جامعة الأخوين

ⵜⴰⵙⴷⴰⵡⵉⵜ ⵏ ⵓⵅⴰⵡⴰⵢⵏ

AL AKHAWAYN UNIVERSITY

SCHOOL OF SCIENCE AND ENGINEERING

WEAKLY SOLVING THE THREE MUSKETEERS GAME USING ARTIFICIAL INTELLIGENCE AND GAME THEORY

EGR 4402: Capstone Design

December 09th, 2017

ALI ELABRIDI

Supervised by:

Dr. Nisar Naeem Sheikh

CAPSTONE FINAL REPORT

I applied ethics to the design process and in the selection of the final proposed design. And that, I have held the safety of the public to be paramount and has addressed this in the presented design wherever may be applicable."

Ali Elabridi

Approved by the Supervisor

Dr. Naeem Sheikh Nisar

ACKNOWLEDGEMENTS

I would to express my deepest appreciation and gratitude to Dr. Naeem Nisar Sheikh for his continuous effort into making my journey at Al Akhawayn University since day one such an eye opening and enjoyable one. It was truly a pleasure working with you. This Capstone project would not have been a success without guidance.

Contents

ABSTRACT

The objective of this capstone project is to weakly solve a two-player zero-sum game with perfect information called the Three Musketeers using artificial intelligence and game theory. The purpose of weakly solving a game is to draw a conclusion on who would win, lose, or ends in a draw if both players play optimally from the beginning till the end of the game. A tremendous number of games have been weakly or strongly solved in the past, such as 6 X 6 LOA (Weakly solved and proved to be a win for the first player), Minichess 5x5 (weakly solved and proved to be a draw). The goal of our research is to weakly solve the Three Musketeers game, and prove that the first player (Musketeers) always wins if he plays optimally. This Game has been strongly solved in the past by Johannes Laire in 2009, and our end objective is to confirm his results and provide a thorough analysis of the methodology used.

1. Background

1.1 Idea of Solving Games

This Capstone project will focus on solving a game using artificial intelligence. It will combine two of the most interesting fields in computer science and mathematics: Artificial Intelligence and Game Theory. The principal objective of this capstone project is to witness the incredible capabilities that "intelligent" computers could achieve as they are able to process, store, and handle more data and computations that any human could ever handle by his own. Many researches have focused lately in the field of Artificial Intelligence because of the tremendous potential that the growing computational power is able to achieve in many areas such as health science, predicting market stocks, weather forecasting, autonomous cars, and many others. The growing computational power has also allowed the use of artificial intelligence and strategic planning in game theory as more researchers are able to prove the outcome of a given game by only using their personal computer. Solving a game is to predict the outcome of a game and deduce whether it will result in a win, lost, or draw for a given player. Nonetheless, games differ in many characteristics such as the number of players, the board size, the possible game moves at any state, the information provided to the players, and the winning-losing balance between the players, i.e., zero-sum or non-zero-sum game. The difference in game characteristics results in the increase or decrease of different computational measures such as the state-space complexity, branching factor, game-tree complexity and the average game length, which makes it very hard to completely search for solutions for some games as they require hundreds of thousands of years of computation even with our current technological and computational advances.

1.2 Type of Solving

The purpose of solving a game is the ability to predict the outcome of it. In two-player games with perfect information and zero-sum, multiple concepts of solving have been suggested, among them ultra-weakly solving, weakly solving, and strongly solving.

Ultra-weak: proving that from the initial position(s), and given perfect game play by both players, the game will result in a win, or loss of one of the player or ends up in a draw. This

type of solving does not need to generate the perfect moves, and could use a non-constructive argument.

Weak: this strategy not only relies on proving that from the initial position(s), and given perfect game play by both opponents, the game will result in a win, or loss of one of the player or ends up in a draw, but also provides an algorithm that will secure the win of one of the player assuming that the other player will play optimally, or might result in a draw. The algorithm used should only analyze the moves that are optimal by both players, which does not necessarily mean that the algorithm could reach the same outcome if the other player plays imperfectly.

Strongly: this strategy relies on providing an algorithm that will secure for any of the player a win with a sequence of perfect moves given any possible board configuration even if one of the players has made some mistakes in previous moves. However, securing the win will only be possible if the theoretical-game value of the game is still in favor of that specific player; otherwise, it will be impossible to recover from the previous mistakes and will result in a draw or a win of the other player.

Over the past century, game theorists described ultra-solving as the strongest and most challenging kind of proof as it needs an abstract and deep understanding of the properties and characteristics of the game, and the patterns and possible outcomes that may result in as to prove the theoretical value of the game. On the other hand, strongly solving and weakly solving are seen as trivial as they both only require brute force algorithms such as min-max algorithm and alpha-beta pruning or retrograde analysis to provide the perfect moves for a certain player, and deduce the game theoretical value by analyzing the whole game tree from the leaf nodes to the initial board configuration. However, applying brute force algorithms is not always a possible option as the game complexity and a state-space search of some games have been proven to be too large to analyze the whole game tree in our life time. Therefore, researchers still believe that such solving is a challenge on its own for non-trivial games, and in the necessity to develop optimization techniques for weak and strong solving.

2. Literature Review

2.1 Methods of Solving

2.1.1 Minimax Algorithm

Minimax algorithm is a recursive adversarial search algorithm used in n-player zero-sum games with perfect information in order to determine the perfect move of a certain player depending on the next move of the other player(s) alternatively [1]. It is similar to a depth-first search and is used to determine the maximum benefit (utility) that a player might get into a specific position. The minimax algorithm is based on the idea of minimizing all possible losses that may occur (maximum of the possible gains). In other words, the minimax algorithm ensures the largest value or outcome that a player might get without knowing what exact move his opponent might play; similarly, it is also the minimum value that his opponent(s) might force the player to get depending on his move. For two-player games, the minimax max algorithm is formally defined as in equation (1) and possibility implemented as in Figure (1).

$$v_{maximizer,state} = max_{a \in actions(state)}(min(v_{minimizer,result(state,a)}))$$

Equation 1: formal definition of minimax algorithm

```
MiniMax Input: Start state s, Flag MAX or MIN
Output: Utility value Utility(s)
1  begin
2  |  if TerminalState(s) then
3  |  |  return Utility(s)
4  |  end
5  |  {v_1, ..., v_n} be children of s if MAX then
6  |  |  return max{MiniMax(v_1, MIN), ..., MiniMax(v_n, MIN)}
7  |  end
8  |  else
9  |  |  return min{MiniMax(v_1, MAX), ..., MiniMax(v_n, MAX)}
10 |  end
11 end
```

Figure 1: Minimax algorithm pseudo-code.

The minimax recursive approach will determine the next move for the maximizer in order to maximize his gain or minimize his loss. Every possible state for a player is evaluated by a certain gain or loss value used by the minimax algorithm to determine which action to perform next and called the utility value that describes how convenient is the resulting state for a certain player. This utility value is evaluated using a utility function that assign a value depending on the winning/losing rules of every opponent when it reaches a terminal state that is described as the end of the game. This utility value is then used by both player to maximize their benefit and minimize their opponent's. Figure (2) illustrates an example of a minimax generated game tree. In depth 0, the first player (maximizer) for whom the minimax algorithm is searching for the best course of action, is exploring the minimizer evaluation of the 2 possible actions that it may take (branching factor of 2 in the depth 0). In depth 1, the minimizer is exploring all its possible actions given the moves of the previous player (maximizer) in order to choose the minimum useful one for the maximizer. In depth 2, the maximizer explores all the possible moves that the minimizer may have chosen given its previous move. At this level all the possible actions of the maximizer are terminal states and therefore evaluates their convenience using the utility function that depends on the game-rules. The maximizer chooses then the maximum of every possible action, and the minimizer apprehending his outcome will play a move that will lead to the minimum of his outcomes in depth 1, and so will the maximizer in depth 0 by choosing the maximum outcome from the chosen minimums of the minimizer. This strategy will generate the most optimal game play for both players and will result in a utility of 3 for the game.

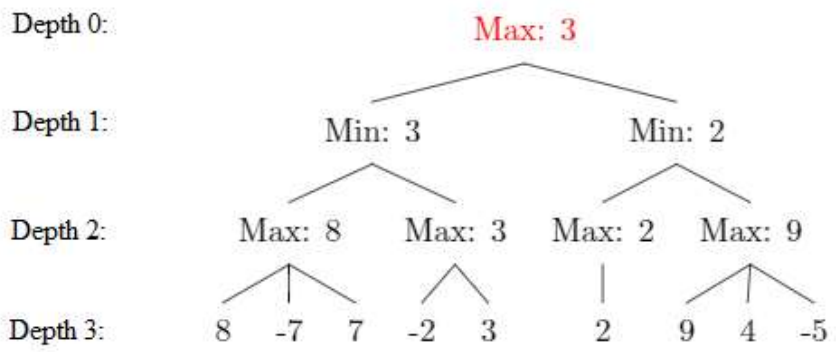

Figure 2: Example of Minimax game tree.

2.1.2 Alpha-Beta Pruning

The computational complexity of the naïve minimax algorithm depends on the maximum numbers of plies (depth of the game tree) and the number of possible actions in every state (branching factor). Therefore, the computational complexity of the minimax algorithm is $O(b^d)$, similar to depth-first search. (b) being the branching factor and (d) the depth of the game tree. However, the performance of the minimax algorithm can be improved by a pruning technique called alpha-beta pruning without affecting the result of the search. This pruning technique is known to reduce the computational complexity to $O(b^{d/2})$ [3]. The core idea of the pruning is that it stops evaluating a certain branch at a certain level by not exploring all its nodes when it has found that the analyzed move happens to be worse than previous analyzed move as illustrated in figure (3) and (4). More specifically, the algorithm keeps track of two values: alpha (α) and beta (β). Alpha represents the worst utility value the maximizer player is guarantee to get at that level or above, and beta is the worst utility value the minimizer is guaranteed to get at that level or above. Both of these values are initialized respectively to -∞ and +∞ that illustrate the worst case for both players at the beginning of the game.

