

Hanoi University of Science and Technology
School of Information and Communication of Technology



VEHICLE ROUTING PROBLEM WITH TIME WINDOWS (VRPWTW)

Duong Minh Quan	20210710
Do Dinh Kien	20214906
Vu Hoang Phuc	20214923

February 5, 2023

ACKNOWLEDGEMENT

We would like to express our appreciation to Dr.Bui Quoc Trung who gave us opportunity to work on this project. This work could not be finished without your helpful advices and thoroughly guidance.

ABSTRACT

This report discusses the problem of solving a Vehicle Routing Problem (VRP) involving scheduling visits to customers who are only available during specific time windows. Next, we illustrate our different approaches to the problem such as graph-based, mathematical and heuristics. Then, we describe generalizations of the problem and evaluate the efficiency of mentioned methods. Finally, we present our conclusions and discuss some open alternatives.

Contents

1	DESCRIPTION OF THE PROBLEM	4
1.1	Problem description	4
1.2	Input and Output	4
1.3	Math modelling	6
2	VRPWTW ALGORITHMS	7
2.1	Exhaustive search	7
2.1.1	Backtracking	7
2.1.2	Branch and Bound	9
2.2	Greedy	11
2.3	Local search - Hill Climbing	12
2.4	OR-TOOL with Constraint Programming (CP)	14
3	RESULTS	15
3.1	Experiment procedure	15
3.2	Experiment result	15
4	CONCLUSION AND POSSIBLE EXTENSIONS	18
4.1	Programming tasks	19
4.2	Analytics tasks	19

Chapter 1

DESCRIPTION OF THE PROBLEM

1.1 Problem description

A delivery man is given a task of picking goods from the warehouse (marked as point 0) and delivering them to N customers $1, 2, 3, \dots, N$. Each customer i is only available to receive the good between $e[i]$ and $l[i]$, which are earliest starting time and latest termination time for the service. $d[i]$ is the duration of unloading activity for the good of customer i (the time required to complete the service). Given $t[i][j]$ and $c[i][j]$ are symmetric traveling time and distance adjacent matrices from customer i to j , respectively. Knowing that the delivery man start from the warehouse at time t_0 , we need to find the **set of routes** which minimizes the total length of the routes without violating the time window constraints.

1.2 Input and Output

a) Input

In our repository, folder `test_case` contain 100 tests generated by `test_gen.py` in the form as follow:

- The first line is N : the number of customers
- The next line is t_0 : starting time
- The next N lines, each line contain 3 numbers: $e[i], l[i], d[i]$
- The next $N + 1$ line, each line contains $N + 1$ numbers: $t[i][j]$ (adjacent traveling time matrix)
- The next $N + 1$ line, each line contains $N + 1$ numbers: $c[i][j]$ (adjacent traveling distance matrix)

For example: In `test_cases/test_0/input.txt`

$$\begin{array}{l} 2 \\ 7 \\ 76 \ 131 \ 9 \\ 92 \ 143 \ 6 \\ 0 \ 9 \ 1 \\ 9 \ 0 \ 8 \\ 1 \ 8 \ 0 \\ 0 \ 6 \ 13 \\ 6 \ 0 \ 5 \\ 13 \ 5 \ 0 \end{array} \quad \Leftrightarrow \quad \begin{array}{l} N = 2 \\ t_0 = 7 \end{array}$$

i	$e[i]$	$l[i]$	$d[i]$
1	76	131	9
2	92	143	6

$$T = \begin{bmatrix} 0 & 9 & 1 \\ 9 & 0 & 8 \\ 1 & 8 & 0 \end{bmatrix}; \quad C = \begin{bmatrix} 0 & 6 & 13 \\ 6 & 0 & 5 \\ 13 & 5 & 0 \end{bmatrix}$$

In our Python scripts, the function `input{test_case_dir}` will enter the data from `input.txt`:

```

1 def input(test_case_dir):
2     input_file = f'{test_case_dir}/input.txt'
3     global n,t0,t,c,customer,d
4     with open(input_file, 'r') as f:
5         n = int(f.readline())
6         t0 = int(f.readline())
7         for i in range(n):
8             u,v,w = map(int, f.readline().split())
9             customer.append([u,v])
10            d.append(w)
11        for i in range(n+1):
12            t.append(list(map(int, f.readline().split())))
13        for i in range(n+1):
14            c.append(list(map(int, f.readline().split())))

```

b) Output

In `output.txt`

- The first line is a number which is the minimum route length satisfied the time window constraints
- The next line is the optimal route solution

Every Python scripts in our repository will running algorithms on a number of test cases, and then export the results in a `csv` file for easily analyzing.

