OR Tools

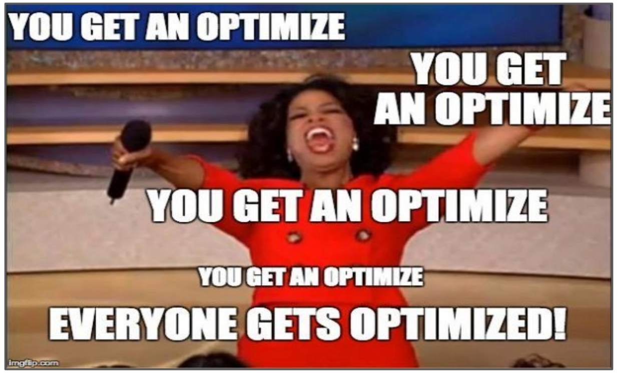
Data Science Engineering Methods and Tools - INFO 6105 (37364)

Project Group **2** - January 26, 2019

Hardik Soni | **Krina** Thakkar | **Yash** Lekhwani | **Yashashri** Shiral

Optimization

- To find best solution to a problem out of large set of possible solution
- We all have finite resources and time & we want to make most of them
  - Scheduling the order at which you will answer the emails
  - Switching to a new route back home to minimize traffic woes
- How Optimization is important to data science
  - One of the main concerns of data scientist is creating model that fits the problem like finding optimal heuristics, minimum function losses
  - Therefore it is crucial to understand optimization frameworks

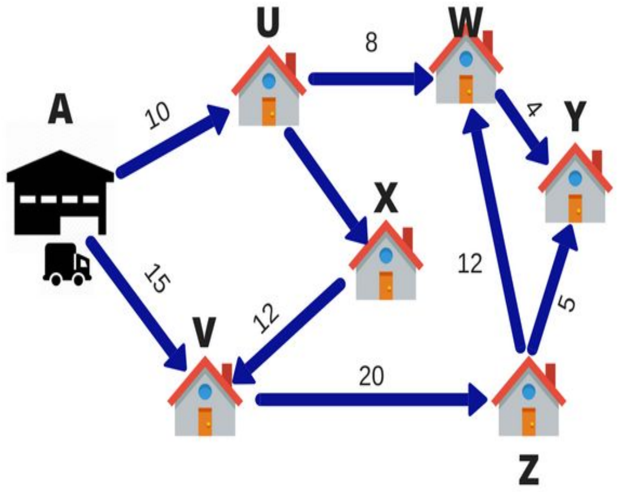


LINEAR OPTIMIZATION

Linear Optimization

- Linear Optimization is method of computing best solution to a problem modelled as a set of linear relationships

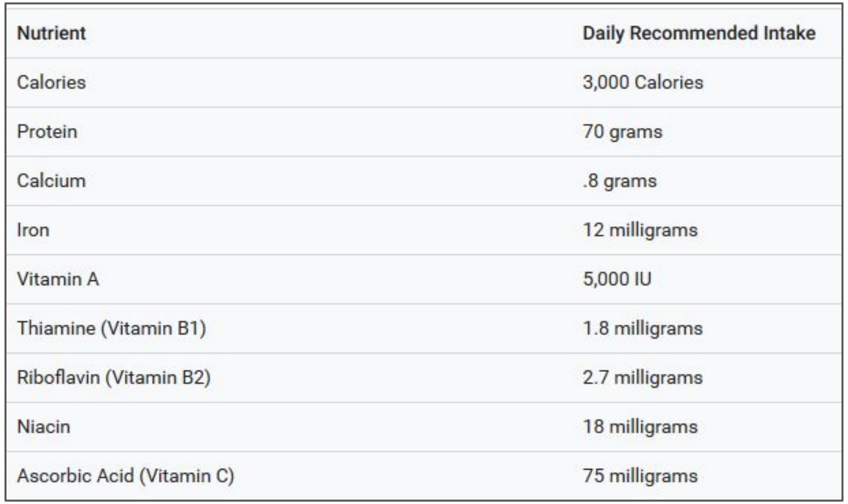
- Problem - FedEx delivery
    - The delivery person will calculate different routes for going to all the 6 destination and then come up with the shortest route.



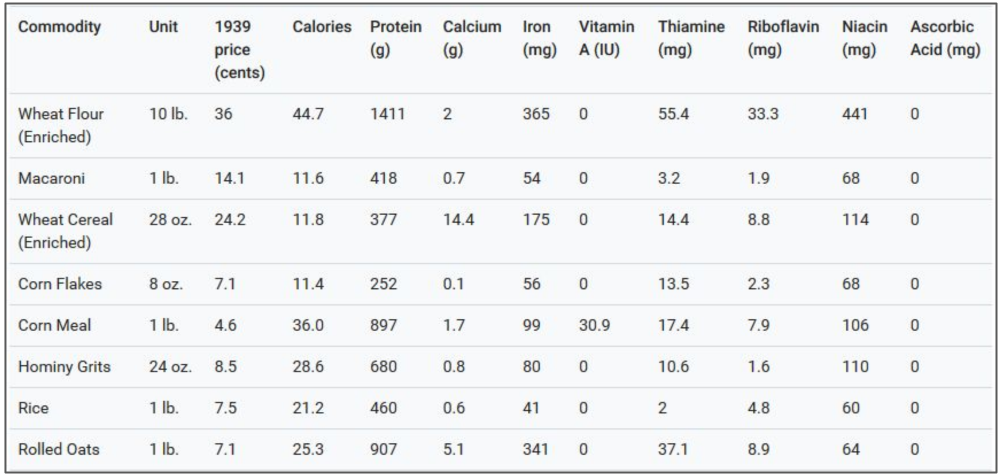
Example - Stigler Diet

Stigler Diet - named for economics Nobel laureate George Sigler who computed an inexpensive way to fulfill basic nutritional needs given set of foods

- Since the nutrients have all been normalized by price, our objective is simply minimizing the sum of foods
- The Stigler diet mandated that these minimums be met:



Nutrient	Daily Recommended Intake
Calories	3,000 Calories
Protein	70 grams
Calcium	.8 grams
Iron	12 milligrams
Vitamin A	5,000 IU
Thiamine (Vitamin B1)	1.8 milligrams
Riboflavin (Vitamin B2)	2.7 milligrams
Niacin	18 milligrams
Ascorbic Acid (Vitamin C)	75 milligrams



