Project 3 (Chess) Write-Up

Project Enjoyment

- How Was Your Partnership?

```
Our partnership went well, but Jayden excelled as a programmer and
partner in this project!
  He appeared to grasp the concepts very easily, yet he was always ready
to slow down and explain
  them to me.
  It went well like always. Working with Aeron these past three projects
have been amazing! Going
  from strangers prior to P1 to friends now in P3 has been great! She
informs me when I make a
  style issue and how to improve it. She has been a fantastic partner
this quarter, with her help
  I have learned a lot from this class.
```

- What was your favorite part of the project?

```
  My favorite aspect of this project was viewing the progress we made
when running EasyChess. It
  is super exciting to see the applications of program in a wider setting
than uMessage. It was
  also interesting to see how chess bots are popular topics in the real
world! When researching
  ideas for improving Jamboree, it was evident that many people take this
seriously :)
  My favorite part was understanding how parallelism works! I was a
little bit confused on how
  the project was going to work out with it especially with Jamboree. I
was running through the
  pseudocode and out code in my head one night trying to make sense of it
when it finally
  clicked in the middle of the night. I was so sure of it I got out of
bed and wrote down the
  idea before I forgot it. When I woke up in the morning I implemented it
and saw that it passed!
  Just the surreal feeling of having something pass on the first try from
working on it other
  than staring and hoping is the best feeling.
```

- What was your least favorite part of the project?

```
  My least favorite aspect of the project was that it seemed like each of
the bots were fairly
  similar to each other. In comparison to projects 1 and 2, this Project
```

```
3 seemed less applicable
  to a variety of computer science fields. Expanding on this, the ability
to write and fully
  understand data structures is very fulfilling for me and the connection
to other classes is
  very clear. Writing chess bots seemed like taking similar algorithms
and ideas and putting
  them into code. Overall, this was less enjoyable for me :P
  Least favorite part of the project was running the test. Dealing with
the long wait and
  confusion of GCP troubled me. I wanted to run two instances and
realized that the cap was
  at 24 when I wanted to run two instances of 16 CPUs'. This led to me to
make a a new
  instance in the uw-west server and run the test from there. This caused
issues later on
  when the data was inconclusive with my partners. This led to re-running
the test on the
  same server. It was a mistake I made that I paid dearly for. :(
```

- How could the project be improved?

```
I might suggest some more guidance on using the server. The handout was
super helpful,
  some settings were chosen and what that means. I also spent a little
too long on changing
  the CPU limit because I was changing it globally, instead of changing
it for a particular
  location. This caused me to not be able to run the 32 CPU tests for
some time.
  Also perhaps some more guidance on the provided tests and how they are
applicable
  to the writeup.
```

- Did you enjoy the project?

```
  I personally really enjoyed implementing data structures because I felt
I got a deeper
  understanding of something crucial to the foundations of  computer
science. I'm not sure I
  gained the same amount of knowledge in this project. I feel like all of
the searchers were
  fairly similar in essence, so I feel like this project was more about
understanding the
  smaller details in searchers.
  I really enjoyed working with this project. I managed to talk about it
at one of my
  interviews and the interviewer was really interested about it. I talked
about game
```

```
  theory and each of the implementations. This led to a long discussion
where we
  both discussed parallelism. I really enjoyed being able to talk about a
project
  and idea outside of the classroom.
```

The Chess Server

- When you faced Clamps, what did the code you used do? Was it just your jamboree? Did you do something fancier?

```
When we faced Clamps, we used a derivative of Jamboree. Each time the
board generated new moves we would sort them.
  To do this we would apply the move to the board, evaluate the board,
then undo the move. We would then put them in a
  HashMap,the key being the integer associated with the evaluated value,
and the value being the move itself.
  From there a new list was made. We would iterate over the HashMap and
find the max value and then add that move to the new
  list of moves while removing it from the HashMap. To do this we used an
iterator to go through and remove the objects.
  In the end, we would return the combination of the new best Move list
and the old move left over in the HashMap.
  This would allow for faster alphaBeta pruning since the better move
value would be looked at first. Also, in the iteration
  of applying the move, if the move was a promotion, we would add an
extra 800 to that value. This is because a pawn
  normally has  a value of 100 when it could promote it would have a
value of a Queen, 900. By adding an 800 we make the
  move more of a priority.
  In the future, I would like to add it to register if it was a capture
move and if so, what would it capture.
  If the capture was a queen, make that move more of a priority
```

- Did you enjoy watching your bot play on the server? Is your bot better at chess than you are?

```
Yes, we had a lot of fun with the server! We even had some intense
moments watching our
  bot play against clamps, especially when the game appeared to be close.
  I haven't played chess in a very long time, so I would say bot is
better than me! It is
  certainly faster at making decisions than I am, which could be an
important element to
  the success of a serious chess player.
```

- Did your bot compete with anyone else in the class? Did you win?

```
We had a very low success rate in bots accepting offers to play. This
unfortunately
  made it difficult to compare our bots to our classmates. Each time I
would send someone
  a game invite who was online, they wouldn't respond. I would then check
to see if they
  were online again and they weren't! I wasn't sure if I spooked them or
not but so far no
  one has accepted a game invite.
  Update: I did manage to play against one team. It was fun to watch our
bots play
  against one another, they made the same moves so the board was mirrored.
I implemented
  move ordering so our timing was faster compared to their movements. We
would both
  always end up in a stalemate.
```

- Did you do any Above and Beyond? Describe exactly what you implemented.

```
We made the JamboreeDerivative as described above, but didn't have time
make
  it beat flexo or bender :(
```

Experiments

Chess Game

Hypotheses

Suppose your bot goes 3-ply deep. How many game tree nodes do you think it explores (we're looking for an order of magnitude) if: - ...you're using minimax?

```
The branching factor of a game tree is the number of children,
    or moves, a node has. From the games handout, the average branching
factor
    in chess is approximately 35. Following the logic from the games
handout, if
    we look 3-ply deep, we explore 35^5, or 42875, game tree nodes. This
is about
    4 * 10^4, so our estimated order of magnitude is 4.
```

- ...you're using alphabeta?

```
    Since alphabeta prunes the game tree, we can expect the number of
```

```
game tree nodes
    visited to be smaller than minimax. In the worst case, alphabeta
won't prune any
    branches, and we would expect to visit the same number of game tree
nodes as regular
    minimax. In the best case, when the moves are sorted, we can expect
alphabeta prune
    half of the branches resulting in a runtime of 35^{3/2}. Our
alphabeta does not
    implement sorting, so we can expect something in between the best and
worst case.
    From this information, we determine our estimate to become 35^{3/1.5}
which is 1225.
    This is about 1 * 10^3, so our estimated order of magnitude is 3.
```

