Algorithms for the Traveling Salesman Problem

Ada Wang, Annie Chen, Arya Wu

May 9, 2023

1 Introduction

This paper investigates the Traveling Salesman Problem (TSP). TSP is a well-known optimization problem that seeks to find the shortest possible route that visits a set of cities and returns to the starting point. It is one of the fundamental problems in the field of combinatorial optimization, with numerous practical applications and theoretical challenges.

The TSP has many real-world applications, including operations research logistics, transportation, manufacturing, circuit design, genome sequencing, and computer network design. The classification of the traveling salesman problem as an NP-hard problem indicates that finding an exact solution to the problem for large instances can be computationally infeasible using conventional algorithms. In light of these challenges, this paper explores three TSP-solving algorithms that were published at different times, with the goal of understanding the potential of each solution, and gaining insights on the future of TSP problem.

2 Problem Definition / Background

The Traveling Salesman Problem is defined as finding the shortest possible route that visits each city exactly once and returns to the starting city. The problem can be modeled as finding a Hamiltonian Cycle in a complete graph G=(V,E), where V is the set of cities and E is the set of edges with weights equal to the distances between cities.

There are several approaches to solving TSP, including approximation algorithms, heuristic algorithms, and dynamic programming algorithms. Approximation algorithms provide an approximation to the optimal solution, with a provable bound on the quality of the approximation. Heuristic algorithms are commonly used to find near-optimal solutions, but they do not guarantee optimality. Dynamic programming algorithms can find the exact optimal solution, but they are only practical for small problem instances due to their high computational complexity.

3 Algorithm(s)

In this section we will discuss three algorithms that are examples of the three different classes of algorithms for solving the TSP problem.

3.1 Christofides Approximation Algorithm

We will start with an approximation algorithm named Christofides algorithm for the problem. Suppose the optimal solution to an NP-hard problem like TSP has a value of v, an approximation algorithm guarantees that the result will end up with a value within a constant factor c of v. However, the constant approximation algorithms for TSP are also NP-hard, so we will restrict ourselves to the subset of all TSPs, the class of Metric TSPs. In the metric TSP, we have:

1) the distance metric $d(A, B) \geq 0$ for any two vertices A and B in G;
2) d(A, B) = d(B,A) since the graph is undirected by the problem definition;
3) and most importantly, the triangle inequality, which suggests that for any three vertices A,B, and C in G, $d(A, B) + d(B, C) \geq d(A, C)$

It is also important to note that the graph is complete so that the metric is defined on all pairs of vertices, ensuring that every edge exists with some value and thus follows the triangle inequality. The metric TSP is also NP-hard, but it will allow us to achieve a 3/2 approximation with the Christofides algorithm.

The method of the Christofides algorithm consists of three main steps: computing a minimum spanning tree (MST) of the graph, constructing a minimum weight perfect matching on the odd-degree vertices of the MST, and combining the MST and the minimum perfect matching to form a Hamiltonian cycle.

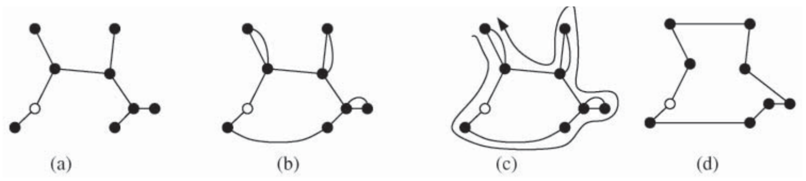

Figure 1: Illustrating the Christofides approximation algorithm: (a) a minimum spanning tree, MST, for G; (b) a minimum-cost perfect matching M on the vertices in W (the vertices in W are shown solid and the edges in M are shown as curved arcs); (c) an Eulerian circuit of G ; (d) the approximate TSP

