Statistical and Empirical Analysis of Boggle

Justin Palpant

11/11/13

Table of Contents

Table of Figures

Comparison of Lexicons

I used the provided class LexiconBenchmark to complete the majority of my analysis of the relative benefits of different lexicons. From what I have observed, there is not one clear "fastest" Lexicon for all situations. In order to compare lexicons, I used LexiconBenchmark for each of the three lexicon structures available to me – SimpleLexicon, BinarySearchLexicon, and TrieLexicon. I also used all four of the different dictionaries that were available to me so that I could control the size of the lexicon. With four dictionaries, it is possible not only to compare the performance of the different lexicons, but also make tentative predictions on how the runtime of each test is related to the size of the dictionary.

Iterating over the lexicon

The first test for each lexicon compares the amount of time it takes to iterate over all of the words in a lexicon. The following graph shows the runtimes of this test for each of the three lexicons compared, versus the size of the lexicon.

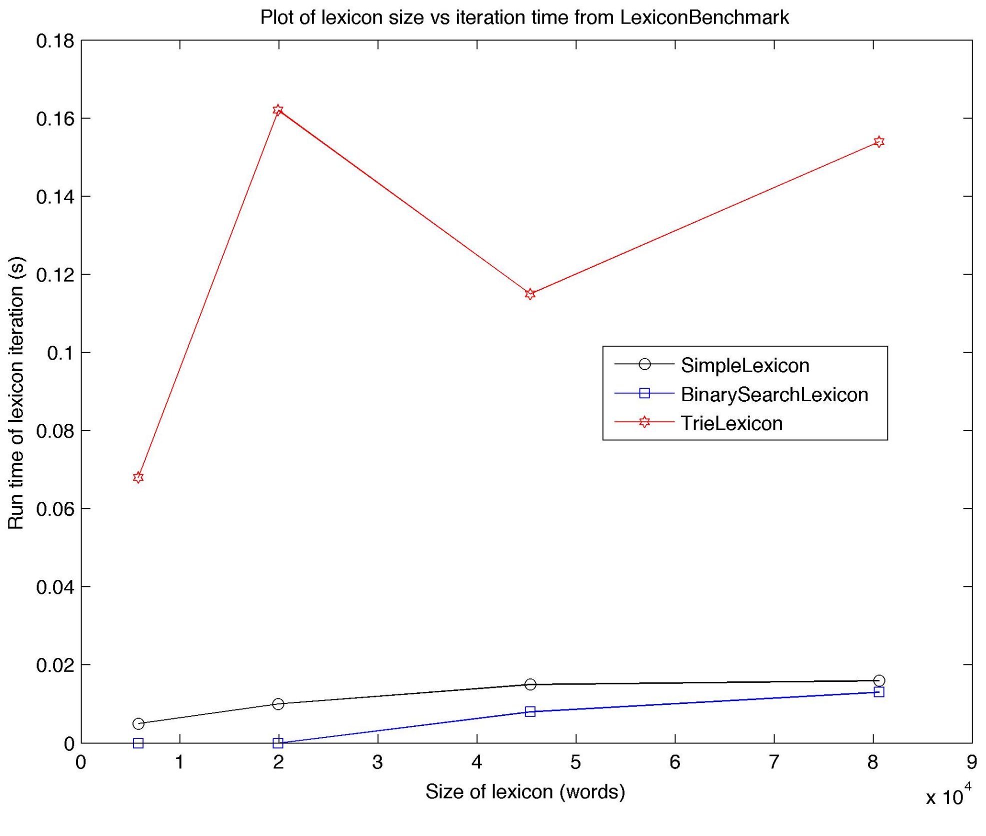

Figure 1. Lexicon Iteration Test

It is immediately apparent that iterating over any size lexicon is much more costly with a TrieLexicon than with either of the other two lexicons. However, all times are relatively small (even when the size of the lexicon exceeds 80,000 words), and all appear to be linear with the size of the lexicon. As such, it is entirely possible that the poorer performance of TrieLexicon is the result of some sort of overhead – we can still expect O(N) performance for this task from all lexicons.

Finding words in the lexicon

The second test involved search through the lexicon to determine whether or not a given word was within the lexicon or not. The graph below shows the runtime of this test for each of the lexicons with each dictionary.

