Hayden Donofrio

CSE 3353

LAB 05 Report

**Overall Comparison:**

In this report I will be comparing the differences between the simulated annealing algorithm and the particle swarm optimization, how they are different, and how they compare to other algorithms (dynamic programming, tabu, and genetic algorithm).

Overall, we can see that our new algorithms: PSO and SA, both performed relatively well in a timing perspective to reach a “good” solution. The one thing I noticed that was very different about PSO compared to the other algorithms was that with smaller graphs PSO took a significantly longer time to reach the optimal solution. However, PSO did not take significantly more time to reach a good solution as the number of nodes increased while the other algorithms had a much steeper curve in respect to nodes/time.

If we were to continue this trend with larger graphs, I believe we would see PSO outperform all of the other algorithms that I have listed in this chart with the amount of time it takes to converge to a good solution. I am very surprised that PSO performed so well compared to genetic algorithm considering both take a significant amount of operations per cycle while a hill climbing approach like SA or Tabu simply moves around the search space. I expected PSO to perform similar to GA, but the results show that PSO nearly outperformed GA by a factor of 5 at N=20. I think this is because of the learning factors that PSO uses, I.e. the local and global learning. GA has a strong element of randomness in it where the only part where learning or improvement happens is in parent selection, and the process of evolution is slow and tedious to ensure that we reach a good solution without converging. PSO has particles that move towards the best solutions which usually leads to a particle discovering a “hot-spot” where many good solutions are grouped together. This causes particles to ideally search good areas of the search space at a higher rate than GA.

All algorithms performed significantly faster than the traditional dynamic programming approach, especially Tabu and PSO. I believe as the search space continues to increase in size, we will see that PSO finds good solutions relatively quickly because we found that as N increased, the time increased at the lowest rate. As we will see in the next section, PSO found fairly good solutions too compared to SA.

**PSO Vs SA: ALL STATS USE 20 A NODE GRAPH**

Both of the algorithms found good solutions in an appropriate amount of time. However, because of the learning approach of PSO, it reached better solutions much faster while SA would simply find a good solution via hill climbing, and then jump to another place in the search space with respect to high energy.

Consistently we see that PSO performed much better than SA in both time and fitness over a 20-node graph with the optimal solution being a distance of 20. PSO was very close to the optimal solution by the execution cut off while SA was not as close. I noticed that SA found solutions in a more functional way (easily represented with a function) while PSO found a hot-spot of solutions, and then it would act dormant for a while. I believe this is because PSO would struggle finding a better solution until one of the population stumbles upon a new best global solution which causes the entire population to begin to search close to this new area of the graph where good solutions are more common. SA found its best solutions at a solid logarithmic rate. Perhaps this is because SA spends less time jumping around as temperature gets closer to 0 so we explore new hills not as often.

**SA**

I have three different versions for starting temperature for SA:

1. 5000 starting temperature
2. 500 starting temperature
3. 100 starting temperature

Before the tests I expected the 100 starting temperature to perform the worst because it would not have a long enough time to search the graph for good “hills” to climb while the other two temperatures had a better chance at searching the search space a bit before the temperature cools down enough to where the algorithm stabilizes. My prediction became true because 100 performed the worst out of the three as it did not have enough time to search the graph before the temperature began to cool. An interesting observation was that 5000 and 500 had similar results, and 500 even performed better than 5000 did. It looks like both of them failed to find the optimal solution. However, the 500 temperature found a better solution much faster than the 5000 did. I believe this is because the 500 starting temperature is enough to jump around the graph successfully, but also search hills effectively, and by the time that the temperature has cooled, we have arrived at a good “hill” to climb. 5000 Simply performed too erratically compared to 500 and it would struggle to completely climb a hill and find a good solution before it had to jump to another place in the graph since the energy was always so high. Therefore, I believe that starting temperature of around 500 is the best for SA since it will find a similar solution to larger temperatures much faster because it is allowed enough time to search with high energy before it begins to cool down.

I had another adjustable variable: the way temperature decreased over time.

1. Decrease by 2 every 50 iterations
2. Decrease by 5 every 50 iterations
3. Decrease by 10 every 500 iterations

This is a graph showing the differences between these three different decreasing types with a 500-starting temperature. The most interesting thing that I noticed from these graphs was that 2 and 5 both found good solutions at a similar rate, but then 5 began to slow down significantly while 2 continued to find better solutions. However, 5 never actually converged to a solution while the two converged to a solution fairly quickly. I think that this is because decreasing by 2 and decreasing by 10 both lower the temperature to a point where the graph fails to explore new areas of the search space but decreasing by 5 allows a large amount of time for exploring but still a good enough temperature to allow for some basic hill climbing throughout the algorithm. Therefore, for the conditions that we have, decreasing by 5 will give us the better solution the most often due to how infrequently it converges.

