**INFO 6205 Spring 2023 Project**

**Traveling Salesman**

**Introduction**

**Aim:** The aim of this paper is to analyze and compare various algorithms for the Traveling Salesman Problem (TSP) in order to determine the most efficient solution method. By evaluating the performance of different algorithms on a variety of TSP instances, we aim to provide insights into the strengths and weaknesses of each approach, as well as recommendations for practitioners seeking to solve similar optimization problems.

**Approach:** In this paper, we apply the Christofides algorithm to solve the Traveling Salesman Problem (TSP), which is a classic combinatorial optimization problem. To further improve the solution quality, we also investigate several meta-heuristic algorithms, including Greedy Match, Simulated Annealing, Ant Colony Optimization, Random Optimization, and Two Opt Optimization. We compare the performance of these algorithms in terms of solution quality, convergence speed, and scalability, using a set of benchmark instances with varying sizes and structures. The Greedy Match algorithm provides a simple and fast baseline for comparison, while the other algorithms explore different search strategies and perturbation techniques. Through our experiments, we aim to identify the most effective algorithm or combination of algorithms for solving TSP instances with different characteristics. To evaluate the performance of the algorithms, we use a set of widely-used metrics, including the tour length, the deviation from the optimal solution, and the running time. We also conduct a sensitivity analysis to examine the impact of different parameter settings on the algorithm performance. Specifically, for the Simulated Annealing algorithm and Ant Colony Optimization algorithm, we investigate the impact of the cooling schedule and pheromone update rule, respectively. For the Two Opt Optimization algorithm, we vary the number of iterations for local search. Our experimental results show that the choice of algorithm strongly depends on the problem instance, and no single algorithm consistently outperforms others on all instances. However, some algorithms demonstrate better performance than others on certain types of instances. We provide recommendations on the choice of algorithm based on the problem characteristics and the trade-off between solution quality and computational time. Overall, our study provides insights into the strengths and limitations of different algorithms for solving TSP, and contributes to the literature on optimization algorithms for combinatorial problems.

**Program**

**Classes**

**ChristofidesAlgorithm:** This is the main class that implements the Christofides algorithm to solve the Traveling Salesman Problem.

**PrimsAlgorithm:** This is a helper class that implements Prim's algorithm to generate a minimum spanning tree of the input graph.

**GreedyMatch:** This is another helper class that implements the greedy matching algorithm to find a perfect matching of the odd-degree vertices in the minimum spanning tree.

**MultiGraph:** This is a data structure that represents a weighted multigraph, which is used to store the intermediate results of the algorithm.

**EulerCircuitGenerator:** This is a class that generates an Euler circuit in the multigraph, which represents a feasible TSP tour.

**GraphNode:** This is a data structure that represents a node in the multigraph, which contains information about its index, degree, and adjacency list.

**EulerCircuitGenerator:** This is the main class that generates an Euler circuit in the multigraph, which represents a feasible TSP tour.

**Data structures**

**double[][] weightMatrix:** This is a 2D array that represents the weight matrix of the input graph, which contains the distances between all pairs of vertices. It is passed as a parameter to the run method of the ChristofidesAlgorithm class.

**int[] minimumSpanningTree:** This is an array that represents the minimum spanning tree of the input graph, which is generated by the PrimsAlgorithm class.

**int[][] matchGraph:** This is a 2D array that represents the weighted bipartite graph obtained by applying the greedy matching algorithm on the minimum spanning tree. It is used to build the multigraph in the MultiGraph class.

**GraphNode[] nodes:** This is an array of GraphNode objects that represent the nodes in the multigraph, which are generated by the MultiGraph class.

**int[] route:** This is an array that represents the final TSP tour generated by the algorithm, which is returned by the run method of the ChristofidesAlgorithm class.

**GraphNode[] nodes:** This is an array of GraphNode objects that represent the nodes in the multigraph, which are generated by the MultiGraph class in the ChristofidesAlgorithm.

**LinkedList<Integer> path:** This is a linked list that represents the path in the multigraph, which is generated by the algorithm.

**Vector<Integer> tmpPath:** This is a vector that stores temporary paths in the multigraph, which are used to generate the final path.

**boolean[] inPath:** This is a boolean array that keeps track of whether a node has been visited in the path or not.

**int[] route:** This is an array that represents the final TSP tour generated by the algorithm, which is returned by the generateEulerCircuit method.

**Algorithm**

**Input:** A complete undirected graph G with non-negative edge weights.

**Output:** A minimum-weight Hamiltonian cycle in G.

* Compute a minimum spanning tree T for G.
* Let O be the set of vertices in T with odd degree.
* Compute a minimum-weight perfect matching M on the vertices in O.
* Form a new graph H by taking the union of T and M.
* Find an Eulerian cycle in H starting and ending at any vertex.
* Traverse the Eulerian cycle to obtain a Hamiltonian cycle in G.

**Explanation:**

**Step 1:** To start, we compute a minimum spanning tree T for the given graph G using a standard algorithm like Prim's or Kruskal's algorithm.

**Step 2:** Next, we find the set of vertices O in T with odd degree. This is because any Hamiltonian cycle in G must have an even number of edges incident on each vertex, and so any vertex with odd degree must be part of the cycle.

**Step 3:** We compute a minimum-weight perfect matching M on the vertices in O. This can be done efficiently using the Blossom algorithm.

**Step 4:** We form a new graph H by taking the union of T and M.

**Step 5:** We find an Eulerian cycle in H starting and ending at any vertex. This can be done using Fleury's algorithm or Hierholzer's algorithm.

