Computational Methods And Optimization

Group 11: Anagha Vasista, Chaitanya Modi, Manan Chawla, Pratham Arora

Capacitated Vehicle Routing Problem (CVRP) With Time Constraints

Project Update - 3

24th November, 2023

Update - 1

Work done:

1. We went through some websites, research papers and YouTube videos (kindly find them in references) to understand how to work through our project problem.
2. Two of our team members (Anagha Vasista and Chaitanya Modi) learnt to work on Git Hub.

Work under progress:

1. We are going through VRP to understand the basics through the VRP code using OR tools and Python.
2. We are currently working on understanding the Mixed Integer Linear Program and also the code to implement our project idea.

Further Work:

We hope to complete going through the entire learning phase for the project by next week and start implementing our solution.

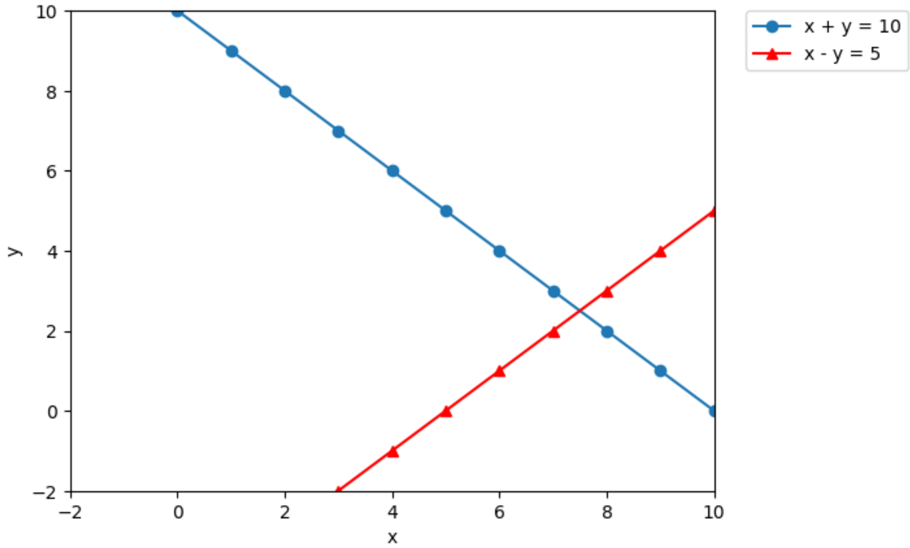
Other updates:

We are going to look into Google Maps API for using real-world locations, distances and routes.

Update - 2:

This week we learnt the MILP problem and solved a few examples. In the update, we have illustrated the Knapsack Problem (using the Tabular/Simplex method) as an example of an MILP. We also learnt Branch and bound method, and dynamic programming as methods to solve MILPs.

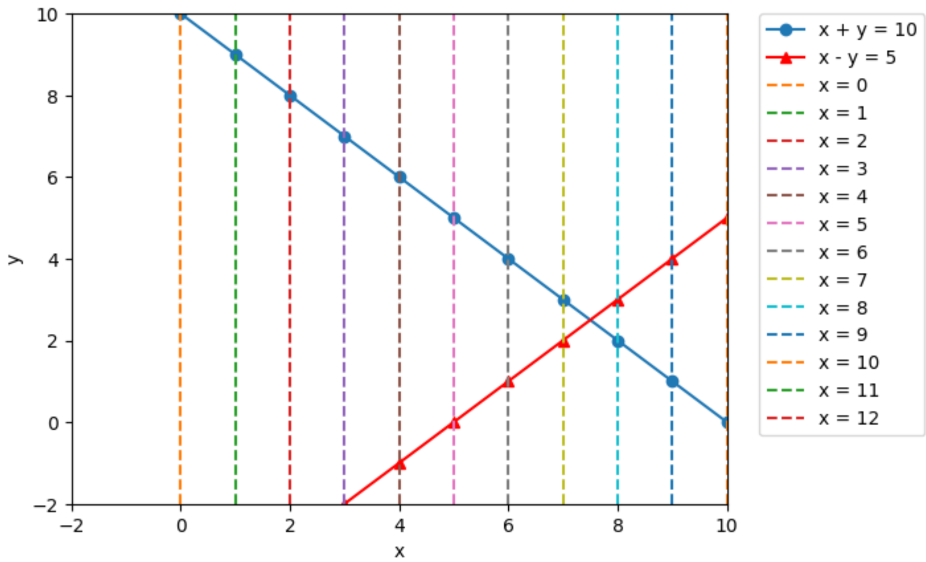
What is Linear Programming?



