Nicholas Crothers

CS3353

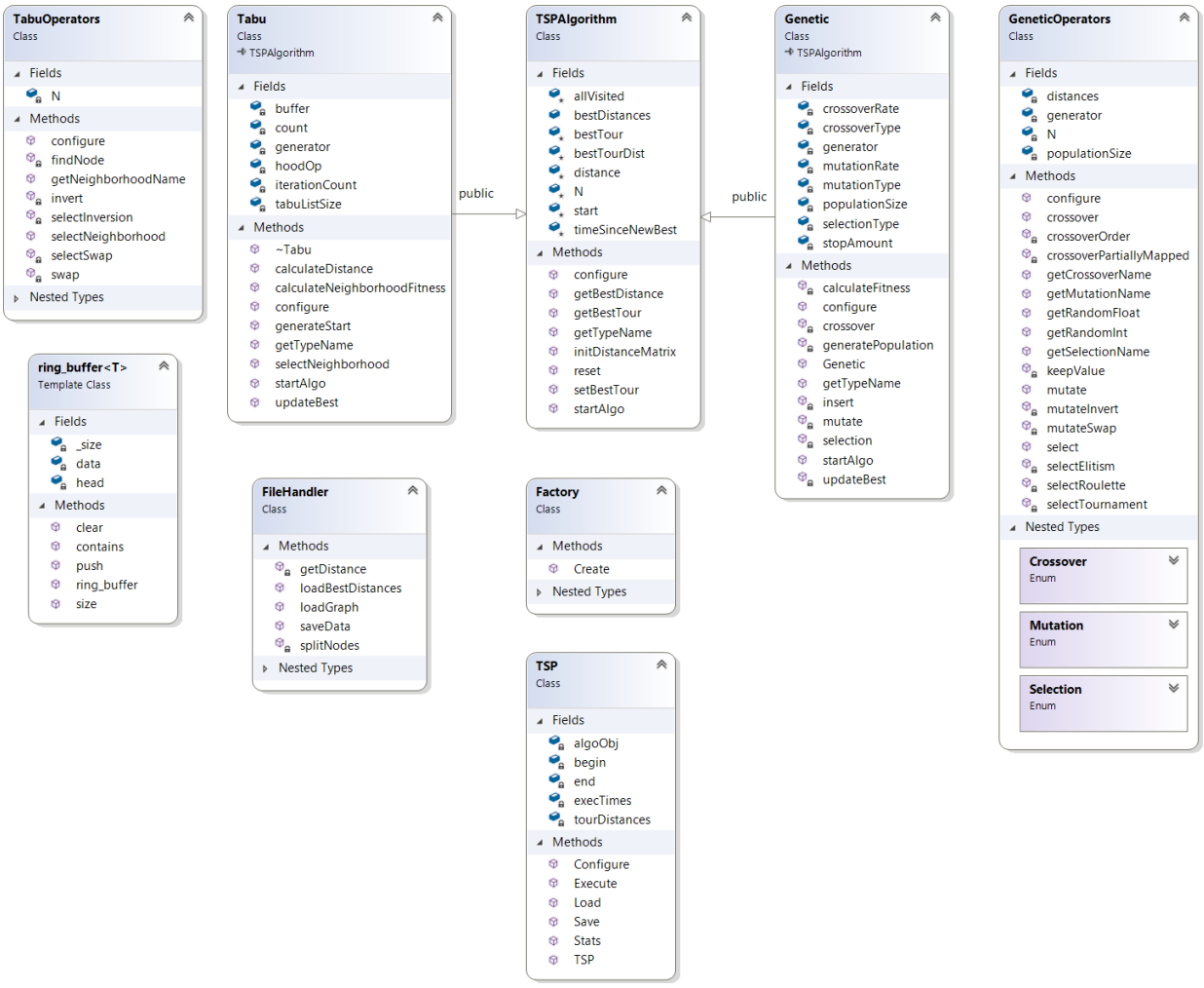
Lab 4 Report

Code Structure

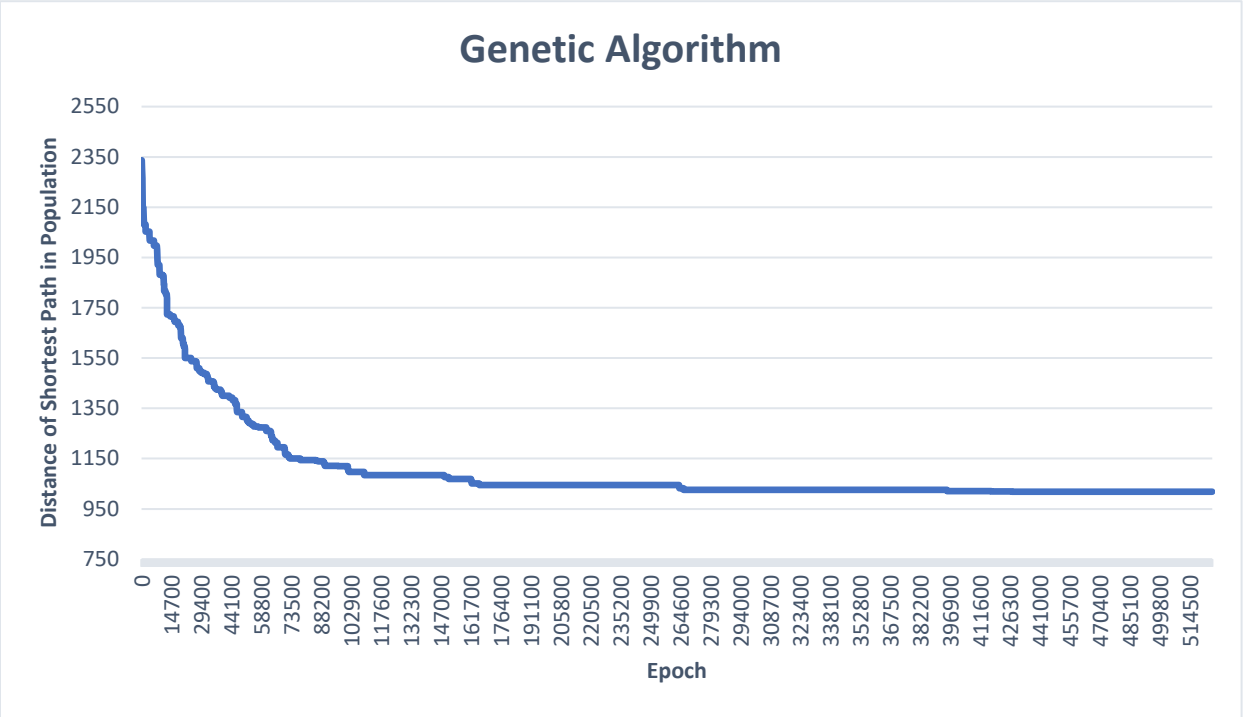
For this lab, I used a similar code structure to the one I created for lab 3. It is a mix between a Factory pattern and a Strategy pattern. The Factory creates the objects based on parameters passed in, but it uses the Strategy pattern to configure the object to use different functions through the `Configure()` function. This function allows the user to change the operators/parameters for Genetic Algorithm and Tabu Search.

The `TSP` object stores algorithm data over multiple iterations through different tour sizes. This information doesn't need to be known by the implemented algorithm class, so it is stored in this object. It stores a vector of the execution times and the best tour distance found during the execution. The only new function in `TSP`, `Configure()`, allows you to pass arguments that configure the `TSPAlgorithm` object. Besides this, `Load()`, `Execute()`, `Stats()`, and `Save()` all have the same functionality as the previous lab and in the example given to us in class. `FileHandler` handles all reading and writing to files; it has static functions that can be called to write things to files, and it doesn't include any user-defined headers, so it is completely self-contained.

