THE SEARCH FOR KNIA, THE SCRABBLE 'KNOW-IT-ALL'

JACOB COOPER (JTC267) AND KURT SHUSTER (KLS294)

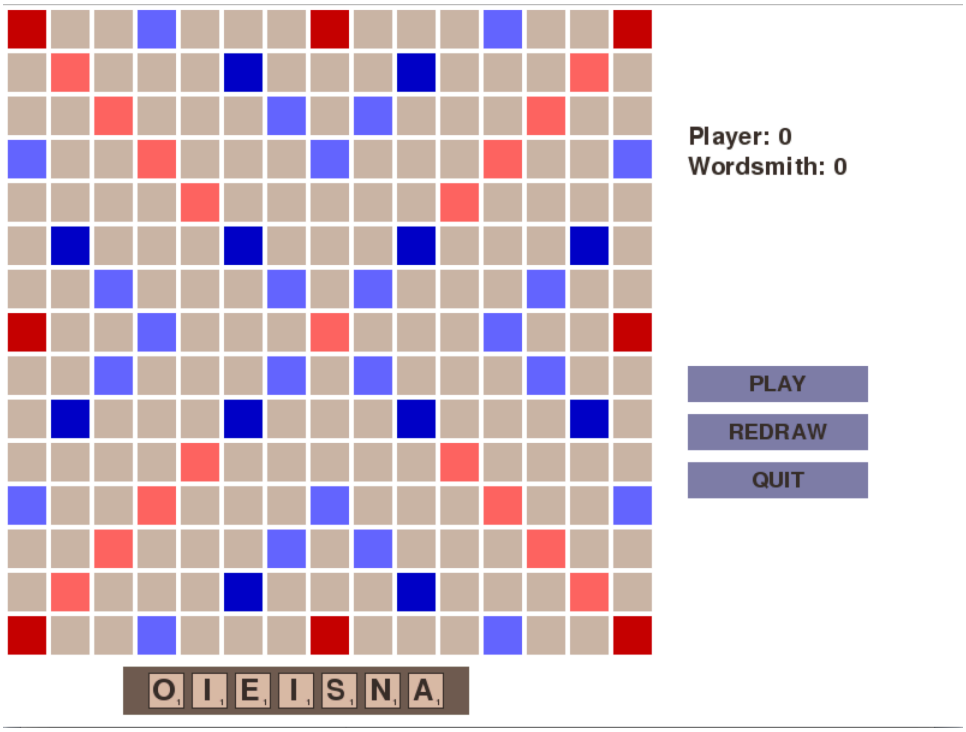
INTRODUCTION

Though viewed by some as a trivial skill, meaningless beyond its specific use, the art of game playing is a microcosm of the skills that humans must utilize to survive and succeed in the real world. In games, we face opponents about whom we may know very little, and perform actions in an effort to both improve our "score" and hinder their progress. In life, we are born into a sea of uncertainty, in which we explore new places and adopt behavior to improve our inherent contentment in living. Thinking and reasoning about games essentially simulates the behavior of rational-minded human beings. Thus, in thinking about an artificially intelligent agent to build for this project, we brainstormed many different ideas for game-playing agents. We sought an environment that suited an agent as the world suits: one that is partially observable, multiagent, stochastic, sequential, dynamic, continuous, and unknown. Focusing on the first four aspects of that list provides a suitable environment in which to train an artificially intelligent to navigate interesting situations.

With this line of thinking, we arrived at the game of Scrabble. Relative to other strategy based games, Scrabble gameplay is easy to understand. Played with at least two participants (note: a multiagent environment), a player's goal is to spell words in a crossword-style fashion and acquire points for doing so. Each opponent has seven randomly selected (note: a stochastic environment) "letter tiles" that are hidden from the other players (note: a partially observable environment). The tiles hold inherent value proportional to the difficulty of playing that tile on a board (i.e. 'E' may have a value of 1 while 'Q' is 10). In addition, the board has some "special squares" that offer multipliers, such as "double word" or "triple letter," that are activated when a tile is placed on that square. Once a word is played, a player draws from the "tile bag" until he/she has seven tiles again. The goal of the game is to continue using one's letters, and subsequently drawing from the "tile bag," to score as many points as possible (note: a sequential environment) while also preventing one's opponents from having good board setups to play their hands. The game ends when the tiles run out, and the player with the most points wins.

