Exploring CAVADP model for Last-Mile Deliveries Under Different Settings

Yinsheng Wang
Yiju Hu

Department of Industrial and Systems Engineering
University of Washington
Seattle, WA 98105, USA

ABSTRACT

This project mainly follows a streamline of research in modeling the value of involving autonomous vehicles into the delivery system, which mainly includes two papers, one in Management Science (Reed, Campbell, and Thomas 2022b) and the other in Transportation Science (Reed, Campbell, and Thomas 2022a). Both of two papers focus on the CAVADP model, which models an autonomous vehicle that can drop off the delivery person at selected points where the delivery person makes deliveries to the final addresses on foot. We discussed several assumptions of the model, tested the key theorems in the distinct solution algorithm and implemented the mixed-integer programming of the full model. The project provides some insights on formulating and solving the optimization models in last-miles deliveries, and laid foundations for future research.

1 Introduction

The Digital Economy Index predicts that the global e-commerce sales will reach \$4.2 trillion this year. U.S. consumers will account for nearly one-quarter of that spending. It also shows that nearly 69% of Americans reportedly indulged in online shopping. The large number of online orders definitely lead to a large number of deliveries. Allen et al. find out that drivers parked 37 times on average per day.(Allen, Piecyk, Piotrowska, McLeod, Cherrett, Ghali, Nguyen, Bektas, Bates, Friday, et al. 2018) In large cities, each time the driver needs about 9 minutes to find out a parking spot.(Cookson and Pishue 2017)Using autonomous vehicles is a good way to improve the delivery efficiency and reduce the delivery cost. However, autonomous vehicles cannot do the delivery work alone because of the complicated real situations. As a result, many companies are now interested in a new pattern of autonomous-assisted delivery that using autonomous vehicles to help the delivery person save time.

To show the effect of this pattern, Sara et al. build a model to solve the capacitated autonomous vehicle assisted delivery problem (CAVADP). They show that 30%-77% of the delivery time can be saved with the help of autonomous vehicles when considering the time to find parking. But the result is under some assumption according to the urban area features. For example, they assume that a customer lies at every intersection of the grid.

What if some of the assumptions are relaxed? Will the autonomous-assisted delivery pattern still be effective? This project aims to relax and test several important assumptions of the CAVADP model. We first review some vehicle routing papers in Section 2. Then in Section 3, based on the paper by Sara et al., we talk about the mentioned distinct solution method (especially calculating the Hamiltonian path) and tested the key theoretical results. Since this paper only consider the urban environment, in Section 4, we extend the urban settings (grid) to rural settings, where grid assumption no longer stands and revisited the mixed-integer programming formulation, under the guidance of a new paper also by Sara et al. Section 5 contains conclusions and our takeaways from the project.

2 Distinct Solution Method

In this part, we mainly focused on the distinct solution method for CAVADP model under the rectangular grid assumptions.

2.1 Algorithm

A solution of the CAVADP on a grid is constructed as follows: First, use Table 1 to determine the first customer s and last customer t that the vehicle will visit. Then, the delivery person follows the Hamiltonian path from customer s to customer t. The reloading points where the vehicle synchronizes with the delivery person are determined based on the values of f , w, and d (Theorem 4). The customers be- tween these reloading points on the Hamiltonian path determine the sets of customers to be served.

Algorithm 1: CAVADP on Solid Rectangular Grid

Input:
Size of solid rectangular grid g × h with n customers;
Location of the depot;
Capacity of delivery person q;
Driving speed of vehicle d;
Walking speed of delivery person w;
Fixed time for loading packages f;

Output: Optimal Solution S

Find the closest customers to the depot.
Use Table 1 to determine the points of entrance s and exit t in the grid.
Determine the structure of the optimal solution (whether $f \leq 1/w - 1/d$ or $f \geq 1/w - 1/d$).
Determine the reloading points and sets served for the delivery person along the Hamiltonian path based on the structure of the solution.

Here the existence of Hamiltonian paths between s and t through the grid in the CAVADP is shown in table 1.

n	Cases	s	
Even	N/A	c_1	$c_2 \in \mathscr{C}_2$[a]
Odd	c_1 is the majority color	c_1	$c_3 \in \mathscr{C}_3$
	c_1 is not the majority color	$c_2 \in \mathscr{C}_2$	$c_2' \in \mathscr{C}_2$ such that $c_2' \neq c_2$

Table 1: Existence of Hamiltonian Paths Between s and t Through the Grid in the CAVADP

Note that $c_1 = (i_1, j_1)$ is the closest customer to the depot, $\mathscr{C}_2$ is the set of second closest customers to the depot and can include the following forms:

$$\mathscr{C}_2 \subset \{(i_1, j_1+1), (i_1, j_1-1), (i_1+1, j_1), (i_1-1, j_1)\} \tag{1}$$

