



NETWORK OPTIMIZATION USING PYTHON

PRACTICE SCHOOL – 1 REPORT



ANIRUDH A – 2018B4A70936H

BITS PILANI HYDERABAD CAMPPUS
PS Station: Ecom Express Pvt Ltd.

ACKNOWLEDGEMENTS

I would like to express my deepest appreciation to BITS Pilani to have provided me with the opportunity of doing a summer internship (in the form of PS-1) at Ecom Express Pvt Ltd, Delhi in the field of data analytics. A special gratitude I give to my instructor / faculty advisor from BITS Pilani, Dr. Ambatipudi Vamsidhar, whose contribution in stimulating suggestions and encouragement helped me to complete my project without going off track.

Furthermore, I would like to acknowledge with much appreciation the crucial role of my mentors in Ecom Express –Mr. Sanchit and Mr. Rahul, who gave valuable feedback and advice and suggested the necessary material to complete work on “*Network Optimization Using Python*”. I would like to thank the heads of the project - Mr. Bhupinder Singh and Mr. Nakshatra Goel (Ecom Express) who have invested their full effort in guiding the team in achieving the goal.

Last but not least, a special thanks goes to my parents for their valuable support and for providing me with the required resources which helped me do the work despite a ‘work from home’ scenario.

ABSTRACT

Ecom Express Pvt Ltd is a logistics solution provider with a focus on nation-wide quick and reliable express delivery service to e-commerce industry. The company is proud of their skilled labour force, expansive reach, speed and reliability in delivering products, dedicated fleet network and use of technology and automation. There are 3 services provided in general - express service, digital services and fulfilment services. Jabong, Amazon, Flipkart, FabIndia, PayTM, Myantra, Decathlon, Shoppers Stop etc are its customers while FedEx, Yodel Delivery Network and Heartland Express are its competitors based on criteria like revenue, gross profit, net income, number of employees, employee rating, social media reach etc.

The project given was 'Network optimization (Vehicle Routing Problem) using Python'. The goal was to develop a Python code that would give us the optimum travel route given a set of destinations such that the transportation cost is minimum. Minimizing the transportation cost would mean optimal scheduling and would therefore increase the revenue. The key areas of focus included TSP (travelling salesman problem), VRP (vehicle routing problem), PDP (Pick up delivery problem), OVRP (Open Vehicle Routing Problem) and the Python modules - OR Tools and Pyomo. The question to be answered was "What is the optimal set of routes for a fleet of vehicles to traverse in order to deliver to a given set of customers?"

Contents

ACKNOWLEDGEMENTS.....	1
ABSTRACT.....	2
INTRODUCTION.....	4
OPTIMIZATION CONCEPTS.....	5
Simplex Algorithm for Optimization Problems	5
Integer Programming.....	6
Dynamic Programming.....	7
Graphs and Networks	10
TSP, VRP and PDP	11
Travelling Salesman Problem.....	11
VRP and PDP	13
Formulation Of VRP	14
Formulation Of PDP	16
PYTHON CONCEPTS.....	18
Pyomo and OR Tools.....	18
Basics of Pyomo and OR Tools	19
CODE FOR TSP, VRP AND PDP	23
PDP and Tabu Search	23
Travelling Salesman Problem Using OR Tools.....	25
Vehicle Routing Problem Code	27
Capacity Constraints	28
Time Windows	30
Pickup Delivery Problem	32
Transportation Problem (Without OR Tools).....	35
INVENTORY CONSTRAINTS.....	37
SOFT SKILLS LEARNT AND CHANGES IN PERSONALITY	40
SUMMARY AND CONCLUSION	41
RECOMMENDATIONS - PATH AHEAD (Topics to Explore)	41
REFERENCES.....	42

INTRODUCTION

The Vehicle Routing Problem (VRP) is a generic name given to a set of problems in which set of routes for a fleet of vehicles based at one or several depots are to be formed for servicing the customers dispersed geographically. The objective of VRP is to form a route with lowest cost to serve all customers.

The vehicle routing problem (VRP) is a **combinatorial optimization** and **integer programming problem** which asks "What is the optimal set of routes for a fleet of vehicles to traverse in order to deliver to a given set of customers?". It generalises the well-known **travelling salesman problem (TSP)**. Often, the context is that of delivering goods located at a central depot to customers who have placed orders for such goods. The objective of the VRP is to minimize the total route cost.

More than 50 years have elapsed since Dantzig and Ramser introduced the VRP in 1959. They proposed the first mathematical programming formulation and algorithmic approach. They also described VRP with a real-world application concerning the delivery of gasoline to service stations. Clarke and Wright (1964) proposed an effective greedy heuristic that improved on the Dantzig-Ramser approach. After these two papers, many models and algorithms are proposed for the optimal and approximate solution of the different versions of the VRP (Toth and Vigo 2002).

Determining the optimal solution to VRP is NP-hard, so the size of problems that can be solved optimally, using mathematical programming or combinatorial optimization may be limited. Therefore, commercial solvers tend to use heuristics due to the size and frequency of real world VRPs they need to solve. The VRP has many obvious applications in industry. In fact, the use of computer optimization programs can give savings of 5% to a company as transportation is usually a significant component of the cost of a product.

OPTIMIZATION CONCEPTS

The key optimization concepts the were learnt and used are: -

1. Simplex method for optimization
2. Integer programming (Branch and bound algorithm, cutting plane algorithm).
3. **Dynamic Programming**
4. Applications of graphs (Prim's and Kruskal's algorithm)
5. **TSP** (travelling salesman problem)
6. Saving's algorithm (greedy algorithm)
7. **VRP and PDP**

Simplex Algorithm for Optimization Problems

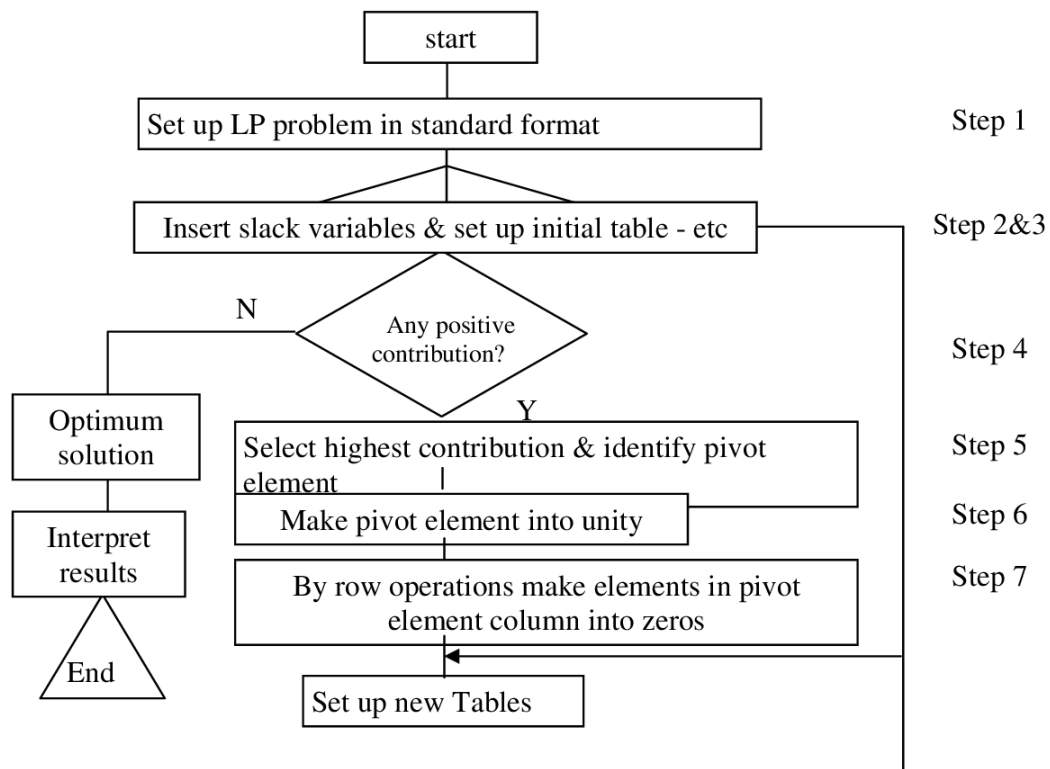
Simplex method is the most basic method and is the building block of all other algorithms and methods. It involves the use of linear algebra and properties matrices.

Steps Involved: -

1. Write all constraints in equation form, add slack or surplus variables if needed and build the simplex table. All variables should be of ≥ 0 type.
2. Classify variables as basic and non-basic. Identity matrix - [I] should be present under the basic variables column and Z rows should have 0's under the Basic variable.
3. For a maximization problem, most -ve variable of z row enters and the row minimum ratio with respect to the solution column enters. Process continues till Z row has only +ve values.
4. Opposite happens for minimization problems.
5. The general form of a simplex table is shown below.

$$\text{Max/Min } Z = AX \text{ subject to } AX = b, X \geq 0$$

Basic Variables	x		Solution
	Basic	Non - basic	
Z	0	$C_B B^{-1} A_j - C_j$	$C_B B^{-1} b$
X_B	[Id]	$B^{-1} A_j$	$B^{-1} b$

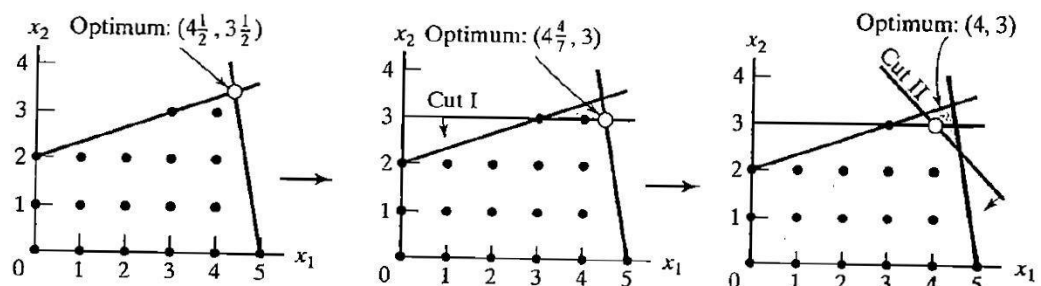


When constraints are of $=$ or \geq form, we can not apply the simplex method directly and artificial variables are introduced. It calls for the 2-phase method. The phase-1 involves the introduction of artificial variables, and minimizing them (forcing them to 0). The 2nd phase involves the use of simplex method directly for the objective function but by using the optimum table from the 1st phase.

