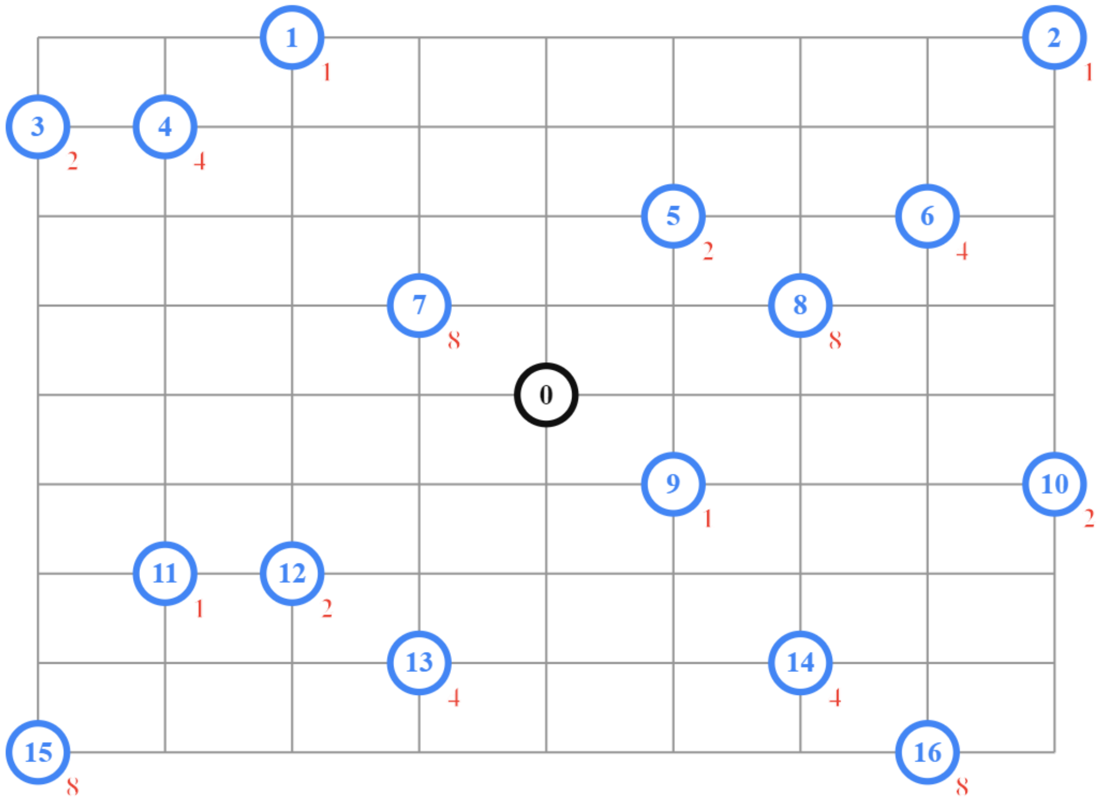
Capacity Constraints

The goal is to find optimal routes for multiple vehicles visiting a set of locations. (When there's only one vehicle, it reduces to the Traveling Salesman Problem.)

optimal routes -> minimize the length of the longest single route among all vehicles.



Creating The Data: -

For this problem, we take node 0 as the depot and assume number of vehicles = 4

```
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]
```

`AddDimensionWithVehicleCapacity` can be used in case of different capacities.

```
def create_data_model():
    # Stores the data for the problem.
    data = {}
    data['distance_matrix'] = [
        [
            0, 548, 776, 696, 582, 274, 502, 194, 308, 194, 536, 502, 388,
354,
            468, 776, 662
        ],
        [
            548, 0, 684, 308, 194, 502, 730, 354, 696, 742, 1084, 594, 480,
674,
            1016, 868, 1210
        ],
        [
```

776, 684, 0, 992, 878, 502, 274, 810, 468, 742, 400, 1278,
1164,
1130, 788, 1552, 754
],
[
696, 308, 992, 0, 114, 650, 878, 502, 844, 890, 1232, 514, 628,
822,
1164, 560, 1358
],
[
582, 194, 878, 114, 0, 536, 764, 388, 730, 776, 1118, 400, 514,
708,
1050, 674, 1244
],
[
274, 502, 502, 650, 536, 0, 228, 308, 194, 240, 582, 776, 662,
628,
514, 1050, 708
],
[
502, 730, 274, 878, 764, 228, 0, 536, 194, 468, 354, 1004, 890,
856,
514, 1278, 480
],
[
194, 354, 810, 502, 388, 308, 536, 0, 342, 388, 730, 468, 354,
320,
662, 742, 856
],
[
308, 696, 468, 844, 730, 194, 194, 342, 0, 274, 388, 810, 696,
662,
320, 1084, 514
],
[
194, 742, 742, 890, 776, 240, 468, 388, 274, 0, 342, 536, 422,
388,
274, 810, 468
],
[
536, 1084, 400, 1232, 1118, 582, 354, 730, 388, 342, 0, 878,
764,
730, 388, 1152, 354
],
[
502, 594, 1278, 514, 400, 776, 1004, 468, 810, 536, 878, 0,
114,
308, 650, 274, 844
],
[
388, 480, 1164, 628, 514, 662, 890, 354, 696, 422, 764, 114, 0,
194,
536, 388, 730
],
[
354, 674, 1130, 822, 708, 628, 856, 320, 662, 388, 730, 308,
194, 0,
342, 422, 536
],
[
468, 1016, 788, 1164, 1050, 514, 514, 662, 320, 274, 388, 650,

```
536,
            342, 0, 764, 194
        ],
        [
            776, 868, 1552, 560, 674, 1050, 1278, 742, 1084, 810, 1152,
274,
            388, 422, 764, 0, 798
        ],
        [
            662, 1210, 754, 1358, 1244, 708, 480, 856, 514, 468, 354, 844,
730,
            536, 194, 798, 0
        ],
    ]
    data['num_vehicles'] = 4
    data['depot'] = 0
    return data
```

