Group 7

Report Project Optimization

Balanced Staff Routing for Maintenance

Đinh Nguyễn Công Quý – 20214927
Vũ Tuấn Minh – 20210597
Hoàng Tú Quyên – 20214929
Lê Tuấn Anh - 20214874

Table of Contents

I. Problem description

There exist N customers where customer i resides in location i $(i = 1,2,3,\ldots,N)$ that demand Internet maintenance service. The maintenance service of the customer i lasts for $d[i]$ (time unit). There are K employees departing from one single depot (denoted by 0). The traveling time between two locations i and j $(i,j = 0,1,\ldots,N)$ is $t[i,j]$. The objective is to schedule a plan in which the maximum working time (equals to the traveling time plus the maintenance time) among the employees is minimized.

II.Modelling

- Denotion

  - $\boldsymbol{N}$: the number of customers

  - $\boldsymbol{K}$**:** the number of employees

  - $\boldsymbol{d} = [d[1], d[2], \ldots , d[N]]$ is a vector containing the maintenance time of every customer

  - $\boldsymbol{t} = [t[i,j] \; for \; i,j \; in \; \{0,1,\ldots,N\}]$ is a $2D$-array that represents the traveling time between location i and location j.

- Decision variable:

  - $\boldsymbol{x[i,k]}$ is a binary variable indicating whether a customer i is contained in a maintenance route of employee k. Explicitly:

    + $\boldsymbol{x[i,k] = 1}$ if customer i is served by employee k.

    + $\boldsymbol{x[i,k] = 0}$ if customer i is not served by employee k.

  where $1 \leq i \leq N$ and $1 \leq k \leq K$.

  - $\boldsymbol{y[i,j,k]}$ is another binary variable that indicates whether the employee k travels from customer i 's location to customer j 's location (including the depot) in his maintenance route. In other word:

    + $\boldsymbol{y[i,j,k] = 1}$ if employee k travels from customer i 's location to customer j 's location in his maintenance route.

    + $\boldsymbol{y[i,j,k] = 0}$**,** otherwise.

where $0 \le i, j \le N, i \ne j$ and $1 \le k \le K$.

- <u>Constraints:</u>

1. $\sum_{j=1}^{N} y[0, j, k] = \sum_{i=1}^{N} y[i, 0, k] = 1 \ (1 \le k \le K)$

2. $\sum_{j=1}^{N} y[i, j, k] = \sum_{j=1}^{N} y[j, i, k] = x[i, k] (1 \le i \le N; i \ne j; 1 \le k \le K)$

3. $\sum_{k=1}^{K} x[i, k] = 1 \ (1 \le i \le N)$

4. $\sum_{(i,j) \in S} y[i, j, k] \le |S| - 1 \ (1 \le k \le K;\ \forall S \subseteq \{1, 2, \dots, N\}; S \ne \emptyset$

5. $x[i, k] \in \{0, 1\}$

6. $y[i, j, k] \in \{0, 1\}$

In the above constraints:

- *Constraint (1)* requires that all k employees must depart from the depot (i.e., point 0) and return to the depot after serving all the customers in his route.
- *Constraint (2)* indicate that there is exactly one incoming path to customer i and one outgoing path from customer i if customer i is contained in traveling tour of employee k.
- *Constraint (3)* implies that each customer i should be visited once and by only one employee.
- *Constraint (4)* indicates the sub-tour elimination constraint. Specifically, the number of edges with their endpoints contained in any nonempty proper subset S of $V \setminus \{0\}$ is no greater than $|S| - 1$, thus prevents solutions consisting of several disconnected closed tours. This image visualizes this constraint in graph presentation.
- *Constraints (5)* and *(6)* represent the domain of variable x and y.

- <u>Objective function:</u> Minimizing the maximum working time among the employees:

$$\max \{\sum_{i=1}^{N} x[i, k] * d[i] + \sum_{i=0}^{N} \sum_{j=0}^{N} y[i, j, k] * t[i, j] \rightarrow min$$

$$i \ne j; 1 \le k \le K)$$

III. Proposed Algorithms

- **Exact Algorithms: ILP and CP (Quyên)**

For this approach, we apply 2 solvers: The OR-tools linear_solver which employs a combination of branching, cutting plane algorithms, and heuristics, for the Integer Linear Programming model, and the OR-tools constraint_solver which implements local search and meta-heuristics for the Constraint Programming model.