Results

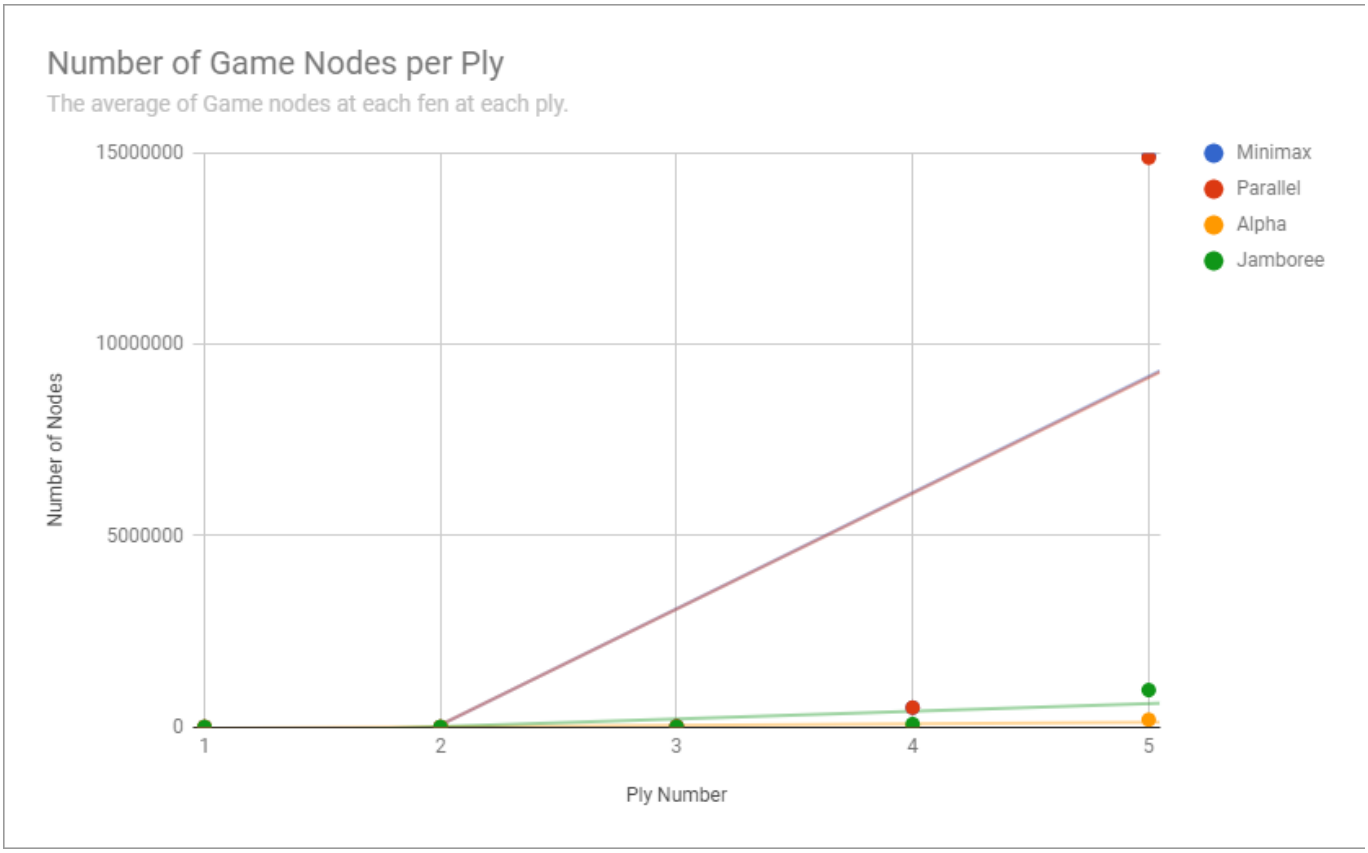
Run an experiment to determine the actual answers for the above. To run the experiment, do the following: 1. Run SimpleSearcher against AlphaBetaSearcher and capture the board states (fens) during the game. To do this, you'll want to use code similar to the code in the testing folder. 2. Now that you have a list of fens, you can run each bot on each of them sequentially. You'll want to slightly edit your algorithm to record the number of nodes you visit along the way. 3. Run the same experiment for 1, 2, 3, 4, and 5 ply. And with all four implementations (use ply/2 for the cut-off for the parallel implementations). Make a pretty graph of your results (link to it from here) and fill in the table here as well:

```
In creating our tests, we followed the suggestions above by creating a
class called GetListOfFens
to play SimpleSearcher against AlphaBetaSearcher and store all fens in a
list. We then created
modified versions of each Searcher and implemented
static fields to store a list nodes visited during each call to getMove.
We use these
features in our test class, CountGameTreeNodes. We create our list of
fens, then
calculate the average of all game tree nodes visited per call to
getBestMove for
each value of ply from 1 through 5 inclusive. Note that with the parallel
searchers,
ParallelSearcher and JamboreeSearcher, we use ply/2 for the cut-off. Our
data is placed
in the tables below.
```

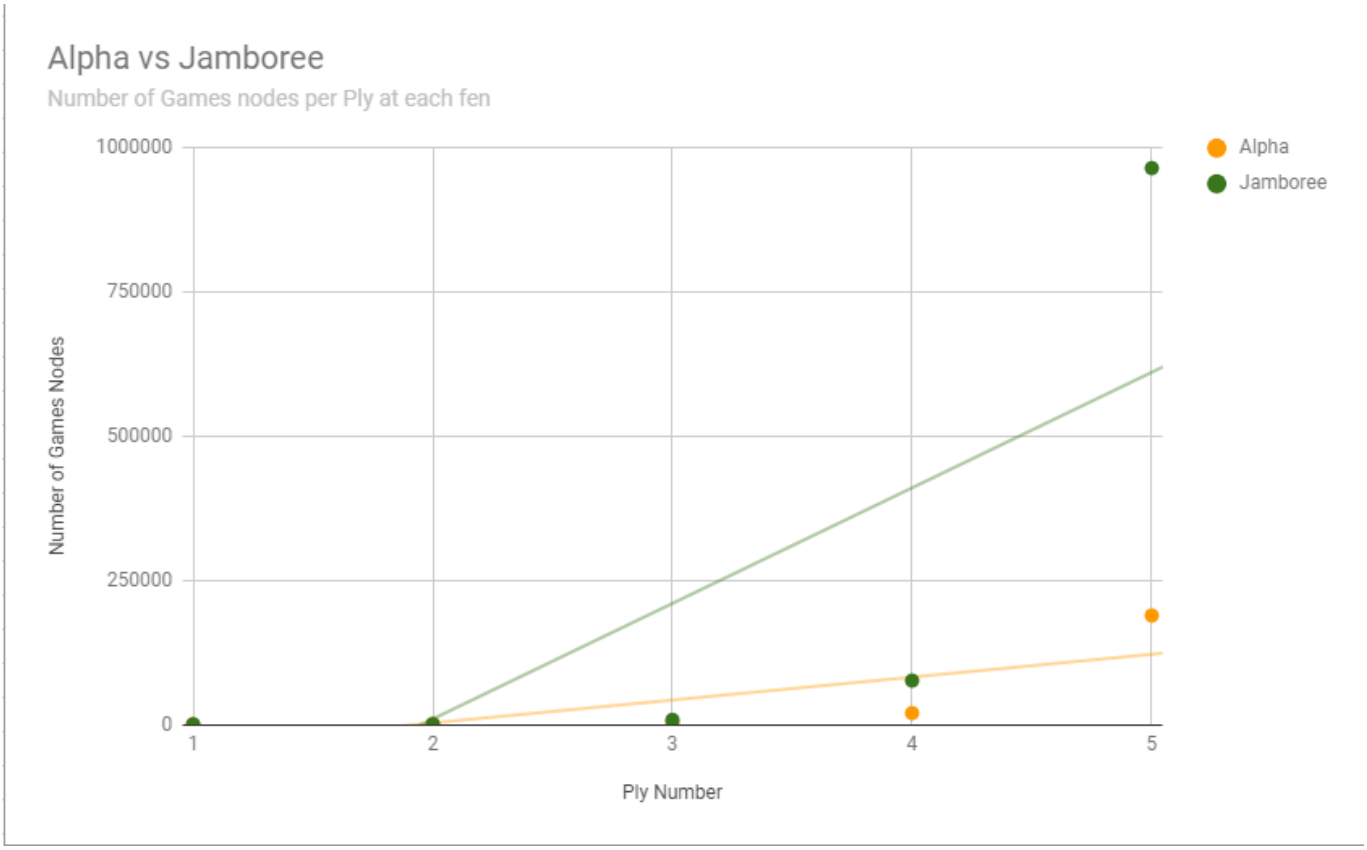
Algorithm	1-ply	2-ply	3-ply	4-ply	5-ply
Minimax	23.3333	638.6666	17266.705	505681.353	14938126.137
Parallel Minimax	23.3333	650.7451	17886.901	506830.862	14883749.725
Alphabeta	23.3333	212	2244.8823	19516.9216	188862.627

Algorithm	1-ply	2-ply	3-ply	4-ply	5-ply
Jamboree	23.33333	458.1176	7545.9411	75865.2352	965313.9412

We also created a graph of the data, shown below.



Note that the purple line represents the Minimax and Parallel. Initially, Minimax was blue, and Parallel was red.
But the overlap causes a purple line. This shows that they follow the same trend in counting nodes.



Conclusions

How close were your estimates to the actual values? Did you find any entry in the table surprising? Based ONLY on this table, do you feel like there is a substantial difference between the four algorithms?

```
We estimated Minimax would visit 42875 game tree nodes, while it actually
visited
about 17267 nodes. This gives us a difference is 25608, which is rather
large. We
can attribute this to the average branching factor of 35 being slightly
different
for the game we played. We estimated Alphabeta would view 1225 game tree
nodes, while
it actually visited about 2245 nodes. This gives means we overestimated
by 1020.
Ultimately, these estimations were not too accurate, but they were based
on other
averaged values. One positive note to point out is that our estimations
produced
correct orders of magnitude for both Minimax and Alphabeta!

We are not too surprised by the data; we expected Minimax and Parallel
Minimax to
visit the same number of nodes, because Parallel Minimax uses parallelism
to decrease
the amount of time used. The slight differences in the results between
```

```
the two
algorithms are accounted for by rounding errors in averaging the
measurements for
each ply. If you view the graph "Number of Game Nodes per Ply" the
trendlines
for both Minimax and Regular Minimax are exactly the same. In fact, they
overlap,
creating a purple trendline. This supports our prediction in that Minimax
and
Parallel Minimax visit the same amount of nodes.

Looking at this table alone, there is not much difference between
algorithms at depths
1, 2, and even 3. We can see the number of nodes Minimax and Parallel
Minimax visit
starts to increase rapidly as we approach depths 4 and 5, compared to
Alphabeta and
Jamboree searchers, whose number of nodes still increases at a slower
rate. However,
with only 5 ply, there is not a surplus of concrete evidence to support a
definitive
substantial difference between the four algorithms. From what we know
from implementing
the algorithms and viewing the graphs, we can start to make educated
predictions as
to what trend the algorithms might follow.
```

Optimizing Experiments

THE EXPERIMENTS IN THIS SECTION WILL TAKE A LONG TIME TO RUN. To make this better, you should use Google Compute Engine: * Run multiple experiments at the same time, but **NOT ON THE SAME MACHINE**. * Google Compute Engine lets you spin up as many instances as you want.

