CS 325
Traveling Salesman Project

Project 14 Group

Christopher Buttacavoli
Hyoung Sik Won
Michael Sio

Introduction

The purpose of this report is to provide an explanation of different types of algorithms used to solve the Traveling Salesman Problem (TSP) and our implementation in our program to solve the problem. The TSP is a classic NP-complete problem that stumps many researchers. The reason is that NP-complete problems do not currently have a way to be solved in polynomial time, thus preventing an efficient algorithm from existing to find the optimal solution. The accepted method for finding a solution to NP-complete problems is to use some sort of heuristic that will give a good, near-optimal solution in a shorter amount of time than brute force. The algorithms that we researched to solve this problem were: Genetic Algorithm, Ant Colony Optimization, and 2-Opt.

Researched Algorithms

Genetic Algorithm

Genetic algorithm is an interesting algorithm that adopted concept of natural selection theory from evaluation into an optimization algorithm. Although it does not guarantee the best solution for a problem such as TSP, it discovers a solution that is close enough to the optimum solution with more efficiency. Here is very simple overview of how the Genetic algorithm is used to solve TSP problem.

Start from selected number of the random population and pick out the candidates with highest fitness and perform crossover to reproduce different population with mixture of good traits. Then it continues until the program doesn't see noticeable improvement. Mutation happens with some probability to prevent local optimization, also known as premature convergence (see figure 1). If it wasn't for local optimization, a simple greedy algorithm would have been sufficient.

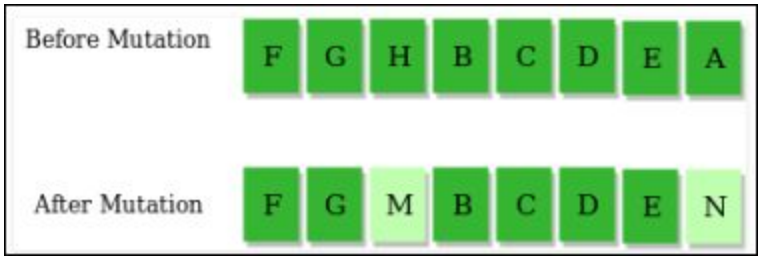

Figure 1. Premature Convergence

Applying the concept to TSP in very simple sense, program picks random orders that include every vertex. Next, we calculate the distance of each and pick the ones with shorter distances. Then the algorithm performs crossover by mixing position of certain

vertices. This crossover is expected to yield solution with better fitness since we are taking the ‘good’ traits from each solution to come up with a new solution (see figure 2).

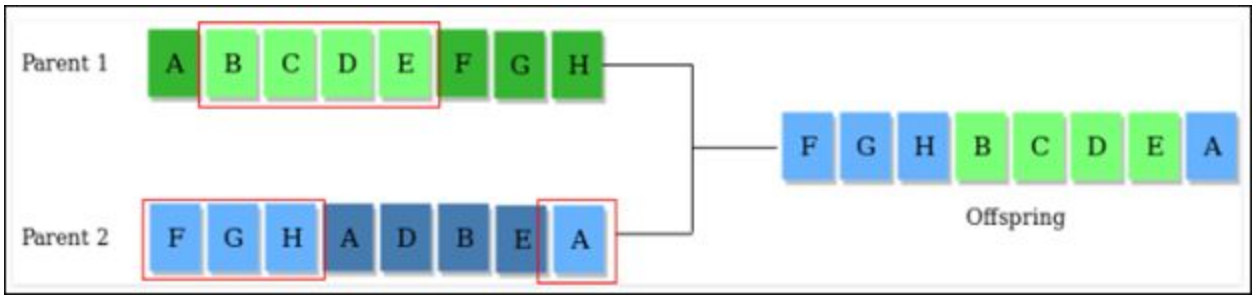

Figure 2. Genetic Crossover

Mutation happens which randomly mixes up the vertices. The process repeats until it doesn’t see a noticeable improvement in fitness or reduction in distance. It may not generate ‘the optimal’ solution, but it will create one that is close enough with more efficiency, and the efficiency will add more value as the size of vertices gets larger. It seems like this algorithm will be useful when there is very large amount of vertices to be visited for TSP.

GeneticAlgorithm(vertices):
Generate the initial population (randomly)
Compute fitness
Repeat until population has converged
    Selection of parent population
    Crossover and generate new population
    Mutation and generate new population
    Compute fitness for new population
Return final population

Ant Colony Optimization

Something that has always fascinated scientists has been how ants always seem to find the fastest way to a food source without any sort of map. This optimization method copies the behavior of traveling ants in order to find the fastest cycle between two points. Ants will begin by randomly choosing paths to traverse.