$x_i \in \Re^n$ ie x can take any rational value.

So, this above equation has a Solution at (7.5, 2.5)

What is Mixed Integer Linear Programming?



$x_i \in \{Z, \Re\}^n$ ie x can take any integer value.

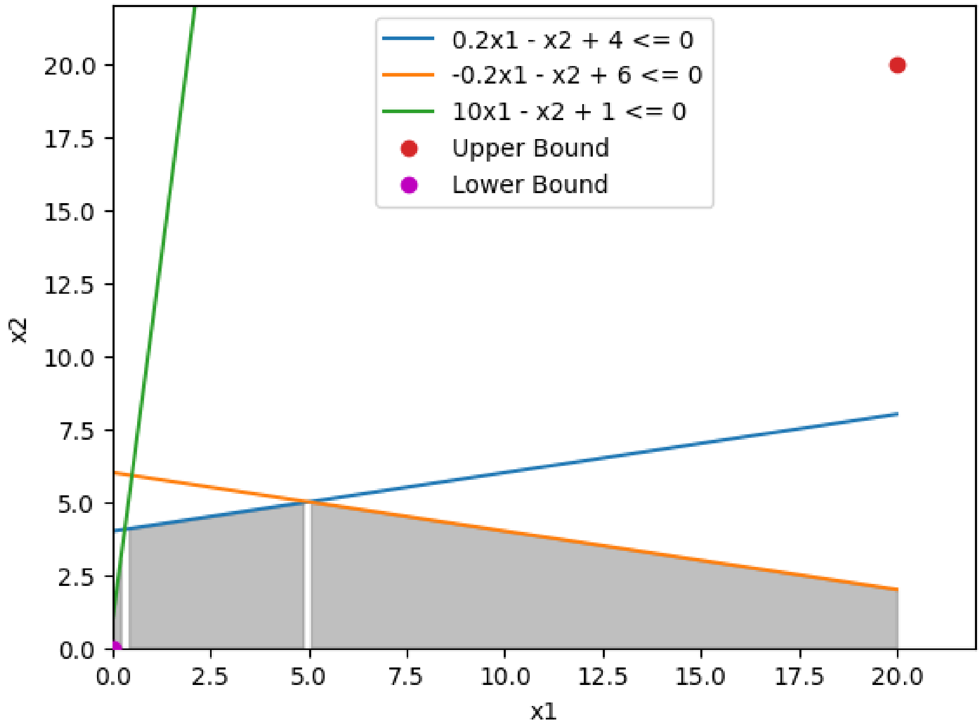
Now that there exists another constraint that the feasible solutions can only be integer values the same set of equations no longer have a solution because now the intersection or the feasible solutions should lie on the dashed lines.

Difference between LP and MILP:

An LP (linear program) involves minimising (or maximising) a linear function subject to linear constraints on the variables. Any solution that satisfies the constraints is feasible. A MILP is an LP with the addition of integrality restrictions on some or all of the variables.
At first glance, MILPs look easier to solve because the finite set of feasible solutions is much smaller than the uncountably infinite number of solutions for an LP.
But, when you take a closer look at the feasible region to be optimised, you can see that for an LP, you only need to look at the vertices of the feasible region. The simplex method does just that and therefore can solve LPs more efficiently than any known algorithm for MILPs.
The barrier method also solves LPs and moves through the interior of the feasible region rather than along the boundary. But it too has a way to efficiently move from one feasible solution to another. This is much more challenging for MILPs, and the branch and bound algorithms currently in favour use binary search trees that can grow exponentially.