Assume we have a complete weighted graph G for the metric TSP, where the weights on its edges follow the metric relations described above. Starting with finding the minimum spanning tree MST of G, the algorithm provides a way to connect all the vertices in polynomial time but not in a valid cycle. Since the TSP is essentially finding a minimum-weight Hamiltonian cycle in the graph, the Christofides algorithm seeks a way to construct a Hamiltonian cycle from an Eulerian circuit that starts and ends at the same vertex and visits every edge exactly once. For a graph to have an Euler circuit, all vertices must have an even degree because the number of edges entering a vertex must equal the number of edges exiting a vertex for an Euler circuit. Thus, the algorithm identifies all vertices with an odd degree in G as a set of vertices W and creates a subgraph H using the vertices in W and all edges connecting them in G. The number of vertices in W is even by the handshaking lemma [1] so that a minimum perfect matching M can be computed in H. In M, every vertex has a degree of one. Combining the edges of MST and M to form a connected multigraph H [2], the degree of all the odd vertices is increased by 1 because of M. Now every vertex in the new graph has an even degree, because of which an Eulerian circuit can be formed. Lastly, the algorithm skips repeated vertices in the Eulerian circuit to obtain a Hamiltonian circuit.

It is important to note that the last two steps do not necessarily find the same path and thus the approximation algorithm can produce several different paths, all of which are at most 1.5 times the length of the optimal solution.

To prove the approximation ratio of 1.5, we denote the shortest path the traveling salesman can take, which is also the minimum weight Hamiltonian cycle of the original graph G, as $c(H_G^*)$. Since H_G^* includes a spanning tree and T is a minimum spanning tree in G, $c(T) \leq c(H_G^*)$. The cost of Euler circuit c(C) from the steps above would equal the sum of the cost in the minimum spanning tree T and the minimum perfect matching M because the circuit traverses all the edges in the combined multigraph. Thus, we have c(C) = c(T) + c(M).

By removing the duplicates, the final graph becomes a Hamiltonian circuit C' whose cost can only be shorter than or equal to the cost of C. This can be proved by the triangle inequality: suppose that an arbitrary vertex B is being visited twice, that is, in the Euler circuit we have a path going from vertex B to A to B to C. We remove the B between A to C and the new path can only be shorter than or equal to the original path because $d(A,B) + d(B,C) \geq d(A,C)$ by the triangle inequality. Thus we have $c(C') \leq c(C)$.

With W representing the subset of odd vertices in G, let H_S^* denote the optimal Hamiltonian cycle on the subgraph H. Since the edges in G (and, hence, H) satisfy the triangle inequality, and all the edges of H are also in G, we get $c(H_S^*) \leq c(H_G^*)$. That is, skipping over vertices in the tour H_S^* will only decrease the cost by the triangle inequality. Now we number the edges of H_S^* and ignore the last edge that returns to the start vertex. The set of odd-numbered edges M1 and the set of even-numbered edges M2 are both perfect matchings since they are part of a Hamiltonian cycle. Because c(M) is the minimum perfect matching, we have $c(M) \leq c(M1)$ and $c(M) \leq c(M2))$; adding the two we get: $2c(M) \leq c(M1) + c(M2)$; $c(M) \leq 1/2(c(M1) + c(M2))$. The costs of the two sets sum up to $c(H_S^*)$, so $c(M) \leq 1/2(c(H_S^*)) \leq 1/2(c(H_G^*))$

Combining all the relations together, we get: $c(C') \leq c(C) = c(T) + c(M) \leq$

[1]The handshaking lemma states that the sum of the degrees of all the nodes is equal to twice the number of edges in the graph; since the sum of the degree of even vertices must be even, the sum of degree of odd vertices must also be even so that the sum is even.

[2]Multigraph is permitted to have multiple edges. If an edge e is in both MST and M, then create two copies of e in the combined graph.

$c(H^*_G) + 1/2(c(H^*_G) = 3/2(c(H^*_G))$. The complexity of the Christofides algorithm is dominated by the complexity of calculating the minimum weight perfect matching, which takes $O(n^3)$ where n is the number of vertices.

