**Introduction**

Lab 5 was an extension of the previous two labs. For this lab, I implemented two additional algorithms for the travelling salesman problem. The Simulated Annealing (SA) and Particle Swarm Optimization (PSO). Simulated Annealing just like genetic algorithm, is a meta-heuristic algorithm. Simulated annealing algorithm is an effective and a more general form of optimization. It is an algorithm useful in finding a good (not necessarily the optimal) solution to a problem. In the presence of large numbers, the simulated annealing algorithm attempts to find the global optima. “Annealing” refers to an analogy with thermodynamics, specifically with the way that metals cool and anneal.  Hence, Simulated annealing uses the objective function of an optimization problem instead of the energy of a material. I also implemented the particle swarm optimization technique. This technique is similar to the genetic algorithm in the sense that it is a population based stochastic optimization technique. However, unlike genetic algorithm, the particle swarm optimization algorithm was inspired by the social behavior of bird flocking or fish schooling. PSO shares a good amount of similarities with the evolutionary computation technique, genetic algorithm. They both initialize with a population of random solutions and searches for optima by updating generations. However, unlike genetic algorithm, PSO has no evolution operator, such as crossover and mutation. In PSO, the solutions, also known as particles fly through the problem space by following the current optimum particles (solution).

**Algorithm Techniques**

For my Simulated Annealing algorithm, I start from a size of 1000, similar to that used in the previous lab (genetic algorithm and tabu search). the algorithm starts with the random probability function which returns a double figure for the random probability generated. This is passed into a function that swaps the solution generated computing the best solution as it goes through the algorithm. Since the algorithm is basically hill-climbing except instead of selecting the best move, it selects a random move. If the selected move improves the solution, then it is accepted. Otherwise, the algorithm makes the move anyway *with some probability* less than 1 (in the case of my project, I use 1.0). The probability decreases exponentially with the “worseness” of the move, which is the amount deltaE by which the solution is worsened (i.e., energy is increased.) for my project I used the formula with the parameter T used to determine this probability.

Prob(accepting uphill move) ~ 1 - exp(deltaE / kT)

**deltaE = eval(tem) - eval(num);  
deltaE /= T;  
prob = 1 / (1 + exp(-1\*deltaE));  
expP = randProb();**

The T in the equation is analogous to temperature in an annealing system.  At higher values of T, uphill moves are more likely to occur.  As T approaches zero, they become more and more unlikely, until the algorithm behaves more or less like hill-climbing.  In a typical Simulated Annealing optimization, T starts high (in my project, I start T at 100000) and is gradually decreased according to an “annealing schedule”.  The parameter k is some constant that relates temperature to energy (in nature it is Boltzmann’s constant.)

**Pseudo-code: Simulated Annealing (SA)**

1. First, generate a random solution
2. Calculate its cost using some cost function you've defined
3. Generate a random neighboring solution
4. Calculate the new solution's cost
5. Compare them:
   * If cnew < cold: move to the new solution
6. Repeat steps 3-5 above until an acceptable solution is found or you reach some maximum number of iterations.

For my particle swarm optimization technique, I initialize the population starting with 1000, for uniformity with the other algorithms and for comparison purposes. The maximum velocity set for my algorithm is 50, but this is modifiable and can be changed in my implementation file. For the algorithm, PSO is initialized with a group of random particles (solutions) and then searches for optima by updating generations. In every iteration, each particle is updated by following two "best" values. The first one is the best solution (fitness) it has achieved so far. (The fitness value is also stored.) This value is called p\_best. Another "best" value that is tracked by the particle swarm optimizer is the best value, obtained so far by any particle in the population. This best value is a global best and called g\_best. After finding the two best values, the particle updates its velocity and positions with following equations

**Velocity = v[] = v[] + c1 \* rand() \* (pbest[] - present[]) + c2 \* rand() \* (gbest[] - present[])   
Position= present[] = persent[] + v[]**

