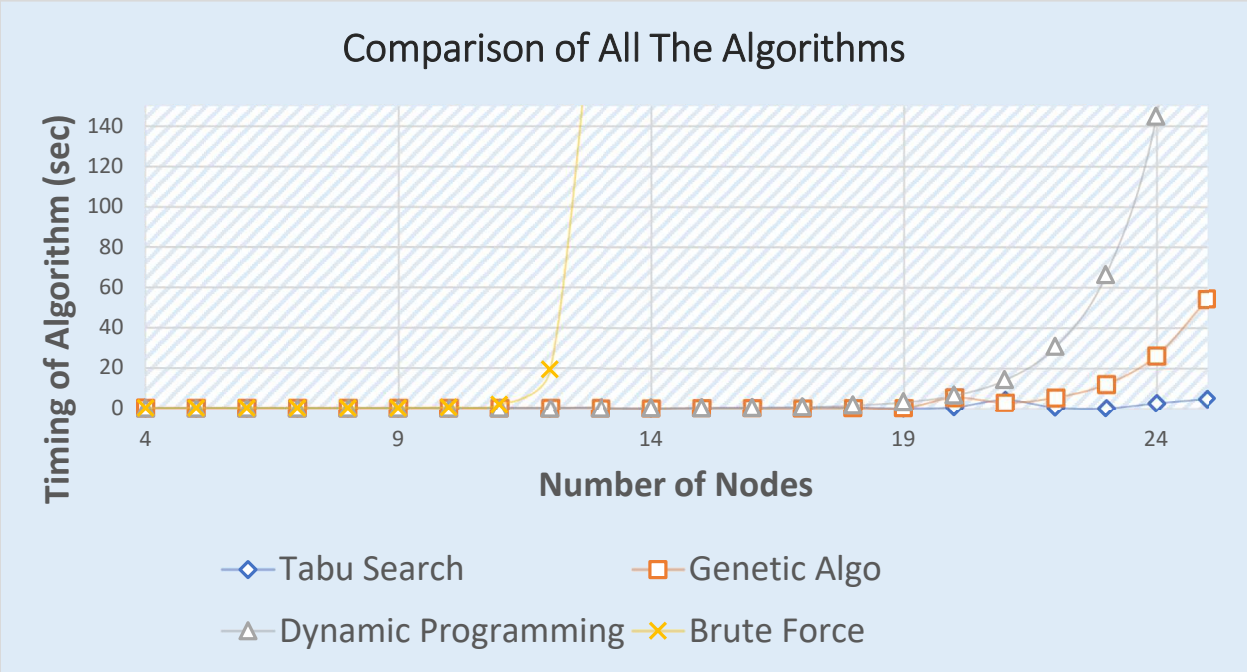
Jayson Werth

Lab 4 – Metaheuristics Searches and TSP

Comparison Graph – Lab3 and Lab4



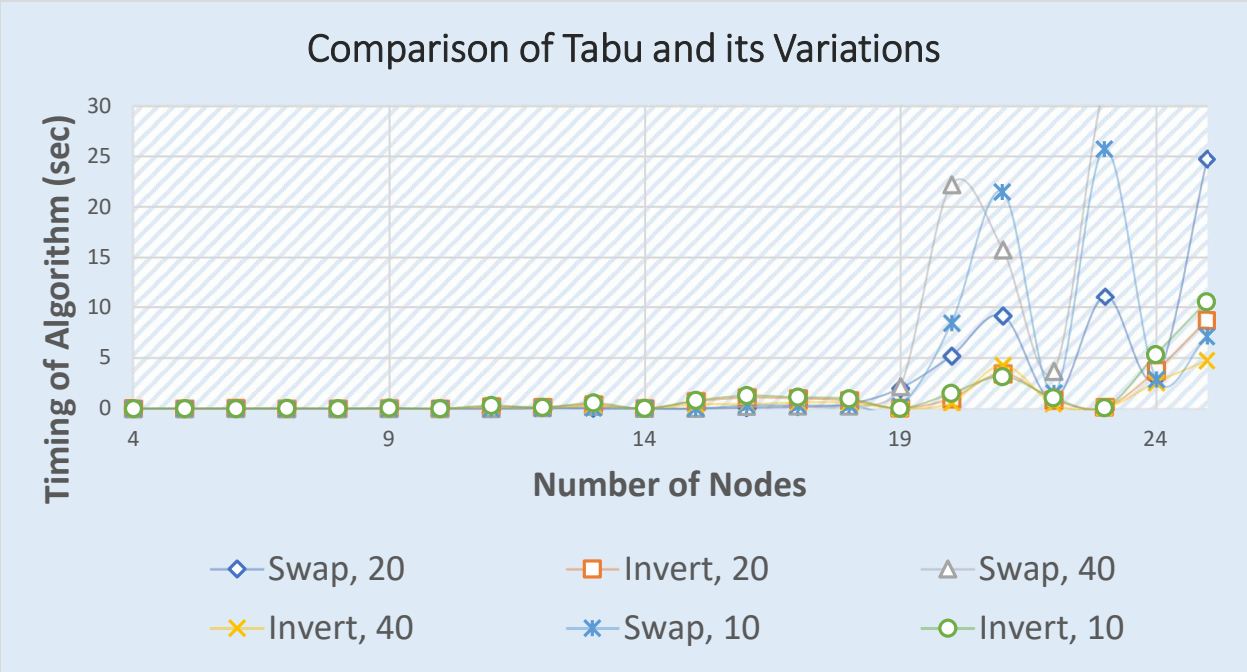
Nodes	Best Tabu Timing	Best Genetic Timing	Dynamic Programming Timing	Brute Force Timing
4	4.776E-05	0.00217208	0.000051579	0.000012620
5	4.682E-05	0.00218837	0.000094388	0.00002716
6	2.437E-05	0.00303936	0.000177028	0.00010665
7	0.035569	0.00534327	0.00032695	0.00062602
8	0.0008528	0.00509398	0.000656288	0.00472562
9	0.0159618	0.00180609	0.00122966	0.035931
10	0.0024198	0.0112125	0.00131349	0.211171
11	0.0482258	0.0136614	0.00297772	1.67526
12	0.036607	0.0108688	0.00677971	19.1431
13	0.269284	0.0114232	0.0158074	242.074
14	0.0130402	0.0190016	0.0738069	-
15	0.413924	0.103151	0.147539	-
16	0.459012	0.0651686	0.302761	-
17	0.602352	0.0291058	0.639005	-
18	0.592531	0.18418	1.54324	-
19	0.0319303	0.0769073	3.09094	-
20	0.606335	5.46588	6.47036	-

21	4.27438	2.86616	14.0575	-
22	0.459317	5.4421	30.589	-
23	0.0794338	12.024	66.2545	-
24	2.601	26.2171	144.993	-
25	4.74475	54.1256	315.96	-

Looking at this data and graph above, we can see that using these more advanced algorithms, we can get the most optimal path at a faster rate than even the dynamic programming algorithm provided. However, as seen in the rang of nodes from 18 to around 22-23, these new searches also come with the downside of this variation since they are nondeterministic algorithms, their performance is often based on a per-run basis, meaning that for each run of the algorithms, often the results in timing will be much different. For example, at 20 nodes, the genetic algorithm performed better than the Tabu Search, but they quickly converge at 21 nodes and then genetic starts to take off away from the timing of Tabu. Also note that:

- Both Genetic and Tabu timing data were taken from the best variation of their configurations so on average, if we were to average throughout their configurations, these algorithms would most likely be worse than that of our dynamic programming from last lab, and
- As stated before, since these algorithms often greatly vary on runtime from run to run, all the data for the Genetic and Tabu algorithms were taken on a set random seed to standardize as much as possible. This also might play into the fact that they performed better for this given seed.

