Efficient delivery of food rations in post-disaster scenario

Agamdeep Singh, Mohit Mohandas

June 2021

Contents

1 The situation

1.1 Post disaster food shortage

In the event of a disaster, it is very common for people to loose access to Food resources; Be it due to homes and local markets being destroyed or due to a famine. In such a case, the local Government has a big task at hand - To **supply food rations to these stranded people.**To be able to do so efficiently, the planning committee has to overcome some computational challenges. To understand these problems, we first must put ourselves in the shoes of the planning committee, and for this we need a model.

1.2 The Model

We assume the following model:

- The planning committee has access to a big granary which we will refer to as the **hub**, where food is stored.
- There are a number of villages surrounding this hub where food must be delivered.
- Trucks are loaded with rations at the granary and sent on pre-defined routes of villages.
- Each truck can feed a certain maximum number of villages. A good approximation is 17 Still, 17 is only treated as a variable and can be changed.

  (Averaging over population sizes in Rural India to get average population size. Dividing standard truck volume with meal package volume to get meals per truck. Then dividing village population by meals per truck to get villages per truck. This made us arrive at the number 17.)

- Truck re-fuelling is not an issue.
- Trucks travel at an average speed of 60km/Hr.
- People can go without food for only a given number of days. They must be fed within this time period.

1.3 The Problem

Under this model, we can state the problem as follows:

> Determine the minimum number of trucks required, and the individual routes they must follow, such that each village is fed within the given time period and no truck has to visit more than 17 villages.

You might ask, **why do I want to minimise the number of trucks?** Suppose you get the minimum trucks required as 4 but you have 8 trucks. In that case, after sending the first 4, you can already send the next 4 when the first 4 are on their way back(or even near the end of their route). Such an approach is necessary when people need to be fed for a long duration and the number of trucks is limited.

2 Mathematical formulation

2.1 Variables

- No. of villages(excluding hub) = n
- villages are provided to us in the form of their x,y coordinates : (x, y) . with village $v_i = (x_i, y_i)$
- the Villages space: A set of villages = $V = v_1, v_2, ..., v_n$
- Route of a truck: a permutation of a subspace of the village space :$R = [v_i, v_j, ..., v_r]$
- time taken by a truck to complete route $R_i = d_i$
- Max no. of days/hours people can survive without food: T
- No. of villages visited by truck i: k_i
- Hub location $h = (x_0, y_0)$

2.2 Inferences and Constraints

- Maximum distance a truck is allowed to travel = $60T = D$. As the maximum distance is proportional to maximum time. Using any of them is equivalent to using the other within our constraints. We go with maximum distance per truck.
- $d_i < D$ for all i
- $k_i < 17$ for all i

2.3 Solution space

The solution space must consist of a function or algorithm F, that takes the variables $V, D, 17, h$ and returns a set of routes $\{R_i\}$ such that the no. of routes in minimal. The cardinality the set will give no. of routes and the individual the routes will give the path each truck must follow.

Hence:

$$A = \{tuples\ of\ the\ form\ (V, D, 17, h)\}$$

$$B = \{sets\ containing\ a\ list\ of\ routes : \{R_i, R_j, .., R_p\}\}$$

$$F : A \to B$$

$$F(V, D, 17, h) = \{R_i, R_j, .., R_p\}$$

such that:

- $|Ri, Rj, .., Rp|$ is minimum.
- for each $i, |R_i| < 17$.
- for each $i, d_i < D$

3 TSP

3.1 Travelling Salesman Problem

The travelling salesman problem is a well researched existing problem that asks and solves the following:

> Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?

So travelling salesman has similar elements to our problem. The difference being, travelling salesman is an optimisation problem without any constraints. Hence is needs to minimise route length. Further there is a single route. So essentially the solution is a permutation of the village space.

3.2 Cost-Constrained multiple travelling salesman Problem

Our Problem differs is different from TSP in the sense that:

- There are multiple 'salesman' or trucks that need to go around.
- There is a maximum length each truck can travel, no such constraint exists in the original TSP.

- There is a maximum no. of villages each truck can visit. In standard TSP, all villages are covered by the single 'salesman'.
- We want to minimise trucks used rather than distance travelled. Any solution within the constraints is acceptable.

We like to call this problem the ***Cost constrained Multiple travelling salesman problem***.

We kept Travelling salesman problem in the name because we are going to use TSP algorithms as a part of our bigger solution algorithm.

4 Minimum trucks per envoy

4.1 Algorithm

The program is written in python and uses standard python libraries. Building on these libraries we made *line-divider* and *recursive-divider* functions. We also used existing python implementations of solving the standard TSP problem(here we use it so that we don't have to return to starting city), this function is called *TSP*. A conjunction of these 3 function along with recursion gives us our algorithm.

***line-divider*(hub, village space)**:

- Draw line passing through hub making 0 degree with x axis.
- Put point in equation of line. Sign tells us on which side of line the point lies.
- Calculate no. of point on each side and take the difference. This is called the weight of the line.

  > *NOTE:* A lower weight represents that the line divides the space quite evenly in terms of no. of cities on each side. This is used as a heuristic method so that if trucks would go to either side of the line. They would have to do roughly equal work(on average).

- Rotate line by a few degrees(we used 5 but it is variable and can be changes if villages are densely positioned) and repeat same process to calculate weight.
- Select line with lowest weight as best line.
- **Return** 2 lists containing the points on each side of the best line.

Hence ***line-divider*** line divider divides the village space into 2 parts with roughly equal work. Return the list of points on each side of the dividing line(2 lists).