To set up the example and compute the distance matrix, assign the following *x-y* coordinates to the locations shown in the city diagram (This is NOT needed for solving the VRP, but is used for better understanding. The Manhattan Distance method has been used to get these values):

```
[(456, 320), # location 0 - the depot
(228, 0),    # location 1
(912, 0),    # location 2
(0, 80),     # location 3
(114, 80),   # location 4
(570, 160),  # location 5
(798, 160),  # location 6
(342, 240),  # location 7
(684, 240),  # location 8
(570, 400),  # location 9
(912, 400),  # location 10
(114, 480),  # location 11
(228, 480),  # location 12
(342, 560),  # location 13
(684, 560),  # location 14
(0, 640),    # location 15
(798, 640)]  # location 16
```

Like in TSP, we need to create a ‘distance call-back function’ which returns the distance between any 2 nodes.

```
def distance_callback(from_index, to_index):
    # 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)
```

We also need to add the demand call back function (similar to distance call back).

```
def demand_callback(from_index):
    # 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')
```

We use the dynamic programming approach. So, we compute the cumulative distance travelled by each vehicle along its route.

To create the distance dimension, we use the 'AddDimension' method. The argument 'transit_callback_index' is the index for the distance_callback.

```
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 method SetGlobalSpanCostCoefficient sets a large coefficient (100) for the *global span* of the routes. Here, it is the maximum of the distances of the routes.

Solution function: -

```
def print_solution(data, manager, routing, solution):
    total_distance = 0
    total_load = 0
    for vehicle_id in range(data['num_vehicles']):
        index = routing.Start(vehicle_id)
        plan_output = 'Route for vehicle {}:\n'.format(vehicle_id)
        route_distance = 0
        route_load = 0
        while not routing.IsEnd(index):
            node_index = manager.IndexToNode(index)
            route_load += data['demands'][node_index]
            plan_output += ' {0} Load({1}) -> '.format(node_index,
route_load)
            previous_index = index
            index = solution.Value(routing.NextVar(index))
            route_distance += routing.GetArcCostForVehicle(
                previous_index, index, vehicle_id)
        plan_output += ' {0}
Load({1})\n'.format(manager.IndexToNode(index),
                                                 route_load)
```

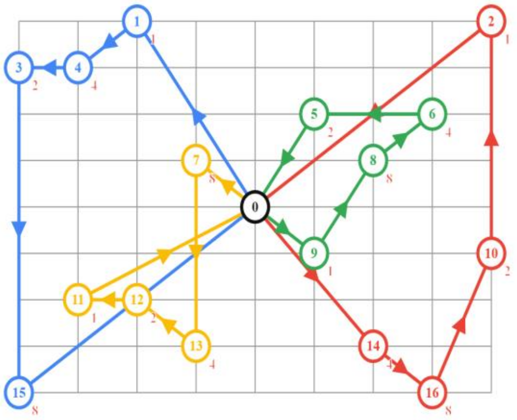
```
        plan_output += 'Distance of the route:
{}m\n'.format(route_distance)
        plan_output += 'Load of the route: {}\n'.format(route_load)
        print(plan_output)
        total_distance += route_distance
        total_load += route_load
    print('Total distance of all routes: {}m'.format(total_distance))
    print('Total load of all routes: {}'.format(total_load))
```

'main' function: -

```
def main():
    # Instantiate the data problem.
    data = create_data_model()

    # Create the routing index manager.
    manager = pywrapcp.RoutingIndexManager(len(data['distance_matrix']),
                                           data['num_vehicles'],
data['depot'])

    # Create Routing Model.
    routing = pywrapcp.RoutingModel(manager)


    # Create and register a transit callback.
    def distance_callback(from_index, to_index):
       # 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)

    # Define cost of each arc.
    routing.SetArcCostEvaluatorOfAllVehicles(transit_callback_index)

    # Add Capacity constraint.
    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')

    # Setting first solution heuristic.
    search_parameters = pywrapcp.DefaultRoutingSearchParameters()
    search_parameters.first_solution_strategy = (
```

```
        routing_enums_pb2.FirstSolutionStrategy.PATH_CHEAPEST_ARC)
    # Solve the problem.
    solution = routing.SolveWithParameters(search_parameters)

    # Print solution on console.
    if solution:
        print_solution(data, manager, routing, solution)


if __name__ == '__main__':
    main()
```

Output: - (Screenshot)



```
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
```