Traveling Salesman Problem with Job-times (TSPJ)

Project Report

Zihan Ma 20238181

1 Background - Traveling Salesman Problem (TSP)

Traveling Salesman Problem (TSP) is a classical optimization problem that seeks the shortest possible route to visit a set of nodes exactly once and return to the original node, which can be defined mathematically as follows Gavish and Graves [1978].

$$\begin{array}{ll} \min & \sum_{i=0}^{n} \sum_{j=0}^{n} c_{ij} X_{ij}, \\ \text{s.t.} & \sum_{j=0}^{n} X_{ij} = 1 \quad \forall i = 0, \ldots, n, \\ & \sum_{i=0}^{n} X_{ij} = 1 \quad \forall j = 0, \ldots, n, \\ & \sum_{j=0}^{n} Y_{ij} - \sum_{j=0}^{n} Y_{ji} = 1 \quad \forall i = 1, \ldots, n \quad i \neq j, \\ & Y_{ij} \leqslant n X_{ij} \quad \forall i = 1, \ldots, n \quad \forall j = 0, \ldots, n \quad i \neq j, \\ & X_{ij} \in \{0, 1\} \quad Y_{ij} \geqslant 0 \quad \forall i \quad \forall j, \end{array}$$

where X_{ij} is to determine whether or not to travel directly from node i to node j, c_{ij} is the corresponding cost, and Y_{ij} represents the sequence between nodes.

The Nearest Neighbor **(NN)** algorithm is a widely employed heuristic approach for obtaining initial solutions to TSP. This algorithm operates by selecting an arbitrary node as the starting point and then iteratively visiting the nearest unvisited node. This process continues until all nodes have been traversed, ultimately yielding a feasible tour for the TSP. Then the tour can be improved by the **2-opt** algorithm, which evaluates the tour lengths that result from swapping all combinations of node pairs and accepts swaps that improve the objective.

2 Traveling Salesman Problem with Job-times (TSPJ)

2.1 Introduction

The Traveling Salesman Problem with Job-times (TSPJ) is a hybrid problem that inherits properties from the Traveling Salesman Problem (TSP) and the Scheduling Problem. The paper Mosayebi et al. [2021] defines TSPJ as follows: There is one depot, n fixed processing nodes, and n jobs, where each node (except the depot) is assigned exactly one job. Each job consists of a single task, and the execution time of each job varies depending on the node at which it is performed. A traveler must visit all nodes exactly once, assign one job to each node, and then return to the depot. After assigning a job, the traveler immediately moves to the next node, allowing the job to proceed autonomously.

The goal of TSPJ is to find the sequence of the nodes to visit along with the assignment of the jobs to the nodes to minimize the maximum completion time of all jobs and the time to return to the depot.

2.2 Mathematical Model

Let Z_{ik} be an indicator variable denoting the assignment of job k to node i, $C_{\max}$ be the maximum completion time of the jobs across all nodes, TS_i be the actual start time of the job at node i, tt_{ij} be the travel time from node i to node j, t_{ik} be the processing time of job k at node i, TSPJ can be defined mathematically as follows:

$$
\begin{aligned}
\text{min} \quad & C_{\max}, \\
\text{s.t.} \quad & C_{\max} \geqslant TS_i + \textstyle\sum_{k=1}^{n} Z_{ik} t_{ik} \quad \forall i = 1, \ldots, n, \\
& C_{\max} \geqslant TS_i + X_{i0} tt_{i0} \quad \forall i = 1, \ldots, n, \\
& \textstyle\sum_{i=1}^{n} Z_{ik} = 1 \quad \forall k = 1, \ldots, n, \\
& \textstyle\sum_{k=1}^{n} Z_{ik} = 1 \quad \forall i = 1, \ldots, n, \\
& \textstyle\sum_{j=0}^{n} X_{ij} = 1 \quad \forall i = 0, \ldots, n \quad i \neq j, \\
& \textstyle\sum_{i=0}^{n} X_{ij} = 1 \quad \forall j = 0, \ldots, n \quad i \neq j, \\
& \textstyle\sum_{j=0}^{n} Y_{ij} - \sum_{j=0}^{n} Y_{ji} = 1 \quad \forall i = 1, \ldots, n \quad i \neq j, \\
& Y_{ij} \leqslant n X_{ij} \quad \forall i = 1, \ldots, n \quad \forall j = 0, \ldots, n \quad i \neq j, \\
& TS_i + tt_{ij} - (1 - X_{ij})\,\mathbf{M} \leqslant TS_j \quad \forall i = 0, \ldots, n \quad \forall j = 1, \ldots, n \quad i \neq j, \\
& X_{ij}, Z_{ik} \in \{0, 1\} \quad TS_i, Y_{ij} \geqslant 0 \quad \forall i \quad \forall j,
\end{aligned}
$$