Given the environment above, we felt that designing an agent to solve this complex situation would be an interesting and appropriate way to apply concepts from artificial intelligence to a game-playing agent. Below, we will discuss some of the methods and tactics we used when designing KNIA, and then discuss our results with implementing these methods.

SCRABBLE IN PYTHON



WordSmith, Scott Young's open-source Scrabble Program

The backbone of our Scrabble game was inspired by an open-sourced project called "WordSmith," written by a writer/programmer named Scott Young (work cited at the end of the report). The program provided by Young gave us the perfect environment to start implementing advanced AI concepts and techniques to build on the original "player" that he wrote for the program. The provided GUI and the gameplay mechanics allowed us to devote maximum effort to implement KNIA.

In his project, Young included a simple AI player, named WordSmith, that, given its simple evaluation schemes, performed surprisingly well. In fact, WordSmith's algorithm for selecting the best possible word given a game state was helpful in our implementation and offered a good starting point from which we could improve. WordSmith iterates over every possible word that can be made given a player's rack and the board tile positions for a maximum of fifteen seconds, keeping track of the best scored word to that point, after which WordSmith plays this word. The only heuristic WordSmith considered when evaluating a word, besides its inherent point value, was conservation of important tiles for later (e.g. the "blank tile," which can be formatted as any letter). The first thing we did, of course, was play against WordSmith to test the prepackaged intelligence. To our delight, WordSmith proved to be a solid though beatable

opponent, serving as motivation to create the ultimate Scrabble player. It became our goal to improve upon his player so that we could never win. KNIA was in sight.

TACTICS, METHODS, EVALUATION FUNCTIONS

WordSmith's algorithm, though great in practice, lacked the foresight that we desired to program into KNIA. Playing with the intent of maximizing one's gain *at that very moment* is a naïve way to deal with a game in which an opponent can directly respond to one's actions. Young comments in his notes about his program WordSmith's inability to look ahead often leads to its ultimate downfall. For example, WordSmith, by playing what appears to be his best possible move on turn i, might leave open a triple-word square for his opponent to take advantage of on turn i+1.

With this in mind, we set out to apply the same methodology used to solve games like Backgammon, Chess, Poker, and a whole slew of adversarial games: maximizing one's own gain while minimizing the damage done by an opponent, an idea that is powerfully and simply captured in an algorithm known as **minimax**. KNIA, we thought, would construct a game tree, alternating levels between her own move and an opponents, in which KNIA's moves are evaluated based not only on their score at ply i, but also her opponent's possible scores at ply i+1, and her scores at ply i+2, etc. Designing KNIA to play moves based on future consequences, in theory, far surpasses the intelligence and game-playing skills of WordSmith.

Unfortunately, minimax in its true form could not be utilized by KNIA. Unlike chess, Scrabble involves an element of chance (i.e. what letters you might draw from a bag after playing a word) and imperfect information (KNIA doesn't know any opponent's letters). The first challenge we faced in implemented minimax was to incorporate these stochastic processes into our algorithm. We found a solution that integrated different concepts in AI to solve these issues: using **Monte Carlo** in conjunction with a k-local beam search on a minimax tree, essentially mimicking the behavior of **expectiminimax**, salted with a few heuristics along the way to both speed up computation and pick what we thought were better long-term words. Thus, a modified expectiminimax algorithm that implemented ideas from k-local beam search to go three plies deep (two turns by KNIA and one turn by an opponent) in the game tree was chosen to pick KNIA's move, and we decided to add at least two new heuristics to KNIA's word picker to improve upon WordSmith's.

The high level description of the idea of KNIA's move selection algorithm is relatively simple. First, given a board and a set of tiles, KNIA calculates the best M words she can play on that board. Then, KNIA creates M boards (the first ply in the game tree), each the result of

playing a word, and from N random opponent racks generates P words that the opponent can respond with *for each board generated* (for a total of N*P total opponent words). With these M * N * P boards, KNIA finally computes Q random racks and R possible words for each rack that she can respond with *for each of those boards*. This results in a total of M*N*P*Q*R boards for KNIA to consider. The scores are totaled for all of these moves, and KNIA picks the best word at ply 1 that maximizes her score while also minimizing her opponent's.

