Double Card - COMP 472 Project

Abdullatif Dalab

27880960 abdulatif.saleh@hotmail.com

1. Introduction

1.1. The Game
Double Card is an 8x12 board game that consists of 24 identical cards and 2 players. Each card consists of 2 colours and 2 dots, and can be configured in 8 different ways. Players get to decided whether they want to play with the dots or colour option. To win the game, a player who chose the colour option should place 4 consecutive identical colours, and a player who chose the dots option should place 4 consecutive identical dots.

1.2. Papers Content
In this report, I will be discussing how the game was structured in code, the heuristic function, why the heuristic is justified, an analysis of the results of the heuristic during gameplay, and how the difficulties encountered in this project have been surpassed.

1.3. Technical Details
The code for the game was divided into the following four main classes: Board class, Card class, AI class, and the Informed_Heuristic class.

The Board class represents the 8x12 board of the game. It consists of attributes such as a Numpy 2-dimensional array, number of cards, last recycled card, game counter, and the last action played on the board. The methods in the class allowed for the insertion/replacement of card objects into/in the Numpy array and the constant monitoring of the board state to check for a win.

The Card class was meant to represent the concept of a card in the game. Each card object held important attributes that uniquely define it such as its primary location, secondary location, orientation, colour configuration, and dots configuration.

The AI class represents the intelligent agent. It encompassed multiple helper methods that build the minimax function. For example, the generate_legal_states method would take in a board object that represents some state, and generate all the possible children states as board objects.

The Informed_Heuristic class consisted of helper methods that were used to create the final evaluation function used in the implementation of minimax. The details of the heuristic are discussed in page 2.

2. The Heuristic

2.1. Description

The heuristic function built looks for the following 3 main features in a board: 4 consecutive winning segments, proximity, and cavity.

Given that the Board class already had functions that check for 4 consecutive winning segments, I decided to leverage that and modify these functions so they can be used in the heuristic. That's the main board feature that has the most weight. If a board that's getting evaluated has a winning sequence in a row or column, it would return a large number as the state evaluation result. Note that the diagonals are not included. This feature was chosen because it guarantees a win every time a generated child state has 4 consecutive winning segments. Time complexity for checking this feature is O(n^2) and space complexity is O(n).

The proximity feature is concerned with the distance between all the identical cards in the board. If the distance between two cards is equal to 1 and they have identical configuration, the value of the proximity variable in its function is incremented by a 1000. Eventually, boards that have more identical cards with shorter distances between them are favoured, giving the ability to generate sequences of cards that may potentially lead to a win. The distance between cards is calculated by adding the absolute value of the difference between the rows and columns of 2 identical cards. Time complexity for calculating proximity is O(n^2) time and space complexity is O(k) where k is the number of cards on the board.

Cavity is the simplest feature. It checks if there are any empty cells in the last row of the board. The less empty the last row is, the higher the evaluation score of the board. The reasoning here is that the less empty cells you have in the last row, the larger the number of possible sequences on the board. Time complexity for calculating cavity is O(8n) and space complexity is O(1).

Overall, the time complexity of the heuristic is O(n^2) and space complexity is O(n).

2.2. Justification

Initially, I had 2 additional features: 2 and 3 consecutive segments on the board.These proved to be costly **(Figure 1)**. In order to balance the speed of the evaluation function with performance, I decided to remove them because proximity already did a good job at building consecutive segments.This change reduced the time required per move by almost 50% **(Figure 2)**. I have also reduced the evaluation time by having the 4 segments checker only check the rows and columns of a board and not the diagonals. That help reduce the time per move to an average of 3 seconds while preserving performance **(Figure 3)**. As the board gets more filled up, the heuristic starts taking less time, so the challenge was to find a version of the heuristic were none of the initial moves took more than 6 seconds.

The main reason I chose proximity, cavity, and 4 consecutive winning segments (rows and columns) board features is because of their simplicity and acceptable time complexity **(Figure 4)** . The proximity feature drove the AI to always build sequences, while the 4 consecutive winning segments feature guided the AI to making the last winning decision.

