Analysis for Boggle Assignment

Yuanjie Jin

1) LexiconBenchmark

I tried LexiconBenchmark as the first approach to test the empirical performance of each of the four Lexicon implementation, namely SimpleLexicon, BinarySearchLexicon, TrieLexicon and CompressedTrieLexicon. The data from LexiconBenchmark are summarized in Table 1 and 2. The time shown is in seconds. These data are representative of data from multiple runs. When testing TrieLexicon and CompressedTrieLexicon, I also implemented their specific methods nodeCount and oneWayCount to count the nodes saved in the Lexicon. The output these two methods returned was entered in the node count and oneway count part in Table 1 and 2, respectively. As shown in the tables, totally four text files are loaded in benchmarking.

From these results, I noticed that the size of Lexicon, the total number of words, the number of matched words and the number of random-generated prefixes got from the lexicon all match across different Lexicons, suggesting that my implementation of different Lexicons work correctly.

In terms of efficiency, I have following observations:

The runtime it takes to iterate through the Lexicon, find words and find prefixes are all close between SimpleLexicon and BinarySearchLexicon, indicating they are similar in efficiency. In contrast, TrieLexicon and CompressedTrieLexicon seem to be more timeconsuming in iteration. CompressedTrieLexicon also takes more time to find prefixes than all the other Lexicons. These phenomena tend to be more significant with larger Lexicon size.

Although CompressedTrieLexicon takes more time to find prefixes compared to TrieLexicon (e.g. about ten-fold as much time as TrieLexicon did when ospd3.txt file was used in benchmarking), it uses less memory than TrieLexicon by significantly reducing the number of nodes stored in the Lexicon.

On the other hand, CompressedTrieLexicon does not necessarily take longer to iterate through all its words than TrieLexicon. For example, it spent shorter time in iteration when ospd3.txt file was loaded in benchmarking. One reason I can think of for this inconsistency is that TrieLexicon has more nodes than CompressedTrieLexicon, i.e. the iterator has to move through more nodes to complete the iteration process.

Table 1. LexiconBenchmark Results

	kwords5.txt				SmallLexicon			
SimpleLexicon:			size of Lexicon	5757			size of Lexicon	19912
	iter time:	0	size:	5757	iter time:	0	size:	19912
	word time:	0	words:	5757	word time:	0.016	words:	19912
	pref time:	0.016	size:	1929	pref time:	0.015	size:	6143
BinarySearchLexicon:			size of Lexicon	5757			size of Lexicon	19912
	iter time:	0	size:	5757	iter time:	0	size:	19912
	word time:	0	words:	5757	word time:	0	words:	19912
	pref time:	0.016	size:	1929	pref time:	0.016	size:	6143
TrieLexicon:			size of Lexicon	5757			size of Lexicon	19912
	iter time:	0.016	size:	5757	iter time:	0.125	size:	19912
	word time:	0	words:	5757	word time:	0	words:	19912
	pref time:	0.016	size:	1929	pref time:	0.015	size:	6143
	node count:	12104	oneway count:	4063	node count:	61617	oneway count:	37549
CompressedTrieLexicon:			size of Lexicon	5757			size of Lexicon	19912
	iter time:	0.016	size:	5757	iter time:	0.015	size:	19912
	word time:	0	words:	5757	word time:	0	words:	19912
	pref time:	0.015	size:	1929	pref time:	0.141	size:	6143
	node count:	8236	oneway count:	195	node count:	30660	oneway count:	6592

Table 2. LexiconBenchmark Results (cont.)

	lowerwords.txt				ospd3.txt			
SimpleLexicon:			size of Lexicon	45356			size of Lexicon	80612
	iter time:	0.015	size:	45356	iter time:	0	size:	80612
	word time:	0	words:	45356	word time:	0	words:	80612
	pref time:	0.016	size:	10852	pref time:	0.031	size:	16466
BinarySearchLexicon:			size of Lexicon	45356			size of Lexicon	80612
	iter time:	0	size:	45356	iter time:	0	size:	80612
	word time:	0	words:	45356	word time:	0	words:	80612
	pref time:	0.015	size:	10852	pref time:	0.031	size:	16466
TrieLexicon:			size of Lexicon	45356			size of Lexicon	80612
	iter time:	0.187	size:	45356	iter time:	0.203	size:	80612
	word time:	0	words:	45356	word time:	0	words:	80612
	pref time:	0.016	size:	10852	pref time:	0.015	size:	16466
	node count:	1E+05	oneway count:	61347	node count:	153759	oneway count:	70578
CompressedTrieLexicon:			size of Lexicon	45356			size of Lexicon	80612
	iter time:	0.282	size:	45356	iter time:	0.078	size:	80612
	word time:	0	words:	45356	word time:	0	words:	80612
	pref time:	0.047	size:	10852	pref time:	0.141	size:	16466
	node count:	73404	oneway count:	24203	node count:	114921	oneway count:	31740

2) BoggleStats

To simulate a number of auto-games, I modified the code in the class of BoggleStats so that it returned time it takes to run for 1,000 games, 10,000 games and 50,000 games as well as the board with the highest score among all the 4x4 and 5x5 boards generated. To compare the performance across different implementations of AutoPlayer and ILexicon, I set the number of seeds in the object of Random class the same (i.e. 12345), which was able to give a reproducible sequence of random boards each time. The data from BoggleStats are summarized in Table 3 and 4. The time shown is in seconds.

