Project Report: Solving a delivery problem.

Alanta Kwok

May 2023

1 Introduction

The motivation behind the project is as follows (denoted as the **delivery problem**):
Suppose a mailman needs to deliver letters to different locations starting from the post office. Given a map of locations that needs to be visited, what is the shortest route that allows him to start from the post office, deliver all letters to the respective locations, then return to the post office?

One can model this as a traveling salesman problem (TSP), i.e. finding the Hamiltonian cycle of a graph with minimum cost. The key concern with treating this as a TSP is that not all graphs contain Hamiltonian cycles; as such the idea behind solving the delivery problem is allowing the mailman to travel through edges multiple times.

This project delves into details of solving the delivery problem as a modification of TSP, its results under different algorithms, possible constraints that affect routing and examples of applications to real-life problems.

2 Notation and vocabulary definition

Prior to discussing the project in detail it is wise to first define key notations and vocabulary to avoid confusion.
General notation related to this project is defined as follows:

Notation	Description
x_e	Edge variable for any edge e
c_e	Time cost for any edge e
$\delta(i)$	All edges adjacent to node i
$\rho(U)$	All edges within node subset U

In the context of the delivery problem, one can represent map with streets and locations as a graph G with set of vertices and edges (V, E) respectively and $|V| = n$.

Every feasible solution to the delivery problem is hereby defined as a route. Vocabulary related to routing are used in the the context of graph theory: e.g. paths are defined as a walk where all vertices and edges are distinct, cycles are defined as closed walks, etc.

3 Integer program

3.1 Route structure

If one were to treat this as a TSP, the program that solves the TSP can be written as:

$$\begin{array}{lll} \min_x & \sum_{e\in E} c_e x_e & \\ \text{s.t.} & \sum_{e\in\delta(v)} x_e = 2 & \text{for all } v \in V \\ & \sum_{e\in\delta(U)} x_e = 2 & \text{for all } U \subseteq V \text{ with } 3 \leqslant |U| \leqslant \lceil \frac{n}{2} \rceil \\ & & x \text{ binary} \end{array}$$

In the case of our delivery problem, the mailman is allowed to cross any edges as many times as desired; x is no longer limited to being binary, and should instead be considered an integer variable. This creates a slew of issues that need to be considered:

- A Hamiltonian path (thus cycle) does not necessarily exist in G, so feasible routes to the delivery problem may contain cycles of cardinality less than n or possibly no cycles at all.
- Since there is no restriction on the number of times each edge can be travelled, there is also no restriction on the number of times each node can be visited (as opposed to TSP where all edges and nodes could only be used once.).

Thus the constraint $\sum_{e\in\delta(v)} x_e = 2$ no longer holds.

Based on the requirement that any route must pass through all locations and loop back to the post office, it is implied that all feasible routes should consist of a spanning tree:

- Spanning tree consists of $n-1$ edges.
- Spanning tree connects across all nodes in V.
- There are no cycles in the spanning tree (i.e. no subtours.)

If the problem has a TSP solution, then its corresponding spanning tree must be a Hamiltonian path.

3.2 Program construction

Every feasible delivery problem solution x has a spanning tree; any edge used in that spanning tree must also be used in x. Constraining the solution to contain a spanning tree requires two sets of binary variables: The first set of binary variables y_e denotes whether edge e has been used in x. This can be represented under the pair of constraints:

$$\begin{array}{rcl} x_e & \geqslant & y_e \\ x_e & \leqslant & M_e y_e \end{array}$$

Here the parameter M_e represents the maximum number of times the edge e can be used; this constant can be adjusted up to one's discretion. If one decides to not enforce a limit, M_e can be chosen to be a large constant that is no smaller than $n-1$, the worst-case scenario.

The second set of binary variables z_e denotes whether edge e was used in any choice of spanning tree within x. To illustrate that any edge used in the spanning tree must be used in solution x, one can put forward the constraint $z_e \leqslant y_e$.
The spanning tree should also consist of $n-1$ edges; the associated constraint is $\sum_{e\in E} z_e = n-1$.

Next, to ensure that there are no cycles in the spanning tree, among all edges that exist within any subset of nodes $U \subset V$ with cardinality $3 \leqslant |U| \leqslant n-1$, less than $|U|$ edges should be chosen for the spanning tree. Then the constraints associated it can be written as:

$$\sum_{e \in \rho(U)} z_e \leqslant |U| - 1 \qquad \text{for all } U \subset V \text{ where } 3 \leqslant |U| \leqslant n-1$$

In the context of TSP, this is an alternate way of writing the subtour constraints $\sum_{e \in \delta(U)} x_e = 2$ for all $U \subseteq V$ with $3 \leqslant |U| \leqslant \lceil \frac{n}{2} \rceil$.

