CS221 Fall 2016 Project Progress: Scrabble AI

Authors: Colleen Josephson {cajoseph} and Rebecca Greene {greenest}

Introduction

The goal of our project is to build an AI that plays Scrabble. In this progress report we describe our model, our preliminary algorithm, present preliminary experimental results, and outline additional work we plan to do before the final deadline.

Model and Algorithms

Extensive vocabulary alone is not sufficient for a competitive scrabble player. If a player optimizes for the best score on every turn they tend to retain tiles that are more difficult to use in play, leading to future racks that will produce a lower score. The best Scrabble players try to maintain their rack in a way that will be conducive to future high-scoring plays.

Computers can be preprogrammed with the entire permissible dictionary, since that dictionary is about 200,000 words long, and the 7 letters on the rack can be combined with those on the board in tens of thousands of different permutations, the search for possible moves becomes a nontrivial undertaking. Most scrabble AIs give themselves a time limit to ensure reasonable progress of play.

Once a list of possible moves has been generated, the AI needs to select the best move as a weighted decision of the score that the move would generate, the opportunities it would provide of the other player, and the effect it would have on the future rack.

Scrabble AIs face the following challenges:

1. **Move generation:** create a list of possible moves from the state of the board, the letters in the rack, and the allowable words in the dictionary. This is a nontrivial search problem.

2. **Rack maintenance:** balance the tradeoff between getting the maximal score for a given turn with maintaining a rack that will be useful for future turns, which usually involves a weighted sum acquired with machine learning plus a number of Monte Carlo simulations.

3. **Adversarial gameplay:** avoid creating opportunities for the other player to place high scoring words

Once the bag has been emptied (all tiles are either on the board or in one of the racks), the game switches to being one with a completely known state. At this point evaluation techniques like minimax become useful to maximize score. This is commonly referred to as *endgame strategy*, and typically only state-of-the-art AIs go into this level of detail.

Scrabble AI

Move Generation

To help solve the search problem, we're using an algorithm created in the 1980s by Andrew Appel and Guy Jacobson. This algorithm remains the backbone of most competitive Scrabble

AIs today. Appel and Jacobson propose restructuring the Scrabble lexicon from a list of words into a trie or prefix tree, where each node is a partial word, the children of a node are words or partial words that can be created using that node (see Fig. 3). All terminal leaves of the trie are words, as are some interim nodes (e.g 'dog' vs. 'dogs'), and the value of a node is a boolean indicating whether the sting is a full word in the dictionary.

To search for moves on the board, the algorithm examines all *anchors*, where an anchor is the space to the left (or above) an existing letter (for horizontal plays), or the space above an existing letter (see Fig. 4). Since a move in Scrabble must attach to an existing word (excepting the first move), this greatly reduces the 15x15 search space. After each new move on the board, the AI should update its list of anchor squares.

Before a potential move is added to the list of generated moves, it needs to be *crosschecked* to ensure that new strings formed in the orthogonal dimension also exist in the dictionary. For example, in Figure 3, when placing DOG underneath CATS, AD and TO are real words, but the vertical SG would fail a crosscheck. Since the crosscheck results for a given tile remain static unless the tiles adjacent to it change, the checks only need only be updated once per move, and only for the squares immediately adjacent to newly placed tiles.

The heart of the algorithm is a backtracking search with constraints that the final result must be a valid word (enforced by the structure of the trie itself), all crosschecks pass, and the new letters placed on the board come from the player's rack. The search algorithm has two recursive parts: ExtendLeft and ExtendRight. Below is pseudocode for the backtracking algorithm to place a horizontal word:

```
ExtendRight (PartialWord, node N, square):
  if square is not empty:
    if the letter l in square is an edge of N (PartialWord + l is a node):
      ExtendRight(PartialWord + l, N.children[l], nextSquare)
  else:
    if PartialWord is a word: LegalMoves.append(PartialWord)
    for each letter l that is in rack, s an edge out of N,in the cross-check set of square:
        remove l from the rack
        ExtendRight(PartialWord + l, N.children[l], nextSquare)
        put tile l back into the rack
```

ExtendLeft places tiles to the left of the anchor point and then calls ExtendRight. 'Limit' is the number of blank tiles between the current anchor point and the preceding anchorpoint (or the end of the board), capped at the rack size of 7.

```
ExtendLeft(PartialWord, node N, square, limit):
  ExtendRight(PartialWord, N, square)
  if limit > 0:
    for each letter l in rack that is an edge of N:
      remove l from the rack
      ExtendLeft(l+PartialWord, N' = N.children[l], nextSquare, limit -1)
      put tile l back into the rack
```

To generate a list of all legal moves for a given board, call LeftExtend("", root node, anchorSquare, Limit) on all anchors. See Figures 1 and 2 in 'Examples and Preliminary Data'

Rack Maintenance and Adversarial Gameplay

The next step of the problem is to try to decide which of the legal moves is best, given consideration of score, maintaining a reasonable rack, and not giving any advantages to the opponent. This is best done through a combination of Monte Carlo simulation and linear predictors with learned weights.

The raw score of a move is as a function of tile values and multipliers on the board, plus any resulting multi-word bonuses or bingos. This is simple to compute, but also a simplified view of the situation. A better metric is the agent/opponent point differential obtained as the result of a move, which is the result of the raw score of the move and also the opportunities it proves the opposing player. This differential is best computed through simulation. For each of the top raw-scoring moves, the AI runs a number of Monte Carlo simulations playing against itself with probable opponent racks. Since the turnover rate of racks is so high, and the computation required per move is rather extensive, most competitive AIs run their models with a search depth ≥ 3. Our preliminary algorithm does not implement this, but we would like to add it for the final product.