where X_{ij} and Y_{ij} keep the same as in TSP, and $\mathbf{M}$ is a fixed large number which can be defined as $\mathbf{M} = \sum_i (\max_k \{t_{ik}\} + \max_j \{tt_{ij}\})$. Apart from the constraints already defined in the TSP, the third and fourth constraints in the above mathematical formulation force the model to assign only one job to each node and vice versa. The penultimate constraint ensures that when traveling from node i to node j, the starting time of the job at node j must be greater than or equal to the starting time of the job at node i plus the travel time between nodes i and j.

2.3 Heuristics

2.3.1 TSPJ-NN

The **TSPJ-NN Algorithm** (Algorithm 1) begins by selecting any node as the starting point and constructing a tour by visiting the nearest neighboring nodes until a complete tour is formed. Once the tour is completed, the algorithm resumes from the last node along the tour and assigns the job with the minimum processing time at that node to that node.

2.3.2 TSPJ-2-opt

The **TSPJ-2-opt Algorithm** (Algorithm 2) starts from the first node of the existing TSPJ-NN tour and evaluates the maximum tour completion time resulting from swapping node pairs. If an improvement is possible, the swap is accepted, and the algorithm starts from the last node along this tour and assigns the job with minimum processing time at the node to it.

2.3.3 Reverse Assignments

The **Reverse Assignments Algorithm** (Algorithm 3) moves backward through the existing tour and assigns the next un-assigned job with minimum processing time to the node till all jobs are assigned.

2.3.4 Local Search Improvement (LSI)

The **Local Search Improvement (LSI) Algorithm** (Algorithm 4) improves TSPJ solutions by first considering jobs swapping while keeping the node sequences fixed and evaluating node swapping for nodes that precede the node that results in the maximum completion time. The LSI Algorithm is the key algorithm in the paper.

Algorithm 1 TSPJ-NN

```
 1: Input: Read Travel Time table (TT) as matrix $t_{ij}$
 2: Input: Read Job-time table (TJ) as matrix $t_{ik}$
 3: Output: Best_sequence
 4: Define $C$: List of completion time of all jobs in the sequence
 5: $C[0] \leftarrow 0$
 6: for $s$ in range(1, Len(TT)) do
 7:     node_pool $\leftarrow TT$
 8:     job_pool $\leftarrow I$
 9:     sequence[0] $= s$
10:     for $i$ in range(1, Len(TT)) do
11:         sequence[$i$] $\leftarrow \arg\min(tt_{i-1,j} \quad \forall j = 1, \ldots, n)$
12:         Remove selected node from node_pool
13:     end for
14:     for $i$ in range(Len(TT), 1, -1) do
15:         job[$i$] $\leftarrow \arg\min(t_{ik} \quad \forall k = 1, \ldots, K)$
16:         Remove selected job from job_pool
17:     end for
18:     new-sequence $\leftarrow$ sequence + job
19:     Calculate $C_{\text{new-sequence}}$
20:     if $C_{\max} > C_{\text{new-sequence}}$ then
21:         Best_sequence $\leftarrow$ new-sequence
22:     end if
23: end for
24: return Best_sequence
```

Algorithm 2 TSPJ-2-opt

```
1: Input: Read Travel Time table (TT) as matrix $t_{ij}$
2: Input: Read Job-time table (TJ) as matrix $t_{ik}$
3: Input: Read sequence
4: Output: Best_sequence
5: Define $C$: List of completion time of all jobs in the sequence
6: Best_sequence ← sequence
7: Start_again
8: while there is improvement do
9:    for $i$ in range(1, Len(TT) - 1) do
10:       for $j$ in range($i$ + 1, Len(TT) + 1) do
11:          if swapping is profitable then
12:             sequence ← Swap nodes
13:             job_pool ← $T$
14:             for $i$ in range(Len(TT), 1, -1) do
15:                job[$i$] ← $\arg\min(t_{ik} \quad \forall k = 1, \ldots, K)$
16:                Remove selected job from job-pool
17:             end for
18:             Update jobs in sequence
19:             Best_sequence ← sequence
20:             Goto Start_again
21:          end if
22:       end for
23:    end for
24: end while
25: Return Best_sequence
```

