Topic 1

February 2023

Fundamental of Optimization

Nguyễn Quang Pháp
Nguyễn Thanh Bình
Bùi Anh Nhật

Table of Contents

I Problem description

The problem is about collecting packages at N points (1, 2, ..., N) using K mailmen starting from the post office (point 0). Given the distance d(i, j) between each pair of points (i, j), with i, j = 0, 1, ..., N, the task is to build a plan for the K mailmen to collect packages, determining which points each man should collect and in what order to minimze:

- The total distance traveled by all mailmen.
- The longest distance traveled by a mailmen.

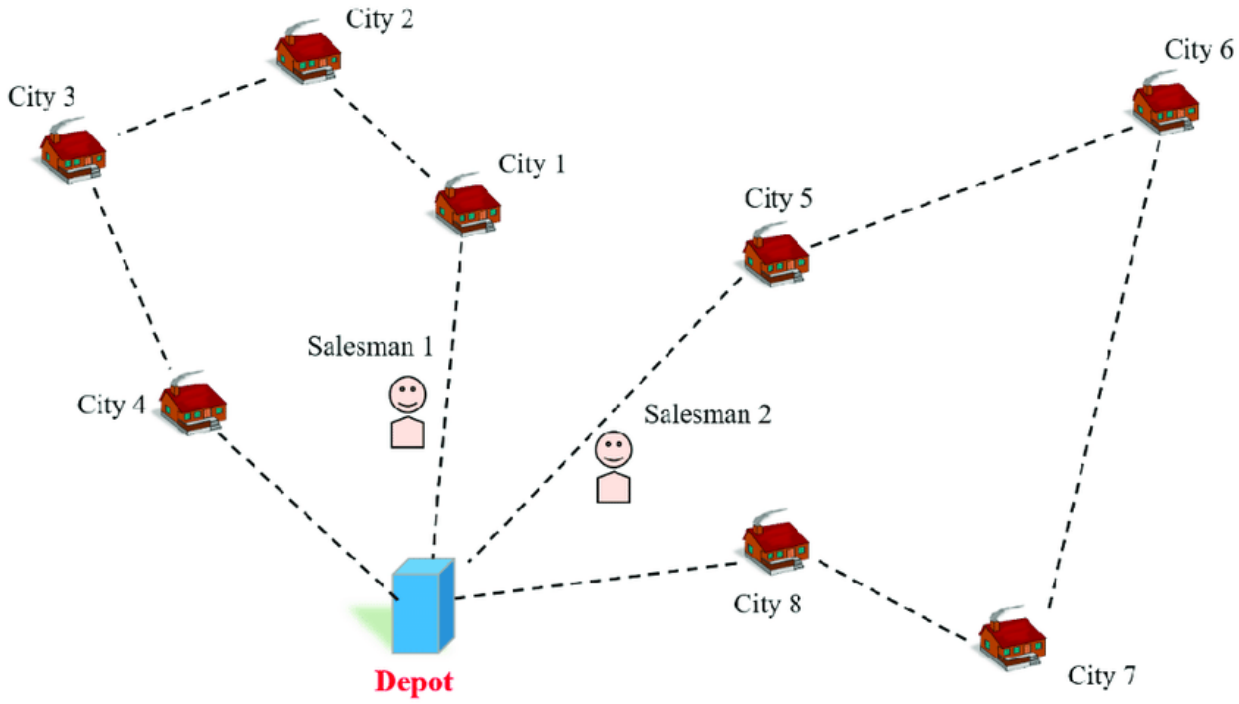

Fig. 1: Problem description

II Modeling

Notations

Mailmen $k(k = 1, 2, ..., K)$ departs from point $N + k$ and terminates at point $N + K + k(N + k$ and $N + K + k$ refer to the central depot 0).

