Comparison of Three Different Techniques To Solve the Traveling Salesman Problem

COMP 4106
Artificial Intelligence
Final Project

Author
Tamara Alhajj
100948027

Professor
John Oommen

April 24^{th} 2019

Contents

List of Figures ii

List of Algorithms iii

1 Introduction 1

2 Motivation 2

3 Implementation 3

4 Results 8

References 11

A Appendix A1

List of Figures

1 Example of reproduction using order 1 crossover 6
2 Non-AI on a Small Dataset 8
3 Best solution found by NN 8
4 Best solution found by GA 8
5 Best solution found by SA 8
6 Output Summary of Algorithms 9

List of Algorithms

1 Greedy Algorithm 4
2 Genetic Algorithm 5
3 Simulated Annealing Algorithm 7

1 Introduction

The domain that I will be examining is the Traveling Salesman Problem. TSP aims to find the minimized route of a graph, such that every vertex is visited exactly once, while looping back to the original vertex [Kizilateş, 2013]. This route is known as a tour. In class we discussed various AI techniques for finding such tours, which are used in the transportation industry. As promised in the project proposal, in this report I will compare the effectiveness of 2 different AI techniques: a genetic algorithm and simulated annealing. Additionally, I chose to implement greedy algorithm to solve the TSP. I have decided to include a non-AI technique to more effectively showcase the success of the AI implementations. This comparison is based on runtime savings, as well as the optimality of the found solutions.

2 Motivation

Originally, I planned to tackle a different NP-hard problem: crew scheduling. However, usable data on this topic proved incredibly difficult to find. I have chosen to solve TSP, because data in this problem domain is much easier to find.

With TSP the number of vertexs and the complexity of the problem are directly proportional; that is, the more variables to consider, then more complex the system is. In actuality it is implied that TSP is NP-hard by the NP-completeness of Hamiltonian cycle problem [Karp, 1972]. The minimized route is cannot be found in polynomial time. Therefore, to have an AI find a good enough solution, rather than say a human or brute force program, could save a lot of time and headache. To consider the entire search space of a TSP, one must search $n!$ states to find every permutation. Even a seemingly simple problem such as traveling between 5 cities has $5! = 120$ possible states. For a human, analyzing this space for a solution would be an impossible task. It proves to be difficult for a computer with significantly large values of n. In all problems of this domain, saving time, rather than space, is of upmost importance.

3 Implementation

To begin, we must discuss knowledge representation. I represent each *tour*, as an array of digits from 0 to V, for each city v in G. Every permutation of this list represents the order in which the salesman will tour these given cities. The distances are saved up in a NumPy look up matrix, by representing each city with the rows and column indices [NumPy]. This prepossessing step yields a triangular matrix distances from city a to city $b \; \forall \; a, b \in G$. This saves the some time of having to repeatedly find the distances. For my purposes, I found a website by Florida State University, which provides both the data sets and the corresponding distance matrices for TSP [Burkardt]. Thus, the step to look up distance between cities takes constant time. Lastly, I use Matplotlib, a Python 2D plotting library, to graph the solutions which are shown in Section 3 [Matplotlib, 2012].

The first algorithm I will discuss, I implemented for a non-AI solution to the TSP. This solution was the one of the earliest known solutions to this problem. The nearest neighbor (NN) algorithm is a greedy algorithm for determining a tour. The salesman starts at an arbitrary city, vertex u, then visits the nearest city, vertex v. From this new city, he repeats the process until all the cities have been visited. To complete the tour, he returns to the starting city. This algorithm is outlined below. To obtain the best result I run the algorithm over again for each vertex and repeat it for n times and return the best possible NN solution. This optimization is known as a Repetitive Nearest-Neighbor Algorithm [Asmerom].

Algorithm 1 Greedy Algorithm

```
procedure NearestNeighbour(G, u)
    tour ← [u]                                   ▷ List of city route
    visited ← {u}                                ▷ Set of visited cities
    while tour.length() < G(V).length() do
        minDistance = ∞
        NN = NULL                                ▷ Nearest Neighbour
        for v ∈ G(V) | v ≠ u do                  ▷ Search vertex set of graph
            if v not in visited then
                if distance(u, v) < minDistance then
                    NN ← v
                    minDistance = distance(u, v)
        u ← NN
        tour.append(v)
        visted.add(v)
    return tour                                  ▷ Tour of graph G, starting at u
```