Finally, to ensure that the spanning tree connects across all edges, one needs to prevent disconnected single-edge loops from occurring in any spanning tree (and thus in any feasible solution.) A disconnected single-edge loop exists if for some edge e_1 with $z_{e_1} = 1$, no other edges $e_2 \in \{E \setminus e_1 : z_{e_2} = 1\}$ are connected to e_1.
In a TSP, since all x_e are binary and are constrained under $\sum_{e \in \delta(v)} x_e = 2$, single-edge loops will never occur; but since the delivery problem allows edges to be crossed multiple times, prevention of such occurrences is mandatory. One can write this as the set of constraints:

$$\sum_{e \in \delta(i), e \neq (i,j)} z_e + \sum_{e \in \delta(j), e \neq (i,j)} z_e \geqslant 1 \qquad \text{for all } (i,j) \in E$$

To finish constructing the program, any feasible solution x consists of a route that loops from and back to the post office. Thus for each node, the sum of the total number of times all adjacent edges have been crossed should be even and non-zero. Describing these requirements under the following constraints:

$$\begin{aligned} \textstyle\sum_{e \in \delta(v)} x_e &= 2w_v \text{ for all } v \in V \\ w_v &\geqslant 1, \text{ integer for all } v \in V \end{aligned}$$

If one were to enforce equality on w (i.e. $w_v = 1$) then the delivery problem is simplified to a TSP.

To summarize, the entire integer program for the delivery problem can be written as follows:

$$\begin{array}{lrcll} \min_x & \sum_{e \in E} c_e x_e & & & \\ \text{s.t.} & \sum_{e \in \delta(v)} x_e & = & 2w_v & \text{for all } v \in V \\ & w_v & \geqslant & 1 & \text{for all } v \in V \\ & x_e & \geqslant & y_e & \text{for all } e \in E \\ & x_e & \leqslant & M_e y_e & \text{for all } e \in E \\ & z_e & \leqslant & y_e & \text{for all } e \in E \\ & \sum_{e \in E} z_e & = & n-1 & \\ & \sum_{e \in \rho(U)} z_e & \leqslant & |U| - 1 & \text{for all } U \subset V \text{ where } 3 \leqslant |U| \leqslant n-1 \\ & \sum_{e \in \{\delta(i) \bigcup \delta(j)\} \setminus (i,j)} z_e & \geqslant & 1 & \text{for all } (i,j) \in E \\ & & & w, x \text{ integer, } y, z \text{ binary} & \end{array}$$

4 Methods

4.1 Practical considerations

As shown in constructing the program, the delivery problem is an extension of the TSP problem, thus practical problems that arise while solving for TSP will inevitably show up when solving for the delivery

problem, and to a much worse extent.

The largest obstacle in solving the delivery problem is the expansion in dimensionality: any feasible solution to the TSP problem is within $\mathbb{B}^{\frac{1}{2}(n)(n-1)}$, whereas any feasible solution to the delivery problem is within $\mathbb{Z}^n \times \mathbb{Z}^{\frac{1}{2}(n)(n-1)} \times \mathbb{B}^{\frac{1}{2}(n)(n-1)} \times \mathbb{B}^{\frac{1}{2}(n)(n-1)}$: there are at least three times the number of variables to consider, and not all variables are binary. As such, finding the optimal solution to a delivery problem is expected to more computationally intensive than its TSP equivalent (if it exists.)

Moreover, TSP itself can be complex to compute for large n as the number of subtours it needs to consider grows exponentially. This flaw persists for the delivery problem; under the current program formulation the number of subtours considered is no less than $n! \,/\, (\lceil \frac{n}{2} \rceil !)^2$.

As such, searching for the optimal solution to the delivery problem under the program can be impractical even for $n = 20$ when using optimization software (later demonstrated in the results section.) When computing through software, one may expect the majority of computation time to be spent on determining the feasibility of the current iterate of the solution, more specifically whether a spanning tree exists.

To minimize this computation time, I've taken a few heuristic approaches that aim to provide a close estimate to the optimal objective value. These approaches are listed in the following section.

4.2 Heuristic approaches

The most intuitive approach is to adopt the minimum spanning tree. This method skips the long process that checks for spanning tree feasibility, which greatly reduces computational time required. In this project, the method was implemented by first generating the MST through the greedy algorithm introduced used in class, then representing it in the program as binary parameters t_e and setting up the constraints $y_e \geqslant t_e$.

