Reproducing A Competitive Scrabble Agent

Casey Pore
Colorado State University
cpore@rams.colostate.edu

Problem Description

In this paper, I describe the process of building a Scrabble agent from scratch and evaluating its potential for competitive play. The unique properties of Scrabble make it difficult to apply existing generalized AI strategies common to other games, such as an adversarial search. In fact, even the best Scrabble agent (Brian Sheppard's Maven) does not utilize any local search techniques until the very end of the game, although it has been shown that the end-game search can be crucial for cinching a win against the best human players (Sheppard 2002b). As we will see, creating a competitive Scrabble player is more about adding features that incrementally improve the agent's chances for higher scoring, such as a combination of move selection heuristics and end-game search, than it is any one over-arching concept of Artificial Intelligence. This sentiment was expressed by Brian Shepard when he said his program, Maven, "is a good example of the 'fundamental engineering' approach to intelligent behavior" (Sheppard 2002b).

Scrabble is neither a game of perfect information nor a zero-sum game. Outcomes are highly dependent on the usefulness of the tiles a player randomly draws from the bag to make high scoring moves. Additionally, it cannot be known what tiles the opponent holds. This makes it difficult (or at least futile) for an agent to plan ahead or anticipate future states of the game from which to make informed decisions about how to play its tiles. This limits the options for quickly generating moves an agent may select to play. To accomplish the goal of generating possible moves quickly, a compact data structure along with an algorithm for identifying valid moves lie at the heart of my implementation, as well as agents that have come before mine. Once this fundamental technique was in place, I worked toward adding heuristics to improve the chances of maximizing the average move score in hopes of drawing my agent closer to being as competitive a player as possible.

After completing the move generation algorithm, the first heuristic implemented was to simply choose the move which produced the the highest score. I found that my implementation, though slightly slower than Appel and Jacobson's player from which I based my implementation, outperformed their player in regards to average move score by a significant margin (Appel and Jacobson 1988). Next, I implemented several heuristics (taking inspiration from a couple different sources) under the hypothesis that each heuristic would add a statistically significant increase to the average move score of my agent and increase its win ratio. An agent was created for each of these heuristics and pitted against an agent using only the basic maximum score heuristic to evaluate their effectiveness. Finally, after evaluating which of these individual heuristics led to a better outcome, I implemented several agents that use different combinations of the heuristics hypothesizing that they would lead to even better outcomes. These multi-heuristic agents were pitted against the maximum score agent to evaluate which combinations worked the best. I found that almost all of the multi-heuristic agents played better than any single-heuristic agent for creating strong Scrabble agent, however, not all of the single-heuristic agents bested the maximum score agent.

Previous Work

Techniques for creating strong AI Scrabble players has been investigated as far back as 1892. The earliest attempt that I encountered was a paper by Shapiro and Smith (S&S) called "A Scrabble Crossword Game Playing Program" (Shapiro and Smith 1982). This first attempt was limited by it's move generation algorithm. Moves were generated by trying permutations of tiles and then using backtracking to check the validity of the attempted move. Move positions were selected by placing words across existing words only. There was no intelligence about forming words with adjacent tiles to make multiple words, however the algorithm still produced moves that scored a respectable 13.125 points on average by selecting either the highest scoring move or the first move it found about some point threshold (Shapiro and Smith 1982). Despite its limitations, this paper laid the foundation for how future programs would achieve a fast search by representing the lexicon as a trie data structure combined with a backtracking algorithm to find valid words.

