COMP 6721: Double Card

Martin Khannouz

40093517 `martin.khannouz@gmail.com`

Introduction

This project report describes how the first project of *Introduction to Artificial Intelligence* was developed. The first section covers the environment and the tools used for the project. The heuristic section describes each heuristic and how they were improved. The difficulties section summarizes the difficulties encountered during the project and how they were overcome.

1 Settings and Structures

1.1 Settings

The developing part of this project was realized on an Archlinux distribution with Python 3.7.2. Vim with the Jedi plugin was used to code and a private Github repository was set to keep track of the changes. The *orwell* machine was used before each deliverable to ensure the project worked with the software version on Concordia's computers. Pytest was used to guarantee no regression has been made.

1.2 stucture

The project is structured in classes with the main one representing the engine. This class is an instance of the game and implements the rules. To ensure this class operates properly, unit tests were implemented before the class methods, based on the use cases given by the teacher. This class was optimized as much as possible to guarantee the fastest execution time for *MinMax* algorithm.

The project also contains three types of players. A *Human* player that parse the input from the command line. A *MinMax* player that implement tweaked minimax version with alpha-beta pruning. The *MonteCarlo* player which was supposed to implement a Monte Carlo Tree Search even though it was a complete failure due to my poor understanding of this algorithm. Finally, a random player was developed during the early stage of the project. This "algorithm" served as a control test for other player types because none of them should lose against such a chaotic player[1].

[1] Based on my statistics, even a 5-years old kid should get a win rate of 50%.

Automatic players (*MinMax* and *MonteCarlo*) rely on heuristic to complete their search. All heuristic functions are grouped in one file: *heuristic.py*. Finally, the class *Move* represents a move that could be played.

In addition to these classes, additional functions were implemented. One function used genetic algorithms to optimize the *Vspace* heuristic, but it was too slow and just gave a hint of how to enhance the heuristic rather than accurate weights. A win rate function was also implemented to compare combinations of player and heuristic against each other. Finally, a non-tweaked MinMax version was developed for deliverable 2.

1.3 Tweaks in *MinMax*

The original *MinMax* algorithm explores the complete tree of possibilities and applies the heuristic function on the leaf. However, the heuristic functions described below do not take into account who won first. For instance, a sequence of move could be preferred if a player scores three lines of four symbols even if his opponent already won at the previous move.

To prevent this behavior, the MinMax algorithm was tweaked to check for winning condition after playing each possible move at any depth. This modification prevents the exploration of game states already won by a player.

Even though calling the winning function from the engine was costly at the beginning of the project, I managed to drastically reduce this time by improving the performances.

To prevent exceeding the six-second limit, a timer was added in the MinMax algorithm. If this limit was reached, the algorithm stops and returns the best move encountered until this time.

The *MinMax* algorithm contains a parameter to sort the moves at each level. This approach was supposed to prune the state space faster. At the beginning of the project, this sort improved significantly the performance, but on later versions of the heuristics, it slowed the algorithm. Therefore, it was barely used nor study.

Finally, when the algorithm runs on an empty board, it returns a random vertical move. A vertical move because the heuristics appear to have a better winning rate when they start with this move type. The move was randomly picked to explore openings and to bring small changes from one game to another.

2 Heuristics

Five heuristics were developed in this project even though only two were made for competitive use. One heuristic is the naive function for the second deliverable. Another is a random heuristic that systematically returns a new random number. It was used to spot runtime errors in *MinMax* and *MonteCarlo* algorithms. The *Basic* heuristics only check for winning situations and was used for basic controls. The two other heuristics are described in subsections of their own.

2.1 Convolution

The convolution utilizes the *convolve2d* function from Scipy to estimate the value of a current state. The game board is split into two matrices: dot and color. Each matrix contains -1, 0, or 1. A zero indicates an empty space while the two other values represent either a filled dot and an empty dot, or a white cell and a red cell. Applying *convolve2d* on these matrices produces new matrices with values that range from -4 to 4. A cell with an absolute value of four indicates a row of four and therefore, a winning position. The heuristic gathers all intermediate values into a histogram (using the *histogram* function of numpy) then give a weight to each value. For instance, a weight of 16 was given to each intermediate value of 3 which represent a row of three.