Test case	N = ?	Output	Running Time	Optimal Route
0	2	11	0	[1, 2]
1	2	5	0	[2, 1]
2	2	8	0	[1, 2]
3	2	No feasible solution	0	No route
4	4	13	0	[4, 2, 1, 3]
5	4	31	0	[3, 1, 2, 4]
6	4	25	0	[2, 3, 4, 1]
7	4	24	0	[2, 3, 4, 1]
8	6	15	0	[5, 1, 3, 6, 2, 4]
9	6	40	0	[4, 5, 2, 1, 3, 6]
10	6	No feasible solution	0	No route
11	6	47	0	[6, 3, 1, 5, 2, 4]
12	8	25	0.15	[8, 5, 1, 4, 6, 2, 3, 7]
13	8	33	0.13	[5, 6, 3, 4, 7, 1, 8, 2]
14	8	65	0.13	[8, 7, 5, 3, 1, 4, 2, 6]
15	8	38	0.16	[6, 1, 8, 4, 5, 2, 3, 7]

`result_backtracking.csv`

1.3 Math modelling

There are some feasible approaches to the problem we have figured out so far such as graph-based, mathematical or heuristics. In this report, we will model the problem for **Constraint Programming** solver:

i. Input variables

- $e[i]$ and $l[i]$: the earliest starting time and the latest termination time for the receiving process of customer i , respectively
- $d[i]$: the duration of unloading activity of the customer i 's good
- $t[i][j], c[i][j]$: the traveling time and distance from customer i to customer j , respectively

ii. Input constraints:

The unloading process must be finished before the latest termination time of each customer

$$\sum_{\forall i \neq 0} e[i] + d[i] \leq l[i]$$

iii. Variables for CP

- A binary variable $x[i][j]$ such that:

$$\begin{cases} x[i][j] = 1, & \text{customer } j \text{ is visited after customer } i \\ x[i][j] = 0, & \text{otherwise} \end{cases}$$

- $s[j]$: the start time of service at customer j

iv. Constraints:

- Each customer is visited once:

$$\begin{cases} \sum_{\forall i} x[i][j] = 1 & \text{(Only one customer } i \text{ is visited before customer } j) \\ \sum_{\forall i \neq 0} x[j][i] \leq 1 & \text{(At most one customer } i \text{ is visited after customer } j) \end{cases}$$

- Start at warehouse (marked as point 0):

$$\sum_{\forall i \neq 0} x[0][i] = 1$$

- Ready time for starting service constraint:

$$s[j] \geq s[i] + t[i][j] + d[i] \quad \text{whenever } x[i][j] = 1$$

- Time window constraints: Start time of service plus time for unloading must be completed before the termination time

$$\begin{cases} s[i] + d[i] \leq l[i] & \forall j \neq 0 \\ s[i] \geq e[i] \end{cases}$$

v. Objective function

$$\text{Minimize} \quad \sum_{\forall i, j} c[i][j] \times x[i][j]$$

Chapter 2

VRPWTW ALGORITHMS

2.1 Exhaustive search

Exhaustive search, also known as **Brute Force** search, is not an efficient algorithm for **NP-hard** problems. The time complexity of this algorithm grows exponentially, meaning the number of possible solutions grows rapidly with the size of input. Exhaustive search involves checking all possible solutions one by one and removing those who fail to satisfy the time window constraints. After that, it will find the one whose total route length is minimum and that is the optimal solution for the problem.

For example, the solution of this problem is a ordered tuple $(v_1, v_2, v_3, \dots, v_n)$ for $v_i \in [1, n]$. Hence, the solution space S includes all the permutations of these n elements. Let's see the table below to see how rapidly the cardinality of S grows:

N	Size of S
1	$1! = 1$
2	$2! = 2$
3	$3! = 6$
4	$4! = 24$
5	$5! = 120$
6	$6! = 720$
...
15	$15! = 1307674368000$

As you can see, the size of the solution space grows exponentially, so Brute Force is difficult to do even on the most modern super computer. For $N = 15$ to enumerate all $15! = 1307674368000$ permutations on the machine with the calculation speed of 1 billion (10^9) operations per second, and if to enumerate one permutation requires 100 operations, then we need 130767 seconds \approx 36 hours just to solve the problem with $N = 15!$.

If $N = 20$ then we may need **7645 years!**

However, it must be emphasized that in some optimization problems (TSP, Knapsack, VRP, ...) we have not found yet any effective methods in terms of optimal solution other than **Brute Force**.