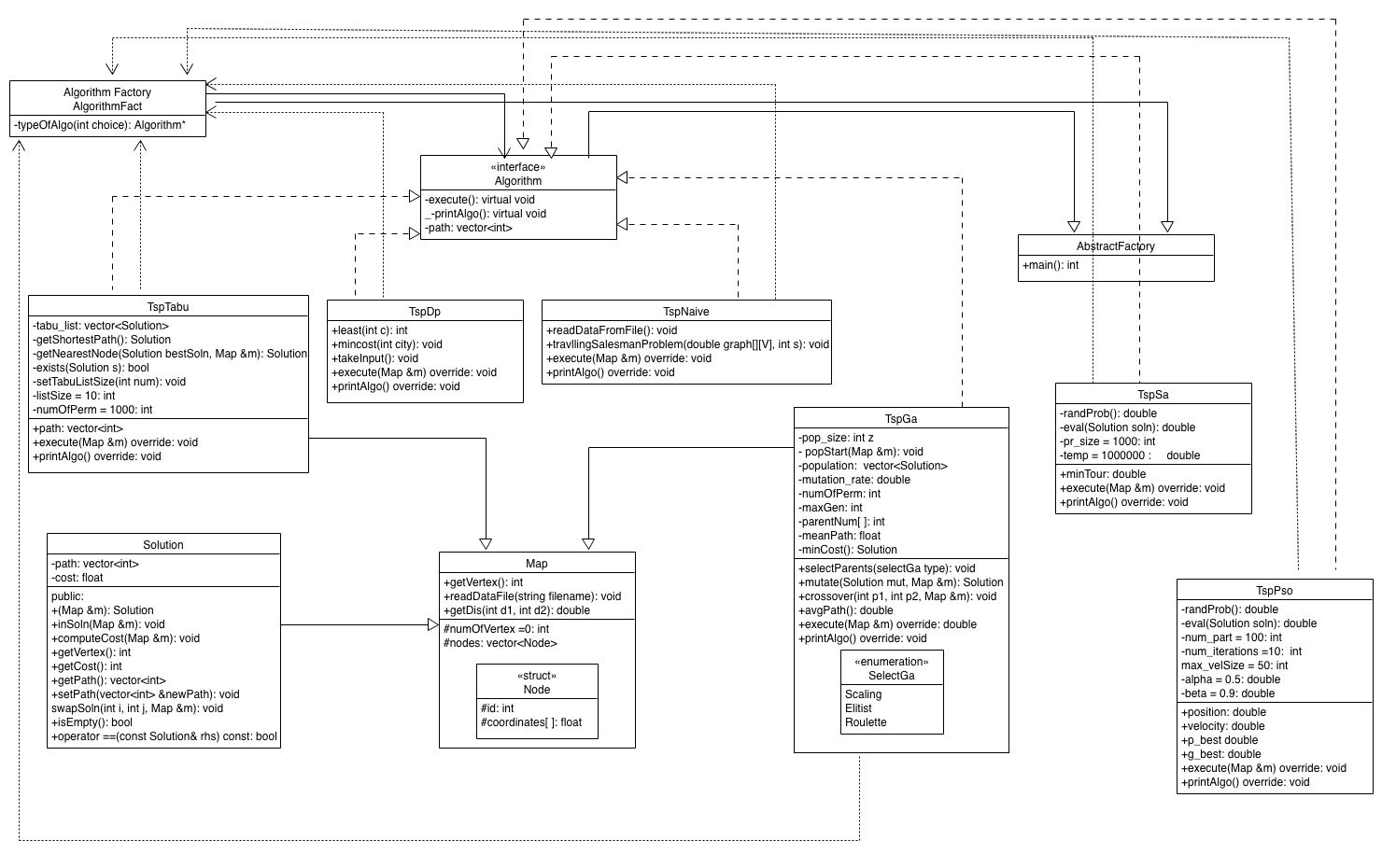
v[] is the particle velocity, persent[] is the current particle (solution). P\_best[] and g\_best[] are defined as stated above. rand () is a random number between (0,1). c1, c2 are learning factors.