Figure 1 shows an example of the board. Figure 2 focuses on the dot player and show the transformation applied to the board for this player. Filled dots are set to 1 and circles to -1. Zeros are not represented for more clarity. The heuristic applies four convolutions: one for the rows, another for the columns and one for each diagonal. These convolutions generate four smaller matrices showed in Figure 3 (left side). The matrices are then flattened, concatenated and the absolute value is applied to every cell. Finally, the histogram is built out of this matrix and each element is multiplied by a weight before being summed. The final value of the board is the difference between the two player values.

This approach is strong for many reasons. It is faster than calling python code because Scipy and Numpy rely on C written functions. Incomplete rows that contain zeros are more valued than incomplete rows that contain opposite values. Finally, because the convolution mask is applied to empty spaces, the heuristic can build complex structures with space in the middle or above.

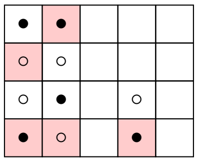


Fig. 1: An example of a board game. Cards are not represented.

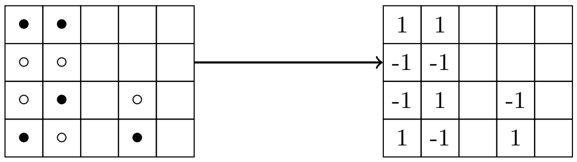
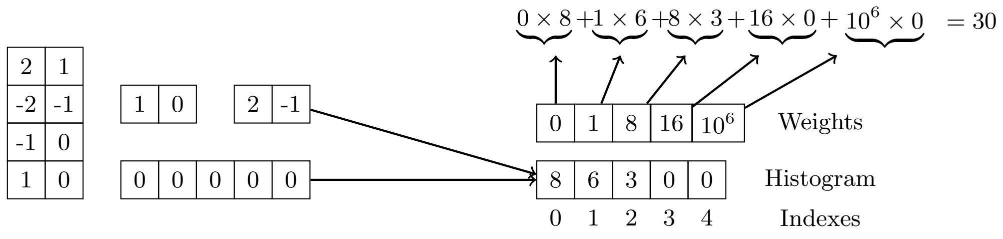


Fig. 2: Example of transformation for the dot player based on the board in Figure 1.



$\underbrace{0 \times 8} + \underbrace{1 \times 6} + \underbrace{8 \times 3} + \underbrace{16 \times 0} + \underbrace{10^6 \times 0} = 30$
2
1
-2
-1
-1
0
1
0
1
0
2
-1
0
0
0
0
0
0
1
8
16
10^6
Weights
8
6
3
0
0
Histogram
0
1
2
3
4
Indexes

Fig. 3: The four matrices obtained after applying the four masks on the matrix that represent the dot, in Figure 2. This Figure also represents the conversion to a histogram and how this histogram is used to find the heuristic value for the dot player.

2.2 Vspace

The goal of the Vspace heuristic is to give value to empty cells in addition to counting rows of similar symbols. Similarly to the previous heuristic, *Vspace* counts the size of the lines, but for every line connected to an empty cell, it uses the size of this line as a value for the empty cell.

Figure 4 and Figure 5 give examples on how the spaces are valued for both players, based on the board at Figure 1. Figure 5 shows that even if the line is split in two with an empty cell in the middle, the combined value of both side gives the value of the space. This is the reason why the two 3-values appears even though there is no line of three.

Possible value for spaces range from 0 to 4 and higher values are downed to 4. When multiple lines are connected to an empty cell, the cell receives the maximum value from them. The heuristic split the space into two categories: the space available in the next turn and the spaces too high to be reached with one card.

Finally, a histogram for each category is made and each value is multiplied by a weight then summed. Note that the two categories have different weights because a space connected to a line of three symbols does not hold the same value if you can play on it on the next turn or not.