Though work to create Scrabble agents started over three decades ago, major advancements were sparse and far between. Scrabble as an AI topic is indeed a narrow niche and not popular among academics due to its clear lack of opportunities for improvement and general application. The next milestone in Scrabble agent improvement came in 1988

in a paper by Appel and Jacobson (A&J) that improved the speed over the fastest Scrabble agents at the time by two orders of magnitude (Appel and Jacobson 1988). A&J expanded upon S&S's program in two ways. First, the trie is reduced to a Directed Acyclic Word Graph (DAWG). While this does not directly improve the speed in which moves are generated, it has the advantage of significantly reducing the amount of memory needed to represent the lexicon. This made it small enough to keep the entire lexicon in memory, which did improve speed - an important enhancement considering the limitations of memory at the time, and still important today in mobile applications. Second, they implemented a better backtracking algorithm. S&S's program simply used backtracking to check if a permutation of letters was valid word. A&J's program improves this by considering information about the board to prune invalid moves from the search. This includes pre-computing cross-checks so that no tiles would be selected from the rack unless they were part of a valid cross-word, and using an "anchor" to select where to start placing "left parts" of a word to play[1]. An anchor is simply an open square with a tile to its right that a new word could be connected to.

A&J's program generates moves by using backtracking to place tiles from the rack that form left parts first. Because the cross-checks have been computed and there is a tile to build the left part from, there is never any tile placed from the rack that is not part of *some* word. These two properties reduce the number of moves that need to be placed and evaluated significantly. Once the left part of each move is formed, the algorithm builds the right part of the word in the same way, except accounts for tiles already on the board. The takeaway here is that A&J's algorithm tries to maintain move validity as it goes. It ensures placed tiles are always part of a word, even if it can't complete it, whereas S&S's algorithm tries an entire permutation, and then checks that the move is valid. Like S&S's program, A&J's implementation simply selected the highest scoring move it found.

Finally, the apex in Scrabble agent development seems to have come in the form of Brian Sheppard's Maven. Development of Maven actually started in 1986 before A&J's program was created and was already a strong agent capable of tournament-level play (Schaeffer 2001). However, it immediately switched to using A&J's superior move generation algorithm when Sheppard discovered it. Maven is the accumulation of Sheppard's long history of building a Scrabble agent, and his experience is evident in its multifaceted approach to play. Beyond A&J's fast move generation algorithm, Maven adds several features, many of which are applicable only to corner-cases, but important for catching every last bit of opportunity for selecting best moves. Sheppard has classified these features into 3 categories: rack evaluation, board evaluation, and search.

Rack evaluation is a mechanism by which the tiles left on the rack (known as a leave) for a given move are evaluated based on their utility to produce higher scoring subsequent moves. There are two main techniques used by Maven for rack evaluation. First, Maven maintains a static list of tile combinations, each resolving to a "learned parameter" which adjusts the utility of the move based on the leave. The learned parameters are not a feature per se, but is the process by which Sheppard gathered statistics about what values are optimal to use with different combinations for evaluation. This parameter learning is not part of Maven (it does not employ any machine learning or adjustment of parameters at run-time), but were gathered as the result of "a day's worth of self-play games," tested for their optimality, and applied to the various leave combinations (Sheppard 2002b). Beyond the learned parameters, Maven uses several heuristics (referred to as "extensions" by Sheppard) for rack evaluation. These heuristics include potential usefulness of tiles left in the bag, vowel/consonant balance, and holding onto a "U" tile if the "Q" tile hasn't been played yet.

Board evaluation is the process of using features of the game board to determine the utility of a move. Several board evaluation techniques were considered. For example, it was hypothesized that choosing a good opening move would prevent the opponent from playing on bonus squares. Or that you should always play a word on a bonus square, so that your opponent couldn't take it, even if it meant sacrificing a higher scoring move. In the end, it was found that almost all of the board evaluation techniques examined were irrelevant, except for one - triple word squares. Like the learned parameters for rack evaluation, Maven has a table of parameters to evaluate how many points could be compromised as a function of the placement of an opening move in terms of its location on the board.