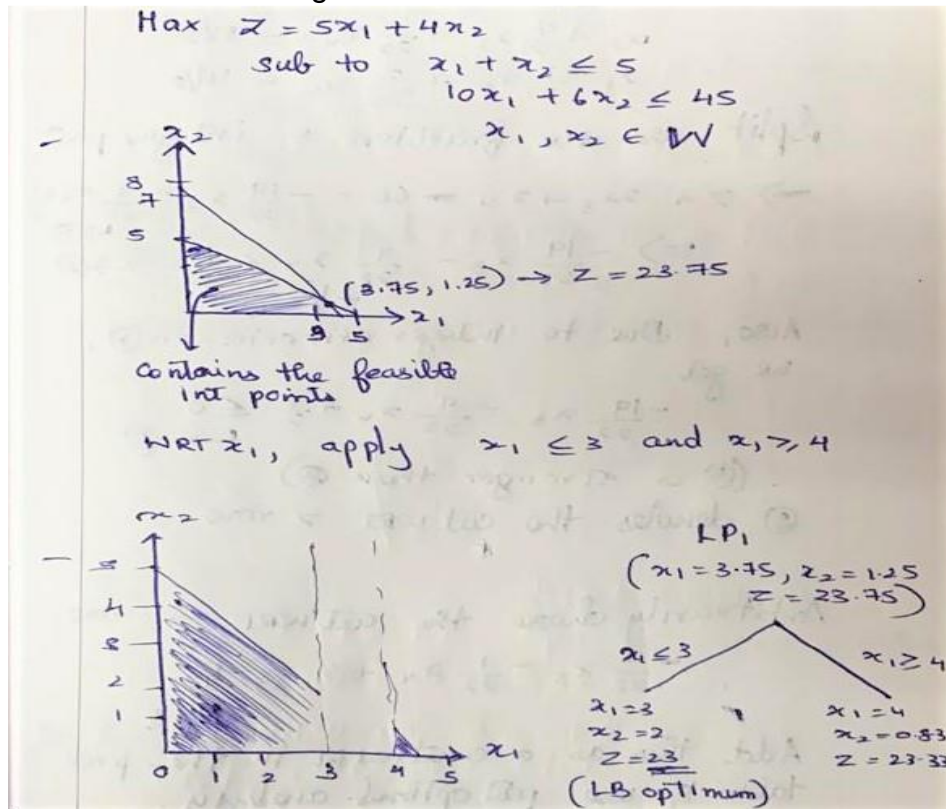
Integer Programming

The integer programming problem can be solved using two method – branch and bound algorithm or the cutting plane algorithm. Both the methods make use of the simplex method to solve the optimization problem first by relaxing the integer constraints. Once a solution is got, the integer constraints are applied to it.

1. Cutting Plane Algorithm: -



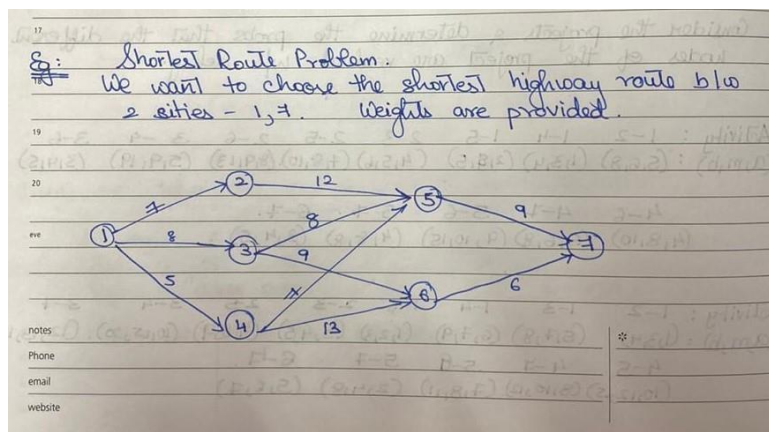
2. Branch and Bound Algorithm: -



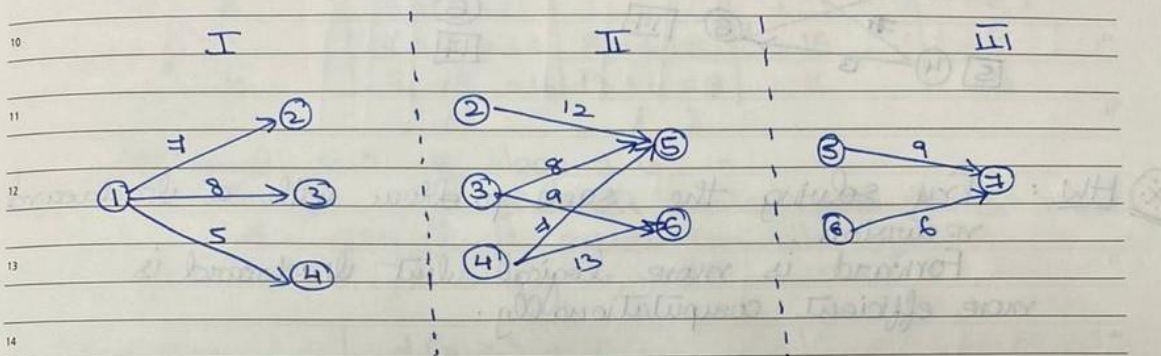
Dynamic Programming

It forms the basics of VRP, PDP, TSP etc. and works on the simple principle that keeping the past iterations in memory would lead to quicker computation. It is a recursive algorithm. For manual computations, the forward recursion is used as it is logical but the backward recursion is used for computational purposes as it has a better time complexity.

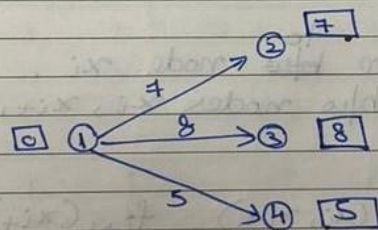
An example of how the dynamic programming can be used to solve the shortest routing problem is shown below.



→ Here 5 routes are given, for longer paths, we may not be able to enumerate & solve exhaustively.
In DP, we decompose the network into subproblems & solve.



I :-



II :-

End nodes are 5, 6.

5 can be reached as (2, 5) (3, 5) or (4, 5).

We find the shortest cumulative distance.

$$= \min(7+12, 8+8, 5+7) = 12.$$

i.e., via node 4.

III²⁴ for node 6, we get the distance as 17 (from node 3).

notes

Phone

email

website

III. To node 7, we have $\min(9+12, 6+17) = 21$.

∴ Shortest route = ① - ④ - ⑤ - ⑦

The backward recursion is used because it can be extended to Knapsack model, Stage coach problem, Single depot VRP etc. Moreover, in real time, there could be many more stops/depots making it difficult to compute manually. The backward recursive approach is shown below.

* Notation:-

- $f_i(x_i)$ = shortest distance ~~to~~ node x_i .
- $d(x_i, x_{i+1})$ = distance b/w nodes x_i, x_{i+1} .

In backward recursion,

$$f_i(x_i) = \min(d(x_i, x_{i+1}), f_{i+1}(x_{i+1})).$$

The order of computation is like $f_3 \rightarrow f_2 \rightarrow f_1$.

Eg: In previous question:-

- Stage 3: Node $x_4 = 7$ is connected to node 5 and 6 ($x_3 = 5, 6$) with exactly one route each.

Assume $f_4(x_4 = 7) = 0$. Then,

x_3	$d(x_3, x_4)$ ($x_4 = 7$)	output sol ⁿ $f_3(x_3)$	x_4^*
5	9	9*	7
6	6	6	7

• III Ex, we draw for stage 2.

$x_2 \backslash x_3$	$d(x_2, x_3) + f_3(x_3)$	$f_2(x_2)$	x_3^*
	5	6	
2	$12 + 9 = 21$	21	5
3	$8 + 9 = 17$ $9 + 6 = 15$	15	6
4	$7 + 9 = 16$ $13 + 6 = 19$	16	5

optimal solⁿ of stage 2 is :-

For ②, ④, shortest routes pass thr' ⑤ while for city ③ it passes thr city ⑥.

Stage 1:-	$d(x_1, x_2) + f_2(x_2)$			$f_1(x_1)$	x_2^*
$x_1 \backslash x_2$	2	3	4		
1	$7 + 21$	$8 + 15$	$5 + 6 = 21$	21	4

Stage 1: Links ①, ④

2: Links ④, ⑤

3: Links ⑤, ⑥

∴ 1, 4, 5, 7
 distance = 21 miles

Both the approaches give us the same final output.

Graphs and Networks

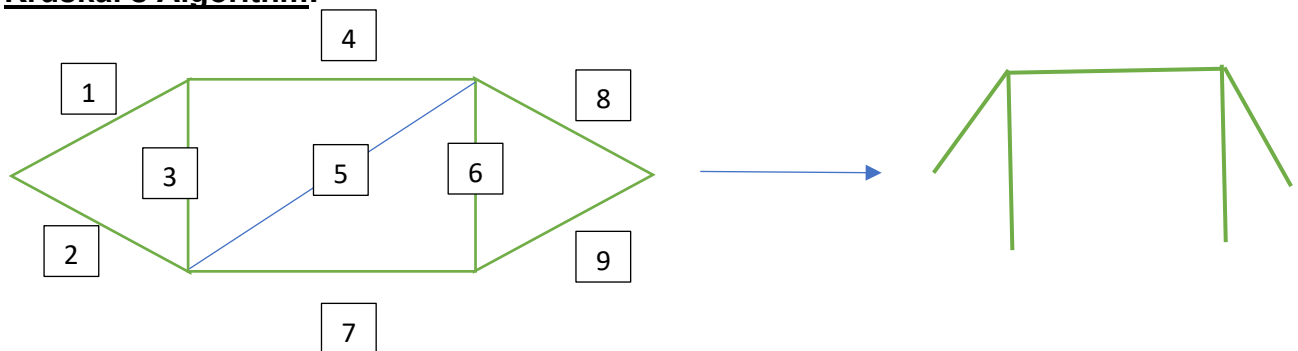
A graph is usually denoted by $G(V, E)$ where V is the set of vertices and E is the set of edges. A vertex is used to denote a node (a depot/city/destination). An edge is used to denote the weights (distance/cost) between two nodes.

A tree is a connected graph which does not contain any cycles. Hamiltonian path: A path that goes through **all** vertices of a graph **exactly once**. A cycle in a graph that contains every vertex is called a Hamiltonian cycle.

In case of PDPs, our goal is to find a Hamiltonian cycle while we want to find a Hamiltonian path in case of OVRPs.

The minimum spanning tree problem can be solved using the Prim's or Kruskal's algorithm.

Kruskal's Algorithm: -



It is an edge-based algorithm.

- First, choose the edge e_1 in G which is of minimum weights.
- Next, choose the edge e_2 in G with second minimum weights and keep continuing.
- While each edge is chosen, make sure that it does not form a cycle.
- Stop once a spanning tree is got.

Prim's Algorithm: -

It is a vertex-based algorithm.

- Take any v , e with minimum cost such that e is adjacent on v .
- For subsequent edges, choose the edge with minimum weights among those edges having exactly 1 of its vertices incident with an edge already selected.
- Stop once a spanning tree is got.

TSP, VRP and PDP

The travelling salesman problem asks the following question: "Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city and returns to the origin city?" It considers the use of just one vehicle.

When we have a fleet of vehicles, the TSP transforms into the vehicle routing problem. The vehicle routing problem could have additional constraints like the capacity constraint, time windows constraint, etc.

One important variant of the VRP is the Pickup and Delivery Problem (PDP). Unlike the classical VRP, where all customers require the same service type, a central assumption in the PDP is that there are two different types of services that can be performed at a customer location, a *pickup* or a *delivery*.