**Pseudo-code: Particle Swarm Optimization (PSO)**

For each particle   
    Initialize particle  
END  
  
Do  
    For each particle   
        Calculate fitness value  
        If the fitness value is better than the best fitness value (pBest) in history  
            set current value as the new pBest  
    End  
  
    Choose the particle with the best fitness value of all the particles as the gBest  
    For each particle   
        Calculate particle velocity according equation (a)  
        Update particle position according equation (b)  
    End   
While maximum iterations or minimum error criteria is not attained

**Architecture Design/Modifications**

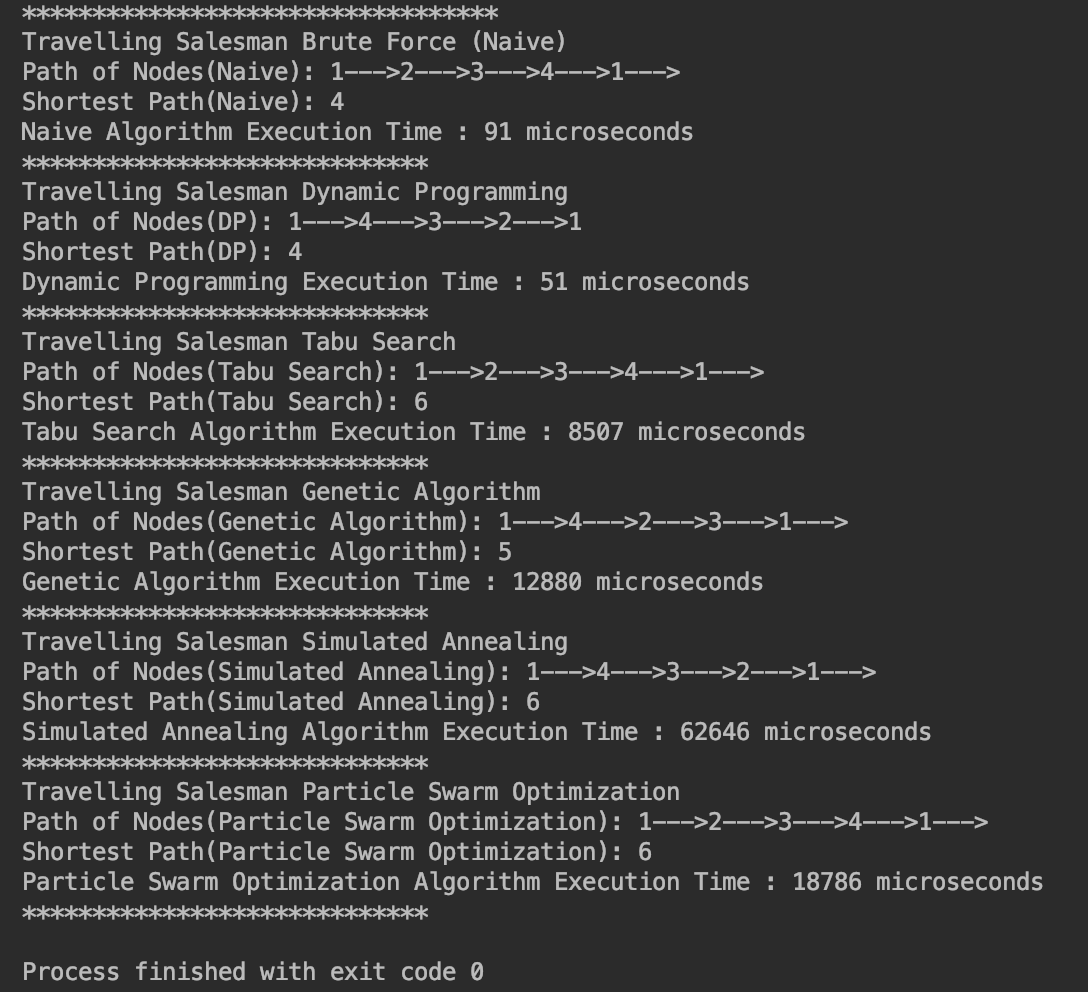
For lab 5, I did not make any significant modifications to the overall architectural design. I did some make some minor modifications necessary for the implementation of the project in terms of extending the solution to be compatible with the previous labs. I created two new classes for the new algorithms (shown in the UML below) for extensibility, the solution class still extends Map in order to make use of the data file as well as some of the functions within the solution class required for Simulated Annealing algorithm. The project still makes use of the components of the factory design pattern with the addition of the afore mentioned modifications. For the Particle Swarm Optimization technique, I made use of several resources to find a suitable implementation, while it might not be optimal, it produces the path and a distance. I believe the solution is not optimal due to the fact that I may have mixed up the p\_best and g\_best values while making use of the probability factor. I found it somewhat difficult to integrate some of the references I found to my design pattern.



