Operation Research Problems Solving in Python

Prepared by Saurav Barua,
Assistant Professor,
Department of Civil Engineering,
Daffodil International University,
Dhaka-1207

Contents

Sl No.	Topics	Pages
1	Chapter 1: Installation of Google OR Tools for Python	1
2	Chapter 2: Finding Feasible Solution	2-3
3	Chapter 3: Mixed Integer Problem	4-5
4	Chapter 4: Traveling Salesman Problem	6-8
5	Chapter 5: Vehicle Routing Problem with Capacity Constraint	9-14

Chapter One

Installation of Google OR Tools for Python

Install OR-Tools

Though Google created OR-Tools in C++, you can use it with Python, Java, or C# (on the .NET platform).

Install OR-Tools for Python

The fastest way to get OR-Tools is to install the Python binary version. If you already have Python 2.7 or 3.5+ (and the Python package manager PIP) installed, you can do so as follows:

```
python -m pip install --upgrade --user ortools
```

After the installation is complete, you are ready to get started with OR-Tools for Python.

Chapter Two

Finding Feasible Solution

Example: finding a feasible solution

A simple example

Here's a simple example of a linear programming problem.

Maximize $3x + 4y$ subject to the following constraints:

$x + 2y$	$\leq$	14
$3x - y$	$\geq$	0
$x - y$	$\leq$	2

Program

```
from __future__ import print_function
from ortools.linear_solver import pywraplp



def LinearProgrammingExample():
    """Linear programming sample."""
    # Instantiate a Glop solver, naming it LinearExample.
    solver = pywraplp.Solver('LinearProgrammingExample',
                             pywraplp.Solver.GLOP_LINEAR_PROGRAMMING)

    # Create the two variables and let them take on any value.
    x = solver.NumVar(-solver.infinity(), solver.infinity(), 'x')
    y = solver.NumVar(-solver.infinity(), solver.infinity(), 'y')
```

```
    # Constraint 0: x + 2y <= 14.
    constraint0 = solver.Constraint(-solver.infinity(), 14)
    constraint0.SetCoefficient(x, 1)
    constraint0.SetCoefficient(y, 2)

    # Constraint 1: 3x - y >= 0.
    constraint1 = solver.Constraint(0, solver.infinity())
    constraint1.SetCoefficient(x, 3)
    constraint1.SetCoefficient(y, -1)

    # Constraint 2: x - y <= 2.
    constraint2 = solver.Constraint(-solver.infinity(), 2)
    constraint2.SetCoefficient(x, 1)
    constraint2.SetCoefficient(y, -1)

    # Objective function: 3x + 4y.
    objective = solver.Objective()
    objective.SetCoefficient(x, 3)
    objective.SetCoefficient(y, 4)
    objective.SetMaximization()

    # Solve the system.
    solver.Solve()
    opt_solution = 3 * x.solution_value() + 4 * y.solution_value()
    print('Number of variables =', solver.NumVariables())
    print('Number of constraints =', solver.NumConstraints())
    # The value of each variable in the solution.
    print('Solution:')
    print('x = ', x.solution_value())
    print('y = ', y.solution_value())
    # The objective value of the solution.
    print('Optimal objective value =', opt_solution)


LinearProgrammingExample()
```

Chapter Three

Mixed Integer Problem

MIP Example (Mixed Integer Problem)

The following is a simple example of a mixed-integer programming problem:

Maximize $x + 10y$ subject to the following constraints:

$x + 7y$	≤	17.5
x	≤	3.5
x	≥	0
y	≥	0
x, y integers		

Program

```
from __future__ import print_function
from ortools.linear_solver import pywraplp


def main():
    # Create the mip solver with the CBC backend.
    solver = pywraplp.Solver('simple_mip_program',
                             pywraplp.Solver.CBC_MIXED_INTEGER_PROGRAMMING)

    infinity = solver.infinity()
    # x and y are integer non-negative variables.
    x = solver.IntVar(0.0, infinity, 'x')
    y = solver.IntVar(0.0, infinity, 'y')

    print('Number of variables =', solver.NumVariables())
```

```
    # x + 7 * y <= 17.5.
    solver.Add(x + 7 * y <= 17.5)

    # x <= 3.5.
    solver.Add(x <= 3.5)

    print('Number of constraints =', solver.NumConstraints())

    # Maximize x + 10 * y.
    solver.Maximize(x + 10 * y)

    result_status = solver.Solve()
    # The problem has an optimal solution.
    assert result_status == pywraplp.Solver.OPTIMAL

    # The solution looks legit (when using solvers others than
    # GLOP_LINEAR_PROGRAMMING, verifying the solution is highly
recommended!).
    assert solver.VerifySolution(1e-7, True)

    print('Solution:')
    print('Objective value =', solver.Objective().Value())
    print('x =', x.solution_value())
    print('y =', y.solution_value())

    print('\nAdvanced usage:')
    print('Problem solved in %f milliseconds' % solver.wall_time())
    print('Problem solved in %d iterations' % solver.iterations())
    print('Problem solved in %d branch-and-bound nodes' % solver.nodes())


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

Chapter Four

Traveling Salesman Problem

Traveling Salesman Problem

The *distance matrix* is an array whose i, j entry is the distance from location i to location j in miles, where the locations are given in the order below:

0. New York 1. Los Angeles 2. Chicago 3. Minneapolis 4. Denver 5. Dallas 6. Seattle 7. Boston 8. San Francisco 9. St. Louis 10. Houston 11. Phoenix 12. Salt Lake City

As an alternative to a distance matrix, you could provide a *time matrix* which contains the travel times between locations.

The data also includes:

- The number of vehicles in the problem, which is 1 because this is a TSP. For general routing problems, the number of vehicles can be greater than 1.
- The *depot*: the starting location for the route. In this case, the depot is 0, which corresponds to New York City.

Program

