Reid Russell

CS 3353

5 December, 2019

Lab 4:

TSP Using a Genetic Algorithm and Tabu Search

Introduction

The ultimate goal in Lab 4 is an extension of Lab 3: Solve the travelling salesman problem. The goal in the travelling salesman problem is: given a “graph” of fully connected nodes or “cities”, find the shortest path that starts at any point, ends at the starting point, and travels to each city exactly once. In Lab3, the methods used to solve this problem were brute force, a O(n!) time complexity problem, and dynamic programming- a time complexity that varies between approaches but from the method that I used: O(n22n). While this was an improvement from n!, the time complexity still limited the search size in reasonable time to around 21 nodes. The two methods for solving the TSP that are used in this lab are Tabu Search(TS) and Genetic Algorithm(GA).

Tabu Search is a meta-heuristic, non-deterministic search algorithm that modifies the basic hill climbing algorithm by adding a “tabu list” of fixed size, where movement to a node causes that node to be pushed back onto the list. If the list is full, the algorithm will pop the oldest member of the data structure off of the list in order to maintain a fixed size. Movement, like in Hill Climbing, is determined by the “neighbor” in the generated “neighborhood” that has the highest or lowest “fitness” depending on the goal. For my solution, each neighbor was a full solution to the TSP, and its fitness was the total distance of traversal in the order that the nodes appear in the neighbor. Because we are trying to minimize the distance in the TSP, the node that will be moved to will be the one with the lowest fitness, but also not in the tabu list, unlike Hill Climbing. In the event that nothing in the neighborhood has a lower fitness than the current node, TS chooses the next best option that is not in the tabu list. Having the tabu list will allow the algorithm to avoid getting stuck in local optima because local optima will be on the tabu list after it has been visited and cannot be explored again until it is popped off of the back of the list.

Genetic algorithm is based on nature. You take individuals with “chromosomes”, which contain all of the necessary pieces for the solution, and generate a population. In this case the chromosome is a valid path of nodes. The population is then sorted by fitness, and depending on certain parameters a certain percentage of the most optimal fitness score seed an entire new population. For my algorithm this was the top 10%, who mates with the top 50% of the population. Genetic algorithms are non-deterministic, so you have to specify a stopping condition like convergence onto a path or max number of iterations- both of which my algorithm does depending on the goal.

Results

Above is the results of running my genetic algorithm from a 22 node graph to a 33 node graph where the algorithm had 10 minutes for each graph to optimize the population and converge on a minimum distance. From the graph, you can see this happens very quickly- around the 100 generation mark for most instances. I was actually very surprised that my algorithm would so quickly converge on a point if looking for a known path, or like in the above graph drastically increase the optimal fitness of the population in most cases. I could not find any case where the most optimal individual increased its fitness away from the direction of optimization- this showed me that it was working correctly. For the population size, I tried several different configurations and found that larger graphs benefitted from larger population sizes. Because of this I made the population size a constant times n, where n is the number of nodes in the graph.

For the mutation rate, I increased it from an initial 5% to 15% because I found that at took low of levels, the algorithm would get stuck in local optima and in higher levels would fail to converge on a good solution and be too random. 15% was used based on trial and error.

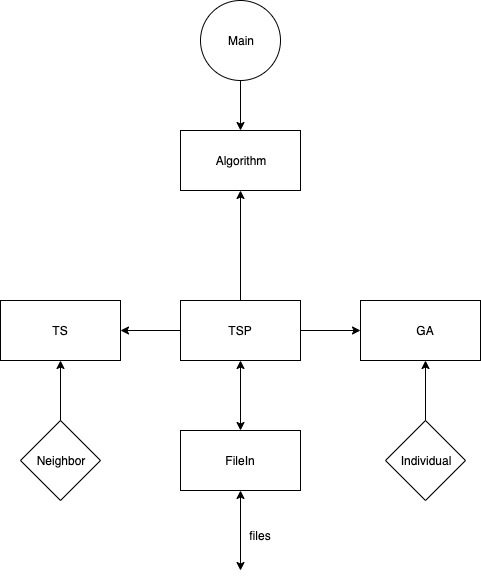
Tabu search, like GA, also rapidly increased the fitness of the population at a few generations. However, the more time TS had to run, the better. From the graph above you can see how the algorithm spikes up and down, indicating areas where the fitness went in the opposite direction of optimal, but TS given enough time would always eventually plateau near the lows. For the generate neighborhood function, I at first was swapping values on opposite sides of a vector of nodes and pushing back each iteration, but I found that this method would calculate and generate 2 of everything- vastly reducing the algorithms effectiveness. Instead I settled on a neighborhood that was generated by swapping each position in the vector with every other position in the vector and then to sort them by fitness.

