Informatics 2 - Introduction to Algorithms and Data Structures

Coursework 3: Heuristics for the Travelling Salesman Problem

Zornitsa Asanska
s1816900

March 2020

1 Part C - Algorithm

For this section of the assignment, I implemented two separate algorithms: the *Christofides algorithm* and a *Greedy algorithm* (different from the one implemented in part B). I will first provide my reasoning for choosing each of the algorithms and then I will proceed to explain and analyse both algorithms.

The *Christofides Algorithms* (CA) is a famous approximation algorithm. The extended implementation of the algorithm guarantees a solution to the metric case of TSP with a factor of 3/2 of the optimal solution length[1]. I have provided a 'more relaxed' implementation which performs within a ratio of 2. The tests provided in Part D show that in most cases the CA performs worse (returns a permutation with a longer tour cost) that the two Greedy algorithms. However, I chose this algorithm because the implementation involves implementing 'subalgorithms' such as finding a Minimum Spanning Tree and an Euler Circuit. Those algorithms were covered in the course of the academic year in DMMR and IADS and I saw the Christofides Algorithm as a good opportunity to make use of the studied material.

The second algorithm I implemented is a *Greedy algorithm* (`Greedy2()`and `Greedy2Ext()`) The provided tests in Part D demonstrate that this Greedy algorithm performs better than the other ones. On some occasions with a small set of nodes (10-15) it finds the optimal solution (the optimal solution was calculated with the Held-Karp Dynamic Programming algorithm).

In addition to those two algorithms, I implemented the Held-Kerp Dynamic Programming Algorithm. This algorithm runs in exponential time and does not comply with the requirements of the coursework. However, I used it when conducting tests on small sets of nodes to see how well the other algorithms perform and to see how close they get to the optimal solution.

1.1 Christofides Algorithm

The Christofides Algorithm can be broken down to 6 steps implemented with helper functions:

1. **Find a Minimum Spanning Tree** (`_primMST()`)
   I implemented *Prim's Algorithm*, which was covered in DMMR. It includes a helper function `_minKey(key,V)`,which returns the next node to be added to the tree. (Running time: $\mathcal{O}(n^2)$)
2. **Find the set of nodes of odd degree** (`_oddDegree(T)`)
   The function takes a tree (in the case of the algorithm that is the MST) as a tuple of edges. It calculates the degree of each node and returns a list of the nodes with odd degree. The Hand-Shaking Lemma guarantees that there is an even number of odd-degree nodes. (Worst-case Running time: $\mathcal{O}(n^2)$)
3. **Find a minimum-weight perfect matching in the complete graph of odd-degree nodes** (`_matching(V)`)
   I used a heuristic greedy approach to find a perfect matching. It removes a node of V and finds and removes a second node closest to the first one. The two form an edge in the matching. The procedure is iterated until V is empty. The function returns an array of tuples representing edges. (Running time: $\mathcal{O}(n^2)$)

4. **Combine the edges of the Minimum Spanning Tree and the Perfect Matching Graph into a Multigraph**
   Done in $\mathcal{O}(1)$ - constant time. The two graphs (presented as arrays) are added together
5. **Find an Euler Circuit in the Multigraph** (the previous steps guarantee that an Euler Circuit exists) (`_eulerCircuit(T)`) ($\mathcal{O}(n^2)$)
   Fleury's Algorithm for finding an Euler Circuit Helper functions:
   - `_circuit()`
     Recursive algorithm for constructing an Euler Circuit ($\mathcal{O}(n^2)$)
   - `_isValidEdge(edge)`
     Checks if an edge is a valid continuation of the path constructed so far. (Running time $\mathcal{O}(n)$)
   - `_DFSCount(v,visited)`
     A Depth-First Search Algorithm for finding the number of unvisited nodes reachable from the node v. The function has a running time: $\mathcal{O}(n+E)$ where E is the number of edges. Since the edges in the Multigraph are at most $(n-1)+\lceil \frac{n}{2} \rceil$. The total running time is linear $\mathcal{O}(n)$.
   - `_addEdge(edge)`and `_removeEdge(edge)`
     Add and Remove an edge (tuple) from the dictionary of neighbours ($\mathcal{O}(1)$)
6. Build a Hamiltonian circuit from the Euler circuit by shortcutting (skipping the repeated nodes) (`_hamiltonianCircuit()`)
   Removing the repeated nodes in the Euler Circuit and the final permutation is assigned to `self.perm.` ($\mathcal{O}(n)$)

The above-mentioned steps are combined in the `_Christofides()` function, which runs in quadratic time.