In practice, using the MST is generally not efficient since there may exist alternate routes with less total costs that are neglected under MST's greedy approach. An example of this shortcoming is how using MST does not yield better performances than the Christofides algorithm in TSP. Given the impracticality of MST, I decided to look into two alternatives to MST that provide more leniency for spanning tree checking.

The first alternative is greedily selecting MST components. The underlying concept is instead of using the entire MST, only a portion of the edges with the least costs are forced into the solution instead. The model then tries to find a spanning tree based on those edges. For this project I decided on selecting the cheapest half (rounded down) of the MST.

The second alternative is performing a 3-change on the MST. In class, k-changes (most notably 3-changes for its consistent performance with small changes) were introduced to provide better alternatives to an initial feasible TSP solution. With the delivery problem k-changes are much harder to implement simply because edges can be used multiple times and traversed in any direction. So instead of doing a 3-change on the delivery problem solution, I went with a 3-change on the MST. The model then searches for the 3-opt solution.

4.3 Implementation of experiments

Experimentation was conducted into two different parts: the first part was comparing the computation time required to solve TSP and the delivery problem on computational software to test the efficiency of computing solutions for large graphs. The second part was comparing delivery problem solutions with different heuristic approaches (including TSP) applied to determine the viability of said methods.

The graphs used for experimentation were randomly generated using Python. Edges were generated based on a probability parameter, with their respective time costs being randomly generated integers between 1 and 10 to correspond to number of minutes required to traverse between its associated nodes. All graphs were guaranteed to have at least one Hamiltonian cycle (thus at least one spanning tree.) As mentioned in the previous section, the MST was obtained using the greedy algorithm. For the purpose of simulating real-life maps, the probability parameter was set to 0.2 (20% chance of generating.)

The delivery problem program was coded into AMPL and optimization was performed using the CPLEX solver. A separate TSP program was also coded for time comparisons between TSP and delivery problem. I wrote scripts that ran the model and added constraints related to heuristic approaches to obtain their respective solutions. Both objective values and computation durations were recorded for 520 graphs of various sizes (see Table 1.) Larger graph sizes were considered early into the project, but abandoned during experimentation due to AMPL's limitations on clique size and computation power (both related to subtour constraints.)

Graph size (n)	11	13	15	16	17	18	19
Sample size	100	100	100	100	70	40	10

Table 1: Graph sizes and respective sample sizes used for experimentation.

All models, data files and results are posted in a *Github repository*[1] for accessibility to readers.

5 Results and Discussion

5.1 Computation time comparison between TSP and delivery problem

Results were computed using a Intel Core i7-8550U CPU 1.80 GHz processor with 8GB RAM. Table 2 shows basic statistics of computation time required for both TSP and the delivery problem. As expected, the delivery problem requires longer computation times, with the time difference being more significant over large n. The mean computation time ratio is expected to approach 2. There is also a significantly larger variance in delivery problem computation times; this can be attributed the worst cases having significantly a longer solving duration relative to the means.

One may also observe the aforementioned exponential growth in computation time over increasing n in Figure 1. It should be noted that at $n = 20$, AMPL fails to run the delivery problem model due to lack of memory space. For a realistically-sized graph, using all subtour/cycle constraints is simply infeasible. Proof of this is by predicting computation times given the trajectory of the bar plot: graphs of size $n = 100$ requires $\approx 10^{29}$ years to compute.

[1] `https://github.com/alantakwok/generalizing-TSP`

Graph size (n)	TSP			Delivery problem		
	Mean	SD	Max	Mean	SD	Max
11	0.8	0.4	2	1.0	0.3	2
13	1.2	0.4	2	1.7	0.5	3
15	3.6	0.6	5	6.1	1.3	11
16	9.4	1.1	13	15.7	3.3	28
17	21.0	2.0	31	39.0	7.8	65
18	48.0	3.9	63	103.1	23.1	160
19	130.1	10.8	148	254.9	57.1	372

Table 2: Computation times (in seconds) for TSP and delivery problem between graphs of different sizes.