3. Analysis Of Gameplay

3.1. Winnings

During the tournament, the AI was able to win 2 games. In the first game, the the opponent placed a card that created a sequence of 3 identical cards. That allowed the heuristic to check for a sequence of 4 identical cards\winning segments in the children states generated, and guaranteed a win. In the second game, the opponents AI ignored the moves of my agent. Through proximity, the AI was able to build a sequence of 3 identical cards, and that opened the door for the 4 consecutive winning segments feature to end the game with a win.

3.2. Losses

The total number of losses in the tournament was 4. By ignoring the opponents moves and status, the AI completed the opponents winning sequence in 2 of the games. For example, the player has 3 cards placed vertically next to each other, and they form a potential winning sequence. The AI disregards that information, placing a card that completes the opponents sequence. The other 2 losses where due to the proximity feature. When the AI starts the game, proximity would lead to building a sequence of identical cards that the opponent can build on. For example, the AI plays a card of orientation 1, and then the opponent places an identical card on top, leading proximity to place the 3rd identical card in the sequence. The opponent can now place the 4th card that leads to a win. Sometimes, the proximity feature drives the AI to shift from building one sequence to another on a different location in the board. This shifting could cause inconsistency in the structuring of a winning sequence, giving the opponent the opportunity to place an undesired move or delaying a possible win.

3.3. Alternative Strategies and Implementations

The heuristic can be considerably improved by having it become more informed about the status of it's opponent during the game. One way that could be done is by adding a feature that checks for 3 or 4 winning sequences that the opponent can reach and then blocking the sequence by placing a card that breaks it. If a more defensive strategy is desired, then more weight would be assigned to this defensive feature in the heuristic function. The optimal strategy would be a mixture of defensive and offensive features.

Given that the AI agents can be set to play against each other, games can be simulated, and so it's possible to generate a dataset that records values of the AI's features per move and the outcome of that move. The outcome can be discrete(win/loss) or continuous. If you decide to train a neural network, then it might be possible for the weights of the features to be learnt.

4. Difficulties Encountered

4.1. Returning the best action

After the first successful implementation of the minimax function, I was able to return the best value up to the root node, but not it's corresponding action. After spending a lot of time looking for a way, I realized that if I wanted to successfully implement a solution, then it's important to be able to trace the recursive function. Following multiple trials, I figured that if the board class had an attribute that stores the last move, then that attribute can be retrieved and propagated back up the tree with the evaluation of that board.

4.2. Generating all possible recycling moves

Given that the number of possible legal recycling moves is greater than that of the placement moves, it was a much more challenging task to generate all possible recycling children states. At first, I tried to hard code the rules of the legal moves, but it didn't take a lot of time to realize that it was not feasible. After giving it some thought, I realized that by storing a list of the cards that can be legally moved during the recycling stage, I would be able to generalize the rules and figure out the legal states in a reasonable way.

4.3. Structuring the card objects

After building the card class, I realized that one card would take two cells on the board. That made me rethink the way I should represent the card object since through my approach, for every card I would have to store two identical objects on the same board. After some thought, I realized that the position of the secondary cell can be directly inferred by the position of the primary cell and the cards orientation. By adding the secondary position attribute to the card class, each part of the card can be identified, so having identical objects on the board is now possible without any complications.

5. References

1. Lague, S. [Sebastian Lague]. (2018, 04,20). Algorithms Explained – minimax and alpha-beta pruning [Video file]. Retrieved from https://www.youtube.com/watch?v=l-hh51ncgDI&t=314s
2. Payne, T. [Trevor Payne]. (2014, 02,01). ALet's Learn Python #21 - Min Max Algorithm [Video file]. Retrieved from https://www.youtube.com/watch?v=fInYh90YMJU
3. Lacobelli, F. [Francisco Lacobelli]. (2015, 06,22). minimax algorithm [Video file]. Retrieved from https://www.youtube.com/watch?v=6ELUvkSkCts

6. Appendix

6.1. Figure 1

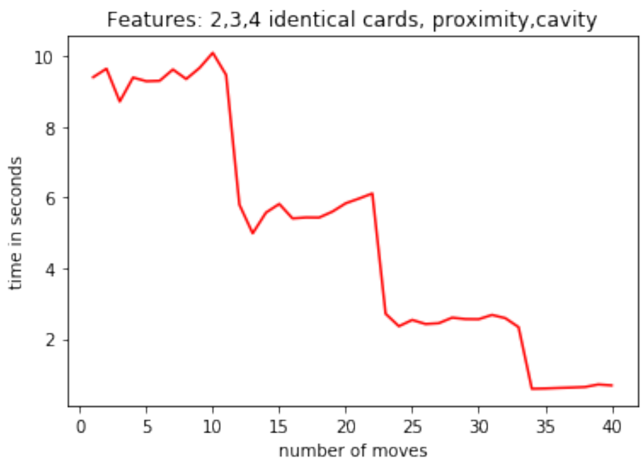

Fig. 1. In this heuristic, the time required per move for the first 20 moves is greater than 6 seconds, making it an expensive evaluation function.

