Andrew Brkich

Traveling Salesman Problem: Heuristic Searches

The primary purpose of this project was to implement brute force, dynamic programming, simulated annealing, and genetic algorithms for the travelling salesman problem, and analyze the results, efficiency, feasibility, and runtime of each.

Brute Force:

The brute force algorithm for the travelling salesman problem has a big 0 (n!). This is because the brute force algorithm generates all possible paths/tours of a graph, and then selects the lowest cost path and returns it as our answer. For this, I used the permutations library in the STL, and added a simple condition that only allowed the permutations to evaluate paths that existed in our graph. While the permutations ran each possible path, I added if statements that only allowed the permutations library to continue along any given path if a connection to the next node existed. If a connection did not exist, the algorithm will break and continue evaluating the next possible permutation. After all paths are generated, we select our lowest cost path and return it. Evaluating cost after generating all paths is important because if we evaluated cost during each permutation, we would evaluate some distances in paths that do not connect and we will have wasted time doing our calculations for a path that does not exist. By waiting to calculate the distances of each path after we find all possible paths, we only calculate distances of paths that actually exist, improving our runtime.

Because of the brutal time complexity of our brute force algorithm, our sample graph of 16 nodes could not feasibly be thoroughly searched. The maximum number of nodes I could run the algorithm on within a feasible time (about 5 minutes) was approximately 13 nodes. In order to save time, in my calculations of the cost (or distance) between each node, I left out the square root function of the distance equation. I did this because during each possible path returned, we calculated the cost. This saves time, and which path is the lowest cost path will not be affected by leaving out the square root function. The only downside to this is if we needed to know the exact cost of the path: it will be larger than it actually is because after we square the x, y, and z values and add them together, we do not square root them in order to save time. That is a simple addition we can make to the code if we decide we want to know the exact cost or distance between nodes.

If we indeed correctly calculate our distances by adding the square root function to our mathematical equation, our time effectiveness gets even worse. The maximum number of nodes able to be searched for a path drops from 13 nodes down to 10. On a graph size of 16, this may not be the end of the world, but on a larger graph of thousands of nodes, this difference would be even more exaggerated. If we want the exact cost of our paths, and the exact distances between nodes, we sacrifice enormous amounts of time.

Dynamic Programming:

For my dynamic programming algorithm, I chose to implement a Held-Karp algorithm. This has a big 0 (2^n n^2) time complexity, significantly better than our brute force algorithm, but still not ideal. Because of this, it was very hard to find other implementations online, as most people describe in algorithm abstractly and theoretically. This makes sense because there are still better solutions out there, and most people would not want to implement a less effective solution for the travelling salesman problem. However, this is still significantly better than the brute force method, and we were able to very easily run this program on all 16 of our nodes. When running the algorithm with my own graph, I found my algorithm could feasibly run (within 5 minutes) up to a graph size of 21 nodes.

The Held-Karp- algorithm takes advantage of a key component of the travelling salesman problem: Every subpath of a path of minimum distance is itself of minimum distance. So, instead of looking at every possible permutation like our brute force algorithm, we use a bottom-up approach where all the information needed to solve the problem is calculated only once. We start at the smallest subpath, and work our way up to larger and larger subpaths. When we look at the larger subpaths, we have already calculated the smaller subpaths, so that's where we save time, because the smaller subpaths have already been computed. These time savings compound exponentially, so the more subpaths, the greater the savings when compared to a brute force solution. The biggest drawback is that it requires a lot of memory to store all this information.

The Held-Karp algorithm works first by acknowledging that the shortest Hamiltonian circuit does not depend on the starting node. No matter what starting node we choose, the shortest circuit will always be the same, so we acknowledge that no matter what source node we have, our algorithm should return the same circuit as another source node. The time complexity, 0 (2^n n^2, exists because the algorithm considers 2^n-2 subsets of our graph. For each subset, we compute the cost of the subset, and there are no more than n of them.

Again, our calculation of distances leaves out the square root function. If we want exact costs, we can simply add the correct equation, and because of the better time complexity of dynamic programming, we can still search all 16 nodes, albeit at a slightly slower pace.