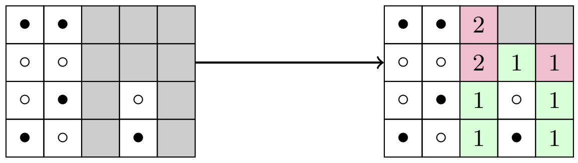


Fig. 4: Example of the Vspace process for the dot player based on the board in Figure 1. the grey cells represent cells that contain nothing interesting. On the right board, green cells are spaces available for the next turn and purple cells are spaces that need at least two cards to be filled.

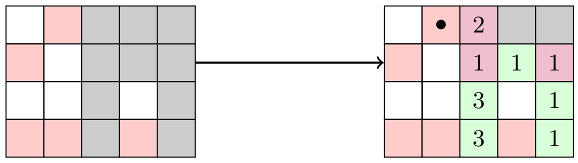


Fig. 5: Example of the Vspace process for the color player based on the board in Figure 1. the grey cells represent cells that contain nothing interesting. On the right board, green cells are spaces available for the next turn and purple cells are spaces that need at least two cards to be filled.

The default weights of *Vspace* were hinted by a genetic algorithm [2]. The DEAP [3] library was used to improve the initial weights which were a simple guess. New weights would compete against a *Vspace* heuristic with the original weights and the winning rate was used to evaluate them. In case the winning rate was zero, the new canditates were sorted by the average length of their game: the longer, the better. After a night of simulation, the results indicate that lines should not be counted except for lines of four symbols. This change, greatly improved the winning rate of *Vspace*.

3 Difficulties

The biggest difficulty encountered during this project was the slowness of python. In the early stage of the project, the code was too inefficient and could not check more than one move in advance within the limit of three seconds.

cProfile [1] is a module to profile python code. In particular, it gives the amount of time spent in each function called. This profiler was used to locate the most time-consuming function, then this information helps to restructure the code and enhance the performance.

However, the results were hardly perceivable and the code remains slow even with the later use of optimized python libraries (Numpy and Scipy [4]). Functions from these libraries rely on built-in functions which are written in C, then compiled into python module. Therefore, the use of this type of function should improve the speed of the engine.

However, none of the approaches above provide significant improvement. Finally, to reach a decent executing time for the tournament, one last step was taken and a python module written in C was developed. Figure 6 shows the evolution of the performances as the most time-consuming function given by *cProfile* were adapted to call the module. However, the python code remains and it is executed by default. Figure 6 shows that the MinMax algorithm is able to check four moves in advance after half of the key part used the module.

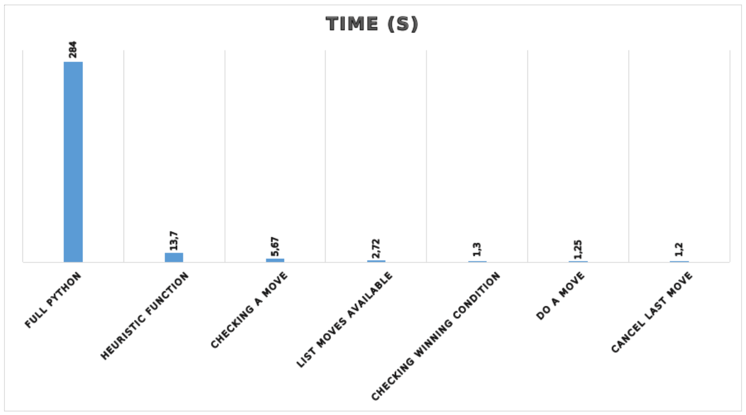


Fig. 6: The amount of time (in seconds) taken to run the MinMax algorithm on a board with the card ”5 B 1” already played. The MinMax algorithm checked up to four moves in advance and it used the Vspace heuristic described in Section 2.2. The left column shows the full python execution time (more than 4 minutes) and the other columns show the execution time as part of the code start using the C module. The third column from the left, ”checking a move”, indicate the time taken when the heuristic function and the function that check the moves use both the C module.

4 Analyze

This section presents three analyses. A comparison between the two heuristics; a brief overview of the alpha-beta pruning impact; and an experiment where the branching factor is limited in order to go deeper.