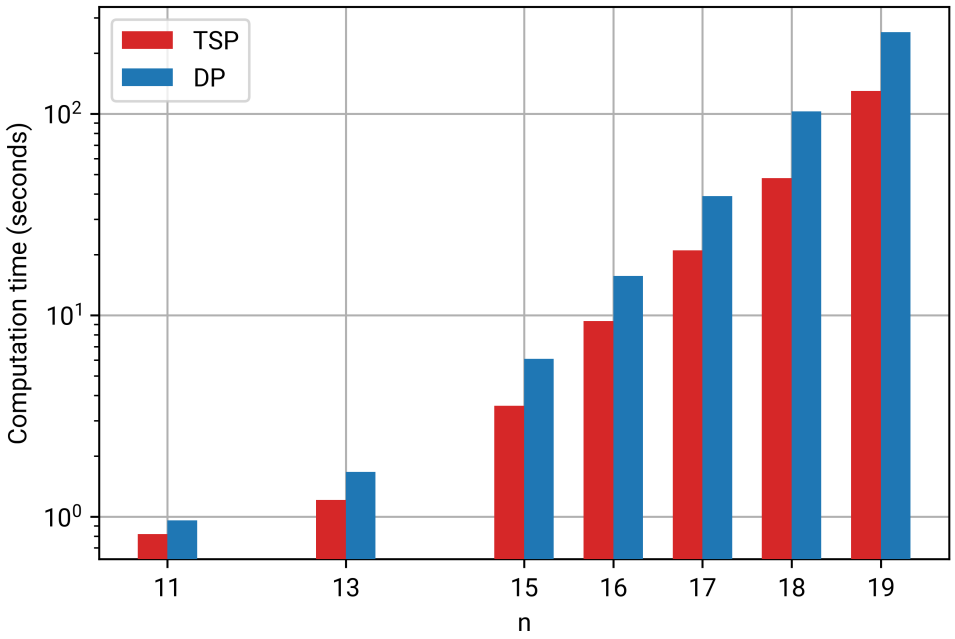


Figure 1: Mean computation time comparison between TSP and delivery problem (labelled as DP) over increasing n.

5.2 Performance of delivery problem solutions over TSP solutions

Figure 2 shows the cumulative frequency of the ratios of optimal objective values of delivery problems compared to that of TSP.

Out of the 520 samples,

- 58% of cases yielded a reduction in optimal objective value (from TSP to delivery problem.)
- 32% of cases yielded a reduction of $\geqslant 3\%$.
- 19% of cases yielded a reduction of $\geqslant 5\%$.
- 5% of cases yielded a reduction of $\geqslant 10\%$.

Even though the delivery problem is significantly more computationally intensive than TSP, the statistics show that there is a good probability that total costs can be considerably decreased (even 3% is significant

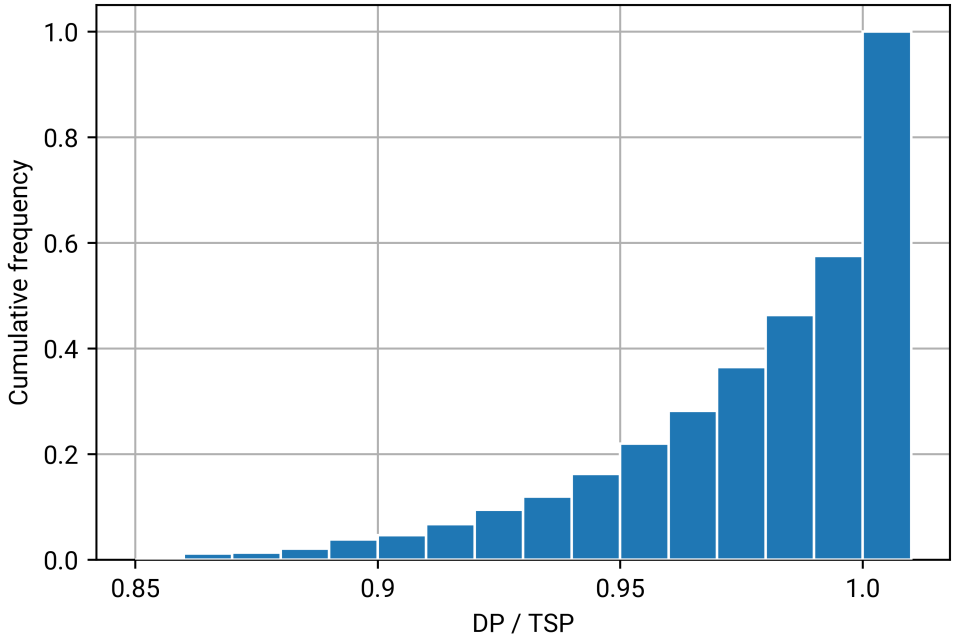


Figure 2: Cumulative frequency histogram of optimal objective value ratios. Each bar represents the cumulative frequency of samples with ratio less than its corresponding right bound tick.

from a corporate perspective.) If not for the increase in computation time, one may find the delivery problem to be a worthy alternative to TSP. The use of heuristics (akin to how k-changes are used for TSP) may significantly reduce the time to find such a solution of similar improvements.