In all these cases, there exists another extra constraint that the vehicle starts and ends at the same depot. If this condition is relaxed, it becomes the open vehicle routing problem (OVRP).

Travelling Salesman Problem

1. Transportation Problem: -

	DE	M	Supply
LA	x_{11} (108)	x_{12} (215)	1000
D	x_{21} (100)	x_{22} (108)	1500
NYC	x_{31} (130)	x_{32} (190)	1200
Demand	2300	1400	3700

- Create an adjacency matrix using the available data.
- Determine the basic feasible solution using one of North-West Corner method, Least Cost method or **Vogel Approximation method** by using optimality conditions.
- Get the best solution by performing revised simplex method to the obtained matrix.

2. Assignment Problem: -

It is ideally used to allocate jobs to workers to get the optimal output. But, it can be treated as a variant of the transportation problem by considering workers = sources and jobs = destination.

Steps Involved: -

1. Obtain the adjacency matrix.
2. Find each row min and subtract it from all members of that row. Do the same for all columns. (Similar to a nested-for loop)
3. Make the assignment (assign each 0 to the worker). This minimizes the cost.
4. Use matrix manipulation if step 3 doesn't give the complete solution.

3. Travelling Salesman Problem: -

“What is the optimal set of routes for a fleet of vehicles to travers in order to deliver to a given set of customers?” The vehicle starts at a depot, visits each customer exactly once, and returns back to the depot. The goal thus is to find a Hamiltonian cycle of minimum length.

Formulation: -

Labelling - $d_{i,j}$ = distance between 2 nodes i, j . If $i=j$, set $d_{i,j}$ = infinity.

Objective function: -

$$\text{Minimize } Z = \sum_{\forall i} \sum_{\forall j} d_{i,j} x_{i,j}$$

$$\text{where } x_{i,j} = \begin{cases} 1, & \text{if } j \text{ is reached from } i \\ 0, & \text{else} \end{cases}$$

Subject to the constraints: -

$$\sum_{i=1}^n x_{i,j} = 1, \forall j \text{ and } \sum_{j=1}^n x_{i,j} = 1, \forall i$$

(i.e., each city is to be visited exactly once).

Steps Involved: -

- First form the adjacency matrix and apply the transportation or assignment model.
- If the assignment/ transportation model gives us a “tour”, then we are done.
- If we get subtours instead, we need to apply the integer programming algorithms (B&B or Cutting Plane algorithm) until we get a “tour”.

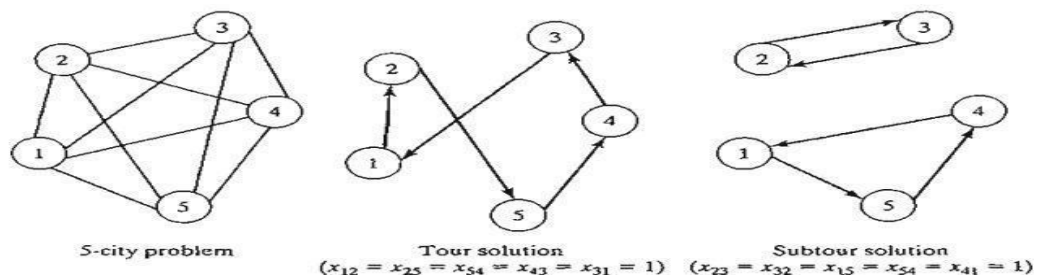


FIGURE 9.11

A 5-city TSP example with a tour and subtour solutions of the associated assignment model

4. Heuristics Method: -

- 1) Nearest neighbour method - **The nearest neighbour method is used first and the solution is improved by using the sub-tour reversal method.**

- Start from an arbitrary node and keep adding the closest node to the list.
- The answer depends on the initial node.

$$d_{i,j} = \text{infinity if } i = j \quad \begin{pmatrix} \text{inf} & \dots & 210 \\ \vdots & \ddots & \vdots \\ 87 & \dots & \text{inf} \end{pmatrix}$$

Say, we start from node 3. We would get 3 -> 2 -> 4 -> 1 -> 5 -> 3 and a total distance of 735km.

- 2) Sub-tour reversal Heuristic - Start with a feasible tour got from 1) and improve by applying 2-city reversals, 3-city reversals and so on.

Say, we have a feasible tour 1 -> 4 -> 3 -> 5 -> 2 -> 1 (745km).

2 - city reversals: -

1 -> 3 -> 4 -> 5 -> 2 -> 1 (4-3)

1 -> 4 -> 5 -> 3 -> 2 -> 1 (3-5)

1 -> 4 -> 3 -> 2 -> 5 -> 1 (5-2)

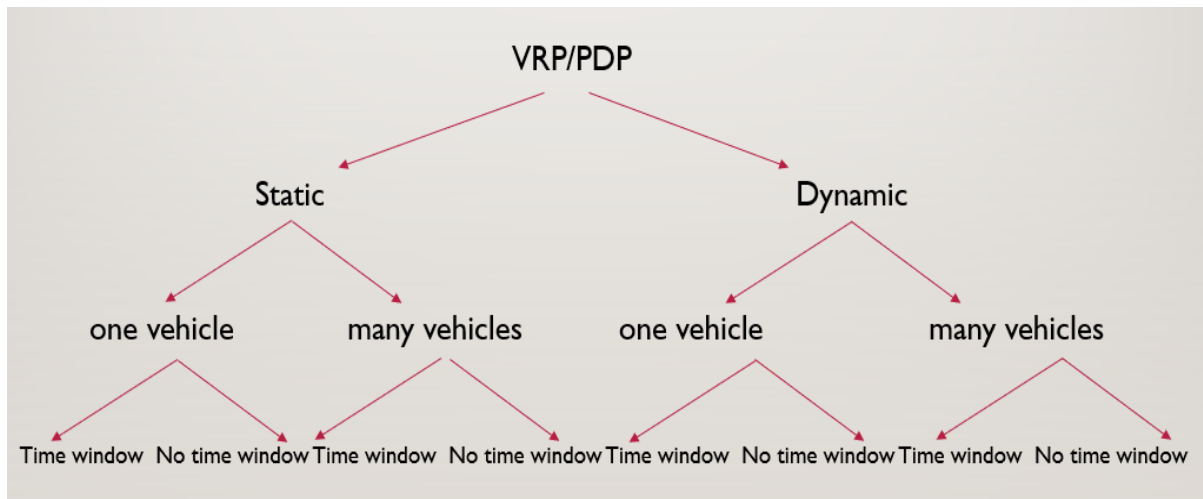
The best of these is chosen and 3 - city reversal is applied. The process continues until we get no better solutions.

Why so many methods?

- In machine learning terms, these problems are NP hard, i.e., it is not possible to get one solution which is the most accurate. So, the interest lies in getting a solution and further improving on it.
- All methods work differently and might produce different results for the optimal path. Each method has a "optimal working capacity". For example, dynamic programming approach is effective for a fleet of 10-20 trucks while the heuristics method is bested suited for larger number of vehicle sets (due to their $O(n)$ values).
- The shortest path need not be the most cost efficient (for political reasons, infrastructure like roads and bridges, weather etc). So, it is better to have more than one route in hand.

VRP and PDP

- The goal in both the problems is to minimize the distance travelled/cost.
- The vehicle routing problem (VRP) is a combinatorial optimization and integer programming problem which asks "What is the optimal set of routes for a **fleet of vehicles** to traverse in order to deliver to a given set of customers? ".
- Unlike VRP, where all customers require the same service type, a central assumption in the PDP is that there are two different types of services that can be performed at a customer location, a **pickup or a delivery**.



Dynamic case can be considered as many static cases. If there is no time window, the time constraints can be relaxed.

Once the formulation is done and once we have accounted for the specifications (if any), we next have to solve the problem using any of the methods we have already seen. Heuristics method is preferred in most cases.

Formulation Of VRP

Labels: -

$N = \{1, 2, \dots, n\}$ = set of customers.

K = set of vehicles.

$o(k)$ = origin, $d(k)$ = destination depot.

For each customer i , the time window is $[a_i, b_i]$ and the demand is p_i .

Load parameter $l_i = p_i$ (becomes d_i if it is to be dropped).

$l_{o(k)}$ = load at origin

Travel time = t_{ij}^k (includes service time)

Cost = c_{ij}^k

Q_k = capacity of each vehicle.

Flow variable = $x_{i,j}$ where $x_{i,j} = \begin{cases} 1, & \text{if } j \text{ is reached from } i \\ 0, & \text{else} \end{cases}$

Load variable = load after service at node $i = L_i^k$

Time variable = time at start of service at node $i = T_i^k$

Objective function: -

$$\text{Minimize } Z = \sum_{\forall k} \sum_{\forall (i,j)} c_{ij}^k x_{ij}^k$$

Constraints: -

Each customer is to be visited exactly once.

$$\sum_{\forall k} \sum_{\forall j \text{ in } NU\{o(k)\}} x_{ij}^k = 1 \quad \forall i \text{ in } N.$$

Flow constraints.

- $\sum_{\forall k} \sum_{\forall j \text{ in } N} x_{o(k),j}^k \leq v$
- $\sum_{\forall j} x_{o(k),j}^k = 1 \quad \forall k$
- $\sum_{\forall i \text{ in } NU\{o(k)\}} x_{i,j}^k - \sum_{\forall i \text{ in } NU\{d(k)\}} x_{i,j}^k = 0 \quad \forall k$
- $\sum_{\forall i} x_{i,d(k)}^k = 1.$

Time constraints.

- $a_i \leq T_i^k \leq b_j$
- $x_{i,j}^k (T_i^k - T_j^k + t_{i,j}^k) \leq 0 \quad \forall k$

Capacity constraints.

- $x_{i,j}^k (L_i^k - L_j^k + l_i) \leq 0 \quad \forall k$
- $l_i \leq L_i^k \leq Q_k \quad \forall k$
- $L_{o(k)}^k = l_{o(k)} \quad \forall k$

Binary constraints.

- $x_{ij}^k \geq 0 \quad \forall k$
- $x_{ij}^k = \text{binary} \quad \forall k$

Assumptions: -

- Travel time is inclusive of the service time (time taken for service at each node).
- No penalty on waiting time.

Logic: -

- An arc (i, j) can be eliminated if $l_i + l_j > Q_k$ or if $a_i + t_{ij}^k > b_j$.
- Eliminating the capacity constraints would give us mTSP (multi-vehicle TSP).
- Quality of service increases with a decrease in time window.

Formulation Of PDP

Labels: -

$P = \{1, 2, \dots, n\}$ = set of pick up nodes

$D = \{1+n, 2+n, \dots, 2n\}$ = set of pick up nodes

$N = P \cup D$

Each request is associated with 2 nodes – (i, n+i)

K = set of vehicles.

$o(k)$ = origin, $d(k)$ = destination depot.

For each customer i , the time window is $[a_i, b_i]$.

Request i = Transport d_i units from node i to $n+i$. Put $l_i = d_i$ and $l_{i+n} = -d_i$

$l_{o(k)}$ = load at origin

Travel time = t_{ij}^k

Service time = s_i

Cost = c_{ij}^k

Q_k = capacity of each vehicle.