Figure 3. Ant locating food source

When an ant (red ant) finds a path between the two points, it will travel back and leave a trail of pheromones (yellow in picture).

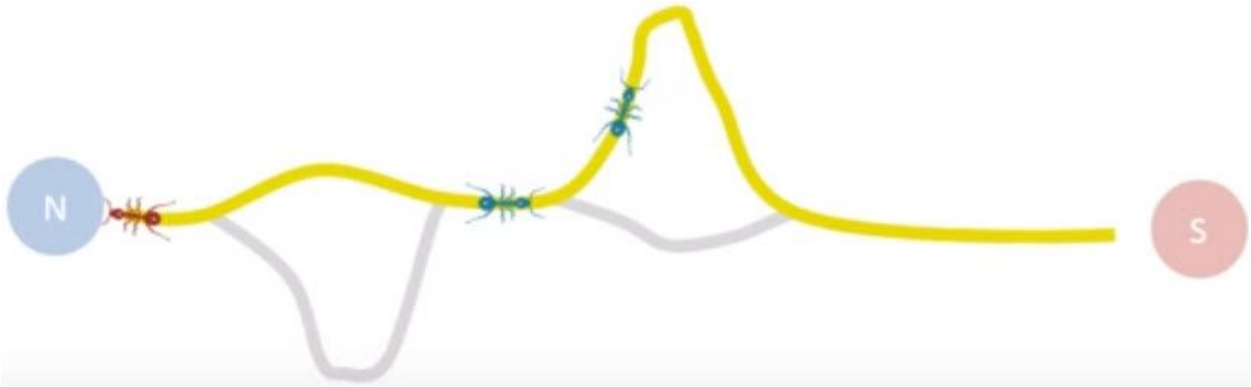

Figure 4. Ant completing a tour

This is a signal for other ants to follow the same path. Ants are *more likely* to follow the trail with the strongest scent of pheromones (meaning more ants travel that way), but because of the adventurous nature of ants they may choose another path. The pheromones on every trail will evaporate over time, thus reducing its attractive strength for other ants. This allows longer, less optimal paths to fizzle out.

Figure 5. Ants finding different routes

Because ants may find alternative paths, they will leave their own trail of pheromones. This results in ants uncovering more possible paths while reinforcing the shortest path with the most pheromones. Over time, the shortest path is uncovered.

For TSP, the idea is to have an ant start at each city. Each ant will perform a Hamiltonian cycle. While traveling, each ant will use a probability function based on the amount of pheromones on each path:

$$p_{ij}^{k}(t) = \frac{[\tau_{ij}(t)]^{\alpha} * [\eta_{ij}]^{\beta}}{\sum_{m \epsilon J_i^k} [\tau_{im}(t)]^{\alpha} * [\eta_{im}]^{\beta}}$$

p_{ij}^{k} = probability for and k to go from city i to city j

τ_{ij} = amount of pheromones between city i and j

η_{ij} = visibility of city j from i. Modeled as the inverse of the distance between 2 nodes.

α, β = exponents of the heuristic functions defining pheromone deposition on the edges. Alpha represents the importance of the trail from pheromones and Beta represents the importance of visibility.

J_i^k = set of cities adjacent to city i for ant k

Figure 6. Probability function for next city selection

When the next city is selected, the ant places pheromones on the traveled edge, but keeps track of this locally. This is done in isolation from other ants so that each ant may complete a tour without being affected by other ants; thus, the exploration of different tours is favored during an iteration of the algorithm.

Once an ant completes its tour, we update the best global path by taking the better of two: the ant's tour or the previous optimal tour. The last step involves updating the pheromones on all edges of the graph using the following formula:

$$\tau_{ij} = (1 - \rho)\tau_{ij} + \sum_{k=1}^{m} \Delta\tau_{ij}^{k}$$

ρ = evaporation rate

$\Delta\tau_{ij}^{k} = Q/L^{k}$, where Q = a constant defining importance of exploration, L is the length of the tour

Figure 7. Global pheromone update formula

Intuitively, the above formula shows that an ant that goes on a longer tour will have a lower concentration of pheromones distributed along its visited paths (Q/Lk). This way, the shorter paths are more likely to be selected during the next iteration. The algorithm

repeats for an arbitrary number of times and the best tour is returned. Below outlines the pseudocode for the algorithm:

```
RunAnts(graph, ants):
For i = 1 to max iterations
        For each ant
                Do until ant completes a tour
                        Select next unvisited city based on probability function
                Update best tour
        UpdateGlobalPheromones(graph, ants)
Return best tour

UpdateGlobalPheromones(graph, ants):
For each ant
        For i = 1 to tour length - 1
                Add pheromones to edge connecting tour[i] to tour[i+1]
```

2-Opt

2-opt is a type of local search algorithm which can find a local optimal solution for a problem. Given a feasible solution, a local search algorithm iteratively improves the current solution by searching a better solution in the pre-defined neighborhood. There are two ways to stop the iteration. One is when there is no better solution in the neighborhood. Another is when a certain number of iteration is reached.