Tabu Search



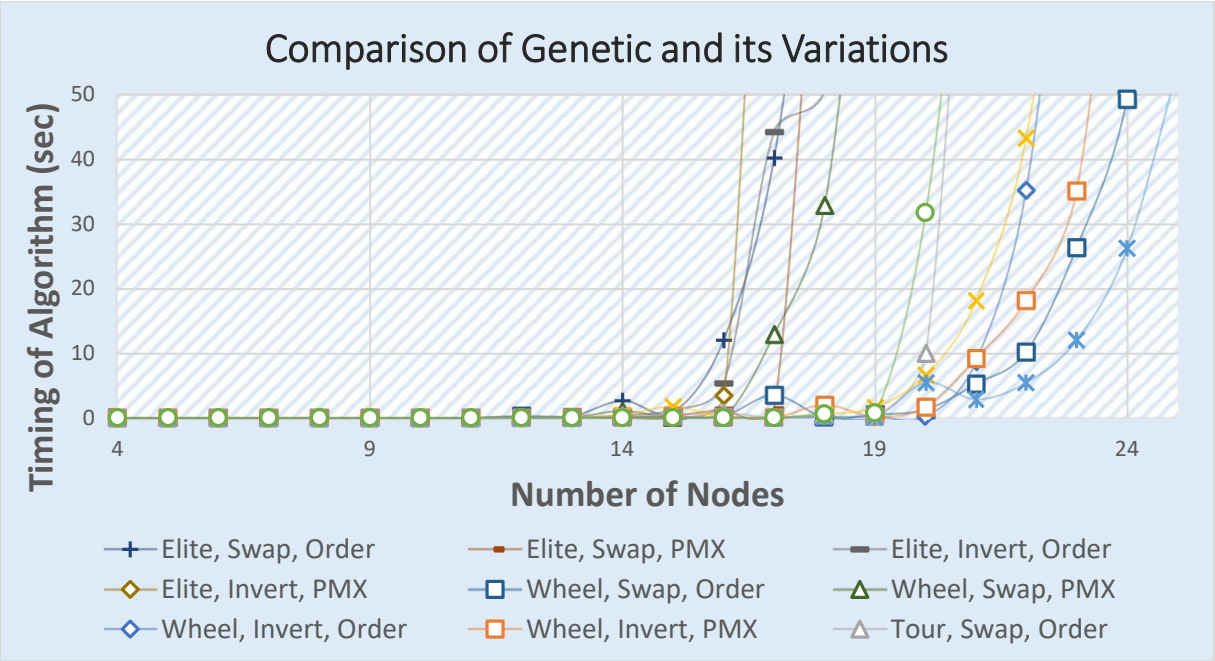
Nodes	Swap, 20	Invert, 20	Swap, 40	Invert, 40	Swap, 10	Invert, 10
4	6.52E-05	5.23E-05	8.54E-05	4.78E-05	5.67E-05	5.34E-05
5	3.26E-05	4.62E-05	8.81E-05	4.68E-05	7.16E-05	4.13E-05
6	0.000759	0.038869	4.47E-05	2.44E-05	1.46E-05	1.43E-05
7	0.000433	0.000619	0.000762	0.035569	0.000541	0.029187
8	0.000606	0.000916	0.001036	0.000853	0.001106	0.000857
9	0.00114	0.032794	0.001738	0.015962	0.001535	0.04601
10	0.001918	0.002666	0.003685	0.00242	0.003629	0.002288
11	0.00268	0.199287	0.002086	0.048226	0.002133	0.30687
12	0.077368	0.16069	0.097517	0.036607	0.012745	0.116522
13	0.029753	0.389958	0.200901	0.269284	0.044263	0.559496
14	0.007129	0.015611	0.012273	0.01304	0.010589	0.011474
15	0.010405	0.710153	0.008603	0.413924	0.010359	0.803949
16	0.085143	1.08982	0.196186	0.459012	0.331321	1.29334
17	0.179762	1.00432	0.249939	0.602352	0.330752	1.11687
18	0.453577	0.807318	0.245244	0.592531	0.3267	0.978664
19	2.01845	0.065562	2.21915	0.03193	0.259793	0.031577
20	5.19627	1.01071	22.2134	0.606335	8.44459	1.51688
21	9.14532	3.46727	15.7612	4.27438	21.4958	3.17501
22	0.880947	0.877596	3.70864	0.459317	1.57332	1.05928
23	11.0816	0.134465	32.1867	0.079434	25.7551	0.080125
24	3.08105	3.75669	38.8376	2.601	2.91868	5.36838
25	24.7373	8.73321	44.7528	4.74475	7.12873	10.6012

Tabu Search, in general, performed at a much better degree than Genetic and Dynamic Programming. While for some of the variations for Genetic the timing got ridiculously large quickly, the worst version of Tabu maxed at a mild 45 secs. However, as the algorithm does consist of some more complex calculations and operations than the ones from lab 3, the timing is a bit slower at first. The timing of this data appears to be on average (even throughout some of the worse configurations) better than that of dynamic programming, meaning it ticks in somewhere slightly better than O(n^2 * 2^n) runtime.