3.2 Very Greedy Genetic Algorithm

Next, we will explore the Very Greedy Genetic Algorithm (VGGA). The Very Greedy Genetic algorithm is a type of genetic algorithm used to solve the Traveling Salesman Problem. A genetic algorithm is a search procedure which models the biological evolutionary process of chromosome mutation to create increasingly optimal generations.

This algorithm maintains a population of strings, where each string is a permutation of integers, which represent individual cities. Each string is called a "chromosome". Chromosomes are selectively chosen to be "parental tours", and parent chromosomes are then used to create "offspring tours". They are chosen based on rank-based, linearly normalized probabilities, so that the chromosomes chosen to be parents are more likely to be shorter tours.

There are three ways the VGGA can modify a chromosome: the crossover operator, the mutation operator, and the deletion operator. A crossover operator is the process of combining two parental tours to build an offspring tour; a mutation operator modifies a single parental tour to produce an offspring tour; a deletion is simply removing tours from the population.

This algorithm uses the classic heuristic crossover operator that many genetic algorithms use. Jog, Suh and Van Gucht [4] describe the heuristic crossover operator as the following. The operator constructs an offspring tour from two parental tours by first picking a random city as the starting point for the offspring tour. Then, it compares the lengths of the edges leaving the city in each of the parent tours, and picks the shorter one. It then will continue to extend the offspring tour by picking the shorter edge between the parental tours which will continue to grow the tour. If the shorter edge introduces a cycle into the tour, then add the other edge instead. If the other edge would also introduce a cycle, then pick the shortest edge in a random selection of edges to be the new edge.

This algorithm's mutation operator randomly selects two points in a tour's permutation and reverses the segment between them to create an offspring tour. The algorithm selects tours to remove from the population by using rank-based probabilities where longer tours tend to have a lower rank and are therefore more likely to be removed. The Very Greedy Genetic algorithm always adds the shortest parental edge to a city not yet visited, if it exists, and this makes it a Greedy algorithm as well.

Each iteration of the algorithm is a generation, where a generation is complete once a fixed number of offspring tours are generated from the population. Like most algorithms, the Very Greedy Genetic algorithm has an inverse relationship between optimality and time complexity. For this algorithm, by changing the number of generations we let happen, we can directly impact how optimal the algorithm will be and how long it will take to complete.

The Very Greedy Genetic algorithm performs at least as well as other similar genetic algorithms. Typical genetic algorithms take $O(g(nm+mn+n) = O(gnm)$ time, where g is the number of generations, n is the number of cities, and m is the population size. In this paper, the researchers used a population size of 5n, so we can simplify the run time to $O(gn5n) = O(gn^2)$. They did not specify the number of generations used, so we cannot simplify the runtime further.

In the study, the researchers found that the Very Greedy Genetic algorithm gets within 1% of the optimal solution for most TSP problems. They ran the Very Greedy Genetic algorithm on 30-, 50-, 75-, 100-, and 105-city problems, and found that there wasn't correlation between problem size and optimality. Most or all trials on the 30-, 75-, and 100-city problems found optimum tours, yet only four 50-city trials and three 105-city trials found solutions that were optimal.

3.3 Deep Policy Dynamic Programming Algorithm

The Deep Policy Dynamic Programming (DPDP) is a more recent attempt at the metrics TSP problem that combines the approximate solutions learned by graph neural networks, and the near-optimal solutions guaranteed by dynamic programming algorithms. DPDP has been implemented for Euclidean TSPs modeled as a graph with n nodes, where the cost for edge (i, j) is given by c_{ij}, the Euclidean distance between nodes I and j. Similar to the algorithms discussed above, DPDP seeks to find a TSP tour that minimizes the total cost of its edges.

DPDP performs restricted dynamic programming on a sparse heatmap of promising route segments obtained by pre-processing the problem instance using a deep graph neural network (GNN). GNN is pre-trained using supervised learning on a set of example training graphs with known optimal TSP tours to learn a mapping from the TSP graph to a set of output edges that form an optimal tour.