5.3 Efficiency of delivery problem heuristics

The list of heuristics used in the experiments were:

1. Approximation through **TSP solution**
2. Solution is forced to have entire MST; hereby referred to as **MST solution**
3. Solution is forced to have half of (rounded down if MST has odd cardinality) the edges of the MST with the least cost; hereby referred to as **MST-half solution**
4. 3-change is performed onto MST; hereby referred to as **MST-3-opt solution**

Figure 3 shows the efficiency of using the above heuristics to provide a close approximation to the true optimal objective value of the delivery problem. Table 3 offers statistics for a numerical comparison.

Ratio	$\leqslant 1.03$	$\leqslant 1.05$	$\geqslant 1.10$	**Max**
TSP	67.8%	79.4%	5.8%	1.19
MST	5.2%	10.8%	65.8%	1.33
MST-half	60.2%	76.9%	3.8%	1.14
MST-3-opt	87.1%	95.6%	0.2%	1.16

Table 3: Computation times (in seconds) for TSP and delivery problem between graphs of different sizes.

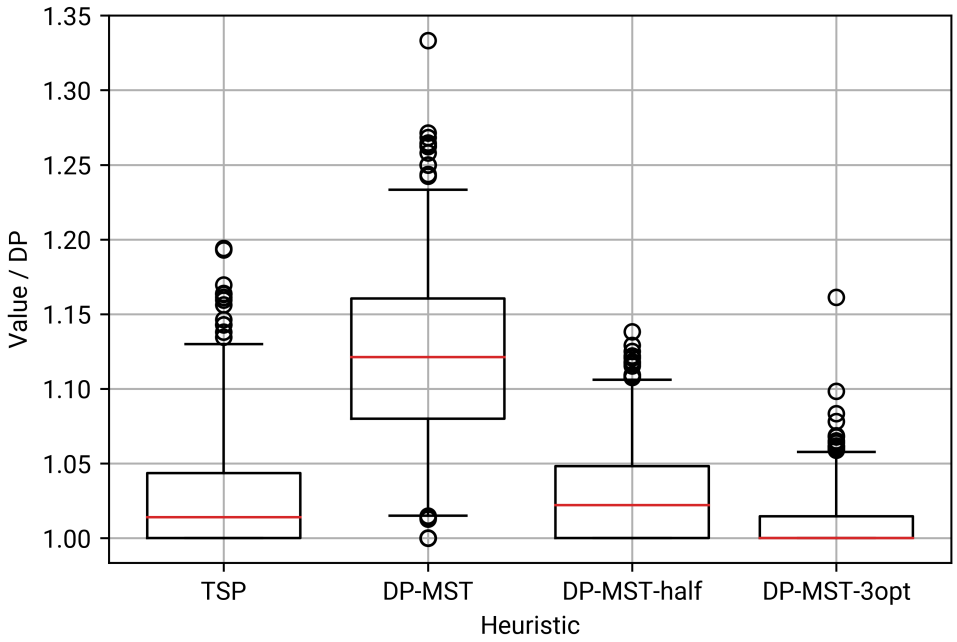


Figure 3: Boxplots of optimal objective value ratios for different heuristics. Medians are indicated by the red lines. Boxes represent quartiles of the distribution. Outliers represent cases outside of the middle 95%.

First looking at TSP solution performances, statistics from Table 3 show that TSP offers a reliable approximation for a majority of the cases, there remains a noticeable portion of the distribution that exists in higher extremities, with its worst outlier ratio nearing 1.2. The goal of the other heuristics is to minimize this portion (denoted by $\geqslant 1.10$) as much as possible while maintaining a similar, if not better performance on a majority of the cases compared to TSP.

The MST solution does a terrible job in this regard as this high-exteremity portion consists of roughly two-thirds of the distribution. This falls within expectations as MST's greedy algorithm approach does not account for optimizing the entire route. What I had not expected was how terrible some of outlier cases were; as evident by the boxplots in Figure 3, there were rare instances where MST solution costs had a ratio of more than 1.25. The only upside to using the MST solution is that it takes little time to compute, but the tremendous loss in accuracy does not justify its usage even for large n.

Using the cheapest half of the MST seems to improve results, with distribution comparable to TSP with seemingly lower ratios for outliers on the higher end. As of currently, I cannot say with certainty that this distribution is tighter; more graphs of significantly larger sizes should be examined before a conclusion can be made.

Thankfully, using MST 3-opt solutions appear to have the same effect on delivery problems like they do with TSPs; this heurstic obtains the exact delivery problem optimal values for more than 50% of the cases, comes within 5% of the value for more than 95% of cases and 10% of the value for more than 99% of cases. Using larger k-changes in the MST can potentially result in a better distribution, but as it stands currently 3-opt solutions are a great estimator.