Some basic/common ways to solve MILP:

1. **Simplex Method**



Way to Solve MILPs by this method:
1. Initial step:
   Start in a feasible basic solution at a vertex.
2. Iterative step:
   Move to a better feasible basic solution at an adjacent vertex
3. Optimality test:
A feasible basic solution at a vertex is optimal when it is equal or better than feasible basic solutions at all adjacent vertices.

2. **Branch & Bound Method**

$x_1 + x_2 \leq 50$

$4x_1 + 7x_2 \leq 280$

$x_1, x_2 \geq 0$

The first step is to relax the integer constraint. We have two extreme points for the first equation that form a line: [x1 x2] = [50 0] and [0 50]

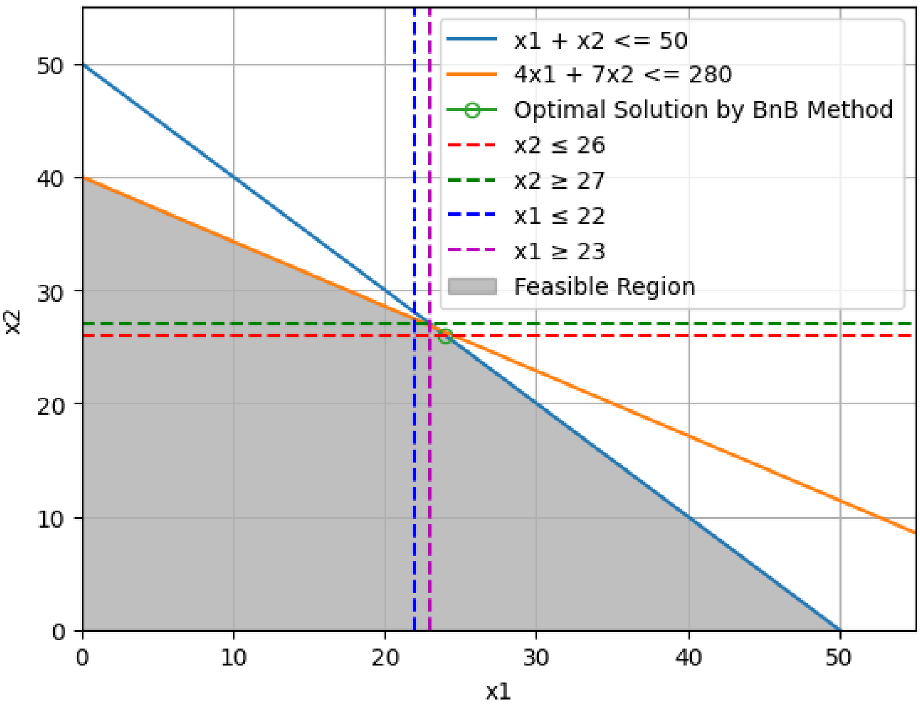
We get the second line with the vector points [0 40] and [70 0].

The third point is [0 0].

This is a convex hull region so the solution lies on one of the vertices of the region. We can find the intersection using row reduction, which is [70/3 80/3] with a value of 276.667

We test the other extreme points by sweeping the line over the convex region and find this is the maximum over the Reals.

We choose the variable with the maximum fractional part, in this case x2 becomes the parameter for the branch and bound method. We branch to x2 ≤ 26 and obtain 276 at (24, 26). We have reached an integer solution so we move to the other branch x2 ≥ 27 and obtain 275.75 at (22.75, 27). We don't have an integer solution so we branch x1 to x1 ≤ 22 and we find 274.571 (22, 27.4286). We try the other branch x1 ≥ 23 and there are no feasible solutions. Thus, the maximum is 276 with x1 at 24 and x2 at 26.