Flow variable = $x_{i,j}$ where $x_{i,j} = \begin{cases} 1, & \text{if } j \text{ is reached from } i \\ 0, & \text{else} \end{cases}$

Load variable = load after service at node $i = L_i^k$

Time variable = time at start of service at node $i = T_i^k$

Objective function: -

$$\text{Minimize } Z = \sum_{\forall k} \sum_{\forall (i,j)} c_{ij}^k x_{ij}^k$$

Constraints: -

Each customer is to be visited exactly once.

- $\sum_{\forall k} \sum_{\forall j \in NU\{d(k)\}} x_{ij}^k = 1 \quad \forall i \in N.$
- $\sum_{\forall j \in N} x_{i,j}^k - \sum_{\forall j \in N} x_{i+n,j}^k = 0 \quad \forall k$

Flow constraints.

- $\sum_{\forall j \in PUd(k)} x_{o(k),j}^k = 1 \quad \forall k$
- $\sum_{\forall i \in NU\{o(k)\}} x_{i,j}^k - \sum_{\forall i \in NU\{d(k)\}} x_{i,j}^k = 0 \quad \forall k$
- $\sum_{\forall i} x_{i,d(k)}^k = 1.$

Time constraints.

- $a_i \leq T_i^k \leq b_j$
- $x_{i,j}^k (T_i^k - T_j^k + t_{i,j}^k + s_i) \leq 0 \forall k$

Capacity constraints.

- $x_{i,j}^k (L_i^k - L_j^k + l_j) \leq 0 \forall k$
- $l_i \leq L_i^k \leq Q_k \forall k$
- $L_{o(k)}^k = l_{o(k)} \forall k$

Binary constraints.

- $x_{i,j}^k \geq 0 \forall k$
- $x_{i,j}^k = \text{binary} \forall k$

Some specific constraints.

- $T_i^k + t_{i,n+i}^k \leq T_{i+n}^k$ (pick up node before arrival node)
- $L_{o(k)}^k = 0$ (initial load constraint).

Assumptions: -

- Travel time is inclusive of the service time (time taken for service at each node).
- No penalty on waiting time.
- Sets P and D are disjoint.

Logic: -

- An arc (i, j) can be eliminated if $l_i + l_j > Q_k$ or if $a_i + t_{i,j}^k > b_j$.
- Quality of service increases with a decrease in time window.
- The maximum time duration of a route could be $b_{d(k)} - a_{o(k)}$.
- Arrival time at node j –
Put $x_{i,j}^k = 1$ in the time constraint. $T_j^k = \max \{a_j, T_{i,j}^k + t_{i,j}^k + s_i\}$

PYTHON CONCEPTS

Some of the basic Python topics covered are listed below.

- Numbers
- Conditionals and Loops
- Functions
- Strings
- Data Structures (files, lists, dictionaries, tuples).
- OOP (object-oriented programming), Decorators and Generating functions.
- GUI Basics (Graphical User Interface)
- Working with Jupyter notebook and text editors.
- How to install various libraries.

Some of the frequently used Python libraries that were covered are listed below: -

- Matplotlib – For data visualization (graphs, bar graphs, histograms, scattered plots, pie charts etc).
- Pandas – For data analysis
- Numpy – For arrays, random numbers generation, mathematical operations etc.
- Seaborn – Similar to matplotlib but enables statistical graphics as well

Pyomo and OR Tools

- Pyomo is an object model for describing optimization problems.
- Pyomo objects exist within `coopr.pyomo` name space.
- Installation of Pyomo: -
`conda install -c conda-forge pyomo`
- Importing pyomo: -
`from pyomo.environ import *`
- How is it different from python?
Unlike Python, the type declaration has to be given by the coder while writing the program.
The 'Set' command of Pyomo is different from the 'set' command in Python. Semi-colons may be used in *model data* whereas Python doesn't use semi-colons in general.

1. Defining the Model: -

Pyomo makes use of 2 types of models.

Concrete models: -

- a. Immediately constructed.
- b. Data must be present/available when components are declared.
- c. Straightforward logical process; easy to script.

Abstract models: -

- a. 2-pass construction
- b. model first, then data
- c. Familiar to modelers with experience with AMPL
- d. Pyomo stores the basic model declarations, but does not construct the actual objects. Details on how to construct the component are hidden in functions, or *rules*.

Defining the model: -

```
model = ConcreteModel ()  
model = AbstractModel ()
```

2. Declaring Variables: -

The 'Var' keyword is used. When a variable is declared, one could additionally put in a few restrictions as shown below.

- `model.x=Var(within = NonNegativeReals)`
- `model.y=Var(bounds=(0, None))`
- `model.z=Var(initialize = 1.0, bounds = (-2,2))`

3. Declaring Objective Functions and Constraints: -

They are declared similar to variables. The default objective function is minimization and it could be changed by explicitly calling for maximization.

```
model.abc=Objective(expr= (2-model.x)**2 + 100*(model.y)**2,  
sense= maximize)
```

While declaring the constraints, 'expr' is used which can hold expressions as well as functions.

- `model.constr=Constraint(expr = model.x+ 5*model.y <= model.z)`
- `model.constr= Constraint(expr=(None, model.x+model.y,10))`

4. Constraints and Variables as Lists: -

Constraints and variables can be declared as a list as indexing would provide easier access and storage. Examples are shown below.

```
➤ model.xyz=ConstraintList()
  model.xyz.add(30*model.a + 15*model.b + model.c <=10)

➤ model.a=Var(IDX)
  model.b=Var(IDX_A, IDX_B)
```

Once indexed, they can be manipulated as follows.

```
model.IDX=range(10)

model.a=Var()

model.b=Var(model.IDX)

model.constr = Constraint ( expr = sum( model.b[i] for i
in model.IDX <= model.a)
```

This is same as writing $\sum_{i \text{ in } \text{IDX}} b_i \leq a$.

5. Helper Functions: -

- `display(model)` – displays the entire model
- `display(model.x)` – displays the setup of variable x
- `value(model.x)` – displays the value of variable x
- `sequence(start, stop, step)` - generates sequences(lists) that are an arithmetic progression of integers.

6. Pyomo Sets: -

Pyomo Sets are also iterable and are declared as shown below. The difference is that unlike indices, Sets can be initialised from any iterable.

```
model.IDX= Set (initialize = [1,2,5])

#Here the indices are 1,2,5.
```

Range Sets: -

For OR purposes, the 'up to but not including rule' is NOT followed.

```
model.IDX= Set(initialize = range(5))

# This gives [0,1,2,3,4]. (Like in Python)

model.IDX= RangeSet(5)

# This gives [1,2,3,4,5].
```

7. Abstract Modelling and 'Rules': -

```
model.IDX= Set(initialize = range(5))
model.a=Var(model.IDX)
model.b=Var()
• def c4_rule(model,i):
    return model.a[i]+model.b <=1
    model.c4 = Constraint(model.IDX, rule=c4_rule)
• model.IDX2 = model.IDX * model.IDX
    def c5_rule(model,i,j,k):
        return model.a[i] + model.a[j] + model.a[k] <=1
    model.c5 = Constraint (model.IDX2, model.IDX, rule =c5_rule)
```

8. The Summation Operator: -

- Summation can be used instead of the 'sum' operator and it has a simpler syntax.

```
model.N = Set (initialize = [1,2,3])
model.x = Var( model.N, within=NonNegativeReals)
summation(model.x)
```

This gives $\sum_{i=1}^3 x_i$.

- Comparing 'sum' and 'summation' – Both the lines of code below are equivalent.

```
➤ model.c= Constraint (expr = sum (model.x[i] for i in
    model.N)<=0)
➤ model.c= Constraint (expr = summation (model.x) <=0)
```

- 'denom' can be used to have a denominator. The following gives $\sum x_i/y_i$.

```
model.c= Objective (expr = summation (model.x, denom=model.y))
```

- When used between 2 arrays, the 'summation' operator acts as a 'dot product'.

```
model.a= Objective (expr=summation (model.x, model.y))
```

9. The First Pyomo Code: -

Write a programme to choose P warehouses from N that minimizes the total cost of serving all M customers where $d_{n,m}$ is the cost of serving customer 'm' from warehouse location 'n'.

The above question can be mathematically formulated as shown below.

Minimize $Z = \sum_{\forall n,m} d_{n,m} x_{n,m}$

Subject to: -

$$\sum_{\forall n} x_{n,m} = 1 \quad \forall m$$

$$x_{n,m} \leq y_n \quad \forall m,n$$

$$\sum_{\forall n} y_n = P$$

$$0 \leq x \leq 1$$

$$y \in \{0,1\} - \text{binary}$$

```
from pyomo.environ import *
#enter sample values or take them as an input.
#define the adjacency matrix as a dictionary.
N=3
M=4
P=3
d = {(1,1): 1.7, (1,2): 7.2, ..., (3,4): 9.3}
#Could also be generated randomly or taken as an input
model= ConcreteModel()
model.loc = range (N)
model.cust = range (M)
model.x = Var(model.loc, model.cust, bounds= (0.0,1.0)) #Location
x customers - 2 indices
model.y = Var(model.loc, within= Binary)
model.obj = Objective(expr= sum(d[n,m]*model.x[n,m] for n in
model.loc for m in model.cust) #Double summation - across locations
and customers.
➤ model.single_x= ConstraintList()
    for m in model.cust:
        model.single_x.add(sum(model.x[n,m] for n in model.loc)==1.0)
➤ model.bound_y=ConstraintList()
    for m in model.loc:
        model.bound_y.add(model.x[n,m]<=model.y[n])
➤ model.num_facility=Constraint( expr= sum(model.y[n] for n in
model.loc)==P)
```

10. Some Useful OR Tools Operations: -

- The method `manager.IndexToNode` converts the solver's indices to the numbers for locations.
- `transit_callback_index` creates a distance call back function and registers it with the solver.
- The `arc cost evaluator` tells the solver how to calculate the cost of travel between any 2 nodes.
- The method `SetGlobalCostCoefficient` sets a large coefficient for the global span of routes (by default, it is the maximum distance among the routes).

CODE FOR TSP, VRP AND PDP

A sample data was taken from the internet and codes for TSP, VRP and PDP (including time window and capacity constraints) were written using OR Tools.

Screenshots of important parts of the code are shown.

PDP and Tabu Search

VRP is an important problem in the fields of transportation, distribution and logistics. Often the context is that of delivering goods located at a central depot to customers who have placed orders for such goods. The pickup and delivery problem with time windows (PDPTW) is a generalization of the VRP which is concerned with the construction of optimal routes to satisfy transportation requests, each requiring both pickup and delivery under capacity, time window and precedence constraints.

Tabu Search: -

It uses a local or neighbourhood search procedure to iteratively move from one potential solution x to an improved solution x' in the neighbourhood of x , until some stopping criterion has been satisfied (generally, an attempt limit or a score threshold).

Tabu search enhances the performance of local search by relaxing its basic rule. First, at each step *worsening* moves can be accepted if no improving move is available (like when the search is stuck at a strict local minimum). In addition, *prohibitions* are introduced to discourage the search from coming back to previously-visited solutions.