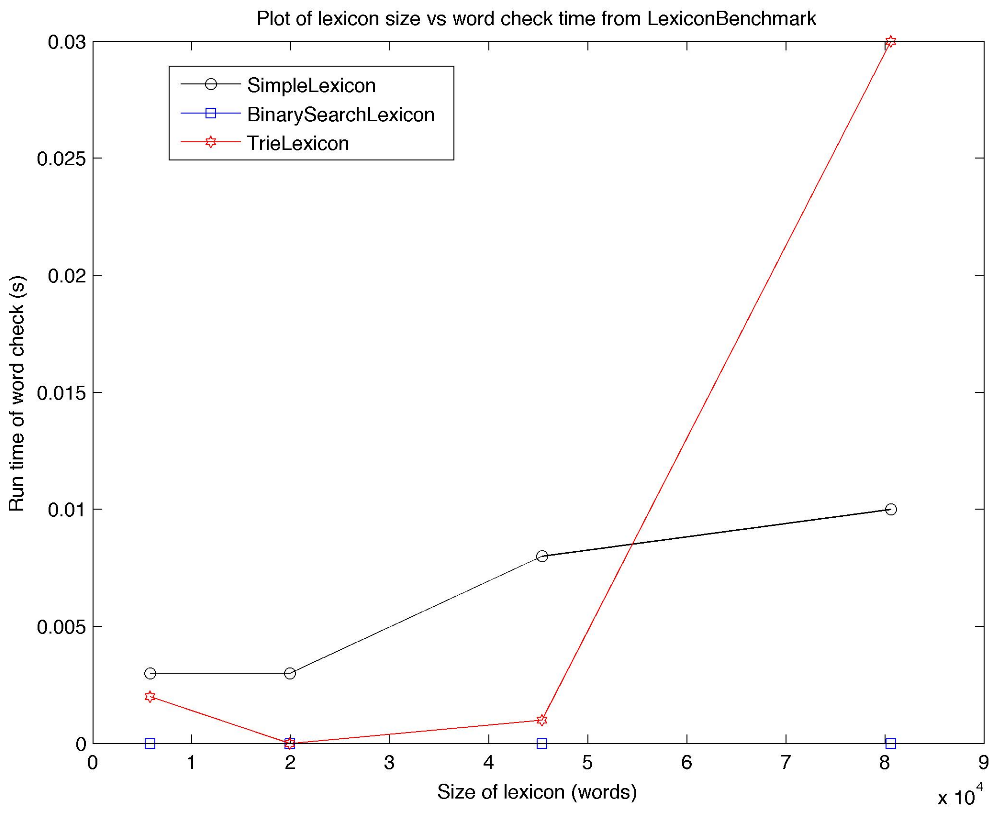

Figure 2. Lexicon word test

In this case, the results are much more scattered and harder to predict, possibly because of the lack of sufficient data points. It seems that for a BinarySearchLexicon it takes almost no time to correctly determine if a word exists – the values were consistently 0.000s. This may be because the BinarySearchLexicon uses the built-in method Collections.*binarySearch*(), followed by very fast comparison statements, so that the performance of the method is entirely based on the apparently excellent performance of Collections.*binarySearch*(). For a SimpleLexicon, the time is roughly linear with the size of the lexicon, and this trend appears to be consistent, so we can assume that there is O(N) performance for checking the status of a word in a SimpleLexicon. The TrieLexicon has no discernable trend with respect to lexicon size, and seems to vary widely. It is never as fast as the BinarySearchLexicon, but is both faster and slower than SimpleLexicon. I think this is because TrieLexicon is a class largely coded independently, so it doesn't make use of built-in Java structures. This means that the runtime performance may be less stable – sometimes faster, sometimes slower, than the other lexicon. In spite of the trend in the graph, I do <u>not</u> think that TrieLexicon has any sort of exponential or polynomial runtime, but we cannot determine the actual big-O notation of TrieLexicon from this data.

Finding prefixes in a lexicon

The final test involved finding prefixes of words in a lexicon. The graph for this data, which was generated at the same time as the previous two data sets, is shown below. In this case, we still use the size of the lexicon as the x-axis variable, even though the number of prefixes is distinct from the size of the lexicon.