Knapsack Problem (0-1 Method):
This is the tabular method for solving the Knapsack Problem.

Example:
Suppose we have the following items with their values (v_i) *and weights* (ω_i):

Item	Value	Weight
1	3	2
2	4	3
3	5	4
4	6	5

And the maximum weight capacity (W) *of the knapsack is* 5 *and total items* (n) *is* 4

Tabular Method:

Create a table with rows representing items (from 0 *to* 4*) and columns representing the knapsack's weight capacity (from* 0 *to* 5*). Initialize the table with zeros. terate through each item and each possible knapsack weight capacity, updating the table based on the optimal value.*

The entry $T[i][j]$ *represents the maximum value that can be obtained with the first i items and a knapsack capacity of j.*

$$T[i][j] = max(T[i-1][j],\ v_i + T[i-1][j-\omega_i])$$

If the weight of the current item (ω_i) *is greater than the current capacity* (j)*, we simply copy the value from the cell above. Otherwise, we consider whether it's more valuable to include the current item or not.*

$$\begin{bmatrix} & 0 & 1 & 2 & 3 & 4 & 5 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 3 & 3 & 3 & 3 \\ 2 & 0 & 0 & 3 & 4 & 4 & 7 \\ 3 & 0 & 0 & 3 & 4 & 5 & 7 \\ 4 & 0 & 0 & 3 & 4 & 5 & 7 \end{bmatrix}$$

To identify the items that must be put into the knapsack to obtain the maximum profit, start from the bottom – right corner of the table and backtrack to reconstruct the selected items.
If $T[i][j]$ *is equal to* $T[i-1][j]$*, the item i was not selected. Otherwise, it was selected. In this case,* 7 *is in item* 2 *and not in item* 1*, so item* 2 *will be selected. Then we look for* $T[i][j] - i[v_i]$*.*
$7 - 4 = 3$ *and repeat the process.*
After all entries are scanned, the marked labels represent the items that must be put into the knapsack.
In this case the maximum possible value that can be put into the knapsack $= 7$*. We put items* 1 *and* 2 *intot he knapsack.*

Tasks for Week 3:

1. For next week, we will learn how VRP can be solved using the methods we learnt this week, and then move on to CVRP and work on the code to solve CVRP using Python and Google OR tools.
2. We will also be solving and explaining the Knapsack problem using Python.

