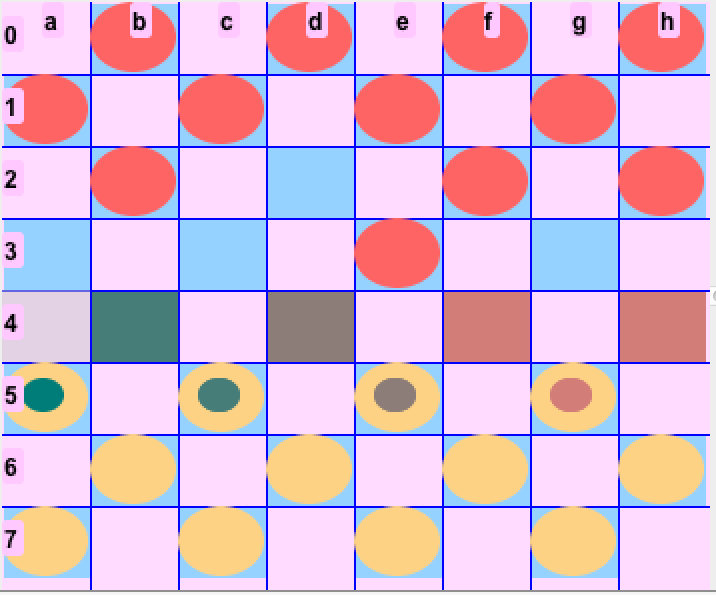
Aditi Joshi, Arun Shrestha, and Catalina Ionescu