I separated these two graphs purposefully because combining them would have lost all of the information that would distinguish them because of TS’s spikey nature. Both graphs are on the same scale, and both show 22 to 33 node graphs.

Comparison to Lab 3 Timing

I was unable to generate the graphs to display the timing differences from these two algorithms and Lab 3 algorithms due to time constraints. But I did experimentally show how much faster they were- especially when using GA. The huge difference in speed of these algorithms compared to deterministic has put the benefits on non-deterministic algorithms into perspective on why they are so useful- even if you aren’t 100% sure that the answer it spits out is the best option, but it will be a good option (If the configurations of the algorithm are correct).

UML



For this program I decided to design my code using an overall Strategy Pattern. I chose the strategy pattern because this is the way I structured Lab 3 so I was able to re-use most of the interface, only replacing the individual algorithm classes. The Strategy Pattern offers the best compartmentalization of code so that in the future it will be simple to add other solutions for the TSP.

The strategy pattern also forces code to be more concise and reusable because each algorithm added conforms to the exact same interface, making the downtime for adding a new algorithm minimal once the interface has been set up. In this design, Algorithm is a pure virtual function that provides the layout for the interface. TSP is where that interface is implemented in a way that it can use the same code for different algorithms. The main function with this design is the its reusability and scalability.

Aside from the strategy pattern implementation, the rest of my code followed the object-oriented approach, that allows the program to abstract features and variables into a bundle that can be reused. For the file input I created a “FileIn” class that was responsible for taking in the file-path, loading a particular file, storing the contents of that file, and writing statistics to csv files to be graphed later on. I chose to do this so that TSP merely has to hold a FileIn object and interact with that object on all things files instead of having to handle file reading, storage, output itself. I also built a simple Node class, that is used in storing the contents of the file and having information on each node ready to return when needed. The data stored in the nodes was a node id and a vector of the x, y, and z positions of that particular node. This way I merely had to keep a vector of nodes and the nodes themselves kept track of their positions and ids.

The Genetic Algorithm is split up into the large genetic algorithm class and an individual class. The individual keeps track of paths in the population, as well as each path fitness and hosts several helper functions that are used in mating, sorting the population, mutation, and generation. The GA handles the overall execution of the algorithm and keeps track of each generation and the things like the population size, mating ratio, and techniques like Elitism. When developing the GA, I tested out several different populations sizes and mutation rates. One of the things I discovered was that larger population sizes were much better for graphs with higher amounts of nodes even though they slow down each generation, so to scale with whatever graph is being used the population size is a factor of n\*15. Why multiply by 15? Because after allot of trial and error that seemed to most often give me the exact correct solution- especially in larger graphs. When I realized my algorithm was getting stuck on local optima, I started tweaking the mutation rate- specified in Individual::mate(). Originally I had the mutation rate set to 5%, but in this case the algorithm would often get stuck in local optima. When using convergence, a higher mutation would make it much harder to converge onto a solution, so after much trial and error I set the mutation rate to 15%. For all of the graphs that were successfully traversed in Lab3, my Genetic Algorithm was able to converge onto the same solution- but much faster. For known solution graphs, I have the algorithm “converge” onto the most likely answer and return after a certain number of generations and their difference in distance do not change.

The Tabu Search algorithm was fairly simple to implement. Much like GA, I split it up into two classes, to simplify the subproblems. The major subproblems I categorized were: to generate the “neighborhood”, to get the next move from that neighborhood and then to repeat until a desired end condition is met. I created a “neighbor” class that much like the individual in GA, would keep track of path and fitness for each neighbor in the neighborhood. To construct the tabulist itself, I used a deque. I chose this data structure because it was the one thing I could find that would allow me to push to the front and pop from the back while still being able to iterate through to compare and see if a value was already there.