UML Diagram


TabuOperators
Class
Fields
N
Methods
configure
findNode
getNeighborhoodName
invert
selectInversion
selectNeighborhood
selectSwap
swap
Nested Types
Tabu
Class
TSPAlgorithm
Fields
buffer
count
generator
hoodOp
iterationCount
tabuListSize
Methods
~Tabu
calculateDistance
calculateNeighborhoodFitness
configure
generateStart
getTypeName
selectNeighborhood
startAlgo
updateBest
public
TSPAlgorithm
Class
Fields
allVisited
bestDistances
bestTour
bestTourDist
distance
N
start
timeSinceNewBest
Methods
configure
getBestDistance
getBestTour
getTypeName
initDistanceMatrix
reset
setBestTour
startAlgo
public
Genetic
Class
TSPAlgorithm
Fields
crossoverRate
crossoverType
generator
mutationRate
mutationType
populationSize
selectionType
stopAmount
Methods
calculateFitness
configure
crossover
generatePopulation
Genetic
getTypeName
insert
mutate
selection
startAlgo
updateBest
GeneticOperators
Class
Fields
distances
generator
N
populationSize
Methods
configure
crossover
crossoverOrder
crossoverPartiallyMapped
getCrossoverName
getMutationName
getRandomFloat
getRandomInt
getSelectionName
keepValue
mutate
mutateInvert
mutateSwap
select
selectElitism
selectRoulette
selectTournament
Nested Types
Crossover
Enum
Mutation
Enum
Selection
Enum
ring_buffer<T>
Template Class
Fields
_size
data
head
Methods
clear
contains
push
ring_buffer
size
FileHandler
Class
Methods
getDistance
loadBestDistances
loadGraph
saveData
splitNodes
Nested Types
Factory
Class
Methods
Create
Nested Types
TSP
Class
Fields
algoObj
begin
end
execTimes
tourDistances
Methods
Configure
Execute
Load
Save
Stats
TSP