Artificial Intelligence: Final Project

Professor Michael Skalak

December 18, 2018

**Implement a game of Checkers using different searching algorithms**

1. **Introduction**
   1. **Checkers**  
       Checker is a strategic board game for two players. The board size can most often be of sizes 8 x 8, 10 x 10 or 12 x 12 and squared in shape. For the purpose of this game, we chose an 8 x 8 board as commonly used in American Checkers. In this game, there are 2 players, each owning 12 pieces. The players take alternating turns. For each turn, a player moves one piece diagonally forward.  A player can either move one step into an empty square, or jump over an opponent's piece, into an empty square, and remove the opponent's piece. A piece that reaches to the opponent’s side of the board becomes a King, gaining the right to move diagonally backward. A player wins when the opponent runs out of pieces or moves.
   2. **Goals** The main goal of the experiment is to analyze and compare the performances of MiniMax, AlphaBeta, NegaScout and Quiescence algorithms in the game of Checkers. To accomplish this goal, several essential factors have been considered including: nodes/second, average branching factor, and effective branching factor. The **nodes/second** an algorithm can generate indicates the speed of the chosen search. The **average branching factor** indicates the number of total possible successors. This could further help a node can have, and the **effective branching factor** of the algorithms are also included in this analysis. The average branching factor indicates the average rate at which the algorithm generates nodes. The effective branching factor indicates the rate at which the algorithm expands those nodes.  Finally, we have included the number of wins an algorithm registers against another algorithm.
   3. **Searching Algorithms** As mentioned above, we implemented 4 different searching algorithms: MiniMax, AlphaBeta Pruning, NegaScout, and Quiescence. Since we discussed Minimax and AlphaBeta Pruning in class and implemented it in second programming assignment: Othello, we are not going to give detail information for it and will move forward discussing the other two.  
      **NegaScout** NegaScout is a search algorithm that improves on AlphaBeta pruning and implements the NegaMax approach. The NegaMax approach is an implementation of MiniMax using the mathematical relation max(a,b) = -min(-a,-b). Rather than using two functions for the evaluation of the minimum and maximum value, NegaMax passes a negated score to execute its calculations. Likewise, NegaScout improves on AlphaBeta pruning by creating a minimal search window which is the alpha and beta values. The basic idea is “that most moves after the first will result in cutoffs, so evaluating them precisely is useless. Instead, it tries to prove them inferior by searching a minimal alpha-beta window first.”[1] Our implementation of the NegaScout is the same as the thesis paper[1] and hence it is crucial to mention that when the node the algorithm is investigating is a superior node, the subtree must be visited to return the precise minimax value[1]. It is also important to note that there are cases in which NegaScout does not perform the precise search i.e. the initial search window is not enough to determine a minimax value. In our implementation, the two cases are when the depth is more than 2 and when the window is not identical to AlphaBeta pruning. This can be observed in the pseudocode attached at *Appendix B* *Figure 1*. Since it is an improved version of alpha-beta pruning, NegaScout also requires a move ordering where the best moves are visited first. We have not implemented this due to the complexity level.  
      **Quiescence**  
       Quiescence search, a searching algorithm is an improved way of evaluating minimax game tree.  As shown in Appendix B: Figure 2, it is implemented by revising the alpha-beta search so that it will only search capturing moves and will stop the search if the current evaluation is already good enough. Typically, a searching algorithm faces horizon problem. This happens when a tree search algorithm reaches a depth limit and chooses an apparently optimal move. That move could hide some consequences below depth limit, that could make the move less optimal.  Quiescence searching algorithm addresses the horizon problem by expanding captured moves deeper than the given limit. This is important as a capturing moved - considered optimal - places the player in spot where it will be easier to capture by the opponent. In such a scenario, the evaluation function will not be able to see the recapture, so the move will be evaluated against you although you were about to regain parity. Hence, quiescence search will protect us from this scenario as it checks for this types of behavior to a deeper level.
2. **Methods** 
   1. **Checkers Rules and game assumptions** For the game, we followed the same rules that we explained in the description of Checkers in section 1.1 above and we added some assumptions to facilitate our experiments.  A player wins when their opponent does not have any piece left on the board or it is left out of moves. The tie occurs when both players are either out of the moves or have less than 3 pieces left on the board. For the purposes of the experiments, we added a condition that resolves in a tie when the game exceed 1 minute in duration and it does not involve a human player.
   2. **Code implementation** The project represents an adaptation of the Othello programming assignment, that now plays Checkers. We created the following classes for the players: *MiniMaxCheckersPlayer,  AlphaBetaCheckersPlayer, NegaScoutCheckersPlayer, QuiesenceCheckersPlayer,* each corresponding to one searching algorithm, as indicated in their name. Additionally, we created a *Move* class to help represent each of the potential moves of a player. The classes *Checkers, Square, CheckersPanel, CheckersPlayer*, and *GameState* have also been modified as needed to adjust for the Checkers game rules and requirements.  **Square Class** The Square class maintains the *col* and *row* fields of Othello, in order to keep track of the board coordinates. Additionally, *Square* contains a field *owner* and *isKing*. The *owner* field indicated whose player the piece will be on the board, and the *isKing* field indicates whether a specific piece can move backward. **Move Class**The move class contains 3 fields: *from*,*to* and *player.* The *playe*r indicates the player executing the move. The *from*, the starting position (the current position of the piece) and the to indicate the position at which the player aims to move the piece. It also contains methods that help calculate the vertical and horizontal distances between *from* and *to*. Additionally, it helps to change the owner of *from* and *to* and modified the king status. The purpose of these methods is to help coordinate the state of the board and keep track of each player's moves. As opposed to Othello, in Checkers a move needs to be applied to a specific piece that is already on the board, instead of placing new pieces on the board. Hence, in Checkers, we need to account for both the origin and the destination of the move.    
      **GameState Class** In the GameState class, the enumerations are taken from the Othello class, meaning: *Direction*, *GameStatus*, *Player*. However, the *Direction* has been modified to include the valid directions in a game of Checkers.At the beginning of the game, the board is initialized as a 2D array of type *Square*. Each Square will have an *EMPTY* owner, and then some specific locations are assigned a new owner, such that the final configuration simulates the board of Checkers.The main point of the class is to help calculate the valid moves, apply them to get the successor and calculate the score and the status of the game. The valid moves are calculated accounting for the vertical and horizontal differences between the *from* and *to* position and also of the status of a piece - whether is a king or not. When applying the moves, the king is also accounted for, as that information is also maintained on the board.Since Checkers allows for one player to jump over multiple pieces in specific situations, the *GameState* class contains a recursive function *applyMoveRecurse* that returns the final state of the board after those consecutive changes have been made. After applying all the valid moves, we get a set of all the possible successor's state.Since the searching algorithms require a static evaluation function, we created a *getScore* method in the *GameState* class, such that each of the algorithms would use the same method to evaluate the board. The goal of this method is to guide the algorithms in making the optimal decision. The aim is to maximize the number of the kings, moves, and pieces a given player has while minimizing the ones its opponent has. Additionally, the opponent has no moves in the given state, the player gains extra 1000 'points' so that the algorithm would be guided to take that route.Eventually, the *GameStatus* defines the end of the game, according to the criteria stated in *Checkers Rules and game assumptions* above. **Checker Class***Checkers* class has maintained a similar utility to the Othello class. For the purposes of the experiment, an additional stopping condition has been added to prevent the game from running indefinitely and facilitate the automation of the experiment. In some specific tie conditions, Checkers runs indefinitely. If none of the players is of type *HumanCheckersPlayer*, the game would forcibly in at most 6000 milliseconds, resolving in a tie. Additionally, Checkers handles a GUI interface, that can be run with the command *java Checkers <Player1> <Player2>* , where *Player1* and *Player2* should be replaced by the names of any of the player classes.  **UI Board Structure**Othello is also played on an 8 x 8 board, hence its UI was appropriate for Checkers as well. The main changes changes were done in the *GameState* and *Checkers* class to coordinate the output of the board accordingly. Designwise, board color is blue and pink, Player 1 pieces are represented by red and Player 2 pieces are represented by yellow. If a piece is a king, then it is marked by a smaller black dot on top. Additionally, when a *HumanCheckersPlayer* takes turn, the player has the ability to drag and drop a piece on the board. Their valid moves will be highlighted with color matched with the piece that can execute that move.
   3. **Experiments**This experiment was conducted using IntelliJ on Mac OS with Intel i5 processor. The subjects analyzed were the 4 implemented algorithms with their corresponding Player: *MiniMaxCheckersPlayer*, *AlphaBetaCheckersPlayer*, *NegaScoutCheckersPlayer* and *QuiesenceCheckersPlayer*. The data collected consisted of the number of nodes generated per second, the approximate average branching factor and the approximate effective average branching factor. A round robin tournament was ran an all the above mentioned classes, each of the games being run 20 times. The average of the data collected was calculated and used further in the analysis.  To reduce potential bias-ness, the player that made the first move was chosen at random. The depth limit was fixed to 3 because of the system and time issues resulted from a higher value.  Each game would run for at most 6000 milliseconds, to avoid tie situations in which the 2 players can run forever.
3. **Results** All the results obtained by the experiment described above has been attached in Appendix A. Based on the average result of the experiment, we got the following results.  **Graph 1**: Winners for each experiments. As we can observe in **Graph 1**, *QuiesenceCheckersPlayer* wins against all the other algorithms, proving to be an optimal AI strategy for this game. This comes as a result of *QuiesenceCheckersPlayer* breaking the horizon problem, by allowing a deeper exploration of the game nodes, even after the the depth limits has been reached. *QuiesenceCheckersPlayer* and *NegaScoutCheckersPlayer* perform very well overall as it has less average branching and effective branching. This indicates that the algorithm doesn’t branch out much, which most of the time act positively.*NegaScoutCheckersPlayer* always loses 100% of the time against the other players . The expected reason behind the poor performance was explained in Limitations *and Issues below*.  However, it is also important to note that this behavior and performance might be credited to other factors as well. One of the reasons might be the horizon problem, which means NegaScout is likely to return a not so good move due to the horizon problem. Also, we used a fixed depth limit of 3 in our experiments, which could have influenced the behavior we observed of NegaScout. Once again, we have suggested further work below regarding this. As we can see above, both *MiniMaxCheckersPlayer* and *AlphaBetaCheckersPlayer* have an average winning rate of 56.67 % and of 61.67% respectively, which implied they have the second and third optimal AI strategy. When we conduct experiment between them, *AlphaBetaCheckersPlayer* and *MiniMaxCheckersPlayer* do not show a significant winning rate difference, with 45% and 35% winning rate respectively. To further explore their performance, we will look at the other computational data that we collected as provided below:

|  |  |  |
| --- | --- | --- |
|  | **AlphaBetaCheckersPlayer** | **MiniMaxCheckersPlayer** |
| **Nodes** | 169205 | 358022.25 |
| **Evaluation** | 133481 | 321930 |
| **Average Branching** | 8.27 | 8.814 |
| **Effective Branching** | 4.14 | 8.814 |
| **Number of Wins (out of 20)** | 9 | 7 |
| **Average time taken (sec)** | 13.3 | 20.2 |
| **Nodes generated per sec** | 12722.18045 | 17723.87376 |

Table1: *AlphaBetaCheckersPlayer* Vs. *MiniMaxCheckersPlayer*   
According to **Table 1,** *MiniMaxCheckersPlayer* has a comparatively higher total number of nodes generated and nodes generated per second than *AlphaBetaCheckersPlayer*. AlphaBeta algorithm is much more efficient than MiniMax algorithm as AlphaBeta, due to pruning.  The effective branching factor of *AlphaBetaCheckersPlayer* is significantly less than the effective branching factor of *MiniMaxCheckersPlayer* because the number of successors generated for each node we are exploring is less due to pruning. Therefore, when we look at the overall performance, *AlphaBetaCheckersPlayer* has a better performance than *MiniMaxCheckersPlayer*. The result we obtained here is similar to what we obtained for ABOthelloPlayer and MMOthelloPlayer while conducting experiments for Othello. Hence, we can conclude that AlphaBeta algorithm performs better than MiniMax.

1. **Limitations and Issues for code implementation**  
    The main trade-off that affects the results of this experiments if depth vs. accuracy. Some experimental results indicated that a depth of 4 could potentially render more accurate results. While at a depth of 3 *NegaScoutCheckersPlayer* has a very low winning rate, when the experiments were run with a depth of 4, *NegaScoutCheckersPlayer* showed a better behavior. However, since a depth of 4 was not efficient enough, often running out of play time, it proved to not be a good choice for the purpose of the experiments.   
    *QuiesenceCheckersPlayer* requires more optimizations. One improvement that has been considered is adding a way of generating the successors of the capturing moves. When this method was implemented, the performance of *QuiesenceCheckersPlayer* did not improve as would have been expected. As the reasons behind this behavior could not be explained, Quiescence was applied to all successor states.This experiment and the result of the algorithms could be dependent on the effectiveness of the evaluation function. The evaluation function was chosen empirically, so it has some room for improvement.
2. **Conclusion**Our experiments resulted in the conclusion that *QuiescenceCheckersPlayer* is optimal for the game of Checkers, both as a winning strategy and based on efficiency.. *NegaScoutCheckersPlayer* performs the most poorly in terms of the number of wins it gathers against other three searching algorithms. However, *NegaScoutCheckersPlayer* performs well when we take the average effective branching factor into account. This suggests that *NegaScoutCheckersPlayer* prunes the search tree better than the other three searching algorithms. *MiniMaxCheckersPlayer* and *AlphaBetaCheckersPlayer* perform similarly which might be credited to the game design and rules of Checkers. Also, the heuristic should have contributed to provide an advantage to *MiniMaxCheckersPlayer* and *AlphaBetaCheckersPlayer* since the heuristic tries to minimize the number of valid moves available to the opponent. However, when we compare the overall performance including the total number of nodes explored, nodes per second explored, average branching factor and effective branching factor, *AlphaBetaCheckersPlayer* performs better than *MiniMaxCheckersPlayer*.