To speed up the game and simulate real Scrabble play, KNIA had 3 minutes to choose her move. The above description of the algorithm thus highlights a crucial problem in KNIA's evaluation: the time it takes to compute all of these boards. To generate a 3-ply search tree takes O(M*P*R) time per computation. Young's original algorithm gave WordSmith a 15 second timeout, which quickly adds up when attempting to build a game tree of possible moves and subsequent opposing moves. For example, suppose we set M = P = R = 5. To move 3 plies deep in the tree, i.e. to calculate 5 opponent moves *in response to each board state* and then 5 of KNIA's moves in response to those, would take around 5 * 5 * 5 * 15 * 15 + 15= 28140 seconds ≈ 7.8 hours, a level of computation that is simply infeasible given a time constraint of 3 minutes.

Fortunately, it became apparent that the issue could be solved with parallelization of the computation. Each board computation is independent of the next, and feeding each new board to a new process would dramatically improve computational time. Thus, we implemented a pool of processes to handle the computations of words for each board. This dramatically improved our runtime, and allowed computations of many more words than we thought would be possible given our time constraint.

At this point, KNIA's algorithm seemed straightforward to implement: generate good moves for KNIA, then generate an opponent's moves in response to each, and finally generate KNIA's moves in response to those. The final step to complete KNIA was to fully develop a new word picker that would apply to all plies in the tree, by essentially taking WordSmith's picker and modifying it with our own heuristics.

These two additional heuristics were meant to simulate "thinking ahead," that is picking a move based on future consequences. The first evaluates the value of the KNIA's "rack-leave," i.e. the tiles left in her rack after playing a move. Any Scrabble master will be quick to describe that a huge strategic component of the game is "rack management," and this heuristic judges a play not only on how many points would be scored using a specific word, but also on what sort of options it leaves KNIA with left continuing the game with her unused tiles. For example, being left with letters "A, I, E" is much worse than being left with "E, R, S," as you can make many more words with the latter rack. Just as important as rack management is "board management," in which a

player avoids moves that leave good spots for his/her opponent to make very good words. The second heuristic thus evaluates a board after KNIA makes a move, in the hopes of capturing her opponent's possible gain from her move even before the next ply in the game tree is considered.

IMPLEMENTATION

Below, we list in pseudocode the functions we wrote to assist KNIA in her gameplay. To help with understanding some of the terminology found in functions, here is a diagram of a board state as seen by the game engine. The single-quote characters are the blank spaces where there are no words, and the letters indicate tiles that have been placed on the board.

```
'''''''''''''''
'''''''''''''''
'''''''''''''''
'''''''''''''''
''''M''''''''''
''''A''''''''''
''''N''''''''''
'''DIRTLE''''''
''''T''''''''''
''''O''''''''''
''''U''''''''''
'''''''''''''''
'''''''''''''''
'''''''''''''''
'''''''''''''''
```

When selecting a move, WordSmith first considers a "seedPosition", which is just a fancy term for the coordinates of a letter on the board. A seedPosition can be any coordinate from which a player can build words. In the board above, for example, a list of some of the seedPositions might look like [(5, 5), (5, 6), (10,8)] (corresponding to "M", "A", and "E" respectively), etc.

A "tileSlot" is a term the game uses to symbolically represent a feasible list of coordinates in which to place a word *given a seed position*. Suppose, for example, our seed position is (5, 5), where the 'M' is on the board. For the seed position (5,5), the tileSlots would be every horizontal and vertical continuous set of coordinates, of size 2 to 8, that includes M. A large tileSlot for (5, 5) would be [(5, 1), (5, 2), (5, 3), (5, 4), 'M', (5, 6), (5, 7), (5, 8)], while a smaller tileSlot might be [(5,4), 'M', (5,6)]. Vertical sets of coordinates could also be selected using 'M'.