For ILP model, we introduce two more variables:

+ $\boldsymbol{c[k]}$: The total working time of employee k $(1 \le k \le K)$, which is equal to the sum of the maintenance time of all customers in his serving tour, and the total traveling time.

+ $\boldsymbol{z}$: indicates the maximum working time among the employees.

The problem can be formulated as an Integer Linear Programming (ILP) with the help of the ***pywraplp*** library. The formulation is quite similar to the mathematical model above, except that we add the linearization constraints to the problem, thus changing the objective function:

Additional constraints for the ILP model:

+ Define $c[k]$:

$$c[k] = \sum_{i=1}^{N} x[i,k] * d[i] + \sum_{i=0}^{N}\sum_{j=0}^{N} y[i,j,k] * t[i,j]$$

$1 \le k \le K$

+ Linearization Constraint

$z \ge c[k] \quad (1 \le k \le K)$

The objective function is to minimize the maximum working time among the employees, which can be written as:

$$z \to min$$

In the above formula:

- $x[i,k] = 1$ if customer i is in the maintenance schedule of employee k; $x[i,k] = 0$ otherwise. (with $1 \leq i \leq N;\ 1 \leq k \leq K$)
- $y[i,j,k] = 1$ if the travel distance from i to j is in the maintenance schedule of employee k; $y[i,j,k] = 0$ otherwise. (with $0 \leq i,j \leq N;\ 1 \leq k \leq K$)
- $d[i]$ is a vector containing the maintenance time of every customer
- $t[i,j]$ is a $2D$-array that represents the traveling time between location i and location j.

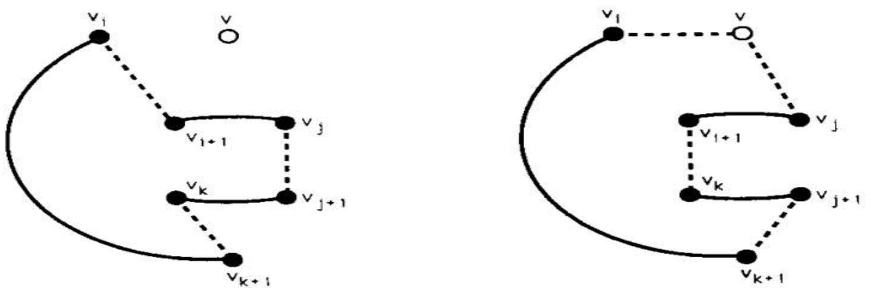
As for the constraint_solver we utilized the **pywrapcp** library and found that this solver gave much better performance and more steady increase in the execution time than the ILP model via the same test data.

However, both of these models could not solve the problem when the test data size becomes larger (N > 100). To tackle this matter, we would apply the heuristic algorithms, so Quý will continue with our presentation.

- **Greedy (Quý)**

The idea for the first algorithm is that we initialize K points from the set of N locations, except for the depot point 0, as centroids, then construct for each a return route between the point and the depot.

Next, consider in turn each unrouted point and insert it in a route that produces the least increase of the MinMax objective function, then remove it from the unrouted points list and update this list. The algorithm stops when a feasible solution is obtained, or explicitly, when all of the points have been routed.



Insertion Method

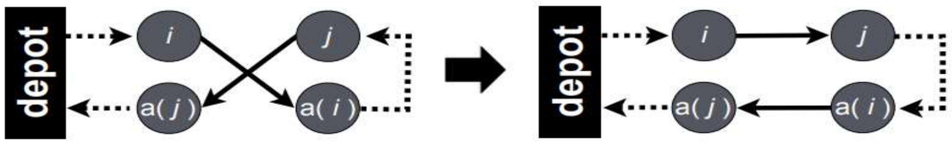
We tried processing the algorithm with the unsorted and sorted list of unrouted points and noticed that the sorted by maintenance time in decreasing order returned the least bias among the values of the objective function of each iteration.