It should be noted that in theory, the duration required to search both MST-half and MST 3-opt solutions should be shorter compared to that of the general delivery problem since the number of feasible solutions that exist are much smaller, making the heuristics even more viable as alternatives to TSP. During implementation however I decided to find these solutions by adding additional constraints onto the model which ultimately did not change how long it took to find those results.

6 Implementing realistic limitations

In real life, streets and roads often have limitations which affects the optimal route, such as one-way streets, no left turns, speed limits, etc. Dealing with these limitations requires the program to be altered: most notably it is convenient to convert our program to be applicable to directed graphs for wider flexibility, since in its current state the solutions it produces does not account for direction, one of the more trivial features for route planning.

6.1 Program for directed graphs

Prior to altering the program one should first create directed graphs that represent the map. For each edge in the original undirected graph $e = (i, j)$, create two seperate two edges in the directed graph: $e = (i, j)$ with direction $i \to j$, and $\bar{e} = (j, i)$ with direction $j \to i$. For clarity, denote the set of original undirected edges as E and the set of directed edges as E_D.

Moving on to the program itself, we'll consider the constraints one by one:

- The even total adjacent edges constraints $\{\sum_{(i,j)\in\delta(v)} x_{(i,j)} = 2w_v \text{ for all } v \in V \ ; \ w_v \geqslant 1\}$ remain unchanged since these constraints are direction-independent.
- Constraints for binary edge variables y should be changed to accomodate for direction:

$$\begin{aligned} x_e + x_{\bar{e}} &\geqslant y_e \quad \text{for all } e \in E \\ x_e + x_{\bar{e}} &\leqslant M_e y_e \quad \text{for all } e \in E \end{aligned}$$

- The remaining constraints from the original program are all based on binary edge variables that only occur for the set of undirected edges E, thus making it unnecessary to change them.
- The updated objective is $\min_x \sum_{e\in E_D} c_e x_e$ with $c_e = c_{\bar{e}}$.

One final aspect to consider for the directed graph program is flow conservation, where the number of adjacent edges in a solution that end at a vertex should be equal to the number of adjacent edges in the solution that start from the same vertex. This can be mathematically represented as:

$$\sum_{(i,j)\in\delta(j)} x_{(i,j)} = \sum_{(j,k)\in\delta(j)} x_{(j,k)} \quad \text{for all } i, j, k \in V,\ i \neq j,\ i \neq k,\ j \neq k$$

The entire program for directed graphs can be found on the next page. One may then proceed to add additional constraints that may reflect the current status of roads in the map.

$$
\begin{array}{lrcll}
\min_x & \sum_{e\in E_D} c_e x_e & & & \\
\text{s.t.} & \sum_{(i,j)\in\delta(v)} x_{(i,j)} & = & 2w_v & \text{for all } v \in V \\
& w_v & \geqslant & 1 & \text{for all } v \in V \\
& \sum_{(i,j)\in\delta(j)} x_{(i,j)} & = & \sum_{(j,k)\in\delta(j)} x_{(j,k)} & \text{for all } i,j,k \in V,\ i \neq j,\ i \neq k,\ j \neq k \\
& x_e + x_{\bar{e}} & \geqslant & y_e & \text{for all } e \in E \\
& x_e + x_{\bar{e}} & \leqslant & M_e y_e & \text{for all } e \in E \\
& z_e & \leqslant & y_e & \text{for all } e \in E \\
& \sum_{e\in E} z_e & = & n-1 & \\
& \sum_{e\in\rho(U)} z_e & \leqslant & |U|-1 & \text{for all } U \subset V \text{ where } 3 \leqslant |U| \leqslant n-1 \\
& \sum_{e\in\{\delta(i)\bigcup\delta(j)\}\setminus(i,j)} z_e & \geqslant & 1 & \text{for all } (i,j) \in E \\
& & & w, x \text{ integer},\ y, z \text{ binary} &
\end{array}
$$

6.2 Additional constraints

For this project, I have only considered one-way streets, U-turn penalties and fuel constraints, which are relatively easy to implement.

One way streets (or temporarily closed streets) can easily be implemented by simply by constraining the edge variable(s) of specific edges to be equal to 0. Fuel constraints can also be easily implemented by creating fuel parameters for each edge and then adding constraints relative to fuel requirements in the form of the strict inequality $Ax < b$. If one desires to add fuel stations and its corresponding penalties respectively one may consider the inclusion of lifted inequalities similar to that of knapsack problem systems.