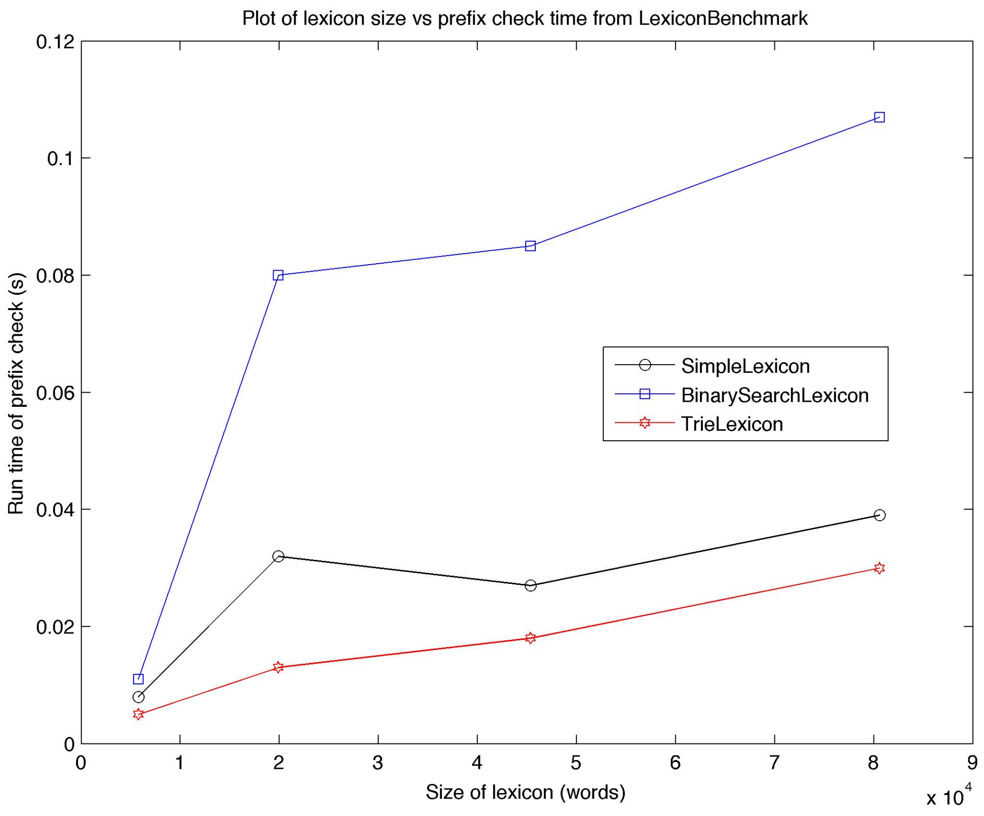

Figure 3. Lexicon prefix test

The behavior of all three lexicons is very consistent in this test – all appear to provide some sort of linear increase with the size of the lexicon, and it appears that it is possible, for this test, to rank the lexicons, with TrieLexicon being consistently faster than SimpleLexicon, and BinarySearchLexicon being slower than either of the other two. This is somewhat surprising, because BinarySearchLexicon's performance in the previous test was so superior to the other lexicon's performances.

Comparison of AutoPlayers

The second part of the analysis of Boggle involved comparing the two computer player classes we developed (LexiconFirstAutoPlayer or LexFirst and BoardFirstAutoPlayer, or BoardFirst), and determining the maximum scoring board in a series of randomly generated boards, and making an estimate for the time-cost of counting all of the words on 100,000 boards, and 1,000,000 boards.

First, it can be stated that in all cases, BoardFirstAutoPlayer is the superior player, and is much quicker at finding words than LexiconFirstAutoPlayer. This is true regardless of lexicon. It is worth investigating, however, by how much BoardFirst exceeds LexFirst.

AutoPlayer and Lexicon analysis

To begin the analysis, first we compare the performance of each autoplayer with respect to lexicon, to see if one lexicon performs better in real play than another. This is similar to the analysis we already did, but in a more realistic setting. The graph below shows the runtime of LexFirst with each of the three available lexicon types, always using the largest possible lexicon of 80,000+ words.

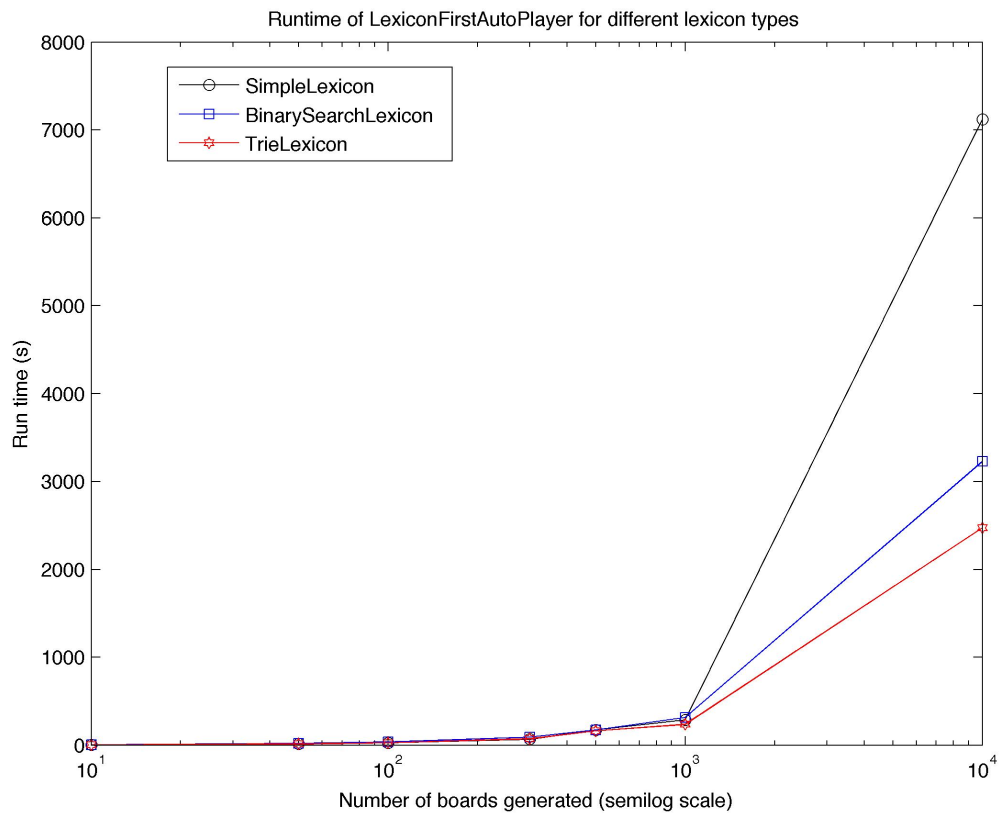

Figure 4. LexiconFirstAutoPlayer timing

This graph uses a semilog scale, with the x-axis in number of trials on a logarithmic scale, for ease of viewing. We can see on this graph that there is very little difference visible between the different lexicons. In the extreme case, there is a great difference – when 10,000 trials are attempted, the SimpleLexicon appears to take twice as long. This is an artifact of poor testing, however, as I suspect that test occurred across a time when I closed my computer, halting the execution of the test for approximately an hour. As such, some 3600 of the over 7000 seconds which the test took to complete may be the result of the shutdown. Other than that case, all the lexicons appear to behave similarly.