2.1.1 Backtracking

The main idea of this algorithm is very simple: generate all possible solutions and check which solution satisfied the time window constraints. Then we find the best solution whose the length route is minimum among the satisfied constraints. The piece of code below demonstrates above idea:

```

1 def Try(k): #Backtracking
2     for i in range(1,n+1):
3         if mark[i] == 0:
4             res[k] = i
5             mark[i] = 1
6             if k == n:
7                 solution()
8             else:
9                 Try(k+1)
10            mark[i] = 0

```

We use a list `res` to build the solution and a list `mark[i]` to check whether the i^{th} element exists in the list `res` or not. If $k = n$ then we will check whether or not the current solution satisfies the time window constraints and is better than the best solution found so far by the function `solution()` :

```

1 def solution():
2     global minn_distance
3     time_current = t0
4     sum = 0 # total distance moved
5     for i in range(1, n+1):
6         if time_current + t[res[i-1]][res[i]] + d[res[i]] <= customer[res[i]][1]:
7             sum += c[res[i-1]][res[i]]
8             '''If the time at which the delivery man is ready to unload the good isn't reach the earliest
9             time of customer res[k], then he need to wait until time customer[res[k]][0]'''
10            if time_current + t[res[i-1]][res[i]] < customer[res[i]][0]:
11                time_current = customer[res[i]][0] + d[res[i]] #need to wait
12            else:
13                time_current += d[res[i]] + t[res[i-1]][res[i]]
14        else:
15            return
16    if minn_distance > sum:
17        minn_distance = sum
18        route.append([minn_distance, res[1:]])

```

Let have an example for implementing this checking function. From the test case mentioned in chapter 1.1.2 ([click here](#)), all the possible route generated will be: $\{(0, 1, 2); (0, 2, 1)\}$. Variable `time_current` is initialized as $t_0 = 7$

- Route (0, 1, 2):

- From $0 \rightarrow 1$:

$$\text{time_current} + t[0][1] + d[1] = 7 + 9 + 9 = 25 < 131$$

$$\text{sum} = 0 + 6 = 6$$

$\text{time_current} = 76 + 9 = 85$ (Since the time the delivery man comes to customer 1 $< e[1]$, so he need to wait until $e[1] = \text{customer}[1][0]$. After finishing unloading the good, the current time is equal to $e[1] + d[1]$)

– From 1 \rightarrow 2 :

```
time_current + t[1][2] + d[2] = 85 + 8 + 6 = 99 < 143
```

```
sum = 6 + 5 = 11
```

time_current = 85 + 8 + 6 = 99 (Since the time the delivery man comes to customer 2 $> e[2]$, so he does not need to wait and he can instantly unloading the good for customer 2. After finishing the service, the current time increases by $t[1][2] + d[2]$)

After evaluating the first solution, we have the min route length is 11 with route (0, 1, 2)

- Route (0, 2, 1) : Similarly, although this route does not violate time window constraints but its length is $18 > 11$

\Rightarrow The route (0, 1, 2) is the optimal solution.

The upper bound running time of **Backtracking algorithm** would be $O(n.n!)$ since the time taken to evaluate each permutation is linear in n (calculates the total travel time and distance for each permutation) and there are total $n!$ permutations.

2.1.2 Branch and Bound

Basically, in **Branch and Bound** algorithm, the idea of checking time window constraints and finding the route whose length is minimum among feasible solutions is similar to **Backtracking** algorithm. The biggest difference here is that: while **Backtracking** checks whether all the fully-constructed solutions satisfy the constraints or not, **Branch and Bound** checks the constraints simultaneously with the process of building solutions. When building the k^{th} element of a solution, this algorithm will check whether or not the current value of that element satisfies the **lower bound** and **upper bound** function. If this value passes both of these conditions, then we will go further in the current branch `Try(k+1)`, else cutting off this branch to save the running time

```

1 def Try(k):
2     for i in Value:
3         if exist[i] == False:
4             solution[k] = i
5             exist[i] = True
6             sum = update(sum)
7             time_current = update(time_current)
8             if k == n:
9                 if get_route_length(solution) < min_route_length and \
10                    check_constraints(solution) == True:
11                     min_route_length = get_route_length(solution)
12                     best_route = solution
13             else:
14                 if lower_bound(solution) == True and \
15                    upper_bound(solution) == True:
16                     Try(k+1)
17             sum = previous_value(sum)
18             time_current = previous_value(sum)
19             exist[i] = False
20             solution[k] = 0

```

Pseudo code for **Branch and Bound** algorithm

i. **Lower bound function:**

Denote $c_{min} = \min\{c[i][j]; \quad i, j = 1, 2, \dots, n; \quad i \neq j\}$: the smallest distance (cost) between all pairs of customer's positions.