Commodity	Unit	1939 price (cents)	Calories	Protein (g)	Calcium (g)	Iron (mg)	Vitamin A (IU)	Thiamine (mg)	Riboflavin (mg)	Niacin (mg)	Ascorbic Acid (mg)
Wheat Flour (Enriched)	10 lb.	36	44.7	1411	2	365	0	55.4	33.3	441	0
Macaroni	1 lb.	14.1	11.6	418	0.7	54	0	3.2	1.9	68	0
Wheat Cereal (Enriched)	28 oz.	24.2	11.8	377	14.4	175	0	14.4	8.8	114	0
Corn Flakes	8 oz.	7.1	11.4	252	0.1	56	0	13.5	2.3	68	0
Corn Meal	1 lb.	4.6	36.0	897	1.7	99	30.9	17.4	7.9	106	0
Hominy Grits	24 oz.	8.5	28.6	680	0.8	80	0	10.6	1.6	110	0
Rice	1 lb.	7.5	21.2	460	0.6	41	0	2	4.8	60	0
Rolled Oats	1 lb.	7.1	25.3	907	5.1	341	0	37.1	8.9	64	0

Stigler Diet - Solution

1. Declare the solver using the Python wrapper *pywraplp*

```python
from ortools.linear_solver import pywraplp

def main():
  # Instantiate a Glop solver, naming it LinearExample.
  solver = pywraplp.Solver('LinearExample',
                           pywraplp.Solver.GLOP_LINEAR_PROGRAMMING)
```

2. Define the data and constraints for the problem
3. Create the variables and define the objective

```python
food = [[]] * len(data)

# Objective: minimize the sum of (price-normalized) foods.
objective = solver.Objective()
for i in range(0, len(data)):
  food[i] = solver.NumVar(0.0, solver.infinity(), data[i][0])
  objective.SetCoefficient(food[i], 1)
objective.SetMinimization()
```

4. Invoke the solver and display the results

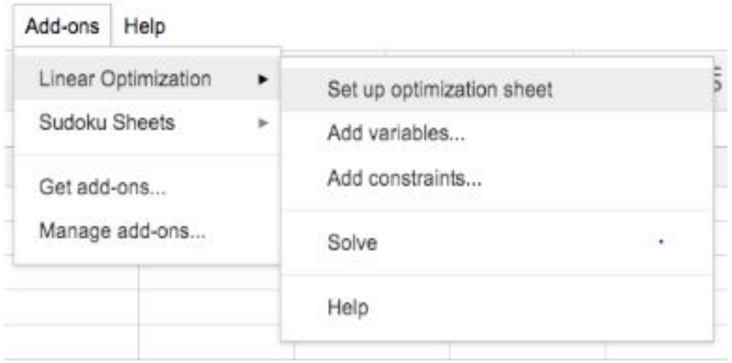
Glop solves the problem on a typical computer in less than 300 milliseconds

```
$ PYTHONPATH=src python stigler.py
Wheat Flour (Enriched) = 0.029519
Liver (Beef) = 0.001893
Cabbage = 0.011214
Spinach = 0.005008
Navy Beans, Dried = 0.061029

Optimal annual price: $39.66
```

Add-On for Google Sheets

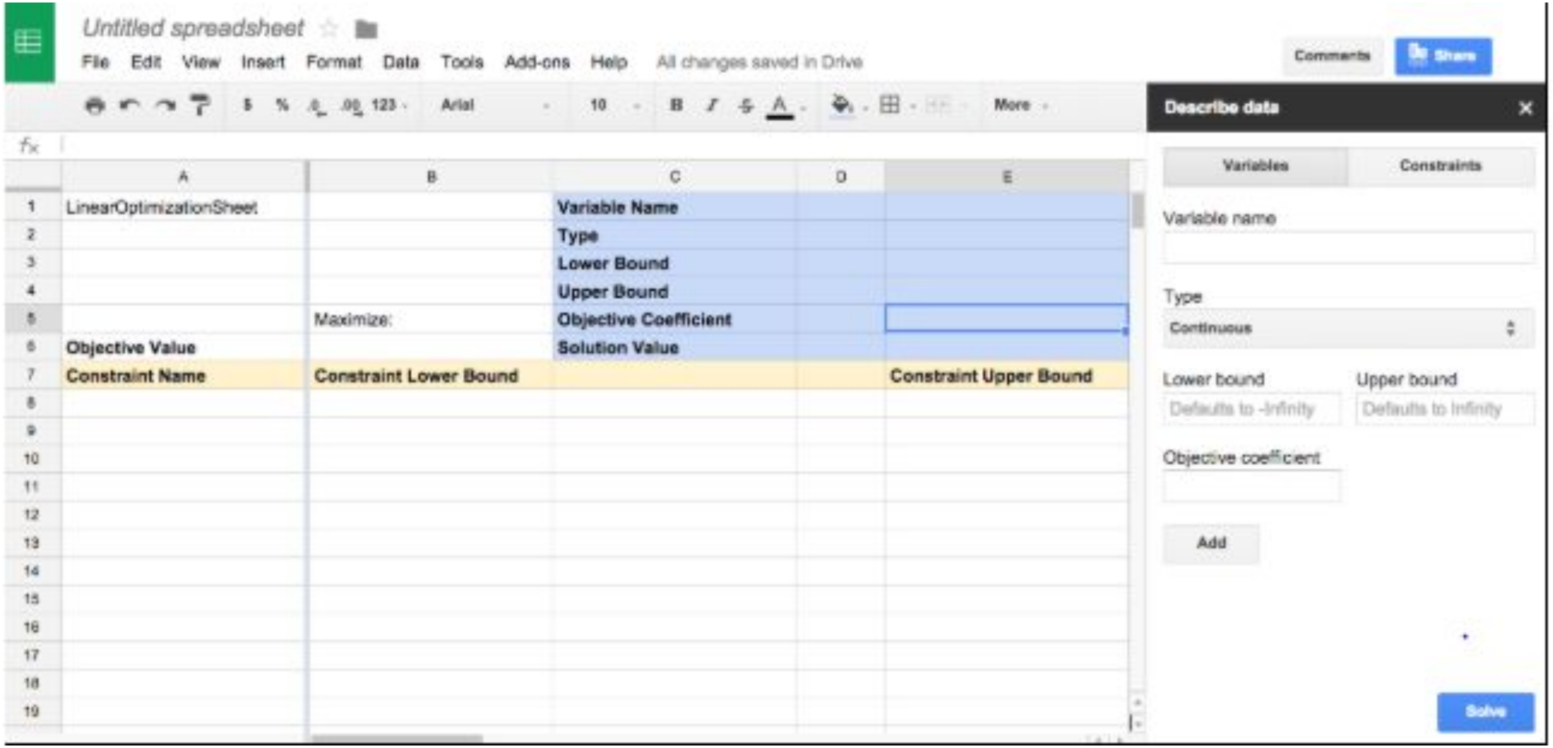
- Consider a chocolate manufacturing company which produces only two types of chocolate – A and B
- Both the chocolates require Milk and Choco only
  - To manufacture each unit of A and B, the following quantities are required
    - Each unit of A requires 1 unit of Milk and 3 units of Choco
    - Each unit of B requires 1 unit of Milk and 2 units of Choco
  - The company kitchen has a total of 5 units of Milk and 12 units of Choco. On each sale, the company makes a profit of
    - Rs 6 per unit A sold
    - Rs 5 per unit B sold

