Olivia Lackner

**Lab 4**

In this lab, I again developed algorithms which would be used to solve the TSP. In the previous lab, I used Brute Force and Dynamic Programming to solve the TSP and analyzed the timing of those algorithms. With Brute Force and Dynamic Programming, my final path with a certain cost was known to be the path with the lowest cost.

In this lab, I implemented two heuristic algorithms, which try to quickly find a very good solution, though it is not guaranteed that the solution found will be the absolute best (lowest distance in our case of the TSP). The two different algorithms I implemented were the Genetic Algorithm (GA) and Simulated Annealing (SA).

**Genetic Algorithm**

Genetic Algorithm, or GA, was inspired by Darwin’s theory of evolution and survival of the fittest. It makes use of the ideas of selection, breeding, and mutation to generate better and better solutions to our problem until we converge on a very good solution, ideally the best solution. In my implementation of GA, each chromosome in the population, or generation, is a potential solution to the TSP. GA has many tweakable parameters, which means that thorough testing and analysis is necessary to determine which parameters provide the best solutions most often.

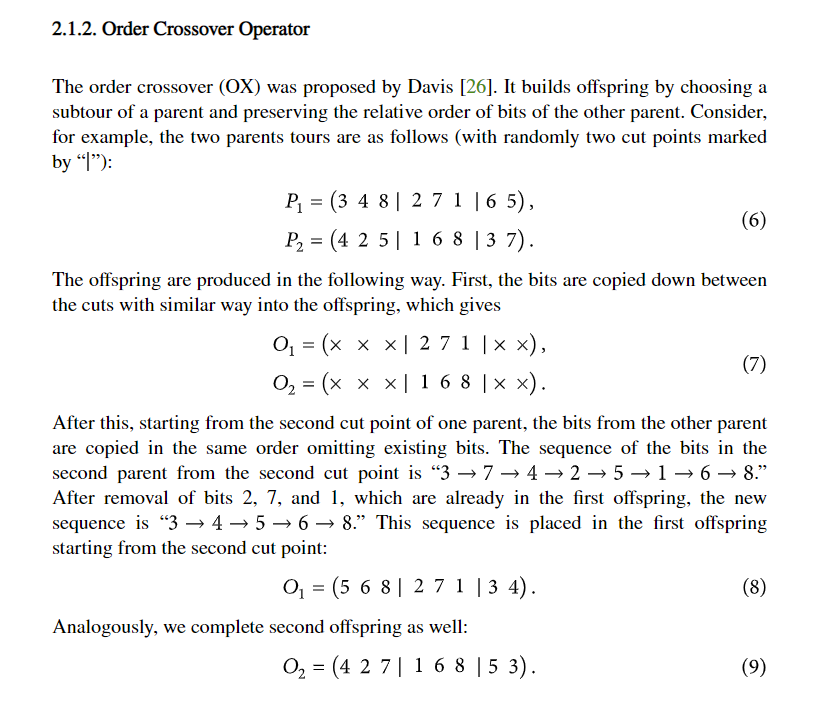
For my GA implementation, I implemented two different techniques each for selection, crossover, and mutation.

**Selection**

For selection, or how the parents for the next generation of potential solutions will be selected, I implemented both Elitism and Tournament selection. Elitism works by selecting the solutions with the highest fitness values (lowest cost in our case) to breed in order to produce the next generation. Tournament Selection selects a set of chromosomes, from which it then picks the one with the best fitness value to be a parent, repeated multiple times.

**Crossover**

There are many techniques for crossover, or breeding, when coding GA, though the options are slightly limited by the nature of our problem. In TSP, each city (node) can only be visited once. This constrains the various crossover options we can use. For my first crossover technique, I coded Ordered Crossover, or OX, for short. It is quite difficult to succinctly explain OX, so I have included an image below, detailing how OX works (<https://www.hindawi.com/journals/cin/2017/7430125/>).



For my second crossover technique, I implemented a technique in which each child replicates one of the parents except for a single crossed gene. In my research online, I saw that this technique can sometimes produce very good results when paired with significant mutation.

**Mutation**

It is essential that mutation exists in GA so that we do not get stuck in a local optima and fail to arrive at the global optima. In this lab, I implemented both a swap mutation, where I swap two nodes in a chromosome, and an inversion mutation, where I invert a section of the chromosome.

**Analysis of GA Configurations**

In order to analyze the various techniques for GA that I implemented, I plotted learning curves for the different parameters. Analyzing the learning curves allows me to determine if my algorithm is learning. Ideally, the learning curve should be steadily decreasing, since we are trying to minimize the distance of the path.

The following learning curves were generated with 80 individuals in the population, a mutation rate of 10%, and a path of size 40.

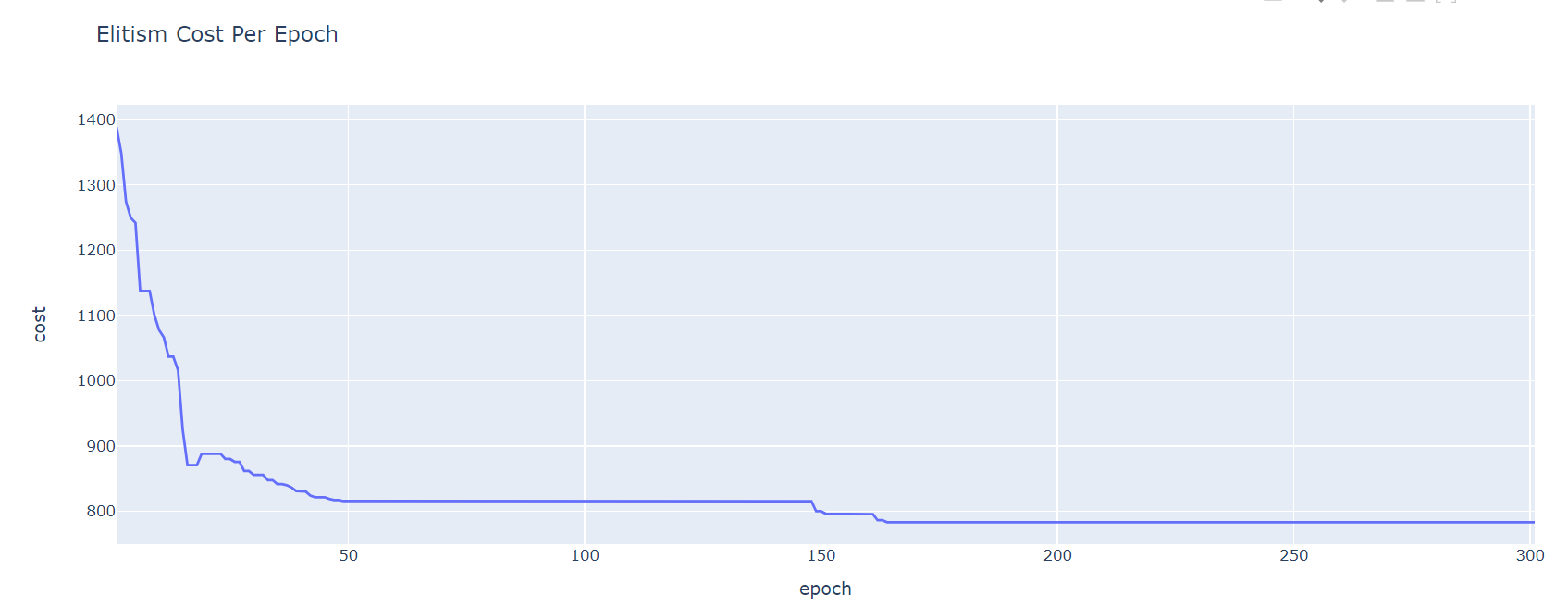
For all of the comparisons, I generated at least 2 learning curves - one with a very small number of epochs - 300 and one with a large number of epochs – 10,000.

This is the case because when I initially generated my learning curves, my GA configurations took a few minutes to run 300 epochs, so I thought 300 epochs would be sufficient. The next day, however, I was running GA configurations and noticed I could run substantially more epochs in the same amount of time. I am not sure what the reason for this is. It may have to due with the fact that my laptop is running out of storage space, or it may have been due to what other software was running on my laptop when I was running my algorithm. Nonetheless, the two graphs with different epochs provide more data points, which is never bad.

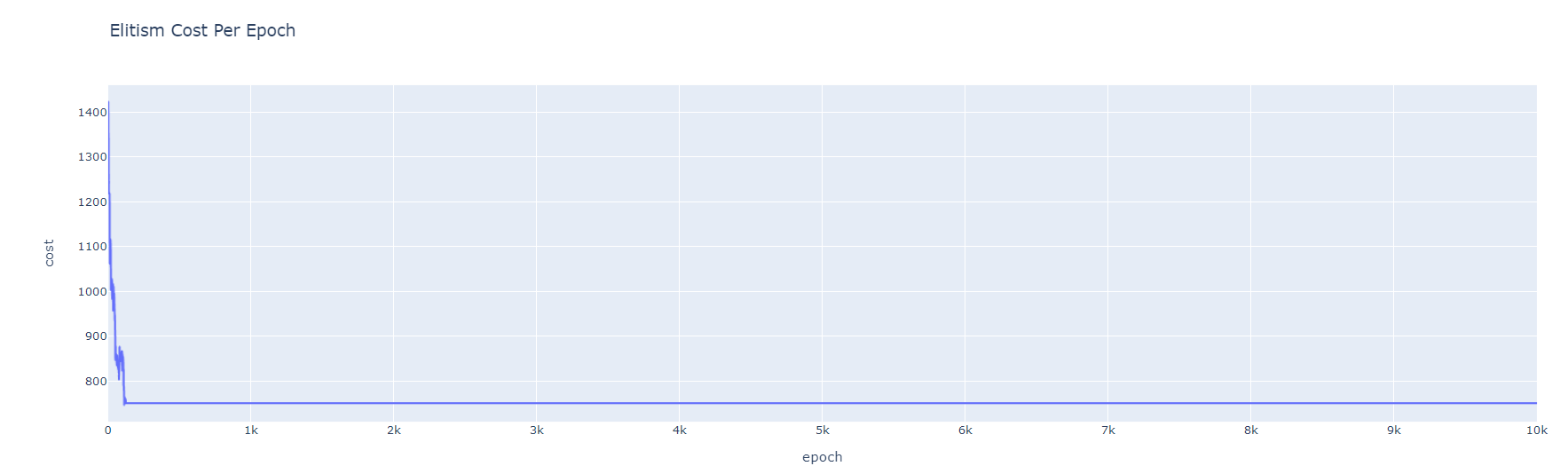
**Selection Comparison**

To compare my selection techniques, I plotted the learning curves for elitism and tournament selection when the other parameters were Ordered Crossover as the breeding technique and Inversion as the mutation technique.

**Elitism**

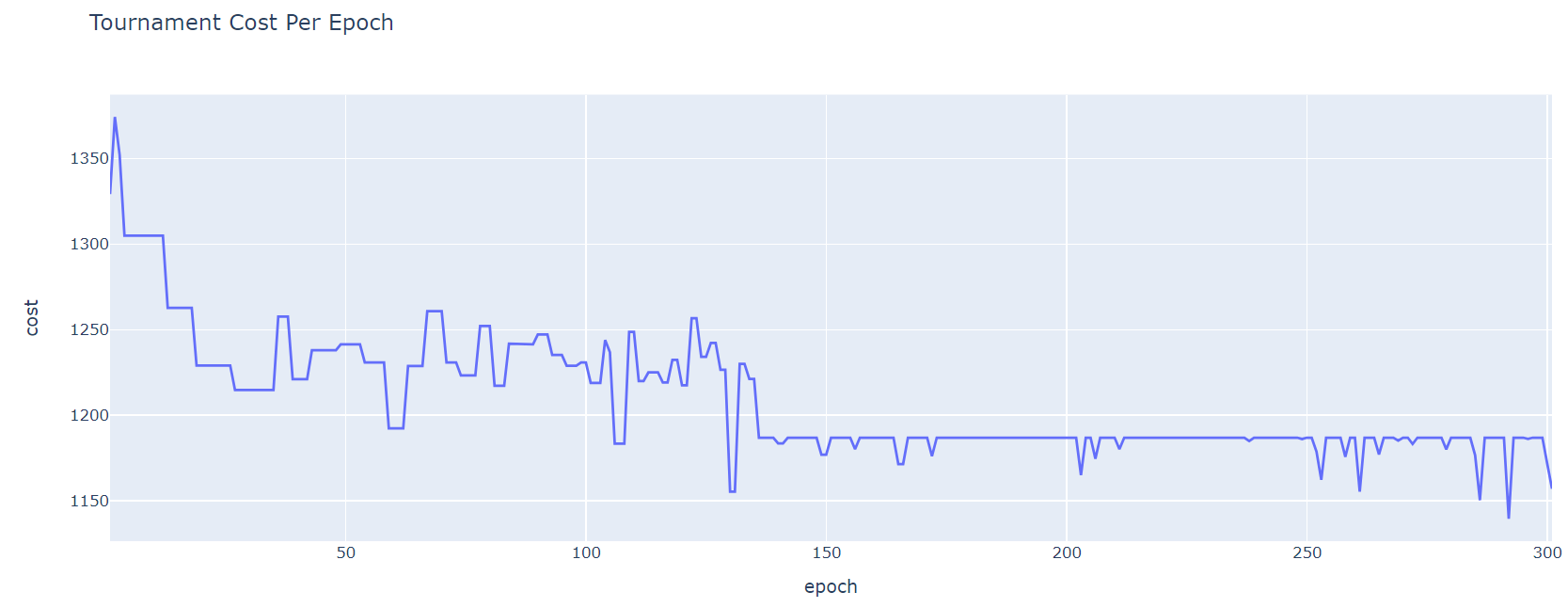


This learning curve indicates that the algorithm is learning since the curve is a smooth, downward-sloping line. I ran this algorithm on more epochs to analyze the learning rate more.

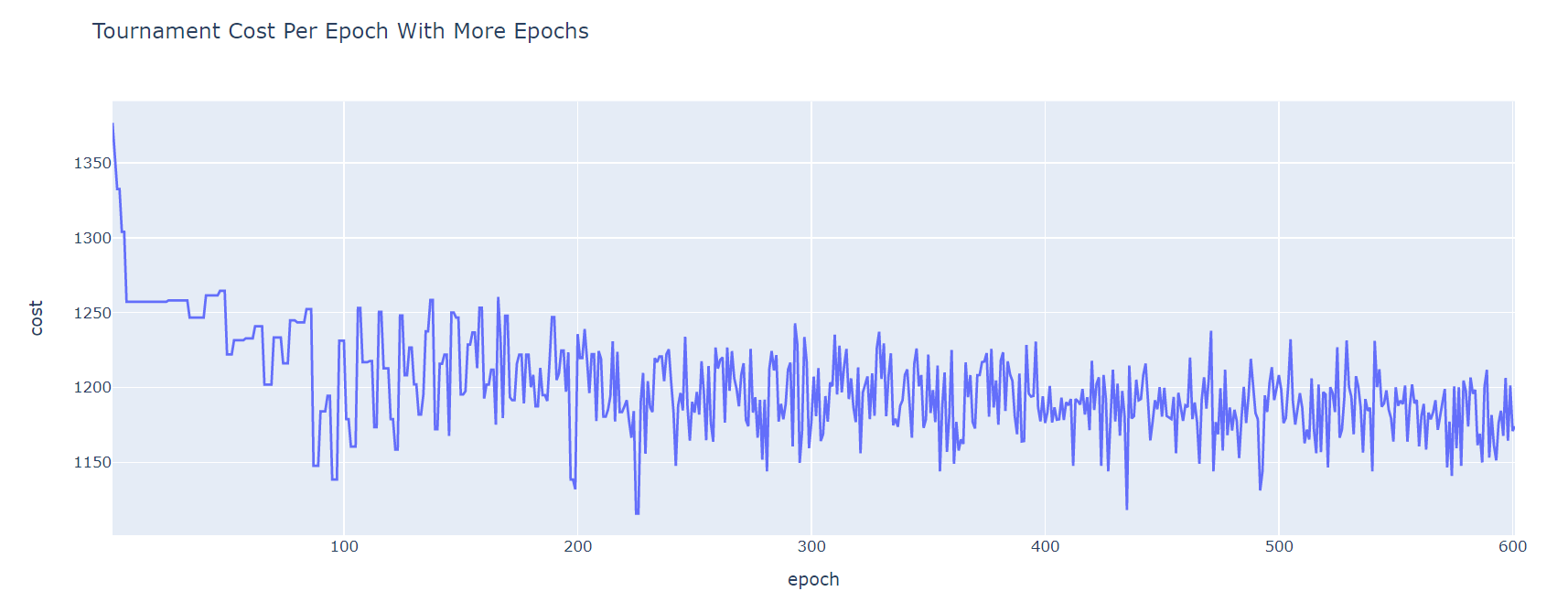


This learning curve is relatively good. The cost is being minimized over time, and it is clear the algorithm converges. The only concern I have with this learning curve is how quickly the curve converges.

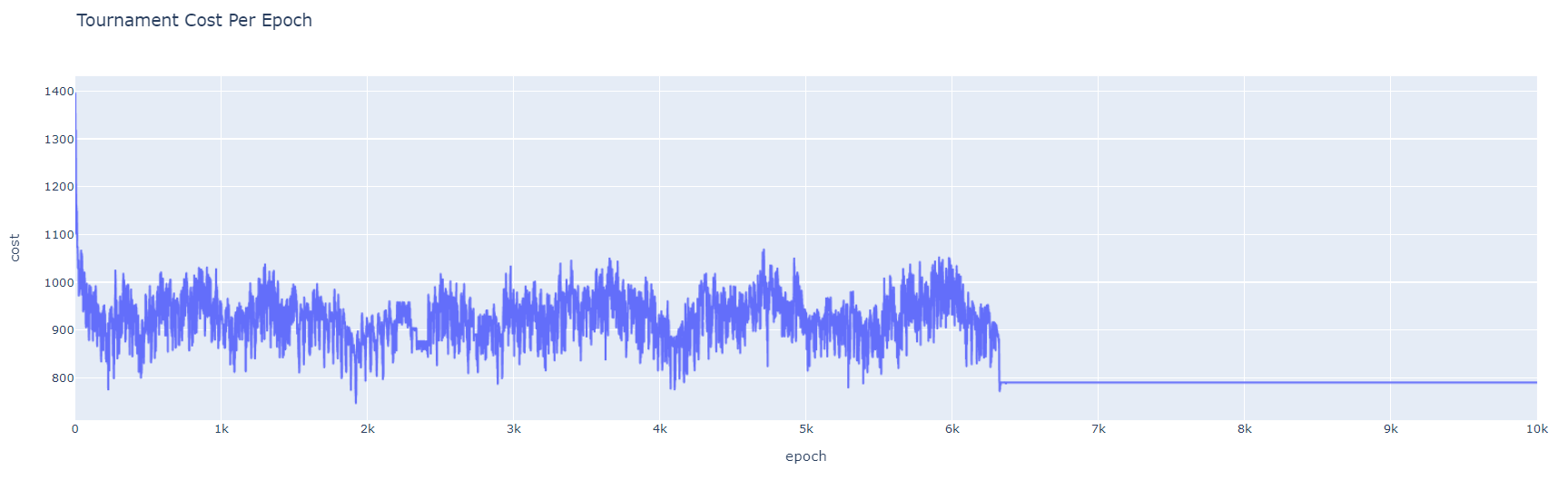
**Tournament**



The above learning curve does not appear to converge. I ran this algorithm again with 600 iterations, or epochs, and again plotted the learning curve to better analyze tournament selection.



The curve still does not appear to converge. I ran this algorithm for 10,000 epochs to further analyze it. The plot is below.



In the last graph, with 10,000 epochs, we can see that the cost converges.

This learning curve is not impressive. There is not a steady decrease in the cost and the final value the cost settles on is around 800, which is not that “good.” Later in this analysis of the various GA configurations we will see that a path with a cost around 700 exists. Additionally, from about 50 to 6,200 epochs, the cost jaggedly oscillates between high and low costs. This does not suggest that the algorithm is learning since an algorithm that is learning would have a curve that is steadily decreasing.

From these, I conclude Elitism selection is more likely to outperform Tournament as a selection technique. The learning curve for Elitism is a downward-sloping line, indicating that the algorithm is learning, and it converges to a much better solution (cheaper) than tournament.

This is likely the case since Elitism selects the best parents to breed for the next generation every epoch, and theoretically, the parents each generation should be better and better solutions. Hence, breeding better and better parents should produce better and better offspring.

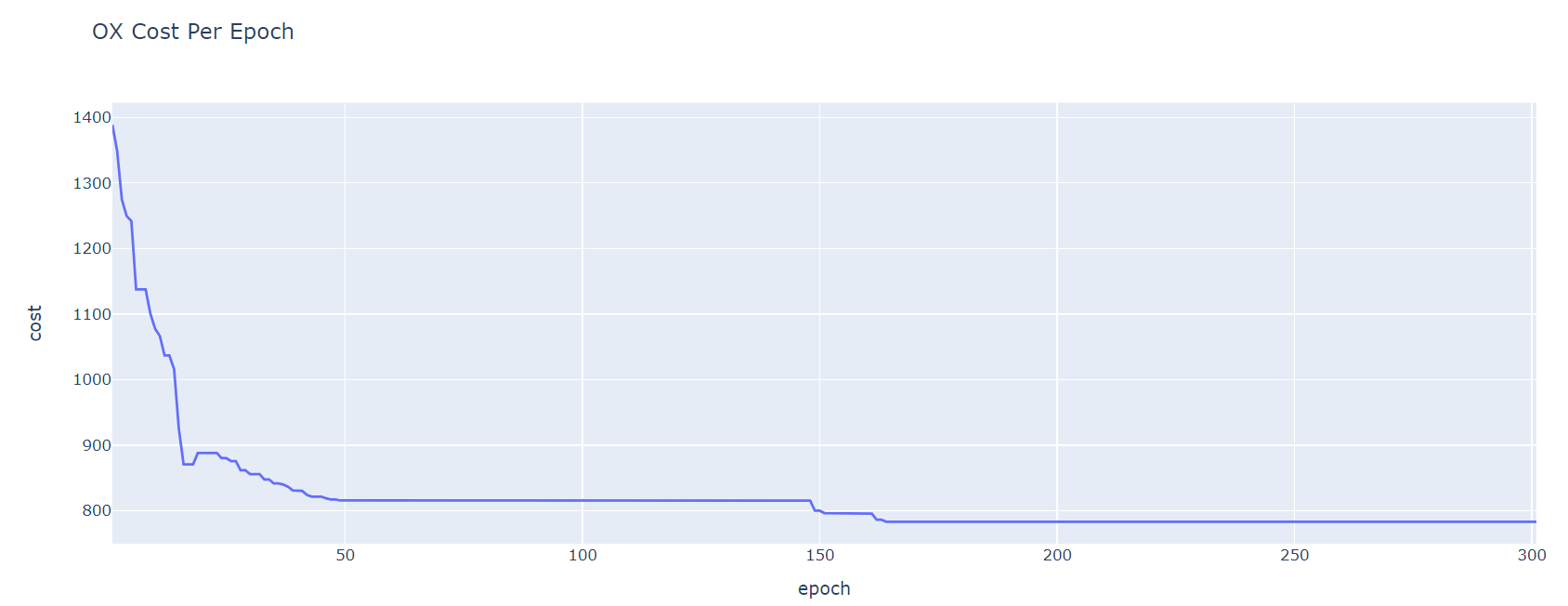
Tournament likely oscillates so much and does not appear to learn because the basis of choosing a parent to be in the mating pool is based off of comparing one parent’s fitness value to another parent’s . If both parents you select to compete in a tournament are bad solutions (and have low fitness values), you are going to end up selecting a bad parent to breed, which likely dilutes the next generation.

Elitism is more consistent in picking the better parents, and this is likely why Elitism appears to be a better configuration than Tournament.

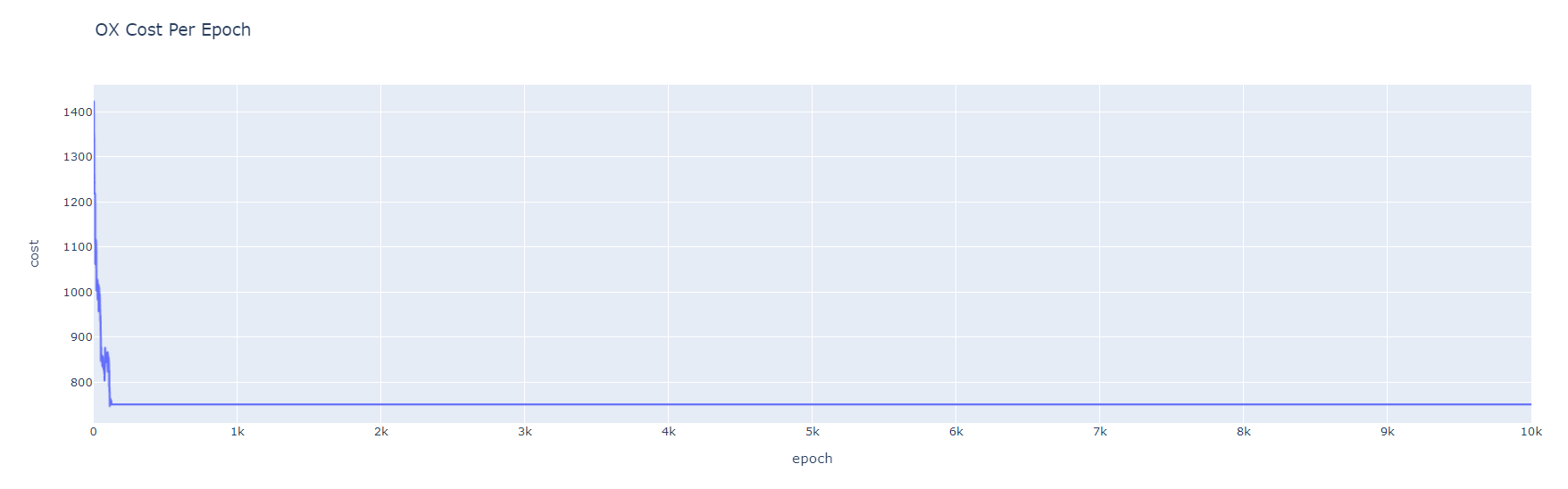
**Crossover Comparison**

To compare my crossover techniques, I plotted the learning curves for Ordered Crossover (OX) and Single Cross breeding techniques. When testing these, the other parameters were Elitism as the selection technique and Inversion as the mutation technique.

**OX**

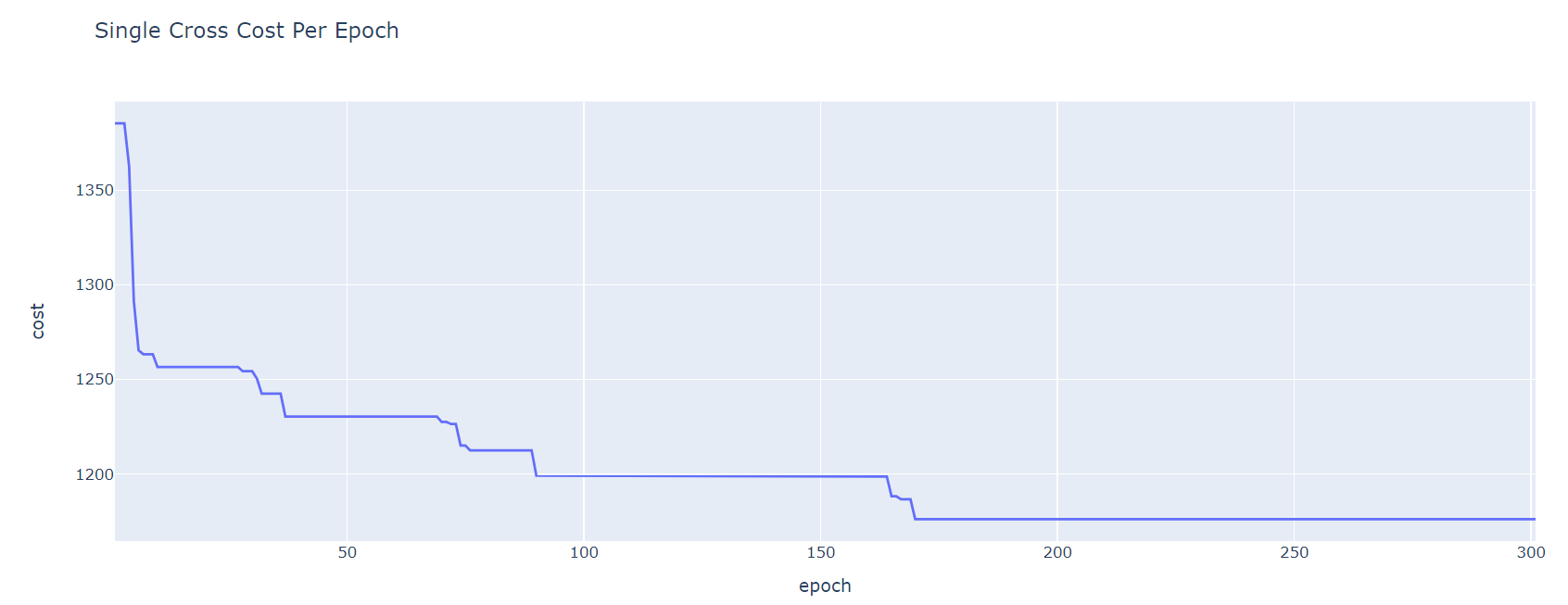


This learning curve indicates that the algorithm is learning since the curve is a smooth, downward-sloping line. I ran this algorithm on more epochs to analyze the learning rate more.

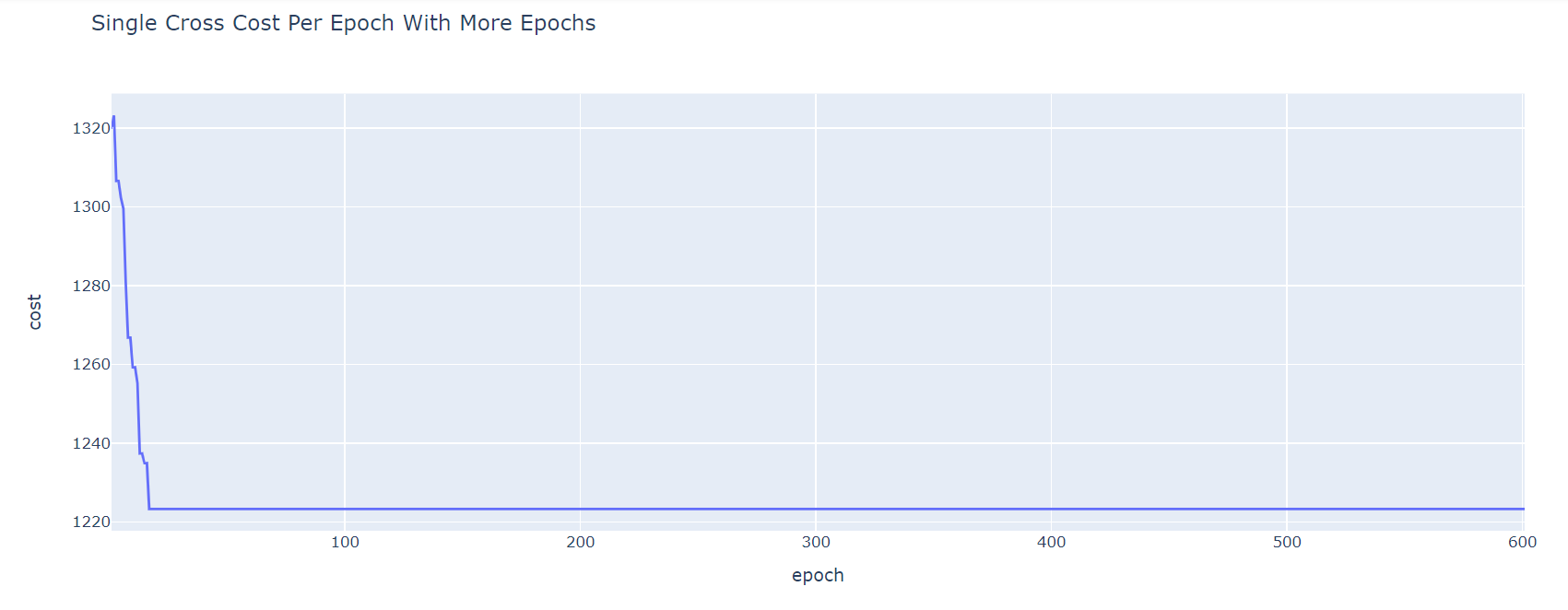


This learning curve is relatively good. The cost is being minimized over time, and the algorithm clearly converges. The cost the curve converges to is around 750, which is a reasonable solution to settle on, as shown by the costs the other configurations converge to. The only concern I have with this learning curve is how quickly the curve converges.

**Single Cross**

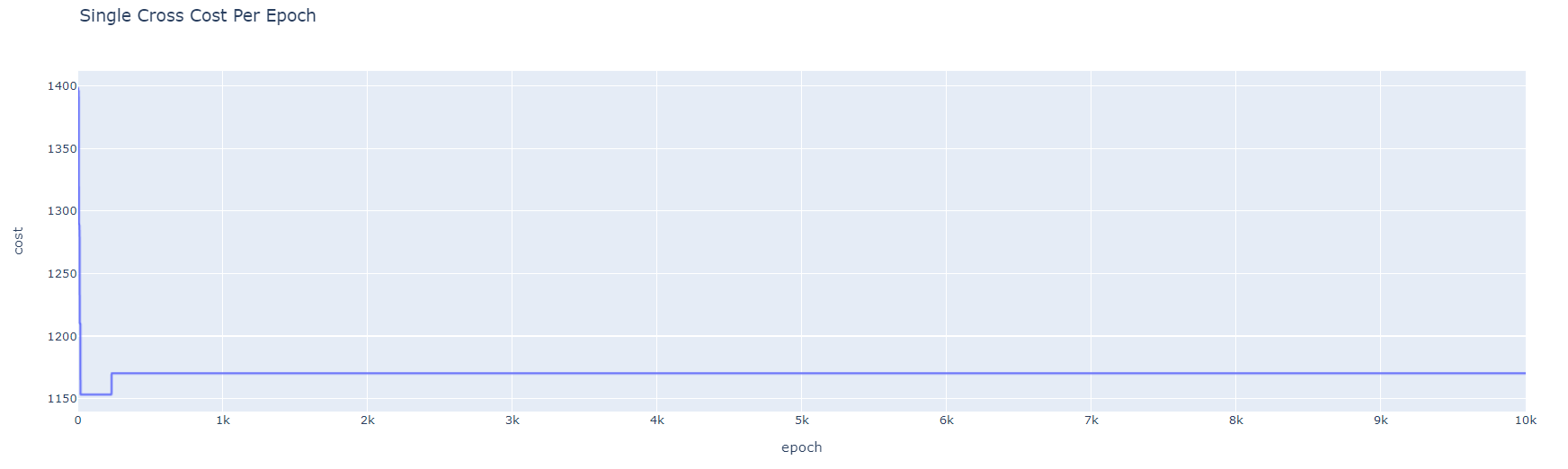


It is unclear whether this curve has converged. I ran this algorithm again with more iterations, or epochs, and again plotted the learning curve to better analyze it. Below is the algorithm run for 600 iterations.



This algorithm appears to converge, but the cost it converges to is around 1,225, which is not impressive at all since some of the curves converge to a cost around 700 for this same set up.

I ran this configuration one last time with 10,000 iterations. The plot is shown below.



We can see that the learning curve has converged, though by comparing this curve to the others, we can easily tell that the number it has converged to is not close to optimal. The curve also drops very quickly and then increases slightly and does not improve more, which is not ideal.

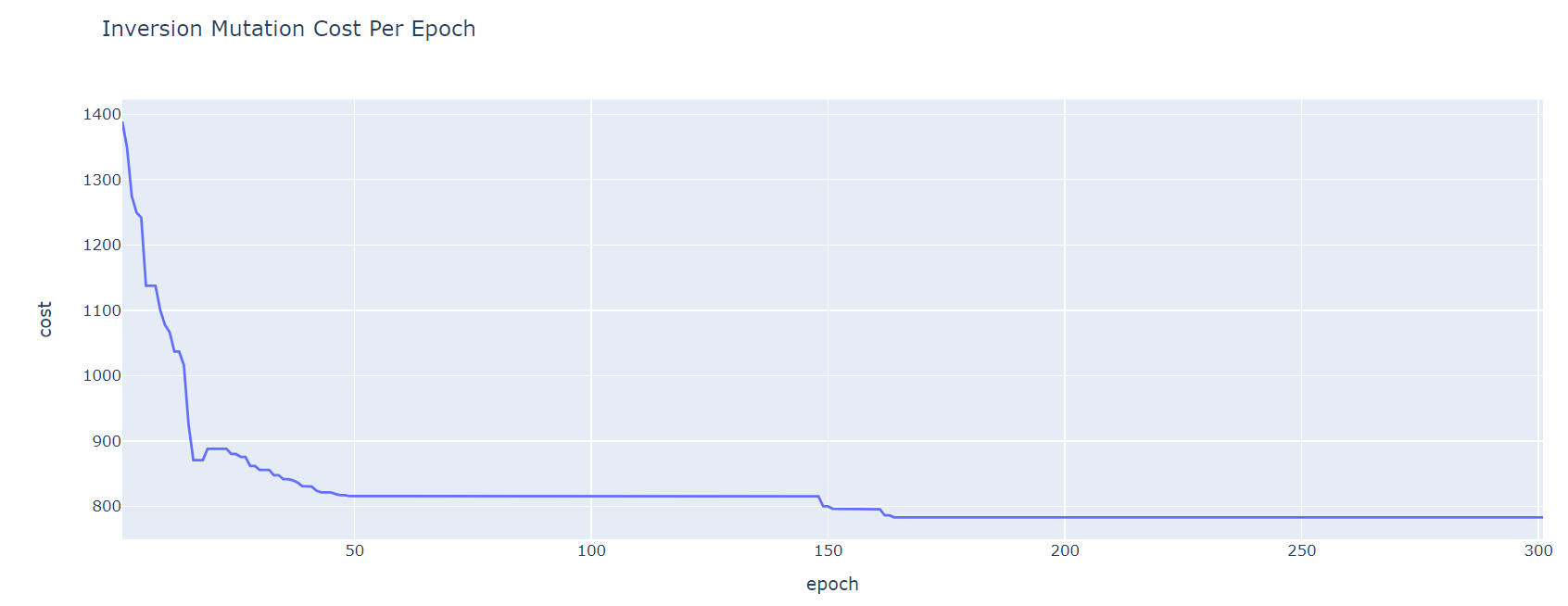
From these plots, I conclude Ordered Crossover is more likely to outperform Single Cross as a breeding technique.

I am not surprised by this conclusion. Ordered Crossover swaps multiple genes in swaths, which maintains the parents’ sequences to some degree while still combining the genes from both parents. With my implementation of Single Cross, each child replicates one of the parents except for a single crossed gene. This crossover technique probably does not produce an ideal learning curve since the parents’ genes are barely being crossed to form the children. This crossover technique might have produced a better learning curve if it was paired with frequent and significant mutation.

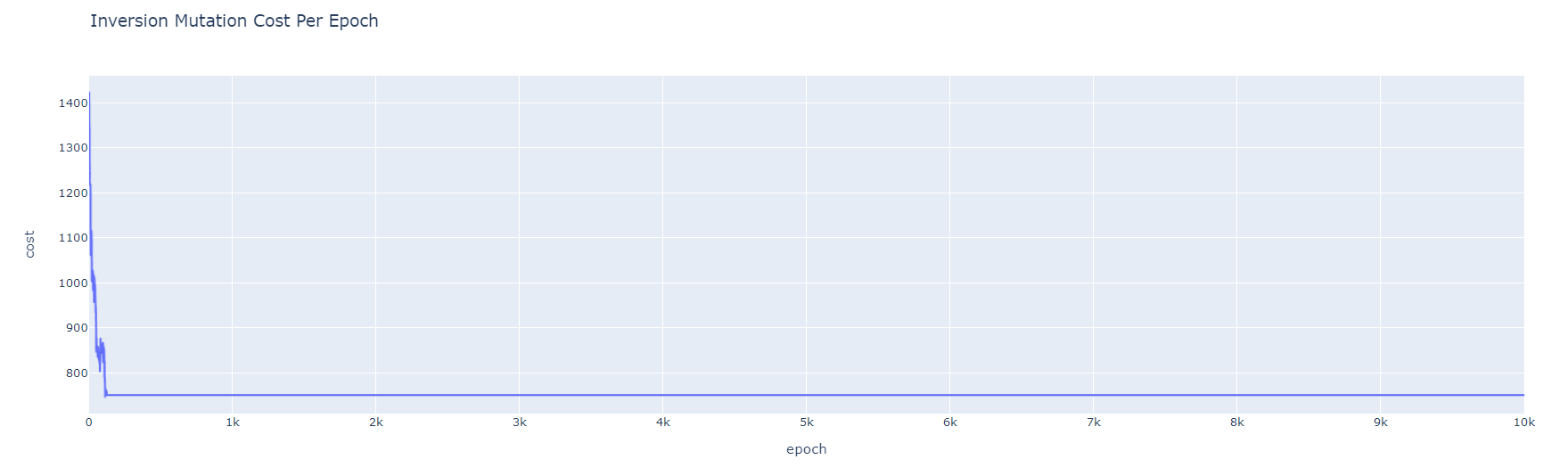
**Mutation Comparison**

To compare my mutation techniques, I plotted the learning curves for Inversion and Swap mutation techniques. When testing these, the other parameters were Elitism as the selection technique and OX as the crossover technique.

**Inversion**

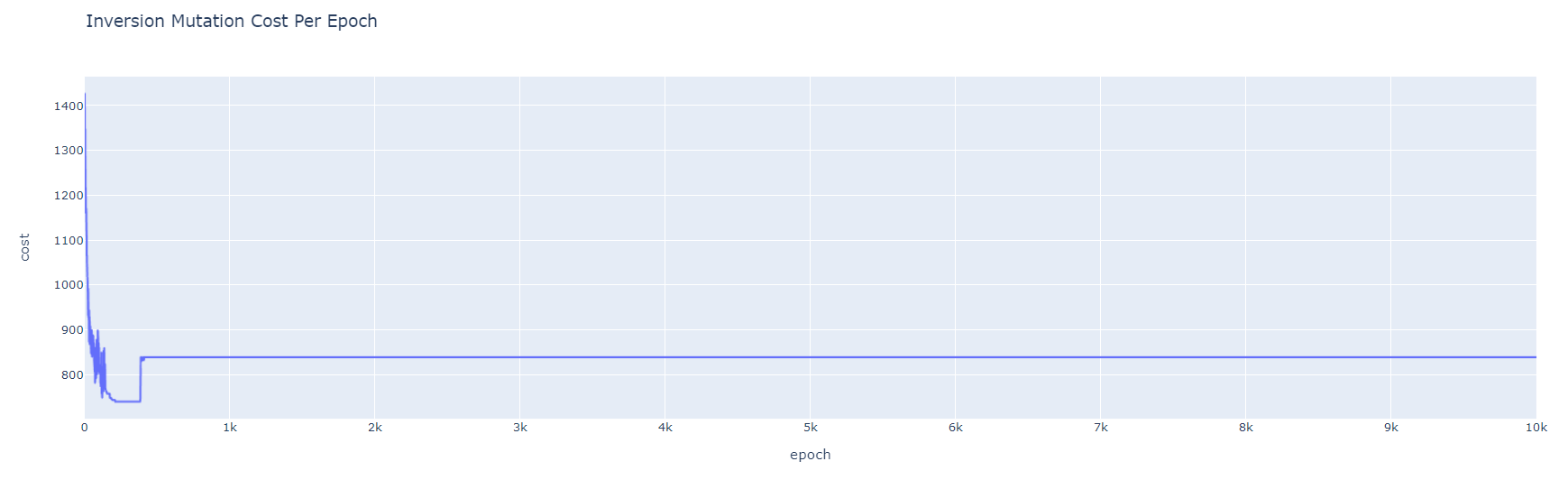


This learning curve indicates that the algorithm is learning since the curve is a smooth, downward-sloping line. I ran this algorithm on more epochs to analyze the learning rate more.



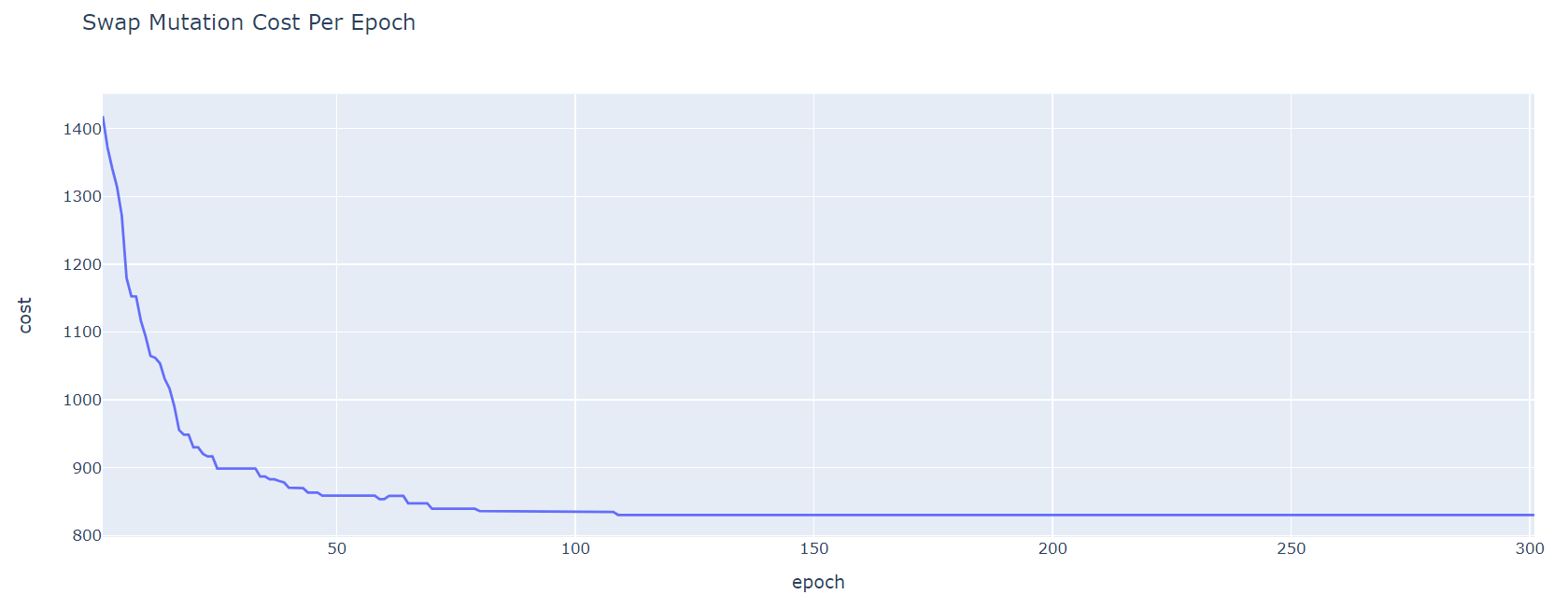
This learning curve is relatively good. The cost is being minimized over time, and the algorithm clearly converges. The cost the curve converges to is around 750, which is a reasonable solution to settle on, as shown by the costs the other configurations converge to. I am slightly concerned with how quickly this learning curve converges.

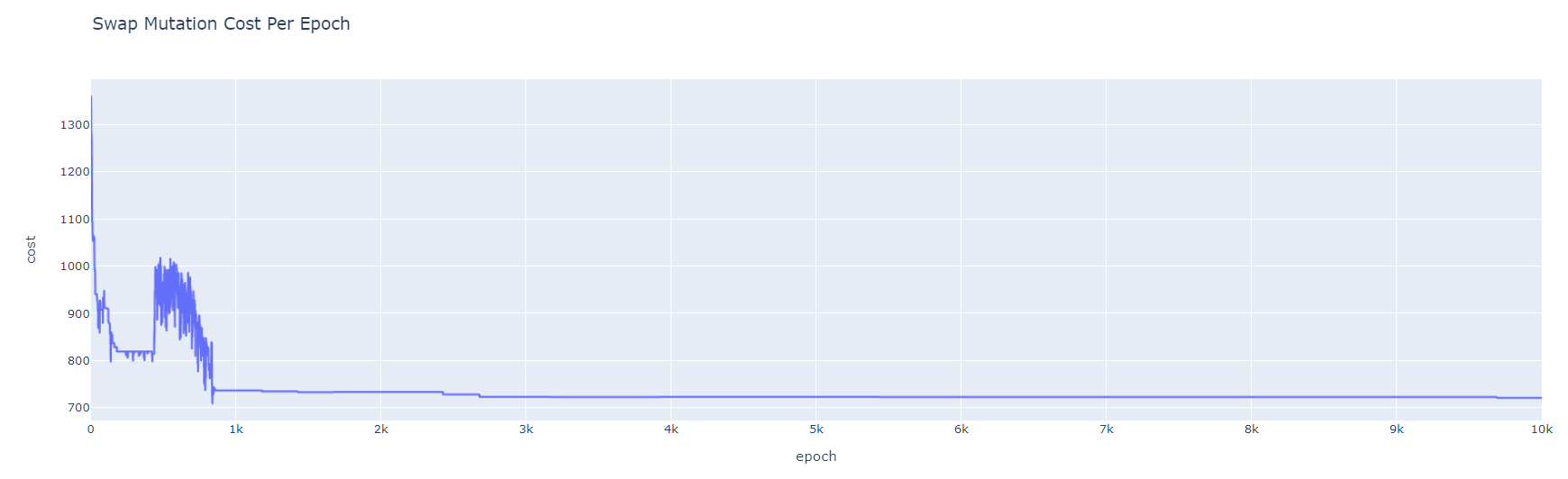
I ran this algorithm on the same number of epochs once more to see if it would converge to a lower cost.

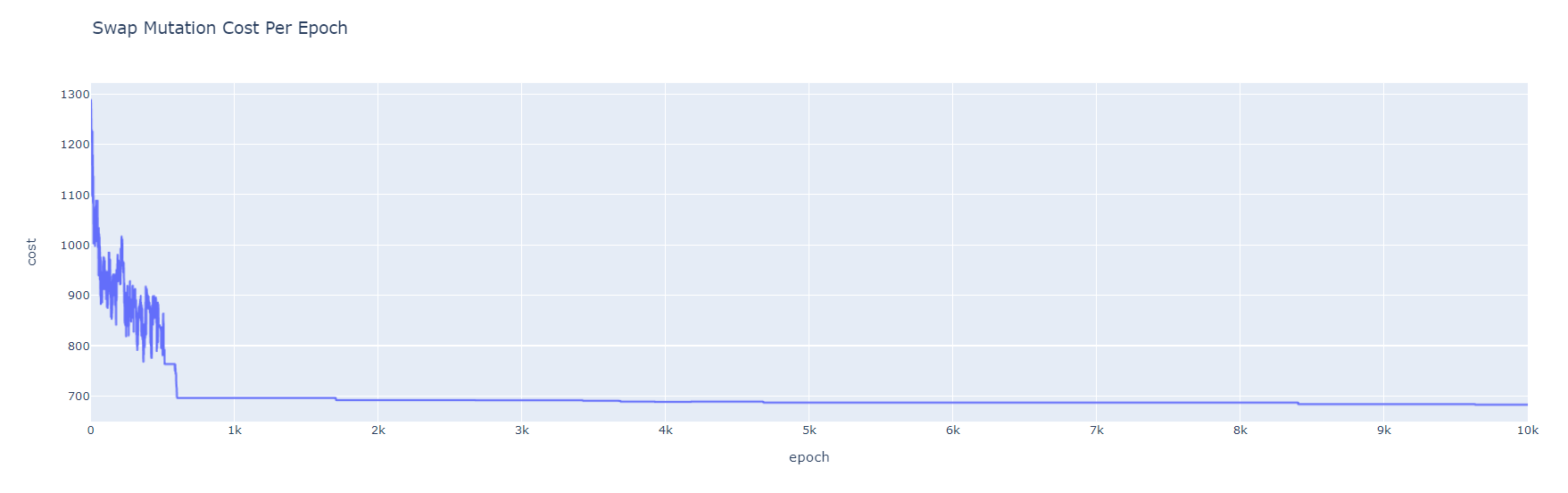


The plot above shows that this algorithm is prone to getting stuck on mediocre solutions. Knowing that the cost for this set up can be as low as 700, converging to a cost of 850 is not ideal. Additionally, the jump up in cost around 300 epochs is undesirable, and this configuration converges very quickly. While this algorithm is performing decently, I hope to find a better configuration.

**Swap**



The learning curve above looks very good. The curve is a smooth, downward-sloping line, indicating that this GA configuration is learning. I will run this configuration on more epochs to further analyze it. 

This learning curve is also pretty good. Besides the one jump in cost around 500 epochs, this curve trends downwards and converges to a relatively low cost for this set up- 750. I think this configuration is my best configuration so far, so I will run it once more on 10,000 epochs to analyze. 

This learning curve again indicates that the algorithm is learning since cost is generally decreasing as the epochs increase and then converges. I particularly favor this learning curve since it is for the most part continuously decreasing and it converges to a very cheap solution. In the plot above, the solution converges to a cost of 700, which is quite good for this set up.

From these plots, I conclude Swap mutation is more likely to outperform Inversion as the mutation technique.

This is likely the case since Swap mutation is a bit more consistent, as it swaps two randomly chosen genes each mutation. Inversion mutation, on the other hand, inverts a different number of genes in a chromosome each mutation, which is less consistent. Sometimes 30 genes might be changed, while other times just 2 genes might be changed if Inversion mutation is used.

To summarize my conclusions from this section, I determined that Elitism selection, OX crossover, and Swap mutation are most likely to produce the best results for TSP.

Looking at the above plots, specifically at the costs the lines appear to converge to, I noticed that some of the lines converge to costs that are significantly higher than the costs some of the other plots converged to. This indicates that some of the algorithms are not converging to solutions close to the best, even when being run for minutes. These algorithms are not performing well enough to be used to find the timing data it takes to find the best solution in the next section since they do not tend to find a good solution even when run for minutes.

The configurations that do converge to a “good enough” solution and will be used in the next section for timing graphs are:

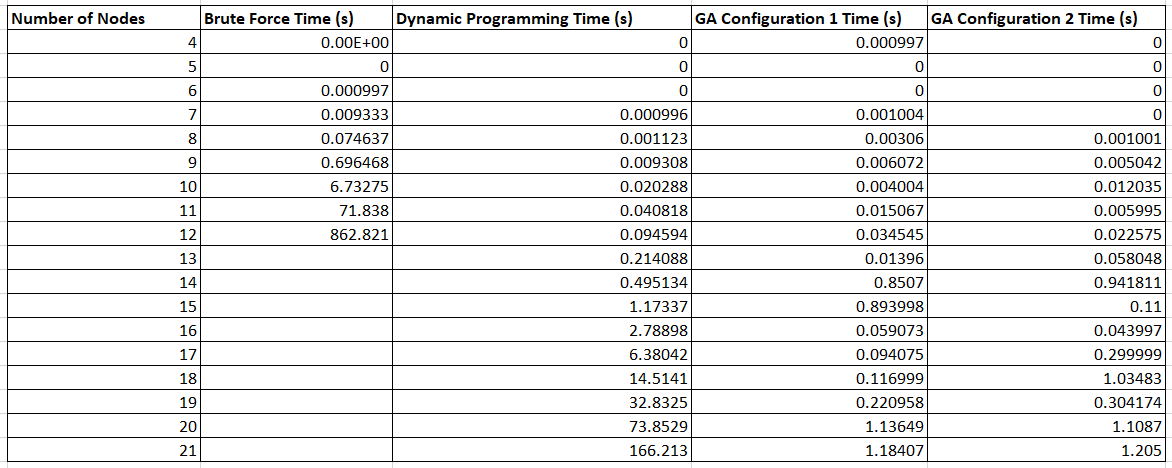
1. Elitism, OX, and Swap
2. Elitism, OX, and Inversion

**GA Timing Data Comparisons**

In Lab 3, I implemented a brute force and dynamic programming algorithm and determined the best path and the associated cost given the number of nodes to visit. From lab 3, I know what the solution to the TSP is for each number of nodes 4 through 21. For this lab, I ran my 2 configurations, noted above, until they were within at least 5% of the best cost (known from Lab 3) and collected the associated timing data.

A table summarizing the data obtained is shown below.

**Table I. Brute Force, DP, GA Configurations Timing Data**

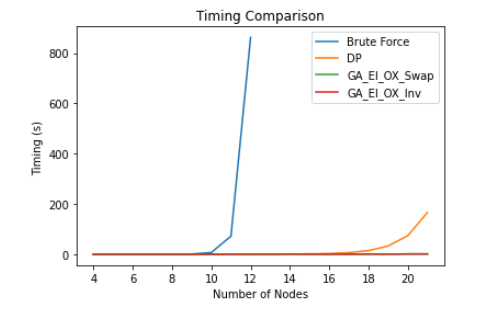


*Note: GA Configuration 1 is GA with Elitism, OX, Swap*

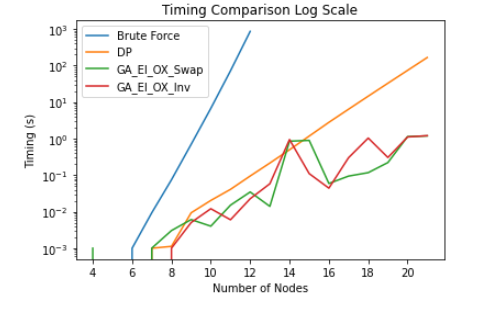
*Note: GA Configuration 2 is GA with Elitism, OX, Inversion*

*Note: Timing with brute force above 12 nodes was not obtained due to timing growth*

The graphs displaying this data are below:



The GA implementation with the configurations of Elitism, OX, and Swap and Elitism, OX, and Inversion are both straight lines as far as I can tell from this graph. To be able to more clearly see the timing data, I plotted this graph with a log scale on the y axis. The graph with a logarithmic y scale is shown below.



Analyzing the above graph, it is clear that the two different configurations of GA tend to be faster than the DP algorithm once the number of nodes is 10 or above. The GA algorithms are better than brute force at about 6 nodes in the path. Between the two different GA configurations, one is not consistently faster than the other.

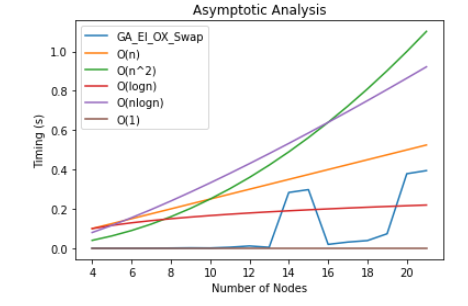
It makes sense that the GA algorithms tend to be faster than Brute Force and DP past a certain number of nodes since heuristic algorithms are meant to quickly find a good solution rather than methodically finding the absolute best solution.

**Analysis of Best GA Configuration**

I will now perform more analysis on the GA configuration I concluded to be the best of my configurations earlier.

This configuration was GA with Elitism, OX, and Swap.

I attempted to determine the time complexity of this algorithm by plotting the timing data along with timing curves for common time complexities. I normalized the values for these lines before plotting.

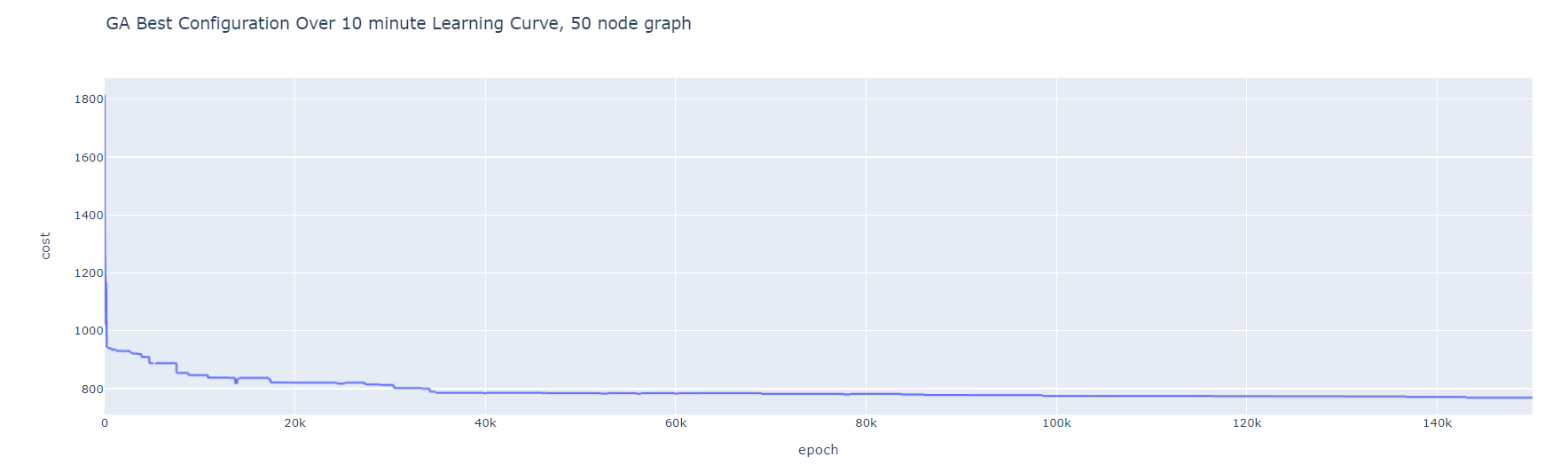


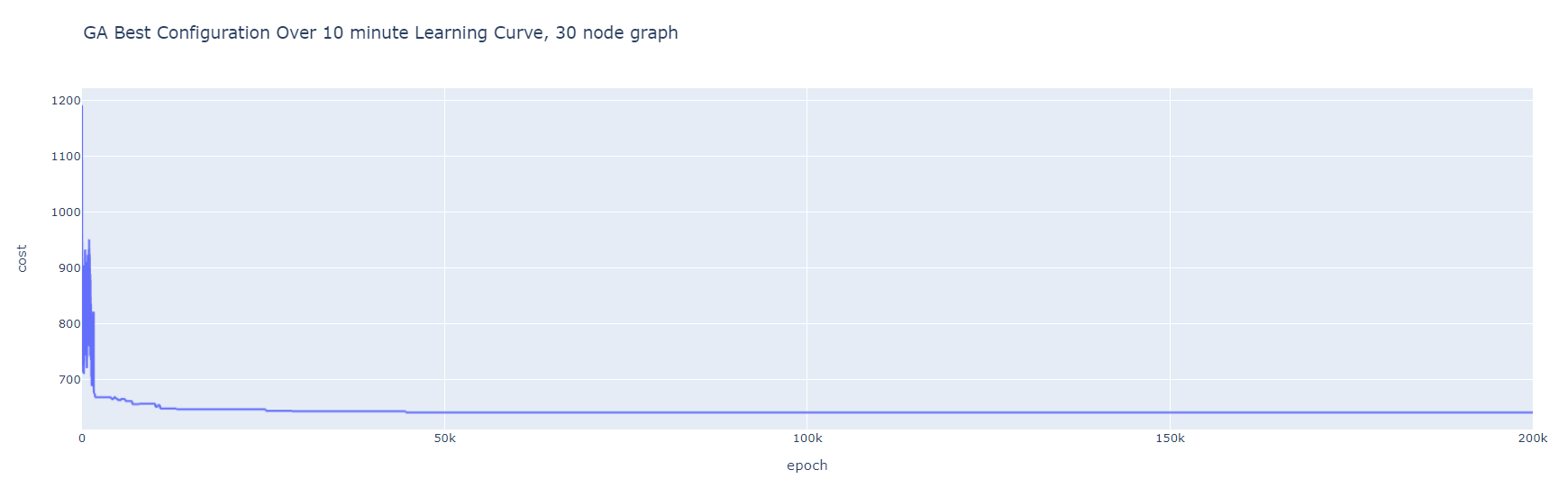
From the graph above, I conclude that this GA configuration has time complexity of O(n), where n is the number of nodes. Analyzing the time complexity of this heuristic algorithm by considering the parts of GA is quite difficult since there are so many operations performed in the algorithm and so many varying parameters. Nonetheless, I believe in my implementation of GA, the most expensive operation I ever perform is O(n). I know I perform O(n) operations a few times in my GA implementation. One example is generating the initial vector, which is size, n. This operation is O(n), since I push back a value n times. This graph also gives a nice visual of how the timing scales as the number of nodes increases, and it appears that this GA configuration is bound by O(n). This time complexity is considerably faster than the time complexity of DP, which is O(n22n ) and Brute Force, which is O(n!).

It makes sense that the timing line for GA is quite jagged since how quickly the algorithm finds the solution is very dependent on random events, such as how good each random mutation is or how good the random, initial population generated is, etc.

**10 Minute Learning Curves**

To further analyze the intelligence of this configuration of GA, I have plotted a few learning curves below, which were all run for more than ten minutes on different graphs.





For both of these plots, we see the curve decreasing and converging, which supports that this configuration of GA learns, even on graphs of different sizes.

**Simulated Annealing**

Simulated Annealing is an optimization method inspired by the way metals cool and anneal. With SA, there is always a current solution. The energy of the current solution is found. Then, a next solution is generated, and the energy of the next solution is determined. If the next solution’s energy is greater than the current solution’s energy, the next solution is taken to be the current solution. Otherwise, the next solution can be taken as the current solution with some probability less than 1. The probability is determined by an equation, which takes into account the current temperature and the energy difference between the current and next solution. The probability of moving to a worse solution decreases as the temperature decreases, and the temperature decreases each epoch. Hence, the likeliness of moving to “worse” solutions decreases as the algorithm runs.

For my SA algorithm, I implemented different techniques for temperature scaling over the search in addition to varying the initial temperature value.

For temperature scaling, I implemented proportional scaling and linear scaling.

Linear scaling: temperature = temperature – alpha

Proportional scaling: temperature = temperature \* (1-alpa)

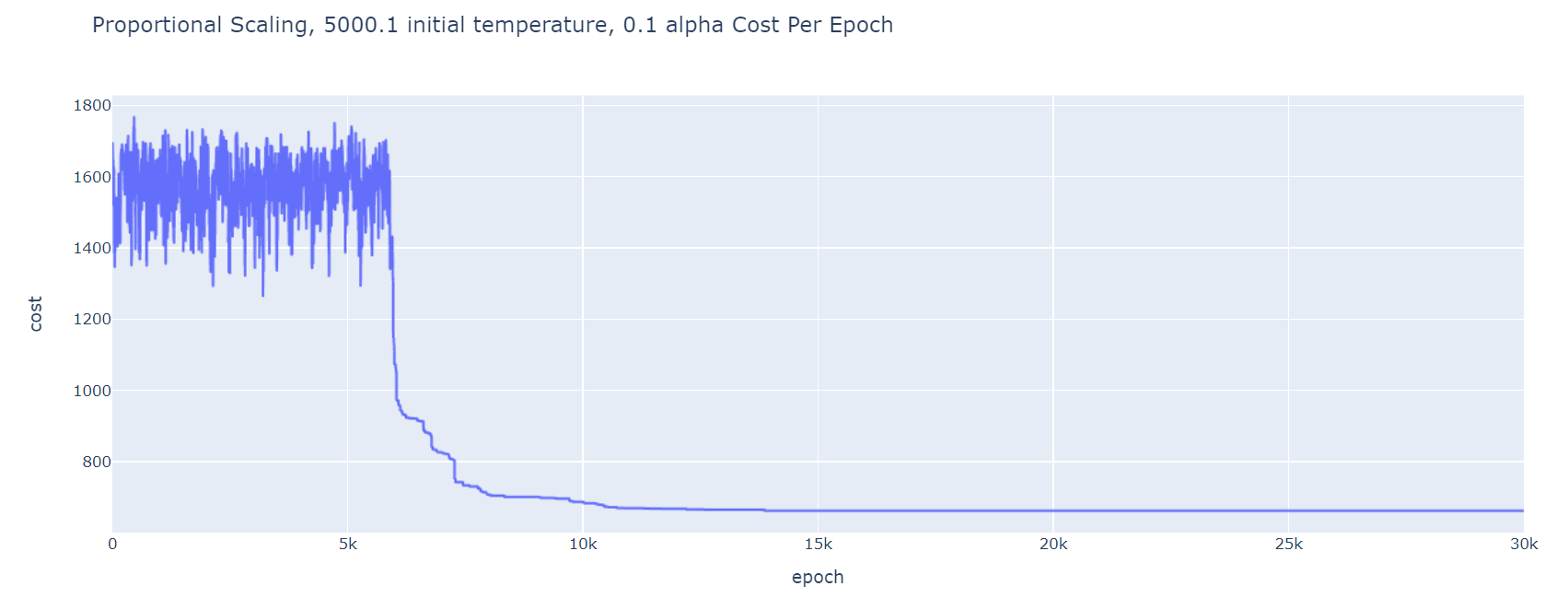
Additionally, a user can vary the starting temperature and the value of alpha.

**Analysis of SA Configurations**

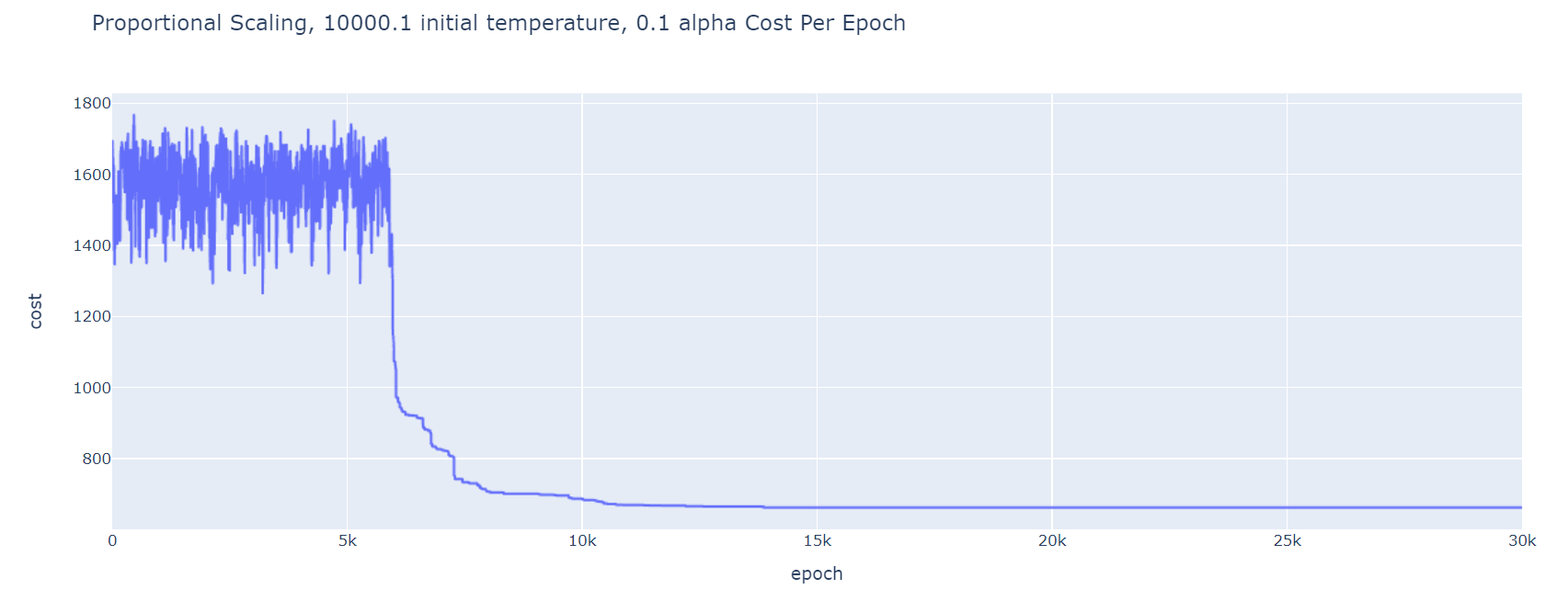
In order to analyze the various techniques for SA that I implemented, I plotted learning curves for the different parameters. Analyzing the learning curves allows me to determine if my algorithm is learning. Ideally, the learning curve should be steadily decreasing, since we are trying to minimize the cost.

**Proportional Scaling**

**Proportional With Alpha = 0.1:**

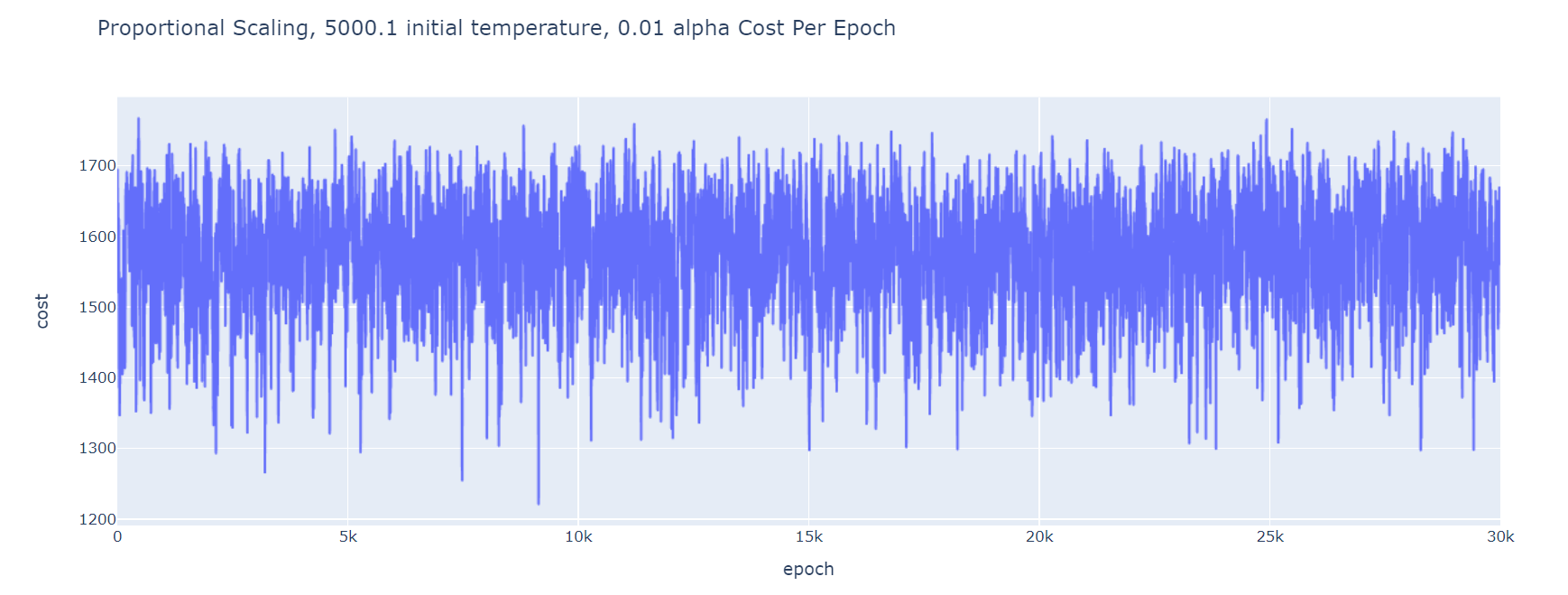


The learning curve above suggests that the algorithm starts learning after about 5,000 epochs, and then continues to learn until converging.

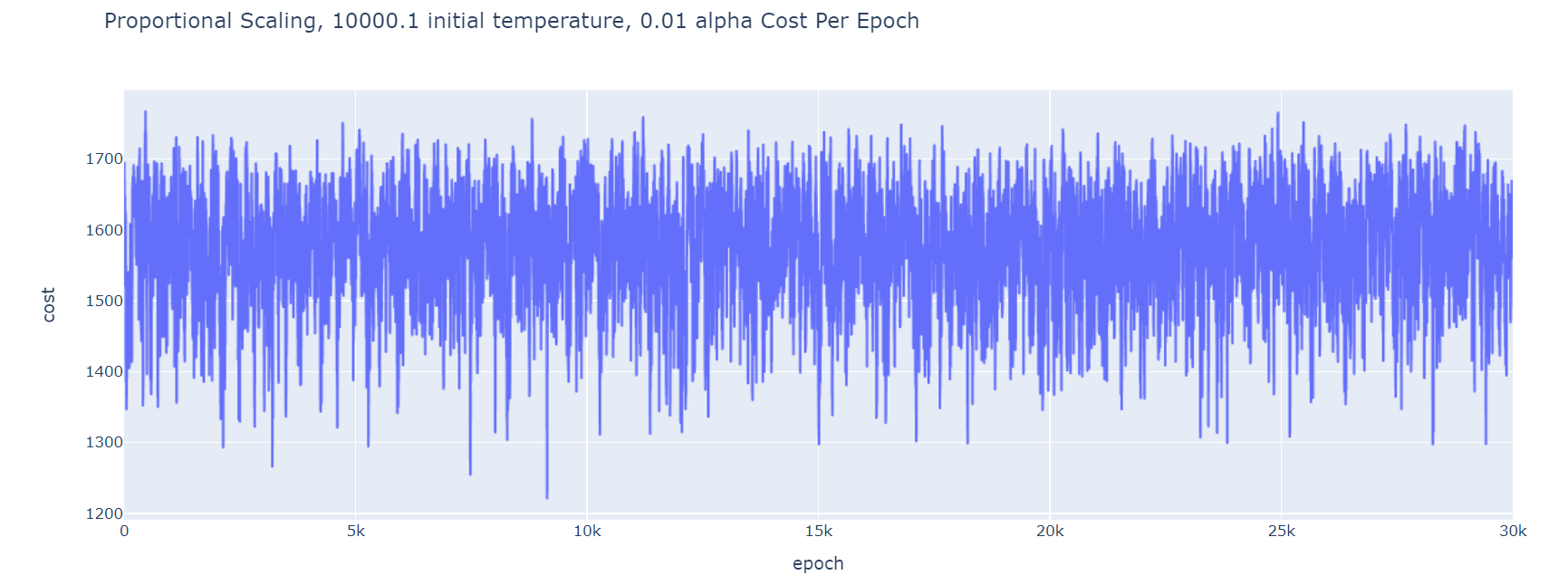


For this learning curve, the parameters were the same as the previous graph with the exception that the initial temperature was 10,000.1. The graphs essentially look the same.

**Proportional With Alpha = 0.01:**



This algorithm does not appear to learn as time passes since the cost oscillates up and down without a discernible downwards trend and does not converge. With a smaller alpha, the temperature decreases by a very small amount with each epoch and thus, the probability to accept “worse” solutions decreases by a very small amount with each epoch. This explains why the cost jitters up and down – the rate at which “worse” solutions are accepted is barely being decreased as the epochs pass.



This algorithm does not appear to learn as time passes since the cost oscillates up and down without a discernible downwards trend and does not converge. The reason for this is likely the same as the reasoning I provided for the previous plot.

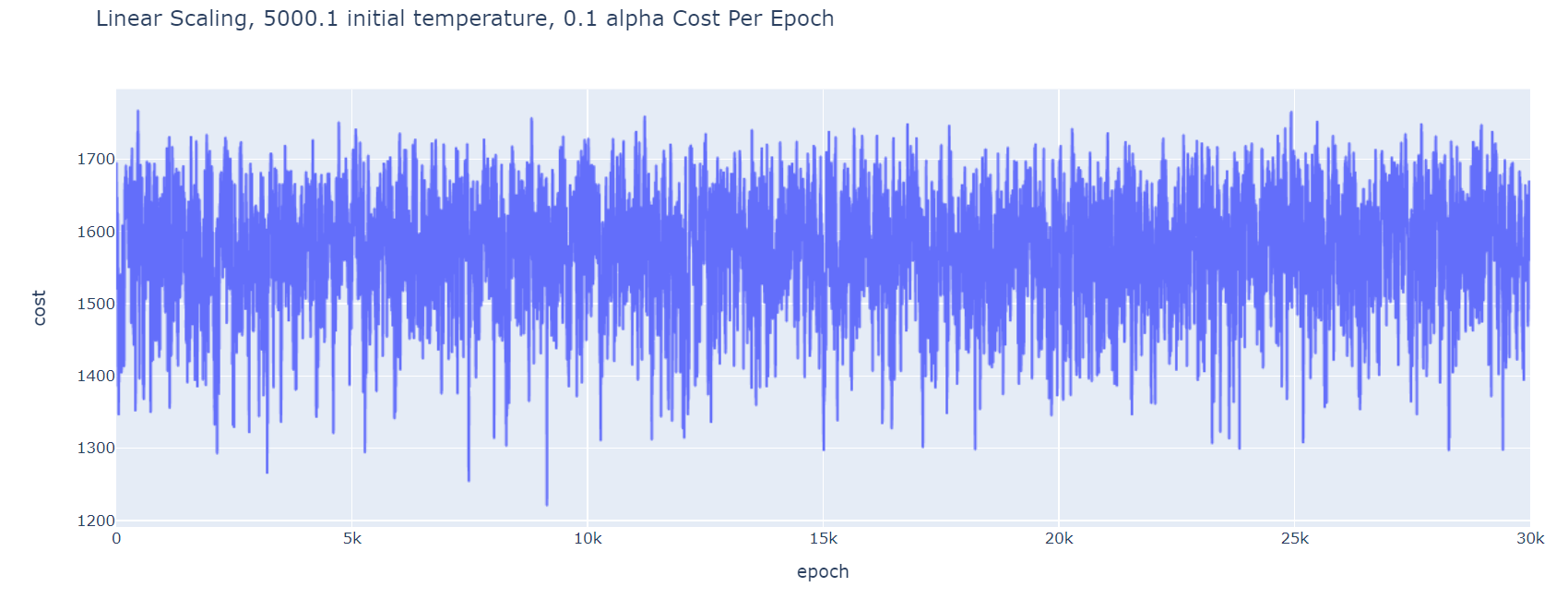
From the above learning curves, I conclude that for proportional scaling, an alpha of 0.1 performs better than an alpha of 0.01. This is likely the case since a larger alpha means that the temperature is being decreased by a greater amount with each epoch, decreasing the likelihood of accepting a “worse” move.

**Linear Scaling**

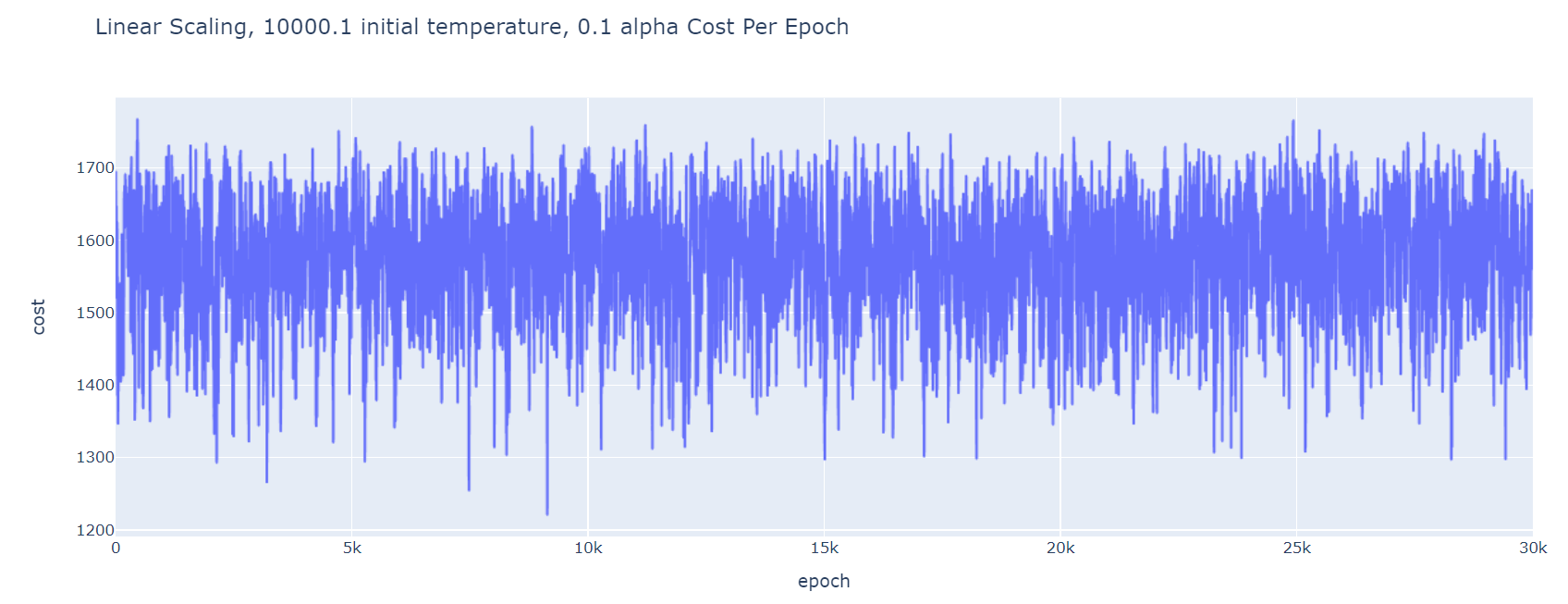
I will now visualize the learning curves for SA when linear scaling is used.

First, I will analyze linear scaling with alpha = 0.1 and differing initial temperature values.

**Linear With Alpha = 0.1:**



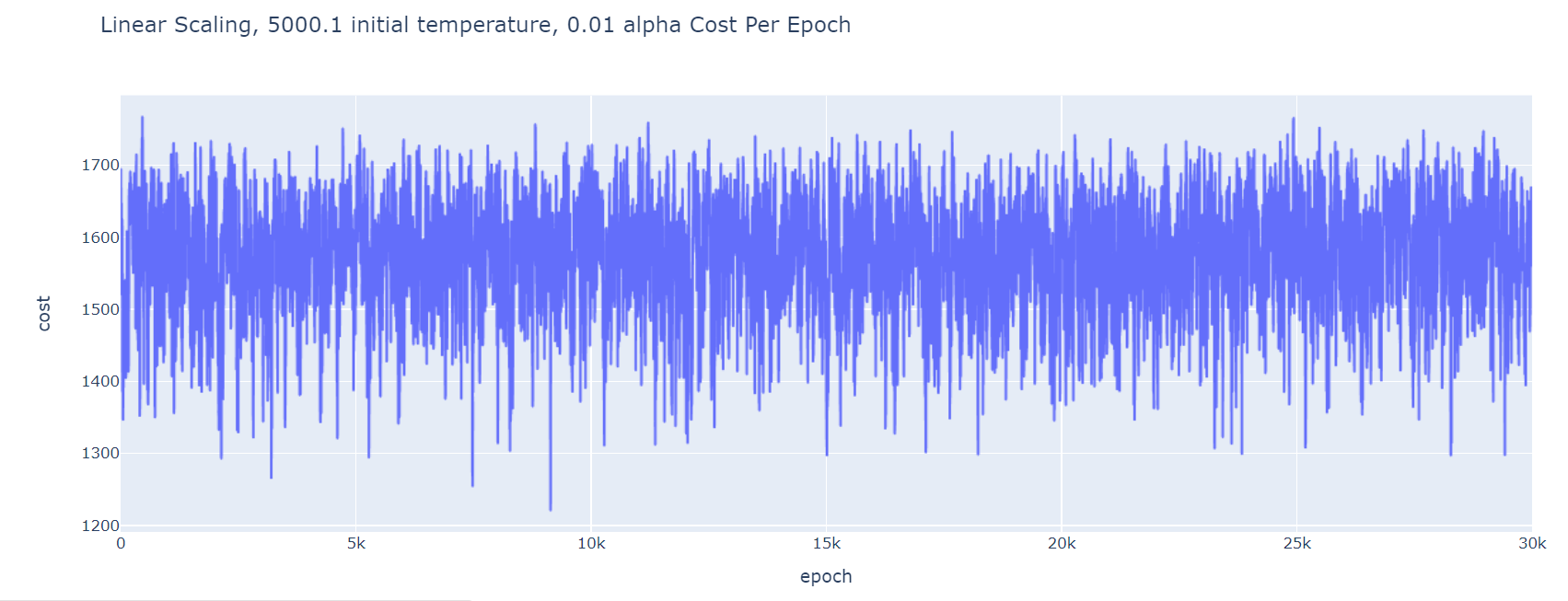
This algorithm does not appear to learn as time passes since the cost oscillates up and down (instead of decreasing), and it does not converge.



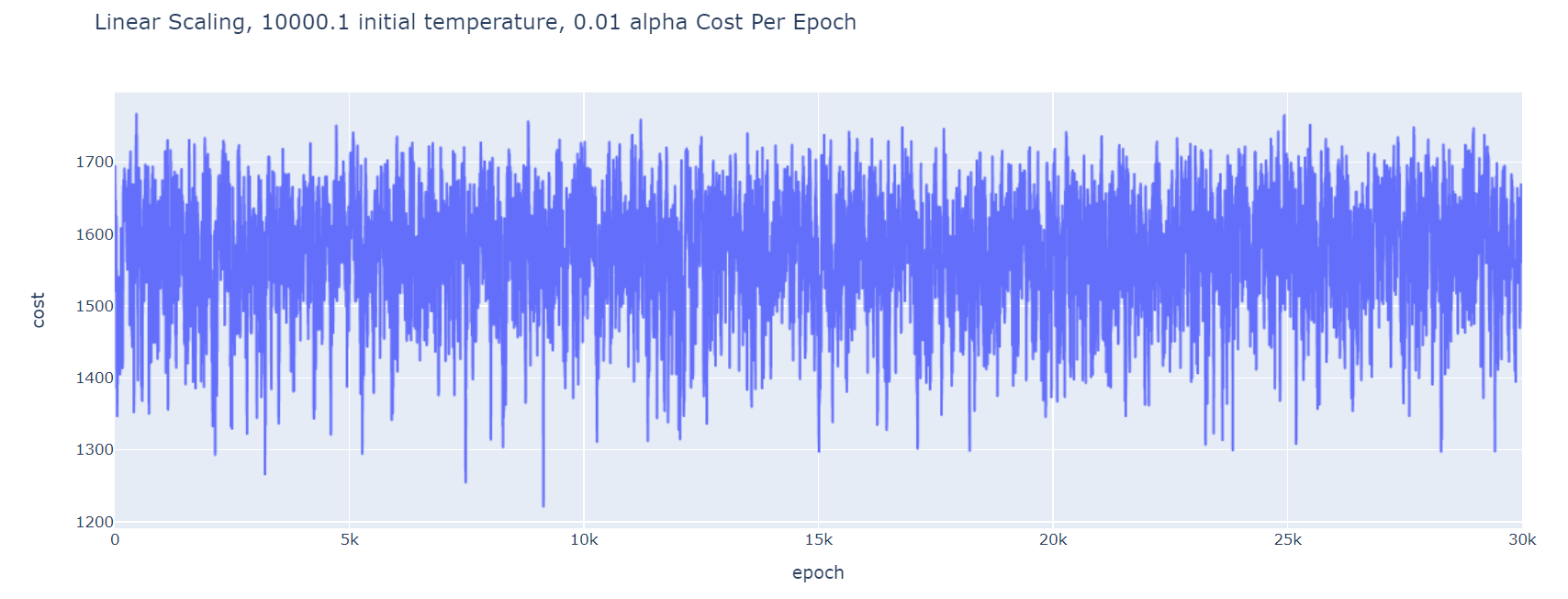
This algorithm also does not appear to learn as time passes since the cost oscillates up and down (instead of decreasing).

I will now analyze linear scaling when alpha = 0.01**.**

**Linear With Alpha = 0.01**



This algorithm also does not appear to learn as time passes since the cost oscillates up and down without a discernible downwards trend.



This algorithm does not appear to learn as time passes since the cost oscillates up and down (instead of generally decreasing).

I can understand why the linear scaling did not converge. With a low alpha – 0.1 and 0.01, the temperature decreases by an extremely small amount with each epoch and thus, the probability to accept “worse” moves decreases by a very small amount with each epoch. This explains why the cost jitters up and down – the rate at which “worse” solutions are accepted is barely being decreased as the epochs pass.

**SA Learning Curves Conclusion**

From these learning curves, I concluded that proportional scaling with alpha = 0.1 tends to be the configuration that performs the best. Changing the initial temperature value from 5,000.1 to 10,000.1 does not produce a noticeable difference.

Of all my configurations for SA, only 2 actually converged – proportional with alpha = 0.1 for 2 different initial value temperatures.

The configurations that do converge to a “good” solution and will be used in the next section for timing graphs are:

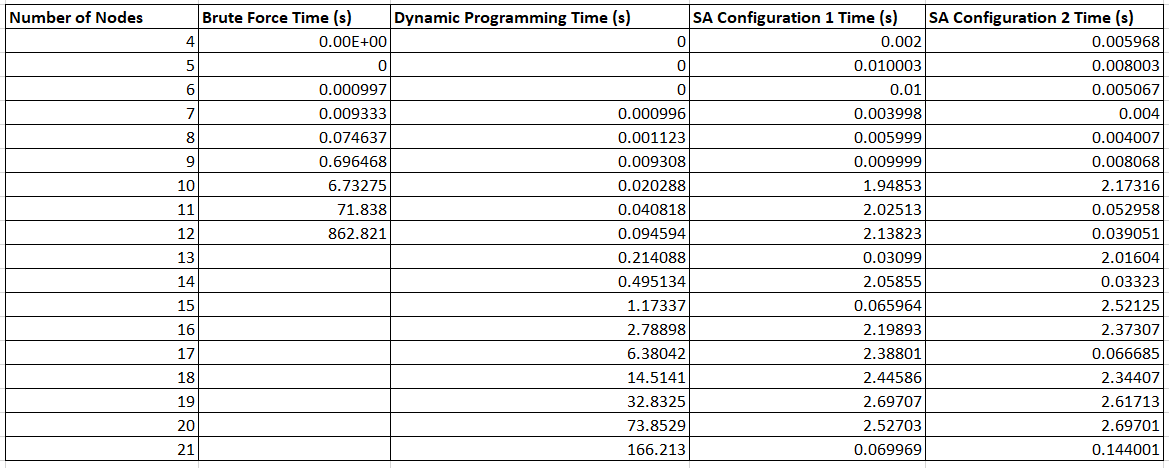
1. Proportional, alpha = 0.1, and initial temperature = 5,000.1
2. Proportional, alpha = 0.1, and initial temperature = 10,000.1

**SA Timing Data Comparisons**

In Lab 3, I implemented a brute force and dynamic programming algorithm and determined the best path and the associated cost given the number of nodes to visit. From lab 3, I know what the solution to the TSP is for each number of nodes 4 through 21. For this lab, I ran my 2 SA configurations, noted above, until they were within at least 5% of the best cost (known from Lab 3) and collected the associated timing data.

A table summarizing the data obtained is shown below.

**Table II. Brute Force, DP, SA Configurations Timing Data**

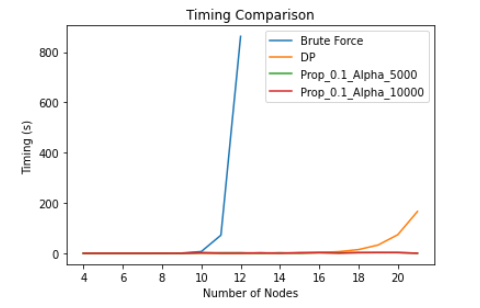


*Note: SA Configuration 1 is SA with Proportional Scaling, alpha = 0.1, and initial temperature = 5,000.1*

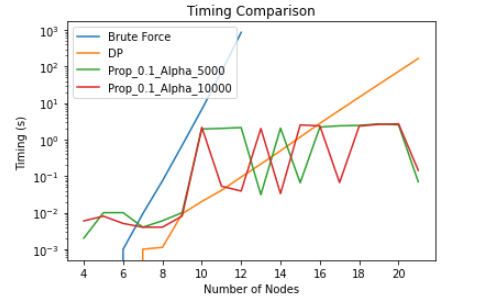
*Note: SA Configuration 2 is SA with Proportional Scaling, alpha = 0.1, and initial temperature = 10,000.1*

*Note: Timing with brute force above 12 nodes was not obtained due to timing growth*

The graphs displaying this data are below:



The two SA configuration lines are both straight lines as far as I can tell from this graph. To be able to more clearly see the timing data, I plotted this graph with a log scale on the y axis. The graph with a logarithmic y scale is shown below.



Analyzing the above graph, it is clear that the two different configurations of SA tend to be faster than the DP algorithm once the number of nodes is 17 or greater. The SA algorithms are better than brute force at about 7 nodes in the path. Between the two different SA configurations, one is not consistently faster than the other, though it seems as though the configuration with an initial temperature of 10,000.1 may be faster than the configuration with initial temperature of 5,000.1 more often than the reverse. Earlier, I concluded that these two configurations of SA had essentially the same performance, but based on this timing data, I will consider an initial temperature of 10,000.1 to be better than an initial temperature of 5,000.1.

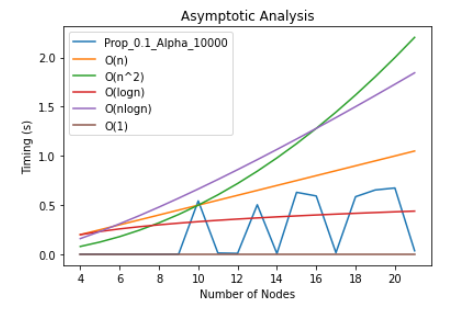
It makes sense that the SA algorithms tend to be faster than Brute Force and DP past a certain number of nodes since heuristic algorithms are meant to quickly find a good solution rather than methodically finding the absolute best solution.

**Analysis of Best SA Configuration**

I will now perform more analysis on the SA configuration I concluded to be the best of my configurations earlier.

This configuration was SA with proportional temperature scaling, an initial temperature of 10,000.1, and alpha = 0.1.

I attempted to determine the time complexity of this algorithm by plotting the timing data along with timing curves for common time complexities. I normalized the values before making this plot. The graph is shown below.

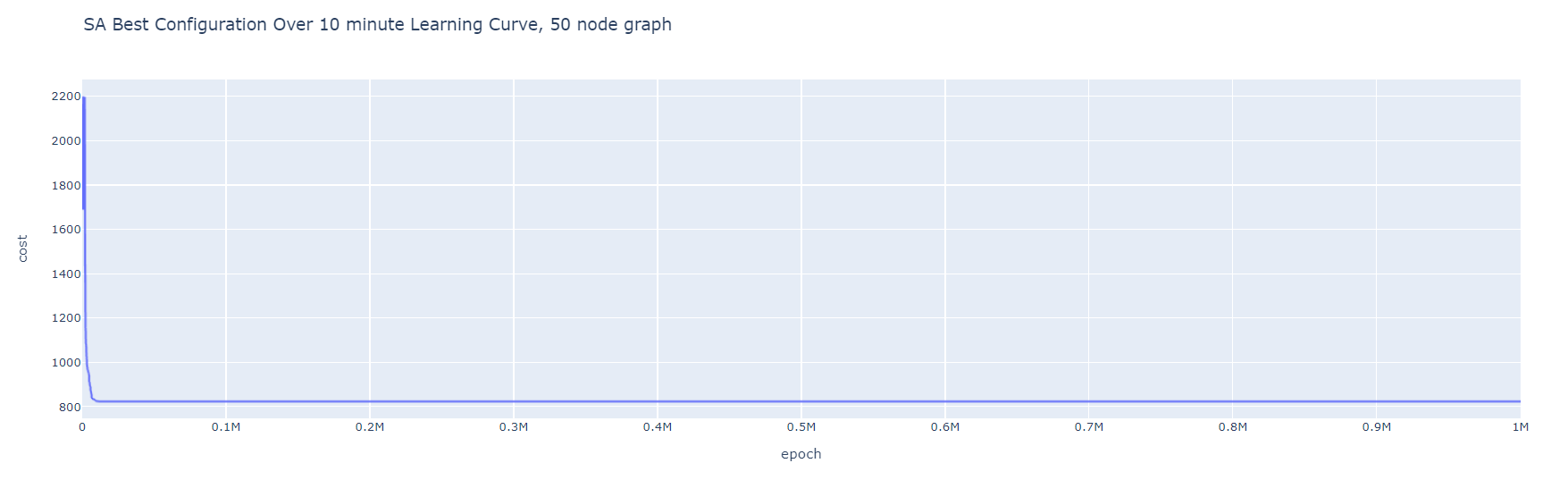


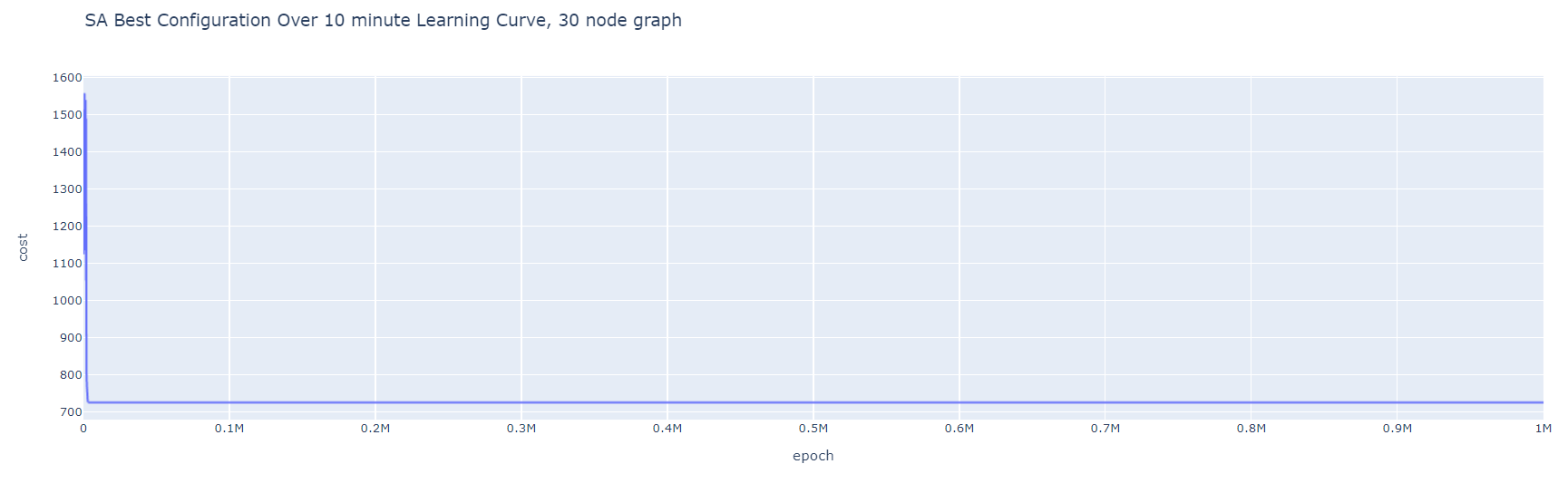
From the graph above, I conclude that this SA configuration has time complexity of O(n), where n is the number of nodes. Analyzing the time complexity of this heuristic algorithm by considering the operations of SA is quite difficult since there are so many varying parameters. Nonetheless, I believe my most expensive operation in the algorithm, which I do regardless of configuration, is generating the initial vector, which is size n. This operation is O(n), since I push back a value n times. This graph also gives a nice visual of how the timing scales as the number of nodes increases. This SA configuration appears to be bound by O(n). This time complexity is considerably faster than the time complexity of DP, which is O(n22n ) and Brute Force, which is O(n!).

It makes sense that the timing line for SA is quite jagged since how quickly the algorithm finds the solution is dependent on randomness.

To further analyze the intelligence of this configuration of SA, I have plotted a few learning curves below, which were all run for more than ten minutes on different graphs.

**10 Minute Learning Curves**



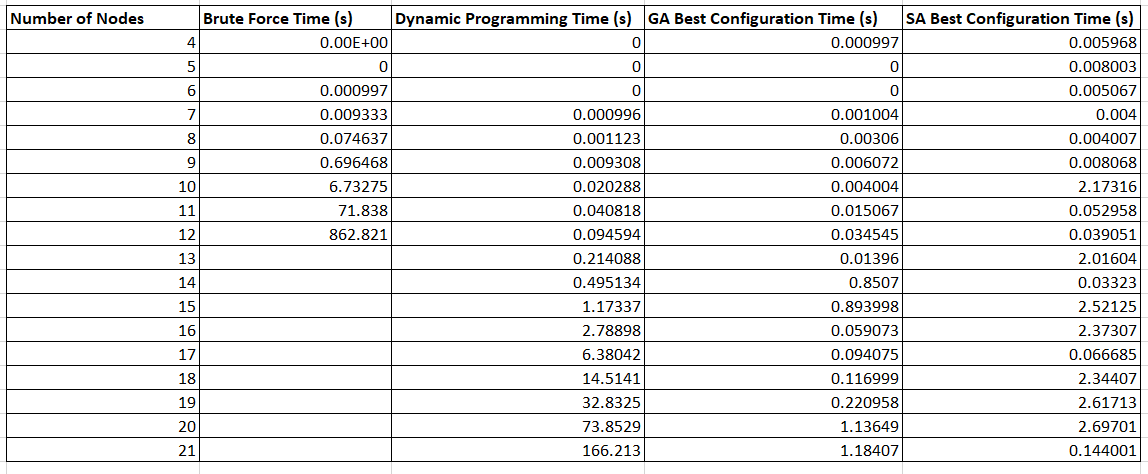


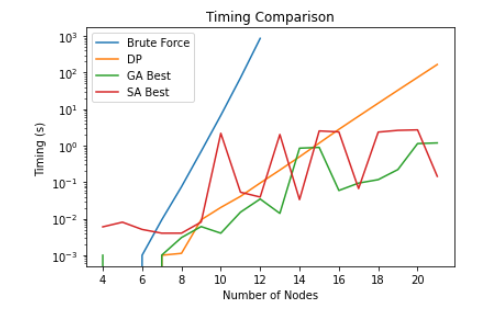
For both of these plots, we see the curve decreasing and converging, which supports that this configuration of SA learns, even on graphs of various sizes.

**Comparison of Node Timings**

To compare the timing of my best GA configuration, best SA configuration, DP, and Brute Force methods, I have provided a table and accompanying graph below.

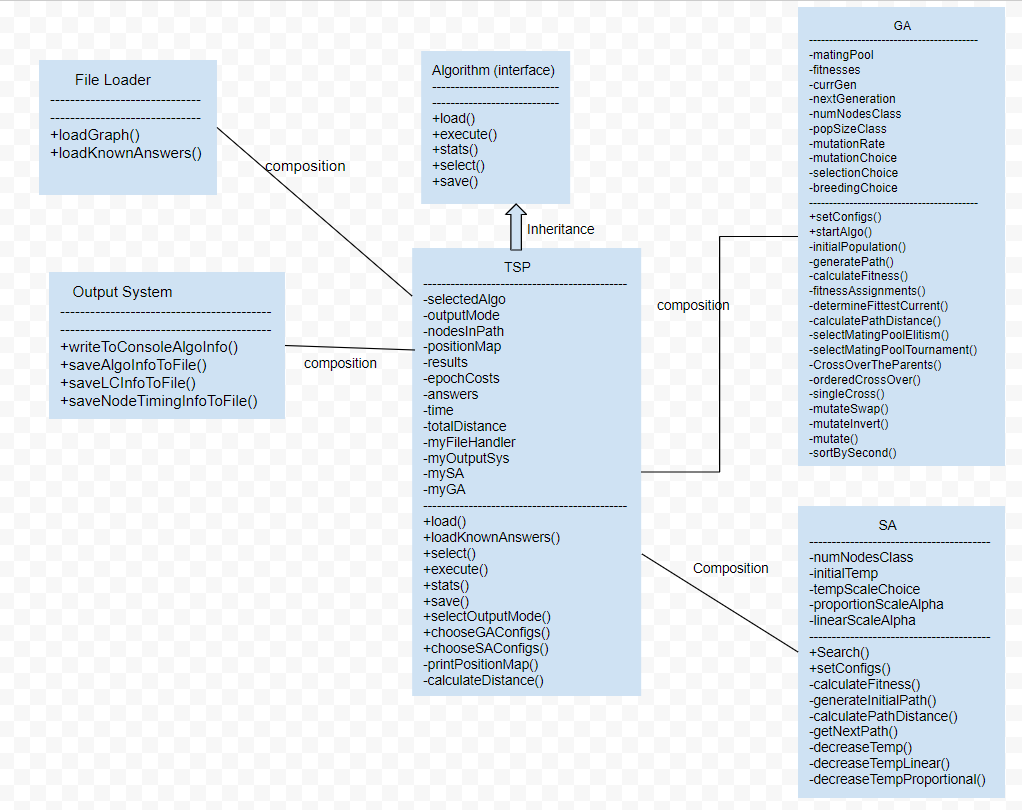
**Table III. Brute Force, DP, Best GA and Best SA Configurations Timing Data**





Analyzing the above graph, it appears as though the best GA configuration tends to find a good solution faster than the best SA configuration, though one is not consistently faster than the other. It is also evident that past about 17 nodes, the heuristic algorithms are faster than both DP and brute force. Even at 7 nodes, the heuristic methods find a good solution faster than the brute force method. As discussed earlier, it makes sense that the heuristic algorithms tend to be faster than Brute Force and DP past a certain number of nodes since heuristic algorithms quickly find a good solution rather than methodically finding the absolute best solution.

**UML and Design**



When designing my code, I had to deeply think about how configurable and extensible I wanted my design to be. Due to all of the data collection I needed to perform and all of the potential configurations I wanted a user to be able to choose from, I decided to base my design off of the Builder pattern and the Strategy Design pattern. The Builder pattern enables users to set their configurations for an object before using it. I utilized the Builder pattern so that a user had the ability to select GA or SA, and then for their selection of GA or SA, select specific parameters. For example, for GA, a user is able to not only specify the mutation choice, selection choice, and breeding choice, but also the population size and mutation rate. For SA, a user is able to specify the initial temperature, scaling type, and alpha value (value used for scaling). Additionally, users can specify an output mode to generate learning curve data or node timing data. A user merely configures all of the settings they want before calling execute(). I was extremely pleased with how configurable my design was.

I continued to utilize the Strategy Design Pattern in this lab by having my TSP class contain an instance of SA and GA and inherit from my Algorithms interface. Though not needed for this lab, I could easily include an instance of my Dynamic Programming class and Brute Force class in TSP and dynamically change between my Dynamic Programming, Brute Force, SA, and GA implementation, which is extremely powerful. I could also easily add classes that would inherit from Algorithms and maintain the interface established by the Algorithms class. My design is very extensible and functional.