We need to evaluate the lower bound for the partial solution $(0, u_1, u_2, \dots, u_k)$ corresponding to the delivery route that has pass k customers:

$$0 \rightarrow u_1 \rightarrow \dots \rightarrow u_{k-1} \rightarrow u_k$$

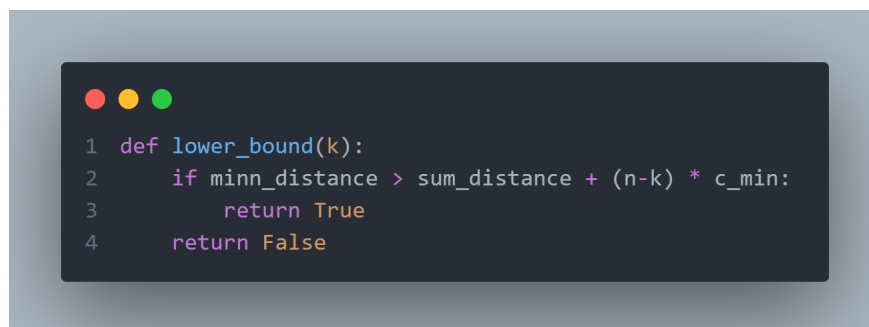
The cost need to pay for this partial solution is : $sum = c[0][u_1] + c[u_1][u_2] + \dots + c[u_{k-1}][u_k]$

To develop it as the complete journey:

$$\underbrace{0 \rightarrow u_1 \rightarrow \dots \rightarrow u_{k-1} \rightarrow u_k}_{\text{Cost: sum}} \rightarrow \underbrace{u_{k+1} \rightarrow u_{k+1} \rightarrow \dots \rightarrow u_n}_{\text{Cost: (n-k) } \times c_{min}}$$

We still need to go through $n - k$ segments, each segments with the cost at least c_{min} , thus the lower bound of the partial solution $(0, u_1, u_2, \dots, u_k)$ can be calculated by the formular:

$$g(0, u_1, u_2, \dots, u_k) = sum + (n - k) \times c_{min}$$



```

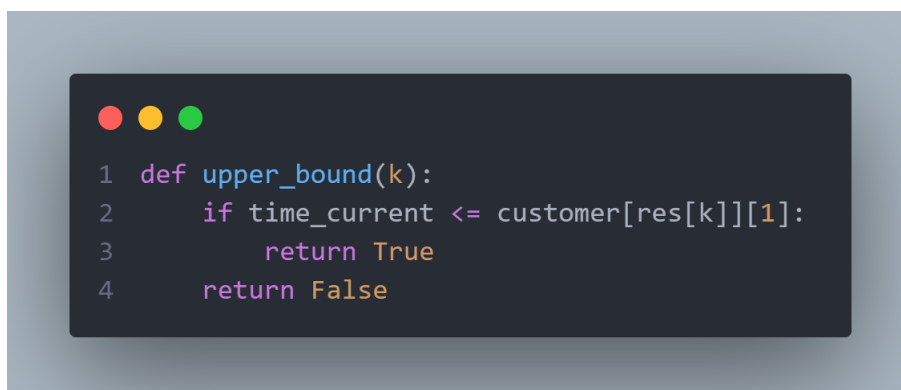
1 def lower_bound(k):
2     if minn_distance > sum_distance + (n-k) * c_min:
3         return True
4     return False

```

If the partial solution is *potential* (lower bound function < minimum route's length so far), then
return `True`, else `False`

ii. **Upper bound function:**

Similar to the time window check condition in the line 6 of the Python script above ([click here](#))



```

1 def upper_bound(k):
2     if time_current <= customer[res[k]][1]:
3         return True
4     return False

```

`time_current` now is the time when the delivery man finished the service for customer k

The running time of **Branch and Bound** algorithm is much better in compare with **Backtracking**, since it does not need to explore all branches. We can see how better it is in next chapter.

2.2 Greedy

i. Algorithm description:

Greedy algorithm is a simple, intuitive algorithm that makes a locally optimal choice in the hope that it leads to a globally optimal solution.

For this **VRPWTW** problem, a greedy algorithm would start at a given starting point and then always select the next closest customer's position which satisfies the time window constraints to delivery goods until all customers have received their goods.