The set of third closest customers to the depot $\mathscr{C}_3$ can include the following forms:

$$\begin{aligned} \mathscr{C}_3 \subset \{ & (i_1-1, j_1+1), (i_1-1, j_1-1), (i_1+1, j_1-1) \\ & (i_1+1, j_1+1), (i_1-2, j_1), (i_1+2, j_1) \\ & (i_1, j_1-2), (i_1, j_1+2)\} \end{aligned} \tag{2}$$

2.2 Finding Hamiltonion path

The key part of the distinct solution method is to find a Hamiltonian path between the predefined starting point s and ending point t. The codes are implemented in Python. Here are some testing cases:

- We first tested a general 6×6 grid. Here both the starting and ending points are located on the edge of the grid. The path could be detected in seconds as shown in Figure 1.

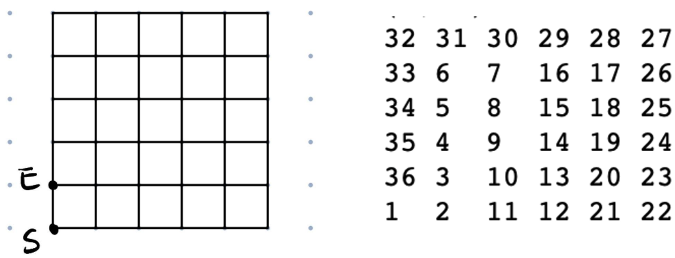


Figure 1: General 6×6 case.

- We then tested several scenarios shown in Table 1. First, when n is even, let S lies at c_1 and E at any of the adjacent points. Then, according to the Table 1, we could find a path. The experiment also showed the found path in Figure 2. Notice that if the grid is of size $2 \times g$, $g \times 2$, $3 \times g$, or $g \times 3$ for some $g \in \mathbb{N}$, c_2 should be chosen such that $c_1 c_2$ is a boundary edge. We found that in the 2×4 grid, if $c_1 c_2$ is not chosen as a boundary edge, there is no Hamiltonian path on the grid.

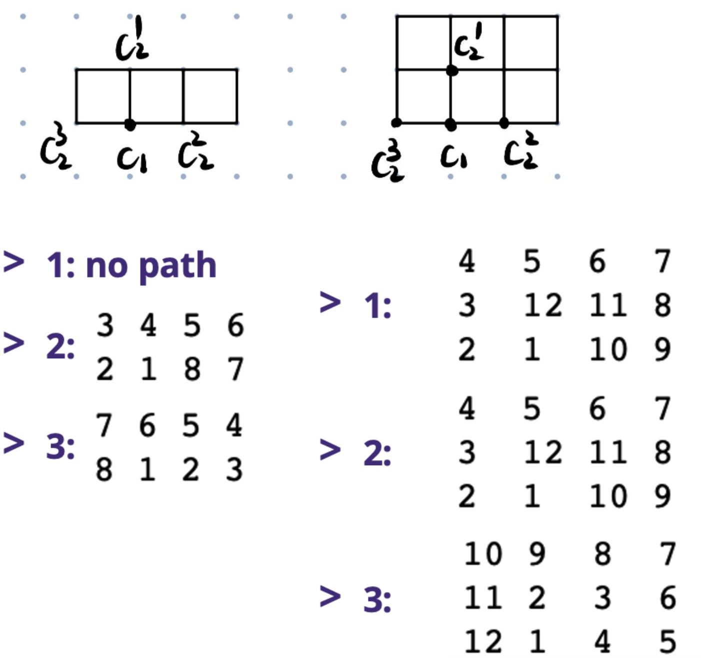
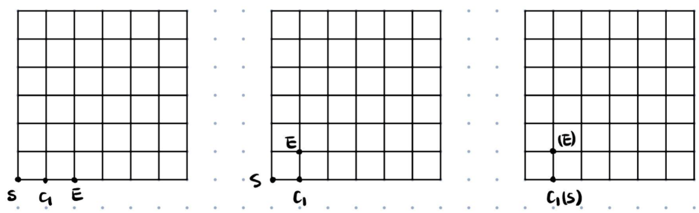


Figure 2: 2×4 case and 3×4 case.

- Second, when n is odd, and c_1 is not the majority color, the s and t should be found in the set $\mathscr{C}_2$. See the first and second cases in Figure 3. For the third case, the starting point is located in c_1, then it doesn't provide any possible path on the grid.