- $B = \{1, 2, ..., N + 2K\}$: set of all points.
- $F_1 = \{(i, N + k)|i \in B, k \in \{1, 2, ..., K\}\}$: set of edges that go back to the starting points.
- $F_2 = \{(N + K + k, i)|i \in B, k \in \{1, 2, ..., K\}\}$: set of edges that start from terminating points.
- $F_3 = \{(i, i)|i \in B\}$
- $F_4 = \{(i, j)|i \in \{N + 1, ..., N + K\}, j \in \{N + K + 1, ..., N + 2K\}\}$: set of edges that go directly from starting points to terminating points.
- $A = B^2 \backslash F_1 \backslash F_2 \backslash F_3 \backslash F_4$: set of edges that can be chosen.
- $A^+(i) = \{j|(i, j) \in A\}\ \forall i \in B$: set of points that can go from point i
- $A^-(i) = \{j|(j, i) \in A\}\ \forall i \in B$: set of points that can go to point i

Variables

$$X(k, i, j) = \begin{cases} 1, \text{if mailman k travels from } i \text{ to } j \\ 0, \text{otherwise} \end{cases}$$

$U(k, i) =$ the rank of point i visited by mailman k.

Constraints

- $\sum_{j=1}^{N} X(k, N + k, j) = \sum_{j=1}^{N} X(k, j, N + K + k) = 1\ \forall k \in \{1, 2, ..., K\}$

  This constraint guarantees that mailman k starts from point N+k and terminates at point N+K+k only once.

- $X(k, i, j) = 0\ \forall j \in \{1, 2, ..., N\}\ \forall i \in \{N + 1, N + 2, ..., N + K\} \backslash \{N + k\}$

- $X(k,i,j) = 0 \ \forall i \in \{1,2,...,N\} \ \forall j \in \{N+K+1,...,N+2K\} \backslash \{N+K+k\}$

  These two constraints guarantee that mailman k can only starts at point N+k and terminates at point N+K+k.

- $\sum_{k=1}^{K} \sum_{j \in A^+(i)} X(k,i,j) = \sum_{k=1}^{K} \sum_{j \in A^-(i)} X(k,j,i) = 1 \ \forall i \in \{1,2,...,N\}$

- $\sum_{j \in A^+(i)} X(k,i,j) = \sum_{j \in A^-(i)} X(k,j,i) \ \forall i \in \{1,2,...,N\} \ \forall k \in \{1,2,...,K\}$

  These two constraints guarantee that only one mailman visits a point in $\{1,2,...,N\}$ and exactly that mailman departs that point. These also guarantee that all point 1, 2, ..., N will be visited.

- $U(k,i) = 0 \ \forall i \in \{N+1,...,N+2K\} \ \forall k \in \{1,...,K\}$

- $X(k,i,j) = 1 \Rightarrow U(k,j) = U(k,i) + 1 \ \forall k \in \{1,...,K\} \ \forall i,j \in \{1,2,...,N\}$

- $U(k,i) - U(k,j) + N * X(k,i,j) \leq N - 1 \ \forall k \in \{1,...,K\} \ \forall i,j \in \{1,2,...,N\}; i \neq j$

  This is Miller-Tucker-Zemlin subtour elimination constraint. These three constraints guarantee that there is no subtour or partial tour in a solution.

Objective function

The function represent the total distance traveled by all mailmen is:

$$f_1 = \sum_{k=1}^{K} \sum_{(i,j) \in A} d(i,j) \times X(k,i,j)$$

The function represent the longest distance traveled by a mailman is:

$$f_2 = \max_{k \in \{1,2,...,K\}} \sum_{(i,j) \in A} d(i,j) \times X(k,i,j)$$

Our mission is to minimize the two functions, so we propose an objective function that can represent both functions above:

$$f = f_1 + K \times f_2$$

where K is the number of mailmen.

Our job now is to optimize(minimize) the objective function.

III Algorithms used

III.1 Brute Force with Branch Cutting

Overview

A brute force algorithm for this problem is an algorithm that generates all possible $X(k, i, j)$ such that $(i, j) \in A$, and selects the one that satisfies all above constraints and gives the result that minimizes the objective function.

For this algorithm, we add another constraint to help the algorithm runs faster.

- $X(k, i, j) = 0 \ \forall (i, j) \notin A$

This constraint prevent the algorithm from generating invalid edges that do not belong to the valid edges set A.

Properties

- Completeness: The brute force algorithm guarantees to find the optimal solution, if it is run to completion.
- Running time: The running time of the brute force algorithm is exponential in the number of cities, making it infeasible for large instances of the problem.
- Determinism: The brute force algorithm is deterministic, meaning that it produces the same result each time it is run with the same input.
- Efficiency: The brute force algorithm is inefficient, as it generates and evaluates a large number of candidate solutions, most of which are not optimal.