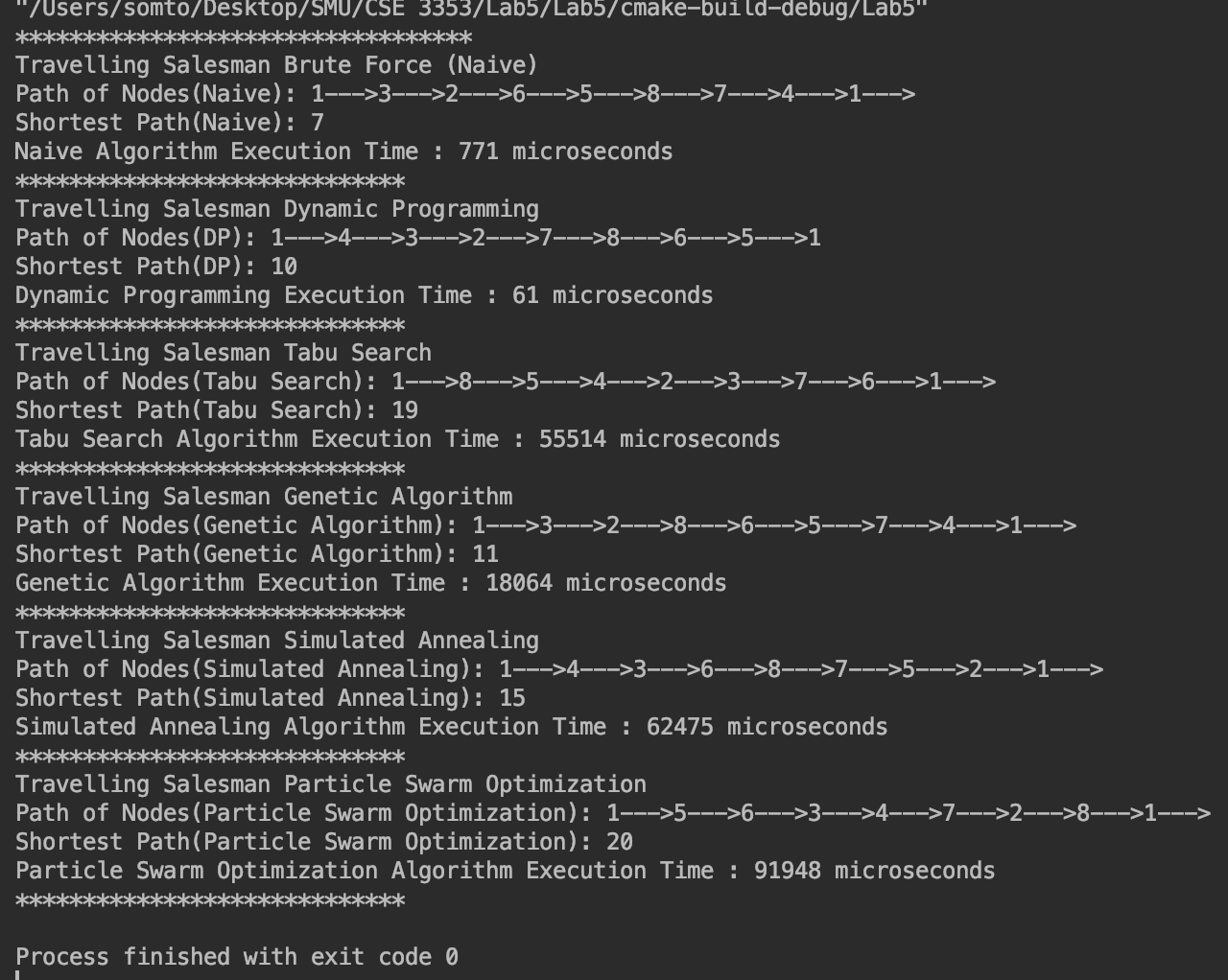
**Figure 1:** UML diagram for design architecture. (A raw expanded version can be found on the repository.)