Additionally, the amount of time to complete the test appears to be linear with the number of trials – remember, the semilog scale on the x-axis makes the graph appear exponential, when in fact it would be linear in a normal scale. However, this assumption will be tested later.

The following graph shows the same test and the same information for BoardFirst:

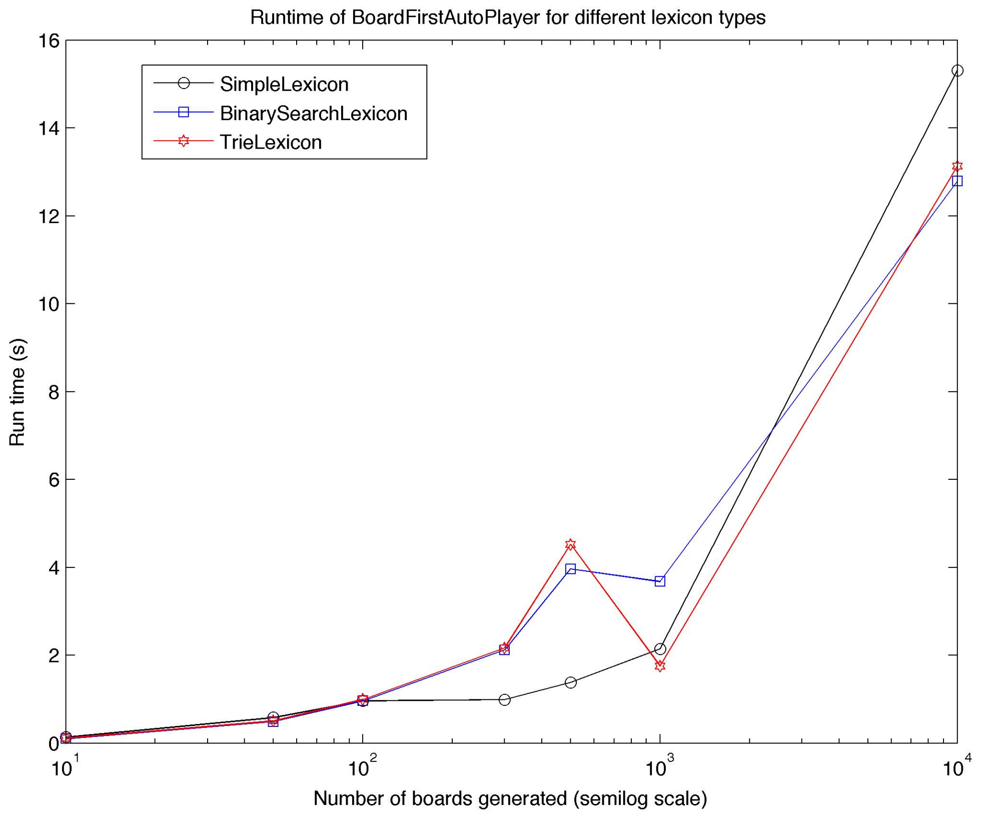

Figure 5. BoardFirstAutoPlayer timing

In this graph we can see a much clearer, more gradual exponential pattern, indicating (because this graph also uses the semilog scale on the x-axis) another linear pattern for BoardFirst. Also, we can once again see that there is no significant difference between lexicons. However, it is most interesting to note the absolute times on the y-axis – BoardFirst never takes more than 16 seconds to complete the trial, compared to upwards of several thousand seconds for LexFirst. This difference is the next thing to explore.

Comparison of Average Performance of Autoplayers

In order to accurately compare autoplayers, and to make use of all of the data collected, we first take the average of the time-values at each point. This means that we are treating the different lexicons simply as different trials, and are taking the arithmetic mean of three trials in each case.

From this average data it would be immediately apparent that LexFirst is inferior, but in order to find the degree of inferiority, we are going to attempt to model the time-vs-number of trials relationship for each lexicon.

Power-law fit

A power-law fit of a data set is a fit which behaves according to the equation

$$\hat{y} = a * \ \hat{x}^b$$

Or, for our data

$$\widehat{Time} = a * \ num\widehat{Trials}^b$$

We will then attempt to find a model with this form which minimizes the least-squares of the residuals between the model and the actual data points we have at hand.

This is remarkably easy in MATLAB, and the method for doing so can be seen in the MATLAB code included at the end of this report. Below is a plot of the two original data sets (the times for LexFirst and BoardFirst) and their respective fits. This plot is made with a log-log scale, meaning that both of the axes are logarithmic. This will make all power fits appear linear, and is primarily done for convenience so that the difference in the data sets are clear and all data points are easily visible.