- Profit: Maximize ***Z = 6X + 5Y***
  - $X + Y <= 5$
  - $3X + 2Y <= 12$
  - $X >= 0$ & $Y >= 0$



Add-On for Google Sheets

- Spreadsheet


Untitled spreadsheet

File Edit View Insert Format Data Tools Add-ons Help All changes saved in Drive

	A	B	C	D	E
1	LinearOptimizationSheet		Variable Name		
2			Type		
3			Lower Bound		
4			Upper Bound		
5		Maximize:	Objective Coefficient		
6	Objective Value		Solution Value		
7	Constraint Name	Constraint Lower Bound			Constraint Upper Bound

Describe data

Variables | Constraints

Variable name

Type: Continuous

Lower bound: Defaults to -Infinity

Upper bound: Defaults to Infinity

Objective coefficient

Add

Solve

CONSTRAINT OPTIMIZATION

Constraint Programming

- Based on feasibility rather than optimization
- Focuses on the constraints and variables rather than the objective function

- **Feasible Solution**: A set of values for the decision variables that satisfies all of the constraints in an optimization problem
- Let’s take a simple example of scheduling 4 employees into 3 shifts with an aim to assign 3 of its employees to 8 hours each while giving the fourth a day off:
  - How many possible ways are there for a manager to assign employees per day?
  - How many possible ways are there for a manager to assign employees per week?
- Our goal is to narrow down a very large set of possible solutions to a more manageable subset by adding the following constraints to the problem:
  - Each employee should work some minimum hours per week
  - No employee should work two shifts in a row

Constraint Programming

OR tools provides two solvers for constraint programming:

- CP-SAT solver
- Original CP solver

Note: CP-SAT solver is technologically superior to the original CP solver and should be preferred in almost all situations

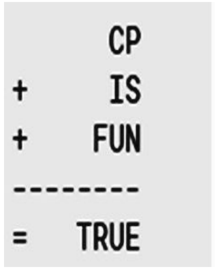
Constraint Programming: CP-SAT

A manager needs to create a schedule for 4 employees over a three-day period, subject to the following conditions:

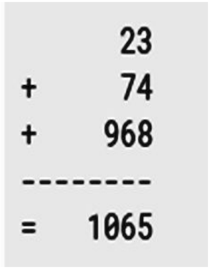
- Each day is divided into three 8-hour shifts
- Each employee is assigned to at least two shifts during the three-day period
- Every day, each shift is assigned to a single employee, and no employee works more than one shift in a day

Constraint Programming - Cryptarithmetic

- A cryptarithmetic puzzle is a mathematical exercise where the digits of some numbers are represented by letters (or symbols)
- The goal is to find the digits such that a given mathematical equation is verified
- Variables: Letters that can take any single digit value
- Constraints:
  - CP + IS + FUN = TRUE
  - Each of the ten letters must be a different digit
  - C, I, F, and T can't be zero (since we don't write leading zeros in numbers)



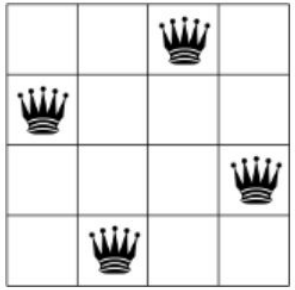
```
      CP
+     IS
+    FUN
--------
=   TRUE
```



```
      23
+     74
+    968
--------
=   1065
```

N Queens Problem

- How can N queens be placed on an N x N chessboard so that no two of them attack each other?
- Constraint: No two queens are on the same row, column, or diagonal



INTEGER OPTIMIZATION

Integer Optimization

- Problems that require some or all variables to be Integers

- For example:
  - Quantity of consumer items to manufacture per month
  - A manufacturing company is considering expansion in Boston, New York or in both the cities with one new warehouse in any one of the cities. This problem involves "yes-or no" decisions for building the warehouse.

- Google's OR tools provides two types of solvers for such problems:
  - Mixed Integer Programming (MIP) solver
  - Constraint Programming (CP) solver

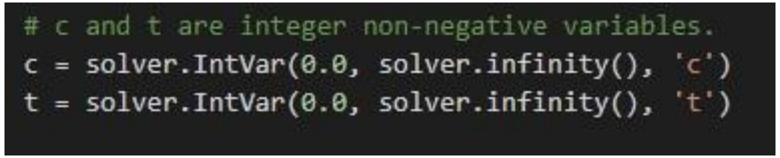
Mixed Integer Programming (MIP) solver

- Can be used when the variables are pure integer or a combination of integer and continuous
- Let's take an example of table and chair manufacturing company, XYZ:
- **Background**: Suppose XYZ considers producing chairs and tables using only ***21 Sq m*** of wood. Each chair requires ***6 Sq m*** and table requires ***7 Sq m*** of wood. Each chair is sold at ***$12*** and each table is sold at ***$13***
- **Problem**: Let ***c*** denote chairs and ***t*** denote table, find out how many table and chairs could be manufactured out of *21 Sq m* wood with maximum revenue.

Mixed Integer Programming (MIP) solver

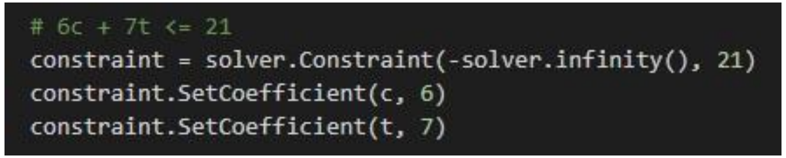
1. Define the problem in IP formulation: ***12c + 13t***
2. Identify the constraints:
   a. ***6c + 7t <= 21***
   b. ***c, t >= 0 and integers***

3. Declare the variables ***c*** and ***t***



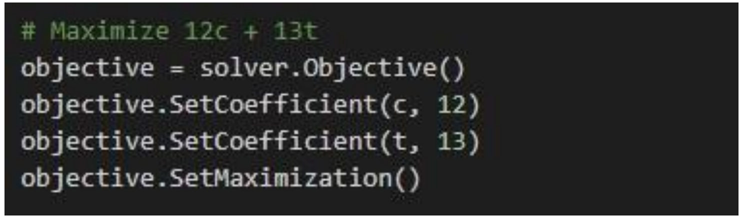
```
# c and t are integer non-negative variables.
c = solver.IntVar(0.0, solver.infinity(), 'c')
t = solver.IntVar(0.0, solver.infinity(), 't')
```

4. Define the constraints



```
# 6c + 7t <= 21
constraint = solver.Constraint(-solver.infinity(), 21)
constraint.SetCoefficient(c, 6)
constraint.SetCoefficient(t, 7)
```

5. Define the objective



```
# Maximize 12c + 13t
objective = solver.Objective()
objective.SetCoefficient(c, 12)
objective.SetCoefficient(t, 13)
objective.SetMaximization()
```

