Fundamental of Optimization

Mid-term Project Presentation

Nguyễn Thanh Bình	20210106
Bùi Anh Nhật	20210657
Nguyễn Quang Pháp	20214921

TABLE OF CONTENTS

Problem discription

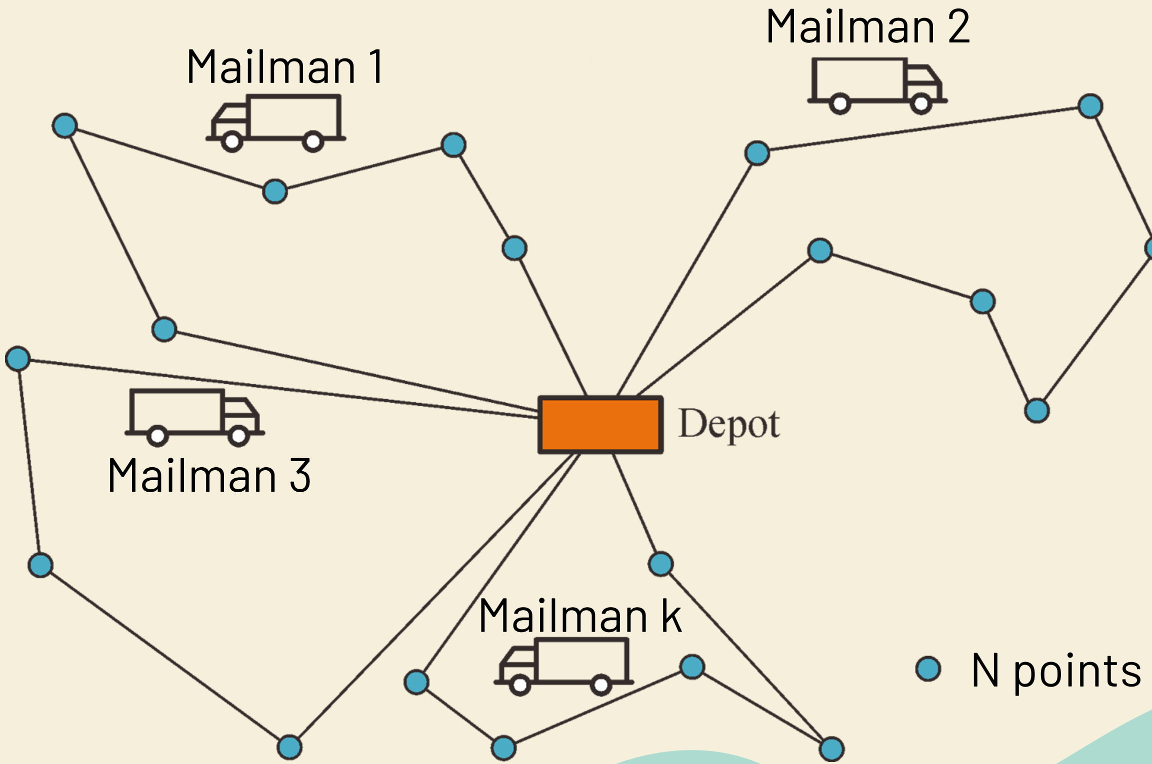
01

PROBLEM: Topic1

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 carriers to collect packages, determining which points each carrier should collect and in what order to minimze:

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

PROBLEM



Modeling

02

Notations and Constraints

Notations

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

- $B = \{1, 2, \ldots, N+K, \ldots, N + 2K\}$: set of all points.
- $F1 = \{(i, N + k) \mid i \in B, k \in \{1, 2, \ldots, K\}\}$: set of edges that go back to the starting points.
- $F2 = \{(N + K + k, i) \mid i \in B, k \in \{1, 2, \ldots, K\}\}$: set of edges that start from terminating points.
- $F3 = \{(i, i) \mid i \in B\}$
- $F4 = \{(i, j) \mid i \in \{N + 1, \ldots, N + K\}, j \in \{N + K + 1, \ldots, N + 2K\}\}$: set of edges that go directly from starting points to terminating points.
- $A = B^2 \backslash F1 \backslash F2 \backslash F3 \backslash F4$: set of edges that can be chosen.

Variables

- $X(k, i, j) = \begin{cases} 1, if\ mailman\ k\ travels\ from\ i\ to\ j \\ 0, otherwise \end{cases}$
- U (k, i) = the rank of point i visited by mailman k

Constraints

- Mailman k starts from N+k and terminates at N+K+k only once:

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

- Mailman k only starts at N+k and terminates at N+K+k:

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

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

Constraints

- One mailman visits each point and all points are visited:

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

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

- No subtour or partial tour in the solution:

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

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

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

Objective Functions

- Total distance:

$$f1 = \sum_{k=1}^{K} \sum_{(i,j)\in A} d(i,j) * X(k,i,j)$$

- Longest distance:

$$f2 = \max_{k \in \{1,2,\dots,K\}} \sum_{(i,j)\in A} d(i,j) * X(k,i,j)$$

=> *<u>Objective function:</u>*

f = f1 + K × f2 (K is number of salesmen)