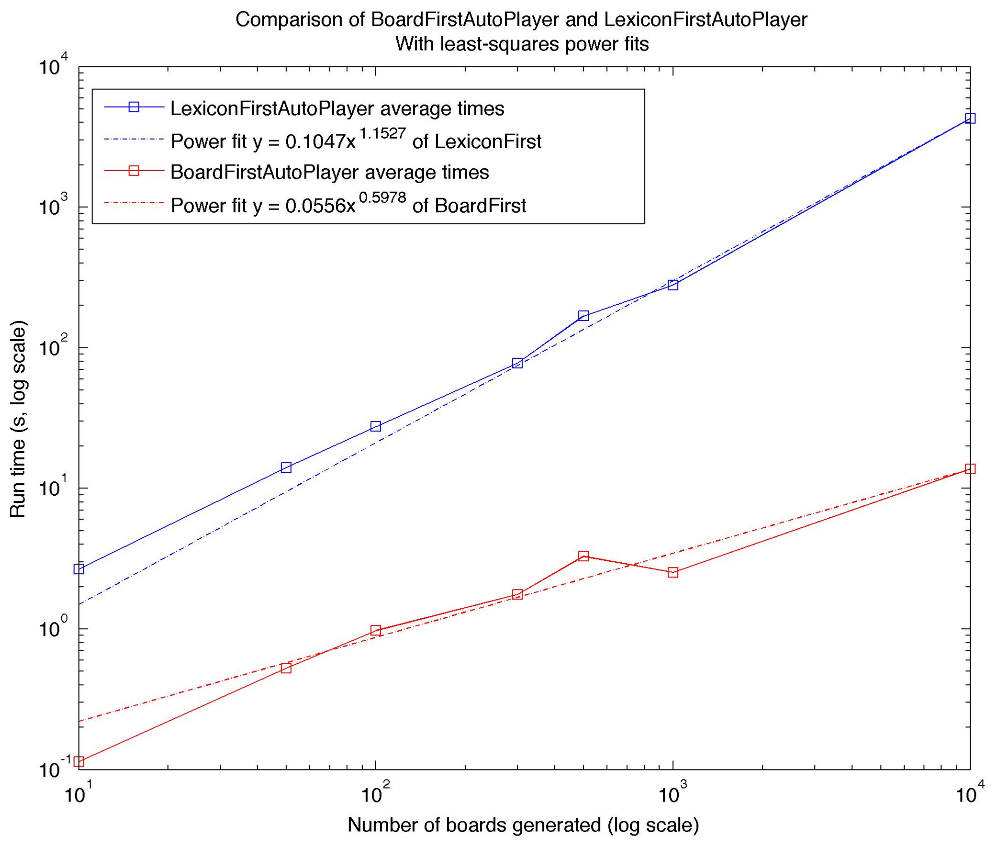

Figure 6. Log-log plot of AutoPlayer average performance, and power fits

We can see on this log-log plot that the power-law relationship very closely models both data sets. We can also see that BoardFirst performs better than LexFirst, and this difference is *enormous* – it is between two and three orders of magnitude faster than LexFirst.

This huge difference in speed can also be seen in the coefficients a and b of the power fits. LexFirst has coefficients a = 0.1047 and b = 1.153, indicating a slightly super-linear performance, while BoardFirst has coefficient a = 0.0556 and b = 0.5976, indicating better-than-linear performance. The difference in the b coefficients is particularly striking.

Using these equations, we can estimate the amount of time it would take for each lexicon to run with x = 100,000 and x = 1,000,000. We find that LexFirst should take 4273.2 and 8.6323*10^5 seconds respectively, while BoardFirst should require 13.688 and 214.74 seconds.

However, it is worth being suspicious of the better-than-linear fit of BoardFirst, since the programming of the tests should result in linear performance for both LexFirst and BoardFirst. If we do not accept that these non-linear fits are appropriate, we can attempt linear fitting, and from the linear fitting, we can actual achieve a direct comparison of the performance of the boards.

Power-law fit with exponent clamped at 1

However, we have in the past made the assumption that the behavior of some algorithms is truly linear, O(N), where the exponent on the N is one. If we make the same assumption here, then our power fits cannot be assumed to accurately model the data, even if they do minimize the sum of the squares of the residuals. So instead, we will attempt to find a least-squares fit which minimizes the residuals while only controlling the leading coefficient, while the exponent is clamped at one and invariable. The model equation will then resemble:

$$\widehat{Time} = a * \widehat{numTrials}$$

This is similar to a linear fit where the zero-intercept is clamped at zero and only the slope of the line is variable. This requires almost no extra machinery, and a graph of the values (one the same log-log scale) of the data and the now-linear fits is shown below:

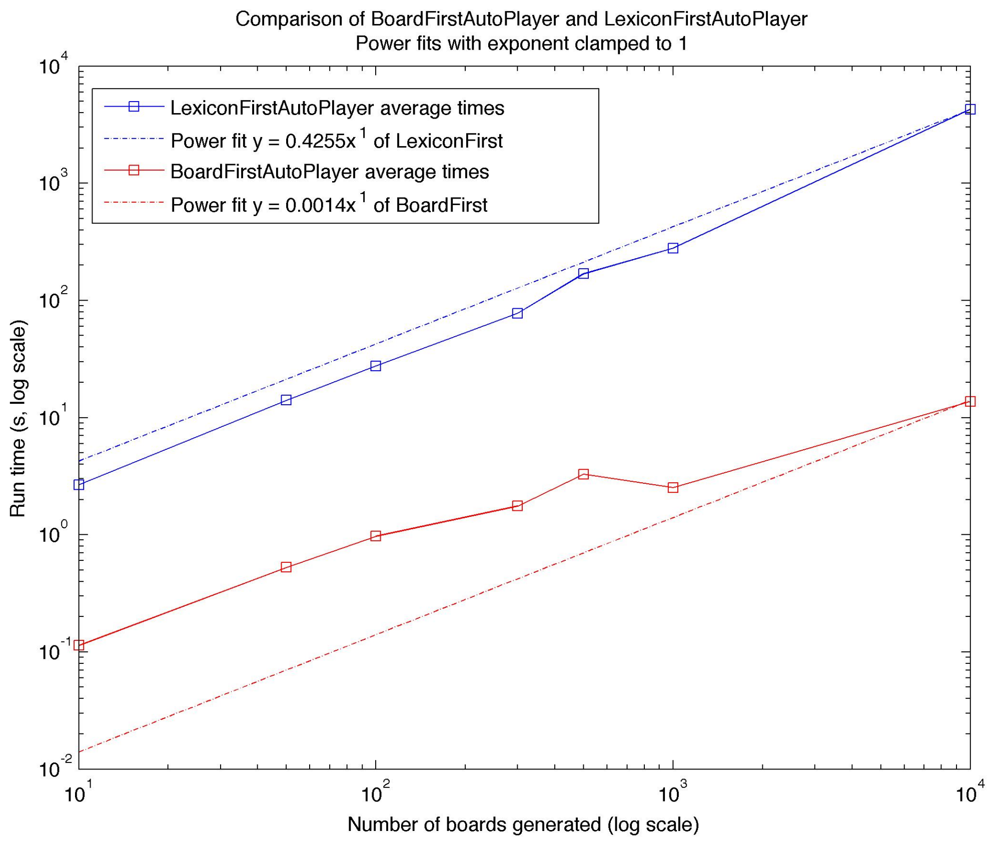

Figure 7. Log-log plot of autoplayer performance, with clamped power fits

This plot once again uses the log-log scale for the axes for ease of view. We can see quickly that the fits that are forced to have an exponent of one and no intercept do not model as well the data. In fact, due to the least-squares requirement, we know that the model will attempt to fit the largest values first, and these models do so with remarkable accuracy, while missing the lower values.

However, the one coefficient which was varied, a, gives us remarkable insight into the superiority of BoardFirst. We see that LexFirst has $a = 0.4255$, while BoardFirst has $a = 0.0014$. The ratio between these two is 303.93 – this gives us a good number to use as a direct comparison of the two autoplayers, and we can say that in a general case, we can expect BoardFirst to perform some 300 times as efficiently as LexFirst. More data points or repeated testing might cause this ratio to vary slightly, but it gives us a good feel for the respective performance of these autoplayers. With these fits, we estimate that LexFirst would take 4255 and $4.255 * 10^5$ seconds, and BoardFirst would take 14.038 and 1403.8 seconds for x = 100,000 and 1,000,000 trials, respectively.

Best Board

As a last note, the assignment requires that we find the board that scored the highest during the trials. In 10,000 trials, the 4x4 board with the highest score had a score of 889, and is the board found when using the seed 12345 on the iteration number 963. The board is below:

G	S	R	G
N	E	T	I
I	O	S	B
P	R	E	N

However, there is no reason at all to claim that this is actually a very word-dense board. In fact, finding the most word-dense 4x4 Boggle board is very challenging, as it is impossible to iterate over all of the possible boards. There are 6^16 possible boards for each arrangement of cubes, with 16! possible permutations of cubes, divided by the eight-fold symmetry of the possible arrangements yields $\sim 7.3 * 10^{24}$. This is obviously an over-estimation, as it assumes all of the letters on the board and all of the cubes – some more conservative estimate state the number of boards is on the order of $4.5 * 10^{20}$. On my computer, the BoardFirstAutoPlayer can score a single board in, based on our power fit with clamped exponents of 1, 0.0014 seconds, or approximately 700 boards per second. To search even the conservative estimate of the entire space would take over 20 billion years, so another method would need to be devised.

I have made a very quick attempt to do so with the additional class BoggleBoardHunter, included in the submission. This class attempts to use gradient descent to search for a better board by making small, iterative changes to a board and only accepting those changes that improve the board's score.

First, it is important to note that there are two kinds of fundamental changes which can be made randomly to a Boggle board: either a single cube can be rotated so that a random side faces up, or two cubes can be swapped without rotation. I coded two methods which take a given Boggle board and return a new board with one of these two changes made to it, without altering the original board. I did so using the list of Boggle dice contained in the StandardBoardMaker class. Additionally, I employed a BoardFirstAutoPlayer for scoring purposes, and used the file *ospd3.txt* with a BinarySearchLexicon, and a minimum word length of 3 letters.

The class BoggleBoardHunter is written so that it will only accept the changes to a board if the new board has more words present than the old board. Additionally, it will only permit failure a limited number of times before deciding that the current board is the best possible board, specified in *FAIL_LIMIT*. For obvious reasons, the length of the search will be related to the allowed number of failures, and a better board will be found, in general, if more failures are permitted. The number of failures, also, is reset each time a good board is found. The number of allowed failures is the number of random changes that will be attempted prior to giving up, but remembering that there are more than 10^{20} possible boards, no reasonable number of allowed changes will result in the class searching all of the possible boards. Instead, we must trust that the gradient descent search to give us better performance than the random generation of boards which is used in BoggleStats.