CP Approach to Integer Optimization solver

- Based on feasibility (i.e. finding a feasible solution) rather than optimization (i.e. finding an optimal solution) and focuses on the constraints and variables' domain rather than the objective function
- Constraint programming looks first to reduce the set of possible values of the decision variables

- Integer Programming: For standard integer programming problems, in which an optimal solution must satisfy all constraints
- Constraint Programming: For problems when the optimal solution satisfies one of the constraints

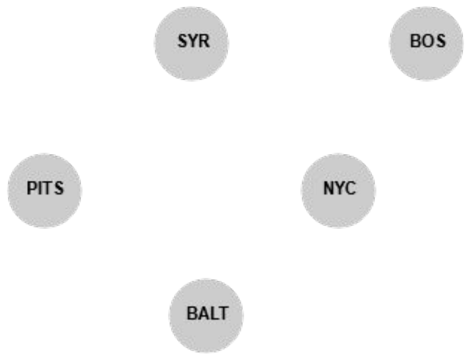
ROUTING

Routing

- Find efficient paths for transporting items through a complex network
- Routing problems can be divided into two main types
  - node-routing problems
  - arc-routing problems
- OR-Tools includes a specialized routing library to solve many types of node-routing problems:
  - Traveling Salesman Problems (TSP)
  - Vehicle Routing Problems (VRP)
  - Capacitated Vehicle Routing Problems (CVRP)
  - Vehicle Routing Problems with Time Windows (VRPTW)
  - Vehicle Routing Problems with Resource Constraints
  - Vehicle Routing Problems with Pickup and Delivery (VRPPD)

Traveling Salesman Problem

- Famous problem in the field of operations research
- Suppose you decide to drive a car around north east of the US
  - you will start in Boston
  - the goal is to visit New York City, Pittsburg, Baltimore, and Syracuse before returning to Boston
- What is the best itinerary?
  - how can you minimize the number of kilometers yet make sure you visit all the cities?
  - If there are only 5 cities it’s not too hard to figure out the optimal tour
- As we add cities to our tour, however, it is much harder to figure out the optimal tour



Traveling Salesman Problem

1. Define cities and the distance matrix

```
def main():
  # Cities
  city_names = ["Boston", "New York City", "Baltimore", "Pittsburg", "Syracuse"]

  # Distance matrix
  dist_matrix = [
      [0, 215, 402, 572, 311], #Boston
      [215, 0, 203, 371, 247], #New York City
      [402, 203, 0, 248, 332], #Baltimore
      [572, 371, 248, 0, 361], #Pittsburg
      [311, 247, 332, 361, 0]] #Syracuse
```

2. Declare the solver

```
tsp_size = len(city_names)
num_routes = 1
depot = 0

# Create routing model
if tsp_size > 0:
  routing = pywrapcp.RoutingModel(tsp_size, num_routes, depot)
  search_parameters = pywrapcp.RoutingModel.DefaultSearchParameters()
  # Create the distance callback.
  dist_callback = create_distance_callback(dist_matrix)
  routing.SetArcCostEvaluatorOfAllVehicles(dist_callback)
```

3. Create the distance callback

```
# Distance callback
def create_distance_callback(dist_matrix):
  # Create a callback to calculate distances between cities.

  def distance_callback(from_node, to_node):
    return int(dist_matrix[from_node][to_node])

  return distance_callback
```

4. Invoke the solver and display the results

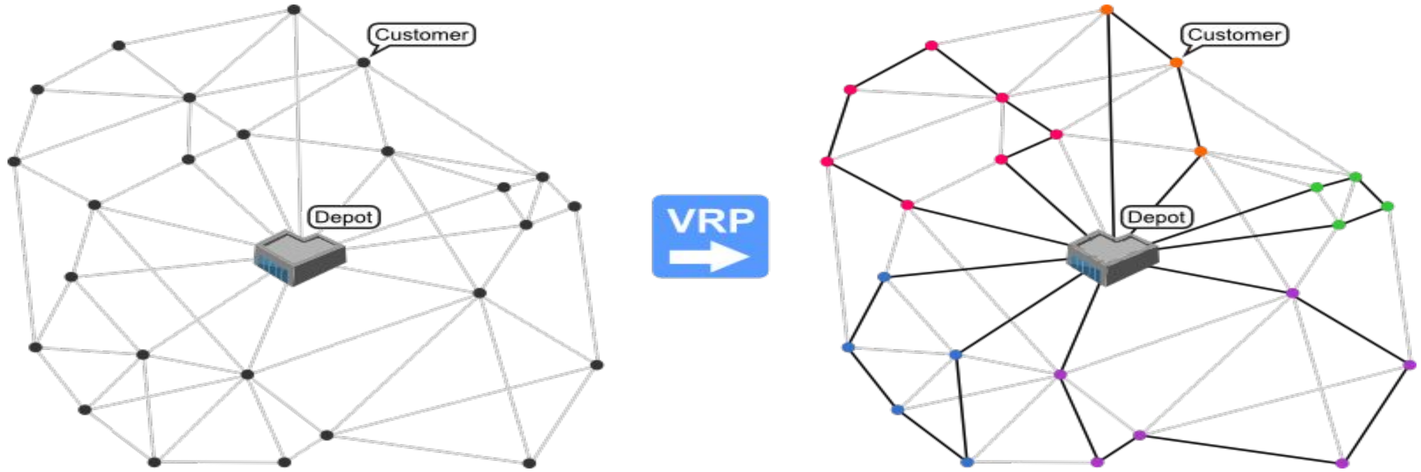
```
  # Solve the problem.
  assignment = routing.SolveWithParameters(search_parameters)
  if assignment:
    # Solution distance.
    print ("Total distance: " + str(assignment.ObjectiveValue()) + " miles\n")
    # Display the solution.
    # Only one route here; otherwise iterate from 0 to routing.vehicles() - 1
    route_number = 0
    index = routing.Start(route_number) # Index of the variable for the starting node.
    route = ''
    while not routing.IsEnd(index):
      # Convert variable indices to node indices in the displayed route.
      route += str(city_names[routing.IndexToNode(index)]) + ' -> '
      index = assignment.Value(routing.NextVar(index))
    route += str(city_names[routing.IndexToNode(index)])
    print ("Route:\n\n" + route)
  else:
    print ('No solution found.')
else:
  print ('Specify an instance greater than 0.')
```

Total distance: *1338 miles*

Route: *Boston -> New York City -> Baltimore -> Pittsburg -> Syracuse -> Boston*

VRP - Example

- The goal is to find the optimal set of routes for a fleet of vehicles delivering goods or services to various locations
- VRP constraints
  - Capacity Constraint: the total *demand* of the locations on a vehicle's route cannot exceed its *capacity*
  - Time Window: each location must be serviced within a time window $[a_i, b_i]$ and waiting times are allowed
- Consider where we have a depot surrounded by a number of customers who are to be supplied from the depot



