Traveling Salesman and Multiple Traveling Salesman Problem [DRAFT]

Navid Kooshkjalali
Advisor: Dr. István Albert

Autumn 2020

Abstract The multiple traveling salesman problem (mTSP) is a generalization of the famous traveling salesman problem (TSP), where more than one salesman is allowed to be used in the solution. While there is a considerable body of literature surrounding the TSP and variants of it like the vehicle routing problem (VRP), such as branch and bound[1], cutting planes[2], 2-opt[3], particle swarm[4], simulated annealing[5], ant colony[6, 7], neural network[8], tabu search[9], and genetic algorithms [13, 10, 11, 12], mTSP which seems to have various real-life applications is not yet researched thoroughly. The purpose of this survey is to apply a Genetic Algorithm (GA), a stochastic derivative-free method which produces a feasible solution within reasonable time, to solve the mTSP in Scala. Genetic algorithms are evolutionary techniques of optimizing according to survival of the fittest proposed by Holland[13]. They are useful algorithms for NP-Hard problems. The genetic algorithm depends on the selection criteria and the types and proportions of crossover and mutation operators in each generation.

1 Introduction

1.1 Problem Statement

1.1.1 TSP

Given a set of nodes, let there be a salesman located at a single depot node. The remaining nodes (cities) that are to be visited are called intermediate nodes. Then the TSP is finding the tour that starts and ends at the depot, such that such that each intermediate node is visited exactly once and the total cost along that tour is minimized. This is equivalent to finding the least weight Hamiltonian cycle in a complete weighted graph. Since mTSP is an extention of TSP all mTSP solutions are also valid TSP solutions. TSP was documented by Euler in 1759, whose interest was in solving the knight's tour problem. It is the fundamental problem in the fields of computer science, engineering, operations research, discrete mathematics, graph theory, and so forth. In 1972, Richard M. Karp showed that the Hamiltonian cycle problem was NP-Complete[17], which implies the NP-Hardness of TSP.

1.1.2 mTSP

Given a set of nodes, let there be m salesmen located at a single depot node. The remaining nodes (cities) that are to be visited are called intermediate nodes. Then, the mTSP consists of finding tours for all m salesmen, who all start and end at the depot, such that each intermediate node is visited exactly once and the total cost of visiting all nodes is minimized. This is a relaxation of the VRP, where the capacity restrictions are removed. This means that solutions for the VRP are also applicable to mTSP by giving sufficiently large capacities to the salesmen. However the scope of this paper will be limited to mTSP and solutions for the VRP will not be discussed.

1.2 Exact Solutions

Exact solutions are given by exact algorithms, which always solve an optimization problem to optimality. Unless P=NP, an exact algorithm for an NP-Hard optimation problem cannot run in worst-case polynomial time. In the case of TSP the only work when the number of cities is relatively small. They begin to take progressively longer as the size of the input grows such that using them is functionally impractical.

1.3 Genetic Algorithms

Genetic Algorithms represent a computational model inspired by evolutionary sciences. Usually GAs represent optimization procedures in a binary search space. GAs are based on improving a set of solutions; a population. They then produce a successor population by mutations and recombinations of best solutions of the predecessor population. Thus at each iteration part of the current population is replaced with offspring of the fittest solutions, where the fitness of a solution has to be defined as a scalar measure. This implies that the population figuratively evolves towards an optimal solution. Genetic algorithms use three class of rules at every iteration to produce the successor population.

- **Selection Rules** select the individual solutions, called parents that whose chromosomes will contribute to the next generation.
- **Crossover Rules** combine two parents to form children to form the next generation.
- **Mutation Rules** apply random changes to individual parents to form children.

A simple pure genetic algorithm consists of the following steps.

1. Create an initial population of K chromosomes.
2. . Evaluate the fitness of each chromosome.
3. Choose $\frac{K}{2}$ parents from the current population via proportional selection.
4. Randomly select two parents to create offspring using crossover operator.
5. Apply mutation operators for minor changes in the results.
6. Repeat steps 4 and 5 until all parents are selected and mated.
7. Replace old population of chromosomes with new one.
8. Evaluate the fitness of each chromosome in the new population.
9. Terminate if the number of generations meets some upper bound, otherwise go to step 3.

2 Traveling Salesman Problem

The decision version of TSP (where given a length K, the task is to decide whether the graph has a tour of at most K) belongs to the class of NP-Complete problems. Therefore the worst case running time of the TSP increases superpolynomially (but no more than exponentially) with increase in the number of cities.