The input to this algorithm is a graph G and some arbitrary starting vertex $u \in G$. The inner loop is the greedy portion of this greedy algorithm. It minimizes the distance at every vertex u to neighbour v, which runs in $O(V)$ time. The outer loop cycles through every vertex each time a new neighbour is found. This also runs in $O(V)$ time. In total, Nearest Neighbour runs in $O(V^2)$ time. If the data set used did not include the prepossessed distance matrix, it would take $O(V^2/2)$ time to prepossess myself. Note, this would be over 2 because the matrix is triangular, and would not effect the runtime for significantly large vertex sets, since $O(V^2/2) + O(V^2) = O(V)$. The performance of this TSP solution is presented in more detail in the **Results** section of this report. Although a solution is found in polynomial time, it is certainly not likely to be the optimal one. Greedy algorithms optimize locally, but not necessarily globally.

Next, I implemented a genetic algorithm to intelligently optimize the solution. A genetic algorithm is an AI technique modeled after Darwin's theory of evolution [Nicolle, 2017]. The over all algorithm is quite simple.

Algorithm 2 Genetic Algorithm

```
procedure SurivalOfTheFittest(G, size, stop)
    population = generatePopulation(size)
    for 0 to stop do                                        ▷ Generations
        selection(population)          ▷ Evaluate fitness for each individual
        reproduction(population)
        mutation(population)
    solution ← select fittest from ending population
    return solution                                         ▷ Tour of graph G
```

Let us discuss each step in more depth. We begin by generating an initial population of a specified size. This is comprised of individual tours, each a random permutation of $G(V)$. Then for the specified number of generations we loop the evolutionary process. Firstly, we we separate the wheat from the chaff *selection()*, and returning the fittest half of the population. For this step we choose the best tours, those with the shortest cycle route of G, then delete the worst half.

This insures the best *genes* get passed down. I define an individual as a unique tour, so that a gene i is the city at *tour[i]*. So to find the best genes, I find the tour order with a minimal distance. Specifically, I return the reciprocal tour distance from my fitness function because I want the lowest distance to be deemed more fit.

The next function call is *reproduction()*, which involves the mating of two chosen parents. I choose the fittest two parents to mate with a high probability and two others with a much lower probability. This optimization I have implemented, is to ensure the best are passed down to converge to an optimal solution without getting stuck with a highly similar population. When testing I found it advantageous to mate other, less fit, individuals with a low probability, because this introduces genetic diversity which produces novel solutions. With the genetic algorithm it seems that there is a balance between diversity and selective breeding which is discussed in more detail in the next section. For the actual mating I do not use the conventional crossover technique which involves picking a pivot i such that $child = parent1[0 : i] + parent2[i, n]$ where n is the total number of cities. Since there can be no repeated trips in a solution to the TSP, this conventional method would not work. Instead, I optimize for TSP using the Order

Parent 1: 8 4 7 **3 6 2 5 1** 9 0
Parent 2: 0 ~~1 2 3~~ 4 ~~5 6~~ 7 8 9
Child 1: 0 4 7 **3 6 2 5 1** 8 9

Figure 1: Example of reproduction using order 1 crossover

1 Crossover which is outline in the example below,

It is clear that no repetition can happen with this method, since repeated cities are not added to the child in the first place.

The final step of the evolutionary loop is *mutation()*. Since I do not want repeated cities, I cannot simply change one index of tour. Instead I swap two random indices. Moreover, mutation happens with an arbitrary index. I found a moderate probability, of 0.3 to 0.6, to be best for mutation to yield successful results.

Finally, the last AI algorithm I implemented was the simulated annealing algorithm. It is a randomized algorithm to approximate the optimal solution to a TSP. It is based on the methods of controlled cooling used in annealing metallurgy [Heaton Research, 2018]. We begin the loop at some incredibly high temperature then with each iteration we *cool* the temperature by decreasing the value by the cooling factor. Slow vs fast cooling will yield different results. The algorithm in simple terms is shown below.

Algorithm 3 Simulated Annealing Algorithm

```
procedure ANNEALING(G, initialTemp, coolingRate)
    solution ← pick a random initial solution
    temperature ← initialTemp
    while temperature > 1 do
        temperature *= 1 − coolingRate
        newTour ← randomSwap(solution)                    ▷ Swap 2 cities
        currentDistance ← distance(solution)
        newDistance ← distance(newTour)
        loss ← newDistance − currrentDistance
        if New distance is shorter then solution ← newTour
        else
            if e^(loss/temperature) > randomUniformProbability() then
                solution ← newTour                        ▷ Choose worse solution
    return solution                                       ▷ Tour of graph G
```

Notably, we accept a worse solution with a specific probability. This acceptance probability is based on the Gibbs distribution. This is the probability that a system will be in a certain state, as a function of that state's energy scaled by the current temperature [Adhikari, 2017]. This allows for the algorithm to randomly jump to another solution in the search space, so that we are more likely to find the global maxima. Without this probability we get something akin to the nearest neighbour algorithm. So with simulated annealing we not only finding a solution quickly, but we are also likely to find an optimal solution since we do not get stuck at local maxima.