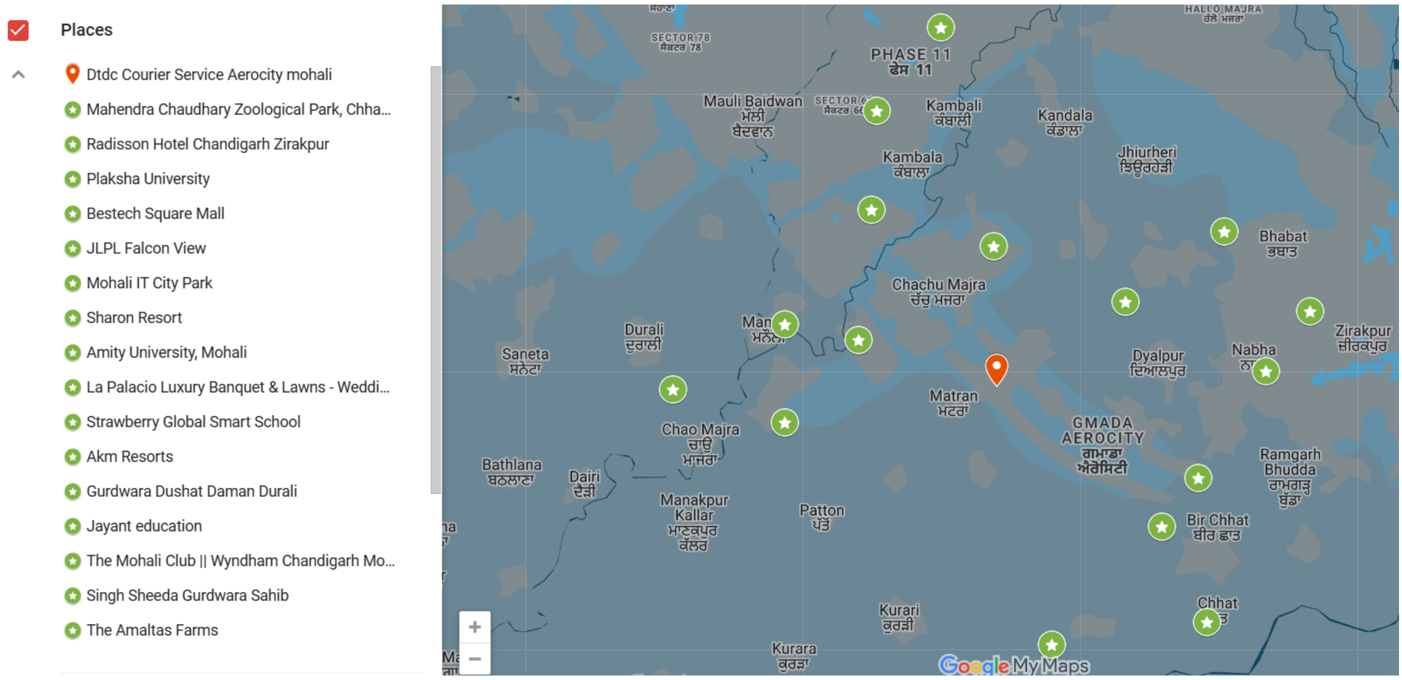
References:

1. https://developers.google.com/optimization/routing/vrp
2. https://satpal-singh.medium.com/solving-cvrptw-using-or-tools-in-python-214247f6d680
3. https://youtu.be/v9tUEsHD6BE?feature=shared
4. https://www.youtube.com/playlist?list=PLaoe2MTbJBvpFPyMMSOB-WrHofdHo3e74
5. Guignard, M. M., & Spielberg, K. (1972). Mixed-integer Algorithms for the (0,1) Knapsack Problem. IBM Journal of Research and Development, 16(4), 424–430. doi:10.1147/rd.164.0424 http://www.or.deis.unibo.it/kp/Chapter2.pdf
6. https://en.wikipedia.org/wiki/Knapsack_problem
7. H4.pdf (tue.nl)
8. Using the Simplex Method in Mixed Integer Linear Programming (loria.fr)
9. https://www.math.wsu.edu/students/odykhovychnyi/M201-04/Ch06_1-2_Simplex_Method.pdf
10. 0/1 Knapsack Problem | Dynamic Programming | Example | Gate Vidyalay

Update 3:

This week we worked on the VRP and CVRP codes to see what exactly we solve and how exactly we solve them using commercial solvers like OR-tools by Google.
We worked on these codes because our Final Code will be the code of VRP with “Capacity” and “Time Window” Constraints so we programmed our VRP and VRP with “Capacity” constraint.

Before going onto the codes we finalized our real world locations which are uniformly distributed in all directions of the depot.



Here our Depot is in Red and in Green are the Locations.