It is obvious to see that LexiconFirstAutoPlayer runs much slower than BoardFirstAutoPlayer. This makes sense because I expect that LexiconFirstAutoPlayer would do more repetitive work in searching words with the same prefixes because it starts over the search algorithm every time it looks up a word/prefix in the Lexicon. In my computer, whatever Lexicon implementation I used, it took at least 1.5 – 2 hours for LexiconFirstAutoPlayer to run for 10,000 games. Because of time constraint, I did not try to run LexiconFirstAutoPlayer for 50,000 games or more; instead, the runtime was predicted. However, I did run BoardFirstAutoPlayer to get the board with the highest score after playing 50,000 games since LexiconFirstAutoPlayer and BoardFirstAutoPlayer should return the same results for that.

The runtime for 100,000 games or more is all predicted. My predictions are based on my hypothesis that the runtime and the number of games are almost linearly correlated for each specific AutoPlayer and Lexicon. This hypothesis was consistent with the data shown. For example, the runtime for 10,000 games is approximately 10 times as much as the runtime for 1,000 games.

When comparing performance between different implementations of Lexicons using LexiconFirstAutoPlayer, I find that the runtime is close between SimpleLexicon and BinarySearchLexicon, and they are both faster than the other two Lexicons. TrieLexicon and CompressedTrieLexicon spend similar time with TrieLexicon being slightly slower possibly due to the reason that TrieLexicon sometimes goes through more nodes when looking up a word. These results are basically consistent with what is shown in LexiconBenchmark and some are more significant.

As SimpleLexicon stores words in a TreeSet and BinarySearchLexicon does binarysearch for words in a sorted ArrayList, looking up a word should be an O(logN) operation for both of them where N is the number of words in the whole Lexicon. This explanation well fits the above data in which their runtime are close to each other.

When using BoardFirstAutoPlayer, however, TrieLexicon turns out to be the fastest, then BinarySearchLexicon and SimpleLexicon, with CompressedTrieLexicon being slowest. This interesting result could be caused by the dramatic reduction in the number of words (and thus the cumulative length of all words) to be searched for in BinarySearchLexicon compared to LexiconFirstAutoPlayer, since looking up a word or a prefix in a regular Trie is an O(W) operation where W is the length of the string.

Table 3. Results of BoggleStats for 4x4 board

(Yellow-color shaded regions are predicted time)

4x4 board		Simple Lexicon	BinarySearch Lexicon	Trie Lexicon	Compressed-TrieLexicon	max score	max score board
1,000 games	LexiconFirst AutoPlayer	501.64	492.047	600.985	576.031	889	g s r g n e t i i o s b p r e n
	BoardFirst AutoPlayer	4.094	3	1.687	6.813		
10,000 games	LexiconFirst AutoPlayer	4809.265	4735.828	5702	5627.031	889	g s r g n e t i i o s b p r e n
	BoardFirst AutoPlayer	39.359	29.266	16.391	65.547		
50,000 games	LexiconFirst AutoPlayer (predicted)	~24046	~23679	~28510	~28135	1011	c l i t s m e r b d a s c l e h
	BoardFirst AutoPlayer	196.875	149.172	83.188	329.844		
100,000 games (predicted)	LexiconFirst AutoPlayer	~48093	~47358	~57020	~56270		
	BoardFirst AutoPlayer	~394	~298	~166	~660		
1,000,000 games (predicted)	LexiconFirst AutoPlayer	~480927	~473583	~570200	~562703		
	BoardFirst AutoPlayer	~3938	~2983	~1664	~6597		

Table 4. Results of BoggleStats for 5x5 board

(Yellow-color shaded regions are predicted time)

5x5 board		Simple Lexicon	BinarySearch Lexicon	Trie Lexicon	Compressed TrieLexicon	max score	max score board
1,000 games	LexiconFirst AutoPlayer	797.063	789.375	865.265	860.219	1301	o t r p w d b n o l r e s e s s t n i m w n i s h
	BoardFirst AutoPlayer	11.687	8.734	5	20.235		
10,000 games	LexiconFirst AutoPlayer	7858.609	7804.828	8632.047	8525.735	2120	p a c o d o x s e r a t n t r n i e a s d r n c e
	BoardFirst AutoPlayer	116.953	88.093	51.531	204.625		
50,000 games	LexiconFirst AutoPlayer (predicted)	~39293	~39024	~43160	~42629	2120	p a c o d o x s e r a t n t r n i e a s d r n c e
	BoardFirst AutoPlayer	588.031	434.75	255.094	1018.531		
100,000 games (predicted)	LexiconFirst AutoPlayer	~78586	~78048	~86320	~85257		
	BoardFirst AutoPlayer	~1176	~870	~510	~2037		
1,000,000 games (predicted)	LexiconFirst AutoPlayer	~785861	~780483	~863206	~852574		
	BoardFirst AutoPlayer	~11760	~8695	~5101	~20370		

3) Conclusion

Taken together, there is no perfect implementation of Lexicons. BinarySearchLexicon and SimpleLexicon are almost equally efficient, with the former slightly faster in looking up a word. Generally speaking, BinarySearchLexicon is the best implementation of Lexicon in terms of speed. In a few cases, like running a BoardFirstAutoPlayer, TrieLexicon seems to perfom faster than all the other Lexicons. Although CompressedTrieLexicon is not advantageous in speed, it manages to save lots of memory compared to TrieLexicon.