Pierre Drego, 301301848, CMPT310

Final Project – Reversi/Othello

For the pure Monte-Carlo algorithm I simply played out each game completely and if the AI won, I added a value for winning and did the same for the AI having a draw or losing. The values were (3) for winning, (2) for a draw and (-4) for losing.

For the modified pure Monte-Carlo algorithm I decided to use four different heuristics, one that factors in the value of having control of the corners (cornerPiecesHeuristic), one that values flipping the most pieces possible(flipPiecesHeuristic), one that values relative points compared to the opponent, in other words how many points you have versus the opponent (relativeScoreHeuristic) and finally one that counts how many pieces are adjacent to yours (adjacentHeuristic). These heuristics were determined randomly, except cornerPieces where I noticed the corner was valuable when playing against another AI in Reversi to learn the game. Relative score seemed important and fortunately it is the most valuable heuristic relative to the others.

Some important notes about the various heuristics include, cornerPieceHeruistic relies on the cornerValue variable to determine how many points a given corner piece gets. flipPiecesHeruistic gets points for each flipped piece as such it can get even more points than the winning value if given the chance, setting a possibly risky precedent. relativeScoreHeuristic, can also get more points than the winning value if given the chance,

Only one heuristic stands up against Monte-Carlo alone and that's the relative value heuristic, the others lost significantly in comparison. I am assuming this was due to the heuristics seemingly only focusing on very specific aspects rather than everything, for example, focusing purely on corner pieces rather than the other options. However, to include the heuristics in the final scoring I performed the same heuristics given the possible locations/scores and added a multiplier called (Heuristic multiplier) that multiplied the (total score for a given location by the heuristic multiplier so (totalScore* heuristicWeightMultiplier)). To determine the heuristic multiplier I made the AI play against the regular pure Monte-Carlo algorithm and if it won a match it would multiply the heuristic multiplier by the given change value, so if it won (heuristicWeightMultiplier * (1-changeValue)), if it lost ((heuristicWeightMultiplier / (1-changeValue)). This was done until I received a multiplier that would win often tie against the pure Monte-Carlo algorithm. I then went the opposite direction of the tie to get a decent heuristicWeightMultiplier value. However, due to a bug in my code my test results may have been flawed, so I had to make do with hours versus days worth of data; I included some of that in the excel sheet for viewing labeled as 'invalid data'.

Fortunately, my heuristics allow my program to perform better than the Monte-Carlo algorithm alone. Based on the data it averages around 25% better performance, surprisingly even the algorithms that lose significantly alone are useful when grouped with the other heuristics. This took me by surprise as I expected the adjacentHeuristic(adjacent pieces to a given location) to not be very useful, it was more of a test. However, it proved to be more useful than previously though as it improved the overall performance of the algorithm. However, something that did require further adjustments was the heuristicWeightMultiplier as can be seen from the data it has a profound impact on the efficiency of the algorithm. An incorrect heuristicWeightMultiplier can reduce performance by more than 50%, as such it was carefully chosen from the data which made it obvious which value to pick.

How does RandomMoveHeuristic work, and what does it do? This function determines a random move based on various heuristics, including corner values and the number of pieces flipped. This still determines the move somewhat randomly, but locations with more points are weighed heavier than the other locations. It works by using randValue = rand()%totalScore, so ex: we get the randValue 500 from a total of 1900, so we have locations[a,b,c,d] [100,200,900,700] we just iterate through adding values until currentScore > randValue. So, 100+200+900 then it stops since (currentScore(1100) > randValue(500)) and thus the value 900 at location c is picked. This makes it so that locations with higher scores are more likely to be picked than other locations, but other locations will still be checked.

NumberIterations impact on data, numberIterations represents the number of times we played out an entire game within our Monte-Carlo program. As this value increases, we play out more games entirely; however, this also comes at the cost of time. Moreover, as we increase the number of iterations the effectiveness of the Monte-Carlo algorithm decreases. This could be because we prioritize certain paths rather than every single one, forcing us to redo the same paths we've already looked at again and wasting an iteration doing the same thing as last time. Regardless, the effectiveness of the modified Monte-Carlo algorithm is seen profoundly with smaller iterations as compared to the Monte-Carlo algorithm. With the relative wins going from 1.8 down to 0.7, meaning it would go from winning almost twice as many games to losing more often than winning. However, that means our heuristic does the job of making our program perform faster and more efficient given the time constraint it has.

Overall, this project was successful and very informative. The modified Monte-Carlo algorithm performs as expected, improving efficiency when there is a given time constraint. As a bonus for this assignment I made the entire game scalable, for example rather than an 8x8 board you can choose whatever size you'd like if it is equal length and width(7x7,14x14). The program performs somewhat efficiently in the fact I cannot beat the program even if it is given a small-time frame/numberIterations; however, that may say more about my abilities than the given program. Moreover, the program does its job of providing a challenging yet fun AI to play against, in a fully functional, hopefully bug free environment.

The tables are in the excel sheet provided, since they are too big. Thank you.