**Analysis and Results**

For the analysis, I ran all algorithms implemented starting from Lab3. That includes, the Naïve, dynamic programming, tabu search, genetic algorithm, simulated annealing and particle swarm optimization. Just like the previous labs, I tested the algorithms on maximum node of 13. For the Simulated annealing algorithm, I noticed increasing the temperature increased the execution time of the algorithm, although producing similar results for the path of nodes and shortest distance. I also noticed that the path of nodes and shortest distance for 4 nodes for the simulated annealing and genetic algorithm was comparable. I believe this is due to the cooling effect of the annealing process. For the particle swarm optimization, as mentioned earlier, I had a difficult time integrating the code, however, I noticed increasing the maximum velocity and/or the number of particles caused an increase in execution time and I believe this is due to the fact that the algorithm makes use of the evolution process and creates new generation, increasing the number of particles, will increase the number of particles that can be found and therefore increase the execution time. Increasing the velocity increases the distance between the neighbors and so causes an increase in the execution time as well.



**Figure 2:** Output of all algorithms with 4 nodes



**Figure 3:** Output of all algorithms with 8 nodes

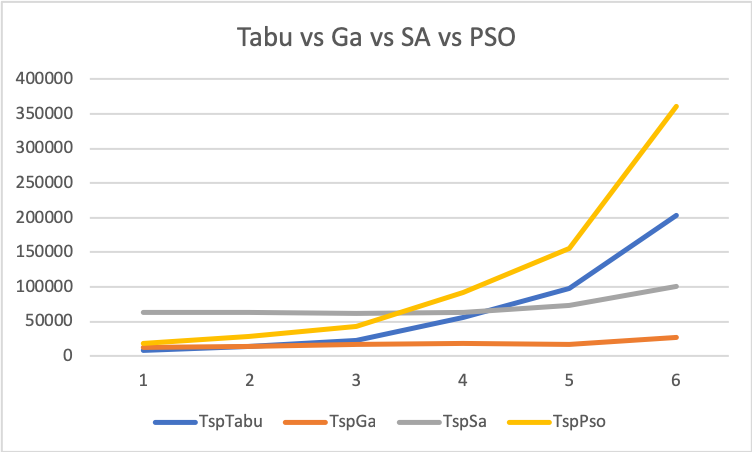
Just like the previous labs, With a quick look at the screenshot above, you notice the path of nodes are not printed in the same order, I had this same problem in the previous lab and I was unable to address it in this lab, but I believe the results are somewhat accurate as the shortest path obtained is for the given path of nodes displayed and so the paths are different for each algorithm.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **# Nodes** | **Tsp Naïve** | **Tsp DP** | **TspTabu** | **TspGa** | **TspSa** | **TspPso** |
| 4 | 91 | 51 | 8507 | 12880 | 62646 | 18786 |
| 5 | 83 | 51 | 14119 | 14131 | 62825 | 28243 |
| 6 | 100 | 58 | 23369 | 16783 | 62084 | 42617 |
| 8 | 771 | 61 | 55514 | 18064 | 62475 | 91949 |
| 10 | 58029 | 143 | 98178 | 16193 | 73200 | 155596 |
| 13 | 750935 | 717 | 203500 | 27019 | 100585 | 361065 |