However, this approach may not always lead to the optimal solution as it does not consider the overall global picture. That's why the result after running this algorithm for the current problem is not stable. It means that for small input size ($N = 2$ for example), just 2 test cases have the optimal solution which similar to Backtracking and Branch and Bound's results. The larger the value of N , the smaller the chance of finding even a feasible solution, let alone the optimal solution (With $N = 4$ or $N = 6$, just one nearly optimal solution is found. For $N \geq 8$, it is really difficult to find a feasible solution)

Test case Υ	$N = ? \Upsilon$	Output Υ	Running Time Υ	Nearly Optimal Route Υ
0	2	11	0	[0, 1, 2]
1	2	5	0	[0, 2, 1]
2	2	No feasible solution	0	No route
3	2	No feasible solution	0	No route
4	4	No feasible solution	0	No route
5	4	No feasible solution	0	No route
6	4	No feasible solution	0	No route
7	4	25	0	[0, 3, 2, 1, 4]
8	6	15	0	[0, 5, 1, 3, 6, 2, 4]
9	6	No feasible solution	0	No route
10	6	No feasible solution	0	No route
11	6	No feasible solution	0	No route
12	8	No feasible solution	0	No route
13	8	No feasible solution	0	No route
14	8	No feasible solution	0	No route
15	8	No feasible solution	0	No route
16	10	No feasible solution	0	No route
17	10	No feasible solution	0	No route
18	10	No feasible solution	0	No route
19	10	No feasible solution	0	No route
20	12	No feasible solution	0.02	No route

It is not worth to trade off good running time for inefficient output

ii. Algorithm implementation:

- Initialize the variables:

```

1 res = [0 for i in range(n+1)] #Store the solution
2 mark = [False for i in range(n+1)] #Check whether element i exist in the solution or not
3 time_current = t0
4 min_distance = 0

```

- Building the k^{th} element:

```

1 for k in range(1,n+1): #Building the k - element
2     feasible = [] #Store the values whose time window constraints are satisfied with the built solution
3     for val in range(1,n+1):
4         if mark[val] == False and check_TWC(time_current, res, k, val) == True:
5             feasible.append([val, c[res[k-1]][val] ])
6
7     if feasible == []: #After evaluating all the possible value, if there is no value is feasible -> return
8         return -1, -1
9     #else find the solution whose cost is the minimum
10    feasible.sort(key = lambda x: x[1])
11    min_distance += feasible[0][1]
12    res[k] = feasible[0][0]
13    mark[feasible[0][0]] = True
14    if time_current + t[res[k-1]][res[k]] < customer[res[k]][0]:
15        time_current = customer[res[k]][0] + d[res[k]] #need to wait
16    else:
17        time_current += d[res[k]] + t[res[k-1]][res[k]]
18
19    return min_distance, res

```

- Line 3 → 5: If the considering value satisfied the time window constraints, then it will be appended to `feasible` list.
- Line 7 → 8: If there is no feasible value for building k^{th} element, then quit this function (That is why it is really hard to find a complete solution)
- Line 10 → 17: If exists feasible solutions, then take the value whose distance from the previous element ($k - 1$) is minimum.

2.3 Local search - Hill Climbing

Hill climbing is a mathematical optimization technique which belongs to the family of local search. It is an iterative algorithm that starts with an arbitrary solution to a problem, then attempts to find a better solution by making an incremental change to the solution. If the change produces a better solution, another incremental change is made to the new solution, and so on until no further improvements can be found. Below are the main steps for this algorithm:

First, we need to calculate for the initial solution:

- Create the initial solution using Greedy algorithm (this will increase the quality of the search space rather than create the solution randomly)

- Calculate the length of this route
- Find all the neighbors of this solution
- Choose the best neighbor among the set of neighbors of current solution

Then we will loop until all of these conditions are satisfied: still found the best neighbor (a neighbor which satisfied the time window constraints and its route length is minimum among the set of neighbors), the found best neighbor is better than the current solution. If the loop is continue, current solution will be reassigned by the best neighbor, together with its related variables. Else, the current solution is the best one we can find.

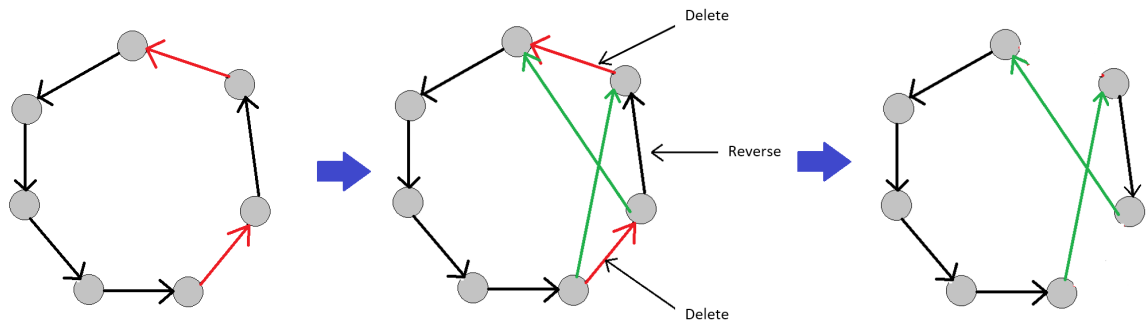
```

1 def Hill_Climbing():
2     # Calculating the initial solution
3     current_sol = randomSolution()
4     current_route_length = getRouteLength(current_sol)
5     neighbors = getNeighbors(current_sol)
6     best_neighbor, best_neighbor_route_length = getBestNeighbor(neighbors)
7     '''
8     Calculate the neighbor whose route_length is minimum - the best_neighbor
9     that we can find
10    '''
11    # Loop until sticking at local optimal solution
12    while best_neighbor != [] and (check_TWC(current_sol) == False or best_neighbor_route_length < current_route_length):
13        current_sol = best_neighbor
14        current_route_length = best_neighbor_route_length
15        neighbors = getNeighbors(current_sol)
16        best_neighbor, best_neighbor_route_length = getBestNeighbor(neighbors)
17
18    return current_sol, current_route_length