Our Distance Matrix for these Locations:
(Values in metres)

0	6152	5761	4462	5776	4888	3133	3082	6750	4283	6604	6762	10501	8860	7795	5622	5433
8588	0	4863	12814	14128	13240	11485	11434	3106	2774	6459	5864	18853	7962	16147	3479	13785
5803	4900	0	10029	11343	10455	8700	8649	5533	3066	3895	1170	16068	3268	13362	2132	11000
6560	10598	10207	0	8386	7498	5743	2215	11197	8730	9214	11208	3336	10632	10405	10068	1916
7843	11881	11490	8646	0	2916	5744	5766	12479	10012	10438	12491	7161	11856	2131	11351	5012
5723	9761	9370	6526	3335	0	3623	3646	10359	7892	8317	10371	6809	9735	4997	9231	4660
5019	9057	8666	5822	5495	2772	0	4442	9655	7188	4832	9667	7301	6250	7514	8527	5152
5665	9703	9312	4024	5969	5081	4758	0	10302	7835	8319	10313	10063	9737	7988	9173	5625
12799	9457	11767	17025	18339	17451	15696	15645	0	10236	13363	12767	16120	14865	20358	11628	17996
7949	4577	6917	12175	13489	12601	10846	10795	2467	0	8513	7917	18481	10015	15508	6778	13146
7978	6407	3778	8824	10138	9250	4861	7444	7040	4573	0	4779	14862	3442	6459	3639	9794
6719	5816	1153	10945	12259	11371	9616	9565	6449	3982	4811	0	16984	2880	14278	3048	11916
12264	16302	15911	7326	6804	5916	11447	4293	16901	14434	14918	16912	0	16336	8823	15772	2197
9394	7908	3245	10240	11554	10666	6277	8860	8541	6074	3441	2879	16278	0	7875	5140	11210
9492	13530	13139	10295	2163	4565	7393	7415	14128	11661	6459	14140	8810	7877	0	13000	6661
3671	2768	2536	7897	9211	8323	6568	6517	3401	934	4132	3537	13936	5635	11230	0	8868
7274	11312	10921	3857	4654	3766	6594	2873	11911	9444	9928	11922	2197	11346	6673	10782	0

Codes:
Though available on Github at https://github.com/prats3992/CVRPTW
Here are the snippets:

1. VRP

```
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 = {}
    with open("distance_matrix.txt", "r") as f:
        data["distance_matrix"] = [[int(num) for num in line.split("\t")] for
line in f.readlines()]
    data["num_vehicles"] = 4
    data["depot"] = 0
    return data


def print_solution(data, manager, routing, solution):
    """Prints solution on console."""
    print(f"Objective: {solution.ObjectiveValue()}")
    max_route_distance = 0
    for vehicle_id in range(data["num_vehicles"]):
        index = routing.Start(vehicle_id)
        plan_output = f"Route for vehicle {vehicle_id}:\n"
        route_distance = 0
        while not routing.IsEnd(index):
            plan_output += f" {manager.IndexToNode(index)} -> "
            previous_index = index
            index = solution.Value(routing.NextVar(index))
            route_distance += routing.GetArcCostForVehicle(
                previous_index, index, vehicle_id
            )
        plan_output += f"{manager.IndexToNode(index)}\n"
        plan_output += f"Distance of the route: {route_distance}m\n"
        print(plan_output)
        max_route_distance = max(route_distance, max_route_distance)
    print(f"Maximum of the route distances: {max_route_distance}m")



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)

    # 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 Distance constraint.
    dimension_name = "Distance"
    routing.AddDimension(
        transit_callback_index,
        0,  # no slack
        30000,  # vehicle maximum travel distance
        True,  # start cumul to zero
        dimension_name,
    )
    distance_dimension = routing.GetDimensionOrDie(dimension_name)
    distance_dimension.SetGlobalSpanCostCoefficient(100)

    # 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)
    else:
        print("No solution found !")
if __name__ == "__main__":
    main()
```

In this the cost is the “Distance Travelled” by the Vehicle.
We get the Output as

```
Objective: 2501362
Route for vehicle 0:
 0 ->  4 ->  14 ->  5 -> 0
Distance of the route: 18195m

Route for vehicle 1:
 0 ->  1 ->  9 ->  8 -> 0
Distance of the route: 24192m

Route for vehicle 2:
 0 ->  6 ->  10 ->  13 ->  11 ->  2 ->  15 -> 0
Distance of the route: 21242m

Route for vehicle 3:
 0 ->  3 ->  12 ->  16 ->  7 -> 0
Distance of the route: 18533m

Maximum of the route distances: 24192m
```