Algorithms

03

Six algorithms for the problem

Proposed algorithms

Brute Force

Brute Force with Branch Cutting

Backtracking

Backtracking and Branch-and-Bound

Greedy

Greedy Algorithm

Hill Climbing

Hill Climbing Algorithm

Iterated Local Search

Repeatedly Local Search

CP-SAT

CP-SAT solver using OR Tools

Brute Force

Generate all possible solutions and select the optimal one

Overview

Note: Add another constraint: $X(k,i,j) = 0 \ \forall \ (i,j) \notin A$

- Generate all possible X(k, i, j) with (i, j) ∈ A
- Select the ones that satisfy the constraints
- Give the result that minimizes the objective function

Properties

Completeness

Guaranteed to find optimal solution if run to completion

Determinism

Deterministic, same result every time

Run time

Exponential running time

Efficiency

Inefficient

Extension

- Use a lower bound on total distance travel f1 to prune suboptional solutions
- Generate new candidate by branching current solution
- The running time of this algorithm is different for different input.

Note: May prune the optimal solution because it only considers f1, still much more efficient than the original Brute Force algorithm

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:
                A_mark = A
                for (i_, j_) in A:
                    if (i_ = i) or (j_ = j):
                        remove (i_, j_) out of A
                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 salesmen 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()
                A = A_mark
    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)
```

Backtracking

Systematically generating and testing all possible solutions by depth-first search and backtrack when needed

Overview

Note: Add another constraint: $X(k,i,j) = 0 \;\forall\; (i,j) \notin A$

- Systematically generate possible solutions
- Check for constraints
- Backtrack (i.e undo) if leading to invalid result

Properties

Completeness

Guaranteed to find optimal solution if one exists

Determinism

Deterministic, same result every time

Run time

Exponential running time, due to many sub-optimal solutions

Efficiency

Inefficient

Extension: Branch-and-Bound

- Use a lower bound on total distance travel f1 to prune suboptimal solutions
- Generate new candidate by branching current solution
- The running time of this algorithm is different for different input.

  Note: May prune the optimal solution because it only considers f1, still much more efficient than the original idea.

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 salesmen 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)
```

Greedy Algorithm

Try to find shortest move for all mailmen
at each step

Overview

- All mailmen start at the depot
- Find the nearest point for each mailman
- Repeat until all points are visited

Properties

Completeness

Not guaranteed to find optimal solution

Determinism

Deterministic, same result every time

Run time

Fast running time

Efficiency

Runs fast, might not find high-quality solution

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 salesmen, showing the postion of
              salesmen at present, initially all zero

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

    salesman_k, next_city = find_next_city(current, visited)
    current[salesman_k] = next_city
    add next_city to routes[salesman_k]

    if check(routes):
    #check if all salesmen have returned to the depot
        return solution
    else:
        Greedy()
```

Hill-Climbing

Start with an initial solution and repeatedly improve it by considering its “neighbor solution” to reach a local optimum

Overview

- Use Greedy algorithm to generate an initial solution
- Check for better 'neighbor solutions' and make it 'current solution'

  Note: Neighbor solutions are generated by swapping 2 points in a mailman's route or 2 points in two mailmen's routes
- End if there is no better neigbor solutions

Properties

Completeness

Not guaranteed to find optimal solution, might get stuck in local minimal

Determinism

Deterministic, same result every time

Run time

Very fast running time

Efficiency

Most reasonable solution in acceptable time

Pseudo code

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

    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
```

Iterated Local Search

Repeatedly perform local search

Overview

- Implementing an iterated local search
- Repeatedly perform local search on a set of randomly generated solutions
- Stop when no neighbor is better then the current solution

Properties

Completeness

Not guaranteed to find optimal solution

Determinism

Non-deterministic

Run time

Difficult to predict, depends on input

Efficiency

Depends on the choice of local search algorithm, stopping criterion and initial solutions

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:
```

```
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
```

CP-SAT

Define constraints and variables then use OR-Tools to solve

Overview

- Define variables and constraints as previously described
- Create a model using OR-Tools
- Minimize objective function using OR-Tools built-in solver

Properties

Completeness

Guaranteed to find optimal solution if one exists

Determinism

Deterministic, same result every time

Run time

Low running time but grows rapidly with large input

Efficiency

Optimal solution, small amount of time.
Can be improved

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 vehicle k travels from i to j
    u[k][i] = rank of point i visited by vehicle k
    z = longest route of all vehicle

    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
```

Result Analyst

04

Result analyst and conclusion

Testing

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

- Completeness
- Running time

Testing

Two distance matrix tests we use:

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

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

Result

- 'Opt' stands for 'Optimal'
- 'N/A' stands for 'No Answer'
- Tuple represents the objective functions result (f, f1, f2).

  The tuple on the top represent the objective function result of the optimal solution.

Result

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

Result

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

Result: CP-SAT

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

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

CONCLUSION

Conclusion

Brute Force and BackTracking

- Deterministic, guaranteed solution
- High running time

Iterated Local Search

- Optimal solution
- Small amount of time with small input

Hill-Climbing Algorithm

- Good solution in acceptable time
- Cannot work with large input when using Python

CP-SAT with OR-Tools

- Can work with large input and provide optimal solution

Conclusion

If the input is not too big, **OR-Tools CP-SAT** and **Iterated Local Search** is the most efficient method to solve the problem

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

In the future, we will add more algorithm and implement them in other programming languages

Thank you for listening