Brute Force Extension

As the time complexity of this algorithm is exponential, we come up with an idea to reduce it, which is similar to the idea of Branch-and-Bound algorithm applied to TSP: use a lower bound (i.e., a bound on the minimum distance traveled by all salesmen f_1 that can be achieved) to prune suboptimal solutions, and to generate new candidate solutions by branching on the current solution. This may prune the optimal solution as it only consider the total distance traveled f_1 but much efficient than the original Brute Force algorithm. The running time of this algorithm is different for different input.

Pseudo Code

```
function Try(k, i, j)
    f1_current = 0
    if ((i,j) in A) and ((i<=N) or (i=N+k)) and ((j<=N) or (j=N+K+k)):
    #this condition reflect the 2nd and 3rd constraints
        for v in [0, 1]:
            if v = 0:
                X(k,i,j) = 0
                if (k = K) and (i = N+K) and (j = N):
                  if check_constraints():
                    solution()
                else if (i = N+K) and (j = N):
                  Try(k+1,1,1)
                else if (j = N+2K):
                  Try(k,i+1,1)
                else:
                  Try(k,i,j+1)
            else if v = 1:
                X(k,i,j) = 1
                calculate_f_current()
                if f1_current+d_min*remain_edge <= f1_min:
                #f1_min is the minimum total distance of
                #all mailmen that has been found so far
                    if (k = K) and (i = N+K) and (j = N):
                      if check_constraints():
                        solution()
                    else if (i = N+K) and (j = N):
                      Try(k+1,1,1)
                    else if (c = N+2K):
                      Try(k,i+1,1)
                    else:
                      Try(k,i,j+1)
                calculate_f_current()
     else:
        X(k,i,j) = 0
        if (k = K) and (i = N+K) and (j = N):
          if check_constraints():
            solution()
        else if (i = N+K) and (j = N):
          Try(k+1,1,1)
```

```
else if (j = N+2K):
  Try(k, i+1, 1)
else:
  Try(k, i, j+1)
```

III.2 Backtracking and Branch-and-Bound

Overview

The Backtracking algorithm is a depth-first search algorithm that can be used to solve this problem by systematically generating and testing all possible solutions. The basic idea of the backtracking algorithm is to construct a solution incrementally and backtrack (i.e., undo) the current decision if it leads to an invalid solution.

Similar to Brute Force algorithm, we add another constraint to help the algorithm runs faster.

- $X(k, i, j) = 0 \ \forall (i, j) \notin A$

Properties

- Completeness: The backtracking algorithm is complete, meaning that it will find an optimal solution if one exists.
- Running time: High, as it may have to consider many suboptimal solutions before finding the optimal solution.
- Determinism: The backtracking algorithm is deterministic, meaning that it produces the same result each time it is run with the same input.
- Efficiency: The backtracking algorithm is inefficient, as although it backtrack the current decision if it leads to an invalid solution, the amount of time it takes to check constraints at each time it generates is too much(even more than Brute Force algorithm, see in Result Analyst).

Branch-and-Bound

Similar to the idea of Brute Force Extension: use a lower bound (i.e., a bound on the minimum distance traveled f_1 that can be achieved) to prune non-optimal solutions, and to generate new candidate solutions by branching on the current solution. Although this may also prune the optimal solution as it consider only f_1, this help the algorithm run much faster. The running time of this algorithm is different for different input.

Pseudo code

```
function Try(k, i, j)
    f1_current = 0
    if ((i,j) in A) and ((i<=N) or (i=N+k)) and ((j<=N) or (j=N+K+k)):
    #this condition reflect the 2nd and 3rd constraints
        for v in [0, 1]:
            if check(v, X(k, i, j)) = True:
                X(k,i,j) = v
                calculate_f_current()
                if f1_current+d_min*remain_edge <= f1_min:
                #f1_min is the minimum total distance of
                #all mailmen that has been found so far
                    if (k = K) and (i = N+K) and (j = N):
                      if check_constraints():
                        solution()
                    else if (i = N+K) and (j = N):
                      Try(k+1,1,1)
                    else if (j = N+2K):
                      Try(k,i+1,1)
                    else:
                      Try(k,i,j+1)
                calculate_f_current()
    else:
        X(k,i,j) = 0
        if (k = K) and (i = N+K) and (j = N):
          if check_constraints():
            solution()
        else if (i = N+K) and (j = N):
          Try(k+1,1,1)
        else if (j = N+2K):
          Try(k,i+1,1)
        else:
          Try(k,i,j+1)
```

III.3 Greedy Algorithm

Overview

In this algorithm, each mailman starts at the central depot and visits the nearest unvisited city, adding it to his tour. The algorithm then repeats this process

for all remaining unvisited cities, until all cities have been visited by all mailmen.