7	8	17	18	27	28	29
6	9	16	19	26	31	30
5	10	15	20	25	32	33
4	11	14	21	24	35	34
3	12	13	22	23	36	37
2	47	46	45	42	41	38
1	48	49	44	43	40	39

7	8	17	18	27	28	29
6	9	16	19	26	31	30
5	10	15	20	25	32	33
4	11	14	21	24	35	34
3	12	13	22	23	36	37
2	49	46	45	42	41	38
1	48	47	44	43	40	39

> no solution

Figure 3: 7×7 case.

- In addition, we also considered adding some blocks on the grid. This is to simulate various traffic environment in different city, and see whether it is possible to develop the similar distinct solution method when changing the strict rectangular grid assumption. Figure 4 and 5 are showing cases where Hamiltonian path is successfully found and no Hamiltonian path is found separately. We could hardly find any deterministic rules that can tell under which circumstances a Hamiltonian path could be successfully found. Thus, in the next part, we stopped the research of CAVADP model under rectangular grid assumption, and revisited the Mixed-integer programming formulation.

```
....... 3  4  5  24 25 26 27
.X..... 2  X  6  23 18 17 28
SE..... 1  46 7  22 19 16 29
....... 44 45 8  21 20 15 30
...XX.. 43 42 9  X  X  14 31
....... 40 41 10 11 12 13 32
....... 39 38 37 36 35 34 33
```

```
....... 3  4  7  8  15 16 17
....... 2  5  6  9  14 19 18
SE....X 1  46 45 10 13 20 X
....... 42 43 44 11 12 21 22
...XX.. 41 36 35 X  X  24 23
....... 40 37 34 31 30 25 26
....... 39 38 33 32 29 28 27
```

Figure 4: Case with blocks successfully finding Hamiltonian path.

```
.......
.......
SE.....
.......
...XX..
X......
.......
```

> no solution

Figure 5: Case with blocks finding no Hamiltonian path.

3 Urban-to-Rural

One limitation of the paper we refer to is the urban area restriction as well as the gird positions of the customers. So we are interested in the performance of the CAVADP under the urban-to-rural setting. So in this section, the grid assumption no longer stands and we revisit and modify the mixed-integer programming formulation following the new CAVADP paper published in April, 2022. We try to solve the optimization problem using *Gurobi Python*, considered adding valid inequalities and explored the visualization of vehicle routes on real maps using the package *VeRoViz* in Python.

3.1 Settings

3.1.1 Real-World Data

Since the rural area and the urban area are different, we cannot use a gird graph to represent the customer geographies in this setting. So here what the model uses is a general graph. Each location has real latitude and longitude coordinates. The customer locations are generated randomly in a square service region, and they are all close to the road. We plot several customer locations on the map in Figure 1. The red point is the depot, and all blue points are customers.

Figure 6: Depot location and some customer locations

Correspondingly, the model also use real driving times and walking times between locations. The real world obstacles are also considered. For example, there are some one-way roads in reality. This will lead to some instances that the walking delivery person have to wait for the vehicle.

3.1.2 Urban-to-Rural Code

The U.S. Department of Agriculture has given each county in America an urban-to-rural code to show its urban-to-rural level in 2013. The code ranges from 1 to 9, meaning urban to rural. It is based on population size and adjacency to metro areas. To compare the performance of the model in areas with different urban-to-rural levels, for each level we can choose one county and then generate the customer data set separately in each county.

3.2 Optimization Model

In the model, there are three series of decision variables. Table 1 is the description of the decision variables. $\bar{C}$ is the set of customers and the depot, C is the set of customers, and S is the set of possible service sets. Here we use the package *VeRoViz* to plot an example in order to show the meanings of the decision variables clearly. In the example, the depot is the red point 0, and the customers are the blue points. The

Notation	Description
x_{ik}	x_{ik} = 1 if the vehicle drives from location i to location k with the delivery person on board for $i,k \in \overline{C}$
$y_{i\sigma k}$	$y_{i\sigma k} = 1$ if the delivery person loads at customer i, serves set σ, and meets the vehicle at customer k for $i,k \in C$ and $\sigma \in S$
v_{ik}	The flow of packages on board the vehicle from location i to location k for $i \in \overline{C}$ and $k \in C$ such that $i \neq k$

