes evaluated ranged from 903 to 103. There is a dramatic rise in the number of nodes evaluated since step 25 because the game entered the recycling mode. In general, the proportion of nodes have been pruned ranged from 39.2% to 94.7%. The effect of α-β pruning was also justified by the time needed for each step for the different algorithm as shown in Fig.1(b).

When the depth is 3, the effect of α-β pruning is even obvious. About 84% to 97% of the nodes have been pruned, as shown in Fig.1(c).

As there are a huge number of repeated nodes in our gaming tree, recognizing previously visited nodes allows one to eliminate large portions of the tree traversed by minimax algorithm. Assume the size of action space of a game is n, then there are A_n^2 kinds of permutation for two actions, which means A_n^2 nodes will be generated. When we use a transposition table to recognize visited nodes, only n*(n-1)/2 nodes will be visited. When the depth is 3, then 5/6 of the nodes will be avoided to visit due to repeating.

In this work, when we set a transposition table to store visited nodes, the proportion of nodes have been ultimately pruned ranged from to 42.6% to 93.6% when depth is 2. The effect of α-β pruning based on a

transposition table is no obvious compared with α-β pruning algorithm alone. When depth is 3, the effect of α-β pruning with transposition table is better than α-β pruning alone. But maintaining a transposition table is demanding in this project, thus the time needed for each step increased significantly, as shown both in figure 1(b) and (d).

Considered the constraint of 6 seconds, we discarded the use of transposition table for our tournament at last.

4.2 The impact of weights in heuristics

Another experiment is worth doing in the project is to compare different heuristic functions based on the weight of different features. According to the previous explanation, the first heuristic value is computed by counting the number of the same consecutive spaces and assign them different weights. In the experiment, three different combinations of weights in h_1 will be used to form an AI against each other. The final score of the h_1 will be these four parameters (weights) multiply the number of consecutive-segments for color and dot respectively (see Table. 1)

Table 1. Three different weights combination for heuristic function 1.

Players	Heuristic Weights
W1	h1_weight1 = (5,0.2,-5,-0.2)
W2	h1_weight2 = (5,0.2,-10,-0.2)
W3	h1_weight3 = (5,0.2,-15,-0.2)

Table 2. Result of the matches.

Matches	Game play				
	Depth	Who first	First plays	Who wins	Steps to win
W1 vs W2	3	W1	color	W2(dot)	6
W2 vs W1	3	W2	color	W1(dot)	10
W1 vs W3	3	W1	color	W3(dot)	6
W3 vs W1	3	W2	color	W1(dot)	10
W2 vs W3	3	W2	color	W3(dot)	6
W3 vs W2	3	W3	color	W2(dot)	10
W1 vs W2	2	W1	dot	W1(dot)	6
W2 vs W1	2	W2	dot	W2(dot)	8
W1 vs W3	2	W1	dot	W1(dot)	6
W3 vs W1	2	W2	dot	W2(dot)	8
W2 vs W3	2	W2	dot	W2(dot)	6
W3 vs W2	2	W3	dot	W3(dot)	8

The only change of these three heuristics is the third weight w_3 which represents the three consecutive scores of the opponent. The reason is that the status of three serial cells is very "dangerous" for the player, the opponent will highly possible to win the game in the next placement. In order to prevent that happening, the weight of that will always greater than himself to address that "danger" and we gradually increase the third argument to see whether it will affect the final result of the match. Also the depth the algorithm will search and steps to win the game will be taken into consideration.

To analyze the correlation of the game-winner and the weights, the variable of depth to search and who play first must be controlled. As the result of winner, we can see that there is no clear evidence that the increase of third arguments in h1 would affect the final winner of the game. On the other hand, the depth and who play first is a good indicator to show the winner. At the deep depth = 3, the second player will always win the game, however; at shallow depth = 2, the first player will always win the game.

4.3 Performance of two heuristics

In this work, we compared the performance of two heuristics (h_1 and h_2) in terms of expenditure and result of competition when looking in different depths ahead.