Many Scrabble AIs don't try to make any assumptions about the opposing player's rack, and just assign the opponent random unseen letters when running simulations. However, it is possible to use Bayes' algorithm to make a probabilistic model of the tiles a player had on their rack at the start of a turn given the move they made during that turn.For instance, if the opposing player used the letters 'C, T' to attach to an A and make "CAT", it is unlikely that they left an S on their rack, because otherwise they would have played "C,T,S" to make "CATS", which is a higher scoring word. More formally $P(leave|play) = \frac{P(play|leave)P(leave)}{P(play)}$, where P(leave) is the probability that certain tiles were left on the rack. When an AI's Monte Carlo simulations draw from this probability space rather a random assignment, the algorithm has better performance on a level that is statistically significant(Richards and Amir, 2007).

Examples and Preliminary Data

For our initial attempt we successfully implemented the Appel/Jacobson algorithm, but didn't consider any features besides the raw score when selecting a move, and didn't attempt to do any opponent modeling. To evaluate our AI's performance, we had it play an open-source scrabble AI 'Quackle', which has defeated human champions. The Quackle code is in C++ so it is possible to make our AI play the Quackle AI in an automated manner (and indeed necessary to get a large dataset for our final paper). Due to time constraints, the interface between the two AIs hasn't been built yet, so we have a collection of results obtained by manually interfacing the two AIs. The results are presented below in Table 1. We will have automated AI vs AI capability to create a larger dataset for the final result.

Here is an example of the input and output of our AI. On the left is an input board where

CS221 AI	234	249	236	181	235	321	279	**Average Score:**	247.7
Quackle	396	278	264	303	386	441	417	**Average Score:**	355

Table 1: Raw scores and averages for 7 CS221 vs. Quackle games

the Quackle AI has played COMPOTE. Underneath is our AI's rack. On the right is the move our AI chose to make, RAJES. Beneath the figures is an example of the LegalMoves list, over which the AI maximizes.

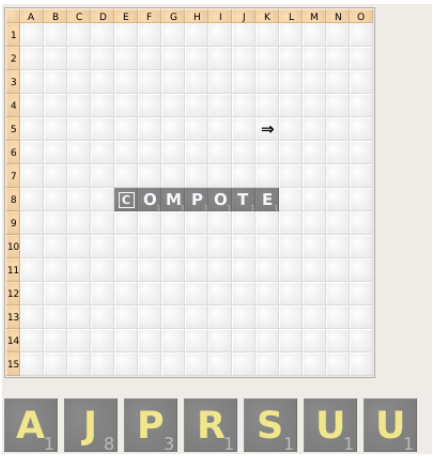

Figure 1: Example of an input board

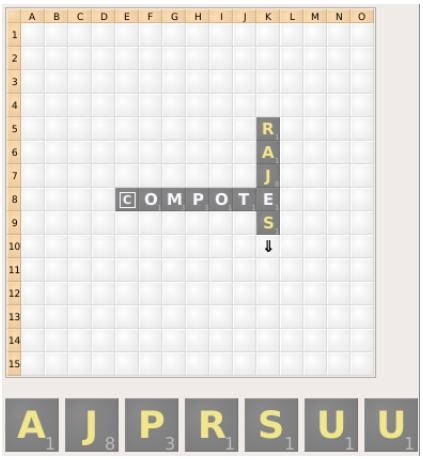

Figure 2: Our AI plays 'RAJES' on the board

```
There are 191 legal moves:  [('JO', (6, 8), 'v', 17), ..., ('RAJES', (4, 10),
'v', 36), ('TRAPS', (7, 9), 'v', 9)])
max word RAJES with score 36
```

Future Work

One of our first tasks moving forward will be to find a way to interface our AI with Quackle so that they can play each other, and we can more easily see the results of our algorithm. We will also be able to use scores against Quackle as a metric for using stochastic gradient descent to create weight vectors for our rack-maintenance heuristics. After that, our goal is to have the AI use depth-2 Monte Carlo simulations to determent the point differential created by a word, instead of using the raw score of the tiles. If we have time we'll create a probabilistic model for the opposing player to make the Monte Carlo simulations more accurate. The probalistic model will cause the Monte Carlo simulations to converge to a minimax model with rack probability = 1 when the endgame state is reached.

Appendix

From Appel-Jacobsen '86

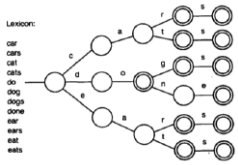

Figure 3: An Example of a Trie

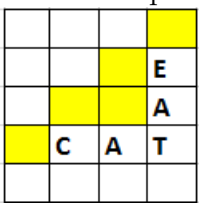

Figure 4: An Example of Anchor Squares

References

A.W. Appel, G.J. Jacobson, The worlds fastest Scrabble program, Comm. ACM 31 (5) (1988) 572578, 585.

Mark Richards and Eyal Amir. Opponent Modeling in Scrabble. IJCAI-07., 14821487, 2007.

Stuart Russell and Peter Norvig. Artificial Intelligence: A Modern Approach. Prentice-Hall, Englewood Cliffs, NJ, 2nd edition edition, 2003

Brian Sheppard. World-championship caliber Scrabble. Artif. Intell., 134:241245, 2002.