First, a game is started by running "python scrabble.py" in a terminal window.
Next, the game calls `executeTurn` when it is KNIA's turn to move

```
/* Executes a turn for KNIA
 * Params: isFirstTurn, a variable indicating whether this is the first move of the game */
executeTurn (isFirstTurn)
      bestMove := evalKNIA(isFirstTurn) //returns best move
      playMove(KNIA, bestMove) //KNIA then plays that move

/* Evaluates a list of moves, computing a weighted average of the ply results and emphasizing on
 * quality of the first move played
 *
 * Params: moves, a list of tuples of the type (word, KNIAScore, opponentScore)
 * Returns: the best move in moves.*/
findMaxOfBestMoves(moves):
      returns the word with the top average score

/* Computes the top words based on word score, rack leave, and post-move board evaluation
 *
 * Params: rack, a player's rack of tiles
 *        board, a the game board
 *        numWords, the number of words to compute
 *
 * Returns: bestMoves, a list of size numWords and of type (word, rack leave).*/
makeAndEvaluateWords(rack, board, numWords)
      //generates the "seedPositions" or places on the board adjacent to a pre-existing tile.
      seedPositions := generateSeedPositions(board)

      //generates the "tileSlots" or possible combinations of board positions that the player
      //can lay their tiles, described in the diagram above. Note that one "tileSlot" is
      //actually a list of positions for a players tiles.
      tileSlotsList := generateTileSlots(seedPositions)

      //Reorder tileSlots in order of longest (8 tiles) to shortest (2 tile) to evaluate
      //longer tileSlots first
      tileSlotsList := sort(tileSlotsList)

      validMovesList := []

      //for each tileSlot, consider every combination of the tiles in rack that can go in the
      //tileSlot, and check each one to see if it is a word and if the resulting board creates an
      //valid crossword. If both conditions are met, the word is added to validMovesList as a
      //tuple of type (move, boardPosition, tileSlot, score)
      for each tileSlot tS in tileSlotsList:
            for each permutation of tiles pTiles in rack:
                  potentialMove := lay pTiles on board in tileSlot
                  if validWord(potentialMove) and validCrossword(potentialMove):
                        validMoves.apppend(potentialMove, position, tileSlot, score)
                  else:
                        take pTiles off the board in tileSlot



      //evaluates the valid moves based on word score, rack-leave, and resultant board
      bestMoves := findKBestMoves(validMoves, numWords)
      return bestMoves
```

```
/*KNIA's Move Evaluation Algorithm
 * Params: isFirstTurn, a variable indicating whether this is the first move of the game
 *         If this param is true, then the only seed position would be (7, 7).
 * Globally Defined: M, the number of top words for KNIA to make at ply 1
 *                   N, the number of random racks for KNIA's opponent at ply 2
 *                   P, the number of top words for KNIA's opponent to make at ply 2
 *                   Q, the number of random racks for KNIA to make at ply 3
 *                   R, the number of top words for KNIA to make at ply 3
 *                   board, the current game board
 *                   KNIARack, KNIA's player rack
 * Returns: bestWord, KNIA's best move
 */
evalKNIA(isFirstTurn)
   //find the best M words, ranked according to evaluation, and the resulting remaining tiles
   (KNIAPly1BestWords, remainingTiles) := makeAndEvaluateWords(KNIARack, board, M)

   KNIAMoves = [] //the list of ply1 moves from which KNIA will choose

   //generates M new boards and M scores for KNIA given the list of words
   ply1KNIAScores, ply2Boards = generateNewBoards(board, KNIAPly1BestWords)

   opponentRacks = randomRacks(ply2Board, N) //Initializes N random racks

   //Though ply2Boards are evaluated in a loop in this writeup, each is in fact sent off to a
   //subprocess that returns KNIA's board utility for playing each word in KNIAPly1BestWords
   for each ply1KNIAScore in ply1KNIASCores and board ply2Board in ply2Boards:

         ply2KNIAScores = []
         ply2OpScores = []

         for each rack oR in opponentRacks:
               //get the P top moves on board ply2Board for rack oR
               opPly2BestWords, opRemainingTiles = makeAndEvaluateWords(oR, ply2Board, P)

               //create P new boards based off ply2Board that incorporate each new move
               ply3OpScores, ply3Boards = generateNewBoards(ply2Board,opPly2BestWords)

               ply3KNIAScores = []

               for each board ply3Board in ply3Boards:
                     //draw Q new hands from the bag
                     KNIARacks = randomRacks(ply3Board, Q)
                     thirdPlyScores = []
                     for each rack kR in KNIARacks:
                           //find the R best words for rack kR on board ply3Board
                           KNIAPly3BestWords = makeAndEvaluateWords(kR, ply3Board, R)
                           averageKNIAPly3Score = average(KNIAPly3BestWords)
                           thirdPlyScores.append(averageKNIAPly3Score)

                     ply3KNIAScores.append(average(thirdPlyScores))

                ply2OpScores.append(average(ply3opScores))
                ply2KNIAScores.append(average(ply3KNIAScores))

         //After evaluating every board, we return with the average score KNIA receives from a
         //given first-ply move, and append it to the KNIAMoves list
         ply2KNIAAverageScore = average(ply2KNIAScores)
         opAverageScore = average(ply2OpScores)
         KNIAMoves.append(ply2Board, ply1KNIASCORE, ply2KNIAAverageScore, opAverageScore)

   //In implementation, KNIAMoves now has the accumulated result of every subprocess's returned
   //result from computations on a single board.
   bestWord = findMaxOfBestMoves(KNIAMoves)
   return bestWord
```