Although the experiment provided us with a reasonable hierarchy for the four algorithms we tested, the experiment was limited to various factors. The algorithms were only allowed to search within the depth limit of 3. Also, the algorithms did not have a deadline to return or make a move. Since Checkers has versions where time limit is a factor, further experiments can be done to see the performance on that implementation. Likewise, we suggest applying Iterative Deepening on NegaScout since it helps to increase the efficiency and performance of NegaScout and compare it again. Move ordering can also be implemented and compared. Also, we could compare other searching algorithms such as SSS\*. Another limitation of our experiment is the heuristic. We applied the same heuristic for all searching algorithms. Hence, there could be experiments conducted where different heuristics are applied to analyze the best heuristic depending on the searching algorithm used.

1. **Bibliography**
   1. **Li**ang, Q. (2003, December). THE EVOLUTION OF MULAN: SOME STUDIES IN GAME-TREE PRUNING AND EVALUATION FUNCTIONS IN THE GAME OF AMAZONS. Retrieved from <http://tuvalu.santafe.edu/~moore/QianThesis.pdf?fbclid=IwAR1LBwwETNBxuR3lEZcxpgdRfo8YCq8DB-Ht0y9PqfLB-mL8Mqo0BIvoYBo>
   2. **I**CS 180, Winter 1999: Strategy and board game programming. (1999, February). Retrieved from <https://www.ics.uci.edu/~eppstein/180a/990204.html?fbclid=IwAR3wBb4EU32P2id61HoEFIctdvq9hNbN5ks4-Jtqo7_V-g27GuJuJbRqc9Q>
2. **Appendix A**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **Player 1: QuiesenceCheckersPlayer** | | | | **Player 2: NegaScoutCheckersPlayer** | | | |  |
|  | **Nodes** | **Evaluation** | **Avg**  **Branching** | **Eff Branching** | **Nodes** | **Evaluation** | **Avg Branching** | **Eff Branching** | **Winner** |
|  | 14997 | 12657 | 10.46237 | 5.56062 | 17986 | 9580 | 7.39252 | 4.86067 | Player 1 |
|  | 24664 | 20367 | 9.65289 | 5.04479 | 36162 | 20126 | 7.82727 | 5.26991 | Player 1 |
|  | 38935 | 31962 | 9.26247 | 4.87785 | 56616 | 31440 | 7.55283 | 5.10486 | Player 1 |
|  | 47470 | 38907 | 9.22465 | 4.84289 | 70015 | 38401 | 7.62275 | 5.09145 | Player 1 |
|  | 56012 | 46128 | 9.27605 | 4.93367 | 80192 | 43890 | 7.54464 | 5.04046 | Player 1 |
|  | 66659 | 55074 | 9.38167 | 5.01309 | 94270 | 51158 | 7.58163 | 5.02678 | Player 1 |
|  | 78272 | 64583 | 9.38518 | 4.99407 | 112663 | 60969 | 7.68716 | 5.07198 | Player 1 |
|  | 90766 | 75113 | 9.39843 | 5.04732 | 126863 | 68547 | 7.60296 | 5.02056 | Player 1 |
|  | 100484 | 83296 | 9.44300 | 5.08445 | 139855 | 75815 | 7.59586 | 5.03074 | Player 1 |
|  | 112080 | 93103 | 9.52488 | 5.13069 | 153419 | 83379 | 7.57587 | 5.03111 | Player 1 |
|  | 123618 | 102983 | 9.63112 | 5.19775 | 165061 | 89770 | 7.51918 | 5.00401 | Player 1 |
|  | 133568 | 110870 | 9.52113 | 5.11735 | 184230 | 100625 | 7.60809 | 5.06806 | Tie |
|  | 142653 | 118519 | 9.55663 | 5.13880 | 195652 | 106548 | 7.59843 | 5.05088 | Player 1 |
|  | 157451 | 130624 | 9.59741 | 5.11520 | 220915 | 120070 | 7.71648 | 5.10755 | Player 1 |
|  | 173959 | 144703 | 9.64317 | 5.17150 | 238185 | 129320 | 7.64467 | 5.06464 | Player 1 |
|  | 184217 | 153274 | 9.65877 | 5.17536 | 252318 | 136880 | 7.64021 | 5.05902 | Player 1 |
|  | 198465 | 165102 | 9.66854 | 5.17699 | 274256 | 147900 | 7.69345 | 5.06326 | Player 1 |
|  | 209971 | 174913 | 9.71009 | 5.20904 | 287023 | 154716 | 7.67298 | 5.05085 | Player 1 |
|  | 222850 | 185732 | 9.76607 | 5.22668 | 305859 | 165006 | 7.71845 | 5.07924 | Player 1 |
|  | 233506 | 194363 | 9.75335 | 5.19861 | 324618 | 175180 | 7.75411 | 5.09963 | Player 1 |
| **Average** | **120529.9** | **100113.7** | **9.57589** | **5.11284** | **166807.9** | **90466** | **7.62748** | **5.05978** |  |