For the vast majority of graphs, the dynamic programming algorithm with be significantly faster. However, for smaller sized graphs (below 7 specifically), the brute force algorithm was slightly faster. This is because n! of 4, for example, equals 24. 2^n n^2 of 4 is equal to 256. However, as the graphs get larger and larger, the brute force algorithm with n! timing will quickly overtake the dynamic programming algorithm. The brute force algorithm should still not be used even if using small graphs because it quickly grows into a terrible algorithm. Both algorithms return the shortest cost path, so the cost of a path for the same number of nodes for each algorithm will be exactly the same. My dynamic programming algorithm with return 9999999 if no legitimate path is found. This happened only once while I was collecting data, and can be fixed by editing our graph file if wanted.

Simulated Annealing:

The simulated annealing algorithm gathers its inspiration from the way metal workers craft their metals: by heating and cooling material to alter its physical properties. While the temperature of a metal is intensely hot, the atoms and molecules in the metal can bounce around all over the place with plenty of room to move. When the metal gets cooled in a tub of water or oil, the metal hardens and the molecules are stuck where they currently are. This is how metalworkers can bend metal into different shapes. In computer science, it is a metahuerristic to approximate global maxima or minima in a large search space. It is a optimization technique commonly used when the search space is discrete.

For our program, we set an initial 'temperature' and throughout each iteration of our code, the temperature slowly decreases. A more basic way to think about this would be to just set the number of iterations that our program will run. However, by using temperatures, we can adjust how slow/fast our cooling process happens. Additionally, while our temperature is hot, we can allow a large chance of randomness in our program, similar to how the molecules in a metal have a randomness and freedom to move around. For our program, the randomness is the probability that we can escape our current local optima. By shuffling our vector, we can escape our current state and randomly check other paths in our search space. As our temperature cools and our iterations continue, the chance for randomness in our program decreases, just as the molecules in a metal are allowed less and less room to roam freely. This means that there is a large chance for breadth early in the program, and a larger chance for depth later on. We do this in the code by simply rotating, swapping, or reversing the vector, as opposed to a complete random shuffle used at the high randomness stage.

Our program eventually ends once our temperature reaches a certain point that we manually set. Because of this, there is no point in our program where we "find the right answer" and exit our code. No matter the answers and optima we find, the program will always run the same number of iterations every time. Furthermore, by adding more nodes to a graph, we will not effect the runtime of our algorithm very much. What primarily changes instead is the accuracy of our algorithm. With a larger search space, the possibility of finding the correct optima decreases, and we would have to allow for more runtime. So unlike the brute force and dynamic programming algorithms, the possibility does exist with the simulated annealing algorithm that we do not in fact find our global optima. The "right answer" may never be found. What makes the algorithm useful is it's ability to run over large search spaces and its success rate. I found through trial and error that I can expect a graph up to 15 nodes to return either the optimal path or a path very close to the best possible within a second. The primary perk to the algorithm is its extreme quickness: It continued to run fast,within a second or two, despite a graph with a large number of nodes and connections such as 30 or 40, although it was not the most accurate at that level.

For my code, I chose to use an initial temperature of 999, with a cooldown rate of .1. This comes out to about 10,000 iterations. For my randomness scale, I started with a 99% probability with a cooldown rate of .01% per iteration. I also skewed my randomness cooldown rate toward a semi-exponential scale by adding .001% to the cooldown rate every iteration, so the cooldown rate for randomness increases each iteration. This scale helped keep the randomness at the start of the algorithm high, and eventually cooled down fast toward the end.

I chose these numbers and the cooling rate to model real life annealing, and the number of iterations proved to be enough to ensure accuracy while also having a short runtime.

The one limitation of my code is that the possibility exists that the simulated annealing algorithm returns a path that is not possible: my algorithm assumes every node can connect to every other node. With the constant breeding and mutating, it would add an extraordinary amount of time to constantly be checking throughout each iteration if each parent and child was indeed a plausible path. However, this is extremely unlikely because the algorithm runs through 10,000 iterations. The odds of never finding a path 10,000 straight occurences would be extremely close to impossible.

Genetic Algorithm:

The genetic algorithm process is derived from the way animals in nature breed: parents crossing genes to create a child, mutations, artificial selection, etc. When applied to code, the algorithm works first by declaring a random population of a given size. Then over a set number of iterations, we select parent nodes by using elitism and breed an entire set of children matching in size to our original population. This happens by splitting our vectors, and taking two halves from two parents and forming a child. We also allow for mutations to happen, and on this occurrence we swap two random elements in a child vector. It is by this randomess that we can examine a large search space and not worry about being trapped in local maxima or minuma, although it is technically still possible depending on how random we want our algorithm to be.