Figure 7: Decision Variables of CAVADP

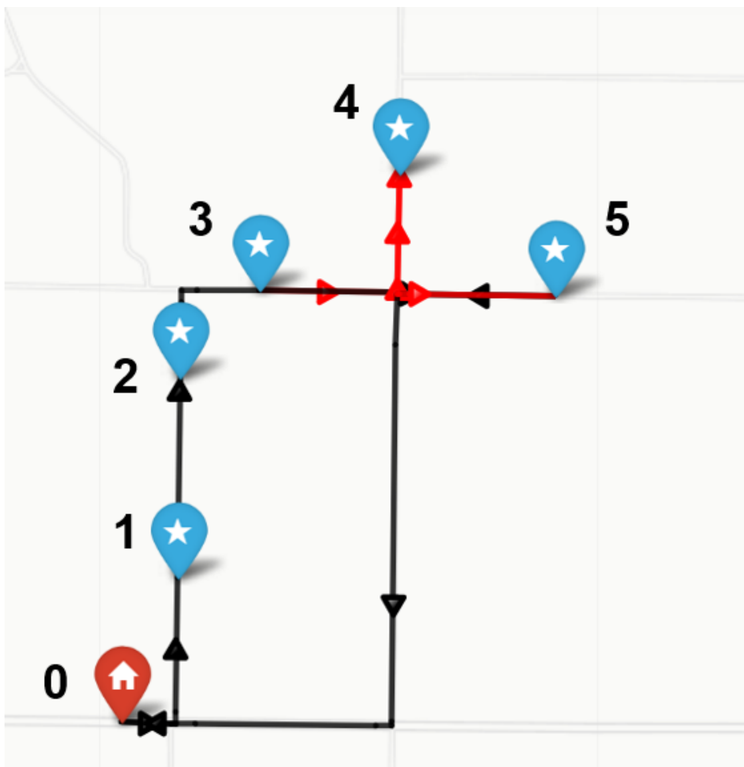


Figure 8: A Route Example

route of the vehicle is 0–1–2–3–5–0, and the walking route of the delivery person is 3–4–5. So some of the decision variables are:

$$C = \{1,2,3,4,5\}$$
$$\bar{C} = \{0,1,2,3,4,5\}$$
$$x_{01} = x_{12} = x_{23} = x_{50} = 1$$
$$x_{35} = x_{45} = 0$$
$$y_{3,(3,4,5),5} = 1$$
$$v_{01} = 5, v_{12} = 4, v_{34} = 0$$

A serious problem is that the number of $y_{i\sigma k}$ is so large that the model will need extreme long time to get the solution. When the size of C is n, the size of S will be $m = \sum_{i=1}^{q} \binom{n}{i}$, and the number of possible y variables will be $n^2 m$, which is large in real instances.
To deal with this problem in the integer program, besides the constraints, there are six claims helping reduce the number of decision variables. Four of them are the prepossessing techniques, and the last two are valid inequalities. In the paper, the authors use the experiment outputs to show whether they are effective. With all the prepossessing techniques in model, the service set that need to be considered is only about 2% of the possible service sets. With all the valid inequalities, the run time in reduced by 49% on average. These claims are quite meaningful in enhancing the ability of solving CAVADP with a general graph in real instances.

3.3 Implementation

We try to implement the model with Gurobi. There are an objective function, twelve constraints, and six claims in total. We write them all in Python, and the codes can be found in Appendix. However, we find out that the model is infeasible.
After hours of debugging, we find out some small errors or typos in the model provided in the paper. Firstly, in constraint (10), the first term of the right-hand side should be $\sum_{k \in C \setminus \{i\}} x_{ik}$, but not $\sum_{k \in C} x_{ik}$. Besides, the last constraint should be

$$v_{ik} \in \mathbb{N}, \quad \forall i \in \bar{C}, k \in C \quad s.t. i \neq k$$

After correcting the two points, there are still something wrong. We find out that without the last constraint and claim 2, which means that we only put the first eleven constraints, claim 1, and claim 3 to claim 6 into the model, the model is feasible. But if we add any one of the two into the model, the MIP will become infeasible.
We also sent the first author, Sara Reed, an email to talk about the problem we met. We fortunately received her reply. She confirms that we are correct about the two errors, and acknowledges that there may be other small typos.

4 Conclusion

In our project, we aimed to relax, test several important assumptions of the CAVADP model. Here is a summary of the project:

- We realized the distinct solution method (especially calculating the Hamiltonian path).
- We tested the key theoretical results in the Management Science paper reviewed before
- We extended the urban settings (grid) to rural settings, where grid assumption no longer stands and revisited the mixed-integer programming formulation
- We tried to solve the optimization problem using *Gurobi* Python, considered adding valid inequalities and explored the visualization of vehicle routes on real maps using *VeRoViz* package in Python

REFERENCES

Allen, J., M. Piecyk, M. Piotrowska, F. McLeod, T. Cherrett, K. Ghali, T. Nguyen, T. Bektas, O. Bates, A. Friday et al. 2018. "Understanding the impact of e-commerce on last-mile light goods vehicle activity in urban areas: The case of London". *Transportation Research Part D: Transport and Environment* 61:325–338.