TSP can be formulated as an **integer linear program**[14, 15, 16] with several known formulations, most notable of which are **the Miller-Tucker-Zemlin(MTZ)** and **the Dantzig–Fulkerson–Johnson (DFJ) formulation**.

Traditional approaches to solving TSP are similar to any NP-Hard problem.

2.1 Exact Aglorithms

Producing exact solutions using exact algorithms (algorithms that solve for optimality of the solution) can only work in reasonable time if the number of cities is small. The most direct of these solutions (the naive, or so-called top-down approach) is to try all possible permutations and find the cheapest by brute force search. The running time of such approach lies within a polynomial factor of $O(n!)$ where n is the number of cities (Stirling's approximation: $n! \approx \sqrt{2\pi n}(\frac{n}{e})^n$). The following is a pseudocode for this approach (For implementation see **com.tsp.BruteForce.bruteForce**).

1. Let S be the set of all $(n-1)!$ possible path permutations.
2. Let T be the a mapping of each path of S to a tuple of that path and the total distance of it. $S \rightarrow T : \forall s_i \in S, \exists! t_i \in T | t_i = (s_i, \text{Cost}(s_i))$
3. Let R be the following reducer: (left: (Path, Cost), right: (Path: Cost)) $\Rightarrow$ if (left.Cost $\leq$ right.Cost) left else right
4. Reduce T by R

It can be observed that this operation does not hold any state, as in calculating the cost of a path does not depend on the cost of any sub-path previously calculated. The function does not *remember* the cost of anything, it calls the **CostMatrix.costOf(path: Path): Option[Cost]** function. So it will have to check the adjacency matric for al adjacent cities and incurr the cost unboxing optional costs every time. While this is inefficient, it gives us an oppurtinity to parallelize the task at hand. By the application concurrency and streaming, one can significantly boost the running time of this approach. For the implementation in Scala I've used the Akka library. Akka is a free and open-source toolkit and runtime simplifying the construction of concurrent and distributed applications on the JVM. Akka supports multiple programming models for concurrency, but it emphasizes actor-based concurrency, with inspiration drawn from Erlang. Akka Streams, gives us the ability to assign chunked subtasks to actors. Actors are simple agents that communicate through a messaging system. Actors will find a thread on a thread pool when given a message to do some computation, execute the computation, and if necessary reply back to to the sender of the message. Akka Streams helps us abstract away the assignment of Runnables to threads and supervision of such an actor hierarchy, without the use of low-level concurrency constructs like atomics or locks. It also provides the possibility of clustered computation on multiple JVMs and resilience and reliability features such as backpressure. All we have to do is define a stream with a Source, optionally a Flow and a Sink. The following is a pseudocode for this approach (For implementation see **com.tsp.BruteForce.bruteForceWithStreams**).

1. Let S be the set of all $(n-1)!$ possible path permutations.
2. Let $G = \{X_0, X_1, ..., X_k\}$ such that X_i are subsets of S and $X_0 \cup X_1 \cup ... \cup X_k = S$ and $\forall X_i$ where $0 \leq i < k$, $|X_i| = const$. Tuning this constant depending on the hardware can give us better performance.

3. Create a thread pool and implement a stream with the following setup:
    - Let R be the following reducer: (left: (Path, Cost), right: (Path: Cost)) $\Rightarrow$ if (left.Cost $\leq$ right.Cost) left else right
    - Let the **Source** be: G
    - Let the **Flow** be: Asynchronously and unordered, map elements of X_i to T_i such that $X_i \rightarrow T_i : \forall x_k \in X_i, \exists! t_k \in T_i | t_k = (x_k, \text{Cost}(s_k))$. Then Reduce T by R.
    - Let the **Sink** be: Reduce incoming $(Path, Cost)$ stream by R.

Despite these efforts such a solution will become impractical even when the number of cities is as small 20. A typical way to improve such brute force approaches is the usage of dynamic programming and memoization.