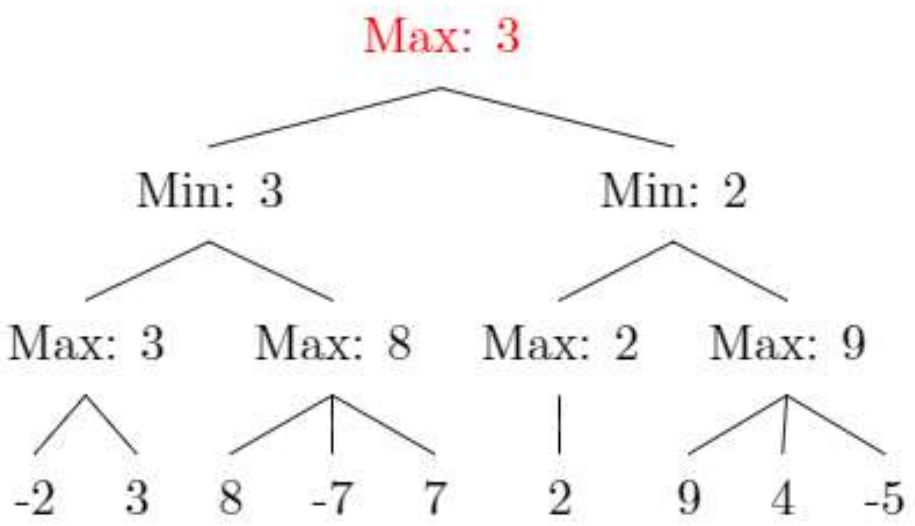

Figure 3: Example of naïve minimax algorithm

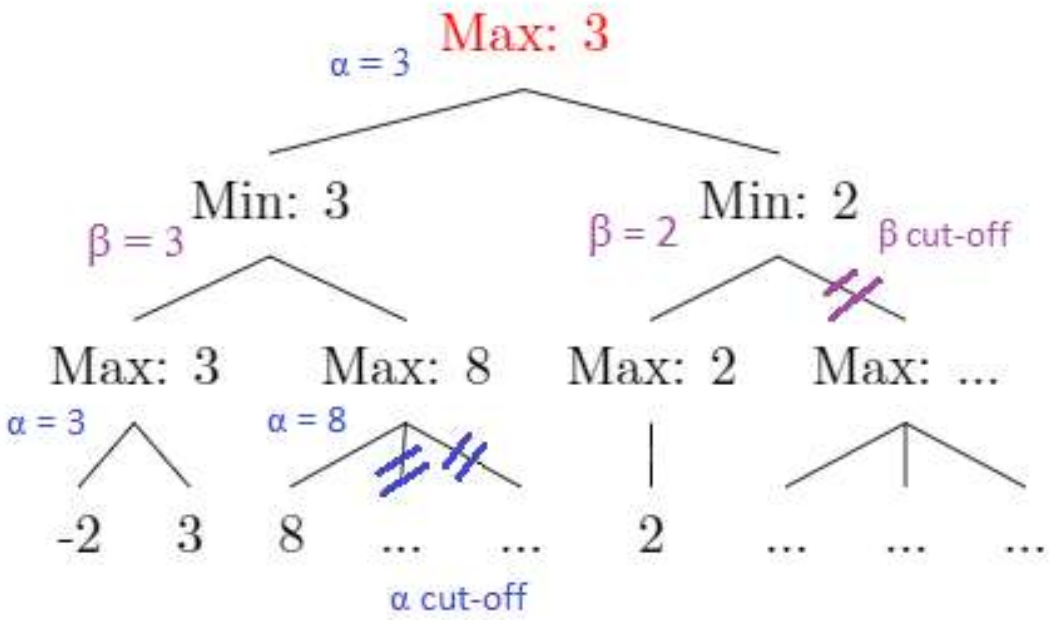

Figure 4: Example of minmax algorithm with alpha-beta pruning

Figure (4) illustrates the branches that will not be explored because the maximizer or minimizer has found a better move. At every depth, the minimizer updates its beta value to be the minimum possible value compared to the utility values found so far, and the maximizer maximize its alpha value to be the maximum possible value compared to the utilities explored so far. The algorithm then checks whether the alpha value is greater or equal to the beta value, in such case, it breaks and does not explore the remaining nodes of that sub-branch as described by the pseudo-algorithm of alpha-beta pruning in Figure (5).

```
AlphaBetaSearch Input: Start state s, Flag MAX or MIN, Alpha α, Betaβ
Output: Utility value utility(s)
1  begin
2      if TerminalState(s) then
3          return utility(s)
4      end
5
6      if MAX then
7          bestVal ← −∞
           foreach {v_1, ..., v_n} be children of s do
8              bestVal ← max(bestVal, AlphaBetaSearch(v_i, MIN, α, β))
               α ← max(α, bestVal)
               if β <= α then
9                  break
10             end
11         end
12         return bestVal
13     end
14     else
15         bestVal ← +∞
           foreach {v_1, ..., v_n} be children of s do
16             bestVal ← min(bestVal, AlphaBetaSearch(v_i, MAX, α, β))
               β ← min(β, bestVal)
               if β <= α then
17                 break
18             end
19         end
20         return bestVal
21     end
22 end
```

Figure 5: pseudo-code of minimax algorithm with alpha-beta pruning

2.1.3 Alpha-Beta Heuristics Improvement

As detailed in earlier sections, the minimax algorithm is an exhaustive search that will explore all the possible game actions for all possible positions or board configurations, and alpha-beta pruning is used on top of minimax algorithm in order to cut-off branches of the game search tree to speed-up the search. However, this cut-off is dependent on the order of actions taken that will change the order of the values of α and β that trigger the cut-off. More generally, the best order is to explore ,from the perspective of the maximizer, children with the highest utility values first, and from the perspective of the minimizer, children with the lowest utility values first [3]. Figure (6) illustrates an optimal alpha-beta ordering game tree.

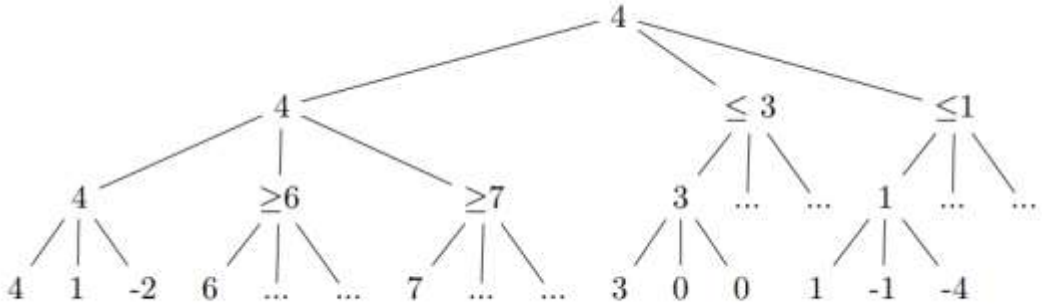

Figure 6: Ordering of a game search tree that produces an optimal alpha-beta pruning

This improvement in the alpha-beta pruning order of actions can be produced by using ordering heuristics that evaluates the best actions and feed them first to the minimax algorithm in order to maximize the cut-off of the game search tree. These heuristics are game dependent. For example, in Chess game, an optimal ordering strategy could be to examine the moves that capture pieces of the opponent before those that do not. Another widely used heuristic strategy called the killer heuristic is to examine the moves that produced the maximum alpha-beta cut-off in previous branches of the same level [4].

2.1.4 Retrograde Analysis Search Algorithm

Another technique used to solve game is the retrograde analysis. The core idea is to start the search bottom-up instead of top-down by generating all end-game terminal board configurations (e.g., checkmate) and then reason backward by generating “unmoves” until reaching the starting position of the game (root node of the game tree). the game-theoretical value of the end-game terminal board configurations can be trivially evaluated, and thus, we can determine the game-theoretical value of all the other board configurations along the way while unmoving by alternating between the two players. The main advantage of the retrograde analysis is that all the possible board game configurations are explored and evaluated, and therefore, it permits to find an optimal solution if exists for any position in the state space. This search strategy is mainly used for strongly solving a game as it generates a database for all possible end-game board configurations. However, the main disadvantages of this technique are that it requires a lot of CPU and memory in order to store the whole state-space. Therefore, close attention is taken to hash the board game in order to distribute the state-space

into multiple databases, and in reducing the game complexity by making use of the properties of the board such as its symmetry [5].

2.2. Previously Solved Games

A tremendous number of attempts have been made to solve some popular non-trivial games in order to predict what would be the outcome of the game if both players play optimally using different strategies. As an example, Hex and NIM game have been ultra-weakly solved and shown to be a win for the first player by using a strategy-stealing argument for the former and mathematically by the latter using what is known as the nim-sum. On the other hand, games such as Qubic, Sim, 6x6 Othello, Checkers have been recently weakly solved. Qubic has been weakly solved in 1980 by Oren Patashnik. Sim has also been weakly solved and proven to be a win for the first player. 6x6 Othello has been weakly proven to be a win for the white player using minimax algorithm in less than 100 hours. Finally, Checkers game (English Draughts) has been recently weakly solved by Jonathan Schaeffer and proven to be a draw if both players play optimally. On the other hand, few games have been strongly solved as it is only possible when the space-search tree is not too large. Games such as Three Musketeers (Musketeers can force a win), Connect Four (first play wins), Pentago (first player win) have been strongly solved. However, some games remain and might forever never be solved because their extremely large game tree would take trillion of years to analyze. As a result, we can only heuristically evaluate their outcome using evaluation functions that estimate the utility of a board configuration in order to create an AI game engine capable to beat humans as it is now the case for Chess, and Go [9].