6.2. Figure 2

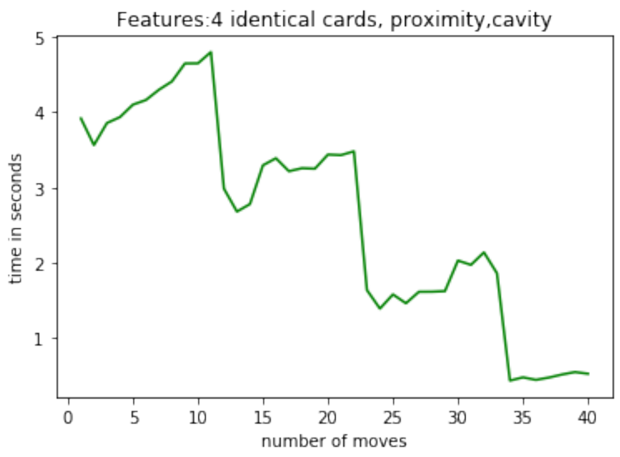

Fig. 2. Given that the proximity feature does a good job at building sequences, removing 2 and 3 identical cards features almost reduced the time required by 50%. (Refer to Figure1 for comparison)

6.3. Figure 3

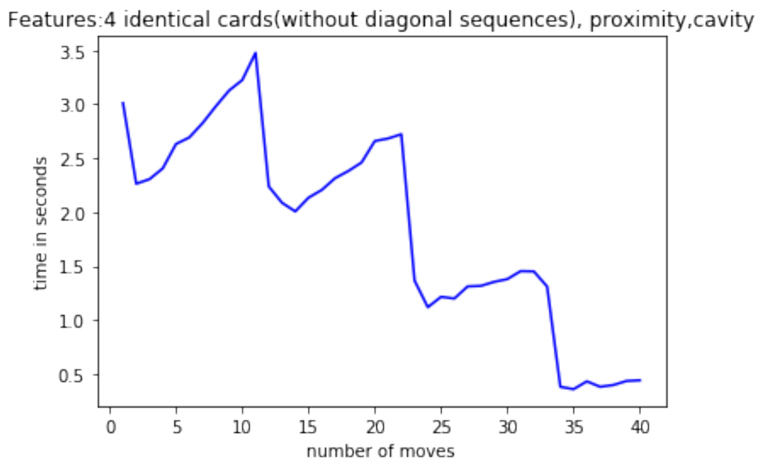

Fig. 3. This is the performance of the heuristic that was chosen in the tournament. The maximum number of seconds given a move was 3.5s. Its average performance and speed made it the best candidate.

6.4. Figure 4

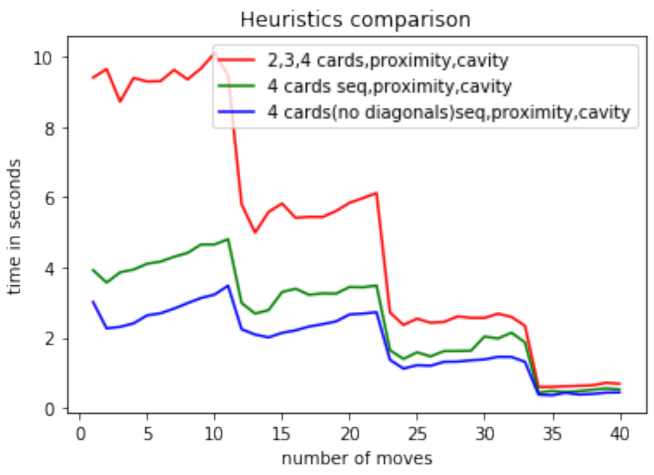

Fig. 4. This is a comparison of all the heuristic functions. The blue line represents the heuristic used in the tournament.