Shown below are the pseudo code for PDP.

Heuristics Function: -

Given a route 'r'

Repeat:

 For each pair of locations in 'r':

 If (latter location is more urgent WRT time window),
 then:

 Swap the current 2 nodes to get a route r'

 Calculate $\text{Diff} = \text{Cost}(r') - \text{cost}(r)$

 If $\text{Diff} < 0$ (i.e., new route is cheaper):

$r = r'$

Until: no better routes available (Tabu Search)

Cost Function: -

#Input weights w_1 , w_2 , w_3 as fractions/decimals. Usually w_2 is largest.

$Cost(r) = w_1 * D(r) + w_2 * TWV(r) + w_3 * CV(r)$

$w_1 + w_2 + w_3 = 1$

$D(r)$ = route duration

$TWV(r)$ = number of time violations

$CV(r)$ = number of capacity violations

Sequential Construction: -

Sort all requests wrt distance between depot, pick up point in descending order.

M = number of vehicles used. (initialize with 0)

Repeat (for all requests)

 Initialize an empty route ' r '

$M += 1$

 For all unassigned requests:

 Get next unassigned request ' i '

 Insert ' i ' at the end of ' r '

 Improve ' r ' #use above heuristic

 If r is feasible:

 Insert ' i '

 Else:

 Remove ' i '

Order Keys (ID): -

Oid = order id (set 0=depot)

Pid = Pick up node Id

Did = Deliver node Id

$Dist$ = Distance b/w pick up, depot

Rid =Route Id
x, y = locations
nid = node Id
demand = request size/quantity (+ (pick up), - (delivery))
twopen, twclose = Time window limits

Route: -

Id = route id
Path [] = ordered list of all nodes
Order [] = list of orders corresponding to the nodes
Update = flags for D, TWV, CV
appendorder(order)
heuristic function ()
update ()
cost function ()

Tabu Search: -

‘Best of neighbourhood’ can be used.

[Travelling Salesman Problem Using OR Tools](#)

For a sample data, the code for TSP has been attached below along with screenshots of the output solution.



Steps Involved: -

Create the data set -> Label the nodes and mark the depot as 0th node -> Create the routing model and distance call back -> Set cost of travel and cost parameter -> Add the solution printer.

Routing Model: -

```
data = create_data_model()
manager = pywrapcp.RoutingIndexManager(len(data['distance_matrix']),
                                       data['num_vehicles'], data['depot'])
routing = pywrapcp.RoutingModel(manager)
```

Distance Call back: -

```
def distance_callback(from_index, to_index):
    """Returns the distance between the two nodes."""
    # Convert from routing variable Index to distance matrix NodeIndex.
    from_node = manager.IndexToNode(from_index)
    to_node = manager.IndexToNode(to_index)
    return data['distance_matrix'][from_node][to_node]

transit_callback_index = routing.RegisterTransitCallback(distance_callback)
```

Cost Function: -

```
routing.SetArcCostEvaluatorOfAllVehicles(transit_callback_index)
```

Solution Printer: -

```
def print_solution(manager, routing, solution):
    """Prints solution on console."""
    print('Objective: {} miles'.format(solution.ObjectiveValue()))
    index = routing.Start(0)
    plan_output = 'Route for vehicle 0:\n'
    route_distance = 0
    while not routing.IsEnd(index):
        plan_output += ' {} ->'.format(manager.IndexToNode(index))
        previous_index = index
        index = solution.Value(routing.NextVar(index))
        route_distance += routing.GetArcCostForVehicle(previous_index, index, 0)
    plan_output += ' {}\n'.format(manager.IndexToNode(index))
    print(plan_output)
    plan_output += 'Route distance: {}miles\n'.format(route_distance)
```

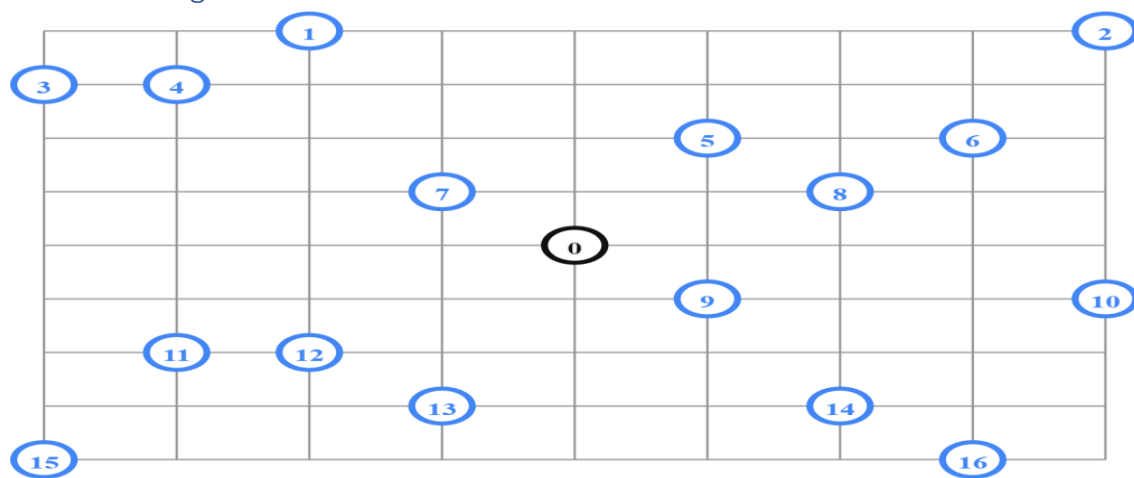
Search Parameters: -

```
search_parameters = pywrapcp.DefaultRoutingSearchParameters()
search_parameters.first_solution_strategy = (
    routing_enums_pb2.FirstSolutionStrategy.PATH_CHEAPEST_ARC)
```

Final Output: -

```
Objective: 7293 miles
Route for vehicle 0:
0 -> 7 -> 2 -> 3 -> 4 -> 12 -> 6 -> 8 -> 1 -> 11 -> 10 -> 5 -> 9 -> 0
```

Vehicle Routing Problem Code



Steps Involved: -

Here, we the sample data calls for a fleet of 4 vehicles.

Create the data set -> Label the nodes and mark the depot as 0th node -> Create the distance call back and a distance dimension -> Set cost of travel and cost parameter -> Add the solution printer -> Main function.

We use the dynamic programming approach. So, we compute the cumulative distance travelled by each vehicle along its route. This is the purpose of the distance dimension function.

```
def distance_callback(from_index, to_index):
    """Returns the distance between the two nodes."""
    # Convert from routing variable Index to distance matrix NodeIndex.
    from_node = manager.IndexToNode(from_index)
    to_node = manager.IndexToNode(to_index)
    return data['distance_matrix'][from_node][to_node]

transit_callback_index = routing.RegisterTransitCallback(distance_callback)
routing.SetArcCostEvaluatorOfAllVehicles(transit_callback_index)
```

```

dimension_name = 'Distance'
routing.AddDimension(
    transit_callback_index,
    0, # no slack
    3000, # vehicle maximum travel distance
    True, # start cumul to zero
    dimension_name)
distance_dimension = routing.GetDimensionOrDie(dimension_name)
distance_dimension.SetGlobalSpanCostCoefficient(100)

```

The other functions are similar to the TSP code and are therefore not shown here.

Output: -

```

Route for vehicle 0:
0 -> 8 -> 6 -> 2 -> 5 -> 0
Distance of route: 1552m

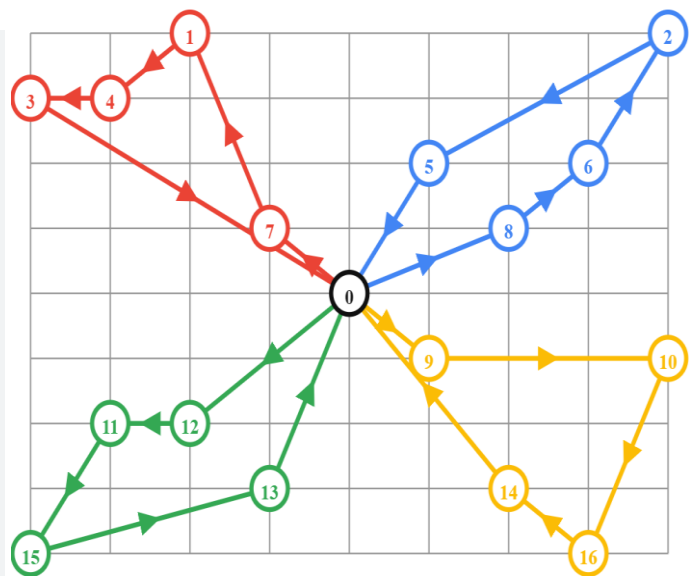
Route for vehicle 1:
0 -> 7 -> 1 -> 4 -> 3 -> 0
Distance of route: 1552m

Route for vehicle 2:
0 -> 9 -> 10 -> 16 -> 14 -> 0
Distance of route: 1552m

Route for vehicle 3:
0 -> 12 -> 11 -> 15 -> 13 -> 0
Distance of route: 1552m

Total distance of all routes: 6208m

```

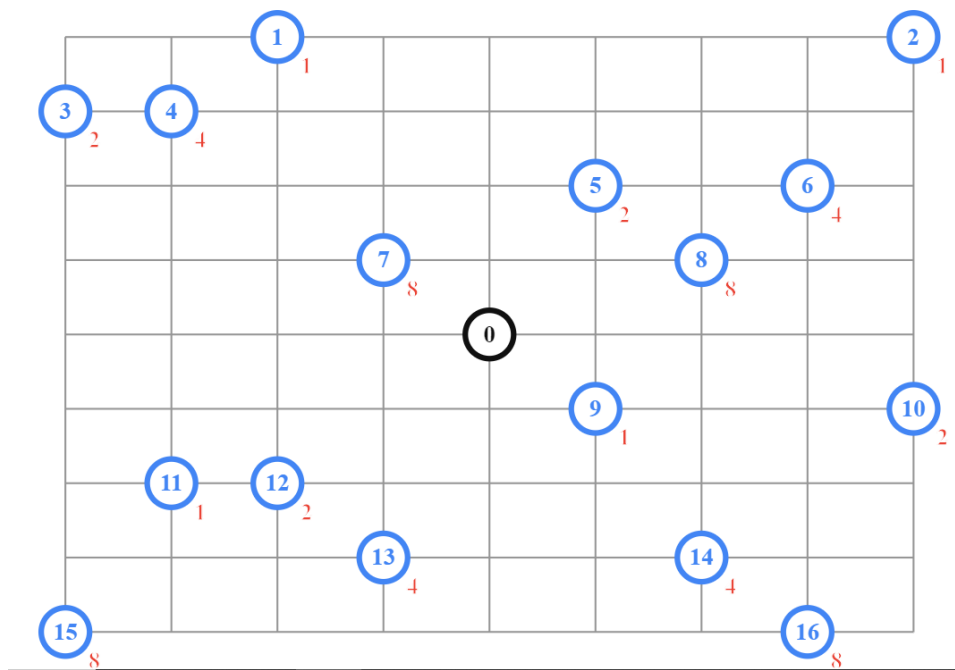


Capacity Constraints

For the sample problem, node 0 is taken as the depot and number of vehicles = 4. An assumption that a homogeneous fleet of vehicles is used is made. If the capacities of each vehicle is different, `AddDimensionWithVehicleCapacity` can be used.