3. Problem Addressed

In this capstone project, we will try to choose a non-trivial game to be solved with a considered state space search that is computable in the time scope of a semester. The game has to be a zero-sum, two-player, perfect information game, in which we can apply adversarial search algorithms to find the best moves for both player in order for them to play optimally. The end goal of the research is to prove that the game will end in a win, loss, or draw for one

of the player. Moreover, another goal of the research is to find a game that has been previously solved in the past and confirm the result using a different approach.

3.1. Three Musketeers Game

The Three Musketeers game is an abstract strategy board two-player game such as Chess, Go and Nine Men's Morrus. It was developed by Haar Hoolim, and It is part of the Sid Sackson' A Gamut of Games book published in 1969 that includes a large number of notorious games [7]. The game is a zero-sum game, meaning that when a player wins, the other one consequently loses. Moreover, the game is a perfect information game, meaning that the information about the current state of the game, and the moves of the other player are known at any time.

3.1.1 Board Game

The board is a 5x5 board game. Two players will take turn to play. One player is represented by the Musketeers, and the other one by the Guardsmen. The game uses chips to represents the two opponents. The initial position of the pieces (Figure 7) has three Musketeers, and the rest of the board is filled with Guardsmen.

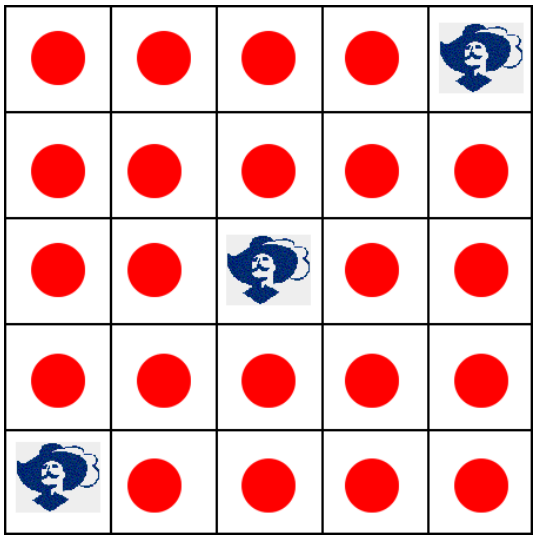

Figure 7: Initial board configuration of the Three Musketeers game

3.1.2 .Legal moves

The two players will take turn each one after the other, starting from the Musketeers. They both have different possible legal moves. The Musketeers are allowed to move horizontally or vertically in any adjacent cell only if there is a guardsman in it, and, the guardsman is removed from the game after the musketeer takes its place. On the other hand, the Guardsmen are allowed to move horizontally or vertically in any adjacent cell as long as it is empty. Figure 8 shows the different possible moves of the musketeers at the beginning of the game.

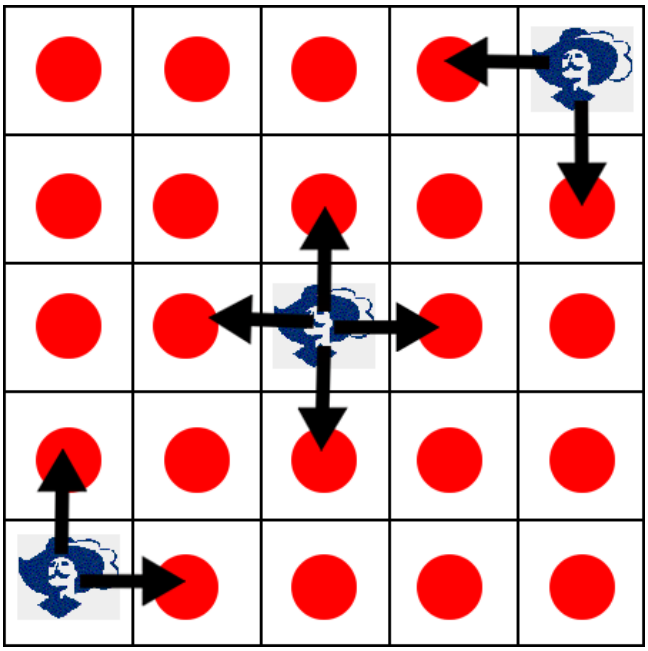

Figure 8: The possible moves of the musketeers at the beginning of the game

3.1.3. Players' Objectives

The players have different objectives. The guardsmen win if they can force the Musketeers to be aligned vertically or horizontally at any moment. On the other hand, the musketeers will win if they do not have any other legal move to play and are not aligned vertically or horizontally. In figure 9, the guardsmen have been able to force the musketeers to align horizontally and thus win the game while in figure 10 the musketeers do not have any possible move left and are not aligned neither horizontally nor vertically, thus they win the game.

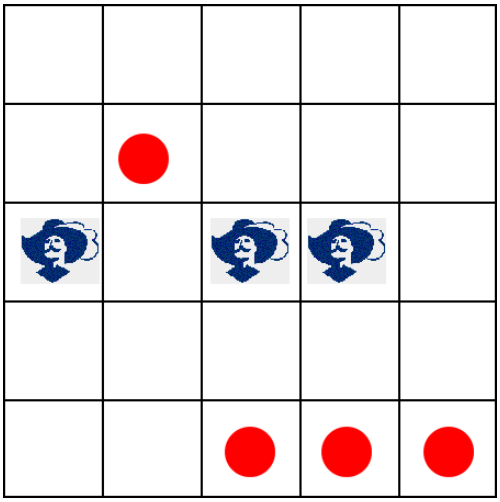

Figure 9: Game state in which the Guardsmen won

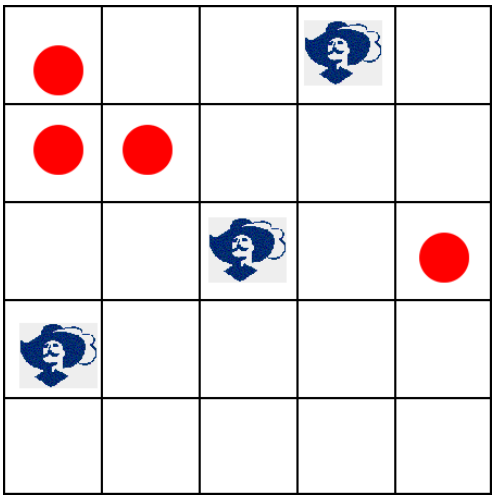

Figure 10: Game state in which the Musketeers won

4. Methodology

4.1. Solving Strategy

The Three Musketeers game has been previously solved in 2009 by Johannes Laire using retrograde search analysis. His work showed that the Musketeers always win if both players play optimally from the initial board configuration [14]. The purpose of this research is to confirm the results of Johannes Laire by attempting to prove that the Musketeers always win if both the Musketeers and Guardsmen play optimally. His work focused on strongly solving the game by not only proving the later claim but also provide a winning sequence of moves for both player if they are in a winning position. We will, on the other hand, use a different strategy of weakly solving the Three Musketeers game by proving in a top-down manner instead of a bottom-up approach the win of the Musketeers using the minimax algorithm that will provide the perfect moves for both players, and analyze the resulting leaf(s) in our game search tree. The implementation of the minimax algorithm is done in Python 3.7 [8], and the structure of the adversarial search algorithm minimax is inspired by the implementation of AIMA of the university of California Berkely[1]. The implemented program includes three main components: game state representation, game representation, and alpha-beta search player. The game state representation includes all the information corresponding to a certain

position and board game configuration. The game representation embeds all the rules corresponding to the Three Musketeers game. Finally, the alpha-beta search player represents the minimax algorithm with alpha-beta pruning, hashing optimization, symmetry checking of the board game, and state memorization (Implementation available in appendix A). This actual code of this research is available on GitHub (https://github.com/alielabridi/Weakly-Solving-Three-Musketeers-Game).

4.2. AIMA Game Structure Implementation

AIMA Game implementation of the University of California Berkely for the Peter Norvig's Book provided a good game representation approach for the minimax algorithm that we follow in implementing the solution for the Three Musketeers game in order to avoid any false-positive results that may happen due to logical errors in the implementation of our algorithm to weakly solve the Three Musketeers game. The AIMA implementation provides an abstract structure for the game representation as a superclass to be used, and a representation for the game states. The game state is represented as a class containing three major attributes that are needed for the description of a certain board in a certain period in the game: to_move, which identifies the players whose move is in that game state, utility, which describes the utility of that state for the player, board, which stores the board configuration of that state. This superclass representing the game structure has 5 methods that needs to be implemented in the subclass representing our game: actions, result, utility, terminal_test, and to_move. The actions method needs to return a list of all possible moves from a specific state. The result method returns a wrapped game state that representes a certain move given a previous state including all the other information to represent a game state. The utility method returns the usefulness of a particular state to a particular player, e.g., the Guardsmen have a negative utility in a terminal state in which they could not align the Musketeers in the same row/column or Musketeers have a positive utility in which they are not able to make any more moves (terminal state) and are not aligned orthogonally. Board configurations that are not terminal states have a utility of 0 as neither the Musketeers nor the Guardsmen are winning in that specific state. The terminal_test method checks whether it is a terminal state given the rules of a specific game. To_move method returns the player whose move is in a given state.