```

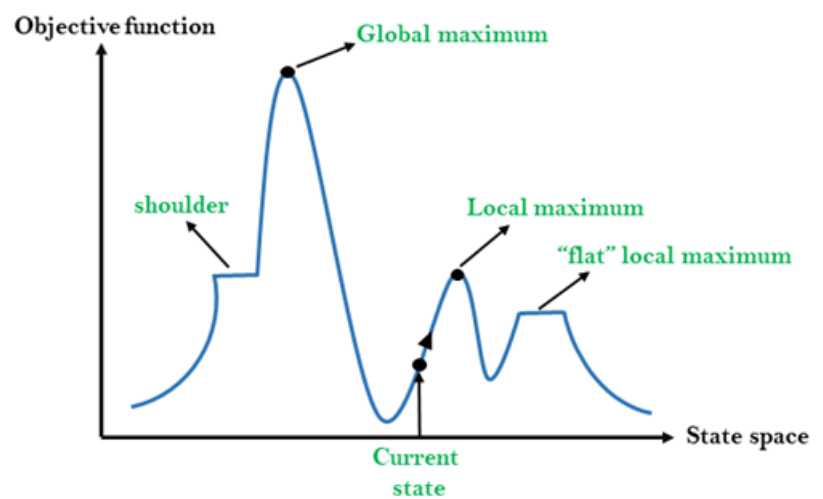
To know more detail about the functions, open `hill_climbing.py` in our repository

- Initial solution:** We have tried to generate the initial solution in two ways: randomly creating and using Greedy algorithm to find nearly optimal solution without time constraints. Of course using the route whose length is the best have better result in comparison with randomly creating, since the likelihood of "optimal" of the solution generated by Greedy is much higher than the random one.
- Get route length:** This function is just calculate the route's length of a particular route.
- Get neighbors:** There are some methods to get the neighbors of the current considering solution we have researched:
 - Swap/exchange: We trying to choose all the pair of customers and swapping them. Although this method is easy to code and understand, it is just appropriate for small N , since the larger the value of N , the larger the solution space, hence the neighbor space here is not enough diversity.
 - Random mutation: This method involves making small random changes to the current solution to generate new potential solutions.
 - 2-opt: For a current solution, this methods will randomly choose 2 edges which do not have the same vertex (customer). Then, it deletes those 2 edges while reversing the edge between. For example:



- iv. **Get best neighbor:** This function check all the neighbor found and choose the one which both satisfied the time window constraints and has the shortest route length among them.

The biggest problem of this algorithm is that, if the search space (neighbors space) of current solution is not diverse enough, algorithm will terminate at local optimum values, hence fails to find optimum solution. That is why we need to thoroughly consider the process of getting neighbors of current solution.



2.4 OR-TOOL with Constraint Programming (CP)

After mathematically model this problem for Constraint Programming, we use CP-SAT Solver provided by OR-Tools and follow these steps to get the optimal solution for the problem:

- Import the libraries
- Declare the model
- Create the variables for the problem
- Create the constraints
- Create the objective function
- Call the solver

Chapter 3

RESULTS

3.1 Experiment procedure

1. A computer with the following specification was used:
 - Processor: AMD Ryzen™ 5 5500U.
 - Graphics Card: AMD Radeon(TM) Graphics Vega 7
 - Hard Drive: SSD Western Digital Green 512GB M.2 NVME.
 - RAM: 16 GB DDR4 3200MHZ
 - OS: 64-bit Window 10 Home.
2. Python library requisition:
 - Python 3.10.0
 - Pandas 1.4.3
 - OR-Tools 9.5.2237
 - random

3.2 Experiment result

Here is the table of average running time in second by size of test cases' input:

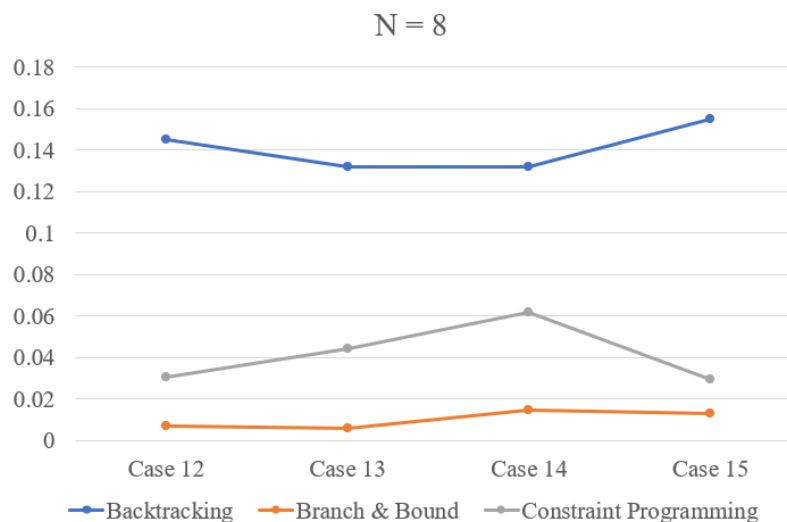
No	Algorithm	Average running time by size of input (second)							
		N = 2	N = 4	N = 6	N = 8	N = 10	N = 12	N = 14	N = 16
1	Backtrack	0.000	0.001	0.002	0.072	12.880	4118.000	> 2000	> 4000
2	Branch and Bound	0.000	0.001	0.001	0.010	0.185	3.538	25.924	> 500
3	Greedy	0.002	0.000	0.000	0.000	0.000	0.008	0.005	0.005
4	Hill climbing	0.000	0.000	0.015	0.115	0.773	4.078		
5	CP-SAT	0.020	0.023	0.024	0.054	0.086	0.379	16.820	>500

Backtracking, **Branch and Bound** and **CP-SAT** solver are able to find the optimal solution. The running time of these algorithms are nearly the same for small test cases ($N < 8$), which are less than 0.01 seconds. Only when $N > 10$ can we see a clear difference between **Backtracking** and the others. While the other exhaustive search algorithms just need less than 1 second to find the optimal solution if $N = 10$, **Backtracking** needs nearly 13 seconds. Since its running time grows exponentially, finding the

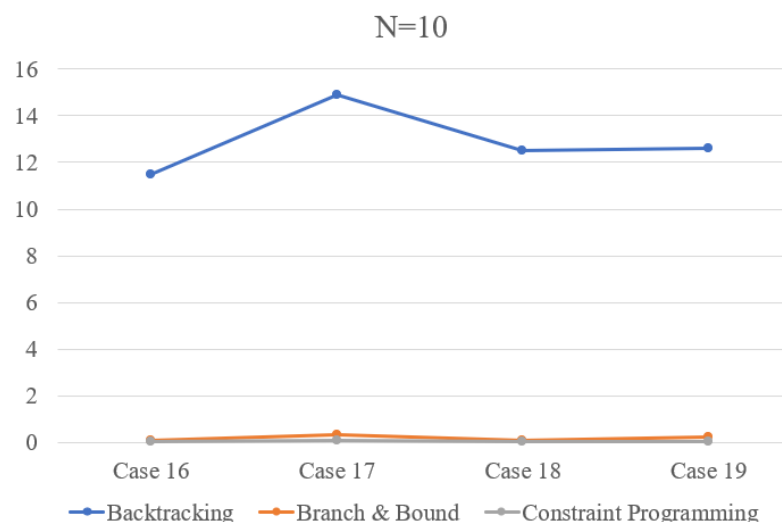
optimal solution for $N \geq 12$ can take us minutes, even hours. **Branch and Bound** uses lower bound and upper bound functions to prevent exploring branches that is not lead to the optimal solution, so this algorithm can save a lot of time in searching process.

The running time of **Hill Climbing** is also positive, but it can not surpass **Backtracking** for small value of test case in term of running time. For large test case ($N \geq 14$), it can not provide an feasible solution since the search space is not enough diverse (In `get_neighbors` function in `hill_climbing.py`, we have try 2 methods of generating neighbors: swapping 2 positions and 2-opt to ensure the diversity of the search space, that's why the running time is longer than **Backtracking** for small test case). For $N \geq 14$, we do not consider the running time here since this algorithm can not provide a feasible solution. The reason is that, our currently getting neighbors method is not enough effective for providing a diverse search space, and we are continue to upgrade it.