Computational results indicate that this algorithm require impressively low execution time and return good or even optimal solution when N and K are small.

- **Local Search (Minh)**

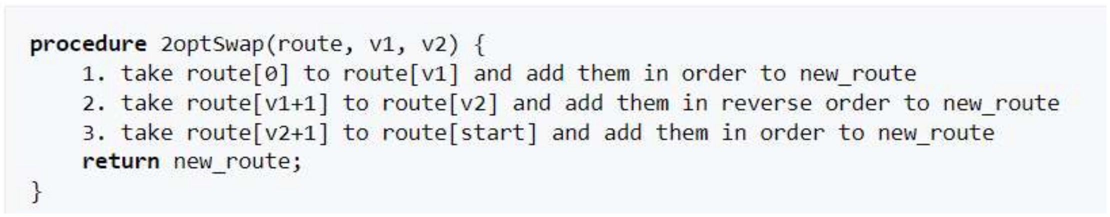
After trying several times with greedy, we realized that greedy has some weakness. In more detail, there is the existence of bad solutions. So, we came up with the local search algorithm as another way to look for a better solution. We utilized two more heuristic procedures which are 2-opt and CROSS-exchange.

The 2-opt algorithm exchanges two paths for other two paths in the same tour. The main idea behind it is to take a route that crosses over itself and reorder it so that it does not. This is a simple visualization of this algorithm.



2-opt algorithm

In the following pseudocode, the mechanism by which the 2-opt swap manipulates a given route is as follows. Here v1 and v2 are the first vertices of the edges you wish to swap when traversing through the route:

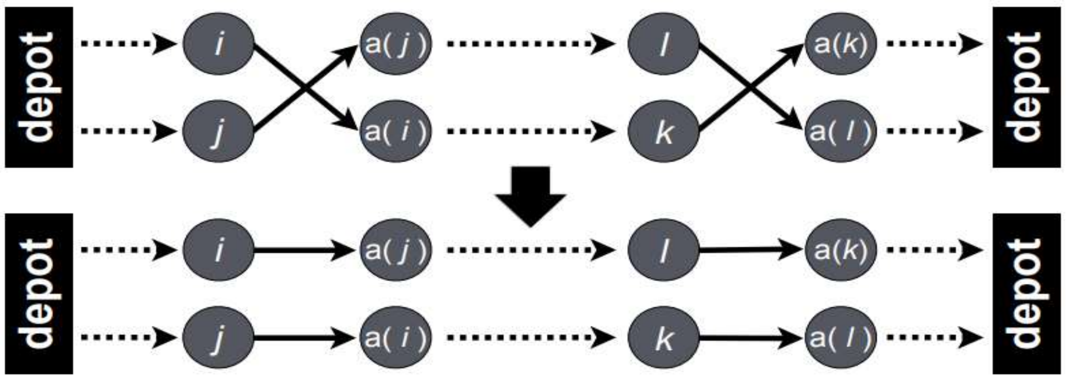


```
procedure 2optSwap(route, v1, v2) {
    1. take route[0] to route[v1] and add them in order to new_route
    2. take route[v1+1] to route[v2] and add them in reverse order to new_route
    3. take route[v2+1] to route[start] and add them in order to new_route
    return new_route;
}
```

Swapping in 2-opt

```
repeat until no improvement is made {
    best_distance = calculateTotalDistance(existing_route)
    start_again:
    for (i = 0; i <= number of nodes eligible to be swapped - 1; i++) {
        for (j = i + 1; j <= number of nodes eligible to be swapped; j++) {
            new_route = 2optSwap(existing_route, i, j)
            new_distance = calculateTotalDistance(new_route)
            if (new_distance < best_distance) {
                existing_route = new_route
                best_distance = new_distance
                goto start_again
            }
        }
    }
}
```

In addition, the CROSS-exchange randomly exchanges two cities from 2 tours in one tour and a partial tour of the other tours. The idea is quite similar to 2-opt except that the swapping method w



CROSS-exchange algorithm

Local search methods move from a current solution to a better solution in neighborhoods of the current solution until an optimal or a local optimal solution is found. In the solution, we create two operators: Move (Move a city from the tour has maximum cost to the tour has minimum cost) and Swap (Swap randomly two cities from maximum tours and minimum tours) to find the best neighborhood. However, in general, local search methods cannot find optimal solutions due to the local minimum problem.