This abstract implementation represents a clean and well-organized structure that we will use in implementing the three musketeers game.

4.3. Game State Implementation

As described in the earlier sections of the AIMA implementation, we will follow the same structure, and the same content and implement the classes as they are described. The game state contains three attributes: to_move, utility, board. These three attributes are sufficient to describe a certain state of the game of the Three Musketeers game to run the minimax algorithm.

4.3.1. To_move attribute

To_move attribute represents one of the two players playing in the Three Musketeers and whose move it is in the given state, i.e., the musketeers or the guardsmen. We will store this attribute as character of 1 bytes in python ‘M’ or ‘G’.

4.3.2. Utility attribute

The utility attribute describes the usefulness of a certain state for a given player. We will assign to the utility attribute three possible values 1, -1, or 0. The value 1 will represent the win of the musketeers or the loss of the guardsmen if it is a terminal state, e.g., if the musketeers are not aligned orthogonally, and there are no more possible moves. The value -1 will represent the win of the guardsmen or the loss of the musketeers in a terminal state, i.e., the musketeers are aligned orthogonally. The value of 0 will be assigned to the utility if any of the player have won yet, and thus, it is not a terminal state.

4.3.3. The board attribute

In every state, we need to keep the configuration of the board in order to generate all possible next moves. The Three Musketeers is a 5x5 board game that may represent three possible value in every position in the board: Musketeers, Guardsmen, or empty. The implementation chosen for the board attribute is a 5x5 2D list of characters that can be one of the possible values: ‘M’ for a musketeer, ‘G’ for a guardsman, ‘ ‘ for an unoccupied position.

4.4. Game Implementation

Game implementation is the class that will include all the rules and mechanisms of the Three Musketeers game. We will implement the different methods presents in the AIMA's abstract game representation superclass in order to use them in our minimax implementation to solve the Three Musketeers game [9]. In addition to the 5 mandatory to implement methods present in the abstract game super class, we will add another method called compute_utility that will compute the utility of a certain board state as the utility values are not static but depends on the board configuration in the Three Musketeers game in that specific state.

4.4.1. To_move method

The to_move method will return the player which will play in that specific state.

4.4.2. Actions method

This method returns the list of possible next moves (similar to a successor function) from a given state. The list of possible next moves depends on whose player it is to play in a specific state as the rules of play differ between the Musketeers and the Guardsmen. As described in the rules of the game earlier, the musketeers can only move to adjacent cells if a guardsman is present in it, and will kill it. In the implementation of the actions method for the Musketeer, it will check first where the musketeers are positioned. Afterwards, it will check the adjacent cells, and if a guardsman is positioned in there it will produce a possible next board with the musketeer moving to that position and emptying his previous position. For the guardsmen, we will check all the position of the guardsmen first, and if found, it will check the adjacent orthogonal cells. if empty, it will produce a new board with the guardsman moving to this new position.

4.4.3. Result method

The result method wraps all the possible game moves of the previous state by generating new game states with the new boards generated by the actions method, and wrap it into game state objects. It will also compute the utility of these new states by calling compute_utility method

for every one of them and switch the player whose turn it is by going from musketeers to guardsmen and vice versa.

4.4.4. Compute_utility method

The compute_utility method will compute the usefulness of a certain state for both players corresponding to the rules of the game. When the guardsmen are winning in that specific state, i.e., the musketeers have been aligned vertically or horizontally, it will return the value -1. If, on the other hand, it is a terminal state, and they are no more moves possible because they are no adjacent guardsmen to the musketeers, and the musketeers are not aligned orthogonally, it will return the value 1 corresponding to a win for the musketeers. However, if there are still possible moves for the musketeers, and the musketeers are not aligned, it will return the value of 0 as the game is still on, and no one won yet.

4.4.5. Terminal_test method

This method checks whether the game has come to an end or not. From the rules of the game, if there are still moves possible, i.e., there are adjacent musketeers to the guardsmen, and the musketeers are not yet aligned orthogonally, then the game continues; otherwise, it ends. This method uses the computed value of the utility of the board to concludes whether it is a terminal state or not, i.e., if the utility is -1 or 1 return true, else if it is 0 return false.

4.4.6. Utility method

This method returns the utility of the given state to the minimax algorithm. However, it has a special implementation as we will be implementing a single minimax algorithm for both players. The sought value of utility for bought players is different: -1 for guardsmen in order to win, and 1 for musketeers in order to win. Since we want both players to play optimally, and act as maximizers in their turn, we want both of them to maximizes their outcomes, and therefore, we will return minus the value of utility for the guardsmen in order to act as maximizers in their turn and shorten the amount of code written for the minimax alpha beta search algorithm.

4.5. Alpha Beta Minimax Search Player

This method represents the usual implementation of the minimax algorithm with alpha-beta pruning that will be used by both players to maximize their outcome. The algorithm is described in the previous literature review section. This later is possible because the utility of both players is switched to be positive when it is a win for either players, and negative if it represents a loss for any one of them. In addition to that, the implementation for the search of the best action also incorporates enhancements to improve the memory usage and execution time such as memoisation, hashing, board game symmetry checking, and search pruning for the musketeers described in detail in the upcoming enhancements section.

4.6. Challenges

Many challenges have been faced in the implementation process in order to produce a valid and bug free program to solve the Three Musketeers game. Most importantly, two other factors, which are the memory usage and the execution time, have been the source of constant stress as the former has exceed the computational resources available and for the latter was not able to produce the results in a desired time scope. In the first runs of the implementation without enhancement, the program would use all the memory available (16GO) in the first ten hours of computation and move to virtual memory, and it will only be able to compute 10,000 states node per minutes which would represent at least 100 days of computation needed to generate the results. Another challenging fact we encountered was the lack of high performing computational resources that would run 24/7 in a constant manner with a high memory capacity, which forced us to rent an external server from Digitalocean.com with 64 GO of memory. All these challenges have been tackled successfully as described in the following section of enhancements.

4.7. Enhancements

As discussed in the challenges section, the implementation of the minimax algorithm in solving the Three Musketeers game has to be extremely careful in order not to produce false-positive results that may change the outcome of the research. Hence, applying enhancements to the search algorithm went through many testing processes in order to guarantee the validity

of the algorithm as it was necessary to add a tremendous number of enhancements in order to lower the extremely huge computational complexity and the memory usage of the program that have been noticed in earlier stages of the solving process. Among them was the repetition of many board game configurations that are being recomputed, the symmetry of the board, and the pruning process.

4.7.1. Transposition Table (Memoisation)

In exhaustive search algorithms such as the minimax, the same game board configuration might be encountered multiple times in the solving process, called transpositions [12]. However, depth-first-like algorithms do not keep track of the positions analyzed previously, and the game-theoretical value does not change when encountering the same board configuration by the same player throughout all the possible moves of the game. In fact, once a certain game configuration has been proved to have a certain utility for a certain player, it is unnecessary to compute it again. Thus, we can ignore the same subtree if it shows up in different places in the search tree by using memoisation techniques that stores the game-theoretical value for that specific board configuration in transposition tables [13]. This technique is similar to dynamic programming. In figure 11, we can see that there are at least two possible paths for the musketeers to be at the top-left of its initial position, in which the positioning of the remaining elements in the board game did not change.

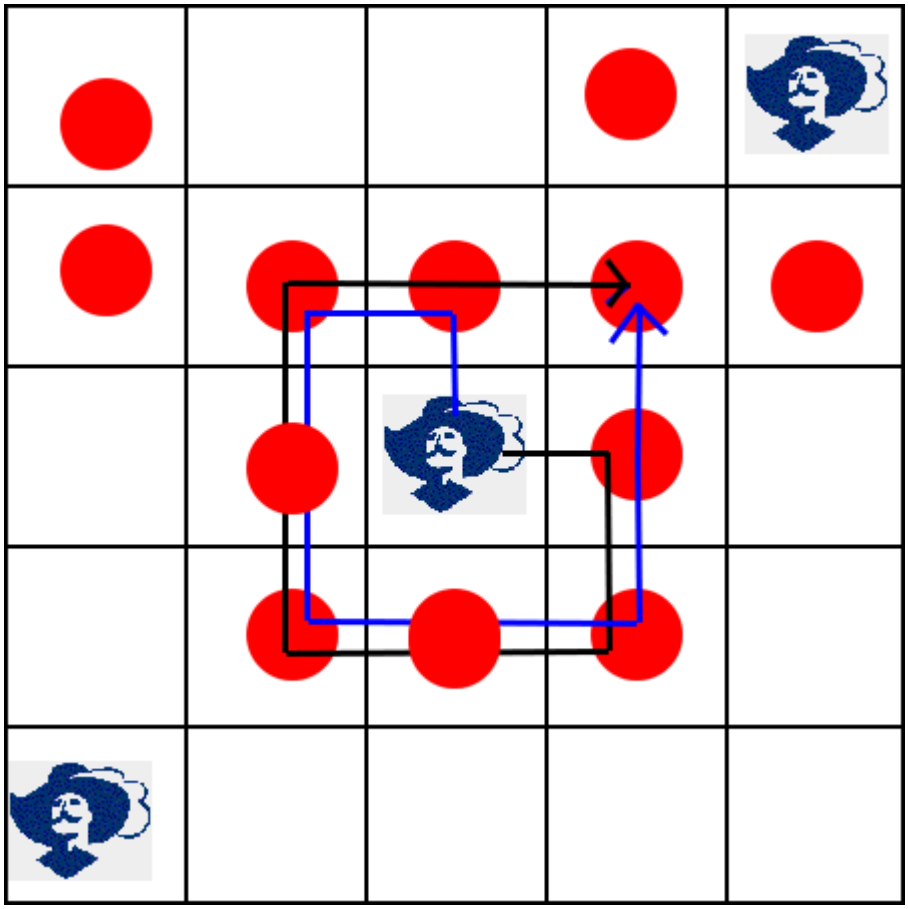

Figure 11: two different move sequences leading to the same state

The memoisation process stores the game-theoretical value corresponding to a specific board generated by the minimax algorithm at every level of the recursion when the search has reached a leaf node from which it can deduce it. This memoisation process is then used in checking whether a certain board configuration exist in a dictionary that stores the already visited sub-trees of the game before going through the minimax search, and returns its value without going through the sub-trees again. Memoisation reduces down greatly the computational process needed to solve the process as shown in the result section. More generally, transposition table can be extremely useful when it uncovered the game theoretical value of a sub-tree with large depth and use it again as it avoids evaluating all the nodes of that sub-tree. Moreover, the memoisation technique further enhance the performance of the program when combined with the symmetry checking of the board game as many of the moves generate the same state but seen from different angles.