Because the genetic algorithm runs for a specific set of iterations, the algorithm ends at the same point each time it is run. Just like the simulated annealing algorithm, it is possible to not find the “right answer”, or the least cost path. However, because of the randomness and large number of iterations, we explore the vast majority of our search space, and the genetic algorithm performs signifcantly better with regards to speed than both the brute force and dynamic programming algorithms. In fact, it also exceedingly outperforms the simulated annealing algorithm with respect to accuracy of path returned. The genetic algorithm always returned a path notibly closer to the optimal path, while still executing in a realistic timeframe. It still can perform well on a large graph the size of 30 or 40 nodes.

For my code, I chose to use a mutation rate of .075%. This was a small enough number to not have too much of a difference in the code, and prevented mass mutations that resulted in a random search. It was also large enough to actually cause some mutations that could prevent my search from being trapped in local minima/maxima. I also chose to use elitism for my breeding techniques. I did this over 5000 generations, a number which proved to be enough to ensure accuracy while trying to minimize runtime. The algorithm runs feasibly over 5000 generations all 35 nodes provided in the graph file.

The one limitation of my code is that the possibility exists that the genetic algorithm returns a path that is not possible: my algorithm assumes every node can connect to every other node. With the constant breeding and mutating, it would add an extraordinary amount of time to constantly be checking throughout each iteration if each parent and child was indeed a

plausible path. However, over 5000 generations and with the introduction of elitism, it would be almost impossible for this to happen.

Data:

Brute Force vs Dynamic Programming:

As you can see in figure 1, the dynamic programming algorithm significantly outsped the brute force algorithm. The dynamic programming algorithm was also able to feasibly run with a graph size almost double the size our brute force algorithm could. Both of these algorithms returned the least cost path, exactly identical to one another.

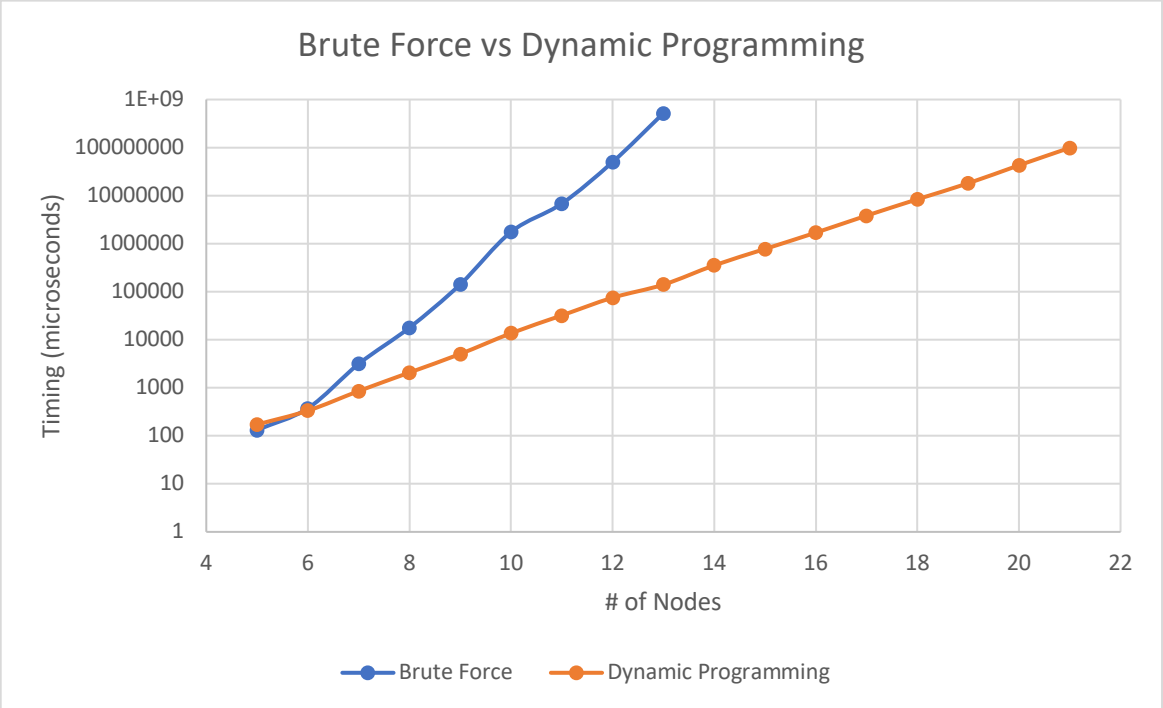

Figure 1

Simulated Annealing vs Genetic Algorithm:

Pictured in the graph below is the same comparison of timing for our two hueristic searches. As pictured below in figure 2, the simulated annealing algorithm significantly outperformed the genetic algorithm with regards to timing. However, as pictured figure 3, the genetic algorithm almost always returned shorter cost paths, and the distance between the two became significantly greater with larger graph sizes.

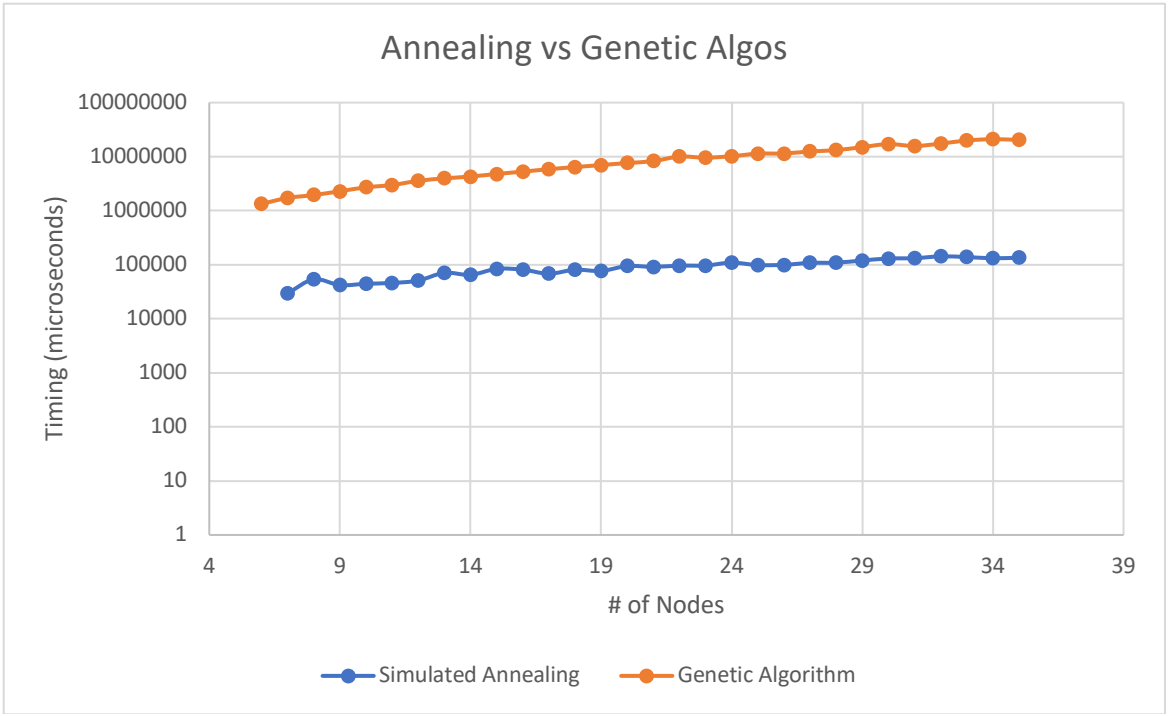

Figure 2

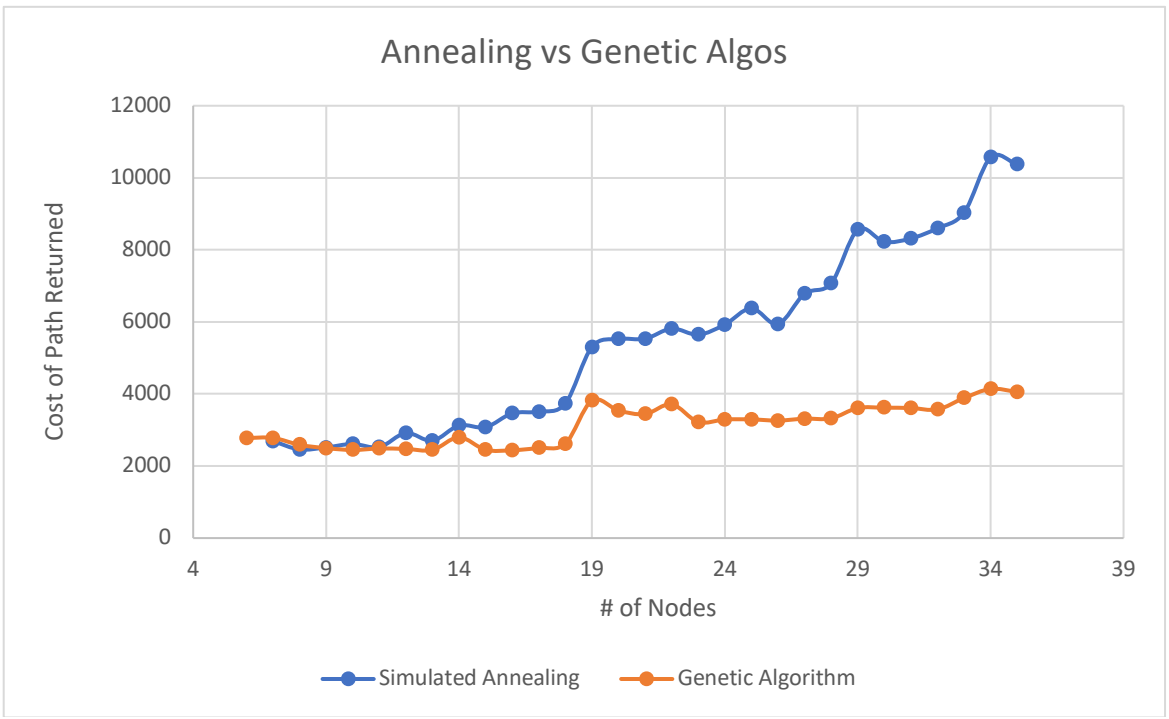

Figure 3

All Four Algorithms:

As seen below in figure 4, the simulated annealing and genetic algorithm techniques were able to run significantly quicker for larger sized graphs. However, due to the complexity of the algorithms, for small sized graphs sized about 5 through 9, the simple and naïve brute force and dynamic programming algorithms are quicker. As seen in figure 5, all four algorithms return either the optimal or close to the optimal path up until a graph size of about 20 nodes. This is where the difference between simulated annealing and genetic algorithm are obvious. Simulated annealing dominates time, and genetic algorithm dominates cost of path returned.