Algorithm 3 Reverse Assignments

```
1: Input: Read sequence as list $X$
2: Input: Read Job-time table ($T$) as matrix $t_{xk}$
3: Output: Reads job list as List $K$
4: $n$ ← Len($X$)
5: $X'$ ← $X$
6: $K$ ← Zeros($n$)
7: while $X'$ do
8:    $K[-1]$ ← argmin $\{t_{xk}, k \in \tau(x)\}$
9:    for $j$ in range (1,n) do
10:       $t_{jk[-1]}$ ← $\infty$
11:    end for
12:    $X'$ ← $X'[: -1]$
13: end while
14: return $K$
```

Algorithm 4 Local Search Improvement heuristic

```
 1: Input: Read Travel Time table (TT) as matrix $t_{ij}$
 2: Input: Read Job-time table (TJ) as matrix $t_{ik}$
 3: Input: Read sequence
 4: Output: Best_sequence
 5: Start_again
 6: while there is improvement do
 7:    if job-time of $C_{\max}$ is not minimum job-time for node then
 8:       Assign minimum job-time to $C_{\max}$-node
 9:       Reassign other jobs
10:       calculate $C_{\text{new\_sequence}}$
11:       if $C_{\max} > C_{\text{new\_sequence}}$ then
12:          Update Best_sequence
13:          Goto Start_again
14:       end if
15:    end if
16:    for $S$ in range ($C_{\max}$-node + 1, Len(TT)) do
17:       if swapping jobs conditions are satisfied then
18:          Swap job between $C_{\max}$-node and $S$
19:          Update Best_sequence
20:          Goto Start_again
21:       end if
22:    end for
23:    $C[0] = 0$
24:    for $S$ in range ($C_{\max}$-node $-1, 1, -1$) do
25:       if reverse swapping conditions are satisfied then
26:          Swap $C_{\max}$-node and $S$
27:          Update Best_sequence
28:          Goto Start_again
29:       end if
30:    end for
31: end while
32: Return Best_sequence
```

2.4 Procedures

The paper considers four procedures for the heuristics discussed above.

- Procedure 1 first constructs a tour considering the job assignments to nodes using TSPJ-NN (Algorithm 1) followed by TSPJ-2-opt (Algorithm 2) to improve the solution.
- Procedure 2 first develops a tour without considering job assignments by NN and 2-opt algorithm. Then jobs are assigned to the nodes by Reverse Assignments (Algorithm 3). Subsequently, the solution is improved using Local Search Improvements (Algorithm 4).
- Procedure 3 starts with an initial tour constructed using TSPJ-NN (Algorithm 1). Jobs and nodes are then swapped using Local Search Improvements (Algorithm 4). Finally, TSPJ-2-opt (Algorithm 2) is used to improve the solution.
- Procedure 4 first uses TSPJ-NN (Algorithm 1) but starts the tour by designating the node that has the greatest distance from the depot as the last node of the tour and selecting the remaining sequence of nodes on the tour. After the tour is determined, node and job swapping is performed using Local Search Improvements (Algorithm 4). Finally, TSPJ-2-opt (Algorithm 2) is used to improve the solution.

3 Implementation

3.1 Structure

```
data/
|--Small_problems/
|--TSPLIB-J/
src/
|--__init__.py
|--cal_time.py // To calculate the maximum completion time
|--LSI.py // Local Search Improvements (Algorithm 4)
|--NN.py // TSPJ-NN (Algorithm 1)
|--re_assign.py // Reverse Assignments (Algorithm 3)
|--two_opt.py // TSPJ-2-opt (Algorithm 2)
|--main.py // To load the dataset and execute each procedure
```

3.2 Data

The data are provided in `https://github.com/TSPJLIB/TSPJ`, including two kinds of dataset:

- TSPJLIB: Based on TSPLIB data for TSP (gr17, gr21, gr24, fri26, bays29, gr48, eli51, berlin52, eli76, and eli101), with job times added as random numbers between 50% to 80% of the optimal TSP-tour units known for each problem.
- Small/Medium/Large Problem Datasets: 300 random problems were generated and categorized by the number of jobs into small, medium, and large problem categories. Each category contains 100 problems. The small, medium, and large problems have a random number of nodes between 40 to 50, 400 to 500, and 1000 to 1200, respectively. Additionally, the distances between nodes for the small, medium, and large problems range between 10 to 50, 50 to 200, and 200 to 500 units, respectively.