As for U-turn penalties, one can check for whether a U-turn has been made at a vertex by looking at whether exactly one adjacent street (independent of direction) has been used. Creating the binary variables u for each vertex, one adds the constraints

$$
\begin{array}{rcll}
\sum_{e\in\{\delta(v)\bigcap E\}} y_e - 1 & \leqslant & 1 - u_v & \text{for all } v \in V \\
\sum_{e\in\{\delta(v)\bigcap E\}} y_e - 1 & \geqslant & M_v(1-u_v) & \text{for all } v \in V
\end{array}
$$

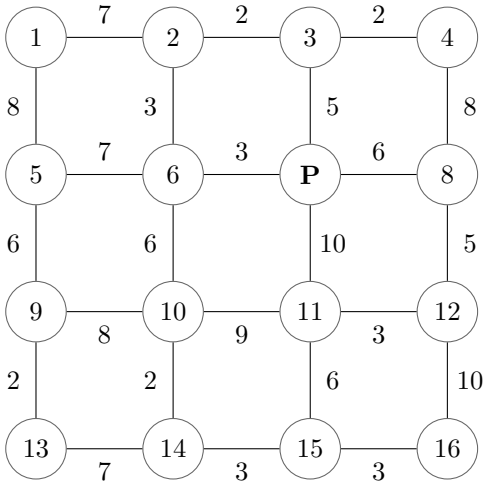
with M being an arbitrary large constant (again $n-1$ should suffice.) The objective function should also be altered by adding a time penalty for each U-turn made.

The following are some other realistic constraints that I did not consider for the project due to either lack of time or being outside the focus of investigation.

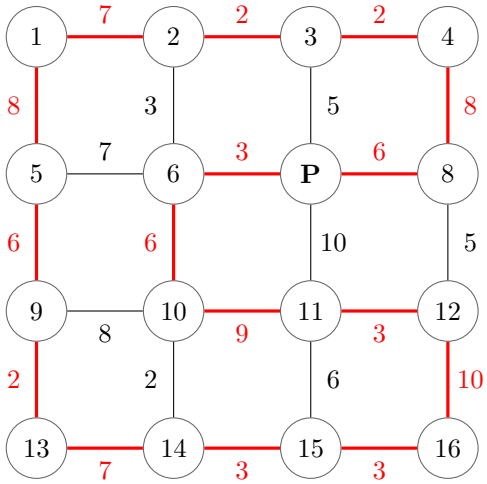
- **Time cost variation**: over the course of this project I have taken the liberty to assume the edge costs are constants independent of the time of day and traffic. Had this assumption been lifted I would've had to implement stochastic processes to further randomize edge costs, or create a large number of edge cost matrices which represent the time cost at each specific hour/traffic instance.
- **Minimization of left turns**: delivery services often promote safety of deliveries by minimizing the number of left turns required. This can be done by implementing time cost penalties to the for every left turn made; the problem with adding this as a constraint is that the increase in the number of constraints typically increases substantially with graph size, having to consider a large number of permutations of edges involved with possible left turns. Implementation would be much easier had I chosen to use dynamic programming to solve the delivery problem.

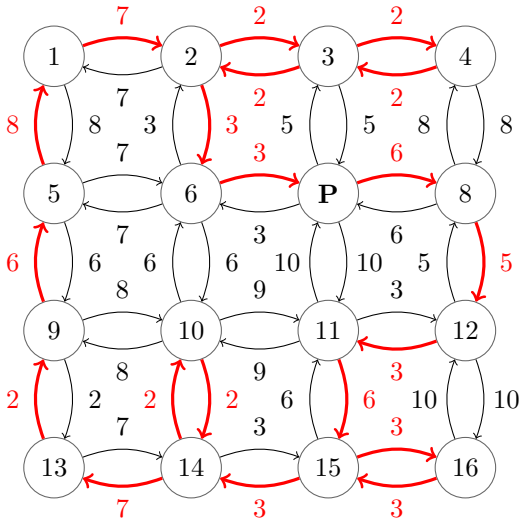
6.3 Example application

Here I shall demonstrate adding realistic constraints to a directed graph.
Suppose the graph in question is as follows, with P denoting the start/end:

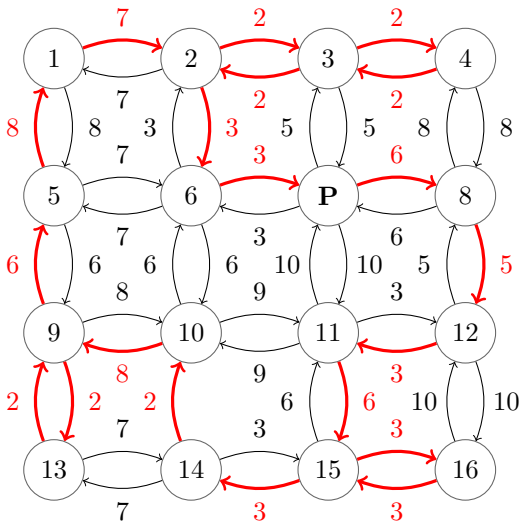


The TSP solution has an optimal objective value of 85 and follows the route on the left. The delivery problem solution has a much improved objective value of 77 with a different route shown on the right. The value obtained was computed without any U-turn time penalties. There are 3 U-turns (at vertices 4, 10, 16.) Suppose it takes one minute to perform a U-turn at each node. After adjusting the penalty parameter in AMPL, the optimal delivery problem solution still remains the same, this time with an objective value of 80.

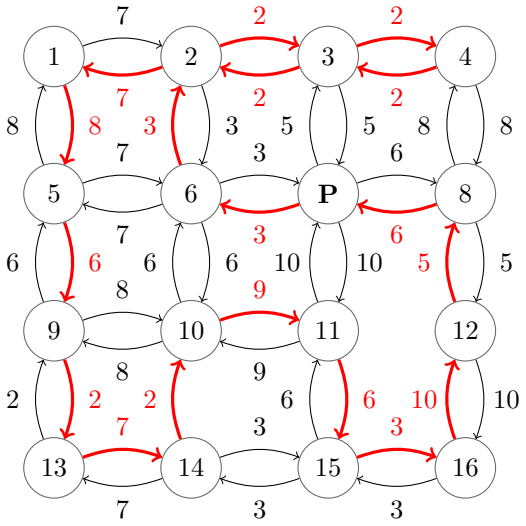




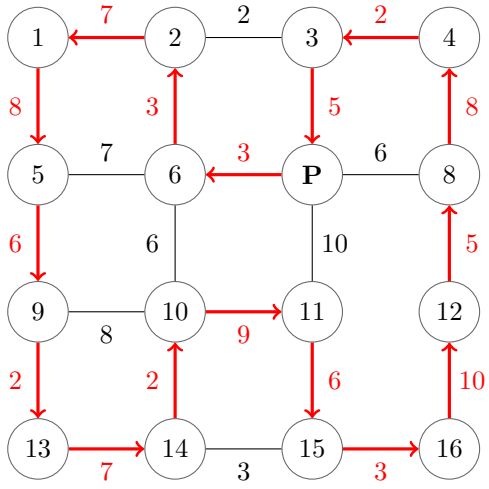
Suppose now that street $(10, 14)$ has been forced into one-way traffic due to construction, specifically $10 \rightarrow 14$ is closed. Adding the respective constraint gives a new route with optimal value 81:



Finally, suppose street $(11, 12)$ has been closed due to a traffic accident. Re-routing (optimal value 86):



Turns out there exists an optimal TSP solution with the same objective value.



Had I decided to implement penalties for multiple visits at a vertex, AMPL would've preferred the above TSP solution instead.

7 Conclusion and afterthoughts

Throughout this project I've shown that generalizing the TSP (i.e. removing the limitation that every vertex must only be visited once) creates a much more complex problem that requires much longer to solve with a favorable possibility of ending with a better objective value. Similarly to TSP, performing 3-change on an initial feasible solution often leads to the optimal value, though the TSP solution itself is generally a decent estimate. I've also shown the flexibility of the generalization by showing how it account for direction by applying it to directed graphs.

The largest obstacle in tackling this project was finding heuristics to more quickly determine the feasibility of a solution. Obviously this issue would've been easily solved if dynamic programming was implemented instead; it's unfortunate that this would be very difficult to achieve given the limitations of AMPL. I had considered using other modeling software/TSP solvers such as PYOMO and CONCORDE, and I believe these softwares would be better at tackling the generalization; the former, based on Python, would've been able to determine spanning trees easily using the NetworkX package, whereas the latter is built to solve TSP formulations automatically and is generally reliable. The only reason why I ended up not using them was that a lot of what I had in mind for the project was much easier to execute through AMPL.

I think that with more time I would've been able to implement even more realistic constraints to make the current delivery program more complete. Overall this was a fun project to tackle from start to finish; every time I ended up making significant progress I would always ask myself if I could make improvements/take it one step further, and I think that if I had started learning about integer/combinatorial optimization a year earlier I would've dedicated this as my research topic for a PhD degree.