In regards to variation between the configurations of the algorithm, it appears that with increasing tabu list size, the more effective the inversion method of determining neighborhoods becomes. This may be due to the fact that swapping two values is often less effective in creating new solutions that haven't been explored than picking a random section of the path and inverting it. Due to this, the tabu list doesn't help much to limit repetition of best neighbors so it may be falling into more local minima than the inversion purely skips over.

Overall, the pair of configurations with the tabu size of 10 performed the best until the graph started increasing around 24 and 25. This may be due to the fact that with smaller data sets this smaller limit on solutions allows for more exploration that is more beneficial when less local minima exist so as the size increases, the timing starts to take off.

Genetic Algorithm



Nodes	Elitism, Swap, Ordered	Elitism, Swap, PMX	Elitism, Invert, Ordered	Elitism, Invert, PMX	Roulette, Swap, Ordered	Roulette, Swap, PMX
4	0.002834	0.00219	0.002576	0.002019	0.005009	0.004547
5	0.00279	0.002169	0.002542	0.002179	0.004982	0.004438
6	0.005003	0.013994	0.007703	0.001944	0.008326	0.024048
7	0.007229	0.003947	0.006516	0.003666	0.004748	0.007685
8	0.024155	0.011963	0.00604	0.004456	0.011348	0.006396
9	0.014268	0.012607	0.01665	0.005283	0.009599	0.008106
10	0.013009	0.005968	0.009472	0.012837	0.012625	0.006912
11	0.040166	0.033348	0.0262	0.024906	0.01476	0.017448
12	0.167349	0.025058	0.049964	0.040306	0.29705	0.101873
13	0.244589	0.082287	0.113494	0.073106	0.138009	0.112637
14	2.7075	0.06638	0.376253	0.074145	0.164117	1.07425
15	0.948629	0.341035	1.31103	0.096571	0.089035	0.051566
16	12.072	1.30498	5.3607	3.50126	0.318544	0.123589
17	40.2178	1.34075	44.25	112.347	3.54138	12.9483
18	-	101.917	50.7868	57.1551	0.097995	32.8953
19	-	-	-	-	0.519282	-
20	-	-	-	-	1.32564	-
21	-	-	-	-	5.2789	-
22	-	-	-	-	10.2367	-
23	-	-	-	-	26.389	-
24	-	-	-	-	49.2801	-
25	-	-	-	-	-	-

Nodes	Roulette, Invert, Ordered	Roulette, Invert, PMX	Tournament, Swap, Ordered	Tournament, Swap, PMX	Tournament, Invert, Ordered	Tournament, Invert, PMX
4	0.004637	0.004317	0.0022044	0.00187236	0.00217208	0.00188382
5	0.004741	0.004137	0.00230599	0.00199229	0.00218837	0.00184576
6	0.010814	0.009416	0.00182878	0.0109367	0.00303936	0.0014104
7	0.004438	0.010794	0.00227006	0.00310187	0.00534327	0.00304661
8	0.003888	0.006104	0.00207645	0.00171426	0.00509398	0.00612713
9	0.011597	0.005262	0.00324267	0.00363284	0.00180609	0.00362143
10	0.011424	0.006345	0.00705943	0.00543729	0.0112125	0.00502345
11	0.025233	0.012055	0.0107459	0.00820578	0.0136614	0.0121646
12	0.017343	0.038622	0.0154479	0.0841215	0.0108688	0.00606055
13	0.039534	0.066787	0.0242896	0.0641432	0.0114232	0.00950826
14	0.025123	0.140737	0.382606	0.49242	0.0190016	0.0467383
15	0.072073	0.200518	0.476534	1.77166	0.103151	0.0213596
16	0.055173	0.189944	0.780564	0.136599	0.0651686	0.038456
17	0.061589	0.067414	0.0930278	0.0237737	0.0291058	0.0251723
18	0.119805	2.01872	0.401316	0.58321	0.18418	0.63079
19	0.208689	0.204561	0.272793	1.66754	0.0769073	0.775405
20	0.217903	1.72235	10.0216028	6.66834	5.46588	31.7446
21	8.7395	9.22172	110.954	18.1269	2.86616	-
22	35.2425	18.1832	-	43.2789	5.4421	-
23	-	35.1256	-	-	12.024	-
24	-	-	-	-	26.2171	-
25	-	-	-	-	54.1256	-