Lastly, Maven performs a search during the pre-endgame and endgame. Before the endgame search, Maven performs a pre-endgame search which begins when there are nine tiles left in the bag plus the seven left on the opponents rack. Sheppard explains very little about how the pre-endgame works other than it "is a one-ply search with a static evaluation function that is computed by an oracle," and admits that it may not be useful, except to prevent an opponent from fishing for bingos (using all seven tiles in one move). The endgame search commences when all the tiles have been drawn from the bag, so the remaining tiles on the opponent's rack can be deduced from what's on the board and the game becomes one of perfect information. The endgame search utilizes a B* algorithm that evaluates N-turn sequences of moves and returns intervals that correspond to the risk associated with the sequence, rather than a utility of the board state itself (Sheppard 2002b).

Approach Taken

One prerequisite that I had for this project is that I plan on later porting it to the Android platform, so I preferred to implement it in Java. I had hoped to find a java implementation that I could use as a starting point to add heuristics and endgame search to. I found many implementations in various languages, but only a few in Java. I didn't feel that any of those where implemented in a way that wouldn't require significant refactoring to meet my needs for this project, so I started from scratch. The only code in this project that

[1]Please note that for brevity, I only explain horizontal move generation throughout this paper. Vertical move generation is simply a transpose of these concepts.

wasn't written by me, was the DAWG used to perform word validation. This was leveraged from a Github project called "android-dawg" (icantrap (Github User) 2013).

I referenced A&J's paper as a performance baseline in terms of speed and player strength. Creating a player that quickly and correctly generated moves comparable to A&J was the first (and biggest) milestone for me to reach for this project. Beyond implementing this agent, I drew inspiration from Sheppard and other sources to add features to my agent that could improve its level of competitiveness. These include combinations of Maven's rack evaluation, board evaluation, and extension heuristics, as well as a few heuristics described in a project called "sharpscrabble" (wsanville@gmail.com (Google Code User) 2010).

Move Generation

While A&J's program was my basis for comparing my program to, I did not implement my agent the same way as them. We share many common techniques for move generation, but my implementation ended up being more of a "middle ground" between the techniques used by S&S and A&J. This shows in regards to the speed in which my agent selects its move to play, which ended up being about 2.5 times slower than A&J, but about 10x faster than S&S. Adjusted for modern hardware, S&S and I are probably on equal ground in regards to speed, but I achieve much more in terms of player strength. In this section, I describe the methods used for move generation in my agent and compare and contrast it with S&S and A&J's implementations.

My program uses the second edition of the Official Tournament and Club Word List (TWL06) as it's lexicon (North American Scrabble Players Association 2009). The TWL06 contains 178,691 words and is 1.8MB is size. The data structure used to represent the lexicon in memory is key to generating moves quickly for all Scrabble agents, including mine. As noted earlier, the data structure used is called a DAWG, which maintains the lookup properties of a trie, but within a more compact memory footprint. Thus, the DAWG was chosen for its compactness, not for any efficiency gains. In my case the lexicon was reduced from its initial 1.8MB to only 687.5KB. Both the trie and the DAWG have equivalent lookup times of $O(L)$, where L is the length of the word to be searched. The longest words in TWL06 are 15 letters long, so at most $L = 15$. The DAWG is finite state recognizer that must be built from the lexicon before it can be used. To build and use the DAWG, I used an implementation that I found on Github, called android-dawg (icantrap (Github User) 2013). This is the only code in the project that I did not implement myself.

Because I chose to use android-dawg for storing the lexicon, this limited how my algorithm was able to leverage the board state to identify valid moves. In A&J's program, the anchor, in combination with the pre-computed cross-checks, allowed their algorithm to prune many unnecessary tile combinations from their search. Android-dawg only allows for two functions, to look up a given word, and to find all subwords given a string of characters (in this case, our rack tiles and potentially tiles already placed on the board). Because of this, finding valid moves is more akin to S&S's implementation that tries permutations of tiles from the rack due to the dawg's ignorance of the board state. I do, however, implement all of the cross-checking and other board state awareness features present in A&J's implementation, but at the expense of added time. My algorithm for generating all valid moves during a players turn can be broken down into three phases: identify open positions, try rack permutations, and move validation.