**Step 6:** Finally, we traverse the Eulerian cycle to obtain a Hamiltonian cycle in G.

The algorithm guarantees that the output is a Hamiltonian cycle in G that has weight no more than 1.5 times the weight of a minimum-weight Hamiltonian cycle in G.

**Invariants**

**The invariants in Christofides algorithm**

1. The input graph is undirected, connected, and has non-negative edge weights.

The algorithm constructs a minimum spanning tree (MST) of the input graph using Kruskal's algorithm or Prim's algorithm.

2. The MST is transformed into an Eulerian graph by adding edges to the MST to create an even degree for each vertex in the graph.

3. The algorithm finds an Eulerian circuit in the Eulerian graph.

4. The Eulerian circuit is transformed into a Hamiltonian circuit by shortcutting repeated vertices.

These invariants ensure that the algorithm produces a valid solution to the TSP that is at most twice the length of the optimal solution.

**The invariants in Simulated Annealing on Christofides Algorithm**

1. The input graph is undirected, connected, and has non-negative edge weights.

2. The Christofides algorithm constructs a minimum spanning tree (MST) of the input graph using Kruskal's algorithm or Prim's algorithm.

3. The MST is transformed into an Eulerian graph by adding edges to the MST to create an even degree for each vertex in the graph.

4. The SA algorithm generates an initial solution by using the Eulerian circuit obtained from the Christofides algorithm.

5. The SA algorithm iteratively improves the initial solution by randomly selecting neighboring solutions and accepting them with a probability that depends on the quality of the new solution and a temperature parameter that decreases over time.

6. The SA algorithm terminates when a stopping criterion is met, such as a maximum number of iterations or a minimum temperature.

These invariants ensure that the SA algorithm applied to the Christofides algorithm produces a valid solution to the TSP that can be further improved by exploring the solution space using a probabilistic approach. The quality of the final solution obtained depends on the temperature schedule used and the stopping criterion chosen.

**The invariants in Ant Colony Optimization on Christofides Algorithm**

1. The input graph is undirected, connected, and has non-negative edge weights.

2. The MST is transformed into an Eulerian graph by adding edges to the MST to create an even degree for each vertex in the graph.

3. The ACO algorithm initializes a colony of artificial ants that traverse the graph to construct a solution to the TSP.

4. At each step of the algorithm, each ant chooses its next vertex to visit based on a probabilistic decision rule that depends on the pheromone level on the edges and the distance between the vertices.

5. After all ants complete a tour of the graph, the pheromone level on the edges is updated based on the quality of the solutions found by the ants.

6. The ACO algorithm iteratively improves the solutions by repeating the ant tour and pheromone update steps.

7. The ACO algorithm terminates when a stopping criterion is met, such as a maximum number of iterations or a minimum change in the pheromone level.

These invariants ensure that the ACO algorithm applied to the Christofides algorithm produces a valid solution to the TSP that can be further improved by exploring the solution space using a probabilistic approach. The quality of the final solution obtained depends on the pheromone update rule, the probabilistic decision rule, and the stopping criterion chosen.

**The invariants in Random Optimization on Christofides Algorithm**

Invariants refer to properties that are preserved throughout the optimization process. In the case of Random Optimization on Christofides Algorithm, there are a few invariants that are maintained:

1. **Feasibility:** The solution generated at any iteration of the algorithm must be a feasible solution to the TSP problem. That is, the solution must satisfy the constraints of the problem, which require that each city is visited exactly once, and the tour is closed.

2. **Tour length:** The length of the tour must be maintained as an invariant. This means that the total distance traveled by the salesman between all the cities must be calculated and stored, and this value must be updated at every iteration of the algorithm.

3. **Local optimality:** At each iteration, the algorithm must generate a locally optimal solution. That is, the solution generated must be the best solution possible given the current state of the algorithm.

4. **Randomness:** The algorithm must make use of randomization in order to explore the search space and avoid getting stuck in local optima.

Overall, the goal of Random Optimization on Christofides Algorithm is to generate a globally optimal solution to the TSP problem by iteratively improving the current solution through the maintenance of these invariants.

**The invariants in 2 Opt Technique on Christofides Algorithm**

When Two Opt Optimization is applied to Christofides Algorithm, there are a few invariants that are maintained:

1. **Feasibility:** The solution generated at any iteration of the algorithm must be a feasible solution to the TSP problem. That is, the solution must satisfy the constraints of the problem, which require that each city is visited exactly once, and the tour is closed.

2. **Tour length:** The length of the tour must be maintained as an invariant. This means that the total distance traveled by the salesman between all the cities must be calculated and stored, and this value must be updated at every iteration of the algorithm.

3. **Local optimality:** At each iteration, the algorithm must generate a locally optimal solution. That is, the solution generated must be the best solution possible given the current state of the algorithm.

4. **Two-Opt optimality:** The algorithm must maintain the Two-Opt optimality invariant, which means that the tour generated by the algorithm must not contain any edges that could be swapped to create a shorter tour. In other words, the algorithm must ensure that it has explored all possible two-edge swaps and that the current solution is locally optimal with respect to these swaps.

5. **Convergence:** The algorithm must converge to a globally optimal solution after a finite number of iterations. This is ensured by the Two-Opt optimality invariant, which guarantees that the algorithm will eventually reach a globally optimal solution.

Overall, the goal of Two Opt Optimization on Christofides Algorithm is to generate a globally optimal solution to the TSP problem by iteratively improving the current solution through the maintenance of these invariants.