Below is a table that shows, for a *FAIL_LIMIT* of 1000, one attempt to hunt for the best Boggle board, showing only the boards that were accepted as improvements. The boards are shown as horizontal strings of length 16, rather than in board form, so conserve space.

Board Number	Board String	Score
5	tjkysrntntrnooga	92
6	tjkysrntngrnoota	94
7	tjkysrntngrnooia	126
10	ajkysrntngrnooia	181
13	ajkysretngrnooia	435
14	ajhysretngrnooia	485
38	ajhysregntrnooia	564
49	njhysregatrnooia	584

51	nbhysregatrnooia	618
54	obhysregatrnnoia	811
60	obsysregatrnnoia	1014
81	obsasregatrnnoia	1037
108	obsasaegrtrnnoia	1049
112	obsasaegrtrntoia	1070
143	obsasaegotrntria	1076
145	obsasaogetrntria	1141
195	absasaogetrntria	1153
219	absataogetrnsria	1213
232	absataogetrnrsia	1220
281	absataogstrnreia	1354
317	absataegstrnreia	1493
321	absftaegstrnreia	1513
325	absptaegstrnreia	1554
359	absrtaegstrnpeia	1581
364	gbsrtaegstrnpeia	1604
387	gpsrtaegstrnbeia	1749
413	gpsrtaegstrnseia	1853
482	npsrtaegstrnseia	1869
484	epsrtaegstrnseia	1892
498	pesrtaegstrnseia	1969
541	perstaegstrnseia	2028
793	serstaegptrnseia	2033
876	serspaegttrnseia	2037
1493	perssaegttrnseia	2325

Figure 8. Output of BoggleBoardHunter

This brief search achieved a maximum score of 2325 words on a single board with the board below:

P	E	R	S
S	A	E	G
T	T	R	N
S	E	I	A

This score far exceeds the 889 words found in the earlier board, and even so, it is not the best possible board. There are other search techniques that could possible give better performance – the search took 43.67 seconds on my computer – but this technique gives us some good estimates of more word-dense boards.