To identify open positions, my program uses the concept of an anchor similar to A&J, but my definition is slightly different and I refer it as a "slot", so as not to overload the term anchor. A&J's anchor is an open space with an adjacent tile on its right which it can connect with to form a word, whereas my slot is the position of a tile on the board that has open spaces either to the west or to the east (or north or south for considering vertical placement). The first step in my algorithm iterates over the entire board to collect a list of these slots to determine where it could make a possible move.

Once all the possible slots have been identified, the algorithm takes the tiles on the rack and generates a list of all possible permutations (without repetition, of course) of the tiles. For a seven tile rack, this list contains 5040 permutations, and takes 35ms on average to create. The high-level algorithm (in Pythonic pseudo-code) for placing placing permutations is as follows:

```
possible_slots = get_possible_slots(
    game_board)
permutations = get_permutations(rack)

for permutation in permutations:
  for slot in possible_slots:
    #places to the right of the slot
    if slot.is_open_east():
      place_permutation_east(
          permutation)
      if placement_ok(permutation):
        move = get_utility(permutation)
        possible_moves.append(move)
    #places to the left of the slot
    if slot.is_open_west():
      place_permutation_west(
          permutation)
      if placement_ok(permutation):
        move = get_utility(permutation)
        possible_moves.append(move)
```

The `place_permutation` functions place the tiles on board relative to each open slot which are then tested for whether it is a valid move. How these permutations are placed is different depending on which side of the slot we are placing them. The `place_permutation_east` function is simple. It places all the tiles in the permutation to the right of the slot, skipping over tiles already on the board. Then it checks that the word formed from the tiles placed from the rack and tiles that already existed on the board is a valid move. The `place_permutation_west` function is easy to understand, but was a bit more difficult to implement. It works the same way as the `place_permutation_east`

function with an additional step. Instead of just placing the permutation once, it must try placing the permutation once for each open tile to the left of the slot, up to the length of the permutation, or if it collides with an existing tile on the board it will terminate at that length.

After a permutation is placed, it is checked to verify whether or not it is a valid move. This includes checks to ensure that the work is in bounds of the board, that all tiles forming cross-words are valid, the formed word is legally connected, etc. If a placement is found to be a valid move, is evaluated by one or more heuristics to calculate its utility and added to a list of possible moves. The extremely over-simplified pseudo-code function for validating a horizontally placed permutation (a `move`) is as follows:

```
#Check that the tiles do not extend
#past board boundaries
if not in_bounds(move):
  return false

#Handles case where a horizontal word
#is placed at the bottom of a vertical
#word (i.e. there are no existing tiles
#on the board in the horizontal word's
#path)
if not is_connected(move):
  return false

#Ensures that the main word formed is
#in the DAWG
if not main_word_ok(move):
  return false

#Does cross-checking to ensure that any
#vertical or adjacent words formed are
#in the DAWG
for tile in move:
  place(tile)
  if has_north_neighbor(tile):
    if not vertical_word_ok(tile):
      return false
  if has_south_neighbor(tile):
    if not vertical_word_ok(tile):
      return false

#Gets the final score for the move,
#including points for adjacent words
get_score(move)

return true
```

Move Selection

The final move to be played is selected from a list of possible moves that are sorted according to the utility assigned to it by various heuristics. The move with the highest utility is selected. The formula for calculating the utility is based on the actual points scored by the move plus or minus a value calculated by a heuristic. My agent doesn't implement the more advanced features of Maven, such a endgame search, but does employ heuristics that are sufficient for besting A&J's program. The heuristics described in this section were inspired by or directly pulled from Maven or sharpscrabble.

Sheppard is protective about the values of the learned parameters of his heuristics, and does not provide many details, so some of parameters used in the heuristics I implemented based off his description may not be optimal. Additionally, Sheppard does not give details about how his heuristics use the learned parameters to calculate the utility of a move and uses cryptic language to say that for a given heuristic, "a linear function implements this concept," and offers little more (Sheppard 2002b). However in his PhD dissertation, he does offer some "Basic" learned parameters I was able to utilize, but still does not provide details about the function to utilize them properly. As such, for many of these functions it was a process of trial and error to determine what worked and what didn't and I was able to achieve positive results (Sheppard 2002a).