Properties

- Completeness: The Greedy algorithm is incomplete, meaning that it is not guaranteed to find the optimal solution. It depends much on the distance matrix d as a single large distance in the distance matrix can significantly affect the solution quality.
- Running time: The algorithm runs very fast, making it well-suited for large inputs.
- Determinism: The Greedy algorithm is deterministic, meaning that it produces the same result each time it is run with the same input.
- Efficiency: Despite running very fast, Greedy algorithm may not efficient to find a high-quality solution.

Although the quality of the solution when we apply Greedy search is not guaranteed to be good, it can be improved by combining this algorithm with other opimization techniques which we will mention in next section.

Pseudo code

```
function Greedy():
    routes = array of k routes, each tour initially empty
    visited = array of n boolean values, initially False
    current = array of k mailmen, showing the postion of
              mailmen at present, initially all zero

    time_can_return(routes)
    #this function changes the boolean value of visited[0]
    #to determine the time a mailman can return the depot

    mailman_k,next_city = find_next_city(current,visited)
    current[mailman_k] = next_city
    add next_city to routes[mailman_k]
    if check(routes):
    #check if all mailmen have returned to the depot
        return solution
    else:
        Greedy()
```

III.4 Hill-Climbing Algorithm

Overview

The idea behind Hill-Climbing is to start with an initial solution (an initial tour for each salesman), and then repeatedly make small changes to the solution in an attempt to improve it. The main point here is Greedy algorithm will be used to make the initial solution, which is a feasible solution. The algorithm moves from the current solution to a "neighboring" solution, and if the neighboring solution is better(first neighbor) (i.e. has a lower objective function), it becomes the current solution. We define the neighboring solutions of a solution in this problem is a solution that is created by swapping 2 points on the route of a mailman or on the routes of 2 different mailmen. This process continues until a local optimum (a solution that is better than all of its neighbors) is reached.

Properties

- Completeness: Hill-climbing is not guaranteed to find the optimal solution, as it only moves to locally optimal solutions and may get stuck in local minimal.
- Running time: The algorithm runs very fast compared to other algorithms.
- Determinism: Hill-climbing is deterministic, meaning that it produces the same result each time it is run with the same input.
- Efficiency: Hill-climbing can be an efficient algorithm for our topic, as it does not need to consider all possible solutions like the brute force algorithm. Instead, it only considers neighbors of the current solution, which can be a much smaller search space. Last but not least, Hill-climbing gives us a reasonable solution in an acceptable amount of time(see in Result Analyst).

Pseudo Code

```
function find_better_neighbor(solution):
    neighbor_sol = swap(solution)
    #swap 2 points on the route of a mailman
    #or on the routes of 2 different mailmen

    if cal_obj_func(neighbor_sol) < cal_obj_func(solution):
        return(neighbor_sol)
    return
```

```
function Hill_Climbing():
    initial_sol = Greedy()
    solution = initial_sol
    while (find_better_neighbor not None):
        solution = find_better_neighbor(solution)
    return solution
```

III.5 Iterated Local Search

Overview

The main idea behind implementing an iterated local search algorithm for this problem is to repeatedly perform local search operations on a set of solutions that is generated randomly in order to find better solutions.

The algorithm stops when for each solution in the set of solutions, there is no neighbor better than it. We then check for the best solution in that set which will be our final result.