2. VRP with “Capacity” Constraints

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

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 = {}
    with open("distance_matrix.txt") as f:
        data["distance_matrix"] = [[int(num) for num in line.split('\t')] for
line in f.readlines()]
    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, solution):
    """Prints solution on console."""
    print(f"Objective: {solution.ObjectiveValue()}")
    total_distance = 0
    total_load = 0
    for vehicle_id in range(data["num_vehicles"]):
        index = routing.Start(vehicle_id)
        plan_output = f"Route for vehicle {vehicle_id}:\n"
        route_distance = 0
```

```python
        route_load = 0
        while not routing.IsEnd(index):
            node_index = manager.IndexToNode(index)
            route_load += data["demands"][node_index]
            plan_output += f" {node_index} Load({route_load}) -> "
            previous_index = index
            index = solution.Value(routing.NextVar(index))
            route_distance += routing.GetArcCostForVehicle(
                previous_index, index, vehicle_id )
        plan_output += f" {manager.IndexToNode(index)} Load({route_load})\n"
        plan_output += f"Distance of the route: {route_distance}m\n"
        plan_output += f"Load of the route: {route_load}\n"
        print(plan_output)
        total_distance += route_distance
        total_load += route_load
    print(f"Total distance of all routes: {total_distance}m")
    print(f"Total load of all routes: {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.time_limit.seconds = 30
    search_parameters.solution_limit = 100
    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)
    else:
        print("No Soln")
if __name__ == "__main__":
    main()
```

Since this is a Constrained Problem we will have to add Heuristics that limit the number of solutions checked and the time it takes for the solver to solve. We can even add Penalties if some location is missed by the Vehicles.

This week we also looked into 2 theoretical/learning concepts in brief which we look into a bit more detail this week while finishing and finalising our VRP with “Capacity” and “Time Window” Code after we programme the VRP with “Time Window” Constraint and find the best heuristics for solving such problems because like in the above CVRP code due to suboptimal heuristics we can find no routes like we did for VRP.

Topics we went through in brief:

1. Genetic Algorithms

   Genetic algorithms (GAs) are a problem-solving approach inspired by the natural process of evolution. They are often used to find approximate solutions to complex optimization and search problems. GAs mimic the process of natural selection, where better solutions are more likely to survive and reproduce, passing on their favourable traits to the next generation.

1. Initialization:
   I. Create a random population of potential solutions.
   II. Each solution represents a possible answer to the problem.
2. Selection:
   I. Evaluate the fitness of each solution.
   II. Fitness measures how well a solution solves the problem.
   III. Select solutions with higher fitness for the next generation.
3. Crossover (Recombination):
   I. Select pairs of parent solutions based on their fitness.
   II. Apply crossover operation to generate offspring solutions.
   III. Crossover operation mimics genetic recombination.
4. Mutation:
   I. Apply random changes to some offspring solutions.
   II. Mutation introduces diversity into the population.
   III. Diversity prevents premature convergence to a suboptimal solution.
5. Replacement:
   I. Use new offspring solutions to replace the old generation.
6. Termination:
   I. Stop the algorithm when a termination criterion is met.

Termination criteria could be:
   I. Fixed number of generations
   II. Satisfactory fitness level
   III. Other criteria

Genetic algorithms (GAs) rely on four crucial components for their operation:

1. Chromosome: A chromosome represents a potential solution to the problem being solved. It is composed of genes, each of which encodes a specific feature or characteristic of the solution.

2. Fitness Function: This function serves as a measure of how well a given solution addresses the problem at hand. The algorithm utilizes this function to assess and compare different solutions, guiding the selection process.

3. Crossover (Recombination) Operator: This operator mimics the biological process of genetic recombination, combining genetic information from two parent solutions to produce new offspring solutions.