We also compared the result of competition and steps needed when competed with a baseline (h_1 with a depth of 2). Six matches were arranged for different heuristics (h_1(3), h_2 (2), h_2 (3)) to compete with a baseline heuristic h1 with a depth of 2 (h_1(2)). Each heuristic played two games, as the first and second player, irrespectively. The result shows that when the depth is bigger than that of baseline, our heuristics can win the baseline easily, in 5~6 steps.

When the depth is 2 for both heuristics, the first player won. The result is in accord with the result of experiment 2.

Table 3. Competing results of heuristic h_1 and h_2

Match	Player1 *			Player2			steps	Winner
	heuristic	depth	time(s)	heuristic	depth	time(s)		
1	h_1	3	2.75	h_1	2	0.07	6	h_1(3)
2	h_1	2	0.12	h_1	3	1.53	5	h_1(3)
3	h_2	2	0.02	h_1	2	0.10	18	h_1(2)
4	h_1	2	0.12	h_2	2	0.11	6	h_2(2)
5	h_2	3	0.88	h_1	2	0.08	5	h_2(3)
6	h_1	2	0.15	h_2	3	1.02	5	h_2(3)
7	h_1	3	2.30	h_2	3	0.68	11	h_2(3)
8	h_2	3	1.42	h_1	3	1.82	4	h_2(3)
9	h_2	4	25.18	h_1	4	51.01	24	h_2(4)

Note:* player 1 is the first player

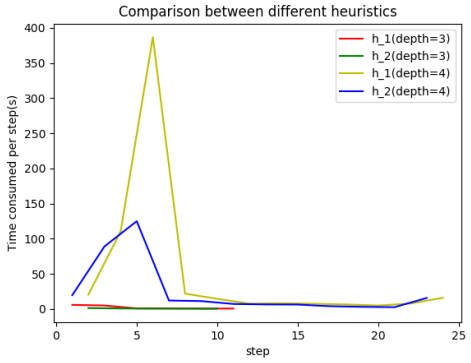

Fig. 2. Time consumption of each step for different heuristics in different depths

The result of three competitions between h_1 and h_2 with the same depth (3 or 4) shows that h_2 is in priority to h_1. H_2 won three matches, while h_1 lost all. The advantage of h_2 was also justified in terms

of time consumption by Fig. 2. When the depth is 3, the speed of h_1 and h_2 is in the same level. However, when the depth rose to 4, time consumption of h_1 increased much more dramatically than that of h_2.

Both heuristic h_1 and h_2 take the number of three-consecutive-segments and two-consecutive-segments as features. They differ in ways of calculating the values of features. Compared with h_1, the advantage of h_2 is that it is easy to recursively calculate the change of value of features based on last move. While h_1 has to brutally search for all surficial and lateral spaces, then find out the number of consecutive-segments o different color and shape, which is demanding.

However, the algorithm of h_1 is clear and robust, and it is easy to calculate e(n) of a state after 24 steps when entering the recycling mode by just considering the change of extra possible removing following the state. When entering recycling mode, it is hard to consider the possible removes after a state, recursively in the way of h_2. To simplify the question, we use the same algorithm for states in recycling mode, unavoidably sacrifice some information.

Overall, both h_1 and h_2 are efficient heuristics for the Double Card game. H_1 may omit some situations when calculating features horizontally, thereby lose the matches to h_2.

4.4 Results for tournament

We used the heuristic function h_1 to take part in the tournament. To improve its performance, we dynamically changed the depth of our heuristic function so that it can look as deep as possible within 6 seconds. Our heuristic won the total of 6 matches in the competition. We didn't use h_2, because, at that time, we set a transposition tree to store e(s), the evaluation value of each visited state for h_2. While maintaining that transposition tree needed huge of extra computation and was error-prone. However, after discarding the transposition tree, h_2 turned to be robust.