VRP - Example

A diagram of the city is shown below, with the company location marked in black and the locations to visit in blue

1. Create the problem data

```
data = create_data_model()
```

2. Declare the routing model solver

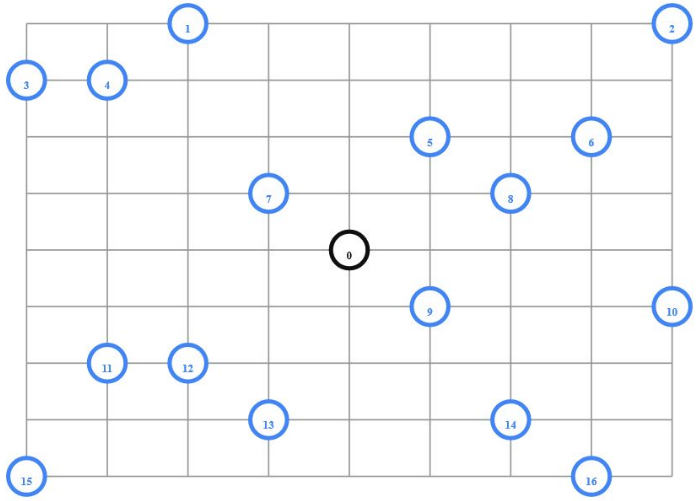
```
routing = pywrapcp.RoutingModel(
    data["num_locations"],
    data["num_vehicles"],
    data["depot"])
```

3. Provide the distance callback so the solver can compute the distances between locations

```
distance_callback = create_distance_callback(data)
routing.SetArcCostEvaluatorOfAllVehicles(distance_callback)
```

4. Add the distance dimension

```
add_distance_dimension(routing, distance_callback)
```



VRP - Example

1. You also have to specify a heuristic method to find the first solution

```
search_parameters = pywrapcp.RoutingModel.DefaultSearchParameters()
search_parameters.first_solution_strategy = (
    routing_enums_pb2.FirstSolutionStrategy.PATH_CHEAPEST_ARC) # pylint: disable=no-member
```

2. Finally, you can run the solver

```
assignment = routing.SolveWithParameters(search_parameters)
```

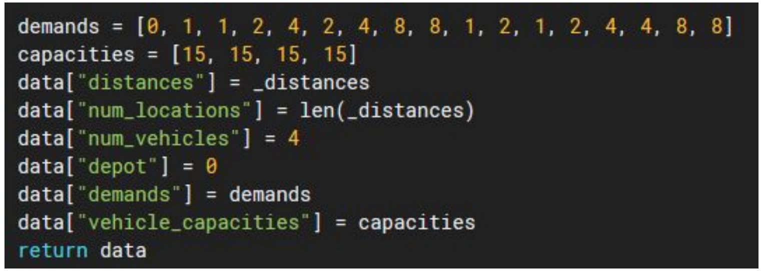
3. And print the solution

```
if assignment:
  print_solution(data, routing, assignment)
```

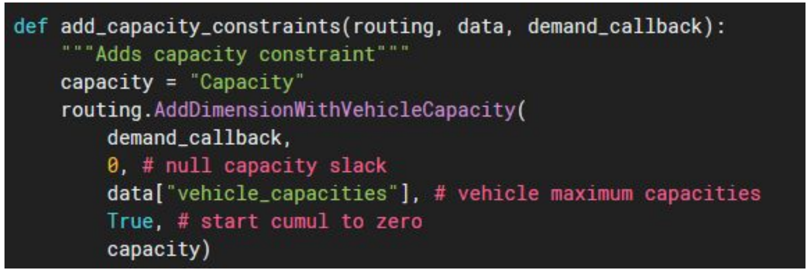
```
Route for vehicle 0:
 0 -> 8 -> 6 -> 2 -> 5 -> 0
Distance of route: 1552m

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
```

Capacitated Vehicle Routing Problem

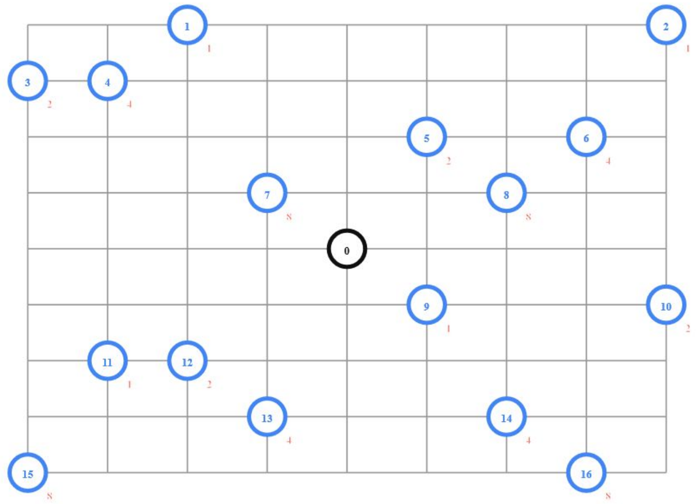
- CVRP is vehicle routing problem with additional constraints on the capacities of vehicles
  - physical quantity, such as weight or volume, corresponding to an item
- For example, at each location there is a demand corresponding to an item to be picked up. Also, each vehicle has a maximum capacity of 15
- And we do the same with our demand callback, so the solver can compute the demand of each node



```
demands = [0, 1, 1, 2, 4, 2, 4, 8, 8, 1, 2, 1, 2, 4, 4, 8, 8]
capacities = [15, 15, 15, 15]
data["distances"] = _distances
data["num_locations"] = len(_distances)
data["num_vehicles"] = 4
data["depot"] = 0
data["demands"] = demands
data["vehicle_capacities"] = capacities
return data
```