**PSO**

For PSO I had two main adjustable variables: the learning factors and the maximum velocity.

For maximum velocity where N = the number of nodes in the graph:

1. N/4 or 25%
2. N/3 or 33%
3. N/2 or 50%

Velocity was the amount I changed for each individual placement of each node for a solution. For example, let’s imagine we have a path of 1,2,3,4,5,6,7,8,9,1 with start and end 1 that do not change. The velocity vector might look something like 0,2,0,1,-4,3,2,6,-2,0 and that would increase/decrease whatever the current position is by that much, so our new position would then look like 1,4,3,5,1,9,9,4,7,1 this is not a valid solution since there are repeats, so we would repeat this process until we get a valid new position vector.

Above is a graph of the three different max velocity types with a learning factor where global=local. Before I began this experiment, I thought the 50% max velocity would be too much to the point where the members of the population would be skipping over large areas of the search space to reach the new maximums that it found. Thereby, reducing the chances of finding a better solution along the way. My hypothesis became relatively true because the 50% max velocity converged extremely fast compared to the others. This is because without a chance to properly search the graph, it removes the chance of the particle stumbling upon a new local or even global best along the way.

The 33% max was the only one that did not converge throughout the execution. I believe this is because it Is sufficient enough to effectively move towards a new local or global best solution with enough room to increase the velocity so that it can remember more information about its previous velocity. 25% is not high enough because if a new best is found. That will almost completely overwrite the previous velocity in the new direction which makes the particle essentially forget about a lot of the previous information that it had in the past. Therefore, to efficiently search a 20-node graph with proper memory about where to go and a fast-enough speed to reach a reasonable destination, it is preferred to go about 33% of the number of nodes in the graph.

I had three different types of learning factor coefficients.

1. Local: 1, Global: 3
2. Local: 3, Global: 1
3. Local: 2 Global: 2

This means that for example Local: 1, Global: 3 takes the local velocity vector and multiplies it by a random number \* 1 while the global velocity vector is then multiplied by a random number \* 3. This is modelled after this equation:

Vnew = Vcurrent + Local\*rand\*(Xlocal best – X) + Global\*rand\*(Xglobal best-X)

These results once again prove to be very interesting as the only learning type that did not converge was Local:1, Global: 3. I believe that Local: 3, Global: 1 converged quickly because each particle essentially got “stuck” in its local best solution because the emphasis to stay where the particle is currently is much greater than the incentive to search towards the global best which is usually much further away from the local best. This discourages large movement which is why I believe Local: 3, Global: 1 converged quickly.

