Scrabble™ Documentation

Frank Enos • fae2002 • frank@cs.columbia.edu

March 28, 2004

Abstract

The Scrabble™ problem, although implemented exhaustively by many people (indeed, one could base one's entire solution on algorithms found on the web), still offers room for creativity, particularly with regard to approaches to fast search and heuristic search. We attempt here to describe novel approaches to some of the subproblems, and to use features of Java that are particularly helpful.

1 Introduction

As has been observed in the homework specification, there are two general problems to be solved in the assigment: (1) the search for valid words given the current hand and board configuration and (2) the search for the optimal move given the context of the game, including the configuration of the board and the opponent's hand. Conveniently, problem (2) is partly addressed by a second application of (1) from the perspective of the opponent, followed by a fairly straightforward application of the mini-max algorithm – to two plies in most cases and four plies in the endgame when the likely (or certain) hands are easiest to compute. My code has the option to use a probabilistic estimate of the likely next hand for 3rd and 4th ply searches of the mini-max tree by simply generating a possible hand based on the current bag.

2 Unit tests

Please note that the code includes extensive unit tests, built with JUnit for many components of the system. These were used to test functionality and to develop heuristics. Please run them to see how they work, using the instructions in the how to run file.

3 Subproblems

Since there are ample methods for solving these problems in the ”public domain”, we have dedicated most of our time thus far in designing a novel approach, particularly to the searches in Section 3.1 (details below). We present pseudo code and descriptions of these algorithms here.

3.1 Word Search / Move Search

There are several subproblems to address here: (1) the problem of representing the dictionary in a way that is compatible with a fast search given the current hand being analyzed; (2) the problem of determining heuristically which subsets of the hand and available letters from the board to search against in (1), and (3) the problem of generating the subsets of a given hand efficiently. Efficiency at each of these phases of course translates directly into covering more of the search space on a given move.

3.1.1 Computing Subsets

The greatest danger here is to waste time computing the subsets from scratch again for each hand. To avoid this problem, we use simplified permutation matrices (simplified in the sense that order of the letters doesn't matter at this phase - or in the search, as we shall see). Consider that for a given hand (not counting the square on the board), we have $2^7 - 1$ (we ignore the empty subset) possible subsets (because 2^n is the row sum over binomial coefficients). With this information in hand, we devise the following algorithm for computing subsets of a hand:

1. At init time, we create permutation matrices, represented as an array of $2^7 - 1$ 7-place bitmaps, or boolean arrays, one for each (binary) number from $(1...2^7)$.
2. For each (alpha sorted) hand, we iterate (from 2^n downward, that is, from the largest to smallest subsets) through the bitmaps, taking the chars from the hand whose positions are indicated by 1's in our bitmap as the current subset. If the current subset contains a blank tile, we take the 26 available options into account at this stage, generating additional subsets. At this point, subsets are evaluated using (3.1.2) to determine heuristically whether the subset will be employed in a dictionary search.

3.1.2 Choosing good subsets

At each iteration, we evaluate the computed subset on the following heuristics to determine whether to pursue a dictionary search for the subset:

We rank combinations with a score composed of the sum of the bigram scores in the word to the power of e, multiplied by the total letter scores of the word. We tried numerous heuristics and this gave the best results.

In this way, we attempt to determine the likely *m* best subsets, which we will then use to compute scores and positions in (3.1.3). We will determine during implementation what *m* creates a balance between size of the search space and maximum score.

3.1.3 Word search

This is possibly the most interesting part of the problem, and the area to which we have dedicated the most effort thus far. The issues to be balanced are those of space of the representation and time necessary for the search.

We have considered several approaches. Use of a trie or a modified suffix tree is one possiblity. Another is representation of individual words via some compact encoding scheme, such as a bitmap, or perhaps

byte map, in order to account for the possibility of duplicate letters.

These methods have advantages: searching the trie is efficient; search using the bitmap could be done in O(log n) time or perhaps better. We propose here a fairly straightforward alternative using one innovation that, after some upfront intensive computation of in building the datastructures, will result in searches that are only trivially more complex than O(1) and make use of java's method of representing String literals on the heap to avoid using too much space. (see below)

The key is the creation of a hash map that guarantees that all words using the same subset of letters will hash to the same bucket, for example:
slave
vales
salve
must all hash to the same bucket as the subset {a,e,l,v,s}. For simplicity, we ignore duplication of letters in the hashing, so that
slaves
salves
also hash to this same bucket, as would the subset (multiset) {a,e,l,v,s,s}.

Even allowing for words with duplicate letters to hash to the same bucket, we have reduced the problem to one hashing operation, followed by a short, (in most cases ($n<10$)) linear search. We further accelerate this search by taking advantage of java's storage of string literals on the heap. Programmers often wrestle with the fact that the "==" operator is not reliable for use with Strings, and thus resort to `s1.equals(s2)` as a method of determining string equality. This is a very expensive operation, forcing the literal comparison of the two Strings. We can, however, guarantee the success of the "==" operator by invoking java's `intern()` method when creating objects using Strings, thus forcing any object that uses the String "abc" to point to the same String literal object on the heap. Comparing two such objects now becomes a matter of simply comparing the pointers via "==" rather comparing than the strings themselves, significantly reducing the operations involved, given that each `Word` object in our program contains the alphabetized representation of its letters.

Finally, once the bucket has been obtained, we do a linear search over the bucket to make sure the cardinality of the letters is congruent. We do this by using an octal encoding of the words and performing an equals comparison based on that encoding.

4 Mini Max

Finally, we use a straightforward implementation of the minimax algorithm to determine the best move at each turn. We have already employed heuristics above to limit the pool of potential moves to those that produce higher scores and use larger numbers of letters. We prune the subtreees as specified in the homework requirements, search the 3rd and 4th plies when the Bag of available letters is empty. We use a simple heuristic here based on the comparative scores of the hands.

5 Architecture

We use the following packages and classes: src: enos EnosScrabbleClient.java

src/enos: scrabble

src/enos/scrabble: datastruct domain io test util utility

src/enos/scrabble/datastruct: HashOfWords.java

src/enos/scrabble/domain: Anchor.java Board.java Dictionary.java Game.java InvalidMoveException.java LetterBonusSquare.java MoveGroup.java Move.java PlainSquare.java Player.java Rack.java Square.java WordBonusSquare.java Word.java

src/enos/scrabble/io:

src/enos/scrabble/test: BoardTest.java HashTest.java ScrabbleTest.java

src/enos/scrabble/util: BigramEvaluator.java MiniMaximizer.java

src/enos/scrabble/utility: