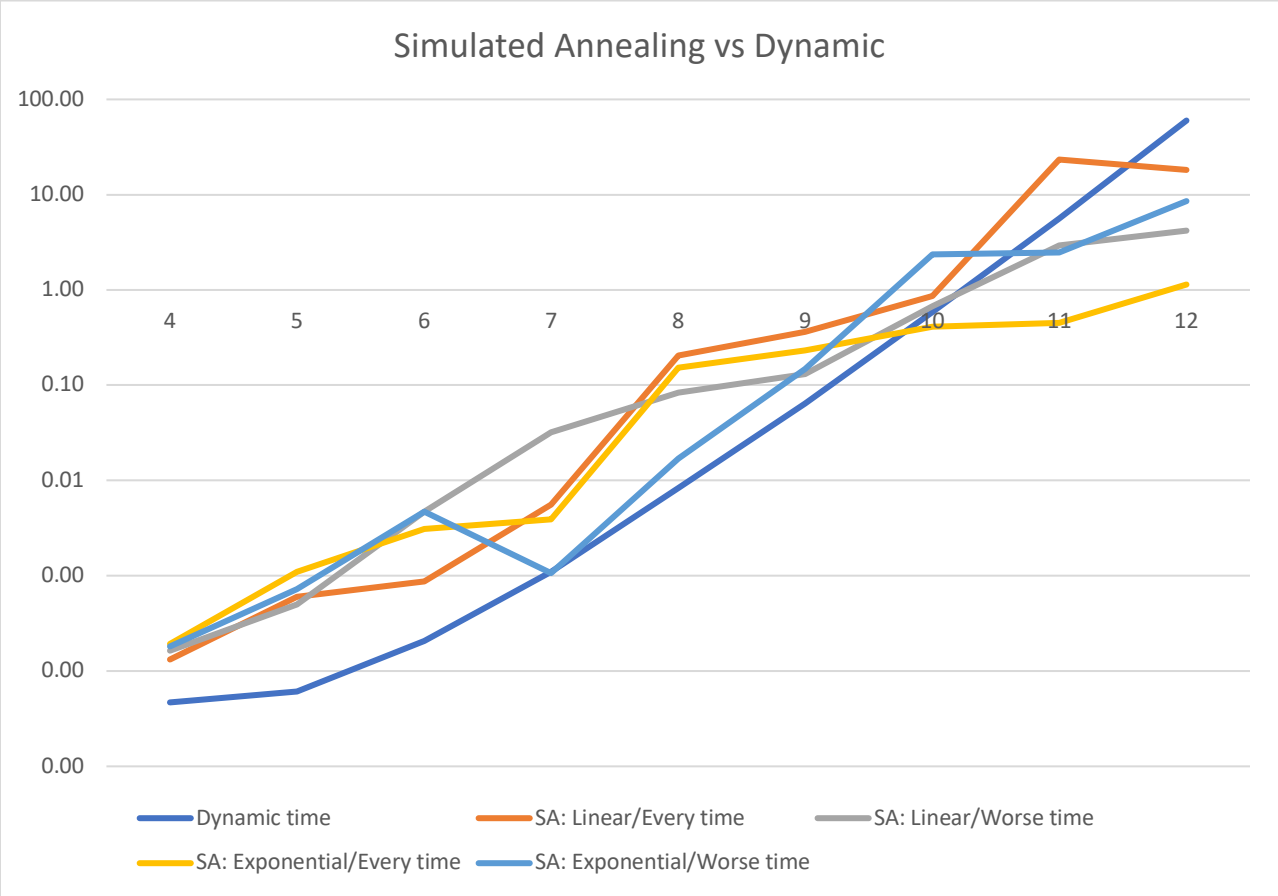
Emily Fashenpour
CSE 3353

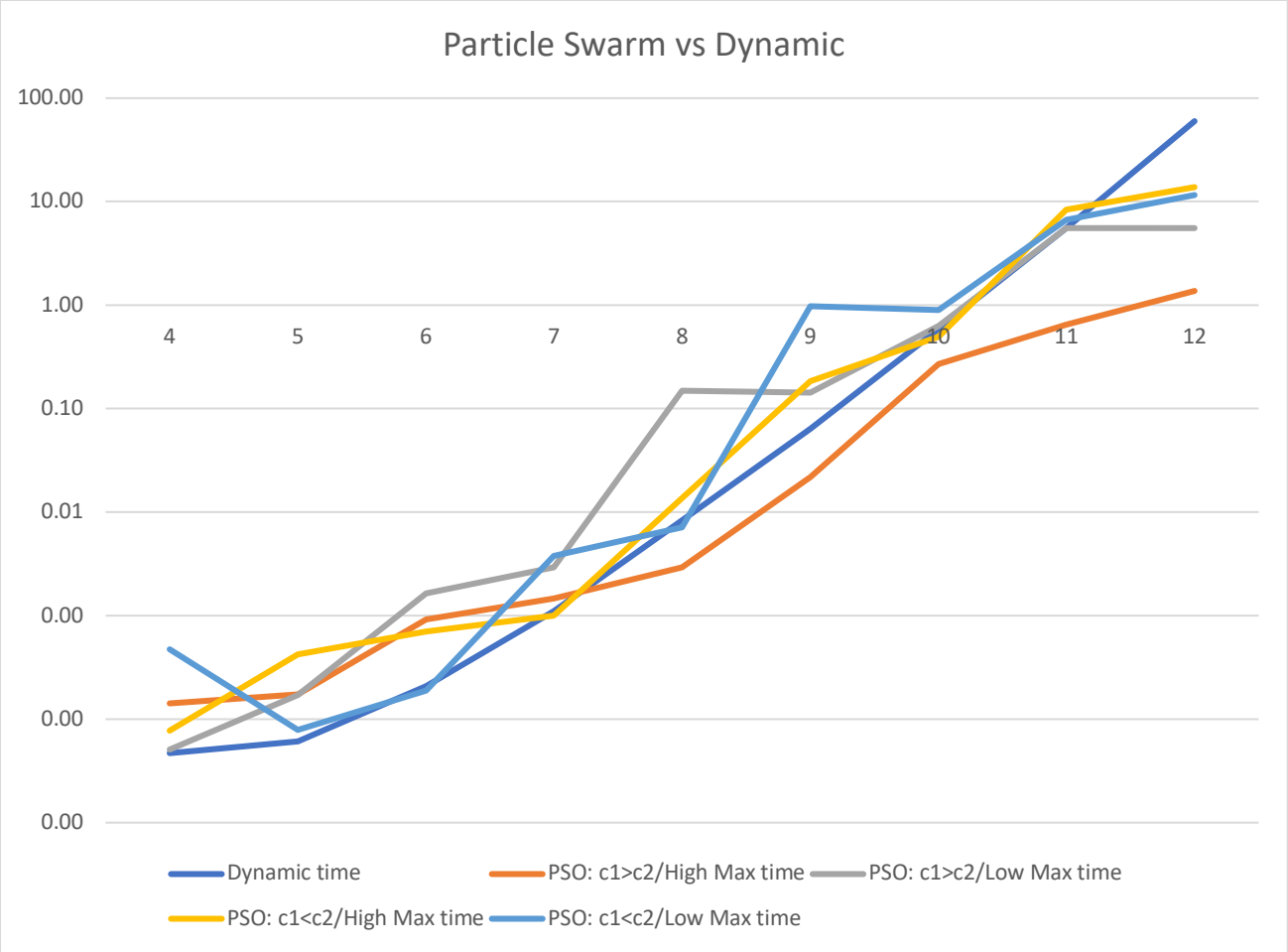
Lab 5 Report: Traveling Salesman Problem

Simulated Annealing and Particle Swarm Algorithm

Timing



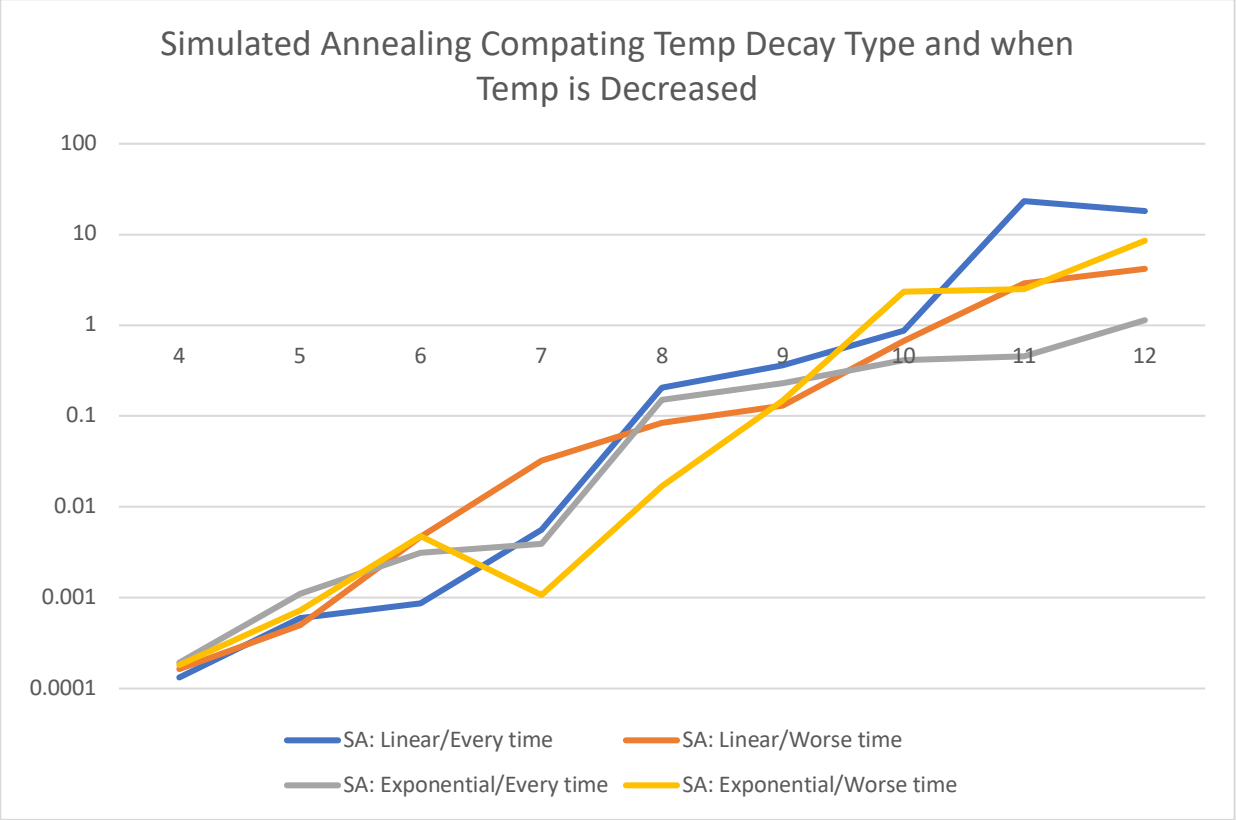
node	Dynamic time	SA: Linear/Every time	SA: Linear/Worse time	SA: Exponential/Every time	SA: Exponential/Worse time
4	4.69E-05	0.000132244	0.000163754	0.000192691	0.000181397
5	6.07E-05	0.000598444	0.000501891	0.00110871	0.000727617
6	0.000208	0.000868897	0.00468564	0.00311098	0.00473021
7	0.00109973	0.00557262	0.0322084	0.00389332	0.00106534
8	0.00828905	0.205079	0.084233	0.152028	0.0168879
9	0.064221	0.365068	0.131221	0.230403	0.149111
10	0.585676	0.866621	0.669972	0.410984	2.35452
11	5.59331	23.3698	2.91524	0.452572	2.49178
12	60.0054	18.141	4.19649	1.1404	8.56321



node	Dynamic time	PSO: c1>c2/High Max time	PSO: c1>c2/Low Max time	PSO: c1<c2/High Max time	PSO: c1<c2/Low Max time
4	4.69E-05	0.000141879	5.08E-05	7.74E-05	0.000470742
5	6.07E-05	0.000174176	0.000171947	0.000424001	7.86E-05
6	0.000208	0.000921315	0.00163687	0.000703284	0.000188516
7	0.00109973	0.00146403	0.00291701	0.000999591	0.00378081
8	0.00828905	0.00292927	0.14825	0.0135906	0.00709194
9	0.064221	0.0216632	0.143283	0.185097	0.974026
10	0.585676	0.26993	0.632483	0.494686	0.898132
11	5.59331	0.649286	5.53693	8.40801	6.70244
12	60.0054	1.3725	5.55373	13.8195	11.587

Summary of Results

Simulated Annealing



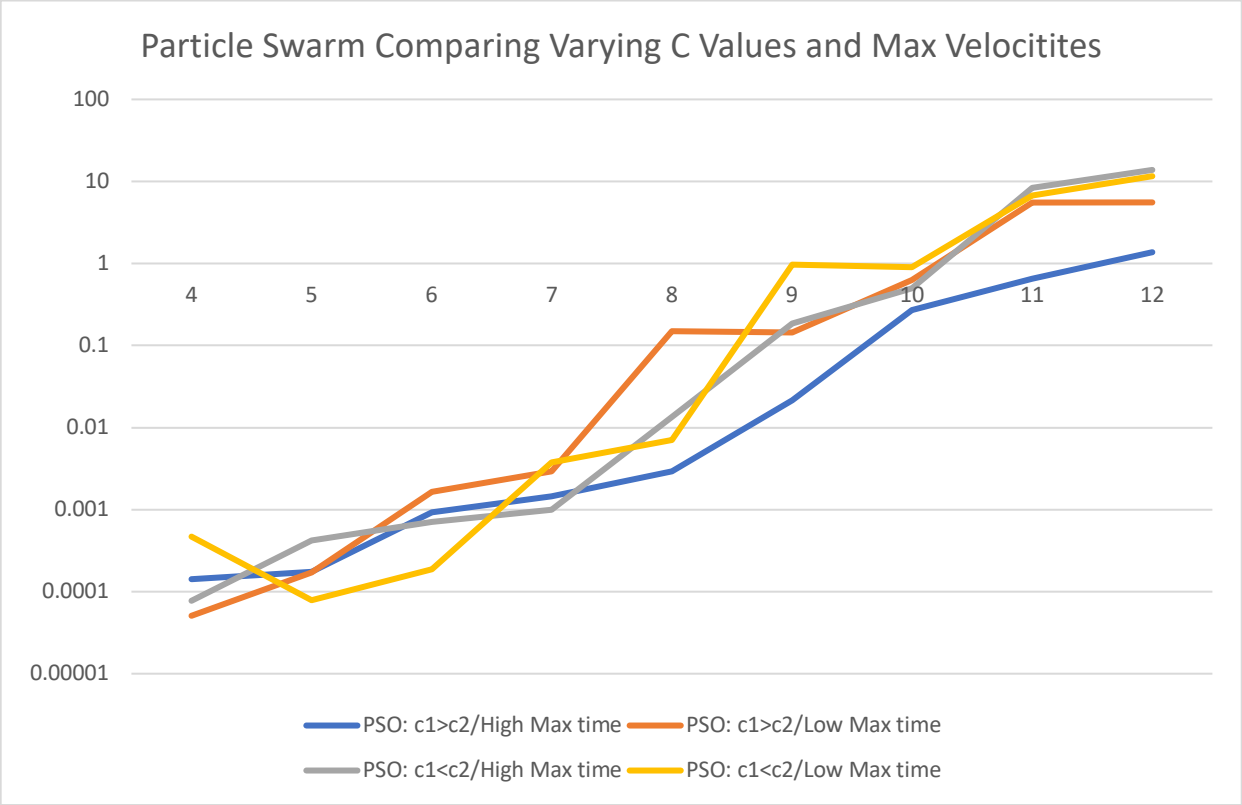
Overall, the best results were found in the quickest amount of time when the temperature decreased exponentially after every iteration. This makes sense because the probability that a worse solution is accepted decreases the quickest here. By decreasing when a worse solution is accepted, there is a chance that bad solutions are being used to generate neighborhood more often than a better solution because the worse ones have a higher probability of being accepted. This is the same idea for decreasing temperature linearly because the probability that a worse solution is accepted does not decrease as quickly as the exponential as time increases, meaning the worse solutions have a higher probability of being used to generate solutions if the random solution chosen is not a better solution than the current one.

Compared to the dynamic approach used in Lab 3, dynamic outperforms the simulated annealing algorithm when there are fewer nodes by once there need to be a solution found for 10 nodes and up, the SA algorithm begins to outperform the dynamic approach.

I also found that as the number of nodes increased, the initial temperature needed to be increased as well in order for the solution to be found, which makes sense because that just gives the algorithm more time to find an optimal solution.

Since the Simulated Annealing algorithm is a stochastic search algorithm, meaning that the different parameters (initial temperature, temperature decrease type, and when to decrease the temperature) used to find the optimized solution use randomness to create and find solutions, it does not have definitive asymptotic complexity. However, SA's complexity is relative to its parameters.

Particle Swarm



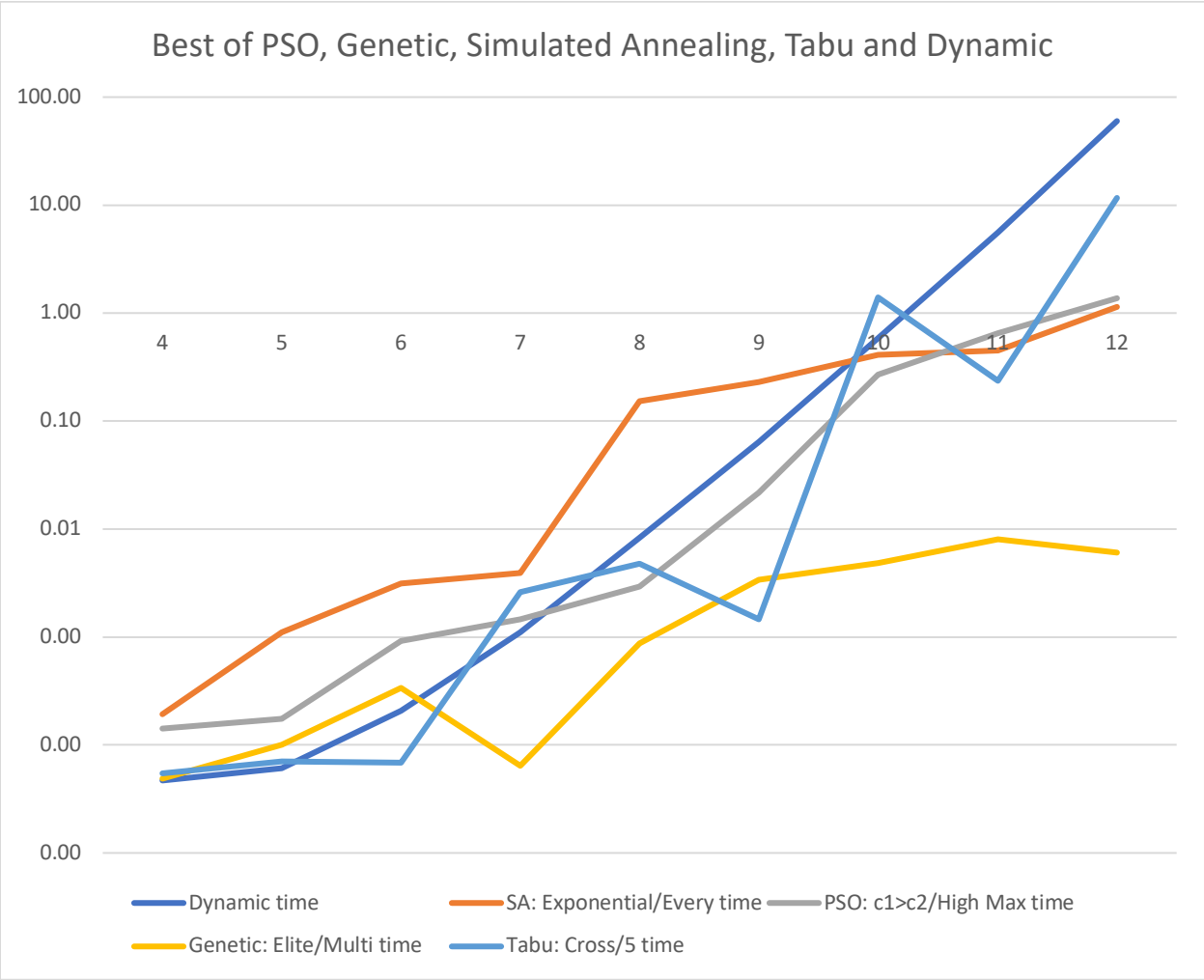
Overall, the best results were found when the C1 value was greater than the C2 value and the max velocity was high. This makes sense because the max velocity helps ensure that there are larger values allowed in the velocity vector. Also, because the more emphasis there was on the local best instead of the global best, all the particles didn't move towards the global and end up in the same area of a solution that was optimal but not the best. The emphasis on the local best solution helped ensure that each particle was moving around its own area with an optimized solution and not all clumped together near the global best and getting stuck in that area.

Compared to the dynamic approach in Lab 3, dynamic outperforms the Particle Swarm algorithm for lower nodes, however the particle swarm algorithm begins to outperform the dynamic approach when there needs to be 7 nodes and more searched.

I found that by increasing the number of particles, the number of neighbors created in the neighborhood, and the number of iterations as the number of nodes increased, the quicker an optimal solution was found.

Since the Particle Swarm algorithm is a stochastic search algorithm, meaning that the different parameters (number of particles, number of iterations, C1/C2 values, and maximum velocity) used to find the optimized solution use randomness to create and find solutions, it does not have definitive asymptotic complexity. However, PSO's complexity is relative to its parameters.

Comparison



node	Dynamic time	SA: Exponential/Every time	PSO: c1>c2/High Max time	Genetic: Elite/Multi time	Tabu: Cross/5 time
4	4.69E-05	0.000192691	0.000141879	4.88E-05	5.46E-05
5	6.07E-05	0.00110871	0.000174176	0.000100618	7.03E-05
6	0.000208	0.00311098	0.000921315	0.000338905	6.88E-05

7	0.00109973	0.00389332	0.00146403	6.41E-05	2.60E-03
8	0.00828905	0.152028	0.00292927	0.000873595	4.78E-03
9	0.064221	0.230403	0.0216632	0.0033984	0.00144959
10	0.585676	0.410984	0.26993	0.00482834	1.38885
11	5.59331	0.452572	0.649286	0.00803185	0.234488
12	60.0054	1.1404	1.3725	0.00607652	11.6539

Overall comparing the best combinations of the different parameters in Genetic Algorithm, Tabu Search, Simulated Annealing, Particle Swarm, and Dynamic, the Genetic Algorithm outperforms all other algorithms. This makes sense because the genetics algorithm (using elite selection and multipoint crossover) uses the least randomness, meaning while there is randomness, the solution generated of randomness are being produced using optimal solutions found in past generations. There are instances where another algorithm could perform better than genetic because there is so much randomness used in each algorithm that the algorithm could randomly stumble upon the best solution at the beginning of its search. It is possible for the search algorithms to perform better if the initial random solution happens to already be a somewhat optimized solution because then the solution generated from the initial already have a good starting place to search from.

For all the search algorithms, especially when the number of nodes became larger, the algorithm had to be run a few times for the most optimal solution to be found and some of the parameters had to be changed as well.

Design Documentation

Design Patterns Utilized

- Strategy Pattern
    - This design pattern will easily allow for other algorithm types to be derived, allowing for future expandability. This also allows for code reusability. In my program, AbsAlgorithm is the parent class and TSP is the derived class. In the future, if another problem needs to be solved like TSP, it can become a derived class of AbsAlgorithm. TSP uses the two algorithm classes Simulated Annealing and Particle Swarm as two different algorithms to solve the TSP problem.
- Factory Pattern
    - This pattern generates an AbsAlgorithm object pointer that creates the TSP object. In the future, if another problem were derived from AbsAlgorithm, there would be a way to create an object pointer of that type in the create function. This helps with future expandability as well.
- Chain of Responsibility
    - This pattern is utilized in the TSP, TSPAnnealing, TSPSwarm, and Particle classes. Chain of responsibility makes it easier to call multiple functions on an object in

one line instead of having several lines with one function call. I used this in my program because I needed to call several functions to set variables and finally to run the algorithm and wanted to reduce the number of lines needed to call multiple methods on an object.

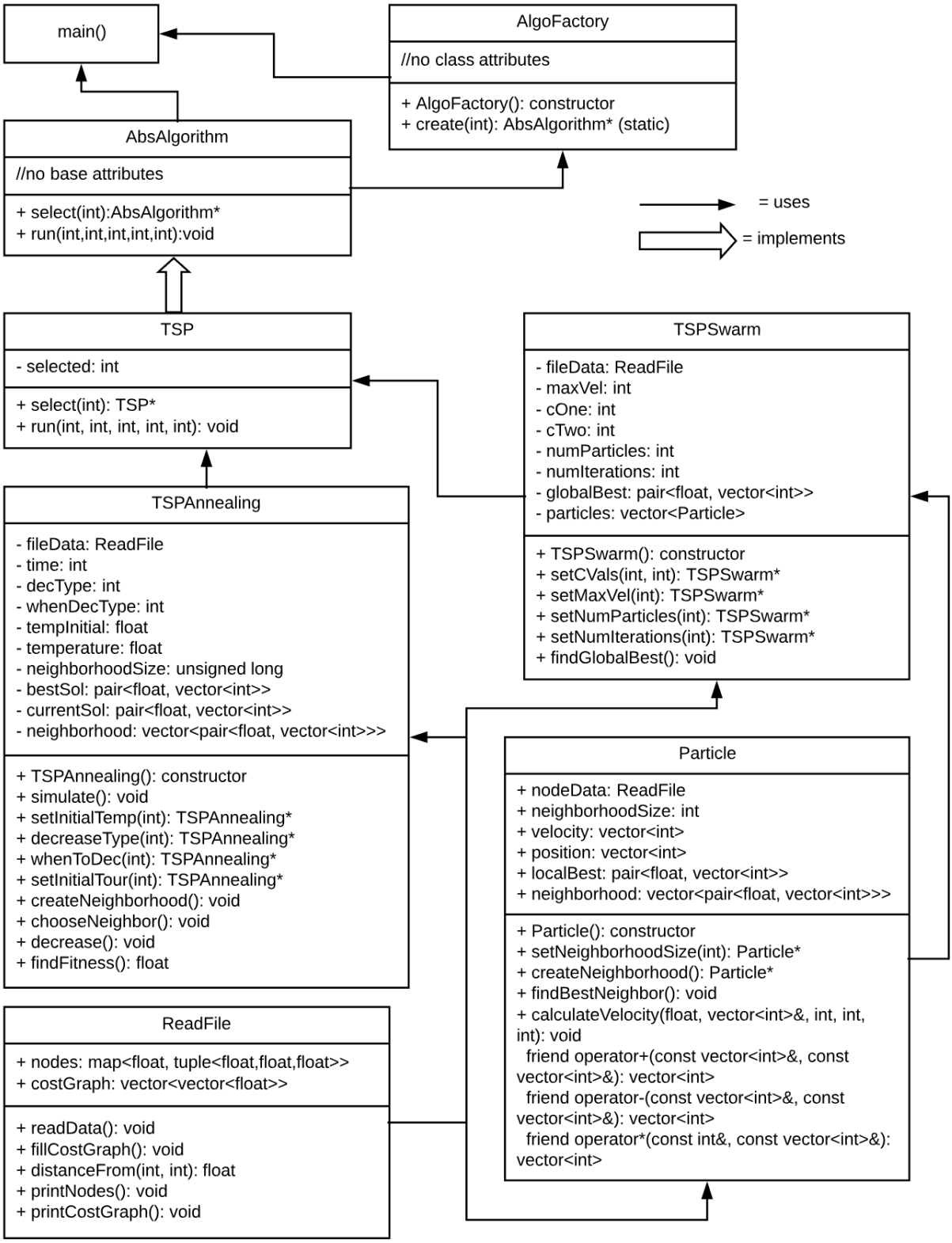
In main, I use the factory design pattern by calling the create function and setting the returning value to an AbsAlgorithm object pointer. The create function takes a problem option enum which determines what type of object is returned by the create function.