Before exploring these analyses, a brief explanation about winning rate tables. When two heuristics/algorithms are compared, two situations are considered: one algorithm play first in the first situation then the roles are inverted in the next situation. A quick experiment with the same algorithm and the same heuristic showed that the winning rate is higher for the second player. Therefore, we need to compare two situations.

4.1 Heuristic comparison

In this section, we compare the efficiency between the two heuristics. Table 1a shows the winning rate of each heuristic out of 20 games. *Vspace* is used with a depth of four (four moves in advance) while the convolution only uses a depth of three. This difference in depth appears because the *Vspace* heuristic is more optimized. Both heuristics are used by the tweaked *MinMax* algorithm described above.

The first row of Table 1a indicates that the *Vspace* heuristic wins 70 % of the game when it starts to play while the convolution heuristic wins the other 30 %. From this table, we see that the Vspace heuristic performs better because it also wins all games when the convolution heuristic plays first.

To investigate the efficiency of the heuristic rather than the combination of its performance and efficiency, the previous experiment was redone with both heuristics using the same depth. Table 1b shows the winning ratio for this second experiment. We denote that the *Vspace* heuristic is more efficient than the convolution.

(a) Optimal depth

First player	Vspace	Convolution
Vspace	70 %	30 %
Convolution	100 %	0 %

(b) Same depth

First player	Vspace	Convolution
Vspace	55 %	45 %
Convolution	100 %	0 %

Table 1: Shows the winning ratio between the two heuristics. On the left, both heuristics use the maximum depth they could reach. On the right, they use the same depth (three moves in advance). The column *First player* indicates which heuristic is playing first.

4.2 Alpha-Beta

The impact of Alpha-Beta pruning is important. A quick experiment with the *Vspace* heuristic on depth 4 showed that the alpha-beta pruning speed-up the algorithm by a factor of 26 (from 43 seconds to 1.6 seconds). The efficiency of the pruning is also seen in the number of time the heuristic function is called: 90000 calls with the pruning and more than 4 million without. Therefore, we can conclude that the Alpha-Beta pruning is a key element of the *MinMax*.

4.3 Limiting the width

During the tournament, I heard that a team added a limitation in width in addition to the depth limitation. Therefore, I decided to evaluate this idea in my own project. I design a new version of the MinMax algorithm where at the beginning of each function (*min* and *max*) each move is rated with the heuristic and only the 25 best are explored. This modification fixed the branching factor at 25 and allows the algorithm to go one level deeper. The factor of 25 was choosen because it allows going just one level deeper within the time limit while still exploring many moves.

Table 2 shows the win rate between the *MinMax* with unlimited width and the one with a branching factor of 25. The Table reveals that the limited search with one level deeper has a poor winning rate whether it starts to play or not. Therefore, in this project with the *Vspace* heuristic, limiting the width is inefficient and does not give an advantage.

Table 2: Win ratio when the width is limited.

First player	Limited	Unlimited
Limited	0 %	100 %
Unlimited	35 %	65 %

5 Conclusion

This project taught me many things about AI programming for two player game. It showed me how the weight of a heuristic can be optimized. It proved me that going deeper in the state tree is not the better option if all possibilities are not properly explored. Finally, I learned to program python module and to combine the performance of C with the simplicity of python when performances are required.

References

1. cprofile documentation. https://docs.python.org/3/library/profile.html, accessed: 2019-03-28

2. Banzhaf, W., Nordin, P., Keller, R.E., Francone, F.D.: Genetic programming: an introduction, vol. 1. Morgan Kaufmann San Francisco (1998)
3. Fortin, F.A., De Rainville, F.M., Gardner, M.A., Parizeau, M., Gagné, C.: DEAP: Evolutionary algorithms made easy. Journal of Machine Learning Research **13**, 2171–2175 (jul 2012)
4. Oliphant, T.: NumPy: A guide to NumPy. USA: Trelgol Publishing (2006–), http://www.numpy.org/, [Online; accessed ¡today¿]