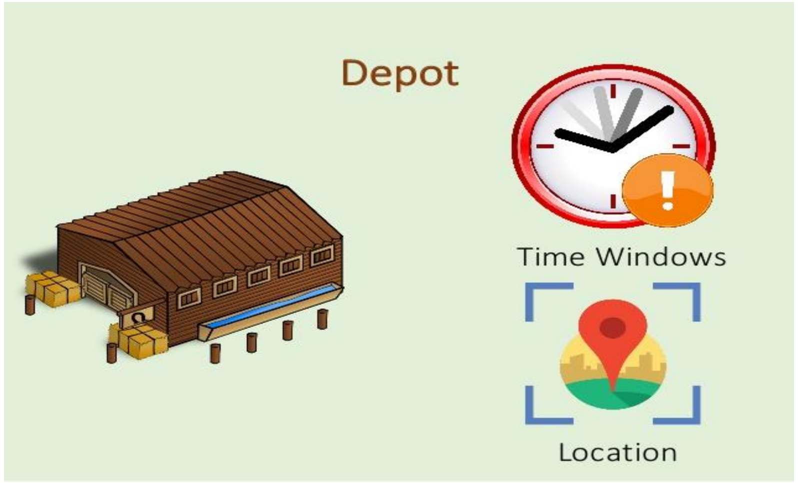


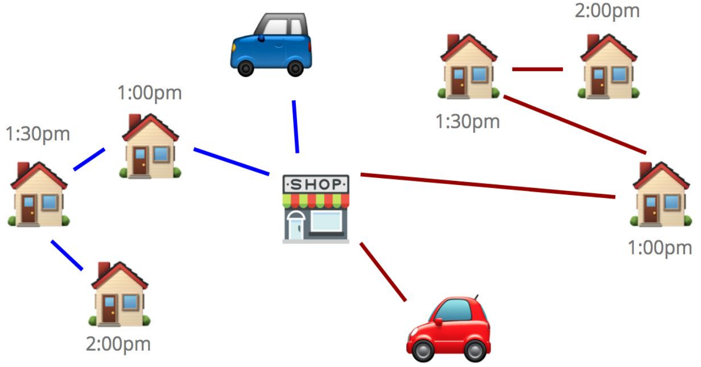
```
def add_capacity_constraints(routing, data, demand_callback):
    """Adds capacity constraint"""
    capacity = "Capacity"
    routing.AddDimensionWithVehicleCapacity(
        demand_callback,
        0, # null capacity slack
        data["vehicle_capacities"], # vehicle maximum capacities
        True, # start cumul to zero
        capacity)
```



Vehicle Routing Problem with Time Window Constraints

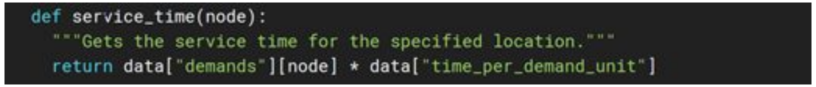
- The vehicle routing problem (VRP) is the m-TSP where a demand is associated with each city, and vehicle has a certain capacity.
- If we add a time window to each customer we get the vehicle routing problem with time windows. In addition to the capacity constraint, a vehicle now has to visit a customer within a certain time frame.
- The vehicle may arrive before the time window opens but the customer cannot be serviced until the time windows open. It is not allowed to arrive after the time window has closed.





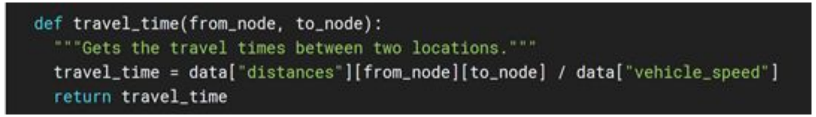
Vehicle Routing with Time Windows - What is new in this model?

1. Define service_time



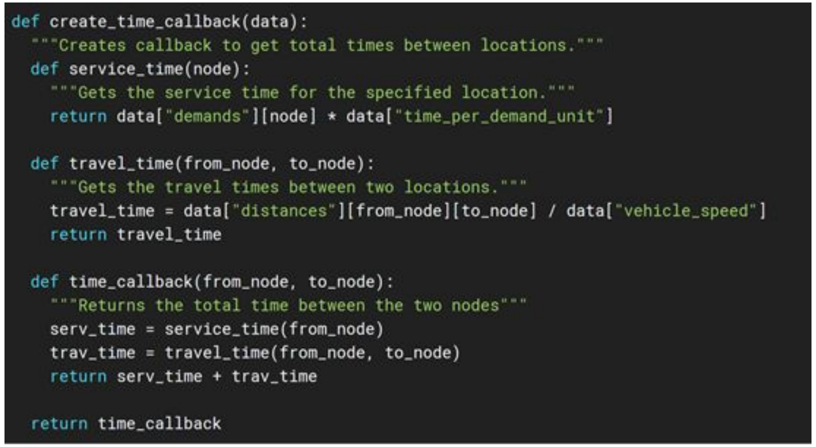
```python
def service_time(node):
    """Gets the service time for the specified location."""
    return data["demands"][node] * data["time_per_demand_unit"]
```

2. Define travel_time



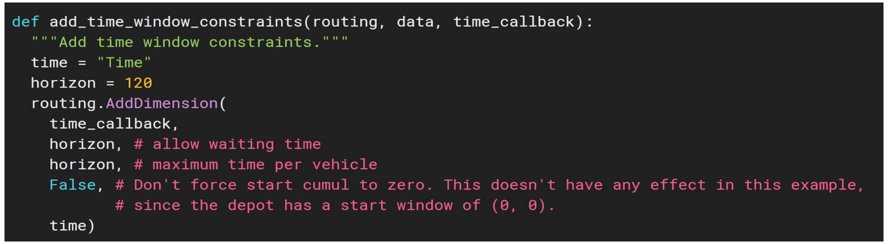
```python
def travel_time(from_node, to_node):
    """Gets the travel times between two locations."""
    travel_time = data["distances"][from_node][to_node] / data["vehicle_speed"]
    return travel_time
```

3. Define a time callback



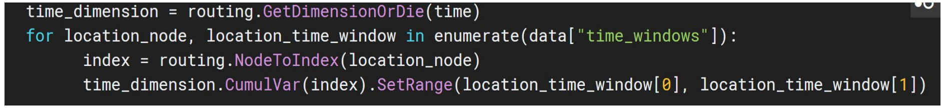
```python
def create_time_callback(data):
  """Creates callback to get total times between locations."""
  def service_time(node):
    """Gets the service time for the specified location."""
    return data["demands"][node] * data["time_per_demand_unit"]

  def travel_time(from_node, to_node):
    """Gets the travel times between two locations."""
    travel_time = data["distances"][from_node][to_node] / data["vehicle_speed"]
    return travel_time

  def time_callback(from_node, to_node):
    """Returns the total time between the two nodes"""
    serv_time = service_time(from_node)
    trav_time = travel_time(from_node, to_node)
    return serv_time + trav_time

  return time_callback
```