4. Mutation Operator: This operator introduces random changes to existing solutions, enabling the algorithm to explore new areas of the search space. By introducing diversity, mutation prevents the algorithm from becoming trapped in suboptimal solutions.

2. Simulated Annealing

Simulated Annealing is a probabilistic optimization algorithm that draws inspiration from the annealing process in metallurgy.
Simulated Annealing is used to find approximate solutions to optimization problems by exploring the solution space and probabilistically accepting less optimal solutions in the hope of escaping local optima.

1. Initialization:
   - Start with an initial solution to the optimization problem.

2. Distance Annealing Schedule:
   - Define a distance schedule that gradually decreases over time.
   - The distance controls the probability of accepting a worse solution.

3. Neighbourhood Search:
   - Define a neighbourhood structure that allows you to explore nearby solutions.
   - Generate a neighbouring solution by perturbing the current solution.

4. Evaluation:
   - Evaluate the objective function for the current and neighbouring solutions.

5. Acceptance Probability:
   - Calculate the probability of accepting the neighbouring solution based on the current distance covered and the difference in objective function values.
   - If the neighbouring solution is better, accept it. If it's worse, accept it with a probability that decreases as the distance decreases.

6. Update Solution:
   - Update the current solution to the accepted neighbouring solution.

Repeat steps 3-6 until the total distance covered reaches a sufficiently low value or a stopping criterion is met.

Why Simulated Annealing is Useful for Optimization Problems:

1. Escape Local Optima:
   - Simulated Annealing has the ability to escape local optima by allowing the acceptance of worse solutions, especially in the early stages when the distance is high.

2. Exploration and Exploitation:
   - The algorithm balances exploration of the solution space (at longer distances) and exploitation of promising regions (at shorter distances).

3. Global Optimization:
   - Simulated Annealing is well-suited for global optimization problems where the solution space is complex and contains multiple local optima.

4. No Gradient Information Required:

- Unlike some optimization algorithms that rely on gradient information, Simulated Annealing can be applied to problems where gradients are not readily available or are expensive to compute.

5. Versatility:
   - Simulated Annealing can be adapted to a wide range of optimization problems, making it a versatile algorithm.

Despite its versatility, it's important to note that Simulated Annealing does not guarantee finding the global optimum, and the quality of solutions obtained may depend on the algorithm parameters and the specific characteristics of the optimization problem. The success of Simulated Annealing often relies on appropriate tuning of parameters, such as the cooling schedule and the neighbourhood structure.

Simulated Annealing operates by iteratively exploring solution spaces, allowing the acceptance of less optimal solutions based on a distance schedule that gradually decreases. This stochastic approach helps escape local optima, enabling the algorithm to find approximate solutions to complex optimization problems. Simulated Annealing is valuable for global optimization, particularly when gradient information is unavailable or expensive to compute. Its versatility and ability to balance exploration and exploitation make it applicable to a wide range of optimization challenges, although successful implementation often requires careful parameter tuning.

References:

1. https://developers.google.com/optimization/routing/vrp
2. Holland, J. H. (1992). Genetic Algorithms. Scientific American, 267(1), 66–73. http://www.jstor.org/stable/24939139
3. https://www.researchgate.net/profile/Franco-Busetti-2/publication/238690391_Simulated_annealing_overview/links/5c0e6c72299bf139c74de536/Simulated-annealing-overview.pdf
4. https://youtu.be/96-u9s6D16k?feature=shared
5. https://youtu.be/InVJWW_NzFY?feature=shared

Work for Next Week:

1. VRP with Time Window code and VRP with Capacity and Time Window Combined.
2. Plot the Routes derived on actual Google Maps.
3. Look a bit more into Simulated Annealing and Genetic Algorithms and how exactly they solve a real world optimization problem (no code).