```
"""Simple travelling salesman problem between cities."""

from __future__ import print_function
from ortools.constraint_solver import routing_enums_pb2
from ortools.constraint_solver import pywrapcp



def create_data_model():
    """Stores the data for the problem."""
    data = {}
    data['distance_matrix'] = [
        [0, 2451, 713, 1018, 1631, 1374, 2408, 213, 2571, 875, 1420, 2145,
1972],
        [2451, 0, 1745, 1524, 831, 1240, 959, 2596, 403, 1589, 1374, 357,
```

```
579],
        [713, 1745, 0, 355, 920, 803, 1737, 851, 1858, 262, 940, 1453,
1260],
        [1018, 1524, 355, 0, 700, 862, 1395, 1123, 1584, 466, 1056, 1280,
987],
        [1631, 831, 920, 700, 0, 663, 1021, 1769, 949, 796, 879, 586, 371],
        [1374, 1240, 803, 862, 663, 0, 1681, 1551, 1765, 547, 225, 887,
999],
        [2408, 959, 1737, 1395, 1021, 1681, 0, 2493, 678, 1724, 1891, 1114,
701],
        [213, 2596, 851, 1123, 1769, 1551, 2493, 0, 2699, 1038, 1605, 2300,
2099],
        [2571, 403, 1858, 1584, 949, 1765, 678, 2699, 0, 1744, 1645, 653,
600],
        [875, 1589, 262, 466, 796, 547, 1724, 1038, 1744, 0, 679, 1272,
1162],
        [1420, 1374, 940, 1056, 879, 225, 1891, 1605, 1645, 679, 0, 1017,
1200],
        [2145, 357, 1453, 1280, 586, 887, 1114, 2300, 653, 1272, 1017, 0,
504],
        [1972, 579, 1260, 987, 371, 999, 701, 2099, 600, 1162, 1200, 504,
0],
    ]  # yapf: disable
    data['num_vehicles'] = 1
    data['depot'] = 0
    return data


def print_solution(manager, routing, assignment):
    """Prints assignment on console."""
    print('Objective: {} miles'.format(assignment.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 = assignment.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)
```

```
def main():
    """Entry point of the program."""
    # 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)


    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)

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

    # Setting first solution heuristic.
    search_parameters = pywrapcp.DefaultRoutingSearchParameters()
    search_parameters.first_solution_strategy = (
        routing_enums_pb2.FirstSolutionStrategy.PATH_CHEAPEST_ARC)

    # Solve the problem.
    assignment = routing.SolveWithParameters(search_parameters)

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


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

Chapter Five

Vehicle Routing Problem with Capacity Constraint

Capacity Constraints

Overview

The *capacitated vehicle routing problem* (CVRP) is a VRP in which vehicles with limited carrying capacity need to pick up or deliver items at various locations.

The new items in the data are:

- Demands: Each location has a *demand* corresponding to the quantity—for example, weight or volume—of the item to be picked up.
- Capacities: Each vehicle has a *capacity*: the maximum quantity that the vehicle can hold. As a vehicle travels along its route, the total quantity of the items it is carrying can never exceed its capacity.

Problems with multiple cargo types and capacities

In more complex CVRPs, each vehicle might carry several different types of cargo, with a maximum capacity for each type. For example, a fuel delivery truck might carry several types of fuel, using multiple tanks with differing capacities. To handle problems like these, just create a different capacity callback and dimension for each cargo type (making sure to assign them unique names).

Program

```
"""Capacited Vehicles Routing Problem (CVRP)."""

from __future__ import print_function
from ortools.constraint_solver import routing_enums_pb2
```

```
from ortools.constraint_solver import pywrapcp


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


def print_solution(data, manager, routing, assignment):
    """Prints assignment on console."""
    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 = assignment.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))
```

```
def main():
    """Solve the CVRP problem."""
    # 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):
        """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)

    # 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.
    assignment = routing.SolveWithParameters(search_parameters)

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


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