When 2-opt is applied to solve TSP, the neighborhood is defined by removing two edges from the current tour and reconnecting the edges to render a new solution.

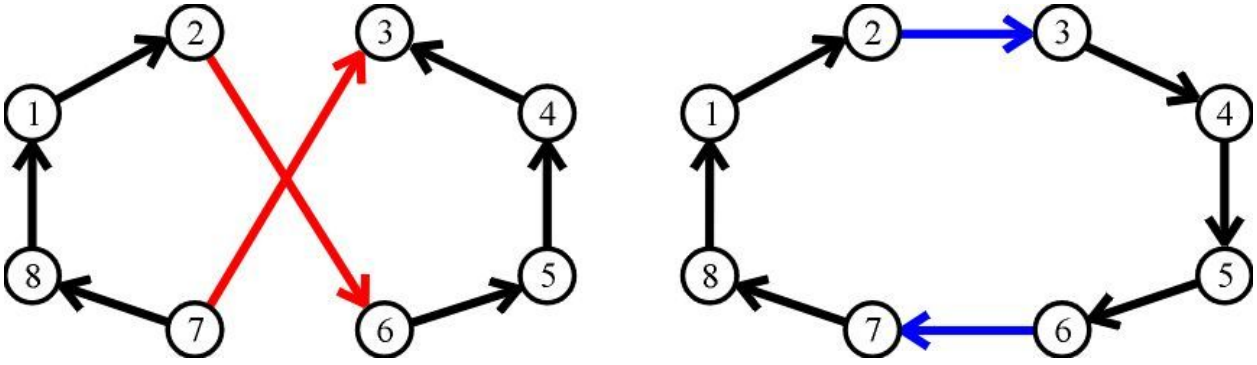

Figure 8. 2-Opt in action

If the new solution is better than the current solution, the result is kept. If not, the result is then discarded. The following is the pseudocode:

```
Let tour be the initial feasible solution
for count = 1 to Iteration times
        new_tour = 2-opt(tour)
        if isBetter(new_tour, tour)
                tour = new_tour

2-opt(current tour):
Let n be the number of cities
Let i and j be a random integer between 1 and n, where i < j
        take tour(1, i) and add them to new_tour
        take tour(i+1, j) and add them in reverse order to new_tour
        take tour(j+1, n) and add them to new_tour
return new_tour
```

As shown in the pseudocode, 2-opt requires an initial feasible solution which can be obtained by greedy or nearest neighbor algorithm. For each iteration, two edges are randomly selected, removed, and then reconnected to give a new solution for comparison.

In overall, it is easy and quick to implement 2-opt. However, since the algorithm relies on a initial approximate solution, the major disadvantage is the lack of ability to escape from the local optimal. For example, as shown in the figure below, the solution is stuck at the local optimal. It is impossible to achieve the global optimal by only switching any two edges in the graph.