Cookson, G., and B. Pishue. 2017. "The impact of parking pain in the US, UK and Germany". *Hg. v. INRIX Research. Online verfügbar unter http://inrix. com/research/parking-pain/, zuletzt geprüft am* 21:2018.

Reed, S., A. M. Campbell, and B. W. Thomas. 2022a. "Impact of Autonomous Vehicle Assisted Last-Mile Delivery in Urban to Rural Settings". *Transportation Science*.

Reed, S., A. M. Campbell, and B. W. Thomas. 2022b. "The value of autonomous vehicles for last-mile deliveries in urban environments". *Management Science* 68(1):280–299.

APPENDIX

Code for Generating Hamiltanion Path:

```
import time

EMPTY_SPACE_SYMBOLS = '.'
STARTING_POINT_SYMBOLS = 'Ss'
ENDING_POINT_SYMBOLS = 'Ee'
OBSTACLE_SYMBOL = 'X'
DIRS = [(-1, 0), (1, 0), (0, 1), (0, -1)]

class HamiltonSolver:
    """Solver for a Hamilton Path problem."""

    def __init__(self, grid):
        """Initialize the HamiltonSolver instance from a grid, which must be a
        list of strings, one for each row of the grid.

        """
        self.grid = grid
        self.h = h = len(grid)
        self.w = w = len(grid[0])

        if any(len(row) != w for row in grid):
            raise ValueError("Grid is not rectangular")


        self.start = None
        self.end = None
        self.legal = set()

        for r, row in enumerate(grid):
            for c, item in enumerate(row):
                if item in STARTING_POINT_SYMBOLS:
                    if self.start is not None:
                        raise ValueError("Multiple starting points")
                    self.start = (r, c)

                elif item in EMPTY_SPACE_SYMBOLS:
                    self.legal.add((r, c))

                elif item in ENDING_POINT_SYMBOLS:
                    if self.end is not None:
                        raise ValueError("Multiple ending points")
                    self.end = (r, c)
```

```python
                    self.legal.add((r, c))
        if self.start is None:
            raise ValueError("No starting point")

    def format_solution(self, path):
        """Format a path as a string."""
        grid = [[OBSTACLE_SYMBOL] * self.w for _ in range(self.h)]
        for i, (r, c) in enumerate(path, start=1):
            grid[r][c] = i
        w = len(str(len(path) + 1)) + 1
        return '\n'.join(''.join(str(item).ljust(w) for item in row)
                         for row in grid)

    def solve(self):
        """Generate solutions as lists of coordinates."""
        path = [self.start]
        dirs = [iter(DIRS)]

        # Cache attribute lookups in local variables
        path_append = path.append
        path_pop = path.pop
        legal = self.legal
        legal_add = legal.add
        legal_remove = legal.remove
        dirs_append = dirs.append
        dirs_pop = dirs.pop

        while path:
            r, c = path[-1]
            for dr, dc in dirs[-1]:
                new_coord = r + dr, c + dc
                if new_coord in legal:
                    path_append(new_coord)
                    legal_remove(new_coord)
                    dirs_append(iter(DIRS))
                    if not legal:
                        yield path
                    break
            else:
                legal_add(path_pop())
                dirs_pop()



def main(PUZZLE_GRID):
    start_time = time.time()

    puzzle = HamiltonSolver(PUZZLE_GRID)

    end = puzzle.end
    print(end)
    for solution in puzzle.solve():
        if solution[-1] == end:
            print(puzzle.format_solution(solution))
            print("Solution with assigned starting and ending points found in {} s".format(time.time() - start_time))
            break
```

```


test_example_1 = '''
......
......
......
......
E.....
S.....
'''.split()

# 2 \times 4
test_example_2 = '''
....
.SE.
'''.split()

# 2 \times 4
test_example_3 = '''
.E..
.S..
'''.split()

# 2 \times 4
test_example_4 = '''
....
ES..
'''.split()

# 3 \times 4
test_example_5 = '''
....
....
.SE.
'''.split()

# 3 \times 4
test_example_6 = '''
....
.E..
.S..
'''.split()

# 3 \times 4
test_example_6 = '''
....
....
ES..
'''.split()

# 7 \times 7
test_example_7 = '''
.......
.......
.......
.......
.......
```

```
.E.....
S......
'''.split()

# 7 \times 7
test_example_8 = '''
.......
.......
.......
.......
.......
.......
S.E....
'''.split()

# 7 \times 7
test_example_9 = '''
.......
.......
.......
.......
.......
.E.....
.S.....
'''.split()



if __name__ == '__main__':
    main(test_example_1)
```