These results were much more shocking that that of the Tabu algorithm, as I initially thought that this algorithm would perform much better than that of Tabu. However, I believe that my methods of crossover were not effective enough in producing elite offspring to continue improving the population to the optimal solution quickly. I think the mutations themselves were more decent as it helped get the populations get out of these local minima/uniform populations and slowly bring the solution down to the optimal path. The selection processes were obvious better when randomness was added as a factor in addition to the fitness scores.

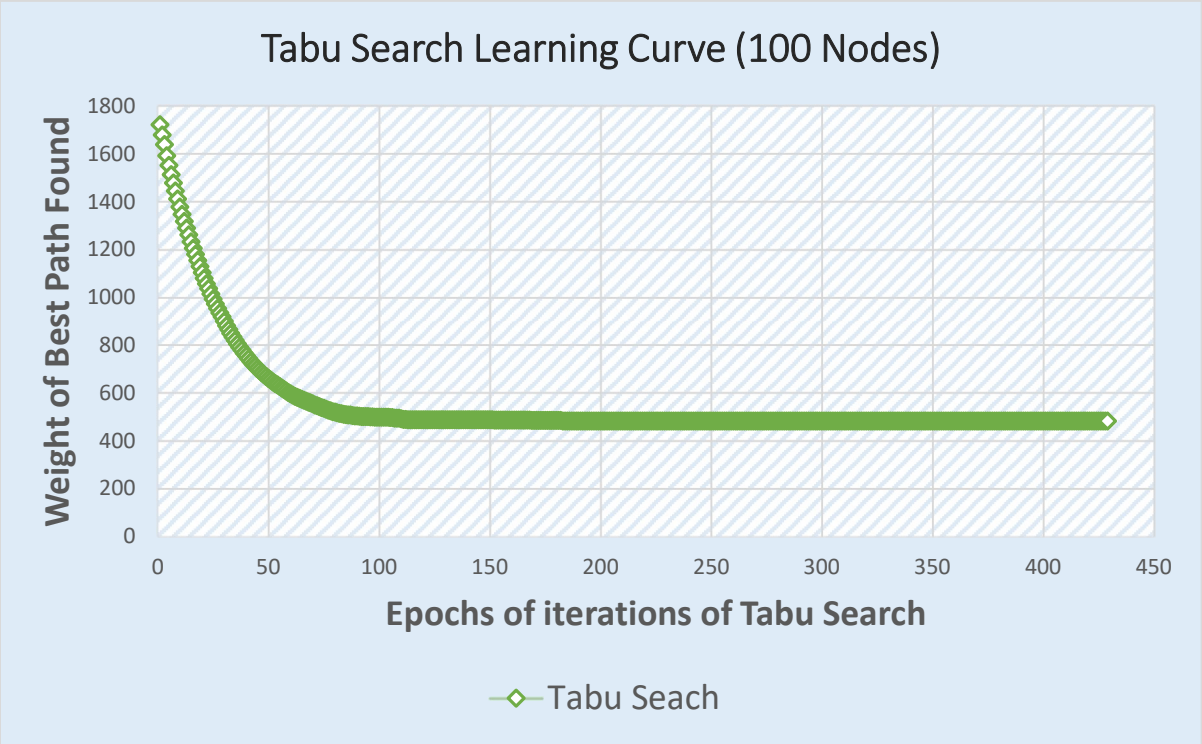
Once more, this algorithm was outperformed by the naïve and dynamic programming methods for the early iterations as the algorithm performs a handful of sorts, creation of new children that slow down even the small paths since a full generation takes some time. However, (at least for the better performing configurations) the algorithm either matches or slightly beats the timing of the Dynamic Programming and much outperforms the brute force. Even though this is a more expensive algorithm than those of lab 3, it still holds a sense of "learning" rather than just clever tricks that are given by dynamic programming so with the breeding and selection of elites, the solution gradually gets better and better due to the fact that there are 100 different paths that are improved and mutated.

