COMP 472 Project 1 Report

Lucas Turpin

ID-40029907 lucas_turpin@hotmail.com

1 Introduction & Technical Details

Project 1 revolved around a fictional board game called Double Card. The project's main purpose was to explore the creation and implementation of heuristics, as well as the mini-max algorithm for state space search in a two player game.

1.1 Double Card Implementation

The Double Card game's rules and interactions were implemented in Python 3.3 using a simple text-based input and output. The game's board state is persisted in memory using an 8×12 matrix for each possible (X, Y) coordinate.

Most of the board game's functionality is captured inside two main classes. All operations relating to the board's state are grouped inside the *GameBoard* class. The *Move* class encapsulates data for each player's moves affecting the board state.

1.2 State Space Search & MiniMax

For the state space search, the *GameBoard* generates a tree structure of possible board states for the MiniMax algorithm to traverse. To avoid expensive duplication of the board's state, each node in the tree is represented by the path of edges from the root to the node. Each edge is represented by the corresponding move leading from the parent node to the child node. The resulting data structure is stored in the form of a multi-level dictionary. At the deepest level of the tree, the structure is flattened into a list as the nodes have no children.

An example of the structure is shown below:

```
root:
    \-> "0 1 A 1" --> "0 1 A 2" --> ["0 1 A 3", "0 2 A 3", ...]
                \-> "0 1 B 1" --> [...]
                \-> ...
    \-> "0 2 A 1" --> "0 1 A 2" --> ["0 1 A 3", "0 2 A 3", ...]
                \-> "0 1 B 1" --> [...]
                \-> ...
    \-> ...
```

2 Heuristic

The MiniMax algorithm requires a heuristic value to determine which states are good or bad for each player. The developed heuristic presents various advantages and disadvantages which shall be presented in the following sections.

2.1 Description

The informed heuristic developed was built around the winning condition for the Double Card game: sequences of 4 identical colors or dots. As either player requires to have an uninterrupted sequence of 4 identical tiles to win, the heuristic associates a value to how close each player is to achieving a full sequence. Partial sequences are attributed increasing weights as they are closer to a full sequence. Sequences that contain both possible colors or both possible dot patterns are worth 0 as it is impossible for the player to gain value from them.

The sequence weights are attributed as follows:

$$f(seq, condition) = \begin{cases} 0 & \text{if } count(seq, condition) = 0 \\ 1 & \text{if } count(seq, condition) = 1 \\ 10 & \text{if } count(seq, condition) = 2 \\ 100 & \text{if } count(seq, condition) = 3 \\ \infty & \text{if } count(seq, condition) = 4 \end{cases} \tag{1}$$

Where the *count* function returns how many of a single winning condition are found inside the sequence. If both possibilities are found (red & white for colors, open & closed for dots), then *count* returns 0.

In order to gather the value of the heuristic for a given board state, the function must iterate over all possible sequences of 4 tiles in all directions and sum their respective weights.

$$e(board) = \sum_{seq \in board} f(seq, \text{COLORS}) - f(seq, \text{DOTS}) \tag{2}$$

The heuristic is also careful to return the correctly signed infinity value based on which player won the game in the case of a draw.

The sequence traversal algorithm for this heuristic was adapted from StackExchange user Phisheep[2]'s answer[1] to a similar problem.

2.2 Justification

Weights In order to properly capture the importance of the length of a sequence, weights are attributed following a logarithmic growth based on length. The base 10 is chosen to make sure that a single sequence of arbitrary length *n+1* will most likely outweigh other possible outcomes with *n*-length sequences.

If a move m_1 affects at least 10 times more sequences of length n than another move m_2 affects sequences of length $n+1$, then m_1 is deemed more likely to win the game than m_2 as it is assumed to lead to significantly more winning board states.