Code for Urban-to-Rural Integer Program(include claims):

```
from gurobipy import *
import pandas as pd
import math
import numpy as np
def read_data():
    drive_data = pd.read_csv('Adams_100_1_Fastest.csv')

    drive_distance = {}
    drive_time = {}
    arc = drive_data['key'].values
    distance = drive_data['Distance'].values
    time = drive_data['Time'].values

    n = len(arc)
    for i in range(0,n):
        a = re.sub("\(" , "", arc[i])
        a = re.sub("\)", "", a)
        a,b = a.split(', ')
        drive_distance[(int(a),int(b))]= distance[i]
        drive_time[(int(a),int(b))] = time[i]

    walk_data = pd.read_csv('Adams_100_1_Walking.csv')

    walk_distance = {}
    walk_time = {}
```

```
    arc = walk_data['key'].values
    distance = walk_data['Distance'].values
    time = walk_data['Time'].values
    n = len(arc)
    for i in range(0,n):
        a = re.sub("\(" , "", arc[i])
        a = re.sub("\)", "", a)
        a,b = a.split(', ')
        walk_distance[(int(a),int(b))]= distance[i]
        walk_time[(int(a),int(b))] = time[i]

    return int(math.sqrt(n)), drive_distance, drive_time, walk_distance, walk_time

f = 2.8 # time for loading packages
q = 3 # capacity

# generate set
def serve_set(n):
    serve_set = set(())
    for i in range(1, n + 1):
        serve_set.add((i,))
        for j in range(i + 1, n + 1):
            serve_set.add((i, j)) # tuple
            for k in range(j + 1, n + 1):
                serve_set.add((i, j, k))
    m = len(serve_set)  # m: number of serve sets
    return m, serve_set

# Testing whether to add the equalities

# for Claim 5,6
# D(i,k) <= W(i,k) for all i,k in C
def D_less_equal_than_W(drive_time, walk_time, n):
    temp = True
    for i in range(1, n+1):
        for k in range(1, n+1):
            if drive_time[(i, k)]>walk_time[(i,k)]:
                temp = False
                break
        if not temp:
            break
    return temp

# for Corollary 1
# Consider i in C. If D(i, k) + f <= W(i, k) and D(k,i)+f <= W(k,i) for all k in C\{i}
def D_plus_f_less_equal_than_W(i, drive_time, walk_time, f, n):
    temp = True
    for k in range(1, n+1):
        if k != i:
            if drive_time[(i,k)] + f > walk_time[(i,k)]:
                temp = False
                break
            elif drive_time[(k,i)] + f > walk_time[(k,i)]:
                temp = False
                break
    return temp

