Barcelona School of Economics

Data Science Methodology Program

21D009 Networks: Concepts and Algorithms

Movie recommendations using network-based approaches

Authors:
Codd, Jonny
Chen, Joshua
Gallegos, Rafael
Pérez, Carlos

Professor:
Milán, Pau
Komander, Björn

December 20th, 2023

Contents

List of Figures

List of Tables

1 Introduction

The travelling salesman problem (TSP) is an archetypal problem in linear programming. The setup tells the reader to imagine a salesman who must visit n cities, then asks them to find the shortest path the salesman can take, visiting each city exactly once before returning to the starting city given the distance between all pairs of cities. Its deceptively simple formulation lies in stark contrast to the vast computational expense required to solve it; it is NP-hard, and despite nearly one-hundred years of literature, we rely on heuristic algorithms to find good solutions.

Our work focuses on a variant of the original problem known as the asymmetric travelling salesman problem (ATSP). The ATSP provides a distance matrix that is asymmetric, meaning the distance to travel from one city to another is different from the distance in the reverse direction (see eqs. 1.1-1.5). The problem has many practical applications if the distance matrix is instead taken to represent the length of time to travel from one city to another. In this case, the journey time may vary according to the direction of travel. It can be formulated as follows:

$$\min \quad \sum_{i\in V} \sum_{j\in V\setminus\{i\}} c_{ij}x_{ij} \tag{1.1}$$

$$\text{s.t.} \quad \sum_{j\in V\setminus\{i\}} x_{ij} = 1 \qquad \forall i \in V \tag{1.2}$$

$$\sum_{j\in V\setminus\{i\}} x_{ji} = 1 \qquad \forall i \in V \tag{1.3}$$

$$\sum_{i\in S} \sum_{j\in V\setminus S} x_{ij} \geq 1 \qquad \forall S \subseteq V, S \neq \emptyset \tag{1.4}$$

$$x_{ij} \in \{0,1\} \qquad \forall i \in V, \forall j \in V \setminus \{i\}. \tag{1.5}$$

where:

- V is the set of cities.
- c_{ij} is the cost or distance from city i to city j.
- x_{ij} is a binary variable that equals 1 if the path from city i to city j is included in the tour, and 0 otherwise.
- S is any subset of cities. The subtour elimination constraints (SECs) ensure that for any subset of cities S, there is at least one path leading out of the subset to prevent the formation of subtours. This is crucial for ensuring that the solution is a single tour covering all cities, rather than multiple disconnected tours.

The ATSP asks the program to find the minimum distance the salesman must travel according to the following constraints: each city must be entered exactly once; each city must be left exactly once;

subtours are not allowed.

All implementations have been developed in Python. The source code can be found in the attached jupyter notebook *TSP_project.ipynb.*

1.1 Algorithms

Integer Branch-and-Price: For a sufficiently large n (which, in practice, is relatively small), specifying the exclusion of all SECs is computationally infeasible. As n increases, the number of SECs increases exponentially with 2^n (n representing the number of cities).

The Integer Branch-and-Price algorithm circumnavigates the computational burden by first solving a relaxed form of the problem which initially ignores the exclusion of subtours. Once the algorithm finds a feasible integer solution to the relaxed problem, it checks for subtours and adds SECs to the original formulation.

$$\sum_{i \in S_l} \sum_{j \in V \setminus S_l} x_{ij} \geq 1 \qquad \forall l \in \{1, ..., k\} \tag{1.2}$$

with k representing the number of SECs violated while searching for the optimal solution.

Fractional Branch-and-Price: The fractional Branch-and-Price algorithm is less stringent in restricting solutions to the relaxed problem. Instead of waiting until the algorithm produces a feasible integer solution, it accepts fractional solutions and checks for violated SECs using the Max Flow Problem. If an SEC is violated, it is added as a constraint.

$$\sum_{i \in S} \sum_{j \in V \setminus S} x_{ij} \geq 1 \qquad \exists S \subseteq V, S \neq \emptyset \tag{1.3}$$

The Boykov Kolmogorov Max Flow algorithm from graph_tool python package is used to solve the Max Flow problem. Max Flow detects violated SECs from any vertex to all other vertices in the subtour; if the obtained max flow is less than 1, we add the violated SEC as a constraint.

Once an integer solution with no subtours is found, both algorithms branch the search at variables of interest to look for better solutions. The algorithms are terminated either when an exhaustive search has been performed, or a specified time limit on computation has been reached.

We subset the data to produce instances of varying size and calculate the time taken to reach the optimal solution (if it is indeed reached); the number of iterations; and the final objective value. The algorithms are limited to a maximum run time of five minutes. If they fail to terminate within this limit, we report a solve time of 300 seconds.

When run on ATSP datasets, both solvers reached the optimal solution in reasonable time. However, in symmetric cases with a comparable number of nodes, the solver took much longer. We found the problems in [**web:tsp**] to be more challenging to solve, even though they are symmetric. This finding was surprising, as we expected the algorithm to leverage the symmetries in the distance matrices to reduce computation time. We believe the longer running times should be attributed to more challenging initial parameters in these datasets, rather than to the symmetric property.

2 Task 1

We found that the integer algorithm was far more efficient across all subsets of our problem. It consistently took less time to converge than the fractional algorithm (Figure 2.1), and the difference in time generally grew as the number of vertices increased. This suggests permitting further relaxation of the constraints of the initial problem (i.e., allowing non-integer solutions), then readmitting them later where necessary, is less efficient, at least in terms of computation time, than simply including the constraints from the beginning.