4. Adding time dimension



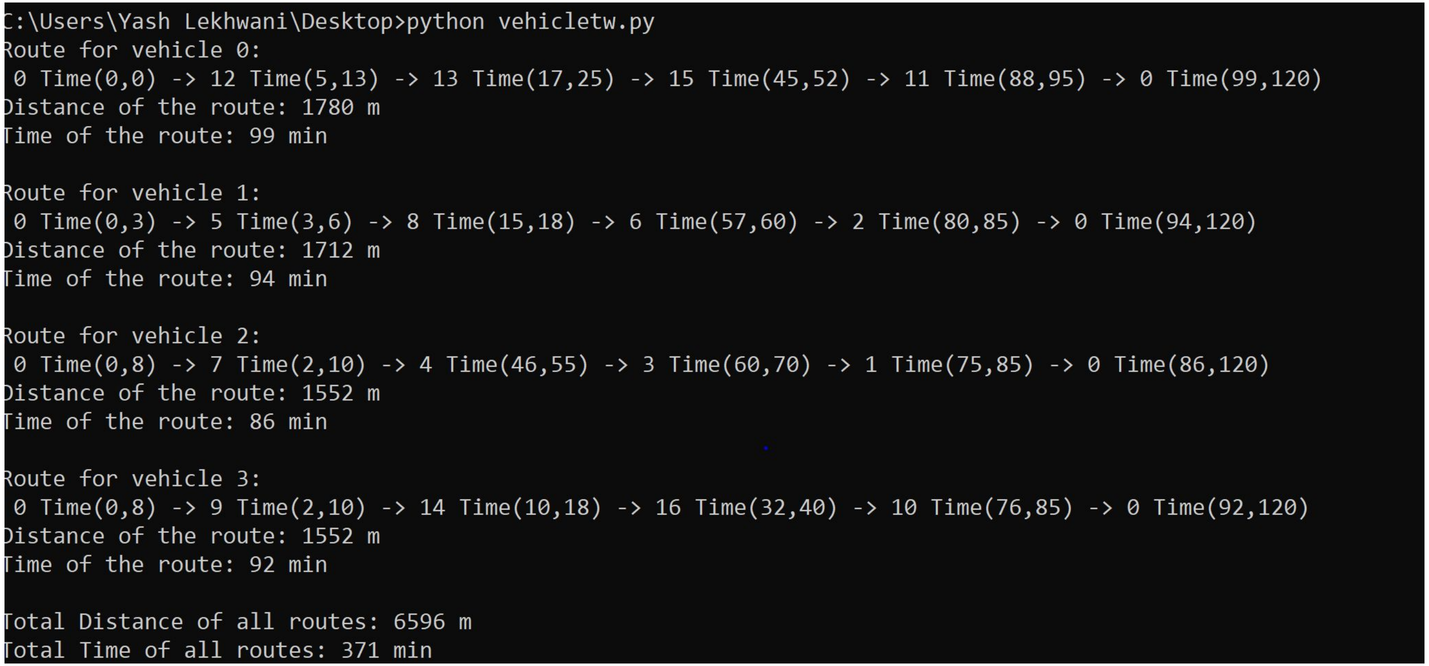
```python
def add_time_window_constraints(routing, data, time_callback):
  """Add time window constraints."""
  time = "Time"
  horizon = 120
  routing.AddDimension(
    time_callback,
    horizon, # allow waiting time
    horizon, # maximum time per vehicle
    False, # Don't force start cumul to zero. This doesn't have any effect in this example,
           # since the depot has a start window of (0, 0).
    time)
```

5. Time Window Constraint



```python
  time_dimension = routing.GetDimensionOrDie(time)
  for location_node, location_time_window in enumerate(data["time_windows"]):
        index = routing.NodeToIndex(location_node)
        time_dimension.CumulVar(index).SetRange(location_time_window[0], location_time_window[1])
```

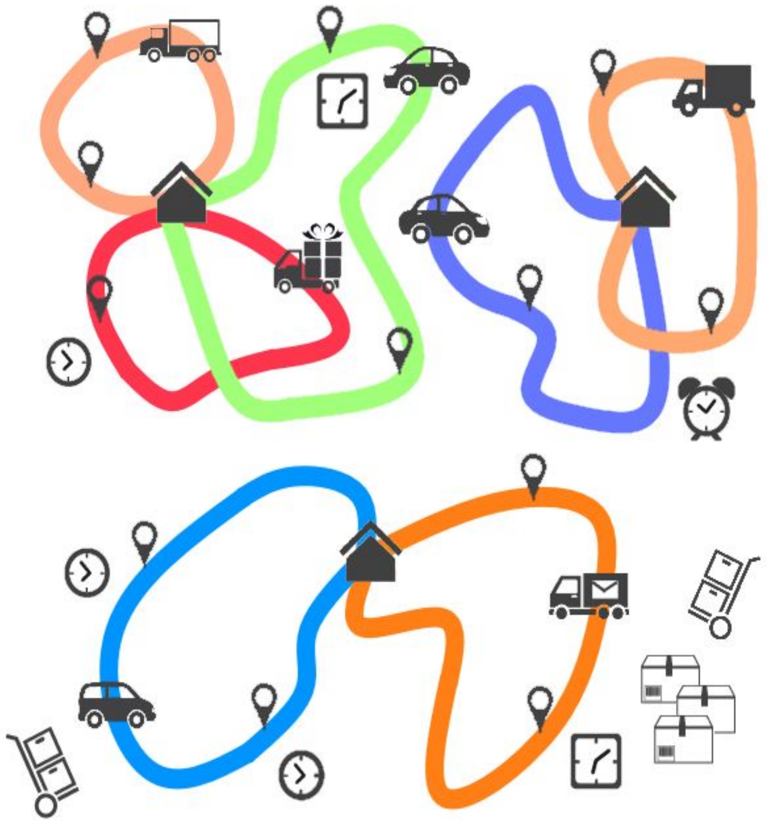
Vehicle Routing with Time Windows - What does the output look like?