```

```python
# for Claim 2
#    in S such that D(i, k)    W(i, k) and D(k,i) W (k,i) for all i in    and k in C
def D_less_equal_than_W_for_i_in_sigma(sigma, drive_time, walk_time, n):
    temp = True
    for i in sigma:
        for k in range(1, n+1):
            if drive_time[(i, k)] > walk_time[(i,k)]:
                temp = False
            elif drive_time[(k, i)] > walk_time[(k,i)]:
                temp = False
    return temp

# n: number of costumers, not include depot
n, drive_distance, drive_time, walk_distance, walk_time = read_data()

# serve set
m, serve_set = serve_set(n)  # m: |S|, length of serve_set
serve_list = list(serve_set)

#  w_i_sigma_k
w_with_set = w_with_set(serve_list, walk_time, n)

# max term in obj function
wait_time = compare_d_w(w_with_set, drive_time)

model = Model("mip")
model.setParam('Timelimit', 3600)

# add variables
x = {}
for i in range(n+1):
    for k in range(n+1):
        name = 'x_' + str(i) + '_' + str(k)
        if i == k:
            x[i,k] = model.addVar(0,0, vtype=GRB.INTEGER, name=name)
        else:
            x[i,k] = model.addVar(0,1, vtype=GRB.BINARY, name=name)

y = {}
for i in range(1,n+1):
    for s_id in range(m):
        for k in range(1,n+1):
            name = 'y_' + str(i) + '_' + str(serve_list[s_id]) + '_' + str(k)
            y[i,s_id,k] = model.addVar(0,1, vtype=GRB.BINARY, name=name)

v = {}
for i in range(n+1):
    for k in range(1,n+1):
        if i != k:
            name = 'v_' + str(i) + '_' + str(k)
            v[i,k] = model.addVar(0,n, vtype=GRB.INTEGER, name=name) # not sure
    include n

# objective function
obj = LinExpr(0)

for i in range(n+1):
```

```
    for k in range(n+1):
        if i != k:
            obj.add(x[i,k]*drive_time[(i,k)])

for i in range(1,n+1):
    for k in range(1,n+1):
        for sigma in serve_list:
            index = serve_list.index(sigma)
            obj.add(y[i,index,k] * (f + w_with_set[(i, index, k)] + wait_time[(i,
    index, k)]))

model.setObjective(obj, GRB.MINIMIZE)

# Constraints

# Constraint 2
obj_c_2 = LinExpr(0)
for i in range(1,n+1):
    obj_c_2.add(x[i,0])
model.addConstr(obj_c_2 == 1,  name= 'Constraint 2')

# Constraint 3
obj_c_3 = LinExpr(0)
for i in range(1,n+1):
    obj_c_3.add(x[0,i])
model.addConstr(obj_c_3 == 1,  name= 'Constraint 3')


# Constraint 4
for l in range(1,n+1):
    obj_c_4 = LinExpr(0)
    for i in range(1, n + 1):
        for k in range(1, n + 1):
            for sigma in serve_list:
                if l in sigma:
                    index = serve_list.index(sigma)
                    obj_c_4.add(y[i,index,k])
    model.addConstr(obj_c_4 == 1,  name= 'Constraint 4_'+str(l))

# Constraint 5
for i in range(1,n+1):
    obj_c_5_1 = LinExpr(0)
    for k in range(1, n + 1):
        for s_id in range(len(serve_list)):
            obj_c_5_1.add(y[i,s_id,k])
    obj_c_5_2 = LinExpr(0)
    for l in range(n + 1):
        obj_c_5_2.add(x[l,i])
    model.addConstr(obj_c_5_1 == obj_c_5_2,  name= 'Constraint 5_'+str(i))

# Constraint 6
for k in range(1,n+1):
    obj_c_6_1 = LinExpr(0)
    for i in range(1, n + 1):
        for s_id in range(len(serve_list)):
            obj_c_6_1.add(y[i,s_id,k])
    obj_c_6_2 = LinExpr(0)
```

```python
    for l in range(n + 1):
        obj_c_6_2.add(x[k,l])
    model.addConstr(obj_c_6_1 == obj_c_6_2,  name= 'Constraint 6_'+str(k))
# Constraint 7
obj_c_7 = LinExpr(0)
for k in range(1,n+1):
    obj_c_7.add(v[0,k])
model.addConstr(obj_c_7 == n,  name= 'Constraint 7')

# Constraint 8
for i in range(1,n+1):
    model.addConstr(v[0,i] <= n * x[0,i],  name= 'Constraint 8_'+str(i))

# Constraint 9
for i in range(1,n+1):
    for k in range(1, n + 1):
        if i != k:
            obj_c_9 = LinExpr(0)
            for s_id in range(len(serve_list)):
                obj_c_9.add(y[i,s_id,k])
            model.addConstr((obj_c_9 + x[i,k]) * n >= v[i,k], name='Constraint 9_'+str(i)+'_'+str(k))
for sigma in serve_list:
    if len(sigma) == 1:
        i = int(sigma[0])
        model.addConstr(y[i,serve_list.index(sigma),i] == 0, name='y' + str(i) + '_' + str(i))

# Constraint 10
for i in range(1,n+1):
    obj_c_10 = LinExpr(0)
    # first term
    for k in range(n+1):
        if i != k:
            obj_c_10.add(v[k, i])
    # second term
    for k in range(1,n+1):
        if i != k:
            obj_c_10.add(-v[i, k])
    # third term
    for k in range(1,n+1):
        obj_c_10.add(-x[i, k])
    # forth term
    for k in range(1,n+1):
        for sigma in serve_list:
            index = serve_list.index(sigma)
            obj_c_10.add(-(len(sigma)-1)*y[i,index,k])
    model.addConstr(obj_c_10 == 0, name='Constraint 10_'+str(i))
# Service Set Reduction

# Corollary 1
for i in range(1,n+1):
    if D_plus_f_less_equal_than_W(i, drive_time, walk_time, f, n):
        obj_claim_1_ki = LinExpr(0)
        obj_claim_1_ik = LinExpr(0)
        for k in range(n+1):
            obj_claim_1_ki.add(x[k,i])
```

```python
            obj_claim_1_ik.add(x[i,k])
        model.addConstr( obj_claim_1_ki == 1, name='Claim 1_ki_'+str(i))
        model.addConstr( obj_claim_1_ik == 1, name='Claim 1_ik_'+str(i))


# Variable Reduction

# Claim 2
for sigma in serve_list:
    index = serve_list.index(sigma)
    if D_less_equal_than_W_for_i_in_sigma(sigma, drive_time, walk_time, n):
        for i in range(1,n+1):
            for k in range(1,n+1):
                if (i not in sigma) or (k not in sigma):
                    model.addConstr( y[i,index,k] == 0, name='Claim 2_'+str(i) +str(k) +str(index))


# Claim 3

for sigma in serve_list:
    index = serve_list.index(sigma)
    for i in range(1,n+1):
        if i not in sigma:
            for k in range(1,n+1):
                min_w, c_first, c_last = find_shortest_time(i, sigma, k)
                if (walk_time[(i,c_first)] - drive_time[(i,c_first)]) >= wait_time[(c_first, index, k)]:
                    model.addConstr( y[i,index,k] == 0, name='Claim 3_'+str(i) +str(k) +str(index))

# Claim 4

for sigma in serve_list:
    index = serve_list.index(sigma)
    for i in range(1,n+1):
        for k in range(1,n+1):
            min_w, c_first, c_last = find_shortest_time(i, sigma, k)
            if (walk_time[(c_last,k)] - drive_time[(c_last,k)]) >= wait_time[(i, index, c_last)]:
                model.addConstr( y[i,index,k] == 0, name='Claim 4_'+str(i) +str(k)+str(index))

# Added valid inequalities
# Only for D(i,k)<=W(i,k)

if D_less_equal_than_W(drive_time, walk_time, n):

    # Claim 5
    for i in range(1,n+1):
        for k in range(1, n+1):
            if i != k:
                obj_claim_5 = LinExpr(0)

                J_ik_idx = [ idx for idx in range(len(serve_list)) if (i in serve_list[idx]) and (k in serve_list[idx]) ]