Genetic Algorithm Ten Minute Learning Curve

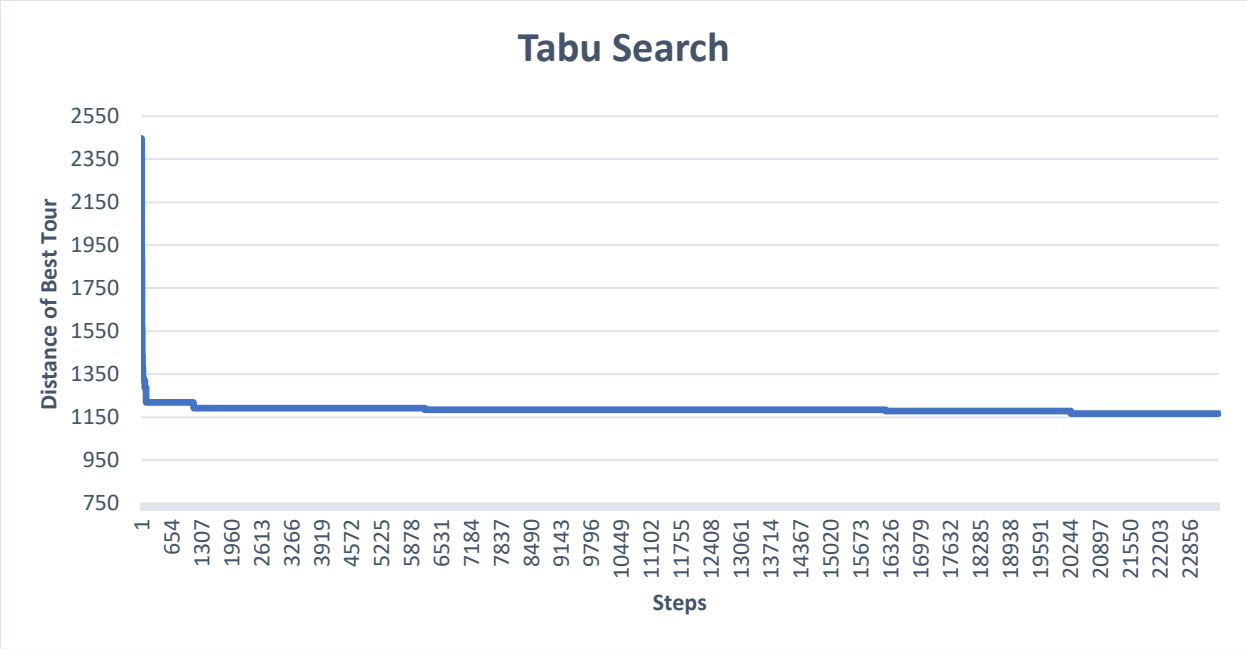


The learning curve shown above was run using a tour of length 40, configured as follows: selection using Roulette Wheel, crossover using Partially-Mapped (PMX), mutation using Inversion. The data follows the curve we expect for a "learning" algorithm to follow; as more generations are created and processed, the fitness increases quickly early on, then slows down. The lowest distance reached is 1018.26, and it hits this point by just over 400,000 generations, remaining constant until the end of the algorithm's execution. Each previous found decrease in distance was a small change, only decreasing by about seven units over the course of around 250,000 generations after the graph levels out completely in the graphic.

I will discuss this more later on in the report, but I believe that, based on the number of generations that are processed in only ten minutes, the crossover process should look for the best sections of each gene to "lock" and ensure they are carried on to their children. In the current version of the algorithm, the best portion of *each* parent is being passed on to their children, but

this is not the only approach. Instead, the algorithm could go through all of the genes in the population and note which gene section(s) is present in the most genes and/or note which section(s) has the lowest distance and ensure that section is carried over to the parents' children. This would be a more intelligent crossover design; the added computations needed to calculate this would likely be overshadowed by the increase in the intelligence and efficiency of the algorithm as a whole to move towards good solutions.