**Table 2: Results of *QuiesenceCheckersPlayer* Vs. *NegaScoutCheckersPlayer***

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **Player 1: MiniMaxCheckersPlayer** | | | | | **Player 2: NegaScoutCheckersPlayer** | | |  |
|  | **Nodes** | **Evaluation** | **Avg**  **Branching** | **Eff Branching** | **Nodes** | **Evaluation** | **Avg Branching** | **Eff Branching** | **Winner** |
|  | 17572 | 15727 | 8.41571 | 8.41571 | 17343 | 9289 | 8.53494 | 5.47343 | Player 1 |
|  | 33052 | 29639 | 8.54719 | 8.54719 | 30684 | 16534 | 8.30869 | 5.38072 | Player 1 |
|  | 60857 | 55070 | 9.30251 | 9.30251 | 50778 | 27817 | 8.36265 | 5.49127 | Player 1 |
|  | 83795 | 75849 | 9.26014 | 9.26014 | 66820 | 36236 | 8.01199 | 5.25516 | Player 1 |
|  | 109062 | 98816 | 9.33111 | 9.33111 | 84872 | 45762 | 7.93122 | 5.18774 | Player 1 |
|  | 130879 | 118647 | 9.36790 | 9.36790 | 100169 | 53883 | 7.87059 | 5.14567 | Player 1 |
|  | 149153 | 135342 | 9.41920 | 9.41920 | 111241 | 59871 | 7.76118 | 5.08965 | Player 1 |
|  | 169884 | 154017 | 9.35588 | 9.35588 | 129819 | 69876 | 7.83553 | 5.12910 | Player 1 |
|  | 202598 | 184122 | 9.60089 | 9.60089 | 154200 | 82690 | 7.94886 | 5.17604 | Player 1 |
|  | 222462 | 201956 | 9.51872 | 9.51872 | 175310 | 93806 | 8.06951 | 5.23079 | Player 1 |
|  | 245372 | 222622 | 9.44574 | 9.44574 | 191735 | 102277 | 7.97069 | 5.16421 | Player 1 |
|  | 264648 | 240179 | 9.47099 | 9.47099 | 206114 | 109290 | 7.95807 | 5.13220 | Player 1 |
|  | 293207 | 266143 | 9.50151 | 9.50151 | 229682 | 122015 | 7.99255 | 5.15868 | Player 1 |
|  | 314061 | 285154 | 9.52768 | 9.52768 | 244051 | 129482 | 7.97188 | 5.14239 | Player 1 |
|  | 334354 | 303364 | 9.46294 | 9.46294 | 261927 | 138960 | 7.97221 | 5.14180 | Player 1 |
|  | 353023 | 320337 | 9.47408 | 9.47408 | 275914 | 147020 | 7.95829 | 5.15307 | Player 1 |
|  | 376233 | 341683 | 9.54930 | 9.54930 | 291294 | 155278 | 7.95385 | 5.15316 | Player 1 |
|  | 394895 | 358758 | 9.57971 | 9.57971 | 302728 | 161004 | 7.92855 | 5.13027 | Player 1 |
|  | 416144 | 378117 | 9.59432 | 9.59432 | 317923 | 169318 | 7.91878 | 5.13104 | Player 1 |
|  | 439037 | 398975 | 9.61114 | 9.61114 | 335759 | 178664 | 7.93569 | 5.13659 | Player 1 |
| **Average** | **230514** | **209225.9** | **9.36683** | **9.36683** | **178918.2** | **95453.6** | **8.00979** | **5.20015** |  |