4 Results

For the small data set of 5 cities both AI techniques found the optimal solution with a tour distance of $13km$ in under a second. However the greedy algorithm, NN, preformed sub-optimally in both solution found and runtime as shown in the figure below.

```
tamrock:AI-TravelingSalesmanProblem$ python GreedyAlgorithm.py

Greedy Algorithm: Nearest Neighbour

Total mean distance:  24.2
Total mean Runtime:  2.079010009765625e-05

Best solution path found: ['city1', 'city4', 'city3', 'city2', 'city0']
Distance Traveled: 21 km
```

Figure 2: Non-AI on a Small Dataset

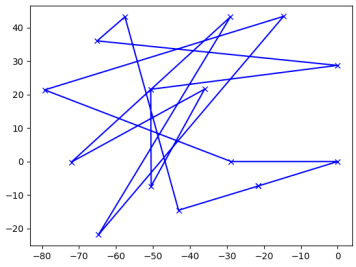

Figure 3: Best solution found by NN

As for the larger data set of 15 cities NN preformed the worst by far as shown by the tour in Figure 3. The genetic algorithm, GA, was much slower but did find a much better solution. The simulated annealing, SA, preformed excellently in regards to both solutions found and runtime. There respective tours are shown by the figures below.

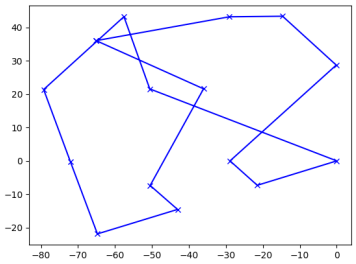

Figure 4: Best solution found by GA

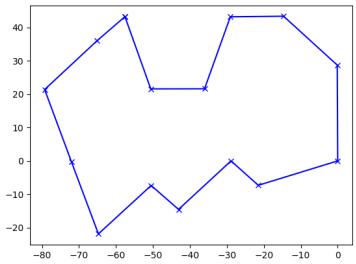

Figure 5: Best solution found by SA

The figure below shows the output, which is a detailed summary of the results. For the AI algorithms I varied the input parameters to compare different methods. Then I ran each of these respective methods of each algorithm 3 times each. For the non-AI algorithm, I simply ran NN V times to test out the algorithm starting the tour with v, as in loop though $NN(G, v) \forall v in G$. I returned the best solution returned by this as well.

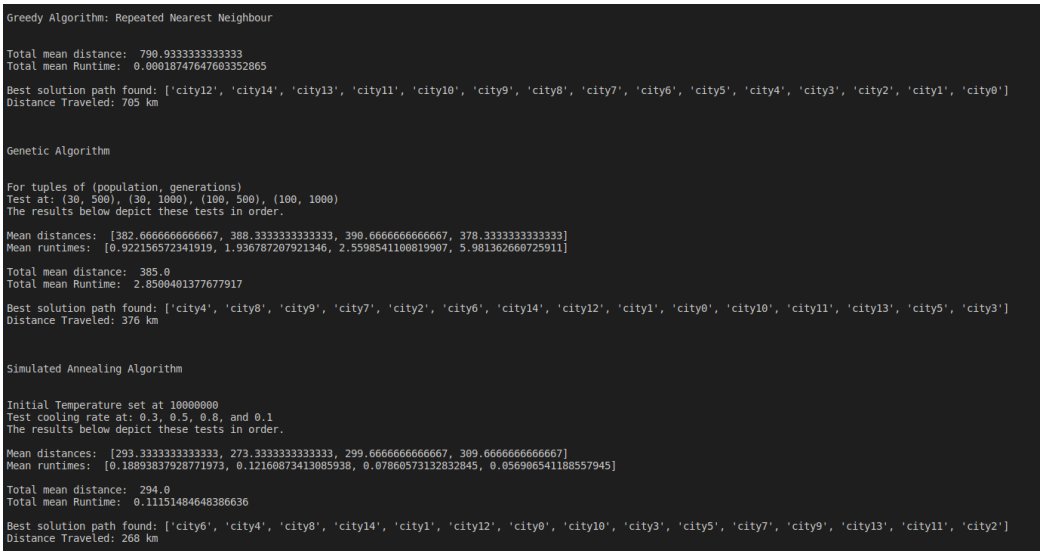

Figure 6: Output Summary of Algorithms

For NN it is clear that starting with 12 yields the best solution for this algorithm. NN finds a solution incredibly fast, but it is no where near optimal. The poor solution returned by NN is shown in the output tour in Figure 3.