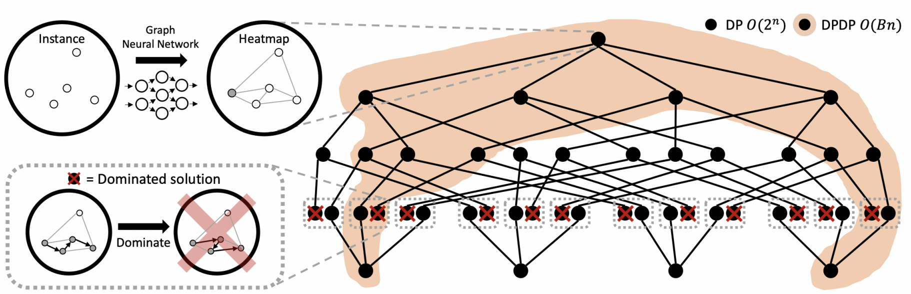

Figure 2: DPDP for the TSP. A GNN creates a (sparse) heatmap indicating promising edges, after which a tour is constructed using forward dynamic programming. In each step, at most B solutions are expanded according to the heatmap policy, restricting the size of the search space. Partial solutions are dominated by shorter (lower cost) solutions with the same DP state: the same nodes visited (marked grey) and current node (indicated by dashed rectangles).

The DPDP algorithm finds the optimal solution to the TSP problem by iteratively solving subproblems of increasing size, in this case increasing number of city nodes. In DPDP, a subproblem is defined as a partial solution that has visited a subset of the nodes in the problem instance. For each partial solution, defined by a sequence of actions a, we track the total cost over the sequence of actions leading to the state, the current node, and a set of visited nodes.

The DPDP algorithm performs a beam search over the DP state space. Beam search is a heuristic search algorithm that explores a graph by expanding the most promising node in a limited set. The DPDP algorithm begins at node 0. Initially at step t=0, the beam contains a single subproblem, which corresponds to the empty solution with $cost(a) = 0, current(a) = 0, visited(a) = \{0\}$.

At step t, the action $a_t \in \{0, ..., n1\}$ indicates the next node to visit, and is a feasible action for a partial solution $a = (a_0, ..., a_{t1})$ if (a_{t1}, a_t) is an edge in the graph and $a_t \notin visited(a)$, or, when all are visited, if $a_t = 0$ to return to the start node. When expanding the solution to $a = (a_0, ..., a_t)$, we can compute the tracked variables incrementally as: $cost(a) = cost(a) + c_{current(a),a_t}, current(a) = a_t, visited(a) = visited(a) \cup \{a_t\}$. Partial solutions are dominated by shorter (lower cost) solutions with the same DP state, represented as the same nodes visited and the same current node. $(i.e. visited(a) = visited(a), current(a) = current(a), cost(a) < cost(a))$. In each iteration, all solutions on the beam are expanded, and dominated solutions are removed for each DP state to reduce the search space.

The policy developed by the preprocessed GNN then selects the B solutions which have the highest score. The graph neural network scores the partial solutions based on the ‘'heat" of the edges it contains, plus an estimate of the ‘'heat-to-go" for the remaining nodes. $score(a) = heat(a) + potential(a)$.The "heat-to-go" initialized for each node is proportional to the node's distance from the start node, but higher weights are also given to nodes closer to the start node as the algorithm runs. This incentivizes the algorithm to explore more of the search space, while also keeping edges that enable it to return to the start node, which together helps avoid the greedy pitfall of getting stuck in local optima. The beam for the next iteration will maintain the B best solutions according to the scoring policy, which is then expanded to generate new subproblems. The DPDP algorithm terminates when all the nodes are visited, and it constructs the final solution by backtracking through the sequence of optimal actions.

DPDP is asymptotically optimal. It is guaranteed to find the optimal solution to the TSP by setting $B = 2^n n^2$, though being computationally expensive for TSP with large problem sizes. The size of B can be changed to make a trade-off between solution quality and computational cost. A smaller value of B may result in faster computation time but lower quality solutions, while a larger value of B may result in higher quality solutions but longer computation time. In this paper, the author implemented DPDP for Euclidean TSPs with 100 nodes on a commonly used test set of 10000 instances. DPDP with $B = 10K$ gives a solution with 0.009% optimality gap and a running time of $10M + 16M$. DPDP with $B = 100K$gives a solution with 0.004% optimality gap and $10M + 2H35M$ runtime.