Aditionally, the runtime limit was reached in the instance of the fractional solver with 350 vertices, so no useful statistics were output. Interestingly, though, in the case with 400 vertices the problem was solved by the fractional algorithm before the limit was reached, suggesting that adding the additional data points made the problem easier to solve. From this finding we inferred that, while the integer algorithm finds a solution more quickly than the fractional algorithm, the exact time difference depends on the parameters of the problem. In cases in which both algorithms converged within the time limit, their objective values were consistent. Overall this analysis implies that both algorithms are accurate, but the integer variant is quicker.

Finally, we discerned no clear pattern between the number of iterations in the integer vs fractional solver.

Vertices	**Integer**			**Fractional**		
	Obj	**Iter**	**Time**	**Obj**	**Iter**	**Time**
43	398	365	0.15	398	179	0.76
93	564	475	0.27	564	1,017	4.20
143	907	3,720	4.20	907	3,120	8.52
193	1196	8,973	4.05	1196	7,771	82.40
243	1645	13,792	9.57	1645	9,642	49.11
293	1807	12,784	16.24	1807	14,346	78.40
343	2071	20,397	19.71	NA	NA	300.00
393	2383	45,017	59.09	2383	41,738	75.83
443	2720	25,485	53.19	2720	42,299	89.35

Table 2.1: Objective Value, Iterations, and Runtime for Integer and Fractional algorithms

3 Task 2

It is computationally impractical to add all SECs to the original LP, so we adopt a more parsimonious approach in an effort to enhance solving efficiency and reduce overall computation time. In particular, we first remove all sub tours of length two and, subsequently, those of length three.

Vertices	**No constraints**			**Cardinality 2**			**Cardinality 3**		
	Obj	**Iter**	**Time**	**Obj**	**Iter**	**Time**	**Obj**	**Iter**	**Time**
43	398	365	0.15	398	193	0.38	398	348	0.21
93	564	475	0.27	564	390	0.53	564	520	1.51
143	907	3,720	4.20	907	2,029	1.78	907	918	10.81
193	1,196	8,973	4.05	1,196	2,666	7.87	1,196	2,854	39.35
243	1,645	13,792	9.57	1,645	4,552	7.75	1,645	4,222	136.02
293	1,807	12,784	16.24	1,807	6,134	17.13	NA	NA	300.00
343	2,071	20,397	19.71	2,071	1,856	10.32	NA	NA	300.00
393	2,383	45,017	59.09	2,383	9,184	30.64	NA	NA	300.00
443	2,720	25,485	53.19	2,720	5,779	30.29	NA	NA	300.00

Table 3.1: Objective Value, Iterations, and Runtime for the Integer algorithm with SECs of cardinality 2 and 3

Fractional Branch-and-Price: By making the initial relaxation tighter, we attenuate the numerical distance between the relaxed solution and the optimal integer solution. This reduces the number of nodes in the branch-and-bound tree, as fewer branches are needed to transform the fractional solution of the relaxation into the integer solution. Conversely, adding many SECs upfront increases the size and complexity of the constraint matrix the solver is confronted with at each node in the branch-and-bound tree. This extra computational expense likely increases solving time.

Table 3.2 summarizes the results for the fractional solver. As for the integer solver, adding SECs of cardinality 2 and 3 reduced the number of iterations required to find the optimal solution (if it was indeed reached). However, SECs of cardinality 2 improved the performance of the algorithm, with reduced computation time (Figure 3.1) and the optimal solution was found in time in all instances. This implies the reduction in required iterations offset the increased computational cost of checking additional constraints. Conversely, SECs of cardinality 3 generally hindered performance, with no optimal solution found in instances of over 293 vertices.

Vertices	**No constraints**			**Cardinality 2**			**Cardinality 3**		
	Obj	**Iter**	**Time**	**Obj**	**Iter**	**Time**	**Obj**	**Iter**	**Time**
43	398	179	0.76	398	171	0.36	398	189	1.04
93	564	1,017	4.20	564	401	3.90	564	520	2.60
143	907	3,120	8.52	907	1,662	18.38	907	1,824	28.63
193	1,196	7,771	82.40	1,196	3,727	17.15	1,196	2,685	47.84
243	1,645	9,642	49.11	1,645	5,191	28.35	1,645	3,713	109.35
293	1,807	14,346	78.40	1,807	4,510	27.43	NaN	NaN	300.00
343	NaN	NaN	300.00	2,071	8,283	103.88	NaN	NaN	300.00
393	2,383	41,738	75.83	2,383	9,924	271.21	NaN	NaN	300.00
443	2,720	42,299	89.35	2,720	12,923	240.33	NaN	NaN	300.00

Table 3.2: Objective Value, Iterations, and Runtime for the Integer algorithm with SECs of cardinality 2 and 3

4 Task 3

	Integer (2720 - 25,485 iter - 53.19s)			**Fractional (2720 - 42,299 iter - 89.35s)**		
SECs	**Obj**	**Iter**	**Time**	**Obj**	**Iter**	**Time**
(2 ,3)	2,720	47,671	45.91	2,720	25,342	234.74
(2, 4)	2,720	10,585	29.35	NaN	NaN	300
(2, 5)	2,720	11,344	33.23	NaN	NaN	300
(3, 3)	2,720	34,783	52.34	2,720	21,655	293.50
(3, 4)	2,720	34,851	89.85	2,720	7,422	267.51
(3, 5)	2,720	24,680	48.07	2,720	21,284	191.74

Table 4.1: Objective Value, Iterations and Runtime for Integer and Fractional algorithms with subset of SECs

5 Conclusions