**Table 3: Results of *MiniMaxCheckersPlayer* Vs. *NegaScoutCheckersPlayer***

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **Player 1: AlphaBetaCheckersPlayer** | | | | **Player 2: NegaScoutCheckersPlayer** | | | |  |
|  | **Nodes** | **Evaluation** | **Avg**  **Branching** | **Eff**  **Branching** | **Nodes** | **Evaluation** | **Avg**  **Branching** | **Eff Branching** | **Winner** |
|  | 11964 | 10119 | 10.01176 | 5.62747 | 15232 | 7845 | 7.61981 | 4.84142 | Player 1 |
|  | 24276 | 20515 | 10.39222 | 5.62075 | 30825 | 16300 | 7.65839 | 4.96919 | Player 1 |
|  | 33557 | 27980 | 9.82185 | 5.26715 | 46255 | 24684 | 7.72720 | 5.03775 | Player 1 |
|  | 43631 | 36501 | 9.78390 | 5.32995 | 58123 | 31164 | 7.60275 | 4.98914 | Player 1 |
|  | 54512 | 45536 | 9.81699 | 5.29809 | 73896 | 39765 | 7.67352 | 5.04341 | Player 1 |
|  | 63487 | 52756 | 9.66726 | 5.17248 | 90682 | 49126 | 7.80530 | 5.14125 | Player 1 |
|  | 72606 | 60373 | 9.63985 | 5.18318 | 103506 | 55849 | 7.80883 | 5.12622 | Player 1 |
|  | 81826 | 68062 | 9.59752 | 5.17493 | 115792 | 62141 | 7.77388 | 5.08486 | Player 1 |
|  | 90410 | 74969 | 9.50051 | 5.10560 | 129805 | 69196 | 7.77974 | 5.05933 | Player 1 |
|  | 104988 | 87359 | 9.62990 | 5.19049 | 148260 | 78927 | 7.79495 | 5.06314 | Player 1 |
|  | 120974 | 100697 | 9.79290 | 5.21845 | 172787 | 93014 | 7.94460 | 5.19137 | Player 1 |
|  | 131981 | 109693 | 9.76712 | 5.18223 | 191390 | 102891 | 7.96794 | 5.19825 | Player 1 |
|  | 140714 | 116915 | 9.72359 | 5.17293 | 203331 | 108982 | 7.93673 | 5.16839 | Player 1 |
|  | 153061 | 126908 | 9.60569 | 5.11687 | 223320 | 119781 | 7.91746 | 5.16046 | Player 1 |
|  | 163648 | 135640 | 9.63092 | 5.10953 | 239534 | 127908 | 7.92844 | 5.14776 | Player 1 |
|  | 177921 | 147624 | 9.63570 | 5.13036 | 257358 | 137506 | 7.88716 | 5.12838 | Player 1 |
|  | 189338 | 157298 | 9.69761 | 5.16358 | 270916 | 144876 | 7.86084 | 5.11836 | Player 1 |
|  | 200280 | 166418 | 9.72012 | 5.16918 | 285848 | 152980 | 7.86399 | 5.12361 | Player 1 |
|  | 213228 | 177275 | 9.71751 | 5.17833 | 301274 | 161316 | 7.81333 | 5.09868 | Player 1 |
|  | 227815 | 189232 | 9.75637 | 5.16306 | 326660 | 174566 | 7.89625 | 5.13513 | Player 1 |
| **Average** | **115010.9** | **95593.5** | **9.74546** | **5.22873** | **164239.7** | **87940.9** | **7.81305** | **5.09130** |  |

**Table 4: Results of *AlphaBetaCheckersPlayer* Vs. *NegaScoutCheckersPlayer***

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **Player 1: QuiesenceCheckersPlayer** | | | | **Player 2: MiniMaxCheckersPlayer** | | | |  |
|  | **Nodes** | **Evaluation** | **Avg**  **Branching** | **Eff**  **Branching** | **Nodes** | **Evaluation** | **Avg Branching** | **Eff Branching** | **Winner** |
|  | 25566 | 20747 | 9.00865 | 4.70655 | 43423 | 38534 | 8.01310 | 8.01310 | Player 1 |
|  | 36740 | 29763 | 8.74158 | 4.61558 | 59150 | 52191 | 7.65795 | 7.65795 | Player 1 |
|  | 47702 | 38553 | 8.73228 | 4.56916 | 77061 | 67921 | 7.60570 | 7.60570 | Player 1 |
|  | 75075 | 60628 | 8.89470 | 4.58529 | 124035 | 109641 | 7.78821 | 7.78821 | Player 2 |
|  | 91947 | 74214 | 8.82462 | 4.56948 | 151184 | 133521 | 7.73715 | 7.73715 | Player 1 |
|  | 113755 | 91804 | 8.79110 | 4.57068 | 186571 | 164778 | 7.73480 | 7.73480 | Player 1 |
|  | 129548 | 105001 | 8.92942 | 4.65531 | 207066 | 182808 | 7.72289 | 7.72289 | Player 1 |
|  | 147532 | 118034 | 8.59172 | 4.44601 | 268662 | 239592 | 8.26958 | 8.26958 | Tie |
|  | 164716 | 131121 | 8.48571 | 4.37446 | 316246 | 282906 | 8.47754 | 8.47754 | Player 2 |
|  | 174865 | 138952 | 8.44706 | 4.34793 | 339583 | 303856 | 8.49021 | 8.49021 | Player 2 |
|  | 185844 | 147502 | 8.42434 | 4.33132 | 363378 | 325264 | 8.51222 | 8.51222 | Player 2 |
|  | 204869 | 163135 | 8.47319 | 4.37483 | 389172 | 347821 | 8.41526 | 8.41526 | Player 1 |
|  | 234564 | 187351 | 8.55446 | 4.42490 | 432158 | 385487 | 8.29510 | 8.29510 | Tie |
|  | 243557 | 194159 | 8.49739 | 4.39229 | 452289 | 403439 | 8.28034 | 8.28034 | Player 2 |
|  | 255926 | 203730 | 8.44720 | 4.36846 | 477372 | 425727 | 8.26146 | 8.26146 | Tie |
|  | 265210 | 211208 | 8.43467 | 4.37128 | 490789 | 437376 | 8.21349 | 8.21349 | Player 1 |
|  | 276504 | 220115 | 8.44079 | 4.36532 | 512573 | 456804 | 8.21958 | 8.21958 | Player 1 |
|  | 288024 | 229046 | 8.43030 | 4.35167 | 541543 | 483076 | 8.27693 | 8.27693 | Player 2 |
|  | 306605 | 243520 | 8.39327 | 4.33364 | 583361 | 520808 | 8.32564 | 8.32564 | Player 1 |
|  | 320587 | 254089 | 8.33975 | 4.30151 | 617424 | 551452 | 8.34334 | 8.34334 | Player 2 |
| **Average** | **179456.8** | **143133.6** | **8.59411** | **4.45278** | **331652** | **295650.1** | **8.13202** | **8.13202** |  |

