The name of our project is Greedy Scrabble AI, for reasons which I will talk about when explaining our heuristic. The members of our team, Team 26, are Anthony Carrasco, Jacob Burton, and Tianyun Wang. Together, we created a demonstrable AI that could generate moves and play a modified version of scrabble against another AI or a human player.

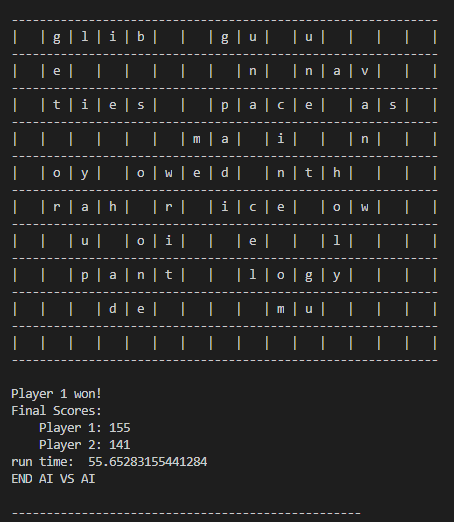
The rules of the game are the rules of scrabble, for the most part. Each player has a rack of seven letter tiles, with each distinct tile having its own value. These tiles must be placed on the board to form words, with the first word able to go anywhere. After that, each word played must be made of the tiles that are in the current player’s rack, must intersect a word already on the board, must be in the dictionary of words, and must not create any nonsense words. The score of each letter in the word that was just played is added up and then added to the current players score. Whenever the player loses tiles through placing words, they must draw more tiles from the bag at the end of their turn. This bag contains a finite number of tiles, and the tiles it contains are predetermined. The rack is randomly refilled from this bag. When the bag is empty, the players must continue with only the tiles in their rack. When a player has no more tiles in their rack, whoever has the most points wins.

We did have to adapt certain parts of the game to be able to create the AI in a limited amount of time. For example, our game does not have multipliers. Normally, there are certain spots on the board that are more beneficial to hit because they give you a score multiplier. We also limited the words that the AI can use to only 4-letter words to make the program run faster. Major challenges for our AI were the complexity of generating moves, which included generating possible words from both the tiles in the current players hand and the tiles already on the board, and then finding a spot to place the newly created word. Different variations in the problem include board size, allowed length of words, and dictionary size.

We started off with a code structure similar to assignment 4, the Connect Four game. When trying to adapt this solution to Scrabble, we had to change every part of it and add quite a few new functions. The following functions were essential parts of our Board. Our “isValid“ function was a lot more complex and checked for different things. This function needed to make sure that an attempted move was within the board, was the first move or intersected another word, was in the dictionary of words, and that the player had the tiles needed to make it. Our “gameOver” function determined the end of the game by two different conditions: when the bag is empty, and a player is out of tiles, or when there are no possible moves left. When the game ended, the winner was not the last player to go, but the player with the highest score. Our “makeMove” function had to get the current words on the board using a function called “updateWords”, and then made the move on the test board, using “isValid” and a few other tests to see if it was a legal move. If it was, then the move was applied, and the current player was changes to the next player. We also had a function for printing out the board, and another for converting the board between a matrix of letters or integers. The board was also where we maintained the bag of tiles to be used, as well as the size and of and all other aspects of the board, and the current player.

The Player contains the heuristic for finding the moves. We started off with minimax algorithm, but we changed it significantly. However, we kept the name the same. Our game is run from a while loop, we each loop generating and making a move for the current player, and the while loop ending when the conditions are met. The move is made by calling the “minimax” function of the current player. The first player is attempting to maximize their score, while the second attempts to minimize its current score. “minimax” first checks the player, which determines whether it is trying to maximize or minimize. It then passes the work onto our “findMoves” function. The “findMoves” function is the meat of the algorithm, as it is what actually generates the moves. This function does this in several steps. First, it takes the highest scoring letter and gets all coordinates on that board that match that letter. Then, it tries all of the words in the dictionary that it can make using the current players tile rack at each coordinate. “findMoves” considers how much free space is around the coordinate, and which words would fit at that point without overlapping any other words or going off the board. It does this for up to 100 generated moves for each coordinate, and then breaks the loop and goes on to find the coordinates and possible words for the next highest scoring letter. Once all of these moves are generated, “minimax” runs a heuristic function on each the moves. The name of this function is “heuristic”. This function takes the move, and then gives it a score. It then generates the moves that player 2 could make if this move was made. “heuristic” then takes the hypothetical player 2 move with the best score and subtracts that score from the score of the player1 move. The result of this is the final score for each move generated for player 1. From this, we should say two things: First, the best move for the current player is based on the next two moves. Second, that our algorithm is called a “Greedy AI” because it wants to win badly enough that it takes the opponents hand into account when making the move. “miniMax” keeps track of which of these moves is the best for its current player based on this score and returns this move. Our initial while loop, which is running all this time, will make the best calculated move for the current player. The Player also contains the rack and functions to manage it, as well as the score dictionary that determines how much each letter is worth. It also contains the function “drawTiles”, which keeps the racks replenished with tiles and depletes the tile bag, and the “removeTiles” function which removes tiles from the rack after use.

The time complexity of our algorithm is something like n^2 as it searches across a grid, but less than that as it only does an operation at tiles that could potential be a good place for a move and the generated move limit is set at 100. It’s also worth mentioning that as more words are added to the board, our algorithm slows down because there are more potential spots for moves/intersecting words. Our AI is somewhat limited by our algorithm. The fact that the AI can take into account the opponents hand means that it should win pretty much every time against a human player, as long as this person follows the same rules as the AI. An alternative that is used by both the commercial scrabble AI Maven, and the open source Quackle is to use a different algorithm for different parts of the game. We felt that given the time, this was a bit beyond our scope. The time that it takes for each move to happen is surprisingly quick, taking usually under a minute to play a full 15x15 board. Below is a chart showing the relationship between the time a move takes, and how many moves have been made. Also below is a picture of a finished game, showing the time that it took to run.



These results were much better than expected. Our algorithm ended up being pretty good for the problem at hand and ran fast enough to give a pretty good demonstration. It was able to handle this sample space just fine, though may have suffered with a longer word length and dictionaries with larger words.

If given time to make improvements, the first thing that we would change is the intersection behavior. As it is now, our algorithm can only do intersections of one letter. Intersections of one letter are easy, but we can’t make a longer word using the entirety of a shorter word, e.g. “that” from “hat”. We also need to address the fact that the AI players can see the other player’s hand. Through this project, we’ve learned that sometime concepts that seem simple to a human can be very complex to teach to a machine, especially when run time is taken into consideration. We also learned that not all algorithms fit all problems, and choosing the right algorithm is important.