For each problem in the dataset, I use two CSV files to store the travel time (tt_{ij}) and job-processing time (t_{ik}) respectively, as defined previously.

4 Experiments

In this section, I first introduce my experimental results and provide some discussions about them. The experimental environment for my experiments is set up on a machine equipped with an Apple M2 CPU and 16 GB of RAM, running macOS 14.2. The programming language used is Python, utilizing the Numpy and Pandas packages. For the Small/Medium/Large problem datasets, I am unable to include the results here because even the small problem dataset takes nearly two minutes to process. Based on the data trends in the paper and the results obtained using Julia, I anticipate that the medium and large problem datasets would require significantly more time to compute.

The experiments using Julia are set up on a machine equipped with 6 Intel Core i5-9600K CPU @ 3.70GHz and 16GB of RAM.

4.1 Solution Performance

To assess the performance of my implementation, experiments were conducted using the TSPJLIB dataset, as detailed in Section 3.2. Table 1 comprises 10 columns. The first column lists the names of each test problem, and the second column presents the GAMS solutions referenced in the paper. Subsequent columns display the C_{max} results from my implementation of four procedures, followed by the respective gaps between my solutions and the GAMS solutions.

	GAMS	**Procedure 1**		**Procedure 2**		**Procedure 3**		**Procedure 4**	
	C_{max}	C_{max}	%	C_{max}	%	C_{max}	%	C_{max}	%
`gr48-J`	7288.0	7560.0	3.6	8395.0	13.2	7560.0	3.6	7589.0	4.0
`berlin52-J`	11087.0	11251.2	1.5	12935.6	14.3	11251.2	1.5	11790.5	6.0
`gr24-J`	1806.0	1858.0	2.8	2022.0	10.7	1858.0	2.8	1858.0	2.8
`eli51-J`	630.0	648.7	2.9	677.6	7.0	648.7	2.9	634.3	0.7
`gr21-J`	7788.0	7959.0	2.1	10102.0	22.9	7959.0	2.1	7959.0	2.1
`bays29-J`	2937.0	2963.0	0.9	3487.0	15.8	2963.0	0.9	2980.0	1.4
`eli76-J`	802.0	821.1	2.3	1000.6	19.8	821.1	2.3	826.4	3.0
`gr17-J`	2760.0	2759.0	(0.0)	3278.0	15.8	2759.0	(0.0)	2759.0	(0.0)
`eil101-J`	947.4	982.8	3.6	1037.0	8.6	982.8	3.6	975.3	2.9
`fri26-J`	1283.0	1309.0	2.0	1338.0	4.1	1309.0	2.0	1300.0	1.3

Table 1: Computational results: the heuristics procedures and the GAMS solutions

4.2 Time Performance

Following the professor's advice, I compared the processing times between my implementation and that of another student to highlight the computational speed of heuristic algorithms for solving TSPJ using Python and Julia. The processing times for each of the four procedures are detailed in Table 2, 3, 4, and 5. In each table, the first column lists the names of each test problem. The second and third columns show the processing times for Julia and Python, respectively, for each problem. The final column provides the ratio of Julia's processing time to Python's processing time. This comparison helps to illustrate the efficiency differences between the two programming languages when applied to heuristic algorithms for the TSPJ problem.

