Problem I: Boggle CS2521

Ruaraidh Macfarlane
Student ID: 51228398

March 27, 2015

Compiling and Running

To compile the program run `make boggle` and then run `./boggle`. In the file `main.cpp` there will be the function used to run each task of the program. I have ran the full program with `./boggle valgrind --leak-check=full` and the program is free of memory leaks.

To change the board size throughout the program, edit the global variables in `board_dimensions.h`.

Task 1

To randomly generate a 5×5 board the main method, located in `main.cpp` should call `game.fill()` and to print this board out the main method can call `game.print()`.

Task 2

To run the first part of this task, in the main method, call `user.oneWord(searchWord, boardEncoding)`. If you want to run this task with the randomly generated board, randomly generate a board with `game.fill()` then call `user.oneWord(searchWord, game.encode())`

To run the second part of this task, simply run: `user.listWords(boardEncoding)`. Again, if wanting to run this method with the randomly generated board then pass in `game.encode()` as the parameter.

Poof that this will run in linear time is that for each letter of the word the board will recursively check all adjacent cells, which at maximum, will be 8. This can be written as a reccurence.

$$T(n) = 8T(n-1) + 1 \tag{1}$$

Assuming that if the word can be completed in the board, the next letter of the word will only appear once in adjacent cells. And also assuming that the

max length word is less than the size of the dictionary. Then this algorithm will run in $\Theta(n)$ time.

It is trivial that this solution will have the space complexity of $\Theta(nm)$ as the program will hold a dictionary of length m with words of length n.

Task 3

To run this task, call the function `user.BSBoggle(boardEncoding)`. If wanting to run this method with the randomly generated board then pass in `game.encode()` as the parameter.

The binary search in this solution takes $\mathcal{O}(logn)$. The problem is when randomly generating words, creates a huge search space. For a board of $n \times n$, the search space at worst case will be $\mathcal{O}((n \times n)!)$ as that is how many different words can be created. This will cause the program to run for ages as for each combination it will binary search the whole dictionary.

This algorithm can be significantly improved by adding a check for prefixes. The algorithm will first binary search the dictionary to check if the current word made from the board is a prefix of any word in the dictionary. If it is, the recursion will continue, otherwise it will return from the current branch of the recursion that generates possible words.

Empirical Analysis

I wrote a ruby script to randomly pick n amount of words from the `Wordlist.txt` file to create different size dictionaries, the script is called `dic_script.rb`. I then ran the program with different randomly generated boards and if at least one word was found, noted the running time for the function.

In each table, m is the dictionary size and each t is the time in seconds taken to run the program.

Task 2 Analysis

	t1	t2	t3	t4	t5	mean
m=100	0.000126	0.000127	0.000113	0.000116	0.000120	0.0001204
m=1000	0.002694	0.008036	0.008796	0.008229	0.004709	0.0065468
m=10000	0.00163	0.006822	0.005242	0.004238	0.004448	0.004476
m=267751	0.841076	0.933726	0.87403	0.827972	0.764547	0.8482702

Task 3 Analysis

Task 4

To run this task, call the function `user.efficientBoggle(boardEncoding)`. If wanting to run this method with the randomly generated board then pass in

	t1	t2	t3	t4	t5	mean
m=100	0.001595	0.001139	0.001134	0.001246	0.001453	0.001313
m=1000	0.005479	0.008151	0.005441	0.00627	0.006934	0.006455
m=10000	0.003409	0.007585	0.006266	0.006126	0.015208	0.007719
m=267751	0.616621	0.869006	0.409174	0.486665	0.459186	0.568130

`game.encode()` as the parameter.

The data structure used in this program to make it more efficient is the use of a Trie (Prefix Tree). The Trie is a representation of the dicitionary. It takes a while to initalise so for the current dictionary size, may not be quicker than Task 3. But for bigger dictionaries this algorithm will be more efficient as it only needs to create the trie once.

The Trie reduces our search space, building the Trie is $\mathcal{O}(MN)$ where M is the dictionary length, and N is the length of the biggest word in the dictionary.

To make this program even more efficient, we could use dynamic programming. This would get rid of the search space completely.

Empirical Analysis

For each program I ran using the same randomly generated board and same dictionary size. For each different t a different board is used and only results that returned at least one found word are noted.

For all of these I generate the Trie of the dictionary before I run all the programs.

Task 4 Analysis

	t1	t2	t3	t4	t5	mean
m=100	0.0008	0.000744	0.000798	0.000732	0.000765	0.0007678
m=1000	0.001973	0.002511	0.002037	0.002365	0.00294	0.0023652
m=10000	0.00381	0.001903	0.001561	0.000889	0.004448	0.0025222
m=267751	0.030319	0.023292	0.021349	0.022759	0.01641	0.0228258