Properties

- Completeness: Iterated local search is not guaranteed to find the optimal solutions as it depends on the input initial solutions that it receive.
- Time complexity: The time complexity of iterated local search is difficult to predict, as it depends on several factors such as the size of the problem, the choice of local search algorithm, and the stopping criterion. In general, iterated local search can be time-consuming for large-scale problems, as it requires the repeated application of a local search algorithm.
- Determinism: Iterated local search in our case is non-deterministic as for different sets of initial solutions, the final ouput result will be different
- Efficiency: The efficiency of iterated local search depends on the choice of local search algorithm, the stopping criterion, and the quality and the number of the initial solutions. In our case, the Iterated Local Search performs outstandingly (see in Result Analyst).

Pseudo Code

```
function generating_sol():
    visited = [False for points in {1, 2, ..., N}]
    for points in {1, 2, ..., N}:
```

```
        randomly choose a point
        if visited[point] = False:
        for mailmen in {1, 2, ..., K}:
            randomly choose a mailman
        add point to mailman route

function find_better_neighbor(sol):
    neighbor_sol = swap(sol)
    #swap 2 points on the route of a mailman
    #or on the routes of 2 different mailmen

    if cal_obj_func(neighbor_sol) < cal_obj_func(sol):
        return(neighbor_sol)
    return

function Iterated_Local_Search():
    while number_of_sol < max_sol:
        sol = generate_sol()
        add sol to sol_set

    for solution in sol_set:
        while (find_better_neighbor not None):
            solution = find_better_neighbor(solution)
    return best_sol in sol_set
```

III.6 ORTOOLS CP-SAT

Overview

OR-Tools is an open source software suite for optimization, tuned for tackling the world's toughest problems in vehicle routing, flows, integer, constraint programming. And one of the most suitable tools to solve this problem is the CP-SAT solver, a constraint programming solver that uses SAT (satisfiability) methods. We only need to define necessary variables, add all constraints to the model and the tool will do the rest.

Properties

- Completeness: The CP-SAT solver is complete, meaning that it will find an optimal solution if one exists.

- Running time: The algorithm's computational time is low but increase rapidly when it comes to large input.
- Determinism: The CP-SAT solver is deterministic, meaning that it produces the same result each time it is run with the same input.
- Efficiency: The CP-SAT solver can also be an efficient algorithm because it gives optimal solution in a small amount of time. Although it works slower with big input, we still can apply some methods to support it(see more in Result Analyst).

Pseudo Code

```
function CP–SAT():
    # Create the model
    model = cp_model.CpModel()

    # Define variables
    # Add constraints
    x[k][i][j] = to check whether or not mailman k travels from i to j
    u[k][i] = rank of point i visited by mailman k
    z = longest route of all mailmen

    model.Add(Constraint 1, 2, 3 ... 8)

    # Add one more constraint
    # to optimize the objective function
    model.Add(Route done by any vehicle <= z)

    model.Minimize(Objective function)
    solver = cp_model.CpSolver()
    status = solver.Solve(model)
    if status == OPTIMAL or FEASIBLE:
        return solution
```

IV Result Analyst

Test and Result

We will use two tests to evaluate whether the algorithm is efficient or not by judging two factors

- Completeness
- Running time

Here is the two distance matrix tests we use:

$$\text{Test 1: } d = \begin{bmatrix} 0 & 1 & 12 & 9 & 9 & 3 & 8 & 6 & 19 & 10 \\ 3 & 0 & 5 & 8 & 16 & 16 & 10 & 3 & 10 & 14 \\ 2 & 11 & 0 & 12 & 10 & 7 & 3 & 7 & 11 & 16 \\ 2 & 4 & 14 & 0 & 3 & 16 & 18 & 10 & 1 & 19 \\ 16 & 15 & 6 & 8 & 0 & 17 & 2 & 20 & 16 & 16 \\ 15 & 10 & 19 & 6 & 14 & 0 & 8 & 9 & 7 & 12 \\ 12 & 2 & 1 & 9 & 20 & 15 & 0 & 1 & 2 & 7 \\ 12 & 2 & 19 & 6 & 12 & 17 & 14 & 0 & 7 & 2 \\ 11 & 15 & 8 & 4 & 15 & 15 & 20 & 19 & 0 & 10 \\ 11 & 19 & 7 & 18 & 8 & 20 & 17 & 9 & 7 & 0 \end{bmatrix}$$