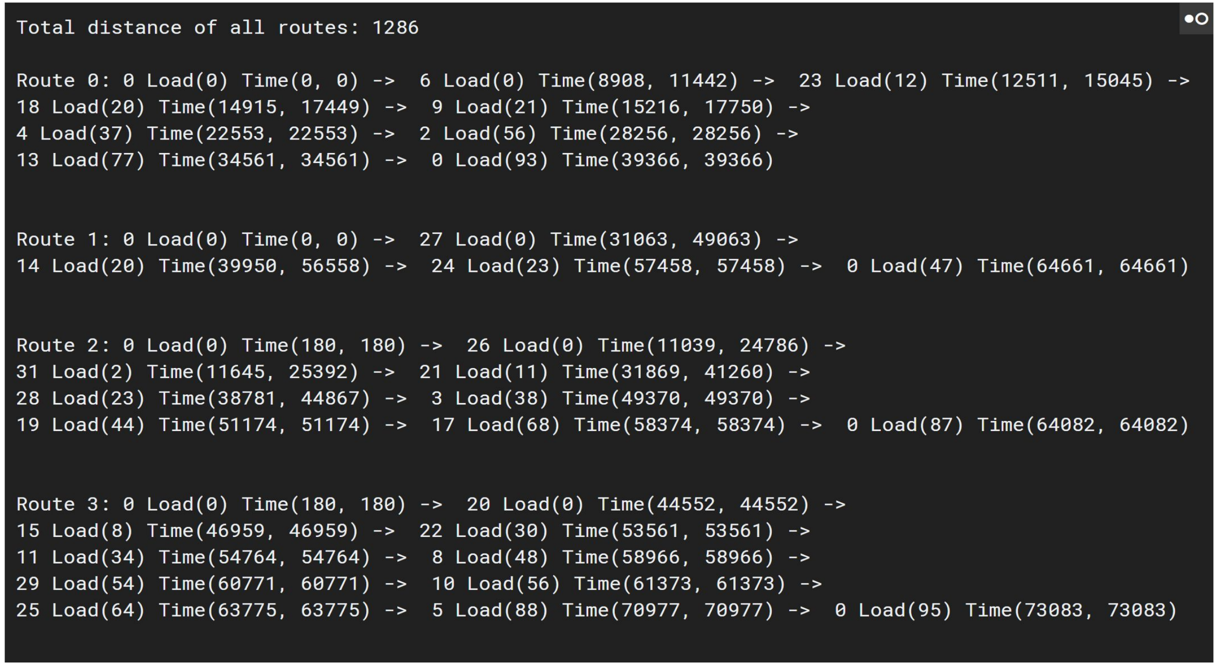
```
C:\Users\Yash Lekhwani\Desktop>python vehicletw.py
Route for vehicle 0:
 0 Time(0,0) -> 12 Time(5,13) -> 13 Time(17,25) -> 15 Time(45,52) -> 11 Time(88,95) -> 0 Time(99,120)
Distance of the route: 1780 m
Time of the route: 99 min

Route for vehicle 1:
 0 Time(0,3) -> 5 Time(3,6) -> 8 Time(15,18) -> 6 Time(57,60) -> 2 Time(80,85) -> 0 Time(94,120)
Distance of the route: 1712 m
Time of the route: 94 min

Route for vehicle 2:
 0 Time(0,8) -> 7 Time(2,10) -> 4 Time(46,55) -> 3 Time(60,70) -> 1 Time(75,85) -> 0 Time(86,120)
Distance of the route: 1552 m
Time of the route: 86 min

Route for vehicle 3:
 0 Time(0,8) -> 9 Time(2,10) -> 14 Time(10,18) -> 16 Time(32,40) -> 10 Time(76,85) -> 0 Time(92,120)
Distance of the route: 1552 m
Time of the route: 92 min

Total Distance of all routes: 6596 m
Total Time of all routes: 371 min
```

Vehicle Routing Problem with Resource Constraints

- In the vehicle routing problems, the constraints have been driven by customers' demands, such as the items they order and the times when they can take deliveries.
- This problem introduces constraints that are based on the delivery company's *resources*, including vehicles and drivers, the space and personnel to load and unload vehicles at the depot, fuel supplies, and so on. These are called *resource constraints*.
- Here, the changes are:
  - Change to the time dimension (cumulzero)
  - Add time windows for loading
  - Add time constraints



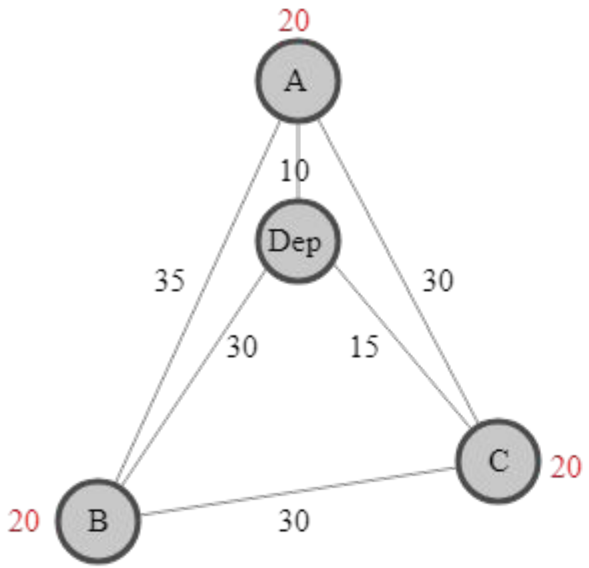
Vehicle Routing with Resource Constraints - What does the output look like?



```
Total distance of all routes: 1286

Route 0: 0 Load(0) Time(0, 0) ->  6 Load(0) Time(8908, 11442) ->  23 Load(12) Time(12511, 15045) ->
18 Load(20) Time(14915, 17449) ->  9 Load(21) Time(15216, 17750) ->
4 Load(37) Time(22553, 22553) ->  2 Load(56) Time(28256, 28256) ->
13 Load(77) Time(34561, 34561) ->  0 Load(93) Time(39366, 39366)


Route 1: 0 Load(0) Time(0, 0) ->  27 Load(0) Time(31063, 49063) ->
14 Load(20) Time(39950, 56558) ->  24 Load(23) Time(57458, 57458) ->  0 Load(47) Time(64661, 64661)


Route 2: 0 Load(0) Time(180, 180) ->  26 Load(0) Time(11039, 24786) ->
31 Load(2) Time(11645, 25392) ->  21 Load(11) Time(31869, 41260) ->
28 Load(23) Time(38781, 44867) ->  3 Load(38) Time(49370, 49370) ->
19 Load(44) Time(51174, 51174) ->  17 Load(68) Time(58374, 58374) ->  0 Load(87) Time(64082, 64082)


Route 3: 0 Load(0) Time(180, 180) ->  20 Load(0) Time(44552, 44552) ->
15 Load(8) Time(46959, 46959) ->  22 Load(30) Time(53561, 53561) ->
11 Load(34) Time(54764, 54764) ->  8 Load(48) Time(58966, 58966) ->
29 Load(54) Time(60771, 60771) ->  10 Load(56) Time(61373, 61373) ->
25 Load(64) Time(63775, 63775) ->  5 Load(88) Time(70977, 70977) ->  0 Load(95) Time(73083, 73083)
```

Dropping Visits in Infeasible Problems - Penalties

- For example, if you are given a capacitated vehicle routing problem (CVRP) in which the **total demands at all locations exceed the total capacity of the vehicles, no solution is possible.**
- In such cases, the vehicles must drop visits to some locations. The question is how to decide which visits to drop.
- To make the problem meaningful, we introduce new costs—called ***penalties***—at all locations. Whenever a visit to a location is dropped, the penalty is added to the total distance traveled.
- The solver then finds a route that minimizes the total distance plus the sum of the penalties for all dropped locations.
- Depot -> A -> C -> Depot
- This is the shortest route that visits two of the three locations (the distance is 55).



Common Routing Tasks

- Search limits:
    - `search_parameters.time_limit_ms = 30000`
    - `search_parameters.solution_limit = 100`
- Storing solutions in an array
- Setting initial routes for a search
- Setting start and end locations for routes
- Allowing arbitrary start and end locations

Google Directions API

- Google also provides a way to solve simple TSPs of real-world locations without downloading OR-Tools. If you have a Google Directions API key, you can solve TSPs of real-world locations with the Directions API, providing the locations in a URL and getting the response back as JSON.

- You'll need your own free Directions API key for development, or an enterprise key for commercial use.

- As an example, here's a URL that can be used to find a short tour of Boston to New York. If you want to try this from your browser, replace *API_KEY* at the end of the URL with your key:

This is our link for the TSP by Google maps from Boston to New York. Even you can do it!!

And the four of us! :)