4.7.2. Zobrist Hashing

As discussed in the challenges section, memory usage is one of the important challenges encountered in the solving process. the profiling, which we used to analyze the performance of the program when the memoisation technique was added on top of the minimax algorithm, has shown that storing the whole board game as a key for the already seen sub-trees dictionary is highly memory consuming (88 bytes for every board game stored). Therefore, there was a need for reducing such memory consumption especially that remembering the exact position of the different game pieces on the board was unnecessary in such circumstance. Therefore, we added on top of the memoisation technique a hashing mechanism that would serve as the key to the memoisation dictionary. Zobrist Hashing is the best hashing method used in many abstract board games such as chess and Go [11]. It randomly generates bit strings for every pair of a specific position and game piece in the game generating a unique position/piece component that is later hashed with the rest of the positions game using bitwise XOR, which results in a hash collision free algorithm that maps every different board configuration to a different hash value used in our memoisation table. Implementing Zobrist hashing dropped the amount needed for storing the key value of a dictionary from 88 bytes to 38 bytes.

4.7.3. Board Game Symmetry

Another enhancement technique used in our program is the checking of board game symmetry that will result in the same game-theoretical value for different moves. This symmetry can reduce drastically the number of explored nodes in the minimax search. This characteristic is possible because different moves lead to the same board configuration when seen from different angles. In figure 12, we can see that move 1 and move 2 made by the musketeer on top right corner of the board are similar when we see the board from angle 1 and angle 2. Similarly, both moves made by musketeer on the top-left of the board are the same moves done by the musketeer in the bottom-left of the board seen from angle 3 and angle 4. Therefore, this technique combined with memoisation cut-off by at least 4 the branching factor of the game. Similar observation can be made by the musketeer at the center of board in which its 4 moves are all similar from different angles; thus, we can compute the theoretical game value of a single move, and relate the same result to the other 3 moves. As a result, we

can deduce that many unrelated board configurations have the same game-theoretical value if we rotate them as seen in figure 13.

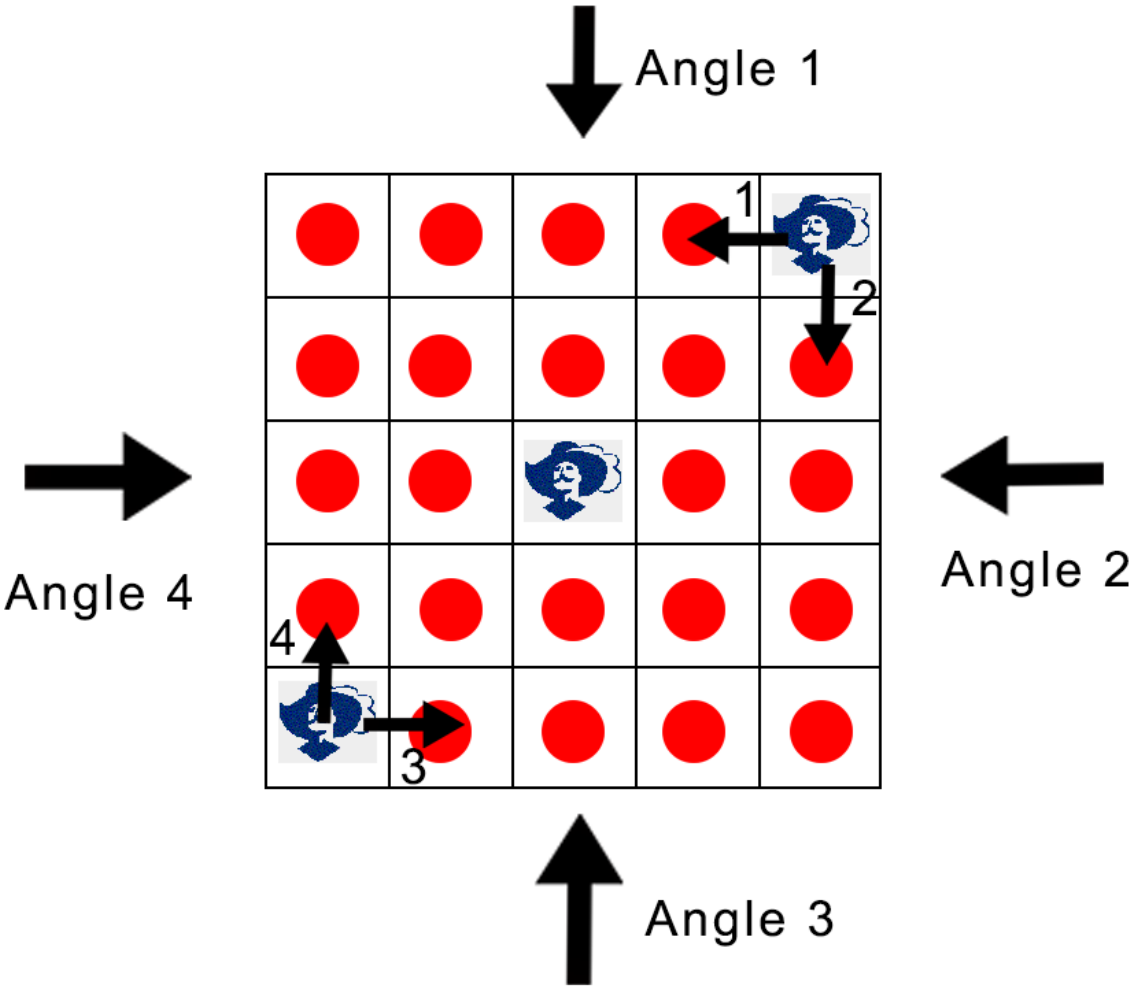

Figure 12: The symmetry of the moves in the Three Musketeers game

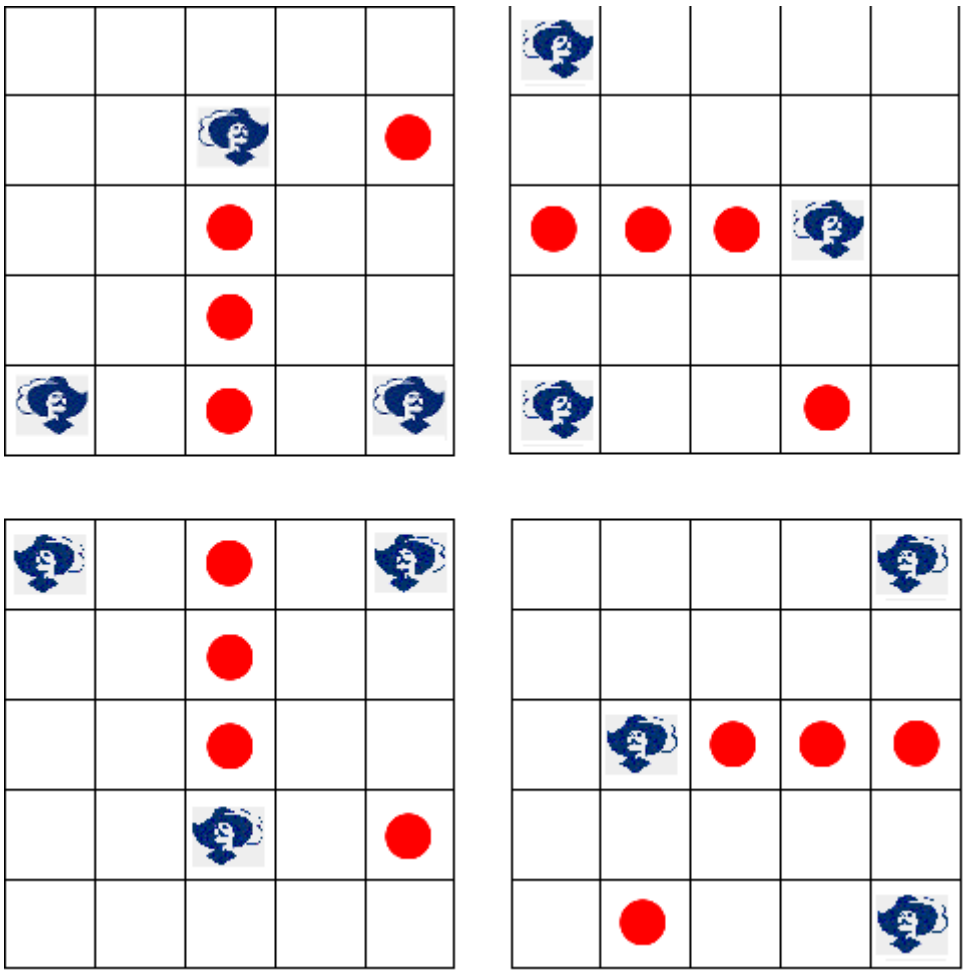

Figure 13: rotation of the same board producing the same game-theoretical value.