$$\text{Test 2: } d = \begin{bmatrix} 0 & 46 & 39 & 20 & 21 & 39 & 50 & 45 & 43 & 44 \\ 41 & 0 & 33 & 45 & 34 & 36 & 32 & 30 & 45 & 36 \\ 34 & 30 & 0 & 23 & 28 & 39 & 26 & 26 & 42 & 27 \\ 21 & 35 & 26 & 0 & 46 & 21 & 23 & 35 & 46 & 20 \\ 49 & 42 & 39 & 40 & 0 & 48 & 47 & 36 & 37 & 41 \\ 39 & 21 & 28 & 42 & 28 & 0 & 33 & 33 & 38 & 42 \\ 33 & 28 & 43 & 48 & 42 & 40 & 0 & 23 & 48 & 28 \\ 42 & 26 & 50 & 40 & 41 & 50 & 25 & 0 & 22 & 49 \\ 39 & 35 & 29 & 31 & 43 & 43 & 36 & 46 & 0 & 36 \\ 27 & 31 & 27 & 41 & 28 & 40 & 33 & 50 & 47 & 0 \end{bmatrix}$$

Here are the result we get:

Test 1								
Completeness Time	N = 6, K = 1 (46, 23, 23)	N = 7, K = 1 (48, 24, 24)	N = 8, K = 1 (58, 29, 29)	N = 6, K = 2 (55, 21, 17)	N = 7, K = 2 (62, 28, 17)	N = 8, K = 2 (77, 33, 22)	N = 6, K = 3 (71, 29, 14)	N = 7, K = 3 (78, 36, 14)
Brute Force	Opt < 1s	Opt 6s	Opt 78s	Opt 50s	Opt 1200s	Opt > 2000s	Opt 1450s	Opt >2000s
Brute Force Extension	Opt < 1s	Opt 3s	Opt 34s	Opt 7s	Opt 87s	Opt 1200s	Opt 308s	N/A >2000s
Backtrack	Opt 37s	Opt 537s	Opt >2000s	Opt >2000s	Opt >2000s	Opt >2000s	Opt >2000s	Opt >2000s
Branch and Bound ver 0	Opt 14s	Opt 177s	N/A >2000s	Opt 142s	Opt 1980s	N/A >2000s	N/A >2000s	N/A >2000s
Branch and bound ver 1	Opt 3s	Opt 22s	Opt 235s	Opt 116s	Opt 1620s	N/A >2000s	N/A >2000s	N/A >2000s
Greedy	(106, 53, 53) < 0.1s	(76, 38, 38) < 0.1s	(134, 67, 67) < 0.1s	(105, 49, 28) < 0.1s	(126, 50, 38) < 0.1s	(162, 68, 47) < 0.1s	(195, 63, 44) < 0.1s	(126, 51, 25) < 0.1s
Hill-Climbing	(64, 32, 32) < 0.1s	(76, 38, 38) < 0.1s	(64, 32, 32) < 0.1s	(62, 30, 16) < 0.1s	(77, 37, 20) < 0.1s	(96, 42, 27) < 0.1s	(111, 42, 23) < 0.1s	(98, 44, 18) < 0.1s
Iterated Local Search	Opt < 0.1s	Opt < 0.1s	Opt < 0.1s	Opt <0.1s	Opt < 0.1s	Opt < 0.1s	Opt < 0.1s	Opt < 0.1s