For GA the runtime complexity is not great, as it takes a few seconds on average to find a solution. However, the solution found is much better than NN's by hundreds of kilometers, as shown in the summary. Input of a small initial population and moderate generations seems to trump other input methods for GA. Repeated generations for a long evolutionary cycle produces similar results with a worse runtime. So it is worth the slight sacrifice in optimality for a timely somewhat optimal result.

Finally, SA is the clear winner of the there algorithms. For all input methods it preforms at sub one second! Moreover it finds the optimal solution in

this short time, as shown by the output tour in Figure 5. It seems that although all solutions of SA are good, again a moderate input method, for $coolingrate = 0.5$, produces the best output. This is likely because it is slow enough to allow for random jumps away from local maxima, but fast enough to converge at a solution.

There is a lot of future work that could be build upon this project. Regarding the UI, it would be advantageous to add more dynamic input for G. That is one could just type in a city, and have it's longitude and latitude looked up. Regarding techniques, it would be interesting to compare more AI and non-AI techniques. I have looked into this and ant colony optimization seems to be a promising intelligent solution to tackle TSP. As for non-AI techniques, a dynamic program as well as a linear program would prove to be interesting. These solutions could also be similarly applied to the Knapsack Problem. This would be an interesting future direction for this work. If I had better equipment, it would be incredibly valuable to test this for much larger graphs. Alternatively the genetic algorithm could perhaps prove more optimal if the old generation is completely replaced by the new one. By testing different ratios of replacement, we could find GA's runtime and solution to improve as less generations are needed. It could also prove a poor approach since the genetic diversity would be diminished by some degree. The results of this are yet to be seen.

References

Adhikari, B. (2017, February 04). *The simulated annealing algorithm explained*
Retrieved from https://www.youtube.com/watch?v=eBmU1ONJ-os

Asmerom, G. A. (n.d.). *Repetitive Nearest Neighbour Algorithm.*
Retrieved from https://www.people.vcu.edu/ gasmerom/

Burkardt, J. (n.d.). *Florida State University TSP Data.* Retrieved from http://people.sc.fsu.edu/~jburkardt/datasets/tsp/tsp.html

Heaton Research. (2018, June 02).
AIFH Volume 1, Chapter 9: Traveling Salesman (TSP): Simulated Annealing. Retrieved from
https://www.heatonresearch.com/aifh/vol1/tsp_anneal.html

Karp, R. (1972). *Reducibility Among Combinatorial Problems.*
Complexity of Computer Computations in
R. E. Miller; J. W. Thatcher; J.D. Bohlinger (eds.).
New York: Plenum. pp. 85–103. doi:10.1007/978-1-4684-2001-2_9.

Matplotlib. (2012). *Pyplot.*
Retrieved from https://matplotlib.org/

Nicolle, L. (2017, August 29).
Getting started with genetic algorithms.
Retrieved from https://blog.sicara.com/

NumPy. (n.d.). *Scientific Computing Library.*
Retrieved from https://www.numpy.org/

Suwannarongsri, S., & Puangdownreong, D. (2012).
Solving Traveling Salesman Problems via Artificial Intelligent Search Techniques.
Faculty of Engineering, South-East Asia University.
Retrieved April, 2019, from http://www.worldses.org/online/2012.html

Kizilateş, G., & Nuriyeva, F. (2013).
On the Nearest Neighbor Algorithms for the Traveling Salesman Problem.
Advances in Intelligent Systems and Computing Advances in Computational

Science, Engineering and Information Technology,
111-118. doi:10.1007/978-3-319-00951-3_11

Wikipedia. (2018, October 20).
Crossover (genetic algorithm). Retrieved from
https://en.wikipedia.org/wiki/Crossover_(genetic_algorithm)

A Appendix

Simple brief User's manual. You may download the source code from my github repository, just follow this link: `https://github.com/TamaraAlhajj/AI-TravelingSalesm`

Once you unzip the repo make sure you have Python 3 installed. Also check that you have the Python library NumPy installed, which you can check with `pip freeze | grep numpy` from your terminal. Then from the working directory of the project run:

```
python GreedyAlgorithm.py && python GeneticAlgorithm.py && python
SimulatedAnnealing.py
```