Appendix: MATLAB Code

```
format shortG

lexsize = [5757 19912 45356 80612];
SimpleLexTimes = [0.005 0.010 0.015 0.016;       %iter time
                  0.003 0.003 0.008 0.010;       %word time
                  0.008 0.032 0.027 0.039];       %pref time

BinaryLexTimes = [0 0 0.008 0.013;
                   0 0 0 0;
                   0.011 0.08 0.085 0.107];

TrieLexTimes   = [0.068 0.162 0.115 0.154;
                   0.002 0 0.001 0.03;
                   0.005 0.013 0.018 0.03];



numTrials = [10 50 100 300 500 1000 10000];
maxWords = [205 410 423 502 502 889 889];

LexiconAutoPlayerTimes = [2.39 11.988 24.61 67.62 172.05 287.22
7121.176;       %Simple Lexicon
                          3.213 17.852 32.786 92.267 172.049 313.814
3227.695;  %Trie Lexicon
                          2.409 12.352 25.187 73.009 161.684 235.656
2471.909]; %Binary Search Lexicon
BoardAutoPlayerTimes =   [0.132 0.583 0.964 0.99 1.381 2.144 15.309;
                          0.099 0.49 0.961 2.124 3.963 3.678 12.79;
                          0.11 0.506 0.993 2.167 4.52 1.752 13.132];




%%Compare lexicons, 3 graphs - iter performance, word performance, pref
performance
figure(1);
clf;
```

```
plot(lexsize, SimpleLexTimes(1,:), 'k-o', lexsize, BinaryLexTimes(1,:),
'b-s', lexsize, TrieLexTimes(1,:), 'r-h');
xlabel('Size of lexicon (words)');
ylabel('Run time of lexicon iteration (s)');
title('Plot of lexicon size vs iteration time from LexiconBenchmark');
legend('SimpleLexicon', 'BinarySearchLexicon', 'TrieLexicon', 0);
print -djpeg -r300 LexIterTimes


figure(2);
clf;
plot(lexsize, SimpleLexTimes(2,:), 'k-o', lexsize, BinaryLexTimes(2,:),
'b-s', lexsize, TrieLexTimes(2,:), 'r-h');
xlabel('Size of lexicon (words)');
ylabel('Run time of word check (s)');
title('Plot of lexicon size vs word check time from LexiconBenchmark');
legend('SimpleLexicon', 'BinarySearchLexicon', 'TrieLexicon', 0);
print -djpeg -r300 LexWordTimes


figure(3);
clf;
plot(lexsize, SimpleLexTimes(3,:), 'k-o', lexsize, BinaryLexTimes(3,:),
'b-s', lexsize, TrieLexTimes(3,:), 'r-h');
xlabel('Size of lexicon (words)');
ylabel('Run time of prefix check (s)');
title('Plot of lexicon size vs prefix check time from
LexiconBenchmark');
legend('SimpleLexicon', 'BinarySearchLexicon', 'TrieLexicon', 0);
print -djpeg -r300 LexPrefTimes


%%Compare Autoplayers
figure(4);
clf;
semilogx(numTrials, LexiconAutoPlayerTimes(1, :), 'k-o', numTrials,
LexiconAutoPlayerTimes(2, :) , 'b-s', numTrials,
LexiconAutoPlayerTimes(3, :), 'r-h');
xlabel('Number of boards generated (semilog scale)');
ylabel('Run time (s)');
title('Runtime of LexiconFirstAutoPlayer for different lexicon types');
legend('SimpleLexicon', 'BinarySearchLexicon', 'TrieLexicon', 0);
print -djpeg -r300 LexFirstAutoPlayer

figure(5);
clf;
semilogx(numTrials, BoardAutoPlayerTimes(1, :), 'k-o', numTrials,
BoardAutoPlayerTimes(2, :) , 'b-s', numTrials, BoardAutoPlayerTimes(3,
:), 'r-h');
xlabel('Number of boards generated (semilog scale)');
ylabel('Run time (s)');
title('Runtime of BoardFirstAutoPlayer for different lexicon types');
legend('SimpleLexicon', 'BinarySearchLexicon', 'TrieLexicon', 0);
print -djpeg -r300 BoardFirstAutoPlayer
```

```
%%Use average irrespective of lexicon - plot full power fit and both
values on
%%same graph
LexiconAutoPlayerAvgs = mean(LexiconAutoPlayerTimes, 1);
BoardAutoPlayerAvgs = mean(BoardAutoPlayerTimes, 1);

range = logspace(log10(min(numTrials)), log10(max(numTrials)), 100);

PowerFit = @(coefs, x) coefs(1) * x.^(coefs(2));

fSSR = @(coefs, x, y) sum((y - PowerFit(coefs, x)).^2);

[LexCoefs, LexSr] = fminsearch(@(VarCoefs) fSSR(VarCoefs, numTrials,
LexiconAutoPlayerAvgs), [1, 1]);
[BoardCoefs, BoardSr] = fminsearch(@(VarCoefs) fSSR(VarCoefs,
numTrials, BoardAutoPlayerAvgs), [1, 1]);


%Plot Lexicon and power fit
figure(6);
clf;
loglog(numTrials, LexiconAutoPlayerAvgs, 'b-s', range,
PowerFit(LexCoefs, range), 'b-.');
%Plot Board and power fit
hold on
loglog(numTrials, BoardAutoPlayerAvgs, 'r-s', range,
PowerFit(BoardCoefs, range), 'r-.');
hold off


LexTime = [PowerFit(LexCoefs, 10000), PowerFit(LexCoefs, 1000000)]
BoardTime = [PowerFit(BoardCoefs, 10000), PowerFit(BoardCoefs,
1000000)]

LexLegend = ['Power fit y = ', num2str(LexCoefs(1), '%2.4f'), 'x^{',
num2str(LexCoefs(2), '%2.4f'), '} of LexiconFirst'];
BoardLegend = ['Power fit y = ', num2str(BoardCoefs(1), '%2.4f'),
'x^{', num2str(BoardCoefs(2), '%2.4f'), '} of BoardFirst'];
legend('LexiconFirstAutoPlayer average times', LexLegend,
'BoardFirstAutoPlayer average times', BoardLegend, 0);
xlabel('Number of boards generated (log scale)')
ylabel('Run time (s, log scale)');
title({'Comparison of BoardFirstAutoPlayer and
LexiconFirstAutoPlayer','With least-squares power fits'});
print -djpeg -r300 AutoPlayerComparison


%%Now, try a power fit with the exponent clamped at 1 - equivalent to a
linear fit
%%with the y-intercept clamped at zero
fSSRLinear = @(a, x, y) sum((y - PowerFit([a, 1], x)).^2);
[LexA, LexSr2] = fminsearch(@(avar) fSSRLinear(avar, numTrials,
LexiconAutoPlayerAvgs), 1);
[BoardA, BoardSr2] = fminsearch(@(avar) fSSRLinear(avar, numTrials,
BoardAutoPlayerAvgs), 1);
```

```
%Plot Lexicon and power fit
figure(7);
clf;
loglog(numTrials, LexiconAutoPlayerAvgs, 'b-s', range,
PowerFit([LexA,1], range), 'b-.');
%Plot Board and power fit
hold on
loglog(numTrials, BoardAutoPlayerAvgs, 'r-s', range, PowerFit([BoardA,
1], range), 'r-.');
hold off

LexLegend = ['Power fit y = ', num2str(LexA, '%2.4f'), 'x^{1} of
LexiconFirst'];
BoardLegend = ['Power fit y = ', num2str(BoardA, '%2.4f'), 'x^{1} of
BoardFirst'];
legend('LexiconFirstAutoPlayer average times', LexLegend,
'BoardFirstAutoPlayer average times', BoardLegend, 0);
xlabel('Number of boards generated (log scale)')
ylabel('Run time (s, log scale)');
title({'Comparison of BoardFirstAutoPlayer and
LexiconFirstAutoPlayer','Power fits with exponent clamped to 1'});
print -djpeg -r300 LinearAutoPlayerComparison

LexTimeLinear = [PowerFit([LexA,1], 10000), PowerFit([LexA,1],
1000000)]
BoardTimeLinear = [PowerFit([BoardA,1], 10000), PowerFit([BoardA,1],
1000000)]
```