IV. Experiments

We ran tests with data instances with gradually increasing N, K.

* Exact algorithms:

data_1: N = 5, K = 2			**data_2: N = 10, K = 2**		
	f	t(s)		f	t(s)
ILP	360	0.03	ILP	550	0.58
CP	360	0.02	CP	550	0.04
data_3: N = 50, K = 5			**data_4: N = 50, K = 10**		
	f	t(s)		f	t(s)
ILP	772	10.45	ILP	484	21.37
CP	772	6.58	CP	484	8.65
data_5: N = 100, K = 10			**data_6: N = 100, K = 20**		
	f	t(s)		f	t(s)
ILP	3690	43.12	ILP	2370	58.76
CP	3690	15.18	CP	2370	19.74

This is the experimental result of the two exact algorithms. The execution time significantly increases as N gets larger. In the cases of $N = 5$ and $N = 10$, the ILP and CP both return the optimal solution in a short span of time. However, the ILP performs much worse than the CP model when N gets larger, but the matter does not concern the case when only K increases. The two model would get stuck at solving the problem of size $N > 100$.

* Heuristic algorithms:

data_1: N = 5, K = 2					
	f_min	f_max	f_avg	std_dev	t_avg (s)
Greedy	360	580	564.3	5.4	0.0004
Local search	550	560	558	4.4	5.43
Local Search + Greedy Initial Solution	360	360	360	0.0	2.12

data_2: N = 10, K = 2					
	f_min	f_max	f_avg	std_dev	t_avg (s)
Greedy	550	580	564.3	5.4	0.001
Local search	550	560	558	4.4	8.9
Local Search + Greedy Initial Solution	550	570	557	6.75	4.0

data_3: N = 50, K = 5					
	f_min	f_max	f_avg	std_dev	t_avg (s)
Greedy	812	914	864.1	18.3	0.004
Local search	841	860	852.1	6.84	33.75
Local Search + Greedy	835	867	854.2	8.09	14.05

Initial Solution					

data_4: N = 50, K = 10					
	f_min	f_max	f_avg	std_dev	t_avg (s)
Greedy	528	598	562.04	16.04	0.006
Local search	535	557	549.1	7.1	9.18
Local Search + Greedy Initial Solution	531	555	542.8	6.92	3.42

data_5: N = 100, K = 10					
	f_min	f_max	f_avg	std_dev	t_avg (s)
Greedy	3770	4190	3738.1	68.01	0.022
Local search	3710	3890	3811.1	59.04	36.67
Local Search + Greedy Initial Solution	3560	3800	3709	77.38	28.78

data_6: N = 100, K = 20					
	f_min	f_max	f_avg	std_dev	t_avg (s)

Greedy	2490	2712	2417.4	89.67	0.04
Local search	2487	2565	2515	24.75	9.78
Local Search + Greedy Initial Solution	2228	2515	2420	86.89	4.44

data_7: N = 200, K = 10					
	f_min	f_max	f_avg	std_dev	t_avg (s)
Greedy	4700	5870	5114.6	184.49	0.063
Local search	5230	5380	5328.89	43.43	262.38
Local Search + Greedy Initial Solution	5200	5660	5362	126.91	162.34

data_8: N = 200, K = 20					
	f_min	f_max	f_avg	std_dev	t_avg (s)
Greedy	3600	4230	3881.6	102.82	0.107
Local search	3900	4100	3898	49.48	250.68
Local Search + Greedy Initial Solution	3620	4070	3901	136.49	25.36

data_9: N = 400, K = 40					
	f_min	f_max	f_avg	std_dev	t_avg (s)
Greedy	3500	4030	3729.5	98.96	0.793
Local search	3700	3910	3891.2	47.89	350.23
Local Search + Greedy Initial Solution	3530	3820	3669	99.15	35.26

data_10: N = 500, K = 50					
	f_min	f_max	f_avg	std_dev	t_avg (s)
Greedy	3530	4070	3767.3	110.58	1.883
Local search	3640	3810	3712.2	98.22	450.23
Local Search + Greedy Initial Solution	3520	3840	3671	102.78	42.07

data_11: N = 1000, K = 100					
	f_min	f_max	f_avg	std_dev	t_avg (s)
Greedy	3500	4000	3783.8	92.09	12.084
Local search	3620	3810	3798.1	56.66	650.23