Fig. 2: Result of test 1

Test 2								
Completeness Time	N = 6, K = 1 (380, 190, 190)	N = 7, K = 1 (426, 213, 213)	N = 8, K = 1 (476, 238, 238)	N = 6, K = 2 (460, 222, 119)	N = 7, K = 2 (502, 244, 129)	N = 8, K = 2 (553, 269, 142)	N = 6, K = 3 (574, 271, 101)	N = 7, K = 3 (609, 300, 103)
Brute Force	Opt < 1s	Opt 6s	Opt 78s	Opt 50s	Opt 1203s	Opt > 2000s	Opt 1448s	Opt >2000s
Brute Force Extension	Opt < 1s	Opt 3s	Opt 38s	(475, 221, 127) 20s	Opt 258s	N/A >2000s	Opt 967s	N/A >2000s
Backtrack	Opt 36s	Opt 540s	Opt >2000s	Opt >2000s	Opt >2000s	Opt >2000s	Opt >2000s	Opt >2000s
Branch and Bound ver 0	Opt 18s	Opt 219s	N/A >2000s	(475, 221, 127) 666s	N/A >2000s	N/A >2000s	N/A >2000s	N/A >2000s
Branch and bound ver 1	Opt 11s	Opt 92s	Opt 738s	(475, 221, 127) 537s	N/A >2000s	N/A >2000s	N/A >2000s	N/A >2000s
Greedy	(418, 209, 209) < 0.1s	(464, 232, 232) < 0.1s	(520, 260, 260) < 0.1s	(475, 221, 127) < 0.1s	(544, 244, 150) < 0.1s	(676, 272, 202) < 0.1s	(602, 221, 127) < 0.1s	(694, 244, 150) < 0.1s
Hill-Climbing	(416, 208, 208) < 0.1s	(460, 230, 230) < 0.1s	(502, 251, 251) < 0.1s	(475, 221, 127) < 0.1s	(544, 244, 150) < 0.1s	(676, 272, 202) < 0.1s	(602, 221, 127) < 0.1s	(694, 244, 150) < 0.1s
Iterated Local Search	Opt < 0.1s	Opt < 0.1s	(480, 240, 240) < 0.1s	(475, 221, 127) <0.1s	(508, 252, 128) < 0.1s	Opt < 0.1s	Opt < 0.1s	Opt < 0.1s

Fig. 3: Result of test 2

- 'Opt' stands for 'Optimal'
- 'N/A' stands for 'No Answer'
- Tuple with three numbers inside represent the objective functions result of a solution (f, f_1, f_2). The tuple on the top represent the objective function result of the optimal solution.

Brute Force(BF) vs Brute Force Extension(BFE)

- The running time of both algorithms is enormous.
- Although the cost of time of BFE is extremely less than that of BF, the final results of it are the same as that of BF in most cases.
- In the case that their results are different, the result of BFE is still worthy to be considered.
- BF runs in the same amount of time for two tests, which is quite easy-understanding as it is an exhausted algorithm.
- BFE runs in different amount of time for different tests as the idea of it is similar to Branch-and-Bound

Backtracking and two Branch-and-Bound(BAB) versions

- The running time of the three algorithms is huge.
- Two versions of BAB run much faster than Backtracking algorithm although their results are the same in most cases.
- In the case that their results are different, the result of two versions of BAB is still worthy to be considered.
- Backtracking runs the same time for two tests, while two versions of BAB run differently for different tests. This is because of the properties of Backtracking and Branch-and-Bound.
- The most noticeable point here is that BAB version 1 run faster than BAB version 0. This result shows us the importance of which branch to expand to minimize the running time when implementing BAB.

Brute Force and Backtracking

It can be seen that Brute Force runs immensely faster than Backtracking as although Backtracking stops generating branches when invalid solution appears, the time it takes to check constraints is too much while the number of branches it stops to generate is too few.

Greedy Algorithm

- For test 1, the final solutions of it are very bad, while for test 2, they are quite good, which indicates that the results of Greedy algorithm depends a lot on its input.
- The most noticeable point here is its running time. Not only is it small, but it also almost does not increase when the input become larger.

Hill-Climbing Algorithm

- For test 1, when the Greedy algorithm results are not good, Hill-Climbing improves it significantly. But for test 2, when the result of Greedy is great, the efficiency of Hill-Climbing is not obvious.
- Hill-Climbing algorithm is one of the two most efficient algorithm when we consider two factors: completeness and running time.

Iterated Local Search

- The most noticeable point in both two results of two tests is Iterated Local Search's result. The final solutions that it returns are optimal in 13/16 times. Although there are 3 times the answers are suboptimal, they are quite near to the optimal ones. Moreover, similar to Greedy and Hill-Climbing algorithm, the running time of Iterated Local Search nearly does not grow while the input becomes larger.
- Iterated Local Search is the most efficient algorithm in the five algorithms we have analyzed so far.