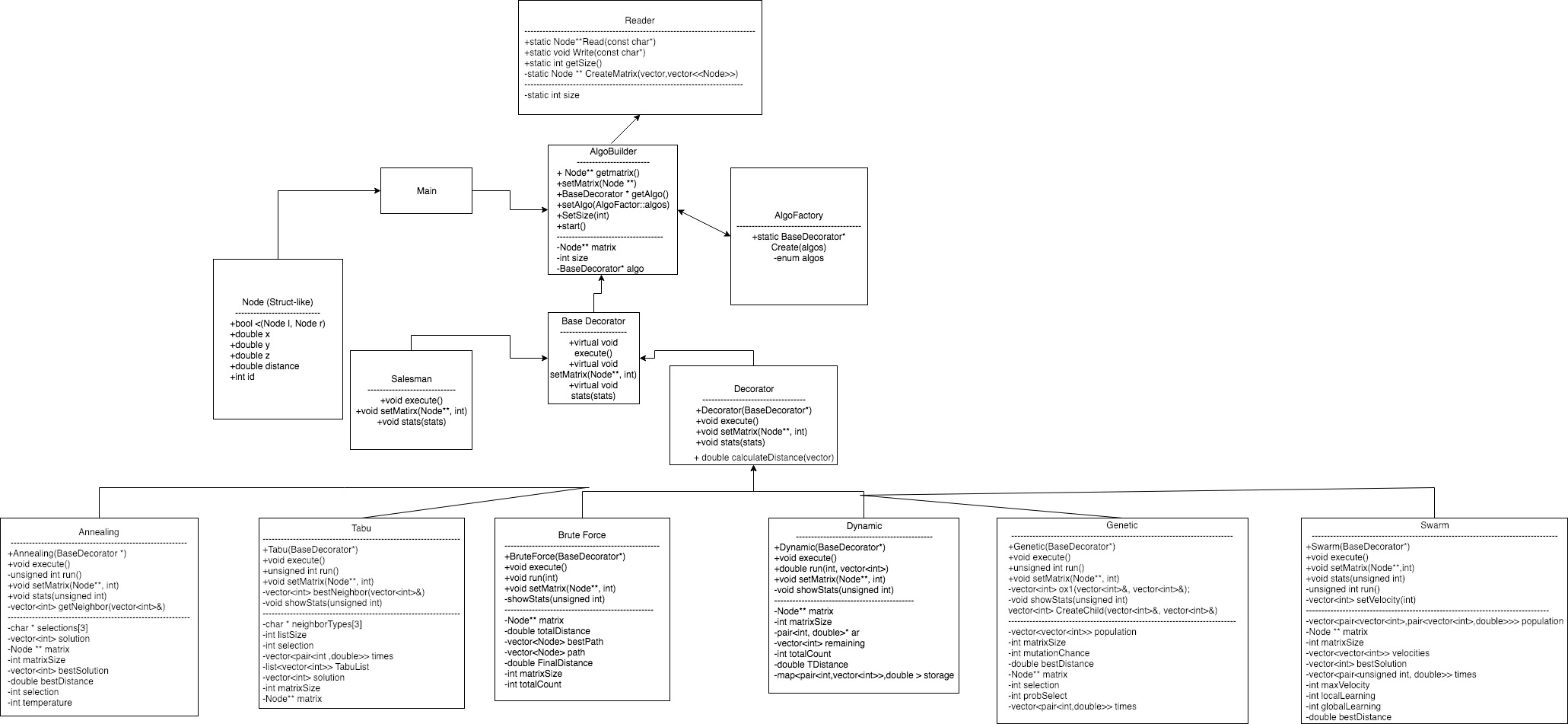
Local: 2, Global: 2 took a longer time to converge than Local: 3 Global: 1, but it still managed to fail to find any good solutions after a certain point in time. This led me to believe that this approach simply confuses the particles since it has no clear direction in where it should point. This leads the particle to wander more than other learning factors. This will eventually lead to particles converging to a global best without the proper direction to explore a lot of the hot-spots.

Local: 1, Global: 3 proved that it took the longest amount of time. It actually terminated before it had a chance to converge which lead me to believe it would have kept going and find even better solutions than the current best solution that it found. I believe it did not converge because usually global bests are further away from local bests which encourages particles to explore more than the other two types. This effectively causes the particle to look around the search space more and perhaps even find better global solutions as it moves towards the possible “hot-spots” where global bests are more likely to be found.

Overall, I believe that Local: 1, Global: 3 is the best way to approach PSO because it encourages particles to search and find better solutions continuously, and since it takes a lot longer for this version to converge it will spend a lot more time searching the graph than the others.

**DESIGN:**

(For a better look at the Jpeg file go to Lab05UML.jpeg in my repo)

****

For this lab I basically did the same exact thing as the last lab. I used a builder interface called AlgoBuilder that stored the graph to send it easily across the entirety of the project. Main calls the AlgoBuilder to read the graph file and save it as an adjacency matrix, and then we use the AlgoFactory interface to create which algorithms we need for the project. The builder makes it easy to send large bits of data across the algorithms which prevents me from having to load in the graph every time a new algorithm is called.

The AlgoBuilder then uses a BaseDecorator which has both a decorator and a salesman inheriting off of it. This is a textbook decorator pattern which allows us to have easy delegation within the decorator, and since they all share the same interface it is very easy to create a new algorithm and use it. The use of the factory and the simplicity of the decorator pattern paired together allows for changing of an algorithm in only around one or two lines of code, and this saves me time from writing the same type of code multiple times.

The algorithms inheriting off of a decorator was also helpful since there were some functions in the decorator class that were very helpful for me and prevented me from writing the same identical methods multiple times. For example, calculating distance was something that is crucial to these algorithms because it determines fitness. I never calculated fitness once in one of the algorithm classes: I let the decorator class do all the fitness calculations for me to save my time.

All of this together has made my main very small since most of the work being done is in the algorithm classes themselves and all of the other classes are necessarily helper classes to allow for delegation. I did not make any changes to this design since lab 3 except for adding new algorithms to the decorator. Thankfully, this design saved me a lot of time and allowed me to focus solely on the algorithms rather than the logistics of moving data around.