The Selection method of Elitism appears to be the worst performing of three selection methods as none of them could pass 18 nodes without running for a very long time. I believe this is because the picking of elite purely on fitness score sounds very good, however if the initial population that was created was the longest path then it is very easy to fall into a "local minima" by virtue of loss of genetic diversity as the population becomes almost all copies of the elite. The addition of the random picking in addition to the use of the fitness score allows for enough variation to at least delay the loss of genetic diversity (see learning curve).

The Mutations don't seem to play a large result on the timing as the selection does (since the selection methods often just make them obsolete) , however the inversion does seem to be performing a little better than the swap. This is probably due to a similar reason to that of the Tabu as the inversion causes a more drastic change that could be sent into the parents of the next generation and thus used to improve the population little by little.

The Breeding/Crossover methods used (Ordered Crossover and Partial Mapped Crossover) don't seem to follow a pattern over all of the configuration, but the two best performing configurations occurred with the ordered crossover. This may be due to that in Ordered Crossover, the ratio of parent 1 and parent 2 genes is often more 50/50 than in that of Partial Mapped since you purely grab a region and then put all the remaining, non-duplicate values in the child.

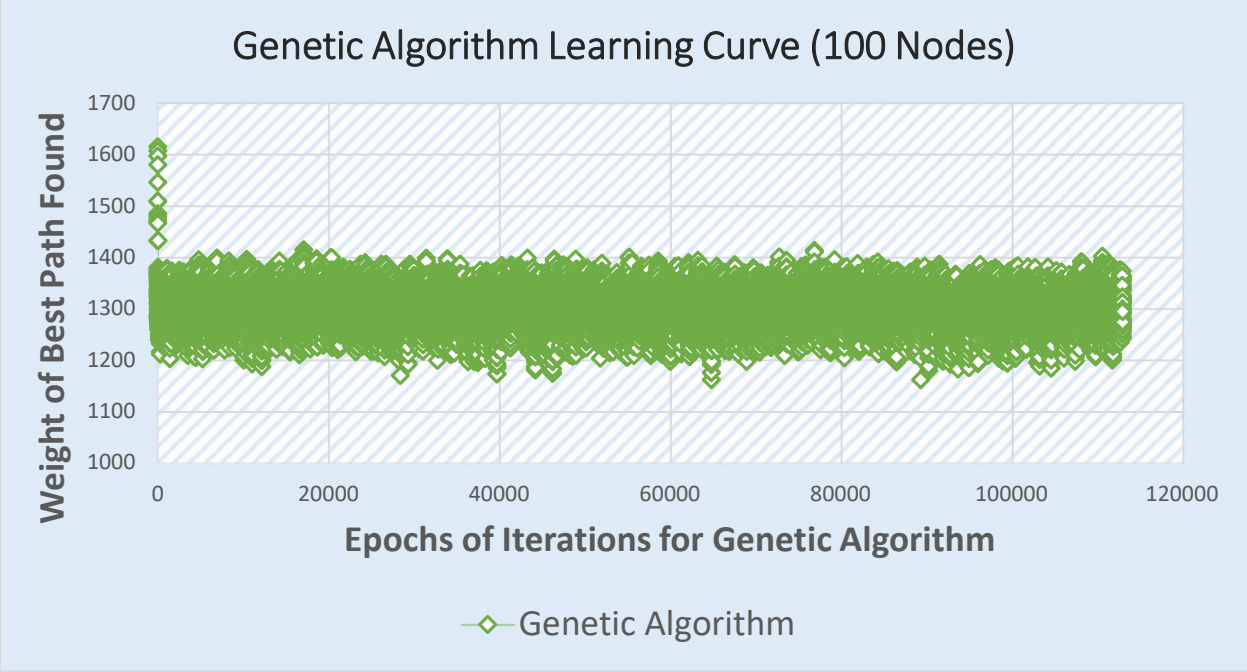
Learning Curve – Tabu



In this graph, we see a very nice slope downwards, showing that the Tabu algorithm has a good searching method that allows it to find good neighbors and also it can escape local minima by using its searching. Oven the period of 10 minutes, the Tabu continued to improve its score (although pretty minimally following 100 epochs) and was learning from its neighbors and making good decisions to keep the better solutions. Although, due to the lack of a mutation-type method that was present in the algorithm, the variation in the solutions was minimal following the 100 epoch mark and also the algorithm only improved, as Tabu should do.