1.2 Second Greedy Algorithm

This algorithm (`Greedy2(start_node=0)`) builds an optimal path by placing one node at a time by starting from node 0. It calculates the partial value of the path generated so far using the helper function `_partialTourValue(path)` ($\mathcal{O}(n)$). Then the next node is placed at the position which gives the lowest tour cost. This is a greedy approach which makes the assumption that every subpath of the permutation is of minimal cost.
Complexity analysis: At each comparison the work done is linear (calculating the partial tour value). The total number of comparisons for constructing the whole path is:

$$2+3+...+(n-1)+n=\sum_{i=2}^{n} i=\frac{n(n+1)}{2}-1 \in \mathcal{O}(n^2)$$

Each comparison can be done in linear time (calculating the partial tour value). Then the algorithm has complexity $\mathcal{O}(n^3)$.
The function `Greedy2Ext()`runs `Greedy2(start_node=0)`with each `self.n`nodes as starting node and assigns the permutation with the smallest tour cost to `self.perm`. This results in a time complexity $\mathcal{O}(n^4)$.

1.3 Held-Karp Algorithm

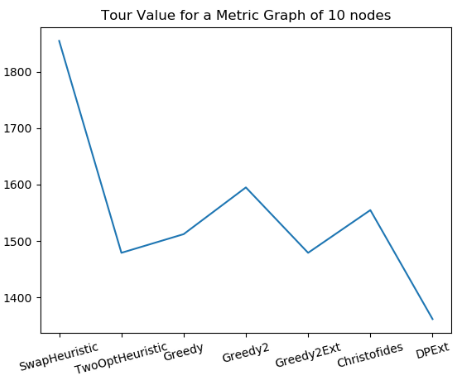
The Held-Karp Algoprithm is a bottom-up dynnamic programming algorithm for finding the solution to the TSP problem [1]. The algorithm assumes a starting node 0. It performs better than a brute-force naive approach but still remains exponential with a running time $\mathcal{O}(2^{\mathrm{n}}n^2)$. The extended implementation `DPAlgorithmExt(visited,pos,start)`finds the overall optimal tour cost by running the algorithm with each node as a starting node ($\mathcal{O}(2^{\mathrm{n}}n^3)$).

2 Part D - Experiments

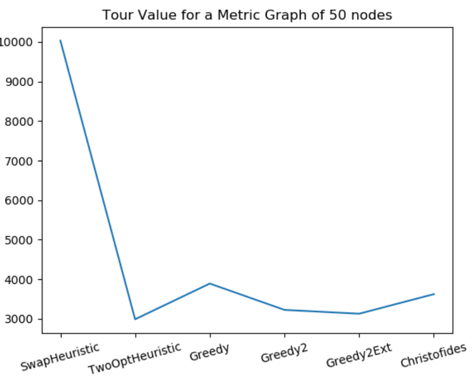
For this segment, two functions were used for generating graphs: `NonMetricGraph(nodes,lim)` and `XYnodes(n,Xlim,Ylim)`. The experiments that I conducted use two additional classes provided in the `tests.py` file. The Christofides algorithm is designed for the metric case of the TSP and I have focused the analysis on metric graphs. The graphs are generated randomly and the performance of each algorithm compared to the others differs depending on the generated graph. The provided figures and data reflect the most commonly arising patterns.
For all tests on graphs of over 15 nodes the DP algorithm is not used due to its exponential time and space complexity.

2.1 Algorithm Performance Comparison

On small sets of nodes, the Second Greedy Algorithm comes closer to the optimal solution (calculated by the Held-Karp DP algorithm `DPExt()`) compared to the Chirstofides. On larger sets of nodes the `Greedy2Ext()`finds a solution with a lower cost - on average the tour value of `Greedy2Ext()`is 93% of the tour cost found by `Christofides()`.