One of the earliest dynamic programming applications of TSP is the Held-Karp algorithm. This algorithm offers faster (but still exponential time) execution than exhaustive enumeration, with the disadvantage using a lot of space: the worst-case complexity of this algorithm is $O(2^n n^2)$ in time and $O(2^n n)$ in space. Held-Karp takes advantage of the property in TSP that ***every subpath of a path of minimum distance is itself of minimum distance***. Therefore as opposed to the **top-down** naive approach of exhaustive enumeration, a **bottom-up** approach is used such that all the intermediate information required to solve the problem is evaluated only once, and them memoized so that it's value can be later read in constant time. Starting from the smallest sub-path, and building larger paths, using the information previously evaluated. To summerize the Held-Karp algorithm saves time by eliminating redundant evaluation of information necessary to solve the problem. The following is the pseudocode for this approach (For implementation see **com.tsp.Dynamic.dynamic**)
Let V be the set of all cities and v be the depot and c be the cost function.

```
1. foreach $w \in V$ do
       $Memo(\{w\}, w) \leftarrow c(v, w)$
       $Parent(\{w\}, w) \leftarrow v$

2. for $i = 2, ..., |V|$ do
       for $S \subseteq V$ where $|S| = i$ do
           foreach $w \in S$ do
               foreach $u \in S$ do
                   $z \leftarrow Memo(S \setminus \{w\}, u) + c(u, w)$
                   if $z < Memo(S, w)$ then
                       $Memo(S, w) \leftarrow z$
                       $Parent(S, w) \leftarrow u$

3. Return the Path obtained by backtracking over cities in Parent starting at Parent(V, v)
```

2.2 Heuristic Aglorithms

A heuristic is a technique designed for solving a problem more quickly when classic methods are too slow, or for finding an approximate solution when classic methods fail to find any exact solution. This is achieved by trading optimality, completeness, accuracy, or precision for speed. The objective of a heuristic is to produce a solution in a reasonable time frame that is good enough for solving the problem at hand. This solution may not be the best of all the solutions to this problem, or it may simply approximate the exact solution. But it is still valuable because finding it does not require a prohibitively long time. Results about NP-hardness in theoretical computer science make heuristics the only viable option for a variety of complex optimization problems that need to be routinely solved in real-world applications. In the following we will briefly discuss some well known heuristic approaches to TSP.

- Nearest neighbor heuristic: A Greedy algorithm. There are sub-classes of instances for which Nearest-Neighbor consistently produces the worst tour[18].
- Clarke-Wright heuristic[19]: Identify a 'hub' vertex. Compute starting cost as cost of going through the hub. Identify 'savings' for each pair of vertices. Take shortcuts and add them to final tour, as long as no cycles are created.
- Minimum spanning tree heuristic: The tour's length is no worse than twice the optimal tour's length.The first heuristic to produce solutions within a constant of optimal. It is very easy to implement as MST can be found efficiently.
    1. First find the minimum spanning tree (using any MST algorithm).
    2. Pick any vertex to be the root of the tree.
    3. Traverse the tree in pre-order.
    4. Return the order of vertices visited in pre-order.
- Christofides heuristic[20]: First find the MST. Identify odd-degree vertices. There are an even number of such odd-degree vertices. Find a minimal matching of these odd-degree vertices and add those edges. Now all vertices have even degree. Next, find an Euler tour. Now, walk along in Euler tour, but skip visited nodes. This produces a TSP tour.
- K-Opt heuristic[3]: Start off with some tour. Find and swap K edges and their endpoints and see if the cost improves.
- Tabu search[9]: Suppose we decide to climb out of local minima. The issue is that we could immediately return to the same local minima. In tabu-search, you maintain a list of 'tabu tours', The algorithm avoids these. Each time you pick a minimum in a neighborhood, add that to the tabu list. A potential problem is a tabu-list can grow very long. Some policy should be used to remove items from the tabu list. For instance Least Recently Used, or throw out high-cost tours.

- Simulated annealing[5]: Annealing is a metallurgic process for improving the strength of metals, where they are cooled slowly because it gives molecules more time to *settle* into a globally optimal configuration. This global minimum correspons to the lowest-energy state. Simulated annealing is a modified local-search used to solve combinatorial optimization problems. Cost is associated energy and the goal is to find the lowest energy state. As mentioned above, the problem with local-search is that it might get stuck at local minimum. Simulated annealing will allow jumps to higher-cost states. Just like local-search, if a randomly-selected neighbor has lower-cost, jump to it. If the randomly-selected neighbor is of higher-cost, flip a coin to decide whether to jump to higher-cost state. If the current state is s with cost $Cost(s)$ and the randomly selected neighbor is $s^\dagger$ with cost $Cost(s^\dagger) > Cost(s)$ then the jump probability is $\exp(\frac{Cost(s^\dagger)-Cost(s)}{kT})$. The probablity of coinflip is decreased as time goes on by decreasing temperature T. As a rule of thumb if a greedy local search algorithm will do well, so will simulated annealing, so it should be experimented with first.