This was generated using the best performing algorithm at 25 nodes for the Tabu search, which happened to be the inversion neighborhood selection and the tabu size of 40. This then helps to provide more proof that the increased tabu size helps at the higher number of nodes as their starts to become more and more duplicate path lengths, leading to an easier transition into local minima. This is then why, although the Tabu performed better and more accurately than the Genetic algorithm, we can't sure on the solution due to the lack of natural variations that occur in the other algorithm.

Learning Curve – Genetic



In this graph, we can see a nice slope downwards for about 200-300 epochs before the graph then gets locked in a local minima and then continues to spike up and down in a range of about 150 distance for the next 9:30. This graph shows the problem that I was facing with my Genetic algorithm, although is wasn't that evident to me when collecting test data, this graph proves that even with the best configuration of the method (Tournament Selection Method, Inversion Mutation Method, and Ordered Crossover Method) the implementation loses genetic diversity very quickly and even the mutation that occurred couldn't get it out of a local minima.

Some factors that may have led to this messy graph may have been by population size that may have become too small as I increases the path size drastically, the selection of too many

"elite" individuals that are used for breeding the next generations, the lack of enough mutation in a single population to provide a way out of the local minima, or maybe the crossover methods that I utilized don't crossover in a way that changes the genes while also maintaining good fitness. And since this is same graph set that the tabu tested, the genetic diversity goes away way to quick and due to the presence of a fairly poop starting population, the genetic algorithm estimated the path length to be around 1300 whereas the Tabu got down to around 400.

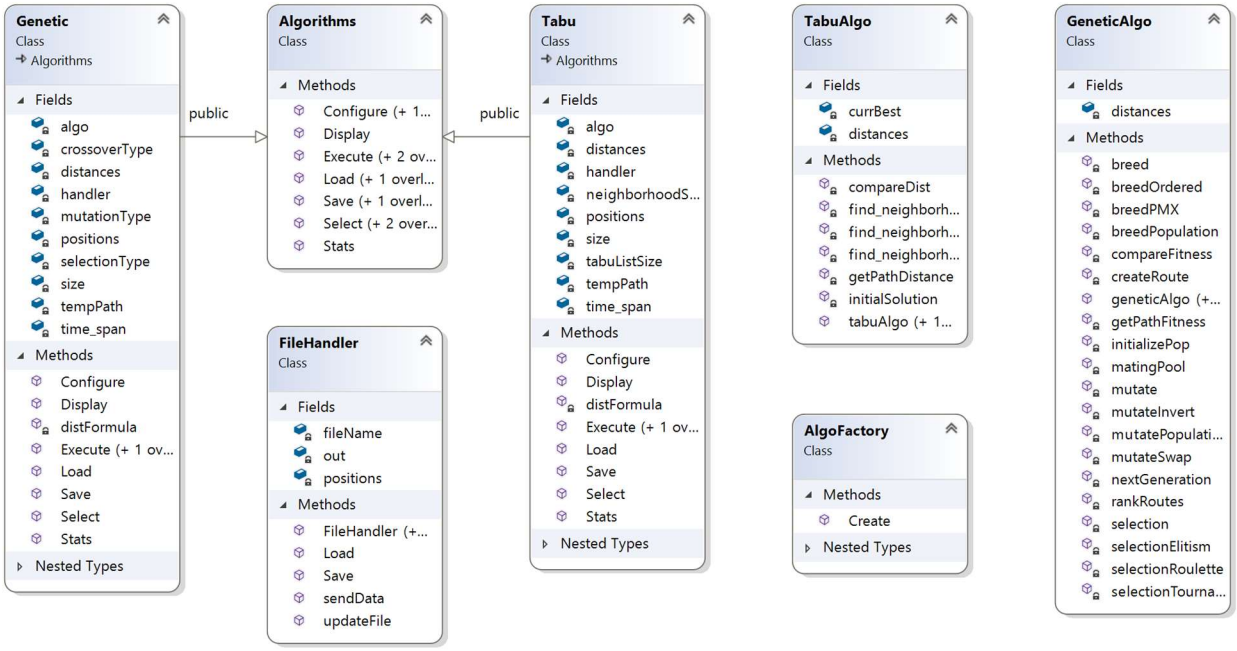
Design Implementation:

For this lab I used a mix of the Strategy Design pattern with the use of a similar Algorithms class to the first two labs and a Factory Builder Pattern to facilitate the creation of the Tabu objects and Genetic objects. However, compared to the similar pattern that was used in lab 3, I used a Strategy pattern much more close to that of lab 2. A UML Class Diagram is located below this section and in the Data folder as Metaheuristics.pdf.