Tabu Search Ten Minute Learning Curve

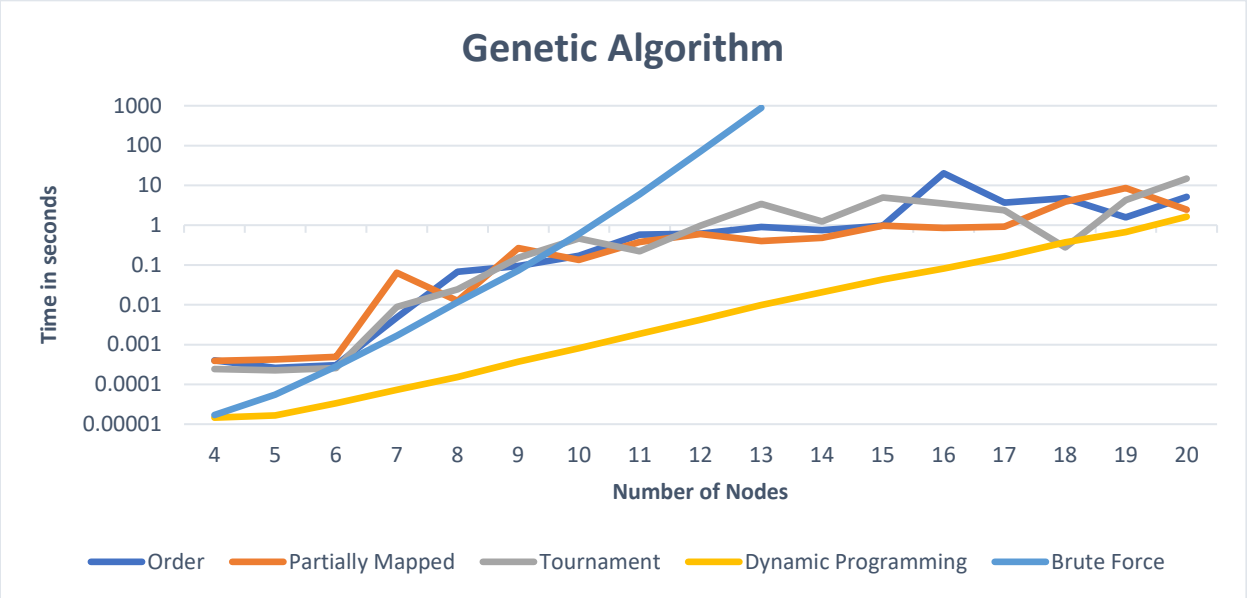


The learning curve shown above was run using a tour of length 40, configured with neighborhood selection using Swap and a Tabu list of size 100. The data forms a very sharp curve, drastically decreasing at the beginning in the first 150 steps or so, then levelling off. It still slowly decreased, but the lowest it reached was 1166.18; this is higher than the distance reached by Genetic Algorithm (1018.26), so it clearly did not find the best path. The algorithm does a random restart when the number of steps since the best tour was updated reaches 500. The decreases later in the algorithm are likely due to random guesses finding a better solution. The quick decrease

early in the algorithm's execution is most likely due to it getting stuck in a local minimum. Even with random restarts, the sheer number of possible paths for 40 nodes prevents the algorithm to efficiently find a shorter path.

One of the limitations of Tabu Search is how easily it becomes stuck in local extrema within the solution space. This is shown here in the graph, as Tabu converges very quickly to a lower distance, but it is not the best path, since the Genetic Algorithm reached a lower distance. This can be remedied by having a wide neighborhood selection to ensure the algorithm leaves a local extreme in combination with the Tabu list.

Genetic Algorithm Performance



The graph above compares the different combinations of genetic operators against each other and against the dynamic programming and brute force results. As shown in the graph, Genetic Algorithm (GA) performs worse than dynamic programming until around 20, where the increase in execution time starts to level out. GA performs better than brute force for values larger

than 10, and the execution time grows much slower than brute force. Additionally, the growth trend in execution time becomes lower than the growth trend for dynamic programming.

While each of the different combinations performed similarly, partially-mapped crossover (PMX) was the best crossover, roulette wheel selection was the best selection, and inversion was the best mutation. For all lines on the graph, the default operators were roulette wheel selection, partially-mapped crossover, and inversion mutation. There was also a swap mutation operator, but it couldn't find the best path for up to 20 nodes, so it didn't make the cut. Because of this, it is clear that the swap operator performs worse than the inversion operator. Based on the data, the best operators are roulette wheel, partially-mapped, and inversion.

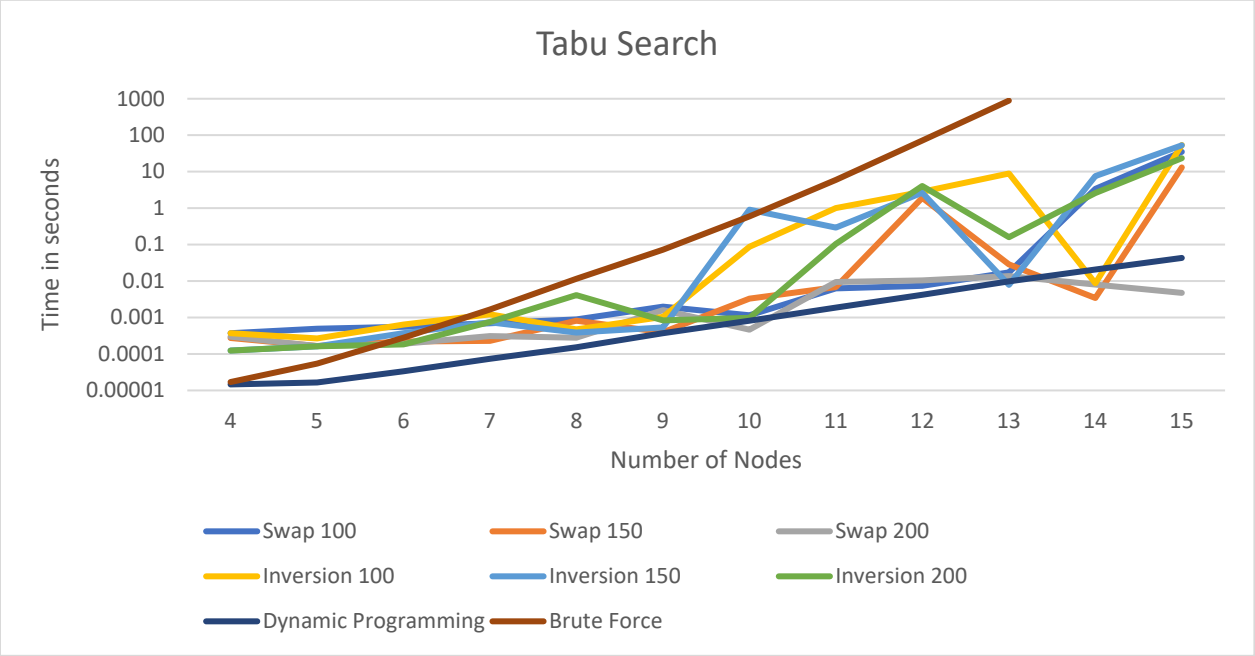
Roulette wheel selection works well because it gives a non-zero chance of being selected within the population, and each gene's probability of being selected is proportional to its fitness. Because of this, better solutions have a higher chance of being selected for breeding, but worse solutions will also be selected. This is important because worse solutions can have certain sequences of nodes that are pivotal for the algorithm to find the shortest path, helping prevent the algorithm from becoming stuck in local extrema.