**Table 5: Results of *QuiesenceCheckersPlayer* Vs. *MiniMaxCheckersPlayer***

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **Player 1: MiniMaxCheckerPlayer** | | | | **Player 2: AlphaBetaCheckersPlayer** | | | |  |
|  | **Nodes** | **Evaluation** | **Avg Branching** | **Eff Branching** | **Nodes** | **Evaluation** | **Avg Branching** | **Eff Branching** | **Winner** |
|  | 15457 | 13621 | 7.60305 | 7.60305 | 10158 | 8177 | 8.72691 | 4.48873 | Player 2 |
|  | 83879 | 77064 | 9.96543 | 9.96543 | 25154 | 18419 | 6.42877 | 3.40979 | Tie |
|  | 105092 | 96068 | 9.65387 | 9.65387 | 35131 | 26239 | 6.91438 | 3.59360 | Player 1 |
|  | 148071 | 134292 | 9.23309 | 9.23309 | 64326 | 50583 | 8.32864 | 4.22975 | Tie |
|  | 186681 | 168239 | 8.80986 | 8.80986 | 89031 | 70502 | 8.46634 | 4.31540 | Tie |
|  | 233078 | 210261 | 8.88255 | 8.88255 | 106133 | 83385 | 8.26804 | 4.20146 | Player 1 |
|  | 273900 | 247185 | 8.94893 | 8.94893 | 123336 | 96805 | 8.33090 | 4.19353 | Player 1 |
|  | 296691 | 267525 | 8.92009 | 8.92009 | 137402 | 108513 | 8.52735 | 4.28671 | Player 2 |
|  | 327811 | 295653 | 8.94778 | 8.94778 | 149929 | 118121 | 8.47522 | 4.25113 | Player 1 |
|  | 350003 | 315161 | 8.82731 | 8.82731 | 163178 | 128684 | 8.44489 | 4.25863 | Player 2 |
|  | 399249 | 359062 | 8.75911 | 8.75911 | 188421 | 148684 | 8.45469 | 4.26639 | Tie |
|  | 429146 | 386096 | 8.77779 | 8.77779 | 199491 | 156946 | 8.37945 | 4.22275 | Player 1 |
|  | 442601 | 397781 | 8.70046 | 8.70046 | 208138 | 163818 | 8.34613 | 4.22212 | Player 2 |
|  | 462854 | 415725 | 8.66963 | 8.66963 | 221481 | 174864 | 8.44498 | 4.26795 | Player 2 |
|  | 491452 | 441309 | 8.65187 | 8.65187 | 234245 | 184657 | 8.39899 | 4.24418 | Player 1 |
|  | 526800 | 473142 | 8.66989 | 8.66989 | 248554 | 195689 | 8.36839 | 4.22596 | Player 1 |
|  | 549784 | 493172 | 8.59158 | 8.59158 | 266898 | 211080 | 8.43767 | 4.28731 | Player 2 |
|  | 567564 | 508925 | 8.57360 | 8.57360 | 278217 | 220337 | 8.49122 | 4.30797 | Player 2 |
|  | 614643 | 550895 | 8.55763 | 8.55763 | 305475 | 242393 | 8.56857 | 4.33957 | Player 2 |
|  | 655689 | 587428 | 8.53728 | 8.53728 | 329417 | 261726 | 8.59998 | 4.35916 | Player 2 |
| **Average** | **358022.3** | **321930.2** | **8.81404** | **8.81404** | **169205.75** | **133481.1** | **8.27007** | **4.19860** |  |