ORTOOLS CP-SAT

In the other hand, CP-SAT solver can give us an optimal solution in just a few seconds. We use the first distance matrix and test 35 cases, from 5 to 9 cities and 1 to 9 mailmen (we didn't test any case that have more mailmen than the number of cities). The table below shows the solution (f_1, f_2) and computational time:

K / N	1	2	3	4	5	6	7	8	9
5	F1 = F2 = 32 Time = 0.035	F1 = 24, F2 = 20 Time = 0.056	F1 = 32, F2 = 17 Time = 0.083	F1 = 50, F2 = 18 Time = 0.113	F1 = 72, F2 = 25 Time = 0.155				
6	F1 = F2 = 23 Time = 0.048	F1 = 21, F2 = 17 Time = 0.067	F1 = 29, F2 = 14 Time = 0.111	F1 = 47, F2 = 18 Time = 0.159	F1 = 70, F2 = 20 Time = 0.218	F1 = 92, F2 = 25 Time = 0.247			
7	F1 = F2 = 24 Time = 0.060	F1 = 28, F2 = 17 Time = 0.078	F1 = 36, F2 = 14 Time = 0.155	F1 = 47, F2 = 18 Time = 0.216	F1 = 65, F2 = 18 Time = 0.251	F1 = 88, F2 = 20 Time = 0.360	F1 = 110, F2 = 25 Time = 0.428		
8	F1 = F2 = 29 Time = 0.065	F1 = 33, F2 = 22 Time = 0.179	F1 = 41, F2 = 16 Time = 0.192	F1 = 52, F2 = 18 Time = 0.333	F1 = 73, F2 = 18 Time = 0.406	F1 = 89, F2 = 21 Time = 0.620	F1 = 111, F2 = 25 Time = 0.697	F1 = 140, F2 = 30 Time = 0.818	
9	F1 = F2 = 35 Time = 0.069	F1 = 35, F2 = 19 Time = 0.190	F1 = 47, F2 = 17 Time = 0.3	F1 = 24, F2 = 20 Time = 0.514	F1 = 74, F2 = 19 Time = 0.709	F1 = 89, F2 = 21 Time = 1.056	F1 = 110, F2 = 21 Time = 1.141	F1 = 132, F2 = 25 Time = 1.911	F1 = 161, F2 = 30 Time = 1.742

Fig. 4: Result of test 1 using CP-SAT

We continue to test with larger inputs. All inputs have N equals to K because we find out that CP-SAT took the longest time to solve in any case like that. Despite the time increasing a lot, it still give us a good feasible solution:

N	TIME
20	55s
30	245s
40	1107s
50	> 5000s

Fig. 5: Running time of large input

V Conclusion

Overall, the problem is how to find an way for K mailmen to go through N cities and collect all packages. To minimize the total distance of all routes and the longest distance traveled by a mailman, we come up with 5 algorithm which includes: Brute Force, Backtracking, Greedy, Hill-Climbing and ORTOOLS CP-SAT. Each of them all has different way to calculate the solution and running time.

The Brute Force algorithm and Backtracking algorithm have the same idea that testing all possible solutions to find out the most optimal. To reduce the computational time, we need to add another constraint and apply Branch-and-Bound algorithm that limit many cases. In exchange, it may prune the optimal solution, makes these two algorithms inefficient.

On the contrary, the greedy algorithm runs really fast, even with large input by just simply choosing the nearest unvisited city and repeat it until all cities have been visited by all mailmen, but the solution may not optimal.

Hill-Climbing algorithm can find a quite great solution in low time, after we combine it with greedy algorithm. Sadly, it can't work when it comes to large input due to the limitation of Python. This algorithm could perform better if it implemented in other programming language. However, Iterated Local Search and CP-SAT can do this. In term of CP-SAT, even it will take a long time to run if N and K are big, CP-SAT still provide us with an optimal solution. Other with CP-SAT, Iterated Local Search only consume small amount of time to run and its final results are quite positive with small N and K

In conclusion, if the input is not too big, ORTOOLS CP-SAT and Iterated Local Search are the most efficient method to solve the problem. Greedy can be used in other cases, as it gives us a temporary solution in low time. In further future, we will add more methods, algorithm and implement them in other programming languages for more general comparison.

VI Work Sharing

- Modeling: Bình, Pháp, Nhật
- Brute Force with Branch Cutting: Bình
- Backtracking with Branch-and-Bound: Bình
- Greedy Algorithm: Bình
- Hill-Climbing: Bình
- Iterated Local Search: Bình
- ORTOOLS CP-SAT: Nhật
- Report: Bình, Nhật
- Slide: Pháp