DESCRIPTION AND ANALYSIS OF IMPLEMENTATION

In `makeAndEvaluateWords()`, KNIA picks the k best words according to word score, rack leave, and board evaluation. This is an example of k-local beam search, which picks the k best successors to generate future moves off of.

In `findMaxOfBestMoves()`, KNIA uses heuristics to calculate which move is actually the best one to use. With our heuristic, a first move that makes a "bingo," Scrabble-speak for using all seven tiles, is usually calculated to be the best move given the heuristics, unless the opponent has very high average scores (if the first move opened up a triple word, for example). This is something that WordSmith would easily overlook and that KNIA is too smart to miss.

In `evalKNIA()`, KNIA uses two levels of Monte Carlo simulation to figure out the relative strength/weakness of her first move.

The first level, generating N opponent hands, is to gain enough samples of opponent moves to sufficiently approximate the relative board position that KNIA leaves her opponent. For example, if KNIA makes a move that opens up a bonus square like a triple word, most if not all of the opponent's hands will likely use it, thus raising the average opponent hand score and lowering the chance that KNIA will pick said move that opens the board up.

The second level of Monte Carlo simulation is necessary for two purposes. In this level, the leftover rack from the first move (all of the tiles that were not used) is combined with Q combinations of randomly drawn tiles from the bag to generate Q random hands. Each hand averages out its R top moves to get the second hand move score. Thus, this level primarily is to counteract the effects that an opponent's move had on the board, e.g. if said move opens the board to a triple word for KNIA. The secondary purpose is to help evaluate the rack-leave of these third-ply moves. For example, a move near the end of game that leaves a "K" in a rack would be considered worse because it is difficult to get rid of uncommon letters in Scrabble's endgame. This dampens KNIA's performance because if the opponent finishes before KNIA, then her leftover rack is *subtracted* from her score and *added* to the opponent's final score. Having a "K" in the middle of the game might not be as bad, and in the beginning it could be quite fortunate to have high scoring letters. Thus, it is necessary to evaluate rack-leaves at all stages in the game.

TESTING AND DEBUGGING

For our first round of testing, we set M = N = P = Q = R=1 and played with a 1-ply game tree search, adding no additional help to the AI. This was a first run-through to test the software out and see how good WordSmith was, to become familiar with what methods were

called when certain actions occurred, and finally to comprehend which files corresponded to which parts of the game, and which object stored the different game states (e.g. the board, the tile bag, the overall score, etc.). We were essentially testing what we were up against.

The next round of testing involved our new and improved word picker, implementing the board evaluation and the rack leave functions. To our delight, the new heuristics did not noticeably slow down the evaluation, and though it is somewhat difficult to test exactly which word picker was better, we at least could qualitatively see that KNIA was picking some pretty good words. We felt confident after running a few games with these new heuristics that KNIA was ready for the next step.

We then increased the game tree to three plies to test our local beam search implementation of expectiminimax. We kept M=1 (i.e. generate only one move at the top the tree) as we were still hesitant to test the time constraints we had set. Note, however, that we were still making R=15 opponent hands in response to KNIA's move, with KNIA countering with Q=1 move in the third ply. Not surprisingly, the algorithm took significantly longer than before, but still finished sooner than expected (most of the time in under a minute). To test what KNIA was considering, we printed out a string representation of the boards (see diagram on page 6) in a terminal window. We were very interested to see that KNIA was in fact considering a sequence of boards in making her decision, and picking words preventing her opponent from making good words.

Without adding in parallelization, we tested a board simulation with M=5 words. The computations were, as expected, extremely slow, and finally completed after 45 minutes. At this point, we really hoped that multiprocessing would save us.

Slotting in a parallelized computation was relatively simple – all we had to do was feed the possible first-ply moves to a pool of python processes, each returning the result of `findMaxOfBestMoves()`. Because these computations could be run asynchronously, the computation took only slightly longer than the computation when M=1 (the increase coming from time spent spawning new processes). This was by far the coolest round of testing, as we saw KNIA considering multiple starting boards, and even selecting moves that, although were not necessarily the shortsighted "best move," were in fact beneficial for later rounds in the game.