Once the AbsAlgorithm object pointer is initialized to a problem option type, the algorithm to be used on the problem is set in the select function and then the run function accepts any arguments that are needed for running the algorithm.

- For Simulated Annealing, the accepted parameters include:
    - Initial Temperature: the initial temperature was set to a value that was used as the stop case for the algorithm. Once the temperature had reached zero, then that was the sign that there can be no more moves made.
    - Temperature Decrease Type: the temperature could either decrease exponentially using exponential decay or it could decrease linearly using linear decay.
    - When to Decrease Temperature: the temperature was either decreased every iteration or when there was not a better solution random picked from the neighborhood, but it was accepted as a worse solution.
- For Particle Swarm, the accepted parameters include:
    - Number of Particles in the Swarm: sets the number of particles used to swarm towards an optimal solution.
    - Number of Iterations: the number of times the particles could swarm to a new position. Acts as the stopping case for the algorithm.
    - C1 and C2 Values: C1 and C2 values are used when calculating the velocity of a particle. C1 has an effect on the weight of the local best solution in a particle and the C2 value has an effect on the weight of the global best solution from all the particles in the swarm when determining where the particle needs to swarm to.
    - Max Velocity: determines the max jump in space a particle can make.

There were no changes made in the structure of the program from Lab 4 to Lab 5. The design patterns where kept the same, just reimplemented for the new algorithms.

UML Diagram


main()
AlgoFactory
//no class attributes
+ AlgoFactory(): constructor
+ create(int): AbsAlgorithm* (static)
AbsAlgorithm
//no base attributes
+ select(int):AbsAlgorithm*
+ run(int,int,int,int,int):void
= uses
= implements
TSP
- selected: int
+ select(int): TSP*
+ run(int, int, int, int, int): void
TSPSwarm
- fileData: ReadFile
- maxVel: int
- cOne: int
- cTwo: int
- numParticles: int
- numIterations: int
- globalBest: pair<float, vector<int>>
- particles: vector<Particle>
+ TSPSwarm(): constructor
+ setCVals(int, int): TSPSwarm*
+ setMaxVel(int): TSPSwarm*
+ setNumParticles(int): TSPSwarm*
+ setNumIterations(int): TSPSwarm*
+ findGlobalBest(): void
TSPAnnealing
- fileData: ReadFile
- time: int
- decType: int
- whenDecType: int
- tempInitial: float
- temperature: float
- neighborhoodSize: unsigned long
- bestSol: pair<float, vector<int>>
- currentSol: pair<float, vector<int>>
- neighborhood: vector<pair<float, vector<int>>>
+ TSPAnnealing(): constructor
+ simulate(): void
+ setInitialTemp(int): TSPAnnealing*
+ decreaseType(int): TSPAnnealing*
+ whenToDec(int): TSPAnnealing*
+ setInitialTour(int): TSPAnnealing*
+ createNeighborhood(): void
+ chooseNeighbor(): void
+ decrease(): void
+ findFitness(): float
Particle
+ nodeData: ReadFile
+ neighborhoodSize: int
+ velocity: vector<int>
+ position: vector<int>
+ localBest: pair<float, vector<int>>
+ neighborhood: vector<pair<float, vector<int>>>
+ Particle(): constructor
+ setNeighborhoodSize(int): Particle*
+ createNeighborhood(): Particle*
+ findBestNeighbor(): void
+ calculateVelocity(float, vector<int>&, int, int, int): void
friend operator+(const vector<int>&, const vector<int>&): vector<int>
friend operator-(const vector<int>&, const vector<int>&): vector<int>
friend operator*(const int&, const vector<int>&): vector<int>
ReadFile
+ nodes: map<float, tuple<float,float,float>>
+ costGraph: vector<vector<float>>
+ readData(): void
+ fillCostGraph(): void
+ distanceFrom(int, int): float
+ printNodes(): void
+ printCostGraph(): void