Mathematically, the runtime of DPDP can be approximated as $O(n^2 * B)$. Prepossessing the problem instance using graph neural network takes $O(n^2)$ time. The Restricted dynamic programming runs for n iterations, each time adding an action to the solution. The beam search takes $O(n * B)$ time, where B is the beam size that's generally much larger than n. The final backtracking to construct the final solution takes $O(n^2)$. The total runtime of DPDP is then approximately $O(n^2) + n * O(n * B) = O(n^2 * B)$.

In general, the time complexity of DPDP is exponential in the size of the problem instance, so it may not be practical for very large instances, and are currently facing the difficulty of scaling to problem instances with nodes more than 100. However, the use of a neural network to predict a sparser heatmap of promising edges can help to reduce the search space and improve the efficiency of the algorithm. The sparsity of the graph can be adjusted by thresholding the heatmap and discarding low-value edges, or using a k-nearest neighbor graph.

DPDP is a flexible framework that can be applied to a variety of realistic routing problems with difficult practical constraints. Each problem should individually define the variables to track while constructing solutions, the initial solution, feasible actions to expand solutions, rules to define dominated solutions, and a scoring policy for selecting the B solutions to keep. The constructive nature of DPDP allows it to efficiently address hard constraints such as time windows, which can't be addressed by theoretical algorithms like Christofides approximation, and are also difficult for heuristics genetic algorithms.

4 Comparison

In this section, we will be comparing the three algorithms discussed in the previous section, focusing on the solution optimality, runtime, and the trade-offs between the two, as well as the practicality with problem sizes and flexibility with constraints.

4.1 Solution Optimality

Christofides algorithm has a theoretically proven approximation guarantee of 1.5 times the optimal solution as proved above, which means that the solution obtained by this algorithm is guaranteed to be within 1.5 times the optimal solution. Very Greedy Genetic algorithm does not provide any theoretical guarantee on the quality of the solution obtained. However, in practice, it can produce near-optimal solutions for TSP problems, with all problems which VGGA has performed on getting within 1% of the optimal solution. Though also without a theoretical bound, Deep Policy Dynamic Programming algorithm is asymptotically optimal with a massive B, due to the DP nature of the algorithms that guarantee optimal solutions. It has been shown to produce high-quality solutions for TSP problems in practice, but only limited to small problem sizes due to the large computational complexity.

4.2 Runtime and Trade-offs

Christofides algorithm has a polynomial runtime complexity of $O(n^3)$, which makes it efficient for small to medium-sized TSP instances. The time complexity is mainly determined by the algorithm of finding the minimum spanning tree. There is no direct trade-off between solution quality and computational cost. The algorithm does not guarantee the optimal solution but is able to achieve the solution within the factor of 1.5 far less time than an exact algorithm. Very Greedy Genetic algorithm has a runtime complexity of $O(g * n^2)$ which depends on the population size (n) and the number of generations (g). The trade-off between solution quality and runtime cost can be controlled by adjusting the number of generations. Increasing the number of generations can lead to better solutions but at the cost of increased computational time. Deep Policy Dynamic Programming algorithm has a runtime complexity of $O(n^2 * B)$.The trade-off between solution quality and computational cost is mainly controlled by the size of B. On a practical level, B is usually set to be around n^2, giving near optimal solutions with acceptable computational cost.