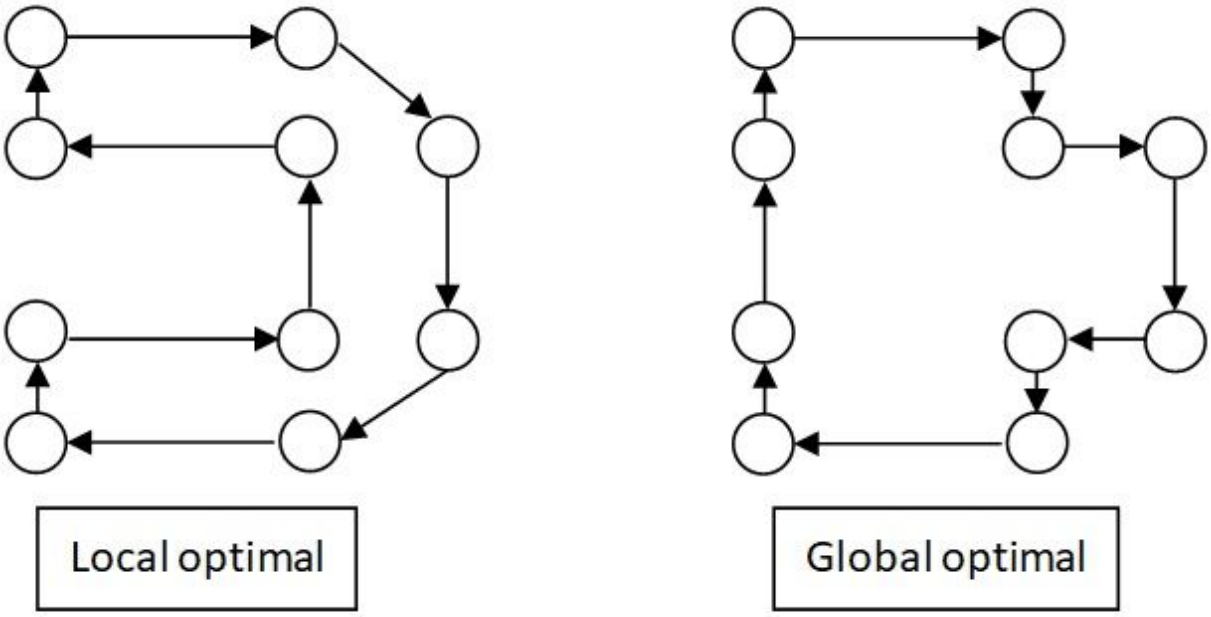

Figure 9. Impossible case for 2-Opt to find the optimal solution

In the sense of research and solving real-life problems, 2-opt is often promoted to hybrid with another algorithm such as artificial bee colony algorithm in order to maximum the performance.

Selected Algorithm and Justification

We selected one algorithm to solve the Traveling Salesman Problem: the Ant Colony Optimization method. This algorithm was by far the most interested one to research and implement for many reasons. The fact that it models the physical phenomena of ants finding the shortest cycle to a food source makes the algorithm intuitive by nature. Compared to many other algorithms other than the ones researched, it seemed to be the most creative approach to solving this problem. In fact, Ant Colony Optimization can be used in other algorithms as well such as: load balancing telecommunication networks, vehicle routing, scheduling, multiple knapsack problem, and even image processing.

One way that Ant Colony Optimization sets itself apart from other algorithms is that it is a probabilistic technique. Many other algorithms when run on the same inputs will yield the same results every time; Ant Colony Optimization instead uses a probability function to determine what the next best optimal solution is. This happens when ants are

encouraged to take new paths that they have not taken before. Therefore, the algorithm will generate a new solution every time it is re-ran.

Tour Results

Below is a table of our tour results. The run time signifies the amount of time the program was run before collecting data for the 3 minute constraint tour (using watch.py). Tour data was collected for every scenario using unlimited time and the 3 minute constraint. If a better value was found in the 3 minute trial run, that value was also used for the unlimited time best tour. Our program seems to struggle with very large N, so some of the value look extremely high because not enough iterations were performed to obtain a reasonable result.

Instance	Runtime (3 min constraint)	Best Tour (3 min constraint)	Runtime (unlimited time)	Best Tour (unlimited time)
tsp_example_1	17 sec	120569	17 sec	120569
tsp_example_2	175 sec	2925	175 sec	2925
tsp_example_3	175 sec	112723382	15 hours	3101974
test-input-1	5 sec	5520	5 sec	5520
test-input-2	37 sec	7738	37 sec	7738
test-input-3	175 sec	13474	175 sec	13474
test-input-4	175 sec	19582	175 sec	19582
test-input-5	175 sec	33983	245 sec	29646
test-input-6	175 sec	89384	132 minutes	45677
test-input-7	175 sec	2218566	4.5 hours	103097

References

- Ant colonies for the Traveling Salesman Problem
- TSP and ACO
- Ant Colony Optimization

- Local Search and the Traveling Salesman Problem: A Feature-Based Characterization of Problem Hardness
- 2-opt
- Four Heuristic Solutions to the Traveling Salesperson Problem
- 2-opt based artificial bee colony algorithm for solving traveling salesman problem
- https://towardsdatascience.com/introduction-to-genetic-algorithms-including-example-code-e396e98d8bf3 (Links to an external site.)
- http://www.theprojectspot.com/tutorial-post/applying-a-genetic-algorithm-to-the-travelling-salesman-problem/5
- https://www.geeksforgeeks.org/genetic-algorithms/