**Table 1:** Nodes for Algorithms and time in Microseconds

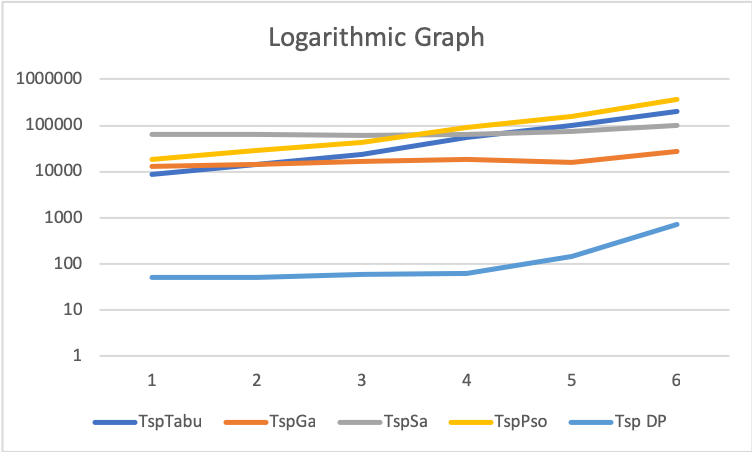
**Algorithm Plot Comparison**

Similar to previous labs, for time complexity, I generated a chart with the table above, and due to the fact that I had a limited number of nodes tested, I decided to plot a linear fit curve as well as a logarithmic curve for better viewing and analysis, charts are shown below;



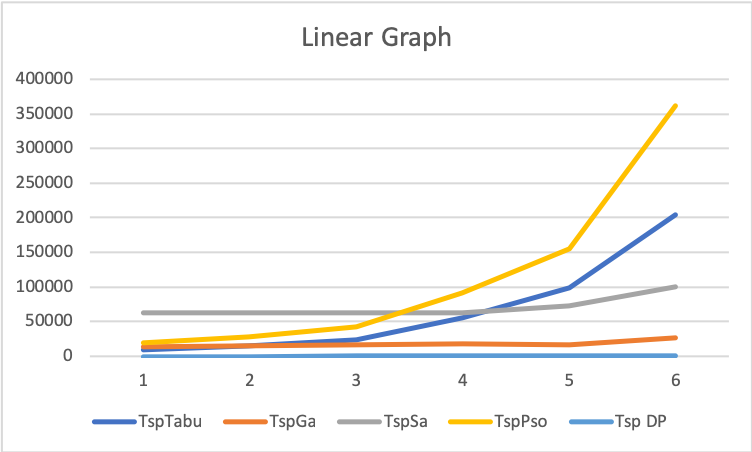
**Chart 1:** Scale Graph of Nodes vs Time of 4 algorithms

I analyzed the two new algorithms to the two algorithms from lab 3, the Particle Swarm Optimization algorithm seems to have had the slowest execution time amongst all 4 algorithms with the genetic algorithm out performing it, and this may be attributed to the way my algorithms were coded.