Partially-mapped crossover is a rather complicated process. However, because it takes a subtour of a parent that contains the shortest link between three nodes, it carries over genetic information that is likely to result in the best solution. The addition of the nodes from the second parent fills out the remaining nodes in the child by mapping it close to where they were within that parent.

Inversion mutation works well because it introduces more genetic diversity within the population by reversing the order of a subtour within an offspring as well as only making a short

"jump" away from the original solution. It changes two of the links within a gene, but, because of the reversed order of the nodes in between those two links, more diversity is introduced for the crossover process. This helps maintain genetic diversity while improving the population's overall fitness.

Tabu Search Performance



The graph above shows the different combinations of operators for Tabu Search. The data only uses 15 nodes as the upper bound because Tabu stopped consistently finding the shortest path above this point, so I set the upper bound to be 15 nodes. As shown in the graph, the Tabu data falls almost completely within the brute force and dynamic programming data, which is interesting. Swap 200 has a consistently low execution time growth rate, but the reason for this isn't clear. The other operators fluctuate rather wildly, and this is likely largely due to the random element of Tabu for not only the selection of a starting position, but also doing random restarts throughout execution. In the configuration for the algorithm I ran, the algorithm would jump to a random point after 500 steps since the best path found was updated.

Based on the results, it is difficult to come to a conclusive "best" configuration, but, overall, it appears that swap consistently grows slower than inversion. There isn't enough data to say this with certainty, but it appears that the swap neighborhood selection chooses a broad yet related subset of solutions to choose from. Based on the data collected, it doesn't appear that Tabu list size affects the efficiency of the algorithm in a significant way, given that the 100 and 200 cross points in a few places, but I doubt this is the case. I would expect a Tabu list that is too small wouldn't do a good job of preventing the algorithm from going to previously visited areas and getting stuck in local extrema; at the same time, a Tabu list that is too large could skip areas of the solution space because the "direction" the algorithm should go is blocked by the Tabu list.

Swap selection works well because it swaps every possible combination of nodes, creating a neighbor for each swap operation. This produces a large neighborhood while maintain a close distance between each neighbor and the source node. This allows the algorithm to cast a large "net," increasing the chances of finding a better solution and path to follow. This works better than inversion because the swap neighbors are closer to the source node, so the algorithm is less likely to skip over better solutions.