Generating A Sample Of Games

Because chess games are very different at the beginning, middle, and end, you should choose the starting board, a board around the middle of a game, and a board about 5 moves from the end of the game. The exact boards you choose don't matter (although, you shouldn't choose a board already in checkmate), but they should be different.

```
We selected three board states from the list of fen in the previous part.
They are listed below:
- Beginning: rnbqkbnr/pp1ppppp/8/2p5/8/2N5/PPPPPPPP/R1BQKBNR w KQkq c6
- Middle: 2kr3r/pp5p/2n1p1p1/2pp1p2/5B2/2qP1QP1/P1P2P1P/R1R3K1 b Hh -
- End: 2k3r1/p6p/2n5/3pp3/1pp1P3/2qP4/P1P1K2P/R1R5 b Hh -
```

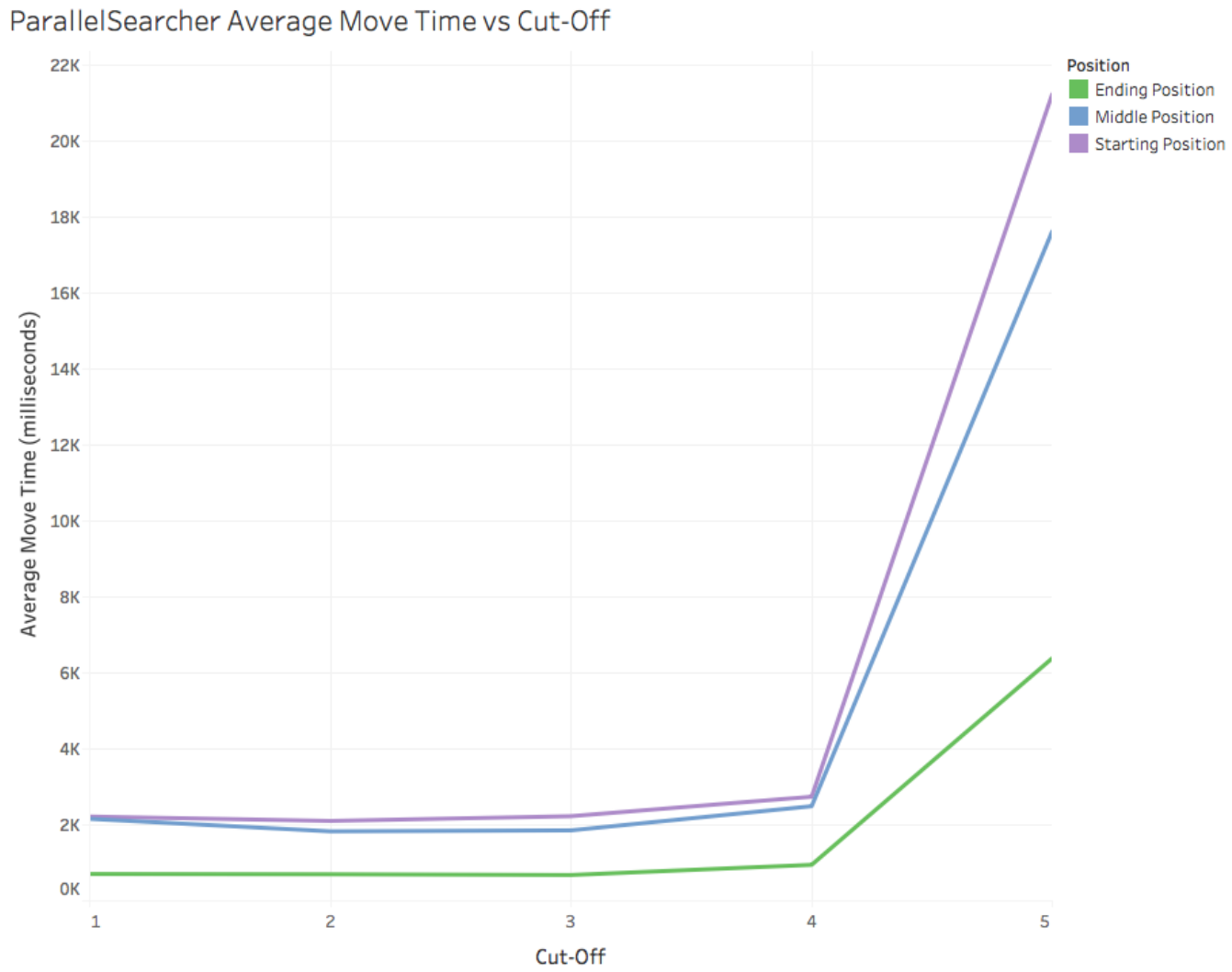
Sequential Cut-Offs

Experimentally determine the best sequential cut-off for both of your parallel searchers. You should test this at depth 5. If you want it to go more quickly, now is a good time to figure out Google Compute Engine. Plot your results and discuss which cut-offs work the best on each of your three boards.

```
We developed our experiment according to the guidelines suggested. Using
the
three board states above, we played ParallelSearcher with a depth of 5
against the
SimpleSearcher with a depth of 2 and cut-off 1. We modified the
ParallelSearcher
cut-off values, using values 1 through 5. We recorded the time it took
for
ParallelSearcher to find each new move, throwing out the first five moves
to decrease
the effect of the JVM warmup, and took the average of all moves in the
game to
present a single averaged move time. We repeated this test for each
cut-off 15 times
and took the average. Then, we repeated this process for each board state
(beginning,
middle, and end) and for Jamboree. Below are tables of our measurements
and graphs
representing the data.
```

ParallelSearcher

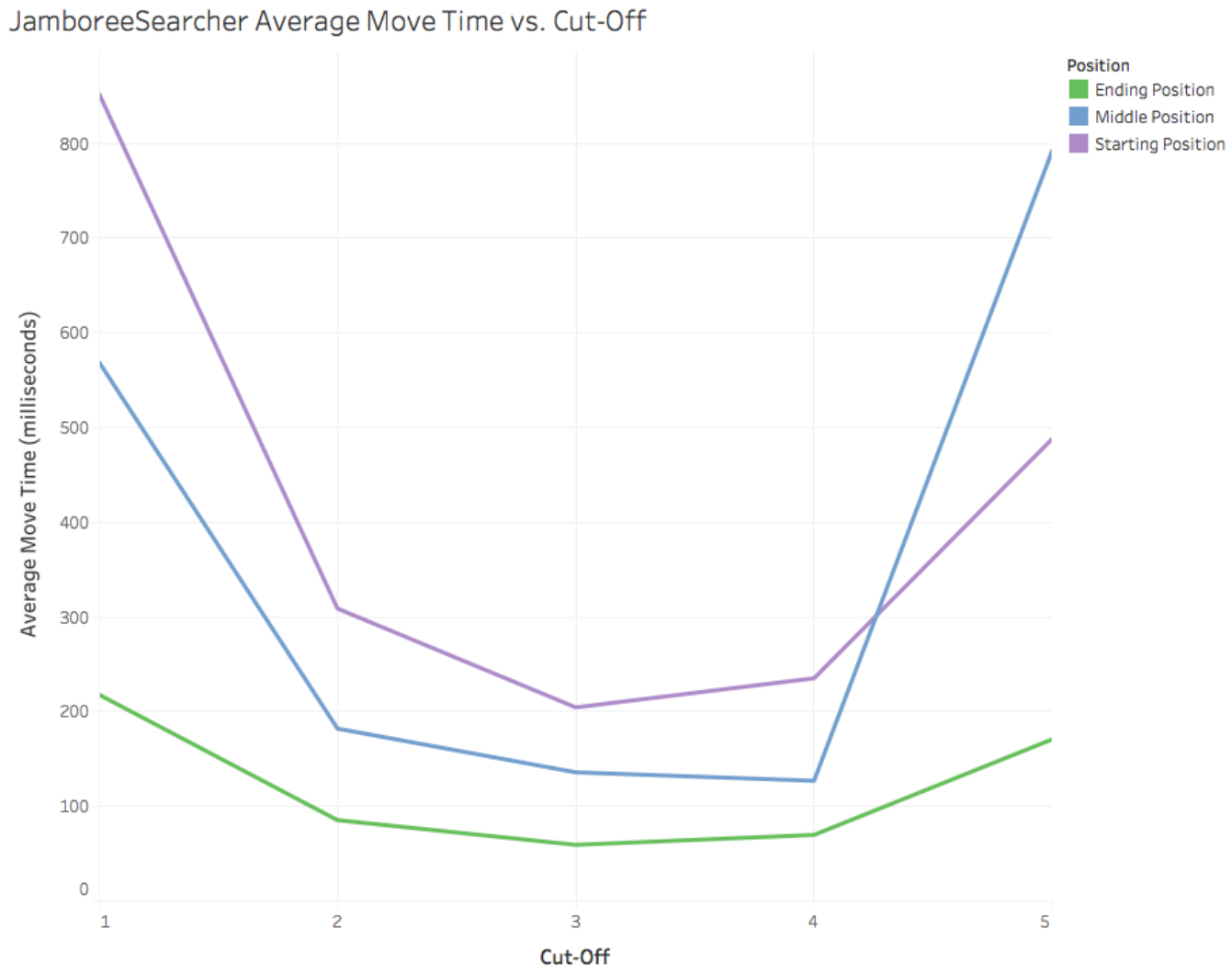
| Cut-Off | Starting Position | Middle Position | Ending Position | | :----------: |:----------------:|:---------------:|:---------------:| | 1 | 2221.98611111111 | 2162.297619047 | 710.36666666667 | | 2 | 2107.4125 | 1833.514285714 | 703.38333333333 | | 3 | 2232.147222223 | 1858.540476190 | 682.91666666666 | | 4 | 2744.4638889 | 2494.25 | 951.55 | | 5 | 21244.5333333 | 17632.88333334 | 6390.7333333334 |



The trend of sum of Average Move Time for Cutoff. Color shows details about Position.

JamboreeSearcher

| Cut-Off | Starting Position | Middle Position | Ending Position | | :----------: |:----------------:|:--------------:|:--------------:| | 1 | 852.102777777779 | 568.8087985769 | 217.8833333333 | | 2 | 308.848611111115 | 182.09866350422 | 85.48333333333 | | 3 | 204.524999999998 | 135.97709785318 | 59.4 | | 4 | 235.258333333333 | 127.01746031746 | 69.88333333333 | | 5 | 488.056944444447 | 792.34523809523 | 170.8166666666 |



The trend of sum of Average Move Time for Cutoff1. Color shows details about Position.

Viewing the ParallelSearcher graph, we can see the runtimes of each test in beginning, middle
and ending position decrease at a relatively low rate as the cut-off value increases until
reaching the local minimum. Then there is a massive increase when the cut-off is 5. From
analyzing the ParallelSearcher algorithm, this makes sense, as ParallelSearcher is essentially
completing Minimax in parallel. The higher the cut-off value, the more game tree nodes must
be processed through minimax. This means when we run ParallelSearcher with a depth of 5 and a
cut-off of 5, we are basically running Minimax at a depth of 5! Using the strategy to estimate
the number of game tree nodes visited, the difference between cut-off of 4 and cut-off of 5 is
35^5 - 35^4, which is 51,021,250. This explains the rapid increase of runtime, a fact that
became evident when it took an unreasonably long amount of time to complete this test.
We see something slightly different occurring with the JamboreeSearcher

```
tests. From viewing the
JamboreeSearcher graph, there is a local minimum when the cut-off is 3 or
4, depending on the
initial position. The runtime for cut-off 1 and cut-off 2 are very high,
compared to the relatively
low runtime values for ParallelSearcher. This deals with finding the
right balance between
dividing nodes and processing them (which is also shown in
ParallelSearcher, though less dramatic).
At a certain point, it is faster to process nodes, rather than split them
up. This demonstrated
through the JamboreeSearcher graph and is also dependent on the number of
processors (discussed
in the next section).
A less interesting aspect to mention is that the games at ending position
generally took less
time than games at the middle position which generally took less time
than games at the
beginning position. This makes sense as there are typically less moves to
make as the game of
chess continues. The intercept between the tests at middle position and
ending position at
with a depth of 5 may be due to selecting a board state which happened to
take more time a
higher depths. Although, without more data at higher depths, we can't
draw any concrete
conclusions about this.
Ultimately, we can draw these conclusions from this data regarding the
best cut-off for each board:
```

Board Type	Parallel Best	Jamboree Best
Starting	2	3
Middle	2	4
Ending	3	3

Number Of Processors

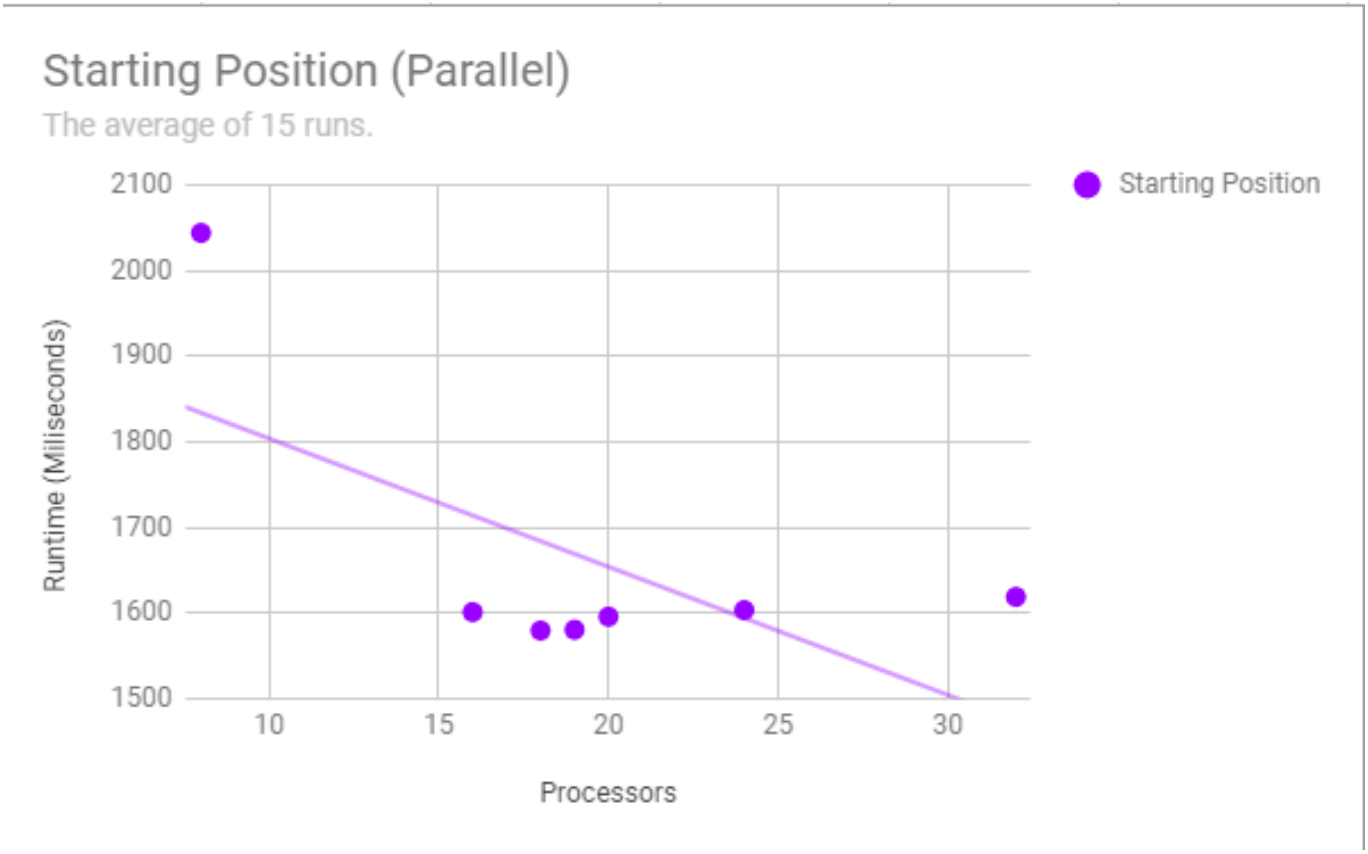
Now that you have found an optimal cut-off, you should find the optimal number of processors. You MUST use Google Compute Engine for this experiment. For the same three boards that you used in the previous experiment, at the same depth 5, using your optimal cut-offs, test your algorithm on a varying number of processors. You shouldn't need to test all 32 options; instead, do a binary search to find the best number. You can tell the ForkJoin framework to only use k processors by giving an argument when constructing the pool, e.g.,

```
ForkJoinPool POOL = new ForkJoinPool(k);
```

Plot your results and discuss which number of processors works the best on each of the three boards.

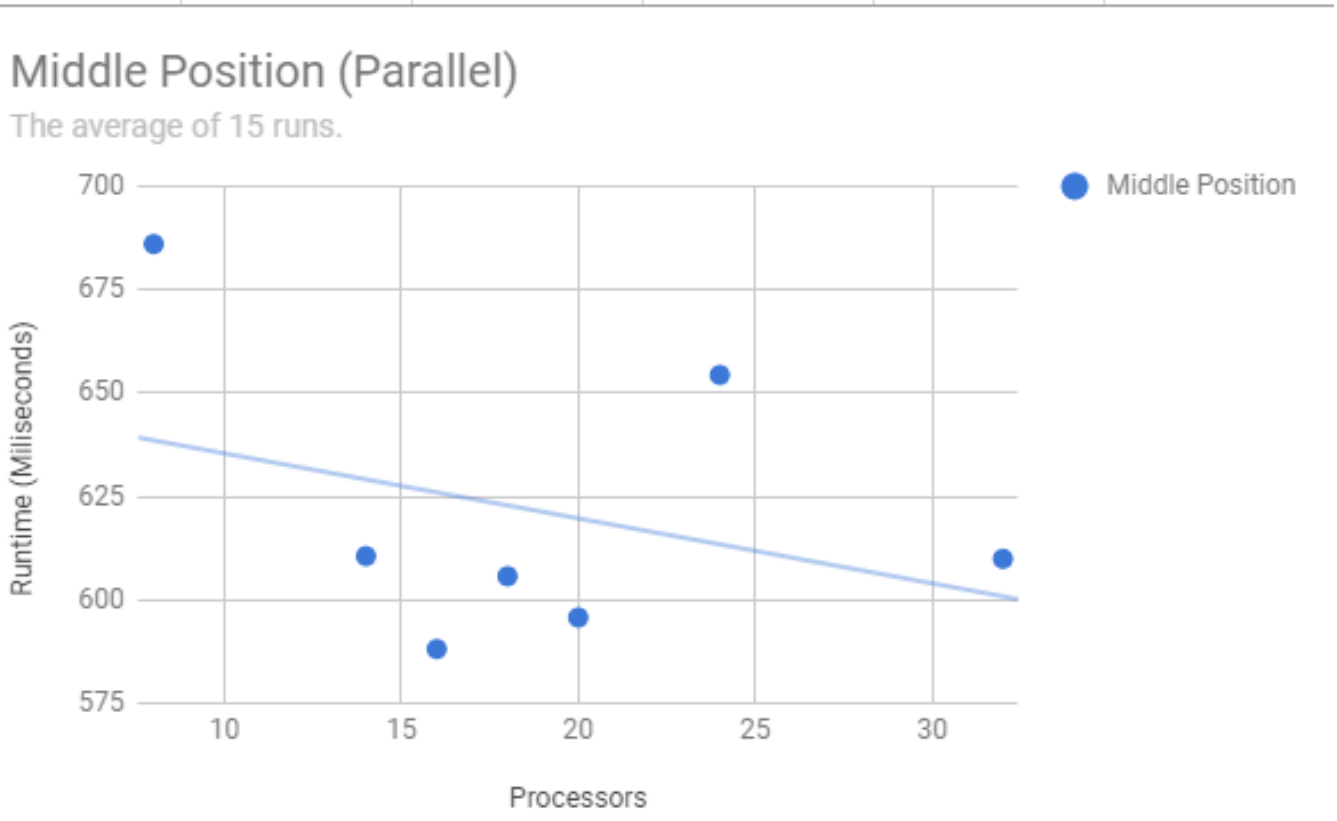
ParallelSearcher
Starting Position - Optimal Cut-Off: 3

| Number of Processors | Runtime (ms) | | :----------: | :----------: | | 8 | 2043.850794 | | 16 | 1601.066667 | | 18 | 1579.549206 |
| 19 | 1580.495238 | | 20 | 1595.714286 | | 24 | 1603.438095 | | 32 | 1618.857143 |



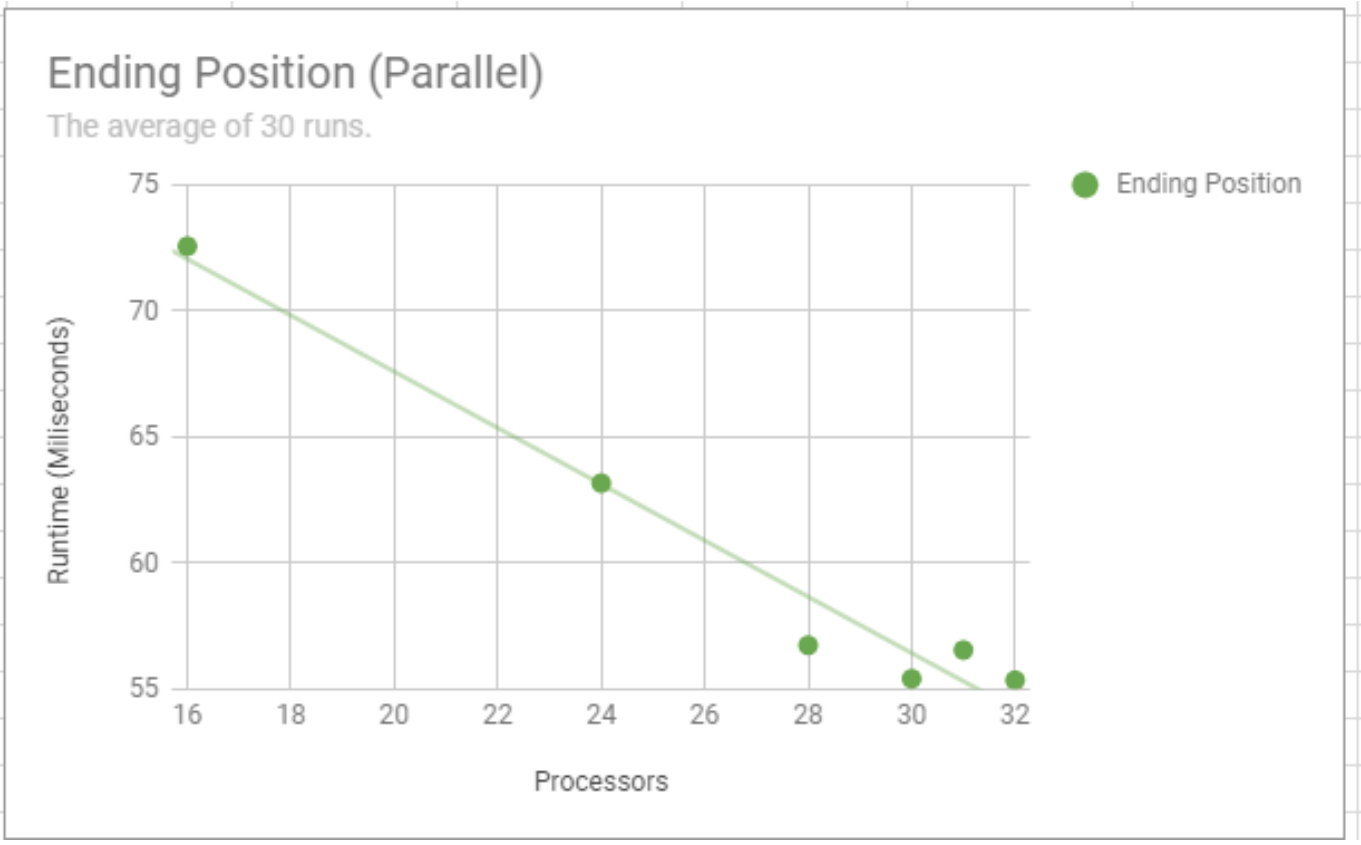
Middle Position - Optimal Cut-Off: 3

| Number of Processors | Runtime (ms) | | :----------: | :----------: | | 8 | 686.0133333 | | 14 | 610.56 | | 16 | 588.0933333 |
| 18 | 605.6933333 | | 20 | 595.72 | | 24 | 654.36 | | 32 | 609.9066667 |



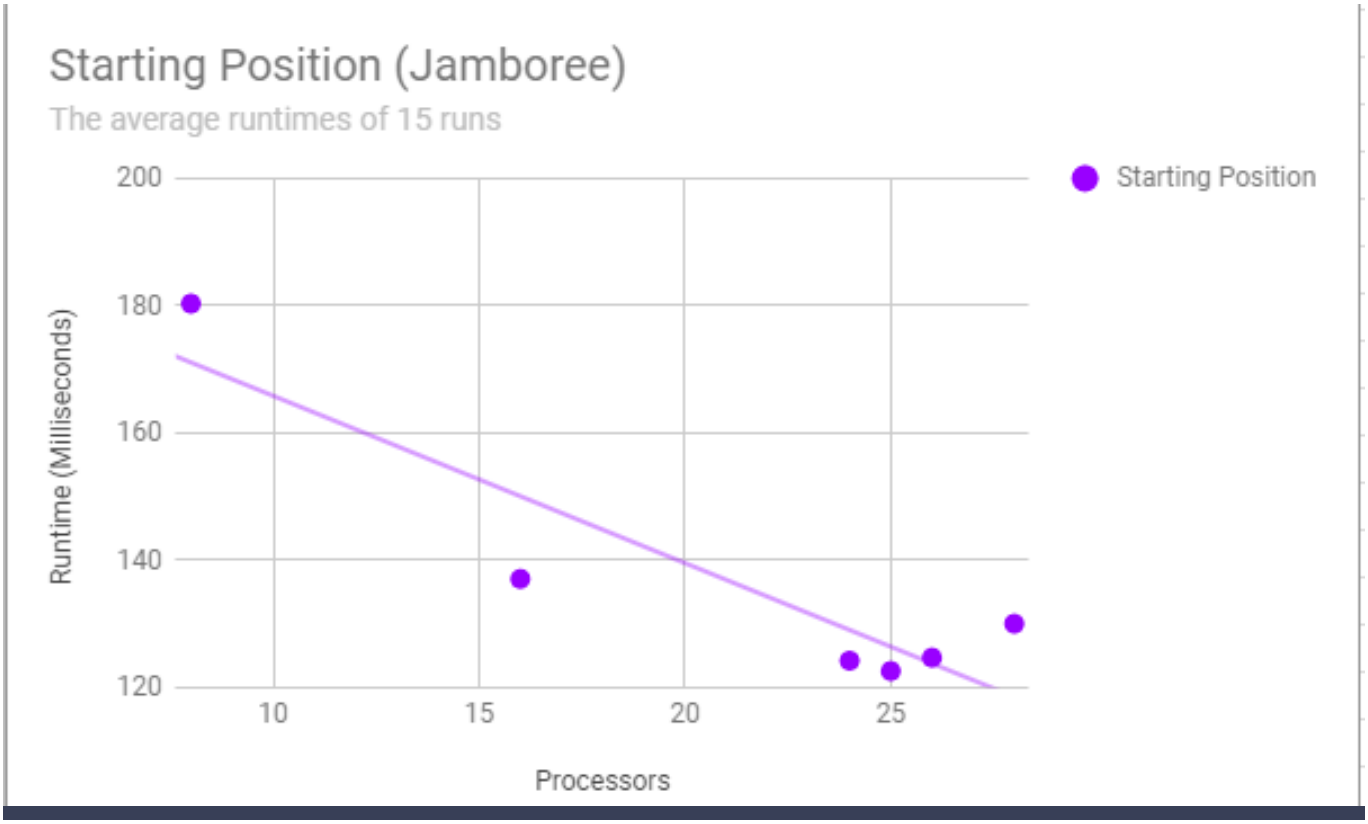

```
Ending Position - Optimal Cut-Off: 3
```

Number of Processors	Runtime (ms)
16	72.56666667
24	63.17254902
28	56.74509804
30	55.41568627
31	56.55294118
32	55.35686275



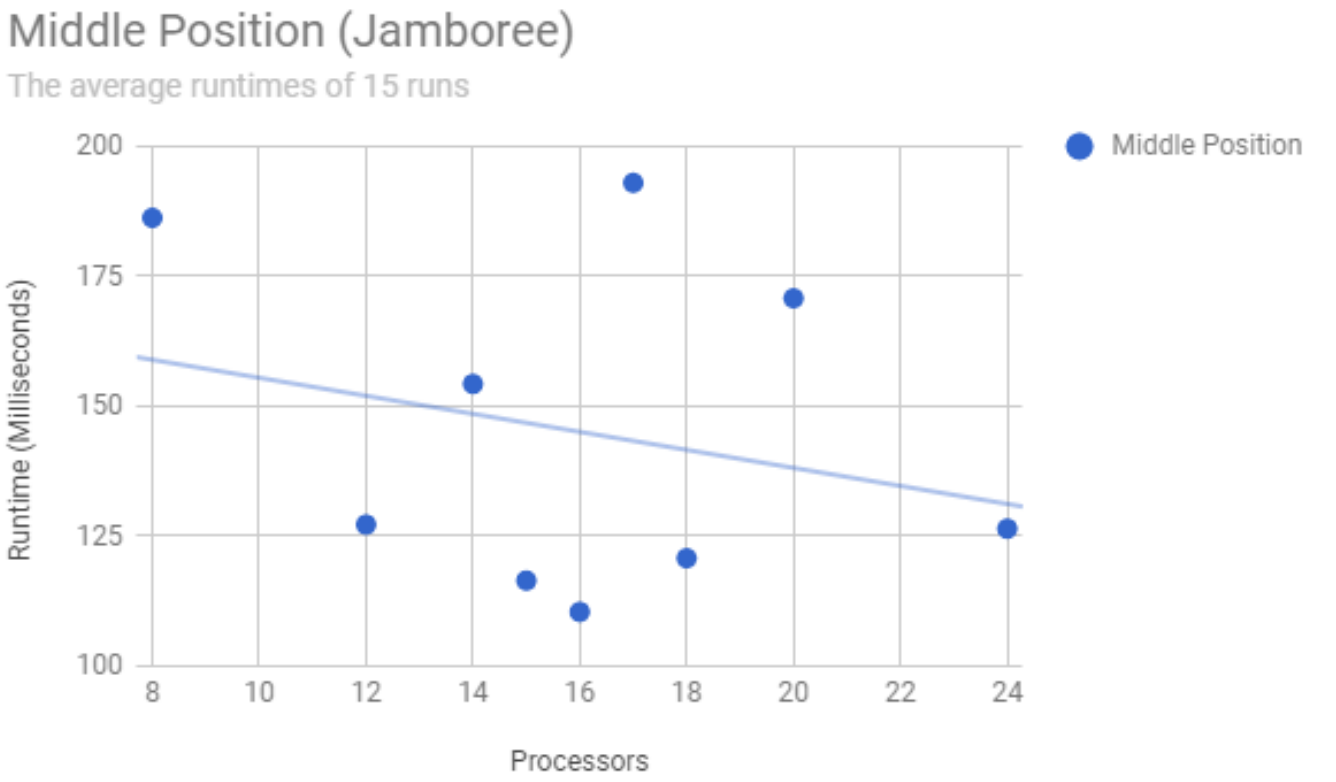
JamboreeSearcher
Starting Position - Optimal Cut-Off: 3

Number of Processors	Runtime (ms)		:----------------:	:----------:		8	180.32638888888886
16	137.07361111111112		24	124.2263888888889
25	122.6		28	130.0388888888889		26	124.7111111111111



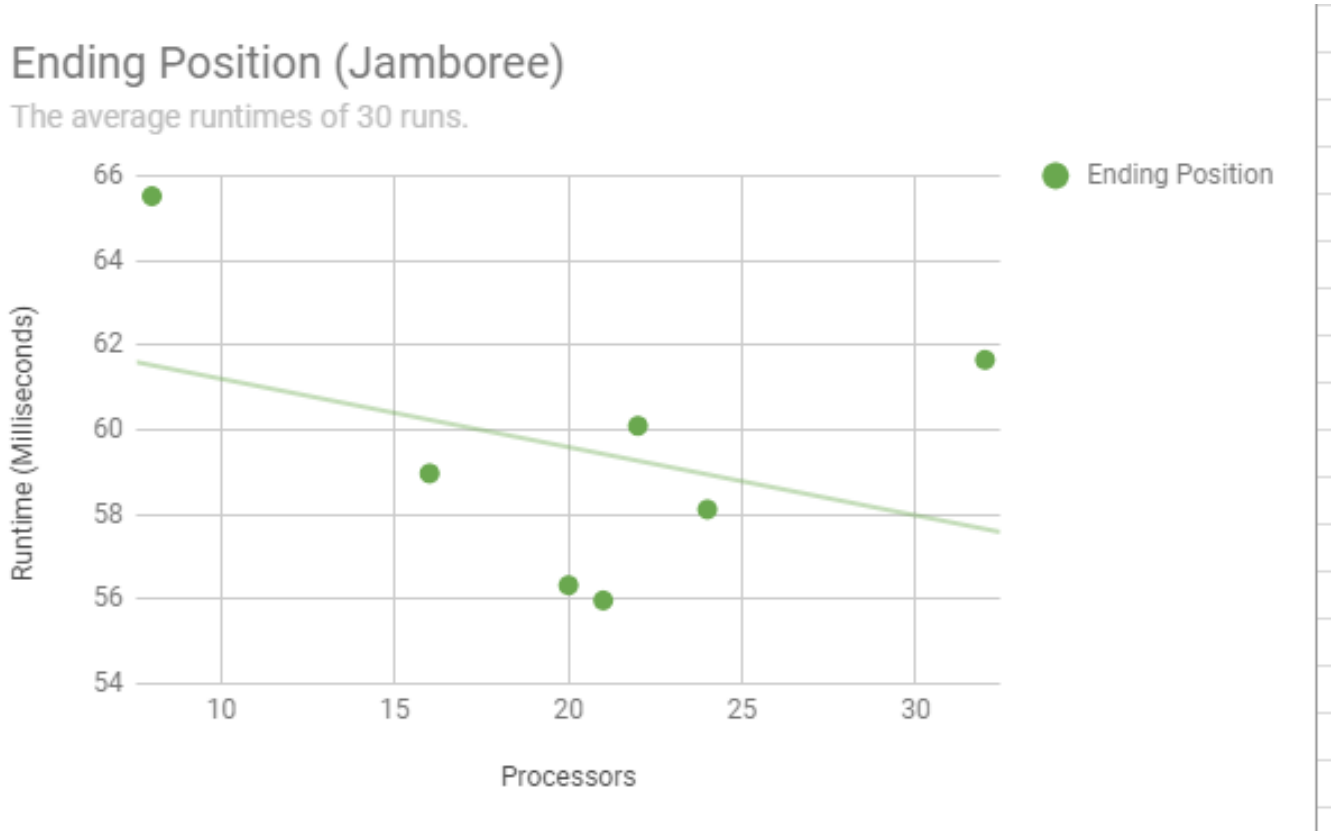
Middle Position - Optimal Cut-Off: 4

| Number of Processors | Runtime (ms) | | :----------------: | :----------: | | 8 | 186.2307619047619 | | 12 | 127.22010934744269 | | 14 | 154.28380952380948 | | 15 | 116.4362857142857 | | 16 | 110.41701587301586 | | 18 | 120.75450793650792 | | 20 | 170.76177777777778 | | 24 | 126.43771063674512 |
| 27 | 192.9733544973545 |



Ending Position - Optimal Cut-Off: 3

| Number of Processors | Runtime (ms) | | :----------------: | :----------------: | | 8 | 65.51666666666667 | | 16 | 58.96666666666667 | | 20 | 56.325 |
| 21 | 55.96666666666667 | | 22 | 60.09166666666667 | | 24 | 61.65 | | 32 | 58.11666666666667 |



To find out the best processor for each board we used a binary search from 0 to 32 to determine the optimal
amount of processors to use. Starting at 16 and checking 8 to 24 for each one after to get a good range on
where to begin. From there we could narrow it down to one singular processor that would cause the fastest
runtime for each board. In order to do this for each fen, we would use a particular searcher's optimal
cut-off to find their optimal processor. We calculated the runtime as we did in the previous experiments,
noting that we threw away the first 5 measurements to account for the JVM Warmup time.
We would find the average of 15 runs on each fen to determine the runtime to get the best move. We would later increase it to 30 for the ending position's fen to collect more
data as the runtime would be slower as it is closer to the end of the game. With the average runtime of
getting the best move per game, then finding the average of that over fifteen iterations is how we could
determine the optimal processor at each fen.
Analyzing the data from the experiments, we've constructed this concise table showing the best number of
processor for each pair of searcher and board game state.

Board Type	Parallel Processors	Jamboree Processors
Starting	18	25
Middle	16	16

Board Type	Parallel Processors	Jamboree Processors
Ending	32	21

```
We immediately notice that the best number of processors for the middle
board type is 16
for both Parallel and Jamboree searchers. However, looking at the other
best numbers of
processors, it is unclear if that is due to random chance or if there is
some pattern to
the data.
One conclusion that we can draw from this dataset is that a number of
processors greater
than 16 is generally beneficial for increased efficiency in parallelism.
Processor counts
less than 16 waste potential parallelizable code for searchers such as
ParallelSearcher and
JamboreeSearcher.
There may be some correlation, however, in relation to the best cut-off,
which was determined
in the previous section. For instance, in JamboreeSearcher, we tested
both Starting Position
and Ending Position with a cut-off of 3, and Middle Position with a
cut-off of 4. In the
table above we see that Starting and Middle states have smaller numbers
of processors (around
16 and 18), while Ending position uses all 32 processors for optimal
performance. A smaller
cut-off value means the algorithm will divide and conquer more nodes,
which means we will
have more sequential chunks of smaller sizes to process. This test helps
balance the cost
of runtime with the finding the optimal combination.
```

Comparing The Algorithms

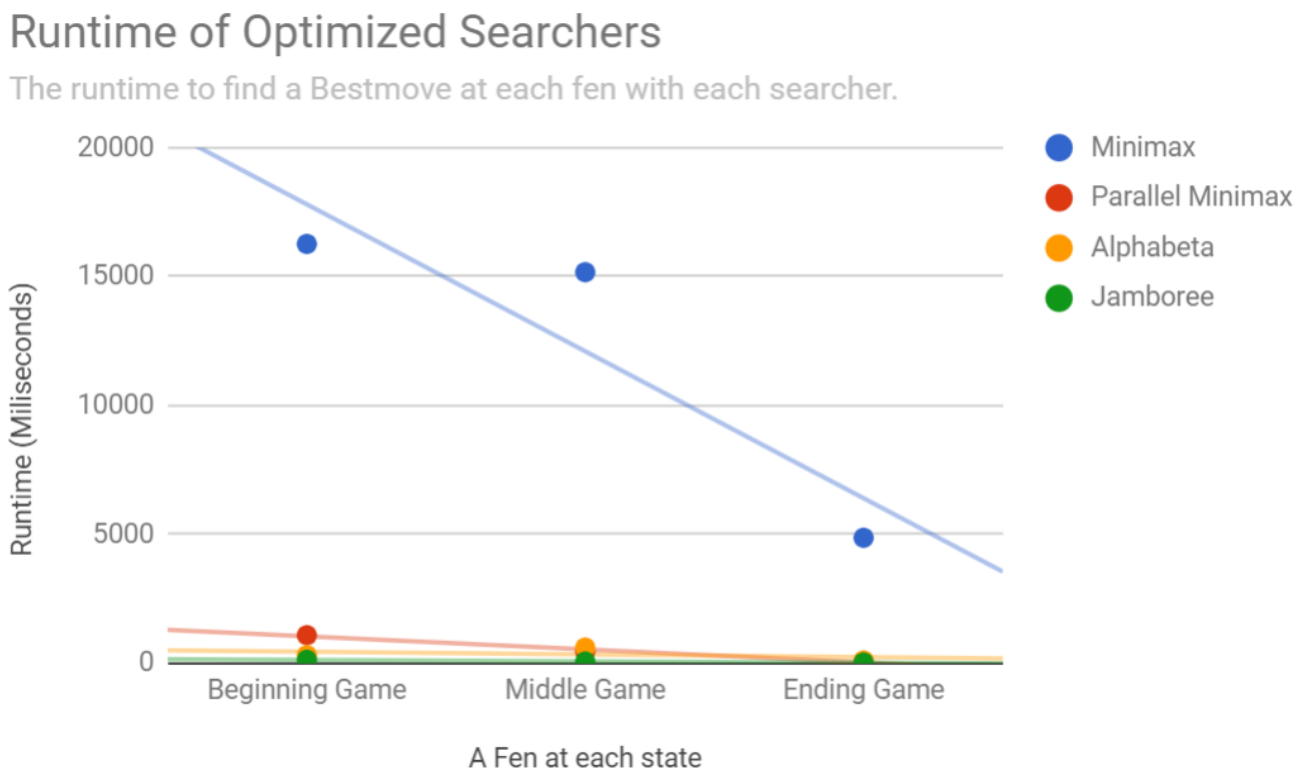
Now that you have found an optimal cut-off and an optimal number of processors, you should compare the actual run times of your four implementations. You MUST use Google Compute Engine for this experiment (Remember: when calculating runtimes using *timing*, the machine matters). At depth 5, using your optimal cut-offs and the optimal number of processors, time all four of your algorithms for each of the three boards.

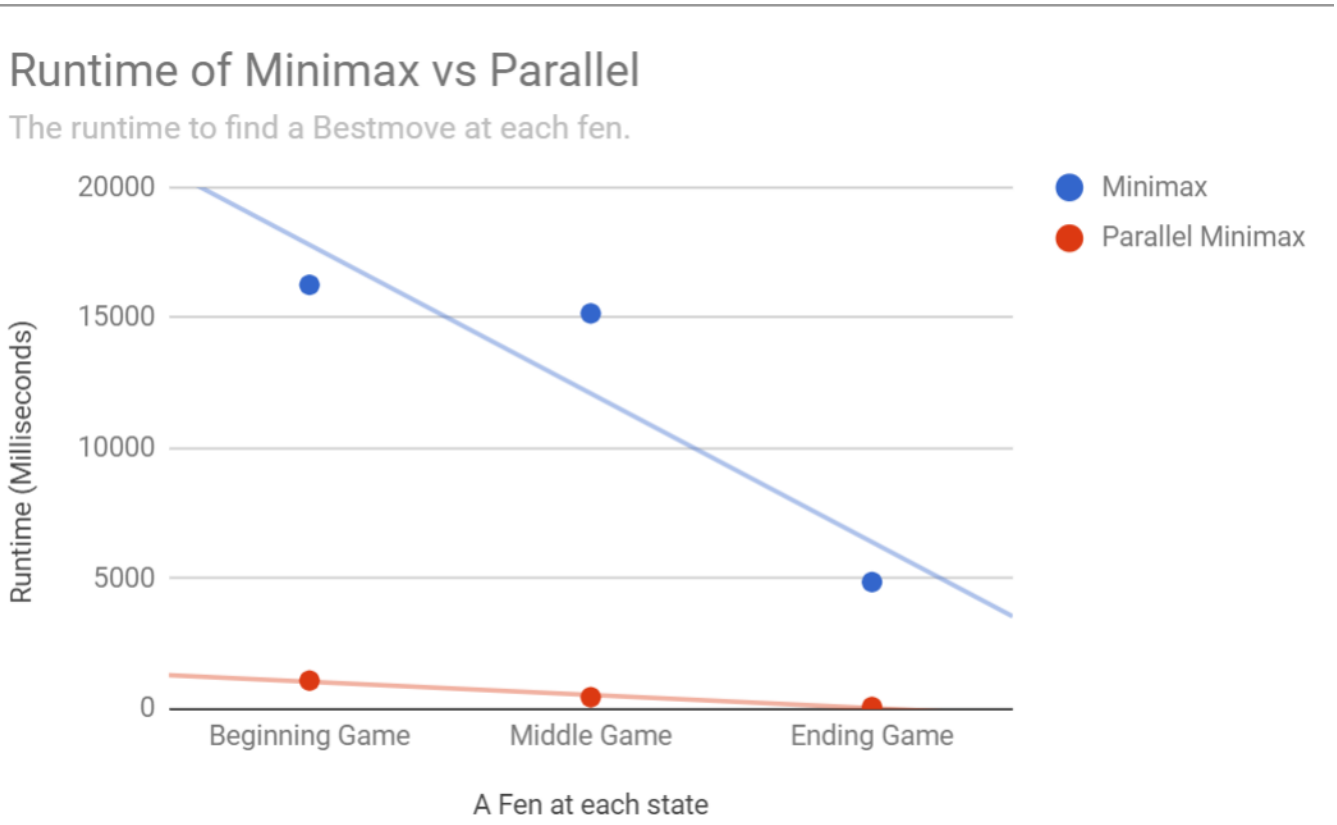
Plot your results and discuss anything surprising about your results here.

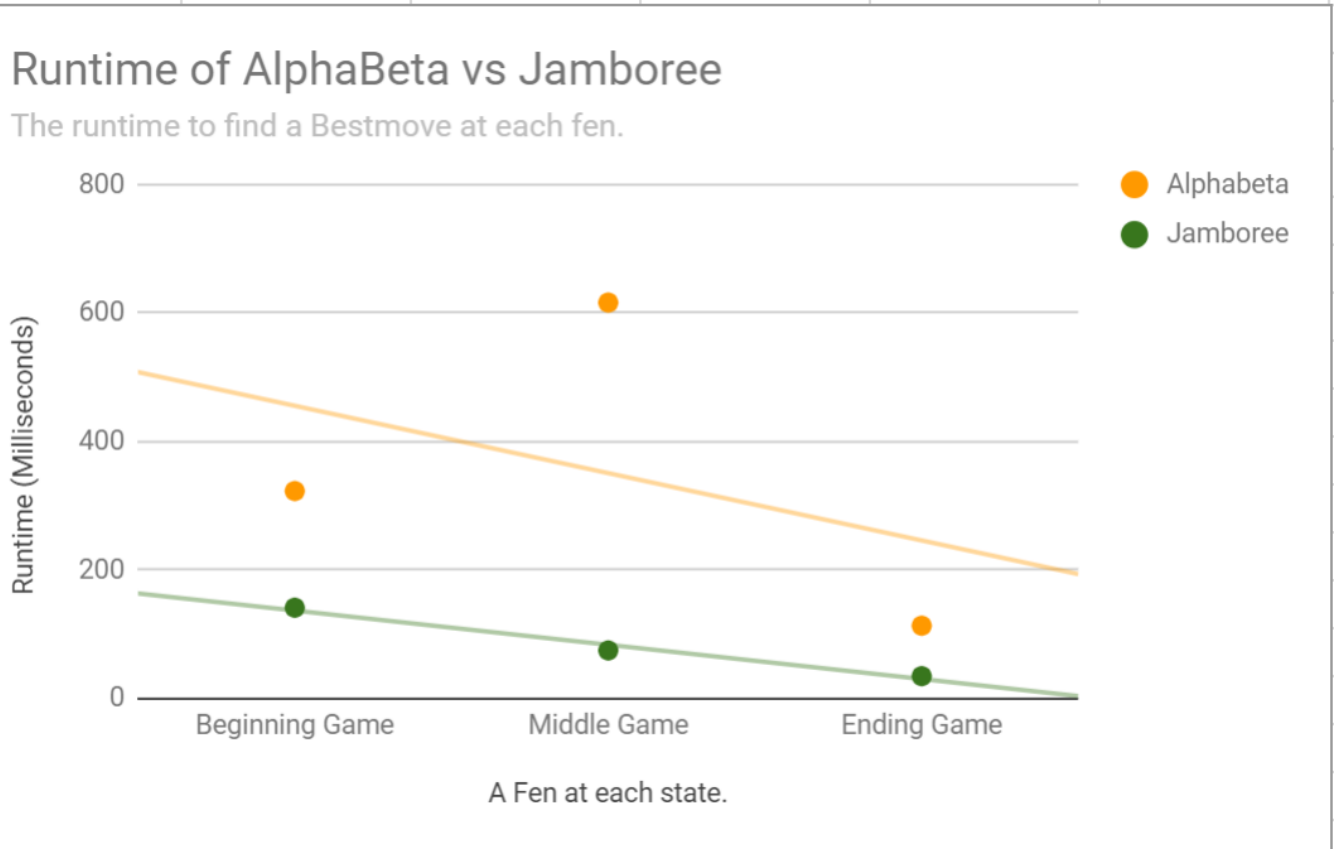
```
We completed this experiment by selecting the best values for cut-off and
processor
determined through the previous experiments for each combination of
Algorithm
and Game type. We recorded the time it took for Minimax to find each new
```

```
move,
throwing out the first five moves to decrease the effect of the JVM
warmup, and
took the average of all moves in the game to present a single averaged
move time.
We repeated this test for each cut-off 15 times and took the average.
Then, we
repeated this process for each Game type (beginning, middle, and end)
with Minimax
and for the other three Algorithms. Below are a table of our measurements
and graphs
representing the data.
```

| Algorithm | Early Game (ms) | Mid Game (ms) | End Game (ms) | | :----------------: |:----------------:|:-------------------:|:---------------:| | Minimax | 16267.076388 | 15172.447619 | 4873.333333 | | Parallel Minimax | 1095.104761 | 458.49333333 | 83.99411765 | | Alphabeta | 323.1222222 | 616.4047619 | 113.55 | | Jamboree | 141.5736111 | 75.04661412 | 35.06666667 |







From our table and graphs, we can see that each Algorithm's runtime decreased as the
size of the game increased. This makes sense, as there are fewer moves to choose
as the game continues.

```
Comparing the algorithms, we see that Minimax has the longest runtime, by
far. This
is due to the fact that it processes game tree nodes sequentially and
does not
prune the game tree. This means that Minimax has more work to do and only
a single
processor to do it, making it very likely to be the slowest. We can see
visually how
slow Minimax is compared to the other three Algorithms in the graph
"Runtime of
Optimized Searchers".
Parallel Minimax is a step up from Minimax, because it splits up the work
from the
game tree so multiple processors can process game tree nodes at the same
time. Although
it essentially follows the same algorithm as Minimax, the parallelism in
combination
with the optimized cut-off and processor values decreases the runtime by
a whole factor
of ten. Something that we found surprising was that Parallel Minimax is
also faster than
Alphabeta for the middle and ending games. Alphabeta cleverly prunes the
game tree,
leading it to view fewer nodes than every other Algorithm (as shown
above). However, it
is still slower than Parallel Minimax. This may be due to the fact that
we were able
to optimize Parallel Minimax (with cut-off and processors values), but
not Alphabeta.
Jamboree is the fastest of all the Algorithms for all three Game types.
This makes
sense as Jamboree combines the best of Alphabeta and Minimax; it prunes
the tree
and performs processing in parallel. With the optimized cut-off and
processor values,
it was able to have an average getBestMove that is less than 150
milliseconds in duration.
```

Beating Traffic

In the last part of the project, you made a very small modification to your bot to solve a new problem. We'd like you to think a bit more about the formalization of the traffic problem as a graph in this question.

- To use Minimax to solve this problem, we had to represent it as a game. In particular, the "states" of the game were "stretches of road" and the valid moves were choices of other adjacent "stretches of road". The traffic and distance were factored in using the evaluation function. If you wanted to use Dijkstra's Algorithm to solve this problem instead of Minimax, how would you formulate it as a graph?

```
  The graph for Dijkstra's algorithm would have "states", or vertices, of
all the different
  destinations in our map. The edges would be the adjacent "stretches of
road" and the weights
  would be the amount of traffic between two locations. We will set our
starting node to be our
  current location and follow Dijkstra's algorithm until we discover the
destination node.
  The spec states that "stretches of road" may have negative values
depending if they are dead
  ends. To implement Dijkstra's correctly, we will have to alter the
original algorithm slightly
  to avoid these  "stretches of road" and not break.
```

- These two algorithms DO NOT optimize for the same thing. (If they did, Dijkstra's is always faster; so, there would be no reason to ever use Minimax.) Describe the difference in what each of the algorithms is optimizing for. When will they output different paths?

```
  Minimax always chooses the smallest available "stretch of road", then
commits to the associated
  state as the next step in it's route. This is essentially following the
series of the smallest
  possible options. In comparison, Dijkstra's algorithm optimizes for the
shortest cost path in
  a graph. Depending on possible routes and how far ahead Minimax can
conceivably look ahead in
  traffic, Dijkstra's and Minimax may differ.
  If an edge has a negative value, Dijkstra's algorithm does not promise
to find the shortest cost path.
  So, depending on how Dijkstra's is implemented, it is possible for
Dijkstra to fail us in the case
  in which we reach a dead end.
```