The technique used in our implementation is to hash the least lexicographical board of the 4 possible rotations of the game. When encountering for the first time a certain board, we will hash its game theoretical value as if it was produced with a board with the least lexicographical value of its rotations, and therefore, the next encounter of it or a different rotation of it will cut-off the search and return directly the result without going through the minimax algorithm. As a note, the least lexicographical board is the order by which M, G, and an empty cell respectively will occur from the top left of the board to the bottom right of the board sequentially.

4.7.4. Search pruning enhancement

Another improvement made on the minimax search algorithm is the ability for the musketeers when searching for the most optimal game move to recognize that the highest theoretical game value reachable for them is 1. This improvement in the search makes the musketeers quit as soon as it encounters the value 1 in a leaf node, and will not search the rest of the game tree whether it can get a higher value than that. This improvement has shown that the decision process made by the musketeers when it is their turn is extremely fact, and will most of the time only analyze a few paths from the root states node (initial board) to a leaf node (terminal state) before finding an optimal move to play. This enhancement is implemented by adding on top of the condition of alpha beta pruning a condition that checks whether the value found so far in the minimax is 1, and if so returns it immediately as the best move without any further checking. On the other hand, this condition can also be implemented for the guardsmen. However, it won't change the checking process as it will go through the whole game search tree because we expect that the guardsmen will not find a utility of 1 (winning utility) since it knows that if the musketeers play optimally, they have no chance of winning from Johannes Laire's results.

4.7. Testing process

The implementation of the minimax algorithm is extremely error prone when dealing with a game with complex rules. In addition to that, the enhancements made on top of it made the implementation more complicated to debug. To avoid difficulty in the debugging process and foreseeing potential problems, we have implemented a number of safe-guards that followed a strict methodology in applying modification to the program, among them are testing the minimax implementation with an easy game (NIM) and generating test cases with a reduced number of pieces that we know the final outcome if both players play optimally.

4.7.1 NIM Game implementation

The purpose of implementing the NIM game to test the minimax alpha-beta player search was to verify that given any heap combination, it will generate the best next move for a given player. The validity of the best next move is then verified using the nim-sum. The

implementation of the NIM game using our abstract structure of a game is extremely light-weight, and only needs few lines of code to describe the rules of the game. This game was implemented in order to isolate the source of error, and easily pin-point whether the bug was occurring because we made a mistake implementing the game and its rules or in implementing the minimax algorithm with its enhancements.

4.7.2 Small Test Cases

Another safe-guard implemented in the testing process was to initiate a minimax result checking on small board configuration that are only few moves away from the end of the game. These test cases feed a board game for which the resulting game-theoretical value is known. As an example, the test in figure 14 is known to produce a utility of 1 because the Musketeers is able to avoid the alignment by moving one of the musketeer to the right or to the left to kill the guardsmen, and generate a winning terminal state for them in which there are no more moves available. On the other hands, in figure 15, the guardsmen are able to force the win as the musketeer in the bottom of the board cannot escape killing them all while a dummy guardsman on the bottom left move up and down for example until the musketeer is aligned with the two other musketeers in the first row of the board, and thus, generating a theoretical game value representing a win for the guardsmen.

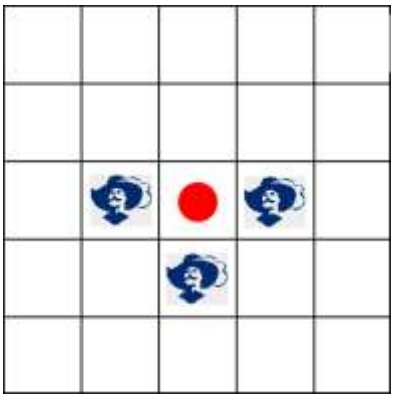

Figure 14: Example of board with musketeers winning

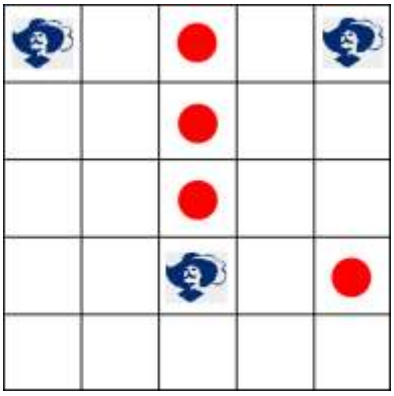

Figure 15: Example of board with guardsmen winning

5. Results

The minimax algorithm has been run to find the best actions for both players, the musketeers and the guardsmen from initial board configuration. It has shown after 32 hours of computation that the musketeers are always winning, and that the game-theoretical value is 1 for the musketeers. Appendix B contains the detailed log of the computation that has led to that conclusion. The logs describe the moves taken by both players alternatively and found that the utility for the musketeers is always 1 (win), and for the guardsmen always -1 (loss) until reaching the terminal state. For the lack of computational resources, the log has been obtained with a less verbose program in order to produce a rapid execution of it. A more verbose version will be available in the second run of the program that will be done before the final submission of the updated capstone report and will include details such as the number of nodes explored, the alpha-beta number of cutoffs, the number of revisited subtrees, and the average branching factor, and maximum depth of the minimax search.

initial state of the board

('G', 'G', 'G', 'G', 'M')

('G', 'G', 'G', 'G', 'G')

('G', 'G', 'M', 'G', 'G')

('G', 'G', 'G', 'G', 'G')

('M', 'G', 'G', 'G', 'G')

---

to_move M utility = 1

('G', 'G', 'G', 'G', ' ')

('G', 'G', 'G', 'G', 'M')

('G', 'G', 'M', 'G', 'G')

('G', 'G', 'G', 'G', 'G')

('M', 'G', 'G', 'G', 'G')

to_move G utility = -1

('G', 'G', 'G', ' ', 'G')

('G', 'G', 'G', 'G', 'M')

('G', 'G', 'M', 'G', 'G')

('G', 'G', 'G', 'G', 'G')

('M', 'G', 'G', 'G', 'G')

….

to_move M utility = 1

(' ', ' ', ' ', ' ', ' ')

('M', ' ', ' ', ' ', ' ')

(' ', ' ', ' ', ' ', ' ')

(' ', 'M', ' ', ' ', 'G')

(' ', ' ', ' ', ' ', 'M')

to_move G utility = -1

(' ', ' ', ' ', ' ', ' ')

('M', ' ', ' ', ' ', ' ')

(' ', ' ', ' ', ' ', 'G')

(' ', 'M', ' ', ' ', ' ')

(' ', ' ', ' ', ' ', 'M')

end state of the board

(' ', ' ', ' ', ' ', ' ')

('M', ' ', ' ', ' ', ' ')

(' ', ' ', ' ', ' ', 'G')

(' ', 'M', ' ', ' ', ' ')

(' ', ' ', ' ', ' ', 'M')

winner is 1

execution time for getting the result: 117239.05204296112060547

6. STEEPLE Analysis

The STEEPLE analysis is an essential component of any capstone project as it reflects the external macro-environmental factors.

6.1. Social

Games, such as the 2-person board games, have long been an important part of our society as a mean for recreation, for example, many of these games have been found in different cultures and in different variations around the world such as Awari and Checkers. The main objective of the capstone research project is not to remove the fun from the game by solving, but by tackling more sophisticated and not easy to memorize games by humans that have larger game trees and develop perfect game strategies that may unveil the weakness of some games such as the first move advantage. This later will help us make games more equitable in case weaknesses have been found in order to improve the games for tournaments as an example.

6.2. Technology

This program solver provides a tool for future quests in solving other gmaes in the field of game theory and artificial intelligence. It also provides a thorough and compact source of information related to solving games.

6.3. Economic

Solving process does not apply only to games, but also to external environments that include two parties maximizing their pay-offs. Solving certain situations result in understanding the distribution of wealth on all the parties taking part of a certain negotiation as an example.

6.4. Environment, Political, Legal

This project does not have any impact on any of these three domains.

6.5. Ethical

Proving that a game is solvable permits a fair distribution of outcomes and pay-offs when it applies, and raise awareness of unfairness when one of the players have an advantage by default.

7. Future work

the purpose of this research was to be able to confirm previously obtained results from Johannes Laine and confirm that indeed the Three Musketeers Game is solvable, and the musketeers will always win if both the musketeers and the guardsmen play optimally. The next step besides performing again the computations and confirming our result, is to be able to strongly solve the Three Musketeers game and provide a winning sequence if exists from any board position and for any player. The found results so far are publishable and we plan to start working on a paper in order to share with the rest of scientific community the strategy that we have used alongside with the challenges that we have encountered in order to be able to solve other more interesting and larger games when the computational power in the future increases. Moreover, we also plan to investigate other variations of the game, for example, playing with on 2 musketeers and/or in larger or smaller board sizes to see whether the outcomes changes given these new parameters.