4.3 Efficiency and Applicability for Different Problem Sizes and Practical Constraints

Christofides algorithm is efficient for small to medium-sized TSP instances, with its performance degrading for larger TSP instances. Additionally, it is not directly applicable to TSP variants that involve additional real-life constraints as it is specifically designed for metric TSPs. Very Greedy Genetic algorithm also runs efficiently for small and medium TSP problems. The researchers did not test the Very Greedy Genetic algorithm on large problems, due to there being limited problems with known optimal solutions in TSP. Deep Policy Dynamic Programming algorithm is again practical for small TSP instances. DPDP has its advantage in its flexible framework, which enables it to handle TSP variants that involve additional constraints by exploiting certain problem structures or characteristics more effectively. For TSP with Time Windows, DPDP also outperforms the best open-source solver available, proving the power of DPDP to handle difficult hard constraints. Both may be better suited for certain types of problems or data sets where they can exploit certain problem structures or characteristics more effectively.

5 Conclusion

Recent advances in solving the TSP revolve around the using machine learning approaches, including neural networks, reinforcement learning, and genetic algorithms. While finding an exact solution in polynomial time to the TSP is not feasible due to its NP-hard nature, many algorithms offer practical solutions with varying levels of optimality, runtime, and flexibility. In this paper, we compared three specific solutions to the TSP, the Christofides algorithm, the Very Greedy Genetic algorithm, and Deep Policy Dynamic Programming algorithm. We evaluated their performance, and showed their potential to produce more promising results with further research efforts.

The Christofide's algorithm has long stood as the polynomial time algorithm with the best approximation ratio on general metric spaces. However, in 2020, Karlin, Klein, and Gharan introduced a fresh algorithm for tackling the Traveling Salesman Problem utilizing a randomly chosen tree from a carefully chosen random distribution instead of the minimum spanning tree and claimed that their approach can improve the approximation ratio to $1.5 - 10^-36$.

For Genetic algorithms, a technique called Evolutionary Divide and Conquer (EDAC) seems very promising. Evolutionary Divide and Conquer efficiently splits the problem into smaller subproblems, runs the solver on the subproblems, and then combines the subproblems to solve the original problem. Valenzuela and Jones [5] have reported promising results on TSP's of thousands of cities when applying the EDAC solution to Genetic algorithms, so EDAC could be the future of Genetic algorithms.

And then finally, while DPDP is not yet a practical alternative in general, the results are highly encouraging for further research. By addressing the scalability to larger instances, the dependency on supervised learning, and the heuristic nature of the scoring function, DPDP could significantly impact the way TSP problems get solved in the future.

In conclusion, while there's not a single best optimal algorithm for solving the TSP problem in general, the ongoing research exploration on different algorithms to TSP builds a larger set of applicable solutions. Out of this pool of solutions, we can experiment and choose the best algorithm that is suitable for the specific TSP problem at hand. Ultimately, this will enable us to better tackle the real-world applications of the TSP and drive progress in the field of combinatorial optimization.

References

[1] Goodrich, Michael T., and Roberto Tamassia. Algorithm Design and Applications. Wiley, 2015. https://canvas.projekti.info/ebooks/Algorithm%20Design %20and%20Applications%5BA4%5D.pdf

[2] Julstrom, B. A. (1995). Very greedy crossover in a genetic algorithm for the traveling salesman problem. ACM Symposium on Applied Computing. https://doi.org/10.1145/315891.316009

[3] Kool, Wouter, Herke van Hoof, Joaquim Gromicho, and Max Welling. "Deep Policy Dynamic Programming for Vehicle Routing Problems." arXiv, December 2, 2021. http://arxiv.org/abs/2102.11756.

[4] P. Jog, J.Y. Suh, and D. Van Gucht (1989). The effects of population size, heuristic crossover, and local improvement on a genetic algorithm for the traveling salesman problem. Proceedings of the Third International Conference on Genetic Algorithms, San Mateo, California: Morgan Kaufmann.

[5] C.L. Valenzuela and A.J. Jones (1993). Evolutionary divide and conquer (I): A novel genetic approach to the TSP. Evolutionary Computation, Vol.1, No.4, pp.313-333.

[6] Karlin, Anna R.; Klein, Nathan; Gharan, Shayan Oveis (2021), "A (slightly) improved approximation algorithm for metric TSP", Proceedings of the 53rd Annual ACM SIGACT Symposium on Theory of Computing, New York, NY, USA: Association for Computing Machinery, pp. 32–45,