

Boggle

John Sheu, Chris Agerton

November 2, 2006

I found "leet" in the dictionary.

1 Overview

Boggle is traditionally a game with where players score points for finding words in a 4x4 grid of lettered dice. It has lately been adapted to test the ability of students to create search algorithms that scale to arbitrarily large square matrices of characters.

The game was broken into a dictionary, an engine, and a user interface. Each component will be discussed separately.

2 Algorithms

2.1 Dictionary

The data structure we chose to represent our dictionary was a trie. This is essentially a multi-way tree with one character of each word at each node; to construct a word from this tree, one traverses the tree downwards toward a leaf node, collecting the characters along the way in order. When the leaf is reached, the collected characters will have reconstructed a the string. Among its other advantages, this structure allows for relatively efficient prefix queries; all words sharing a given prefix will share the same traversal path for the length of that prefix, and the presence of a particular traversal path is enough to guarantee the existence of the corresponding prefix.

Insertion into the trie is a linear-time proposition on the length of the inserted key. Only as many edges as there are characters in the key (actually, one less) need to be traversed during insertion. Word and prefix lookups are similarly linear on key length. Iterating over the dictionary is implemented as a preorder traversal with a stack; as all edges need only be traversed once, and the number of those is close to (exactly one less) than the number of letters in the tree, iteration over the entire tree is linear on the number of characters it holds.

2.2 getAllWords(): Board Driven

We decided to use set of two stacks to emulate a recursive board search algorithm without the overhead. An analogous recursive function would be described by the following pseudo-code block:

```
function boardDriven( history, child ) :
  for ( child in children ) :
    if isprefix( child ) && not history.contains( child ) :
      function( history + self, child )
    if word( history + self ) :
      addword( self )
```

With an arbitrarily small dictionary, each letter on the grid would fail the prefix check and the algorithm would complete in $\Theta(size^2)$ or $\Theta(dice)$ time. As the dictionary size increases, there are more and more prefixes chains on the board that also occur in the dictionary. Eventually, with a dictionary of arbitrarily high maximum word size, if all nodes were hypothetically connected to all other nodes, the time would be

$O((size^2)!)$.

In practical terms, this function traverses all sets of legal prefix combinations on the game board twice. When it initially addresses a prefix, it pushes it into the history and pushes any children prefixes onto the work stack. When it returns down the stack, the top node in the history and work stack will be equivalent and it considers it again with the `dict.contains()` method. Thus, this could be said to be a $\Theta(2legal - prefix - combinations)$ algorithm.

In even more practical terms we did a sample search on a randomly generated 32x32 gameboard in 4.3sec cpu time (less the estimated 1.3 sec overhead is 3sec for this search) using the stock dictionary file and recovered 9147 words, scoring 53402 points for the computer.

A more detailed psudo-code description of the algorithm is as follows:

```
stack work, hist
for ( x, y in game-grid ):
    if ( isPrefix( piece( x, y ) ) ):
        work.push( piece( x, y ) )
while ( stack not full ):
    if ( hist empty ):
        hist.push( work.peek() )
        for ( x in hist.peek().neighbors )
            if ( isPrefix( hist + x ) ):
                work.push( neighbor )
    elseif ( hist.peek() != work.peek() ):
        hist.push( work.peek() )
        for ( x in hist.peek().neighbors )
            if ( x not in hist and hist + x is prefix ):
                hist.push( x )
    elseif( hist.peek() == work.peek() ):
        if ( hist is a word ):
            wordlist.add( hist )
            hist.pop()
            work.pop()
```

Note that `isPrefix()` returns true if word is a word itself, regardless of longer words. The sequence of letters in the history stack is cached in a `StringBuilder` so the algorithm can quickly determine if the history stack creates a prefix or a word.

2.3 getAllWords(): Dictionary Driven

The dictionary-based `getAllWords()` method is very similar to the `addWord()` method. Both solve essentially the same problem (finding a given word on the board), only the dictionary-driven `getAllWords()` performs this search over all words in the dictionary. Essentially, every letter on the grid is pushed onto a stack. Then, with each iteration of the algorithm, a letter is popped from the stack, and this letter compared to its corresponding letter in the given word. If it does not match, this iteration ends and the next iteration begins with the next letter on the stack. Otherwise, all of the neighbors of this letter (that are not already visited) are pushed on the stack and the iteration continues.

This algorithm is $\Theta(n)$ in the number of words in the dictionary, as all dictionary words must be iterated over. Again assuming complete connectedness and an arbitrarily high max word size, the algorithm would be $O((size^2)!)$.

3 Implementation

3.1 GameManager

GameManager is the core of the game; it initializes the dictionary, loads the dice, creates the board, keeps scores, and finds all words on the board. Also, it does these things fast. GameManager was created with flexibility in mind. BoardSize was kept left arbitrary, and while you have to have enough dice to fill it up, the dice don't have to be 6 sided. For this assignment, the most important parts of this assignments are the `getAllWords()` and `addWord()` methods.

3.2 User Interface

The command-line user interface for this application was nothing particularly exciting. It has enough command-line switches, gets the job done, and is generally stable to use. Originally, it was proposed that a curses-like interface be developed, but that concept was not regarded as practical given time constraints. Thus, we end up with a fast and functional interface.

3.3 Searches

There is one distinct optimization that can be performed beyond what our `GameDictionary` implementation already contains. During a board-based search, the previously searched word could potentially be cached by the game. Since our internal representation is a trie, testing for prefixes containing a letter's neighbors would then be as simple as checking if a child node of the cached word with the given letter exists. However, this could potentially violate the contract as given as `BoggleDictionary` and make our `BoggleGame` implementation reliant on customized behavior, a compromise which we did not accept.

4 Testing

Search algorithm testing consisted of the following checks

1. Sanity checks for internal data previously assumed to be correct
2. Tracing the search algorithm with verbose checkpoints, pencil and paper
3. Tailored boards with the alphabet and tailored dictionary with pieces and patterns from the aforementioned board
4. Board/dictionary combinations known to produce known results (e.g. one presented on the class Wiki)

5 Pair Programming

The code was written on three occasions. During the first week, we met in the Taylor Basement and constructed the `GameDictionary`, `AlphaTree`, and `Stacks` classes. One party was out of town over the weekend; thus at the earliest convenience, on Tuesday, we camped out in the Painter basement and constructed `GameManager` and `Boggle`. On Wednesday we collaborated over instant messenger programs and used netcat to move files between computers. The game didn't actually work until Wednesday afternoon and the report wasn't constructed until after the deadline. Coding-wise, we generally reverted back to the two-parallel-programmers approach instead of the pair-programming paradigm, perhaps leading to some issues with buggy code.

It may also be worth noting that during the course of debugging, abnormal behavior which was assumed to arise from the complex search algorithms and was found in peripheral supporting functions. This led to much general consternation and time usage issues.