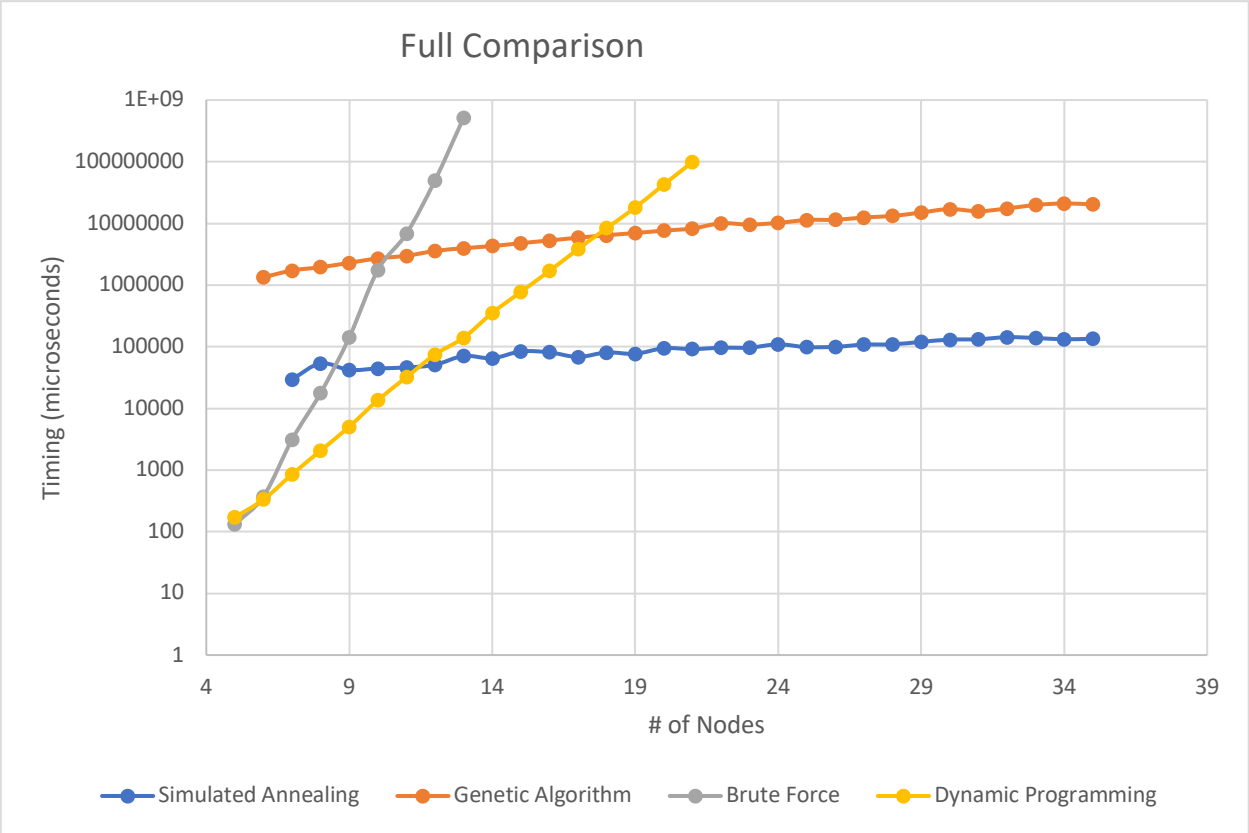

Figure 4

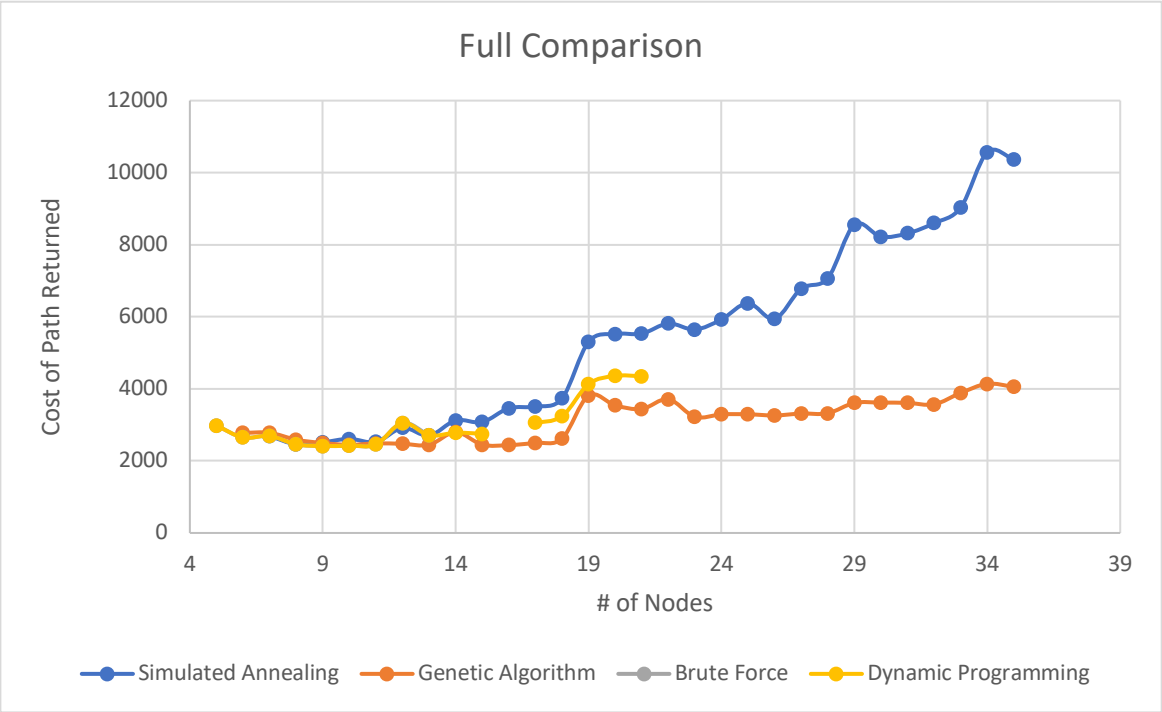

Figure 5

Raw Data:

Algorithm	Timing (microseconds)	Cost of Path	# of Nodes
Brute Force	132	2980	5
Brute Force	374	2650	6
Brute Force	3133	2692	7
Brute Force	17824	2460	8
Brute Force	141853	2414	9
Brute Force	1751211	2424	10
Brute Force	6830051	2456	11
Brute Force	50049751	3056	12
Brute Force	518647008	2708	13
Dynamic Programming	172	2980	5
Dynamic Programming	333	2650	6
Dynamic Programming	853	2692	7
Dynamic Programming	2074	2460	8
Dynamic Programming	5054	2414	9
Dynamic Programming	13628	2424	10
Dynamic Programming	32067	2456	11
Dynamic Programming	75006	3056	12
Dynamic Programming	140143	2708	13

Dynamic Programming	353052	2782	14
Dynamic Programming	774454	2752	15
Dynamic Programming	1692540	No path	16
Dynamic Programming	3830945	3058	17
Dynamic Programming	8391044	3238	18
Dynamic Programming	18050414	4132	19
Dynamic Programming	42762758	4360	20
Dynamic Programming	98716389	4352	21
Simulated Annealing	29709	2692	7
Simulated Annealing	53656	2460	8
Simulated Annealing	42021	2516	9
Simulated Annealing	44449	2612	10
Simulated Annealing	46067	2524	11
Simulated Annealing	50947	2916	12
Simulated Annealing	70912	2702	13
Simulated Annealing	64993	3124	14
Simulated Annealing	83843	3078	15
Simulated Annealing	81082	3462	16
Simulated Annealing	68286	3506	17
Simulated Annealing	80856	3734	18
Simulated Annealing	76796	5304	19
Simulated Annealing	94520	5526	20
Simulated Annealing	91554	5532	21
Simulated Annealing	96246	5814	22
Simulated Annealing	96876	5646	23
Simulated Annealing	109207	5926	24
Simulated Annealing	98558	6378	25
Simulated Annealing	99621	5940	26
Simulated Annealing	108171	6782	27
Simulated Annealing	109275	7070	28
Simulated Annealing	120014	8556	29
Simulated Annealing	130327	8222	30
Simulated Annealing	132234	8316	31
Simulated Annealing	142827	8600	32
Simulated Annealing	138648	9032	33
Simulated Annealing	131932	10568	34
Simulated Annealing	135039	10372	35
Genetic Algorithm	1337583	2778	6
Genetic Algorithm	1723447	2780	7
Genetic Algorithm	1947488	2588	8

Genetic Algorithm	2291912	2496	9
Genetic Algorithm	2707575	2446	10
Genetic Algorithm	2987200	2486	11
Genetic Algorithm	3571312	2474	12
Genetic Algorithm	3961549	2446	13
Genetic Algorithm	4286966	2798	14
Genetic Algorithm	4781397	2450	15
Genetic Algorithm	5285840	2438	16
Genetic Algorithm	5856704	2502	17
Genetic Algorithm	6417594	2618	18
Genetic Algorithm	6987392	3816	19
Genetic Algorithm	7627218	3542	20
Genetic Algorithm	8264599	3442	21
Genetic Algorithm	10033700	3714	22
Genetic Algorithm	9570025	3222	23
Genetic Algorithm	10193856	3292	24
Genetic Algorithm	11359305	3294	25
Genetic Algorithm	11449324	3262	26
Genetic Algorithm	12490761	3314	27
Genetic Algorithm	13196190	3322	28
Genetic Algorithm	15031128	3610	29
Genetic Algorithm	17082963	3618	30
Genetic Algorithm	15734126	3610	31
Genetic Algorithm	17419259	3570	32
Genetic Algorithm	19915619	3886	33
Genetic Algorithm	21048979	4140	34
Genetic Algorithm	20329221	4058	35