**Table 6: Results of *MiniMaxCheckersPlayer* Vs. *AlphaBetaCheckersPlayer***

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **Player 1: QuiesenceCheckersPlayer** | | | | **Player 2: AlphaBetaCheckersPlayer** | | | |  |
|  | **Nodes** | **Evaluation** | **Avg**  **Branching** | **Eff**  **Branching** | **Nodes** | **Evaluation** | **Avg**  **Branching** | **Eff**  **Branching** | **Winner** |
|  | 14808 | 12118 | 7.90022 | 4.66100 | 10307 | 7611 | 6.48319 | 3.42995 | Player 1 |
|  | 56762 | 48508 | 8.36111 | 5.44271 | 28770 | 20687 | 5.94446 | 3.22823 | Tie |
|  | 68890 | 57877 | 8.31712 | 5.12346 | 45705 | 34805 | 6.97214 | 3.76669 | Player 2 |
|  | 97204 | 81042 | 8.59294 | 5.00484 | 69464 | 53442 | 7.42676 | 3.91192 | Player 1 |
|  | 106444 | 88155 | 8.58541 | 4.89150 | 80932 | 62736 | 7.69265 | 4.00872 | Player 2 |
|  | 123080 | 101104 | 8.50935 | 4.75653 | 99716 | 77776 | 7.85400 | 4.08823 | Player 2 |
|  | 137936 | 113351 | 8.66928 | 4.78579 | 109951 | 85488 | 7.85231 | 4.05155 | Player 1 |
|  | 150925 | 123544 | 8.58404 | 4.72305 | 122395 | 95187 | 7.84886 | 4.04598 | Player 2 |
|  | 163688 | 133522 | 8.57287 | 4.66480 | 135273 | 105222 | 7.92126 | 4.04864 | Player 1 |
|  | 182571 | 148329 | 8.57121 | 4.60875 | 154778 | 120622 | 7.98456 | 4.07171 | Player 2 |
|  | 225512 | 185504 | 8.61500 | 4.80744 | 175174 | 135359 | 7.74387 | 3.96178 | Tie |
|  | 241581 | 198572 | 8.64766 | 4.80089 | 187411 | 144660 | 7.75521 | 3.95015 | Player 1 |
|  | 273502 | 225428 | 8.80098 | 4.87821 | 208498 | 160676 | 7.75221 | 3.93482 | Player 1 |
|  | 283880 | 233413 | 8.76469 | 4.83743 | 222257 | 171964 | 7.84731 | 3.98496 | Player 2 |
|  | 293995 | 240964 | 8.71293 | 4.78219 | 236888 | 183993 | 7.93904 | 4.03351 | Player 2 |
|  | 303138 | 248052 | 8.69739 | 4.75540 | 247003 | 192107 | 7.96525 | 4.05049 | Player 2 |
|  | 321488 | 263190 | 8.72075 | 4.76773 | 260408 | 202389 | 7.94970 | 4.04285 | Player 1 |
|  | 338504 | 276927 | 8.72867 | 4.76015 | 274375 | 213147 | 7.96109 | 4.03730 | Player 1 |
|  | 356867 | 291630 | 8.71091 | 4.74286 | 290380 | 225418 | 7.93966 | 4.02607 | Player 1 |
|  | 377566 | 308780 | 8.73680 | 4.75758 | 304148 | 235737 | 7.91761 | 4.00690 | Player 1 |
| **Average** | **205917.1** | **169000.5** | **8.58997** | **4.82762** | **163191.6** | **126451.3** | **7.63756** | **3.93402** |  |

**Table 7: Results of *QuiesenceCheckersPlayer* Vs. *AlphaBetaCheckersPlayer***

1. **Appendix B  
   Figure 1: Pseudo-code for NegaScout Algorithm [1]  
   Figure 2: Pseudo-code Quiescence Algorithm [2]**

  quiesce(int alpha, int beta) {  
        int score = eval();  
        if (score >= beta) return score;  
        for (each capturing move m) {  
            make move m;  
            score = -quiesce(-beta,-alpha);  
            unmake move m;  
            if (score >= alpha) {  
                alpha = score;  
                if (score >= beta) break;  
            }  
        }  
        return score;  
    }

// pos : current board position

// d: search depth

// alpha: lower bound of expected value of the tree

// beta: upper bound of expected value of the tree

// Search game tree to given depth, and return evaluation of root.  int NegaScout(pos, d, alpha, beta) {

if (d=0 || game is over)

return Eval (pos);

score = - INFINITY; // preset return value

n = beta;

moves = Generate(pos); // generate successor moves 7

for i =1 to sizeof(moves) do { // look over all moves

Make(moves[i]); // execute current move

cur = -NegaScout(pos, d-1, -n, -alpha);

if (cur > score) {

if (n = beta) OR (d <= 2)

score = cur;

else

score = -NegaScout(pos, d-1, -beta, -cur);

}

if (score > alpha) alpha = score; //adjust the search window

Undo(moves[i]); // retract current move

if (alpha >= beta) return alpha; // cut off

n = alpha + 1;

}

return score;

}

1. **Appendix C  
   Implementation of Code:  
   Catalina Ionescu**Main contribution was in restructuring the GameState class such that it function according to the rules of  Checkers instead of Othello. Additional contributions include creating the Move class, restructuring the Square class and creating the *QuiesenceCheckersPlayer*.  
   **Aditi Joshi**Main contribution was in writing the *MiniMaxCheckersPlayer*, *AlphaBetaCheckersPlayer* and simplifying Checkers Class. Additional contributions include conducting and automating the experiments, and collecting the data.   
   **Arun Shrestha**Main contribution was writing the *NegaScoutCheckersPlayer*. Additional contributions include improving the UI.  
   **Paper and Presentation:**For paper, we tried our best to match one’s contribution and focus. At times, the paper contribution was done when someone is working on the coding section, so that we utilize the time properly. The presentation were divided to match each one's contribution and focus.