- Genetic Algorithm: As previously discussed, GA is a metaheuristic that applies ideas from natural selection. Genetic algorithms are used to generate 'good enough' solutions to optimization and search problems by applying operators such as mutation, crossover and selection to so called chromosomes which are valid possible solutions. For implementation see **com.tsp.GeneticAlgorithm.geneticAlgorithm**. The idea is to generate a random *Population*, usually a one dimensional sequence of chromosomes. A scalar is then assigned to each chromosome representing its *fitness*. A mating pool has to be drawn from the population using a selection strategy. The most common of which are

  - **Fitness proportionate selection**: where the fitness of each individual relative to the population is used to assign a probability of selection and therefore requires exhaustive enumeration of the population. This approach generally takes the fewest iterations until convergence and also enables a concept called **Elitism** which is means to carry out the best individuals into the next genereation, ensuring that the most successful individuals persist.
  - **Tournament selection** A set number of individuals are randomly selected from the population and the one with the highest fitness in the group is chosen as the first parent. This is repeated to chose the second parent. This requires more iterations for convergence but works well when the call to the fitness function is computationally expensive. It provides built in scaling.

  For this project, fitness proportionate selection is used since the cost function is relatively cheap and we can use elitism. Having created the mating pool, we can initiate the generation of the next population by applying operators such as mutation and crossover. Mutation serves an important function in GA, as it helps to avoid local convergence by introducing novel routes that will allow us to explore other parts of the solution space. TSP has special considerations when it comes

to mutation, namely after the mutation we must generate a valid chromosome. This can be achieved by implementing a **Swap Mutation**. This means that, with specified low probability, two cities will swap genes in our chromosomes (cities in our tours). There are numerous crossover methods, most famous of which is single point crossover where the chromosomes are split at a point and one left and right slice of the offspring come from either parent. Unfortunately in the case TSP this cannot be applied since it might yield an invalid chromosome. The crossover operator currently implemented is the so called **Ordered Crossover (OX)**. In OX we randomly select a subset of genes from one of the parent chromosomes and then fill the remainder of the chromosome from the other parent in the order in which they appear, obviously without duplication of of genes from the first parent's selected subset.
By repetition of this proccess the population can quite quickly converge into 'good' solutions. It is important to note that the process of evolving a GA is a function call which will block a thread. It is possible to evolve the multiple GAs concurrently with supervision to ensure that they traverse sections parts of the solution space. While this is not currently implemented, it will be in the near future.

3 Multiple Traveling Salesman Problem

In the case of mTSP, a set of cities are given, and all of the cities must be visited exactly once by the salesmen who all start and end at the single depot node. The number of cities is denoted by n and the number of salesman by m. The goal is to find tours for all salesmen, such that the total travelling cost (the cost of visiting all nodes) is minimized. The cost metric can be defined in terms of distance, time, etc. Some possible variations of the problem are as follows:

- Multiple depots: If there exist multiple depots with a number of salesmen located at each, a salesman can return to any depot with the restriction that the initial number of salesmen at each depot remains the same after all the travel.
- Number of salesmen: The number of salesmen in the problem can be a fixed number or a bounded variable.
- Fixed charges: If the number of salesmen is a bounded variable, usually the usage of each salesman in the solution has an associated fixed cost. In this case the minimization of this bounded variable may be involved in the optimization.
- Time windows: Certain cities must be visited in specific time periods, named as time windows. This extension of mTSP is referred to as multiple Traveling Salesman Problem with Time Windows (mTSPTW).
- Other restrictions: These additional restrictions can consist of the maximum or minimum distance or travelling duration a salesman travels, or other special constraints.

There are several exact algorithms for mTSP with relaxation of some constraints of the problem. [21, 22] provide algorithms based on the Branch and Bound algorithm. Due to the combinatorial complexity of mTSP, it is necessary to apply some heuristic in the solution, especially in real-sized applications. In [8] mTSP is converted into a single TSP and apply a modified GA to solve the problem. And in [23] a new approach to represent chromosomes is found that reduces the size of the search space by eliminating redundant solutions, this called the *two-part chromosome technique*. According to the referred paper, this representation is the most effective one so far. There are several representations of mTSP, like one chromosome technique, the two chromosome technique and the latest two-part chromosome technique. Each of the previous approaches has used only a single chromosome to represent the whole problem, although salesmen are physically separated from each other. The novel approach presented in the next chapter use multiple chromosomes to model the tours.