Outcomes

Simulated Annealing:

- Initial Solution: a randomly generated permutation of all the nodes.
- Creating the Neighborhood: will generate a neighborhood of similar solutions to the current by swapping a few nodes in the solution.
- Choosing a Neighbor: a randomly generated number between zero and the neighborhood size is generated, if the solution has a better fitness than the current solution, the that becomes the neighbor that the current solution moves to. Also checks to see if the current solution is a better solution than the best solution found so far and sets the best if so. If the solution that is randomly chosen has a worse fitness than the current solution, a probability that the solution is chosen is determined based on the difference in the fitness of the two solutions and the temperature. The higher the temperature, the more likely the worse solution will be accepted. If the solution is not accepted, then the current solution is not changed, and a new neighborhood is generated, and another random solution is chosen.
- Decreasing Temperature: temperature can either be decreased when a worse solution is accepted or after each neighborhood is generated. The way temperature is decreased is either exponentially, like Newton's cooling law, or linearly. The way that the temperature is decreased also has a large effect on the probability that a worse solution is accepted. When decreasing linearly, the worse solution has a better chance of being accepted.
    - Linear: the temperature is decreased by one after every neighborhood is created or when a worse solution is accepted. (Temp = temp – 1)
    - Exponential: temperature decays exponentially at a rate of (1/ (number of nodes * 100)). The smaller the rate, the longer the algorithm has to find an optimal solution. Temp = (T initial) * e^(-rate) – 1
- Stop case: the algorithm will continue to find solutions until the temperature reaches zero.
    - The higher the initial temperature is set, the longer the algorithm has to find the most optimal solution.

Comparing the two decay types, exponential and linear, it makes sense that exponential would outperform linear because of the effect that temperature has on the probability that a worse solution is chosen and then used to create a neighborhood of solutions that may not be as optimal if you were to create a neighborhood using an optimal solution. This is a similar idea to why decreasing temp after every neighborhood is created instead of when a worse solution is accepted because the probability of accepted a bad solution is higher.

Particle Swarm:

- Initializing Initial Particles: a vector of particles the size determined by the used is generated. In each particle, the initial solution is set by creating a random permutation of all the nodes and the initial solution becomes the starting position of the particle.

Then the neighborhood is filled by making a few swaps to the initial solution. Once the neighborhood is full, the best solution from the is found and set as the local best in the particle. After all the particles are initialized, the global best solution is found by finding the best local solution from all the particles.

- Calculating Velocity: In each particle, the velocity is initially set to zero and is changed relative to the global best of all the particles and the local best solution for the current particle, the C1 and C2 values, a random number generated between one and zero, and the maximum velocity.
    - Calculated by finding the vector of (cOne * rand1 * (position of local best – position of current)) + (cTwo * rand2 * (position of global best – position of current)). Once that is found, that vector is then multiplied by maximum velocity and then divided by its magnitude (magnitude calculated before the vector was multiplied by max velocity). Then this vector is added to the initial velocity and will then cause the particle to move.
- Swarming: Once the velocity is calculated for each particle, the particle then moves to a new solution. The index of the two highest numbers in the velocity vector are found and swapped randomly with other nodes in the solution. Also, the newly generated position (solution) is then compared to the global best and global best is set to the new position if it was found to have a better solution.
    - The indexes of the highest values found in the velocity have the greatest variation from the local best and global best so by swapping two not optimal nodes with two random nodes in the solution, it is possible that the random swaps created an even better solution.
- Stop case: the particles will continue to swarm until the set number of iterations is met.

Comparing maximum velocities, it makes sense that a higher maximum velocity would outperform a lower one because it helps ensure that the particles are moving and finding more optimal solutions. Similarly, it also makes sense that putting more emphasis on the local best solution instead of the global best solution when calculating velocity would perform better because by having each particle swarm around its own somewhat optimized solution, it is able to create solutions that are optimal away from other particles, meaning it helps ensure that the particles don’t get stuck and clumped around the global best solution.