Summary Tables

Tabu Search Performance Comparison								
Nodes	Swap 100	Swap 150	Swap 200	Inversion 100	Inversion 150	Inversion 200	Dynamic Prog.	Brute Force
4	0.0003709	0.0002733	0.0002977	0.000369253	0.000124298	0.000123663	1.46E-05	1.70E-05
5	0.0004953	0.0001656	0.0001641	0.000266905	0.000161713	0.000163062	1.66E-05	5.48E-05
6	0.0005576	0.00022	0.0001905	0.000635671	0.000371501	0.000183373	3.35E-05	0.000280721
7	0.0006886	0.0002307	0.0003105	0.00122475	0.000736012	0.000755638	7.33E-05	0.00165383
8	0.0008899	0.0008196	0.0002798	0.000470896	0.00038736	0.00414842	0.000153743	0.0116061
9	0.0019797	0.0003954	0.0015965	0.00104246	0.000537989	0.000838171	0.000371626	0.0722191
10	0.0011587	0.0032787	0.0004631	0.0874685	0.911645	0.000982809	0.000815163	0.598455
11	0.0062735	0.0068129	0.0094234	0.995382	0.290679	0.10499	0.00185637	5.99126
12	0.00736	1.92974	0.0104682	2.82501	2.65681	4.01605	0.00423834	70.8648
13	0.0174222	0.028714	0.0136949	8.83047	0.00786891	0.158052	0.00984726	887.157
14	3.34805	0.0034016	0.0079436	0.00865183	7.63215	2.58679	0.02062	
15	34.9786	13.0624	0.004725	53.972	52.8303	23.4025	0.0430176	

Genetic Algorithm Performance Comparison						
Nodes	Order	Partially Mapped	Roulette	Tournament	Dynamic Programming	Brute Force
4	0.000398	0.00039	0.00039	0.000241	1.46E-05	1.70E-05
5	0.00026	0.000426	0.000426	0.000225	1.66E-05	5.48E-05
6	0.000305	0.000492	0.000492	0.000255	3.35E-05	0.000281
7	0.004824	0.0641	0.0641	0.008812	7.33E-05	0.001654
8	0.067173	0.01263	0.01263	0.024587	0.000154	0.011606
9	0.09404	0.267804	0.267804	0.153237	0.000372	0.072219
10	0.171869	0.133265	0.133265	0.461824	0.000815	0.598455
11	0.580302	0.375265	0.375265	0.219723	0.001856	5.99126
12	0.616734	0.594789	0.594789	0.972291	0.004238	70.8648
13	0.899734	0.398449	0.398449	3.43161	0.009847	887.157
14	0.743419	0.477719	0.477719	1.2244	0.02062	
15	0.963836	0.962038	0.962038	4.95253	0.043018	
16	20.1378	0.845588	0.845588	3.48619	0.081114	
17	3.71251	0.924028	0.924028	2.34656	0.163674	
18	4.76955	3.89169	3.89169	0.277709	0.371077	
19	1.58307	8.57008	8.57008	4.24336	0.675246	
20	5.17871	2.43819	2.43819	14.765	1.63143	

Time Complexity

Because both Tabu Search and Genetic Algorithm are non-deterministic algorithms, they cannot be given a representation in Big-O notation. However, they can be compared to the timing of dynamic programming and brute force.

For Genetic Algorithm (GA), it has a higher starting execution time than both dynamic programming and brute force. With lower tour lengths, the execution time of GA grows quite rapidly, tapering off in its growth rate as the length of the tour increases. Because of this, it could be said that GA would have a relative complexity that is “lower” than dynamic programming; again, however, GA is non-deterministic, so it cannot be assigned a quantitative time complexity range.

For Tabu Search (TS), it also has a higher starting execution time than both dynamic programming and brute force. The growth of execution time as the length of the tour increases is less than brute force but seems to be around the same as dynamic programming, at least given the data that was collected. Given this, it appears that TS isn’t a great algorithm to use for the traveling salesman problem. If the time it takes to solve a problem grows at around the same rate as a dynamic programming approach, then using the deterministic proof of solving the problem would be a more logical approach than using a non-deterministic meta-heuristic search that approaches the best solution but doesn’t necessarily reach it.