Table 2: Processing Time(s) of Procedure 1

	Julia	Python	Ratio
gr48-J	0.000746	1.305472	0.000571
berlin52-J	0.000977	0.578119	0.001690
gr24-J	0.000251	0.074708	0.003360
eli51-J	0.000911	1.528503	0.000596
gr21-J	0.000076	0.023244	0.003270
bays29-J	0.000142	0.148823	0.000954
eli76-J	0.002501	6.335782	0.000395
gr17-J	0.000042	0.012332	0.003406
eil101-J	0.003890	23.532655	0.000165
fri26-J	0.000226	0.058228	0.003881

Table 3: Processing Time(s) of Procedure 2

	Julia	Python	Ratio
gr48-J	0.000640	0.657875	0.000973
berlin52-J	0.000807	0.528469	0.001527
gr24-J	0.000087	0.030047	0.002895
eli51-J	0.000634	0.690816	0.000918
gr21-J	0.000069	0.023042	0.002995
bays29-J	0.000098	0.090209	0.001086
eli76-J	0.001756	3.353377	0.000524
gr17-J	0.000034	0.011452	0.002969
eil101-J	0.004360	13.419471	0.000325
fri26-J	0.000160	0.029087	0.005501

Table 4: Processing Time(s) of Procedure 3

	Julia	Python	Ratio
gr48-J	0.000791	1.264036	0.000626
berlin52-J	0.001236	0.563402	0.002194
gr24-J	0.000192	0.072624	0.002644
eli51-J	0.000944	1.513495	0.000624
gr21-J	0.000094	0.022853	0.004113
bays29-J	0.000183	0.147698	0.001239
eli76-J	0.002521	6.207526	0.000406
gr17-J	0.000037	0.012045	0.003072
eil101-J	0.003886	22.870875	0.000170
fri26-J	0.000223	0.057339	0.003889

Table 5: Processing Time(s) of Procedure 4

	Julia	Python	Ratio
gr48-J	0.000800	1.909018	0.000419
berlin52-J	0.000885	2.166853	0.000408
gr24-J	0.000176	0.082216	0.002141
eli51-J	0.000727	1.729355	0.000420
gr21-J	0.000104	0.023061	0.004510
bays29-J	0.000174	0.253415	0.000687
eli76-J	0.002378	6.782063	0.000351
gr17-J	0.000060	0.012496	0.004802
eil101-J	0.005224	31.527643	0.000166
fri26-J	0.000232	0.100602	0.002306

4.3 Discussion

For the solution performance analysis, Table 6 shows the performance gap between my implementation and the paper for the four procedures. First, the solution performance gaps for Procedures 1, 3, and 4 are similar, with Procedure 2 exhibiting a larger gap than the others, which is reasonable. This is because, in Procedure 2, the initial tour is generated without considering job assignments. Although the job assignments are subsequently adjusted and local search improvements are applied, the overall performance of this procedure is inferior to the others. Second, the solution performance gaps for Procedures 1, 3, and 4 are comparable to those reported in the paper. However, the performance gap for Procedure 2 is notably larger than that in the paper. I suspect this discrepancy is due to errors in my implementation of Procedure 2. Despite my efforts to correct it, I was unable to resolve the issue.

Table 6: Solution Performance Gap(%)

	Procedure 1	Procedure 2	Procedure 3	Procedure 4
Paper	2.61	4.68	2.48	2.64
Mine	2.16	13.23	2.16	2.41

For the solution performance analysis, Table 7 first compares the average processing time for each procedure in my implementation. Despite Procedure 2 resulting in a larger solution performance gap as previously described, it saves significant processing time compared to the other procedures. Additionally, the table compares the average processing time ratios between Julia and Python for each procedure, demonstrating that Julia requires considerably less processing time than Python.

Table 7: Mean Processing Time(s) and Mean Ratio

	Procedure 1	**Procedure 2**	**Procedure 3**	**Procedure 4**
Mean Processing Time(s)	3.359787	1.883384	3.273189	4.458672
Mean Ratio (J/P)	0.001829	0.001971	0.001898	0.001621

5 Conclusion

Through this project, I revisited the knowledge of the Traveling Salesman Problem (TSP) learned in class and extended it to the Traveling Salesman Problem with Job-times (TSPJ). The most challenging part of this project was understanding the logical flow between the various steps in the Local Search Improvement algorithm, which posed significant challenges to my implementation.

Overall, this project not only helped me gain a deeper understanding of optimization methods but also significantly improved my programming skills. I am grateful to the professor for this opportunity.

References

Bezalel Gavish and Stephen C Graves. The travelling salesman problem and related problems. 1978.

Mohsen Mosayebi, Manbir Sodhi, and Thomas A. Wettergren. The traveling salesman problem with job-times (tspj). *Computers & Operations Research*, 129:105226, 2021. ISSN 0305-0548. doi: https://doi.org/10.1016/j.cor.2021.105226.