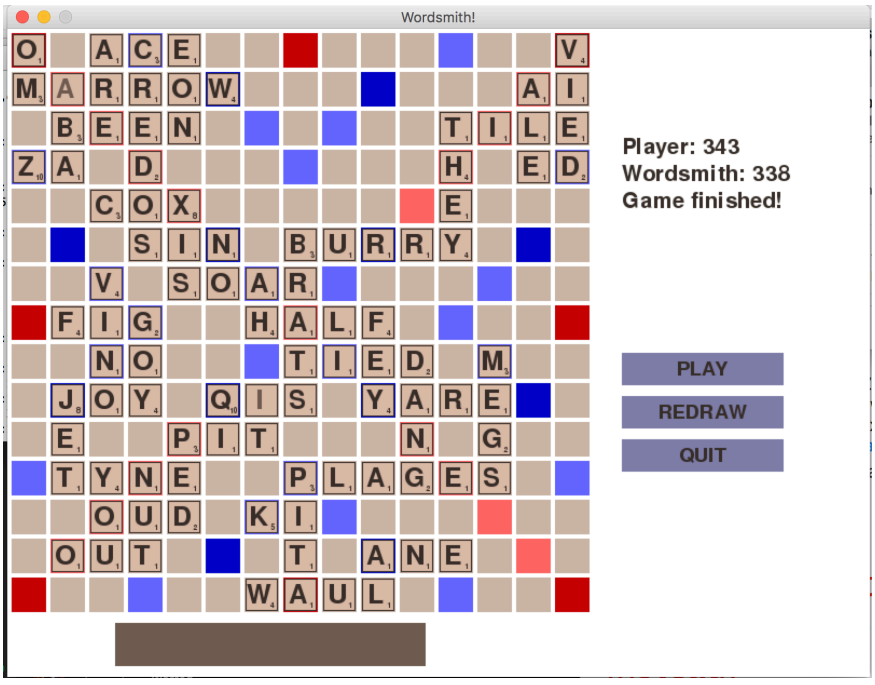
With our initial round of bug searching and functionality-testing complete, it was time to put KNIA up to some real challenges.

PLAYING A SCRABBLE KNOW-IT-ALL

In evaluating KNIA's performance, we iterated through four separate matchups: human vs. WordSmith, human vs. KNIA, KNIA vs. WordSmith, and finally KNIA vs. KNIA.

1) HUMAN VS. WORDSMITH:

As described earlier, we first did a test run of Young's AI WordSmith, by playing against it ourselves. What immediately became clear was that WordSmith's lack of foresight led to a perceptible lack of strategy. Often, WordSmith would make a low scoring word (because its rack was not good) that would open up an opportunity for us to get a high scoring word. These occurrences were nearly evened out by the fact that WordSmith had the entire dictionary at its perusal, and was able to churn out words that used a lot of tiles. Though WordSmith was able to lead most of the game by making the biggest words in its vocabulary from its hand, it often set itself up badly for later turns. This generally happened when it would make a word using the good letters in its rack, e.g. "E,R,S," and leave behind a bunch of vowels, ending with a very poor rack-leave. The result of these poor rack-leaves showed up later when it would have to make lower scoring, vowels-only, words until it could draw more consonants. In the end, we were able to beat Wordsmith on the last move of the game, winning 343 to 338.