- Strategy Aspect – Algorithms Abstract Base Class
    - Similar to the first two labs, I used a abstract base class similar to the provided implementation from the first lab in order to separate the user interface from the actual implementation of the TSP Metaheuristic Search algorithms. In this way, I could use very similar implementations for the Tabu Search and the Genetic Algorithm versions of the TSP and only have to use the single interface of the Algorithms pointer to interact with both.
    - For extensibility of the classes/patterns, the abstract base class allows me to be able to add new methods that would be common to algorithms and then provide distinct implementations of that method in any base class, all that can be accessed by a single call by the Algorithms pointer. For example, I temporarily added an additional version of the Save method that took in an ofstream so I could print the data from the algorithms to a csv file for efficient data collection. Also I added multiple execute functions, at least for Genetic, so that I could add a stop value so the algorithm terminated when reaching a certain cost threshold.
    - Also, the Algorithms class still contains the Configure function that could be used to change specific things about the algorithms themselves. For example, unlike in the previous lab, for the heuristic and metaheuristic searches, I utilized this function to change different variations of the algorithm. For example, For Tabu you can use this function to change the neighborhood search method and the Tabu list size and for Genetic, you can switch between three types of selection, two types of mutation, and two types of crossover
    - Compared to lab 3, this structure is more heavily on the Strategy pattern as I used a similar pattern to lab 2 where the interface and implementation of the algorithm itself is separate yet connected by a function pointer. This allows for even more isolation of implementation from the user and allows for easy addition/alteration of code for each algorithm.
- Factory Aspect – AlgoFactory Class
    - Since we had to implement two different types of TSP Searches once more, I decided to create a factory class that would handle the creation of these specific instances of Search objects rather than having the user (in main.cpp) directly

create a new Tabu or Genetic Search object. In this way, I could simplify the interface in main even further, so now the user just has to call the Create() method with one of the enum values (which are labelled to help identify which algorithm you will create) rather than having to call new on all the different Search Algorithms in main. This also helps remove more dependencies in the main function as the are only included in the actual factory.
    - Also with is class, I tied the Create() method functionality to be in accordance with the Algorithms Strategy Class as it returns an Algorithms pointer with the address of a new algorithm (similar to, in the previous lab, where you'd handle this at the top of main). In this way, the Factory is extensible to all classes that inherit from Algorithms, such as Sort from lab 1, the general Search from lab 2, the dynamic/brute force algorithms from lab 3, and now the heuristic and metaheuristic searches of this lab. With this implementation, we can now simply add new entries to the enum and adding a few more case statements in the Create() method to include all the different Sort/Search/etc. all using this one interface.
- In addition to the Strategy and Factory Pattern classes, I also used the functionality of a FileHandler class, which served as the interface for loading files and saving data from algorithms to an output file.
    - For this lab, both of the TSP algorithms also contained an instance of this class, for which they used to load their positions file and to print the name, path, cost, timing, etc. that was collected during the actual execution of the algorithms. This allowed for both of those files to be much shorter than the implementations of the Sort class and Search class.
    - By using this interface, another level of separation is created from the actual user's interface. However, it also allows the loading and saving of files to simpler for the algorithms. For this lab, the handler is set up to handle the loading of data from a positions file and saving data from TSP algorithms, but by simply adding another load function for a different set of files can extend its functionality.
    - By taking the reading and writing from the actual algorithms, the classes become simpler and easier to read/update since there won't be 90 lines of reading to sift through in each algorithm, and updating the methods in the handler will distribute changes to all the algorithms that use that loader rather than having to go and change each individual one.

UML Class Diagram



Notes

When passing a nodes file, start with 1, the output will print 0-n-1. Also make sure there is not extra line at the end of the file.

Alternatively to numbers for the configures, you can use the enums in each according class

For both algorithms, you can enter a value into the Execute function to add a stop condition so the algorithm doesn't run for x generations or x repeats.