5 Discussion

In our work, we deployed Minimax searching with α-β pruning algorithm for the Double Card gaming searching problem. According to empirical knowledge, we chose the number of three-consecutive-segments and two-consecutive-segments in different directions as features. Based on these features we implemented two heuristics, which differ in the way of searching for these features. The main challenge of this work is to efficiently calculate the number of "alive" consecutive-segments in a special state. The first heuristic h 1 searches only for the surface and lateral spaces. Inspired by the design of our terminal checking function (Terminal checker), our second heuristic, h_2 only calculate the change brought in by last move, recursively. However, we need to calculate all the parent nodes in the levels above.

Besides, it is quite hard to get a general method to calculate the number of consecutive-segments precisely for all situations without repeating and omitting. Sometimes, to reduce the computation, we choose to sacrifice some information.

In addition, it is also difficult to assign a proper weight value for the features of our heuristics. Because we cannot quantify the importance of each component in the score calculation. A simple strategy to cover that is to assign the weight value as we discussed in the experiments like the third argument must be greater than the first one. Such an assumption is not applicable and reliable for the heuristic value.

To improve the effectiveness of α-β pruning, we also tried to establish a transposition table which store all nodes had been visited. Although the effectiveness of pruning rise, maintaining this table increased the computational time for each step dramatically.

Our future work could revolve into these main ideas: First, we could work on enhancing the effect of α-β pruning by changing the order of nodes and finding minimal windows of α-β pruning [1, 5]. We could also learn the weights of our heuristics through machine learning [6]. Finally, we could also try to apply Monte Carlo Tree Search (MCTS) to our task. MCTS is a highly effective tree search technique which originated in 2006[7~10], which uses guided forward simulations of a game to estimate the value of potential moves.

6 Team members and Contributions

Team members:

- Jun Shao
- Zhiyi Ding

Contributions:

Jun Shao:

- Main coder.
- General structure of the program.
- Implementation main algorithm like minimax and alpha-beta pruning.
- Ideas for heuristic function and Implementation.
- Report the technical details, heuristics comparison, experiments of the alpha-beta. pruning effectiveness and discussion.

Zhiyi Ding:

- Support coder.
- Ideas for heuristic function and Implementation.
- Implementation of some other details like board making.
- Report the algorithm of minimax and alpha-beta pruning and pseudo code and impact of weights.

References

1. Schae_er, Jonathan, and Aske Plaat. "New advances in alpha-beta searching." ACM conference on Computer science. 1996.
2. Zhang-Congpin, Cui-Jinling. "Improved alpha-beta pruning of heuristic search in game-playing tree." Los Angeles, California USA: World Congress on Computer Science and Information Engineering, Martie. 2009.
3. COMP 472/6721 Project 1 in Winter2019. https://moodle.concordia.ca/moodle/plugin_le.php/3445782/mod label/intro/COMP 472 2019 Winter Project 1.pdf.
4. https://en.wikipedia.org/wiki/Minimax
5. Jonathan Schae_er. The history heuristic and alpha-beta search enhancements in practice. IEEE Transactions on Pattern Analysis and Machine Intelligence, 11(1):12031212, November 1989.
6. Buro, Michael. "Improving heuristic mini-max search by supervised learning." Artificial Intelligence 134.1-2 (2002): 85-99.
7. Sephton, Nick, et al. "Heuristic move pruning in monte carlo tree search for the strategic card game lords of war." 2014 IEEE Conference on Computational Intelligence and Games. IEEE, 2014.
8. Chaslot, G. M. J. B., et al. "Monte-carlo strategies for computer go." Proceedings of the 18th BeNeLux Conference on Arti_cial Intelligence, Namur, Belgium. 2006.
9. Coulom, Rmi. "E_cient selectivity and backup operators in Monte-Carlo tree search." International conference on computers and games. Springer, Berlin, Hei-delberg, 2006.
10. Kocsis, Levente, and Csaba Szepesvri. "Bandit based monte-carlo planning." European conference on machine learning. Springer, Berlin, Heidelberg, 2006.