```

```
                for s_id in J_ik_idx:
                    for a in range(1,n+1):
                        for b in range(1, n+1):
                            obj_claim_5.add(y[a,s_id,b])
                model.addConstr( obj_claim_5 + x[i,k] <= 1, name='Claim 5_'+str(i)+'_'+str(k))

    # Claim 6
    for i in range(1,n+1):
        for k in range(1, n+1):
            if i < k:
                obj_claim_6 = LinExpr(0)
                for s_id in range(len(serve_list)):
                    obj_claim_6.add(y[k,s_id,i] + y[i,s_id,k])
                model.addConstr( obj_claim_6 + x[i,k] + x[k,i] <= 1, name='Claim 6_'+str(i)+'_'+str(k))

model.optimize()

print("\n\n optimal value:")
print(model.ObjVal)
```

Code for the Plots on Map:

```
import veroviz as vrv
import pandas as pd
node_data = pd.read_csv('Adams_100_1.csv')
depot = [node_data.lat[0],node_data.lon[0]]
node = []
for i in range(1,30):
    node.append([node_data.lat[i],node_data.lon[i]])
nodes2D = vrv.createNodesFromLocs(
     locs=[depot], nodeType        = 'depot',
                            leafletColor     = 'red',
                            leafletIconType  = 'home')
nodes2D = vrv.createNodesFromLocs(
     locs=
         node
         ,initNodes       = nodes2D, nodeType         = 'customer',
                            leafletColor     = 'blue',nodeName = 'C',leafletIconType = 'star', incrementName = True,startNode         = 0)
vrv.createLeaflet(nodes=nodes2D)
node = []
node.append([node_data.lat[24],node_data.lon[24]])
node.append([node_data.lat[19],node_data.lon[19]])
node.append([node_data.lat[16],node_data.lon[16]])
node.append([node_data.lat[6],node_data.lon[6]])
node.append([node_data.lat[28],node_data.lon[28]])
nodes2D = vrv.createNodesFromLocs(
     locs=[depot], nodeType        = 'depot',
                            leafletColor     = 'red',
                            leafletIconType  = 'home')
nodes2D = vrv.createNodesFromLocs(
     locs=
         node
         ,initNodes       = nodes2D, nodeType         = 'customer',
                            leafletColor     = 'blue',nodeName = 'C',leafletIconType = 'star', incrementName = True,startNode         = 0)
```

```python
vrv.createLeaflet(nodes=nodes2D)

exampleAssignments = vrv.createAssignmentsFromLocSeq2D(
                        locSeq          = [depot,  node[0],node[1],node[3],depot],
                        serviceTimeSec  = 0.0,
                        objectID        = 'Truck',
                        routeType       = 'fastest',
                        dataProvider    = 'ORS-online',
    cesiumWeight = 2,
                        leafletColor    = 'black',dataProviderArgs = {'APIkey': ORS_API_KEY})
exampleAssignments = vrv.createAssignmentsFromLocSeq2D(
                        initAssignments = exampleAssignments,
                        locSeq          = [node[1],node[2],node[3]],
                        serviceTimeSec  = 0.0,
                        objectID        = 'Truck',
                        routeType       = 'pedestrian',
                        dataProvider    = 'ORS-online',
     cesiumWeight = 4,
                        leafletColor    = 'red',dataProviderArgs = {'APIkey': ORS_API_KEY})
vrv.createLeaflet(nodes = nodes2D, arcs = exampleAssignments)
```