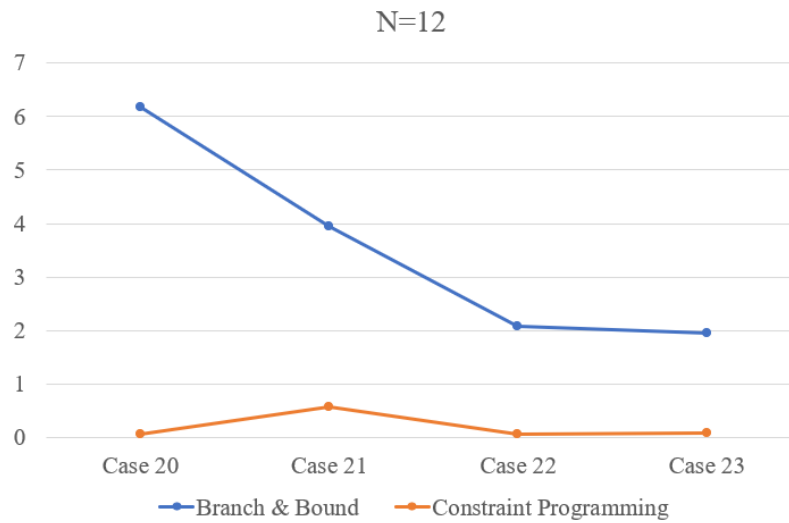
For easily evaluating the results, the line charts below compare the running time of the optimal - reached algorithms (Backtrack, Branch and Bound, CP):



For each value of N , we have generated 4 test cases for easy analysis. With $N = 8$, the difference between Backtracking and the others is clear. The way of using **Upper Bound** and **Lower Bound** function in **Branch and Bound** has decreased the running time by nearly 8 times, which is also the best one among 3 algorithms.



In this graph, the running time of **CP** and **Branch and Bound** are nearly the same (about 0.2s), while Backtracking takes an average 13s to run test cases of $N = 10$. The superiority of adding bounded functions is increasingly evident. So from $N \geq 12$, we do not consider **Backtracking** algorithm any more.



For $N = 12$, the running time of **CP** model is relatively stable and completely outstanding **Branch and Bound** method (0.379s vs 3.538s). This means that, the larger value of N , the more effective of **CP** model comparing with **Branch and Bound**.

Chapter 4

CONCLUSION AND POSSIBLE EXTENSIONS

In conclusion, the Vehicle Routing Problem with Time Windows (**VRPWTW**) is a challenging optimization problem that requires finding the optimal routes for the delivery man to serve a set of customers with time window constraints. In this report, we have explored and evaluated five different algorithms to solve this problem: backtracking, branch and bound, constraint programming, greedy and hill climbing.

Backtracking is a simple and intuitive algorithm, but its exhaustive search approach can be very time-consuming and not practical for larger instances of the **VRPWTW**. Branch and bound is a more efficient algorithm that takes advantage of problem structure to prune search spaces, but it still has limitations in terms of scalability.

Constraint programming provides a more flexible approach to model the problem and handle complex constraints, making it suitable for solving larger instances of the **VRPWTW**. However, its performance heavily depends on the quality of the model and the constraint solver used.

Hill climbing is a local search algorithm that can find good solutions quickly but may get stuck in local optima and miss the global optimal solution.

In addition to the four algorithms mentioned in the report, we also considered the greedy method for solving the **VRPWTW** problem. Although the greedy algorithm is known for its simplicity and fast running time, our evaluation shows that it cannot find feasible solutions for many instances of the problem.

However, the fast running time of the greedy algorithm makes it a good candidate for initializing the solution for other algorithms such as hill climbing. By starting with a feasible solution, hill climbing can then focus on finding the optimal solution by iteratively improving the current solution. This combination of greedy initialization and hill climbing can provide a more efficient and effective approach to solving the **VRPWTW** problem.

Overall, our evaluation shows that there is no algorithm that fit all solution for the **VRPWTW** problem, and the choice of algorithm depends on the size and complexity of the problem and the desired level of optimality. In the future, we will try to improve this problem using some other heuristic methods, such as simulated annealing or another way to modelling like mix-integer programming.

LIST OF TASKS

4.1 Programming tasks

- Duong Minh Quan : Coding the test generator, 5 algorithms above.
- Do Dinh Kien: Updating `ortool_CP.py` , `greedy.py` , testing and reporting bugs
- Vu Hoang Phuc: Testing and reporting bugs, updating `greedy.py` .

4.2 Analytics tasks

- Duong Minh Quan : Writing the report, modelling the problem.
 - Do Dinh Kien: Get insights from results, draw graphs, making slides, modelling the problem.
 - Vu Hoang Phuc: Making slides.
-