8. Conclusion

The game-theoretical value of the Three Musketeers game has been proven to be a win for the first player, the musketeers by weakly solving it. The proof has been done by applying minimax algorithm alongside with alpha beta pruning, memoisation, symmetry checking, and other enhancements on the game play of both players. This research has focused on many aspects of the game which lowered the computational complexity and the memory usage. The game has been proven to have many sub-trees occurring many times in the game search tree, and tackled using memoisation. On top of it, Zobric hashing has been used to lower the size of the key used in the transposition table. In addition to that, the symmetry of the game has been applied to lower the number of nodes explored and further benefit from the memoisation

process. Solving popular games using computers have been a fascinating subject over the years because it underlies the potential achievement and speculate on the future advances of the field of artificial intelligence in general. Even though, our research covers only a small set of the great possibilities that computers could achieve, we are proud to provide a small stepping stone into the future even it does not lead that far.

9. References

[1] I. University of California, "Game-Playing & Adversarial Search," [Online]. Available: https://www.ics.uci.edu/~rickl/courses/cs-171/cs171-lecture-slides/cs-171-07a-Games-MiniMax.pdf. [Accessed 12 3 2017].

[2] P. Norvig, Artificial Intelligence: A Modern Approach, Pearson Education, 2010.

[3] Tom Bylander, "Best-Case Analysis of Alpha-Beta Pruning," University of Texas at San Antonio.

[4] "Killer heuristic," [Online]. Available: https://en.wikipedia.org/wiki/Killer_heuristic. [Accessed 4 12 2017].

[5] H. B. a. V. Allis, "Retrograde Analysis," [Online]. Available: ftp://ftp.lyx.org/pub/distributed_systems/amoeba/orca_papers/retrograde/node1.html. [Accessed 04 12 2017].

[6] J. Laire. [Online]. Available: https://github.com/jlaire/3M.

[7] "Three Musketeers (game)," [Online]. Available: https://en.wikipedia.org/wiki/Three_Musketeers_(game). [Accessed 4 12 2017].

[8] "Python Language," [Online]. Available: https://www.python.org/. [Accessed 4 12 2017].

[9] "Artificial Intelligence: A Modern Approach," [Online]. Available: http://aima.cs.berkeley.edu/. [Accessed 04 12 2017].

[10] L. V. Allis, "Searching for Solutions in Games and Articial Intelligence," 1994.

[11] A. L. Zobrist, "A New Hashing Method with Application for Game Playing," 1969.

[12] "Transposition," [Online]. Available: https://en.wikipedia.org/wiki/Transposition_(chess). [Accessed 4 12 2017].

[13] "Transposition table," [Online]. Available:

https://en.wikipedia.org/wiki/Transposition_table. [Accessed 4 12 2017].

[14] "Three Musketeers (game)," [Online]. Available: https://en.wikipedia.org/wiki/Three_Musketeers_(game).

Appendix A: Implementation of the minimax algorithm for the Three Musketeers game

```
class ThreeMusketeers(Game):
    def __init__(self):
        self.Musketeers_positions = []

    def terminal_test(self, state):
        return 0 if state.utility == 0 else 1

    def to_move(self, state):
        return state.to_move

    def result(self, state, move):
        return GameState(to_move=('M' if state.to_move == 'G' else 'G'),
                         utility=self.compute_utility(move, {},state.to_move),
                         board=move, moves={})

    def utility(self, state, player):
        """Return the value to player; 1 for win, -1 for loss, 0 otherwise."""
        return state.utility if player == 'M' else -state.utility
```

```
    def compute_utility(self, board, move, to_move):
        """If 'M' wins with this move, return 1; if 'G' wins return -1; else return 0."""
        #print self.Musketeers_positions
        # same row(Guardmen winning)
        self.Musketeers_positions = []
        for i  in  range(1,6):
            for j in range(1,6):
                if(board[i][j] == 'M'):
                    self.Musketeers_positions.append((i,j))

        if(self.Musketeers_positions[0][0] == self.Musketeers_positions[1][0] and
         self.Musketeers_positions[1][0] == self.Musketeers_positions[2][0]):
            return -1
        # same column (Gardsmen winning)
        if(self.Musketeers_positions[0][1] == self.Musketeers_positions[1][1] and
         self.Musketeers_positions[1][1] == self.Musketeers_positions[2][1]):
            return -1
        # the game can still continue
        for (x,y) in self.Musketeers_positions:
            for (i,j) in Orthogonal_moves:
                    if(board[x+i][y+j] == 'G'):
                        return 0
        return 1
```

```
def actions(self, state):
    List = []
    board = []
    count = 0
    # possible next moves of the M
    if(state.to_move == 'M' and count <= 3):
        for x  in  range(1,6):
            for y in range(1,6):
                if(state.board[x][y] == 'M'):
                    count += 1
                    for (i,j) in Orthogonal_moves:
                            if(state.board[x+i][y+j] == 'G'):
                                board =  [[i for i in row] for row in state.board]
                                board[x][y] = ' '
                                board[x+i][y+j] = 'M'
                                List.append(board)

    # possible next moves of the G
    elif(state.to_move == 'G'):
        for i  in  range(1,6):
            for j in range(1,6):
                if(state.board[i][j] == 'G'):
                    for (x,y) in Orthogonal_moves:
                            if(state.board[i+x][j+y] == ' '):
                                board =  [[i for i in row] for row in state.board]
                                board[i][j] = ' '
                                board[i+x][j+y] = 'G'
                                List.append(board)
    del board
    return List
```

Figure 16: implementation in python of the Three musketeers game

```
ZobristTable = [\
                [[0,0],[0,0],[0,0],[0,0],[0,0]],
                [[0,0],[0,0],[0,0],[0,0],[0,0]],
                [[0,0],[0,0],[0,0],[0,0],[0,0]],
                [[0,0],[0,0],[0,0],[0,0],[0,0]],
                [[0,0],[0,0],[0,0],[0,0],[0,0]]\
                ]
def init_table_zobrist():
    for i in range (0,5):
        for j in range(0,5):
            for k in range(0,2):
                ZobristTable[i][j][k] = randint(0,2**64)
def compute_hash(board):
    h = 0
    for i in range (1,6):
        for j in range(1,6):
            if(board[i][j] != ' '):
                piece = 0 if board[i][j] == 'M' else 1
                h ^= ZobristTable[i-1][j-1][piece]
    return h

def board_piece_comparaison(p1,p2):
    piece_value = {'M': 3 , 'G' : 2, ' ': 1}
    if piece_value[p1] > piece_value[p2] : return 1
    elif piece_value[p1] < piece_value[p2] : return -1
    return 0
```

```
def least_lexicographical_board_hash(board):
    return compute_hash(board)
    least_lexicographical_board = board
    for rep in range(0,3):
        exitFlag = False
        board_rotation = list(zip(*board[::-1]))

        for i in range(1,6):
            for j in range(1,6):
                comparaison_value = board_piece_comparaison(least_lexicographical_board[i][j], board_rotation[i][j])
                if(comparaison_value == 1):
                    exitFlag = True
                    break
                elif(comparaison_value == -1):
                    exitFlag = True
                    least_lexicographical_board = board_rotation
                    break
            if(exitFlag): break
        board = board_rotation
    return compute_hash(least_lexicographical_board)
```

Figure 17: implementation of Zobrist hashing and Symmetry checking

Appendix B: resulting logs of the minimax search and weakly proving

initial state of the board

('G', 'G', 'G', 'G', 'M')

('G', 'G', 'G', 'G', 'G')

('G', 'G', 'M', 'G', 'G')

('G', 'G', 'G', 'G', 'G')

('M', 'G', 'G', 'G', 'G')

---

to_move M utility = 1

('G', 'G', 'G', 'G', ' ')

('G', 'G', 'G', 'G', 'M')

('G', 'G', 'M', 'G', 'G')

('G', 'G', 'G', 'G', 'G')

('M', 'G', 'G', 'G', 'G')

to_move G utility = -1

('G', 'G', 'G', ' ', 'G')

('G', 'G', 'G', 'G', 'M')

('G', 'G', 'M', 'G', 'G')

('G', 'G', 'G', 'G', 'G')

('M', 'G', 'G', 'G', 'G')

to_move M utility = 1

('G', 'G', 'G', ' ', 'G')

('G', 'G', 'G', 'G', ' ')

('G', 'G', 'M', 'G', 'M')

('G', 'G', 'G', 'G', 'G')

('M', 'G', 'G', 'G', 'G')

to_move G utility = -1

('G', 'G', ' ', 'G', 'G')

('G', 'G', 'G', 'G', ' ')

('G', 'G', 'M', 'G', 'M')

('G', 'G', 'G', 'G', 'G')

('M', 'G', 'G', 'G', 'G')

to_move M utility = 1

('G', 'G', ' ', 'G', 'G')

('G', 'G', 'G', 'G', ' ')

('G', 'G', ' ', 'G', 'M')

('G', 'G', 'M', 'G', 'G')

('M', 'G', 'G', 'G', 'G')

to_move G utility = -1

('G', ' ', 'G', 'G', 'G')

('G', 'G', 'G', 'G', ' ')

('G', 'G', ' ', 'G', 'M')

('G', 'G', 'M', 'G', 'G')

('M', 'G', 'G', 'G', 'G')

to_move M utility = 1

('G', ' ', 'G', 'G', 'G')

('G', 'G', 'G', 'G', ' ')

('G', 'G', ' ', 'G', ' ')

('G', 'G', 'M', 'G', 'M')

('M', 'G', 'G', 'G', 'G')

to_move G utility = -1

(' ', 'G', 'G', 'G', 'G')

('G', 'G', 'G', 'G', ' ')

('G', 'G', ' ', 'G', ' ')

('G', 'G', 'M', 'G', 'M')

('M', 'G', 'G', 'G', 'G')

to_move M utility = 1

(' ', 'G', 'G', 'G', 'G')

('G', 'G', 'G', 'G', ' ')