Human Vs. Wordsmith

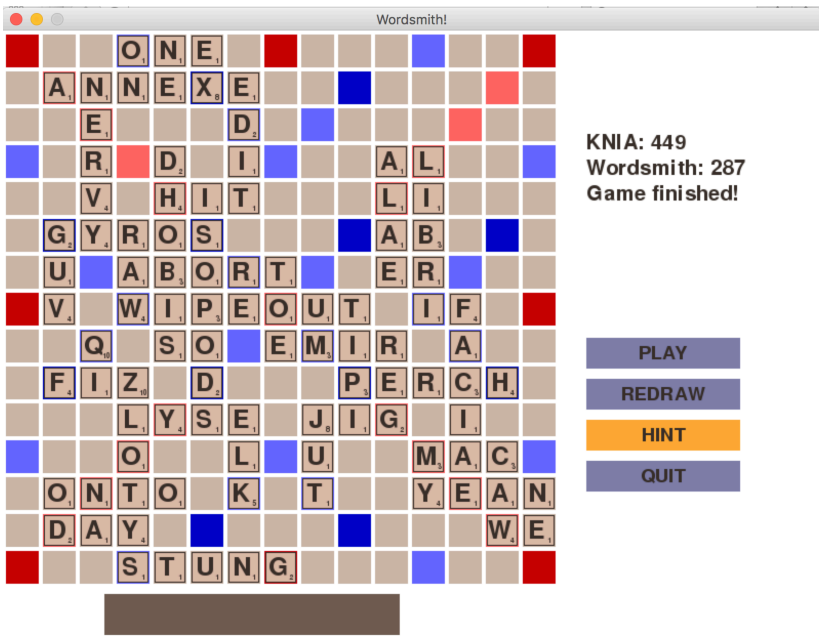
2) HUMAN VS. KNIA:

After we finished implementing KNIA, we decided to play her in a game. KNIA was much more adept at not leaving open as many "bonus squares" for us to use, and midway through the game was already ahead by a sizeable margin. It was much harder to keep up with KNIA, and we even tried using online dictionaries to compensate for the lack of Scrabble words we knew. However, KNIA was too far ahead in the middle of the game. As we had suspected, KNIA, made her moves much slower than WordSmith. This served to her advantage, as she was able to make good words every single move, unlike WordSmith who got stuck with poor hands often because of its rack management.

Human vs. KNIA. Not so good this time

3) KNIA VS. WORDSMITH:

As we had hoped, KNIA was able to beat WordSmith pretty easily. Not only does KNIA have every capability that WordSmith does, but she also has rack-management (what Scrabble pros call the most underrated aspect of the game), ensuring that she was always able to both make offensive moves and almost never get stuck with bad hands. Even though WordSmith procured both blanks in the game, its use of them was ineffective, and KNIA cruised to an even larger victory over WordSmith than over us.



KNIA vs. WordSmith. Annihilation

4) KNIA vs. KNIA:

This game was recorded and sped up for use in our YouTube presentation. It was the longest by far (nearly 45 minutes) and was very close throughout. Each player hovered around relatively low scores as neither player was able to use all of her tiles on any given turn (a distinct lack of so-called "bingos"). What is most apparent in KNIA's play-style was her significantly improved board manipulation– neither player was able to frequently achieve moves with more than one bonus square at a time. Despite this, there was still a lot of competition for the double move slots – just about every double move slot was covered besides the untouched quarter of the board on the top right.

Conclusions and Further Research

We were very pleased with how KNIA turned out. The fact that a mere computer could not only consider an action in the present but also predict its consequences and adjust for that was thrilling to watch unfold. What we also experienced first-hand was the limitation of the gigantic search space on trying to actually "solve" a problem using AI. This really highlighted the importance of good heuristic functions, which motivated us to think about what sort of heuristics in Scrabble would be helpful to reduce the search space and reduce computation.

Thus, our final solution incorporated Monte Carlo, local beam search, word picker heuristics, and parallelized computation to achieve an efficient quasi-implementation of expectiminimax. The possibilities beyond this solution are of course limitless, and if we had to go

back and make it better we would most likely consider a number of different improvements to the software. For example, we would look into the algorithm to iterate through every possible word in order to decrease the computation for any given board. In addition, more heuristic functions could be implemented to select "best words" to play. Finally, more parallelization of asynchronous computations could be implemented to improve evaluation speed.

The ultimate improvement would be to increase the depth of the game search tree to include even deeper analysis of any given starting move. Though limited by time constraints, we were definitely curious to see how far KNIA could go to compute a good move, and what impact this would have on the types of words she chose. This, we felt, was out of the scope of this project, as things such as memory management and improvements to the backend game structure would need to be heavily revised. However, this would be an interesting extension of our work.

Overall, KNIA proved to be an effective AI that exposed us to many of the concepts that are heavily researched in the Artificial Intelligence field. In developing KNIA we have developed the knowledge necessary to tackle other AI problems, and have become familiar with a methodology with which to attack future projects in the subject area. We indeed feel that we have developed a Scrabble Know-it-All, and invite anyone who thinks they can beat her to come try.

Note: Credit to http://www.scotthyoung.com/blog/2013/02/21/wordsmith/