The `MaxScore` heuristic is a basic heuristic that is simply the number of points a move would receive if it were to be placed on the board. All Scrabble agents I came across in my research use it. I won't go into much detail about this one, except to say that in my implementation, all other heuristics use it to add or subtract their utility to/from.

`SaveCommon` is the first the five rack evaluation heuristics that take into consideration that it is better to hold onto or play certain tiles because the relative usefulness of these tiles will lead to better scoring words down the road. `SaveCommon` heuristic was sourced from sharpscrabble, and of the five, this one is the most aggressive. This heuristic works by checking the leave for common tiles, which in this case are the letters A, E, I, N, R and S. For each of these letters found in the leave, 5 points are added to the move's score (wsanville@gmail.com (Google Code User) 2010).

`VowelConsonant` is the second rack evaluation heuristic sourced from Maven. Goal here is to keep a good balance of vowels and consonants on the the rack. Maven has a list of values made from its learned parameters for every possible vowel/consonant combination. Sheppard does not provide this information, but does provide a description of a "Basic" version of the learned parameters in his PhD dissertation that was used in early versions of Maven. Below is a table of these "Basic" parameters. For each tile in the leave, these values are added or subtracted from the base score of move (Sheppard 2002a).

Table 1: Basic Vowel-Consonant Balance Table

		Consonants						
		0	1	2	3	4	5	6
Vowels	0	0	0.5	1.5	0	-3.5	-6	-9
	1	-0.5	1.5	1	0.5	-2.5	-5.5	
	2	-2	-0.5	0.5	0	-2		
	3	-3	-2	-0.5	1.5			
	4	-5	-4.5	-0.5	1.5			
	5	-7.5	-7					
	6	-12.5						

The `TileTurnover` heuristic tries to estimate how good a rack leave is based on the potential for drawing high value tiles from the bag to replace the played tiles.

-UseQ

-UWithQUnseen

-UseBonusSquares

what did you do. Describe your project: what AI techniques are used by it, why you picked these techniques, how was the project structured, who did what, what sort of data was supplied (example problems for learning systems, prior models for non-learners), what results were expected. Code samples must be short.

Evaluation and Analysis

How well did your project do: as expected, better or worse. Why did it perform as it did? What worked and what did not? Were there any surprises? What experiments/evaluation did you run? Include your experiment design to this section; it may have a subsection for the experiment design and another for the analysis.

Experiment Design

-define what makes a good player - max score is already very good

-

Experiment Analysis

Future Work, Conclusions

-performance enahncements

-android application

What did you learn? What would you do in addition or differently?

References

Appel, A. W., and Jacobson, G. J. 1988. The world's fastest scrabble program. *Commun. ACM* 31(5):572–578.

icantrap (Github User). 2013. android-dawg. https://github.com/icantrap/android-dawg. Accessed: 2015-05-01.

North American Scrabble Players Association. 2009. Official tournament and club word list. http://www.scrabbleplayers.org/w/Official_Tournament_and_Club_Word_List. Accessed: 2015-05-01.

Schaeffer, J. 2001. A gamut of games. *AI Magazine* 22(3):29.

Shapiro, S. C., and Smith, H. R. 1982. A scrabble crossword game playing program. Technical Report 119, Indiana University.

Sheppard, B. 2002a. *Towards Perfect Play of Scrabble.* Ph.D. Dissertation, Universiteit Maastricht.

Sheppard, B. 2002b. World-championship-caliber scrabble. *Artificial Intelligence* 134(1):241–275.

wsanville@gmail.com (Google Code User). 2010. scharp-scrabble. https://code.google.com/p/sharpscrabble/. Accessed: 2015-05-01.