Steps Involved: -

Create the data set -> Label the nodes and mark the depot as 0th node -> Create the distance call back and a **demand call back function** -> Set cost of travel and cost parameter -> Add the solution printer -> Main function.



```
data['demands'] = [0, 1, 1, 2, 4, 2, 4, 8, 8, 1, 2, 1, 2, 4, 4, 8, 8]
data['vehicle_capacities'] = [15, 15, 15, 15]
```

Capacity Call-back function

```
def demand_callback(from_index):
    """Returns the demand of the node."""
    # Convert from routing variable Index to demands NodeIndex.
    from_node = manager.IndexToNode(from_index)
    return data['demands'][from_node]

demand_callback_index = routing.RegisterUnaryTransitCallback(
    demand_callback)
routing.AddDimensionWithVehicleCapacity(
    demand_callback_index,
    0, # null capacity slack
    data['vehicle_capacities'], # vehicle maximum capacities
    True, # start cumul to zero
    'Capacity')
```

Output

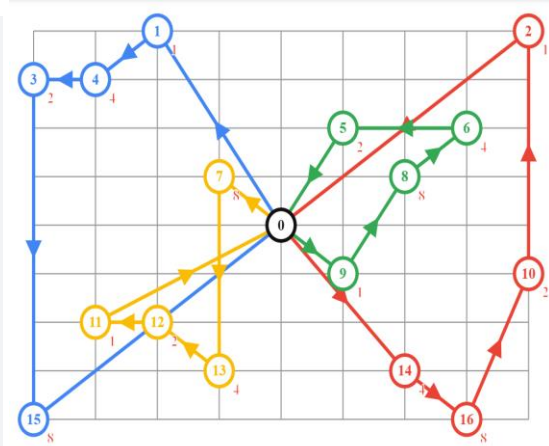
```
Route for vehicle 0:
0 Load(0) -> 1 Load(1) -> 4 Load(5) -> 3 Load(7) -> 15 Load(15) -> 0 Load(15)
Distance of the route: 2192m
Load of the route: 15

Route for vehicle 1:
0 Load(0) -> 14 Load(4) -> 16 Load(12) -> 10 Load(14) -> 2 Load(15) -> 0 Load(15)
Distance of the route: 2192m
Load of the route: 15

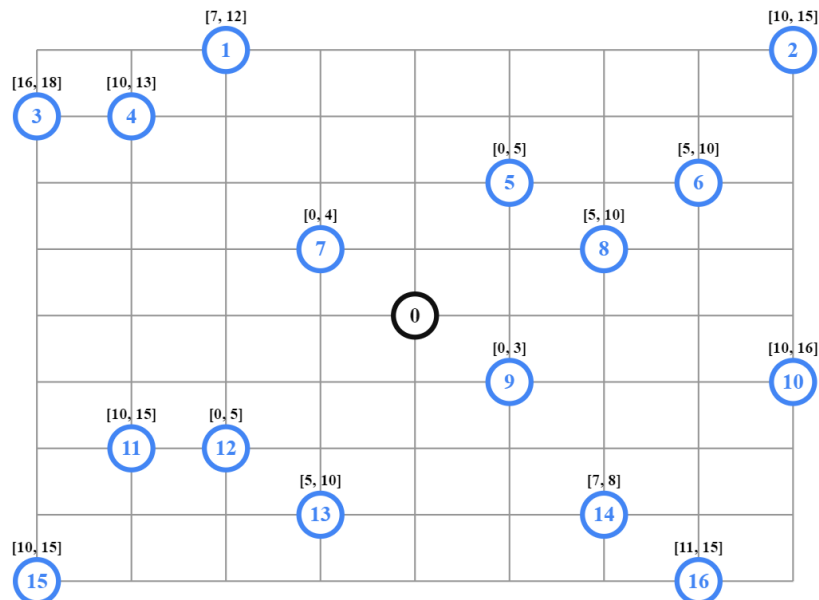
Route for vehicle 2:
0 Load(0) -> 7 Load(8) -> 13 Load(12) -> 12 Load(14) -> 11 Load(15) -> 0 Load(15)
Distance of the route: 1324m
Load of the route: 15

Route for vehicle 3:
0 Load(0) -> 9 Load(1) -> 8 Load(9) -> 6 Load(13) -> 5 Load(15) -> 0 Load(15)
Distance of the route: 1164m
Load of the route: 15

Total Distance of all routes: 6872m
```



Time Windows



Steps Involved: -

Create the data set -> Label the nodes and mark the depot as 0th node -> Create the distance call back and a **time call back function** -> Add time window constraints -> Set cost of travel and cost parameter -> Add the solution printer -> Main function.

Data Set

```
def create_data_model():
    """Stores the data for the problem."""
    data = {}
    data['time_matrix'] = [
        [0, 6, 9, 8, 7, 3, 6, 2, 3, 2, 6, 6, 4, 4, 5, 9, 7],
        [6, 0, 8, 3, 2, 6, 8, 4, 8, 8, 13, 7, 5, 8, 12, 10, 14],
        [9, 8, 0, 11, 10, 6, 3, 9, 5, 8, 4, 15, 14, 13, 9, 18, 9],
        [8, 3, 11, 0, 1, 7, 10, 6, 10, 10, 14, 6, 7, 9, 14, 6, 16],
        [7, 2, 10, 1, 0, 6, 9, 4, 8, 9, 13, 4, 6, 8, 12, 8, 14],
        [3, 6, 6, 7, 6, 0, 2, 3, 2, 2, 7, 9, 7, 7, 6, 12, 8],
        [6, 8, 3, 10, 9, 2, 0, 6, 2, 5, 4, 12, 10, 10, 6, 15, 5],
        [2, 4, 9, 6, 4, 3, 6, 0, 4, 4, 8, 5, 4, 3, 7, 8, 10],
        [3, 8, 5, 10, 8, 2, 2, 4, 0, 3, 4, 9, 8, 7, 3, 13, 6],
        [2, 8, 8, 10, 9, 2, 5, 4, 3, 0, 4, 6, 5, 4, 3, 9, 5],
        [6, 13, 4, 14, 13, 7, 4, 8, 4, 4, 0, 10, 9, 8, 4, 13, 4],
        [6, 7, 15, 6, 4, 9, 12, 5, 9, 6, 10, 0, 1, 3, 7, 3, 10],
        [4, 5, 14, 7, 6, 7, 10, 4, 8, 5, 9, 1, 0, 2, 6, 4, 8],
        [4, 8, 13, 9, 8, 7, 10, 3, 7, 4, 8, 3, 2, 0, 4, 5, 6],
        [5, 12, 9, 14, 12, 6, 6, 7, 3, 3, 4, 7, 6, 4, 0, 9, 2],
        [9, 10, 18, 6, 8, 12, 15, 8, 13, 9, 13, 3, 4, 5, 9, 0, 9],
        [7, 14, 9, 16, 14, 8, 5, 10, 6, 5, 4, 10, 8, 6, 2, 9, 0],
    ]

    data['time_windows'] = [
        (0, 5), # depot
        (7, 12), # 1
        (10, 15), # 2
        (16, 18), # 3
        (10, 13), # 4
        (0, 5), # 5
        (5, 10), # 6
        (0, 4), # 7
        (5, 10), # 8
        (0, 3), # 9
        (10, 16), # 10
        (10, 15), # 11
        (0, 5), # 12
        (5, 10), # 13
        (7, 8), # 14
        (10, 15), # 15
        (11, 15), # 16
    ]

    data['num_vehicles'] = 4
    data['depot'] = 0
    return data
```

Time Call back function

```
def time_callback(from_index, to_index):
    """Returns the travel time between the two nodes."""
    # Convert from routing variable Index to time matrix NodeIndex.
    from_node = manager.IndexToNode(from_index)
    to_node = manager.IndexToNode(to_index)
    return data['time_matrix'][from_node][to_node]

transit_callback_index = routing.RegisterTransitCallback(time_callback)
routing.SetArcCostEvaluatorOfAllVehicles(transit_callback_index)
```

Time Window Constraints

```
time = 'Time'
routing.AddDimension(
    transit_callback_index,
    30, # allow waiting time
    30, # maximum time per vehicle
    False, # Don't force start cumul to zero.
    time)
time_dimension = routing.GetDimensionOrDie(time)
# Add time window constraints for each location except depot.
for location_idx, time_window in enumerate(data['time_windows']):
    if location_idx == 0:
        continue
    index = manager.NodeToIndex(location_idx)
    time_dimension.CumulVar(index).SetRange(time_window[0], time_window[1])
# Add time window constraints for each vehicle start node.
for vehicle_id in range(data['num_vehicles']):
    index = routing.Start(vehicle_id)
    time_dimension.CumulVar(index).SetRange(data['time_windows'][0][0],
                                             data['time_windows'][0][1])

for i in range(data['num_vehicles']):
    routing.AddVariableMinimizedByFinalizer(
        time_dimension.CumulVar(routing.Start(i)))
    routing.AddVariableMinimizedByFinalizer(
        time_dimension.CumulVar(routing.End(i)))
```