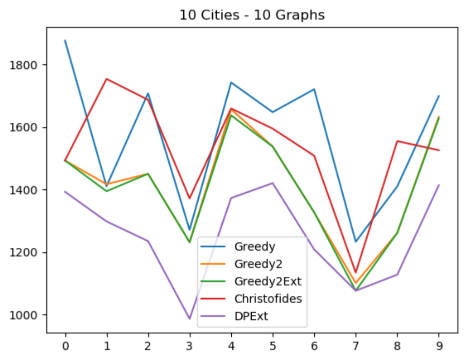
(a) 10 nodes



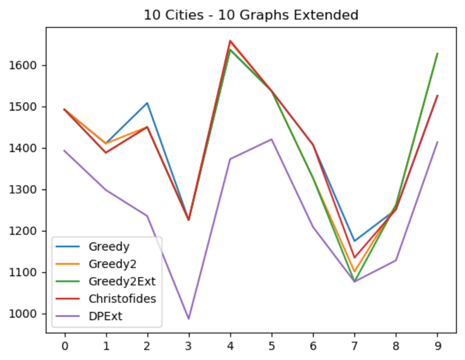
(b) 50 nodes

Figure 1

The tests on multiple graphs give a broader picture of the performance of the algorithms in the average case. On average, on graphs with a small number of nodes ($\leq$15) `Greedy2Ext()` performs within a factor of 1.13 of the optimal solution, compared to the `Christofides()` - with an average ratio of 1.26.



(a) **Before** swapHeuristic() and TwoOptHeuristc()



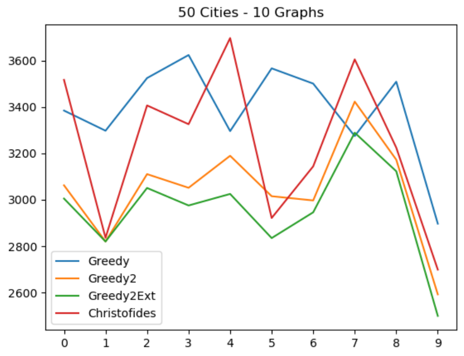
(b) **After** swapHeuristic() and TwoOptHeuristic()

Figure 2

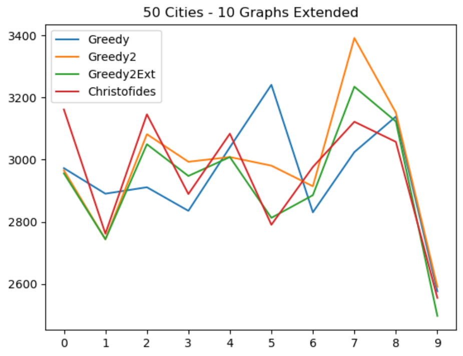
Figures 2 and 3 show how each alogrithm performs for 10 graphs (of either 10 or 50 nodes) and how running `swapHeuristic()` and `TwoOptHeuristic()` improve the optimal solution achieved by the Greedy algorithms and Christofides. On smaller graphs the optimal tour cost of the algorithms becomes almost identical after running `swapHeuristic()` and `TwoOptHeuristic()`. This can be seen in Table 1. The table presents the mean ratio of the tour cost of each algorithm to the optimal solution (found with the Held-Karp DP algorithm `DPExt()`) calculated for 10 graphs with 10 nodes. The four algorithms come almost equally close to the solution of the TSP problem. It is interesting to see that in this case the `_Greedy2Ext()` is not improved. When it is improved on a different set of graphs, the improvement is smaller compared to the improvement of the other algorithms.

Table 1: Mean Ratio to Optimal Solution for 10 Graphs of 10 Nodes

	Greedy()	Greedy2()	Greedy2Ext()	Christofides()
Before swapHeuristic() and TwoOptHeuristic()	1.264	1.152	1.117	1.2360
After swapHeuristic() and TwoOptHeuristic()	1.122	1.119	1.117	1.117



(a) **Before** swapHeuristic() and TwoOptHeurisitc()



(b) **After** swapHeuristic() and TwoOptHeurisitc()

Figure 3

3 Further Improvement

- The implementation of the *Christofides Algorithm* does not guarantee that the perfect matching is of minimum-weight. *Blossom's Algorithm* for weighted graphs can be implemented to find the minimum-weight matching. Blossom's Algorithm has a running time of $\mathcal{O}(n^3)$, which will bring the upper bound of the Christofides Algorithm to $\mathcal{O}(n^3)$
- The Held-Karp Algorithm only gives the tourValue of the optimal permutation but not the permutation itself. The implementation can be amended to return the exact sequence.
- More tests can be conducted on non-metric graphs.

References

[1] Christian Nilsson. Heuristics for the traveling salesman problem. 01 2003.