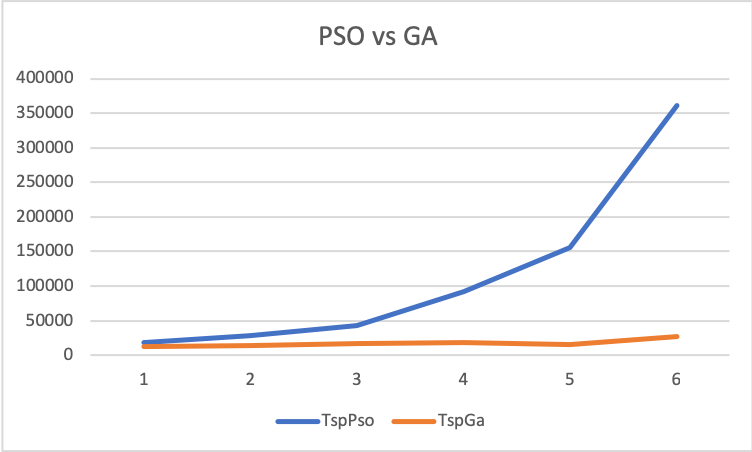
**Chart 2:** Logarithmic Scale Graph of Nodes vs Time of 5 algorithms

I also compared the two new algorithms to the two algorithms from lab 3 and dynamic programming and dynamic programming had the fastest execution time amongst all 5 algorithms.



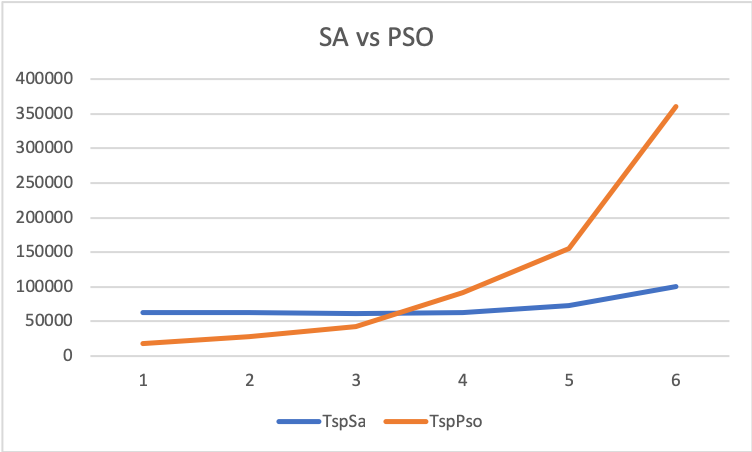
**Chart 3:** Linear Scale Graph of Nodes vs Time of 5 algorithms

This is a similar graph as the one shown above, I used the logarithmic graph for better viewing.



**Chart 4:** Logarithmic Scale Graph of PSO vs GA

Since the Particle Swarm Optimization and Genetic Algorithm shared similarities, I decided to plot them against each other and analyze the results, as predicted, the PSO algorithm was not favored with a large number of nodes.



**Chart 5:** Logarithmic Scale Graph of SA vs PSO

My last analysis was for the two algorithms specific to this lab, the Simulated Annealing and Particle Swarm Optimization technique, we see that the simulated annealing had the fastest execution time with increasing number of nodes.

**References**

<https://github.com/arjun-krishna/TSP>

<https://github.com/QuentinJcb/TSP>

<https://www.codeproject.com/Articles/1182794/%2FArticles%2F1182794%2FSolving-the-Travelling-Salesman-Problem-With-a-Par>

<http://mnemstudio.org/particle-swarm-tsp-example-1.htm>

<file:///var/folders/2j/6t5_yv2d57n53rfb6nt61yz00000gp/T/com.apple.iChat/Messages/Transfers/IJCSI-9-6-2-264-271.pdf>

<https://github.com/icdts/PSO_TSP/tree/master/src>

<https://github.com/marcoscastro/tsp_pso>

<http://www.swarmintelligence.org/tutorials.php>

<http://www.cs.armstrong.edu/saad/csci8100/pso_slides.pdf>