Any move affects up to 17 sequences on the board at once: 4 vertically, 4 horizontally and 2 diagonally per tile, times 2 tiles, minus 3 overlapping sequences. Therefore, varying the logarithmic base of the weights over $]1, 17]$ would result in varying degrees of priority to longer sequences over multiple smaller sequences. A value for the base > 17 would result in always prioritizing longer sequences over any number of smaller sequences.

Features This heuristic is solely concerned with sequences on the board. This is due to how sequences are the only way of knowing if a given board state is a winning board state for either player. Other options such as associating a value to each position on the board and weights for different tile values are not sufficient to determine win conditions. As finding the path to a winning board state is the goal of state space search, using a heuristic that cannot properly qualify the goal state would be counterproductive.

Furthermore, other features such as the order of moves or the most recently played moves could be added but do not impact decision-making as significantly as sequences to warrant including them into the heuristic.

Informedness vs. Look-ahead Due to the strict constraint on allocated run time to find the next best move, the heuristic had to be designed with a compromise on informedness versus speed in mind. The heuristic was chosen to be more informed although much slower to compute. This results in a serious horizon effect as the heuristic could not be computed further than 2 moves ahead of the current board state.

On the other hand, the heuristic will prevent skipping over a winning move and will always block an opponent's winning move, granted they are found within the 2 move horizon. Due to how difficult it is to plan a complex multiple-move strategy in the Double Card game, this low horizon was accepted to make more informed decisions within that horizon.

The high branching factor for the Double Card game also made it quite difficult to increase the maximum horizon significantly using a more naive heuristic. Even very simple iterations could not reach more than 1 level deeper than the more informed heuristic.

3 Results

3.1 Tournament

Testing the heuristic against other teams during the tournament yielded quite encouraging results with a score of 6-0 against three different teams.

The first important result from the tournament is that the heuristic correctly identified winning board states. In all 6 games, the algorithm played the winning move as soon as it was possible to win and always blocked losing board states without giving the opponent a chance to win.

3.2 Local Tests

However, while testing locally by facing the algorithm against itself, some different results are gathered.

The moves selected by both algorithms seem to be deterministic when given the same input parameters (which player goes first and their win condition). This reinforces that the heuristic does not prioritize paths at random and is based on the given board state.

Furthermore, as the moves are deterministic, so is the outcome of each game. The player playing for colors p_c always wins, while the player playing for dots p_d always loses.

When p_c goes first, the game is ended in 9 moves where p_d does not block a sequence of 3 white tiles. When p_d goes first, the game is ended in 14 moves where both players do not block their opponent's respective sequence of 3 but p_c ultimately wins 2 moves later.

This identifies a clear flaw in the algorithm for finding p_d's next best move. What is even more surprising is that p_d evaluates the best path's $e(n)$ equal to positive infinity, therefore acknowledging that the path is winning for the opponent.

4 Discussion

One of the most difficult parts of implementing the heuristic and MiniMax algorithm was making sure that draws were properly handled when evaluating the heuristic to ensure the correct sign was associated to the infinity value. Using a Test Driven Development approach helped tremendously in solving this issue by separating key parts of the algorithm for verification. As multiple modules are involved in the process of generating the player's next move, isolating them through unit tests helped understand the source of bugs as well as how to fix them.

Another source of difficulties was determining how to structure the state space traversal data. The decision to use python dictionaries was made to easily

represent the one to many relationship between parent nodes and their children, this unfortunately required making the *Move* objects hashable to set them as dictionary keys.

As it was expensive to copy the root node's board state to generate child nodes' board state, the decision to store moves instead of boards in the data structure was made. This helped make the evaluation of the expensive heuristic feasible within the given time constraints since inflating the tree data structure was much faster.

References

1. Check five in a row for a 15x15 chess board (part 2) - Code Review Stack Exchange, `https://codereview.stackexchange.com/a/161025`. Last accessed 28 March 2019.
2. User phisheep - Code Review Stack Exchange, `https://codereview.stackexchange.com/users/135889/phisheep`. Last accessed 28 March 2019.