Solution Printer and Output

```
def print_solution(data, manager, routing, solution):
    """Prints solution on console."""
    time_dimension = routing.GetDimensionOrDie('Time')
    total_time = 0
    for vehicle_id in range(data['num_vehicles']):
        index = routing.Start(vehicle_id)
        plan_output = 'Route for vehicle {}: \n'.format(vehicle_id)
        while not routing.IsEnd(index):
            time_var = time_dimension.CumulVar(index)
            plan_output += '{0} Time({1},{2}) -> '.format(
                manager.IndexToNode(index), solution.Min(time_var),
                solution.Max(time_var))
            index = solution.Value(routing.NextVar(index))
            time_var = time_dimension.CumulVar(index)
            plan_output += '{0} Time({1},{2}) \n'.format(manager.IndexToNode(index),
                                                        solution.Min(time_var),
                                                        solution.Max(time_var))

            plan_output += 'Time of the route: {}min \n'.format(
                solution.Min(time_var))
        print(plan_output)
        total_time += solution.Min(time_var)
    print('Total time of all routes: {}min'.format(total_time))
```

```
Route for vehicle 0:
 0 Time(0,0) -> 9 Time(2,3) -> 14 Time(7,8) -> 16 Time(11,11) -> 0 Time(18,18)
Time of the route: 18min

Route for vehicle 1:
 0 Time(0,0) -> 7 Time(2,4) -> 1 Time(7,11) -> 4 Time(10,13) -> 3 Time(16,16) -> 0 Time(24,24)
Time of the route: 24min

Route for vehicle 2:
 0 Time(0,0) -> 12 Time(4,4) -> 13 Time(6,6) -> 15 Time(11,11) -> 11 Time(14,14) -> 0 Time(20,20)
Time of the route: 20min

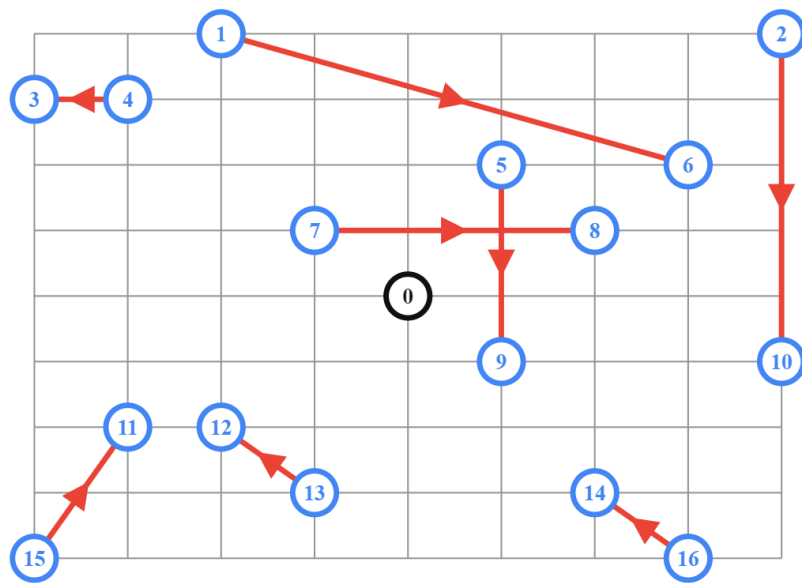
Route for vehicle 3:
 0 Time(0,0) -> 5 Time(3,3) -> 8 Time(5,5) -> 6 Time(7,7) -> 2 Time(10,10) -> 10 Time(14,14)
 0 Time(20,20)
Time of the route: 20min

Total time of all routes: 82min
```

Pickup Delivery Problem

Steps Involved: -

Create the data set -> Label the nodes and mark the depot as 0th node -> Define the pickup delivery requests -> Add the solution printer -> Main function.



Pickup and delivery requests

```
data['pickups_deliveries'] = [
    [1, 6],
    [2, 10],
    [4, 3],
    [5, 9],
    [7, 8],
    [15, 11],
    [13, 12],
    [16, 14],
]
```

Pickup delivery constraints

```
for request in data['pickups_deliveries']:
    pickup_index = manager.NodeToIndex(request[0])
    delivery_index = manager.NodeToIndex(request[1])
    routing.AddPickupAndDelivery(pickup_index, delivery_index)
    routing.solver().Add(
        routing.VehicleVar(pickup_index) == routing.VehicleVar(
            delivery_index))
    routing.solver().Add(
        distance_dimension.CumulVar(pickup_index) <=
        distance_dimension.CumulVar(delivery_index))
```

```
routing.AddPickupAndDelivery(pickup_index, delivery_index)
```

Each item must be picked up and delivered by the same vehicle.

```
routing.solver().Add(  
    routing.VehicleVar(pickup_index) ==  
    routing.VehicleVar(delivery_index))
```

Each item must be picked up before the delivery.

```
routing.solver().Add(  
    distance_dimension.CumulVar(pickup_index) <=  
    distance_dimension.CumulVar(delivery_index))
```

NOTE:

- This is the basic code of PDP. Apart from this, we could add the time window function, capacity constraint function etc from above as per the requirements.
- It is not necessary that service should be given to a pickup – delivery pair before the next destination is taken. It is sufficient if pick up is done before delivery for all pairs of nodes. This can clearly be seen in the output got.

Solution/Output

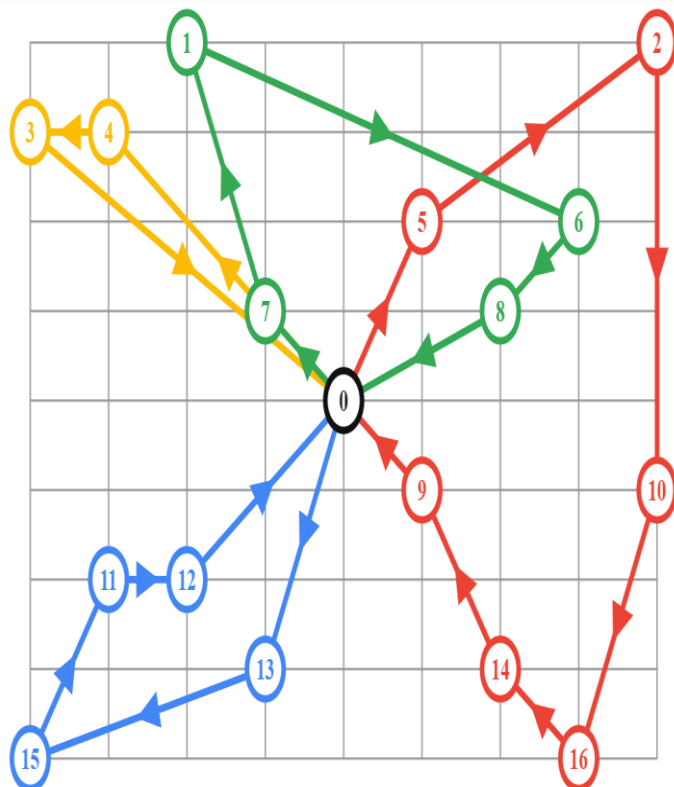
Route for vehicle 0:
0 -> 13 -> 15 -> 11 -> 12 -> 0
Distance of the route: 1552m

Route for vehicle 1:
0 -> 5 -> 2 -> 10 -> 16 -> 14 -> 9 -> 0
Distance of the route: 2192m

Route for vehicle 2:
0 -> 4 -> 3 -> 0
Distance of the route: 1392m

Route for vehicle 3:
0 -> 7 -> 1 -> 6 -> 8 -> 0
Distance of the route: 1780m

Total Distance of all routes: 6916m



Transportation Problem (Without OR Tools).

The problem objective is to minimize the total shipping cost to all customers from all sources.

$$\text{minimize: Cost} = \sum_{c \in \text{Customers}} \sum_{s \in \text{Sources}} T[c, s] x[c, s]$$

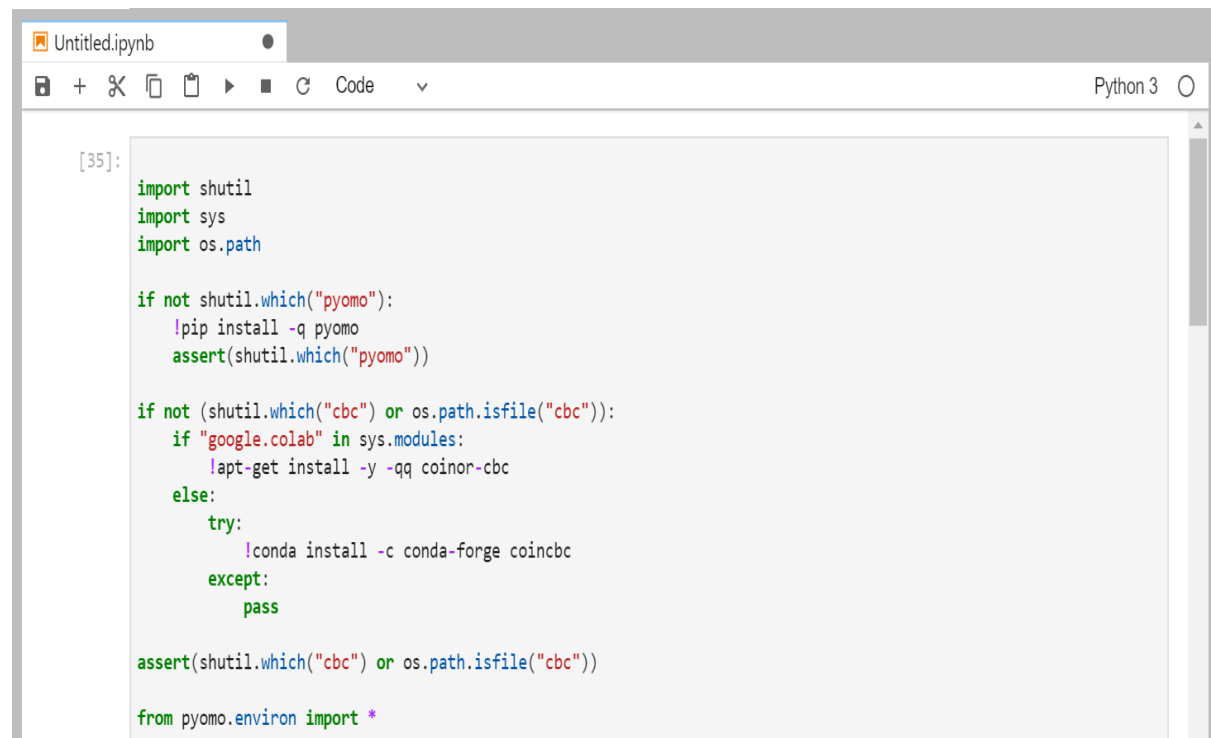
Shipments from all sources cannot exceed the manufacturing capacity of the source.

$$\sum_{c \in \text{Customers}} x[c, s] \leq \text{Supply}[s] \forall s \in \text{Sources}$$

Shipments to each customer must satisfy their demand.

$$\sum_{s \in \text{Sources}} x[c, s] = \text{Demand}[c] \forall c \in \text{Customers}$$

Pyomo Imports and Sample Data

A screenshot of a Jupyter Notebook interface. The top bar shows 'Untitled.ipynb' and 'Python 3'. The code cell contains the following Python code:

```
[35]:
import shutil
import sys
import os.path

if not shutil.which("pyomo"):
    !pip install -q pyomo
    assert(shutil.which("pyomo"))

if not (shutil.which("cbc") or os.path.isfile("cbc")):
    if "google.colab" in sys.modules:
        !apt-get install -y -qq coinor-cbc
    else:
        try:
            !conda install -c conda-forge coinbc
        except:
            pass

assert(shutil.which("cbc") or os.path.isfile("cbc"))

from pyomo.environ import *
```

Concrete Model

```
[33]: Demand = {
    'Lon': 125,      # London
    'Ber': 175,      # Berlin
    'Maa': 225,      # Maastricht
    'Ams': 250,      # Amsterdam
    'Utr': 225,      # Utrecht
    'Hag': 200       # The Hague
}

Supply = {
    'Arn': 600,      # Arnhem
    'Gou': 650       # Gouda
}

T = {
    ('Lon', 'Arn'): 1000,
    ('Lon', 'Gou'): 2.5,
    ('Ber', 'Arn'): 2.5,
    ('Ber', 'Gou'): 1000,
    ('Maa', 'Arn'): 1.6,
    ('Maa', 'Gou'): 2.0,
    ('Ams', 'Arn'): 1.4,
    ('Ams', 'Gou'): 1.0,
    ('Utr', 'Arn'): 0.8,
    ('Utr', 'Gou'): 1.0,
    ('Hag', 'Arn'): 1.4,
    ('Hag', 'Gou'): 0.8
}
```

```
[36]: model = ConcreteModel()
model.dual = Suffix(direction=Suffix.IMPORT)

# Step 1: Define index sets
CUS = list(Demand.keys())
SRC = list(Supply.keys())

# Step 2: Define the decision
model.x = Var(CUS, SRC, domain = NonNegativeReals)

# Step 3: Define Objective
model.Cost = Objective(
    expr = sum([T[c,s]*model.x[c,s] for c in CUS for s in SRC]),
    sense = minimize)

# Step 4: Constraints
model.src = ConstraintList()
for s in SRC:
    model.src.add(sum([model.x[c,s] for c in CUS]) <= Supply[s])

model.dmd = ConstraintList()
for c in CUS:
    model.dmd.add(sum([model.x[c,s] for s in SRC]) == Demand[c])

results = SolverFactory('cbc').solve(model)
results.write()
```