References

[1] G. Finke, A. Claus, and E. Gunn, “A two-commodity network flow approach to the traveling salesman problem,” vol. 41, pp. 167–178.

[2] P. Miliotis, “Using cutting planes to solve the symmetric Travelling Salesman problem,” Mathematical Programming, vol. 15, no. 1, pp. 177–188, 1978.

[3] S. Lin and B. W. Kernighan, “An effective heuristic algorithm for the traveling-salesman problem,” Operations Research, vol. 21, pp. 498–516, 1973.

[4] J. Kennedy, R. C. Eberhart, and Y. Shi, Swarm intelligence, morgan kaufmann publishers, Inc., San Francisco, CA, USA, 2001.

[5] S. Kirkpatrick and G. Toulouse, “Configuration space analysis of travelling salesman problems,” Le Journal de Physique, vol. 46, no. 8, pp. 1277–1292, 1985.

[6] M. Dorigo and L. M. Gambardella, “Ant colony system: a cooperative learning approach to the traveling salesman problem,” IEEE Transactions on Evolutionary Computation, vol. 1, no. 1, pp. 53–66, 1997.

[7] M. Dorigo, V. Maniezzo, and A. Colorni, “Ant system: optimization by a colony of cooperating agents,” IEEE Transactions on Systems, Man, and Cybernetics, Part B: Cybernetics, vol. 26, no. 1, pp. 29–41, 1996.

[8] S. Bhide, N. John, and M. R. Kabuka, ”A Boolean Neural Network Approach for the Traveling Salesman Problem,” IEEE Transactions on Computers, vol. 42, no. 10, pp. 1271–1278, 1993.

[9] F. Glover, “Artificial intelligence, heuristic frameworks and tabu search,” Managerial and Decision Economics, vol. 11, no. 5, pp. 365–375, 1990.

[10] Z. Michalewicz, Genetic Algorithms + Data Structures = Evolution Programs, Springer, New York, NY, USA, 1996.

[11] C. Moon, J. Kim, G. Choi, and Y. Seo, “An efficient genetic algorithm for the traveling salesman problem with precedence constraints,” European Journal of Operational Research, vol. 140, no. 3, pp. 606–617, 2002.

[12] J.-Y. Potvin, “Genetic algorithms for the traveling salesman problem,” Annals of Operations Research, vol. 63, pp. 339–370, 1996.

[13] J. H. Holland, Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence, University of Michigan Press, Oxford, UK, 1975.

[14] Papadimitriou, C.H.; Steiglitz, K., Combinatorial optimization: algorithms and complexity, Mineola, NY: Dover, pp.308-309. 1998.

[15] Tucker, A. W., "On Directed Graphs and Integer Programs", IBM Mathematical research Project, Princeton University 1960.

[16] Dantzig, George B. (1963), Linear Programming and Extensions, Princeton, NJ: PrincetonUP, pp. 545–7, ISBN 0-691-08000-3, sixth printing, 1974.

[17] R. M. Karp, "Reducibility Among Combinotorial Problems", R. E. Miller; J. W. Thatcher; J.D. Bohlinger (eds.). Complexity of Computer Computations. New York: Plenum. pp. 85–103. 1972

[18] G.Gutin and A.Yeo. The Greedy Algorithm for the Symmetric TSP. Algorithmic Oper. Res., Vol.2, pp.33–36. 2007

[19] G. Clarke and J. W. Wright. Scheduling of vehicles from a central depot to a number of delivery points. Op. Res., 12 ,1964, pp.568-581

[20] N. Christofides. Worst-case analysis of a new heuristic for the travelling salesman problem. Report No. 388, GSIA, Carnegie-Mellon University, Pittsburgh, PA, 1976.

[21] Laporte, G., Nobert, Y.: A Cutting Planes Algorithm for the m-Salesmen Problem. Journal of the Operational Research Society 31, 1017–1023. 1980

[22] Ali, A. I., Kennington, J. L.: The Asymmetric m-Traveling Salesmen Problem: a Dualitybased Branch-and-Bound Algorithm. Discrete Applied Mathematics 13, 259–276. 1986

[23] Carter, A.E., Ragsdale, C.T.: A New Approach to Solving the Multiple Traveling Salesperson Problem Using Genetic Algorithms. European Journal of Operational Research 175, 246–257. 2006