Local Search + Greedy Initial Solution	3460	3740	3550	76.59	294.62

data_12: N = 1000, K = 200					
	f_min	f_max	f_avg	std_dev	t_avg (s)
Greedy	2780	3148	2914.7	45.62	30.61
Local search	2810	3120	2915.5	21.53	672.53
Local Search + Greedy Initial Solution	2780	2990	2850	82.19	356.46

For each test case, we run each algorithm for 100 times and take note of the maximum (f_max), minimum (f_min), average (f_avg) objective values, and the average execution time (t_avg) among all iterations, then we calculate the standard deviation (std_dev). The Greedy Algorithm requires the least running time. However, it also returns the highest standard deviation, which means there is a high probability that it will return a poor solution. The local search, on the other hand, takes more time to execute, but can return good feasible or even optimal solutions with relatively low standard deviation.

V. Conclusion

Overall, all the aforementioned algorithms could give rather acceptable or even optimal solutions. But the performance is different between these algorithms.

+ The exact algorithm: When N and K are small, the ILP algorithm returns the optimal solution in the shortest time. However, when N and K become enormous, it takes a long time to return the solution explicitly, when proceeding to N > 50, the algorithm would not return the solution in a given time limit.

+ The greedy algorithm gives feasible solutions in the shortest time and of quite good quality. The bias of this algorithm is that the convergence rate of this algorithm depends much on the random seed_customer, and the failure chance gets higher when the average number of customers assigned to each employee gets larger.

+ The local search algorithm takes longer to return the feasible solution, but the difference among the objective values of each iteration is narrowed, and it could return the optimal solution given enough time. The weakness of this problem is that it depends highly on the quality of the initial feasible solution. The local search algorithm with greedy feasible solution gives much better performance

In conclusion, method using Integer Linear Programming gives the best performance when the data size is small. However, in reality, the method using Greedy algorithm would be recommended due its fast performance and good feasible solution - even though they may not be optimal. The Local Search method, though could give an optimal solution, takes too much time to execute.

For the future work, we would focus on improving the running time for these two exact algorithms and the solution quality of the heuristic methods. In addition, we plan on working more about the Genetic Algorithm, which we came up with during our research but the implementation still takes too long to execute. It has many potentials in both reducing the execution time and improving the solution quality if implemented correctly.

REFERENCE

- Explicit Dantzig-Fulkerson-Johnson formulation

https://how-to.aimms.com/Articles/332/332-Explicit-Dantzig-Fulkerson-Johnson-formulation.html

- Generating subtour elimination constraints for the Traveling Salesman Problem

https://www.academia.edu/36992269/Generating_subtour_elimination_constraints_for_the_Traveling_Salesman_Problem

- Minimizing the Longest Tour Time Among a Fleet of UAVs for Disaster Area Surveillance

http://users.cecs.anu.edu.au/~Weifa.Liang/papers/GPXLJXYW20.pdf

- Solving Min-Max Multiple Traveling Salesman Problems by Chaotic Neural Network

https://www.ieice.org/nolta/symposium/archive/2014/articles/B1L-C1-6137.pdf

- An effective method for solving multiple travelling salesman problem based on NSGA-II

https://www.tandfonline.com/doi/full/10.1080/21642583.2019.1674220#:~:text=Multiple%20Travelling%20Salesman%20Problem%20%28MTSP%29%20is%20an%20extension,a%20salesman%20while%20requiring%20a%20minimum%20total%20cost.

- Ant-Balanced Multiple Traveling Salesmen: ACO-BmTSP

https://www.mdpi.com/1999-4893/16/1/37/xml

- Source code for the linear_solver

Python Reference: Linear Solver | OR-Tools | Google Developers

- Source code for the constraint_solver:

Python Reference: Routing | OR-Tools | Google Developers