('G', 'G', ' ', 'G', ' ')

('G', 'G', ' ', 'G', 'M')

('M', 'G', 'M', 'G', 'G')

to_move G utility = -1

('G', ' ', 'G', 'G', 'G')

('G', 'G', 'G', 'G', ' ')

('G', 'G', ' ', 'G', ' ')

('G', 'G', ' ', 'G', 'M')

('M', 'G', 'M', 'G', 'G')

to_move M utility = 1

('G', ' ', 'G', 'G', 'G')

('G', 'G', 'G', 'G', ' ')

('G', 'G', ' ', 'G', ' ')

('G', 'G', ' ', 'M', ' ')

('M', 'G', 'M', 'G', 'G')

to_move G utility = -1

(' ', 'G', 'G', 'G', 'G')

('G', 'G', 'G', 'G', ' ')

('G', 'G', ' ', 'G', ' ')

('G', 'G', ' ', 'M', ' ')

('M', 'G', 'M', 'G', 'G')

to_move M utility = 1

(' ', 'G', 'G', 'G', 'G')

('G', 'G', 'G', 'G', ' ')

('G', 'G', ' ', 'M', ' ')

('G', 'G', ' ', ' ', ' ')

('M', 'G', 'M', 'G', 'G')

to_move G utility = -1

('G', ' ', 'G', 'G', 'G')

('G', 'G', 'G', 'G', ' ')

('G', 'G', ' ', 'M', ' ')

('G', 'G', ' ', ' ', ' ')

('M', 'G', 'M', 'G', 'G')

to_move M utility = 1

('G', ' ', 'G', 'G', 'G')

('G', 'G', 'G', 'M', ' ')

('G', 'G', ' ', ' ', ' ')

('G', 'G', ' ', ' ', ' ')

('M', 'G', 'M', 'G', 'G')

to_move G utility = -1

(' ', 'G', 'G', 'G', 'G')

('G', 'G', 'G', 'M', ' ')

('G', 'G', ' ', ' ', ' ')

('G', 'G', ' ', ' ', ' ')

('M', 'G', 'M', 'G', 'G')

to_move M utility = 1

(' ', 'G', 'G', 'M', 'G')

('G', 'G', 'G', ' ', ' ')

('G', 'G', ' ', ' ', ' ')

('G', 'G', ' ', ' ', ' ')

('M', 'G', 'M', 'G', 'G')

to_move G utility = -1

('G', ' ', 'G', 'M', 'G')

('G', 'G', 'G', ' ', ' ')

('G', 'G', ' ', ' ', ' ')

('G', 'G', ' ', ' ', ' ')

('M', 'G', 'M', 'G', 'G')

to_move M utility = 1

('G', ' ', 'M', ' ', 'G')

('G', 'G', 'G', ' ', ' ')

('G', 'G', ' ', ' ', ' ')

('G', 'G', ' ', ' ', ' ')

('M', 'G', 'M', 'G', 'G')

to_move G utility = -1

(' ', 'G', 'M', ' ', 'G')

('G', 'G', 'G', ' ', ' ')

('G', 'G', ' ', ' ', ' ')

('G', 'G', ' ', ' ', ' ')

('M', 'G', 'M', 'G', 'G')

to_move M utility = 1

(' ', 'G', ' ', ' ', 'G')

('G', 'G', 'M', ' ', ' ')

('G', 'G', ' ', ' ', ' ')

('G', 'G', ' ', ' ', ' ')

('M', 'G', 'M', 'G', 'G')

to_move G utility = -1

('G', ' ', ' ', ' ', 'G')

('G', 'G', 'M', ' ', ' ')

('G', 'G', ' ', ' ', ' ')

('G', 'G', ' ', ' ', ' ')

('M', 'G', 'M', 'G', 'G')

to_move M utility = 1

('G', ' ', ' ', ' ', 'G')

('G', 'M', ' ', ' ', ' ')

('G', 'G', ' ', ' ', ' ')

('G', 'G', ' ', ' ', ' ')

('M', 'G', 'M', 'G', 'G')

to_move G utility = -1

(' ', 'G', ' ', ' ', 'G')

('G', 'M', ' ', ' ', ' ')

('G', 'G', ' ', ' ', ' ')

('G', 'G', ' ', ' ', ' ')

('M', 'G', 'M', 'G', 'G')

to_move M utility = 1

(' ', 'G', ' ', ' ', 'G')

('G', ' ', ' ', ' ', ' ')

('G', 'M', ' ', ' ', ' ')

('G', 'G', ' ', ' ', ' ')

('M', 'G', 'M', 'G', 'G')

to_move G utility = -1

(' ', ' ', ' ', ' ', 'G')

('G', 'G', ' ', ' ', ' ')

('G', 'M', ' ', ' ', ' ')

('G', 'G', ' ', ' ', ' ')

('M', 'G', 'M', 'G', 'G')

to_move M utility = 1

(' ', ' ', ' ', ' ', 'G')

('G', 'G', ' ', ' ', ' ')

('G', ' ', ' ', ' ', ' ')

('G', 'M', ' ', ' ', ' ')

('M', 'G', 'M', 'G', 'G')

to_move G utility = -1

(' ', ' ', ' ', ' ', ' ')

('G', 'G', ' ', ' ', 'G')

('G', ' ', ' ', ' ', ' ')

('G', 'M', ' ', ' ', ' ')

('M', 'G', 'M', 'G', 'G')

to_move M utility = 1

(' ', ' ', ' ', ' ', ' ')

('G', 'G', ' ', ' ', 'G')

('G', ' ', ' ', ' ', ' ')

('M', ' ', ' ', ' ', ' ')

('M', 'G', 'M', 'G', 'G')

to_move G utility = -1

('G', ' ', ' ', ' ', ' ')

(' ', 'G', ' ', ' ', 'G')

('G', ' ', ' ', ' ', ' ')

('M', ' ', ' ', ' ', ' ')

('M', 'G', 'M', 'G', 'G')

to_move M utility = 1

('G', ' ', ' ', ' ', ' ')

(' ', 'G', ' ', ' ', 'G')

('M', ' ', ' ', ' ', ' ')

(' ', ' ', ' ', ' ', ' ')

('M', 'G', 'M', 'G', 'G')

to_move G utility = -1

(' ', ' ', ' ', ' ', ' ')

('G', 'G', ' ', ' ', 'G')

('M', ' ', ' ', ' ', ' ')

(' ', ' ', ' ', ' ', ' ')

('M', 'G', 'M', 'G', 'G')

to_move M utility = 1

(' ', ' ', ' ', ' ', ' ')

('M', 'G', ' ', ' ', 'G')

(' ', ' ', ' ', ' ', ' ')

(' ', ' ', ' ', ' ', ' ')

('M', 'G', 'M', 'G', 'G')

to_move G utility = -1

(' ', ' ', ' ', ' ', ' ')

('M', ' ', ' ', ' ', 'G')

(' ', 'G', ' ', ' ', ' ')

(' ', ' ', ' ', ' ', ' ')

('M', 'G', 'M', 'G', 'G')

to_move M utility = 1

(' ', ' ', ' ', ' ', ' ')

('M', ' ', ' ', ' ', 'G')

(' ', 'G', ' ', ' ', ' ')

(' ', ' ', ' ', ' ', ' ')

(' ', 'M', 'M', 'G', 'G')

to_move G utility = -1

(' ', ' ', ' ', ' ', ' ')

('M', ' ', ' ', ' ', ' ')

(' ', 'G', ' ', ' ', 'G')

(' ', ' ', ' ', ' ', ' ')

(' ', 'M', 'M', 'G', 'G')

to_move M utility = 1

(' ', ' ', ' ', ' ', ' ')

('M', ' ', ' ', ' ', ' ')

(' ', 'G', ' ', ' ', 'G')

(' ', ' ', ' ', ' ', ' ')

(' ', 'M', ' ', 'M', 'G')

to_move G utility = -1

(' ', ' ', ' ', ' ', ' ')

('M', ' ', ' ', ' ', ' ')

(' ', ' ', ' ', ' ', 'G')

(' ', 'G', ' ', ' ', ' ')

(' ', 'M', ' ', 'M', 'G')

to_move M utility = 1

(' ', ' ', ' ', ' ', ' ')

('M', ' ', ' ', ' ', ' ')

(' ', ' ', ' ', ' ', 'G')

(' ', 'M', ' ', ' ', ' ')

(' ', ' ', ' ', 'M', 'G')

to_move G utility = -1

(' ', ' ', ' ', ' ', ' ')

('M', ' ', ' ', ' ', ' ')

(' ', ' ', ' ', ' ', ' ')

(' ', 'M', ' ', ' ', 'G')

(' ', ' ', ' ', 'M', 'G')

to_move M utility = 1

(' ', ' ', ' ', ' ', ' ')

('M', ' ', ' ', ' ', ' ')

(' ', ' ', ' ', ' ', ' ')

(' ', 'M', ' ', ' ', 'G')

(' ', ' ', ' ', ' ', 'M')

to_move G utility = -1

(' ', ' ', ' ', ' ', ' ')

('M', ' ', ' ', ' ', ' ')

(' ', ' ', ' ', ' ', 'G')

(' ', 'M', ' ', ' ', ' ')

(' ', ' ', ' ', ' ', 'M')

---

end state of the board

(' ', ' ', ' ', ' ', ' ')

('M', ' ', ' ', ' ', ' ')

(' ', ' ', ' ', ' ', 'G')

(' ', 'M', ' ', ' ', ' ')

(' ', ' ', ' ', ' ', 'M')

winner is 1

execution time for getting the result: 117239.05204296112060547