Xing Su

Compsci 201

Professor Peck

04/05/13

Boggle Analysis

1. ***You need to report in your Analysis file information about which Lexicon implementation is fastest. You should compare all the lexicons and report on their relative times --- you shouldn't simply say "this one is fastest". You should have at least three lexicons to test and four if you do the extra credit. You should explain the methods you used to determine this and report on experiments you run, giving times.***

   Below is the output from `BoggleStats.java` with a 5x5 board and 10 trials:

```
LexiconFirstAutoPlayer SimpleLexicon          time: 2.727000
BoardFirstAutoPlayer SimpleLexicon            time: 0.369000
LexiconFirstAutoPlayer BinarySearchLexicon    time: 1.769000
BoardFirstAutoPlayer BinarySearchLexicon      time: 0.230000
LexiconFirstAutoPlayer TrieLexicon            time: 2.066000
BoardFirstAutoPlayer TrieLexicon              time: 0.174000
LexiconFirstAutoPlayer CompressedTrieLexicon  time: 2.161000
BoardFirstAutoPlayer CompressedTrieLexicon    time: 0.060000
```

   Based on the above results, it cannot be concluded which implementation is overall definitely better than the rest. However, for BoardFIrstAutoPlayer, it seems that CompressedTrieLexicon is the fastest. Because of the structure differences, large variations can be seen: for example, because LexiconFirstAutoPlayer need to look through all of the words and thus structuring the words in a BinarySearch fashion will help improving the look up time, thus BinarySearch seems to be the faster implementation. For BoardFirstAutoPlayer, it is trying to find all combinations of words from the board and CompressedTrieLexicon is the most efficient with its compressed nodes and thus the fastest.

2. ***You must write code to play lots of auto-games, not games in which humans play, but games in which all words are found on thousands of boards --- see `BoggleStats` for a starting point. You must provide a board that scores the highest of all the 4x4 and 5x5 boards you test in running at least 10,000 auto-games. and preferably 50,000 games. Report on how many seconds it takes your code to run for 1,000 games; for 10,000 games (or predict that if***

you can't run that many); and predict/justify on how much time it would take your computer and implementation to simulate both 100,000 games and one million games. When doing the experiments be sure to set the random-number generator's seed, currently done already in `BoggleStats` and described above. If you can't run 10,000 auto-games, indicate the maximum number you can run.

Below is the output from `BoggleStats.java` with `BoardFirstAutoPlayer`:

```
SimpleLexicon             count: 1000  max: 558     time: 2.273000
BinarySearchLexicon       count: 1000  max: 558     time: 1.456000
TrieLexicon               count: 1000  max: 558     time: 0.998000
CompressedTrieLexicon     count: 1000  max: 685     time: 0.835000

SimpleLexicon             count: 10000 max: 772     time: 22.423000
BinarySearchLexicon       count: 10000 max: 772     time: 16.037000
TrieLexicon               count: 10000 max: 772     time: 11.316000
CompressedTrieLexicon     count: 10000 max: 897     time: 11.967000

SimpleLexicon             count: 50000 max: 781     time: 108.793000
BinarySearchLexicon       count: 50000 max: 781     time: 83.521000
TrieLexicon               count: 50000 max: 781     time: 57.156000
CompressedTrieLexicon     count: 50000 max: 939     time: 59.922000
```

As we can see from the above data, the increase in time needed is linear in that increasing the number of trials by ten-fold would increase the running time by 10 times. Thus I predict that the running time for 100000 games would be 1100, 830, 585, and 600 seconds respectively for Simple, Binary Search, Trie, and CompressedTrie lexicons. The overall highest score is 781.