Solution Printing

```
[37]: for c in CUS:
    for s in SRC:
        print(c, s, model.x[c,s]())
```

```
Lon Arn None
Lon Gou None
Ber Arn None
Ber Gou None
Maa Arn None
Maa Gou None
Ams Arn None
Ams Gou None
Utr Arn None
Utr Gou None
Hag Arn None
Hag Gou None
```

```
[38]: if 'ok' == str(results.Solver.status):
    print("Total Shipping Costs = ", model.Cost())
    print("\nShipping Table:")
    for s in SRC:
        for c in CUS:
            if model.x[c,s]() > 0:
                print("Ship from ", s, " to ", c, ":", model.x[c,s]())
    else:
        print("No Valid Solution Found")
```

Total Shipping Costs = 1705.0

Shipping Table:

```
Ship from Arn to Ber : 175.0
Ship from Arn to Maa : 225.0
Ship from Arn to Utr : 200.0
Ship from Gou to Lon : 125.0
Ship from Gou to Ams : 250.0
Ship from Gou to Utr : 25.0
Ship from Gou to Hag : 200.0
```

INVENTORY CONSTRAINTS

The inventory component arises because customers consume product over time and have only limited storage capacity. The supplier has to manage product inventory at customers to ensure that customers do not experience a stock-out. The inventory component thus adds a time dimension to the traditional spatial dimension of routing problems. **Inventory holding costs** are to be included as well.

Assumption: - We consider only inventory routing problems involving the distribution of a single product over a finite planning horizon with deterministic and stationary production and consumption rates.

Labelling: -

Consider $G = (V, E)$, where $V = \{S, 1, \dots, n\}$ is the set of vertices and E is the set of edges. Vertex S represents the supplier and vertices $1, \dots, n$ represents the customers.

travel time = t_{ij} and cost = c_{ij} between vertices i, j .

Q = capacity of vehicle.

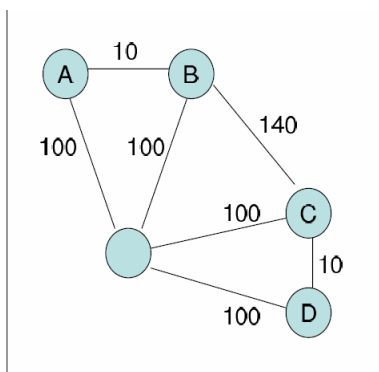
If time is discrete, we denote by q_i the quantity of product consumed per unit of time by customer i . If time is continuous, we denote by u_i the usage or consumption rate.

I_s^o = initial inventory at service station, I_i^o = initial inventory with customer.

I_s^t = initial inventory at service station, I_i^t = initial inventory with customer.

The inventory holding cost at the supplier is h_s and at customer i is h_i .

The length of the planning horizon is denoted by H . **If $H=1$, $Q=\text{infinity}$ and inventory costs are 0, it reduces to TSP.** The decisions to be made are (1) when to deliver to each customer, (2) how much to deliver to each customer, and (3) how to route the vehicles so as to minimize the total cost. A delivery policy has to ensure that the supplier and the customers do not experience any stock-outs, that storage capacities at the supplier and the customers are not exceeded, and that the vehicle capacity limit is respected.



The initial inventory level of each customer i is equal to its holding capacity, that is I_i^o for all customers. There is an unlimited number of vehicles with capacity $Q = 5000$. The objective is to find a periodic distribution policy that minimizes the total cost, that does not cause a stock out at any of the customers, and that does not exceed the storage capacity of the customers and the vehicle capacity.

We first analyse the case in which only transportation cost is considered in the objective function. The optimal solution is obtained by solving the following mixed integer linear programming model:

$$\min \frac{1}{H} \sum_{t \in T} \sum_{k \in K} c_k y_k^t \quad (1)$$

$$\sum_{i \in I} x_{ik}^t \leq Q y_k^t \quad t \in T \quad k \in K \quad (2)$$

$$x_{ik}^t \leq Q r_{ik} \quad t \in T \quad i \in I \quad k \in K \quad (3)$$

$$I_i^t = I_i^0 + \sum_{s=1}^t \sum_{k \in K} x_{ik}^s - t q_i \quad i \in I \quad t \in T \quad (4)$$

$$I_i^t + q_i \leq C_i \quad i \in I \quad t \in T \quad (5)$$

$$I_i^t \geq 0 \quad i \in I \quad t \in T \quad (6)$$

$$x_{ik}^t \geq 0 \quad t \in T \quad i \in I \quad k \in K \quad (7)$$

$$y_k^t \in \{0, 1\} \quad t \in T \quad k \in K. \quad (8)$$

The objective function (1) expresses the minimization of the average daily transportation cost. The constraints (2) guarantee that the total quantity delivered by each vehicle is not greater than its capacity. The constraints (3) guarantee that a delivery to customer i on route k only takes place if customer i is visited on route k . The constraints (4) define the level of the inventory of each customer i for each time instant t . The constraints (5) guarantees that the storage capacity of each customer is never exceeded. The constraints (6) guarantee that no stock-out occurs at any customer i during the planning horizon. Finally, the constraints (7), (8) define the decision variables of the problem.

When inventory holding costs at the customers' side are taken into account in the objective function, the total cost of an optimal solution will obviously increase. An optimal solution can be obtained by solving the following mixed integer programming problem subject to (2), (8).

$$\min \frac{1}{H} \left(\sum_{t \in T} \sum_{k \in K} c_k y_k^t + \sum_{i \in I} \sum_{t \in T} h_i I_i^t \right)$$

To take inventory holding costs at the supplier into account, a term $\sum_{\forall t} h_S I_S^t$ needs to be incorporated in the objective function. Our new objective function becomes:

$$\min \frac{1}{H} \sum_{t \in T} \sum_{k \in K} c_k y_k^t + \sum_{t \in T} h_S I_S^t.$$

This time, new constraints are also to be included. (9) states that the inventory level at the supplier at time t is obtained by the initial level increased by the production up to t and decreased by the quantity delivered to the customers up to t .

$$I_S^t = I_S^0 + \sum_{i \in I} t q_i - \sum_{s=1}^t \sum_{i \in I} \sum_{k \in K} x_{ik}^s \quad t \in T \quad (9)$$

$$I_S^t \geq 0 \quad t \in T. \quad (10)$$

Finally, we consider the case in which inventory holding costs are charged both at the supplier and at the customer's ends and the initial inventory levels are not fixed, but are to be determined by the optimization. It is now necessary to include the *demand constraints* in the model, that is constraints that guarantee that the total quantity shipped to each customer is equal to the corresponding total consumption over the planning horizon:

$$\min \frac{1}{H} \left(\sum_{t \in T} \sum_{k \in K} c_k y_k^t + \sum_{i \in I} \sum_{t \in T} h_i I_i^t + \sum_{t \in T} h_S I_S^t \right)$$

subject to (2) to (8), (9) to (10) and the following constraints:

$$\sum_{t \in T} \sum_{k \in K} x_{ik}^t = q_i H \quad i \in I \quad (11)$$

$$I_i^0 \geq 0 \quad i \in I \quad (12)$$

$$I_S^0 \geq 0. \quad (13)$$

SOFT SKILLS LEARNT AND CHANGES IN PERSONALITY

- Learnt from sources apart from tutorials – learnt to read research papers.
Is what you are reading relevant?
Do you know enough to read the paper?
How much to read?
- Learnt a new language (Python) and about various libraries.
Learnt from sources apart from tutorials – such as documentations, GitHub, research papers for the first time.
- Communication skills improved.
Precise and not beating around the bush.
How to interact in panel reviews and group discussions (Importance of minutes of the meet).
Being precise with no inhibitions.
- Working in group environment – first major exposure.
- Drastic change in my approach towards coding and computer science.
- Importance of prior preparation for even small tasks.
- How to make a plan: -
Divide the project into stages -> Have weekly goals -> Set daily targets ->
Have a buffer day in case there is any delay.

SUMMARY AND CONCLUSION

The project given was 'Network optimization (Vehicle Routing Problem) using Python'. The goal was to develop a Python code that would give us the optimum travel route given a set of destinations such that the transportation cost is minimum. Minimizing the transportation cost would mean optimal scheduling and would therefore increase the revenue. The key areas of focus included TSP (travelling salesman problem), VRP (vehicle routing problem), PDP (Pick up delivery problem), OVRP (Open Vehicle Routing Problem) and the Python modules - OR Tools and Pyomo. The question to be answered was "What is the optimal set of routes for a fleet of vehicles to traverse in order to deliver to a given set of customers?"

The first step was to understand the various optimization techniques along with their time complexities. Basics of Python were learnt and research was done on the libraries Pyomo and OR tools. The differences between TSP, VRP and its variants were understood. A pseudo code for PDP was written and this was followed by writing the code for TSP. More constraints were slowly added – single vehicle to multiple vehicles, time constraints, capacity constraints, pickup and drop constraints etc. Research about inventory constraints was done.

RECOMMENDATIONS - PATH AHEAD (Topics to Explore)

- Multi-trip formulations.
- PDP for line hold design.
- Main to retail store variants – inventory constraints, territorial constraints and maximum waiting time.
- More research on open vehicle routing problem.
- Entire PDP code without using OR Tools.

REFERENCES

Optimization and Operations Research: -

- Operations Research by Hamdy. A. Taha
- The Vehicle Routing Problem by Paolo Toth and Daniele Vigo
- Introduction to Graph Theory by Gary Chartrand and Ping Zhang
- Papers on VRP, PDP from Research Gate
- Operations Research – Computer Science Interfaces By Professor Ramesh Sharda
- <https://www.oreilly.com/library/view/operationsresearch/9788131799345/xhtml/chapter009.xhtml>
- <https://ieeexplore.ieee.org/document/8272945>
- https://en.wikipedia.org/wiki/Travelling_salesman_problem

Python: -

- Basics of Python (Numbers, Data Structures, Functions, OOP) – Udemy (Jose Portilla) and Python for Everybody – University Of Michigan (Coursera)
- OR Tools Documentation
- Frequently used libraries (numpy, pandas, scipy, matplotlib, seaborn) - EdEureka YouTube
- Pyomo Library: -
 - Pyomo documentation
 - Optimization Modelling in Python – Pyomo (Textbook)
 - <https://www.youtube.com/watch?v=cjMkVHjhSBI> (YouTube tutorial).