***TSP*(village space)**
Returns a permutation of the village space. This permutation is the optimal way to travel to all villages in the space successively.

***recursive-divider*(hub, village space, route = []):**
This is the main code which utilises the previous 2 functions. Takes input as village space and hub and a overall route list to which it appends the route an individual truck must take. outputs the overall route list.

***recursive-divider*(hub, village space, route = []):**

- space1, space2 = ***line-divider*(hub, village space)**
- **If** $|space1| < 17$
  - route1 = ***TSP*(space1)**
  - d = distance covered while traversing route1
  - **If** $d < D$:
    - route.append(route1) . [Add route1 to the list of routes]
  - **Else**:
    - ***recursive-divider*(hub, space1, route)**
- **Else**:
  - ***recursive-divider*(hub,space1, route)**
- Repeat for space2
- **Return** route

Hence, our approach is to keep dividing the village space into smaller subspaces till constraints are satisfied in each of the final reported subspaces. The no. of subspaces represents the minimum no. of trucks. This is a heuristic/meta-heuristic approach. Note that if one of 2 sub-spaces satisfies the constraints, only the one that does not must be divided, the one that satisfies remains as it is. To get a better visual understanding of the algorithm, please refer to this video.

You can see this algorithm in action here: Github repository for section 4

4.2 Implementation and examples

4.2.1 What the program outputs

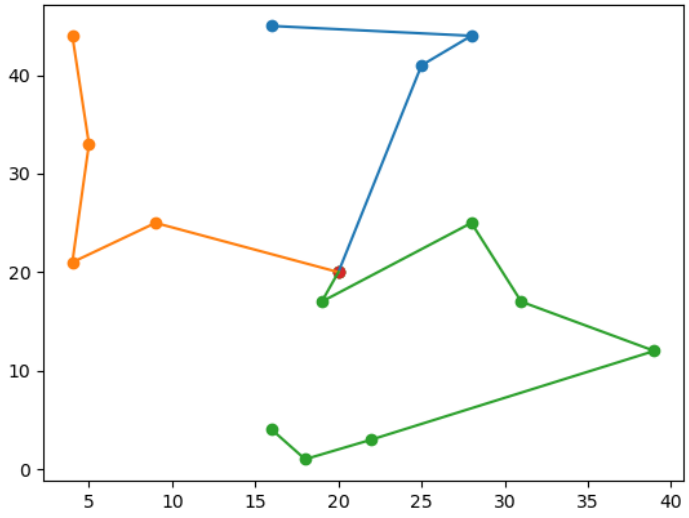
Solving for 15 cities.

Input: [(20, 20), (28, 25), (19, 17), (4, 44), (31, 17), (16, 4), (18, 1), (5, 33), (25, 41), (28, 44), (22, 3), (9, 25), (39, 12), (4, 21)]

Output: [[(20, 20), (25, 41), (28, 44), (16, 45)], [(20, 20), (9, 25), (4, 21), (5, 33), (4, 44)], [(20, 20), (19, 17), (28, 25), (31, 17), (39, 12), (22, 3), (18, 1), (16, 4)]]

Here we have assumed the first element of the input is the hub. The output consists of a lists of lists. Each list is to be followed in order to get a route. the no. of lists is the number of trucks required.

Plotting this solution gives us the following:

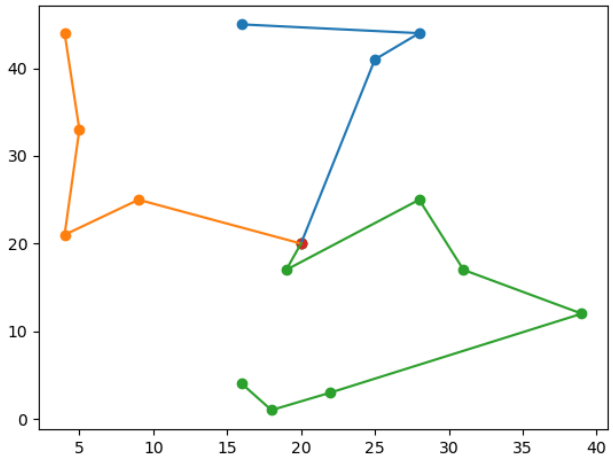


To get a better feel and understanding of how the algorithm operates, we present the following examples. In each example, we change one variable and keep the others constant and see how the solution changes.

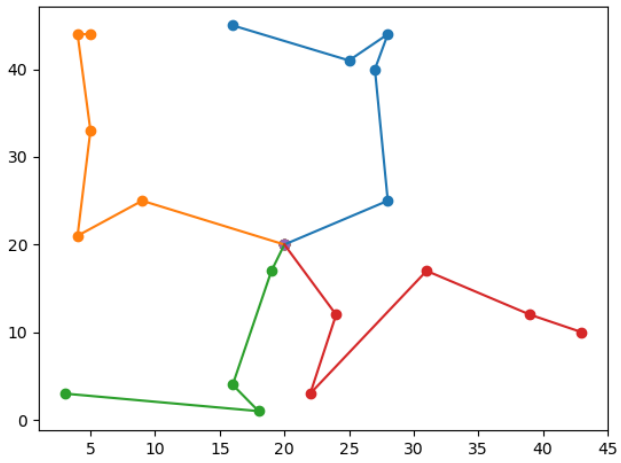
4.2.2 Increasing the number of cities

In the previous example, increasing the number of cities from 15 to 25 yields a different solution.

- 15 cities:



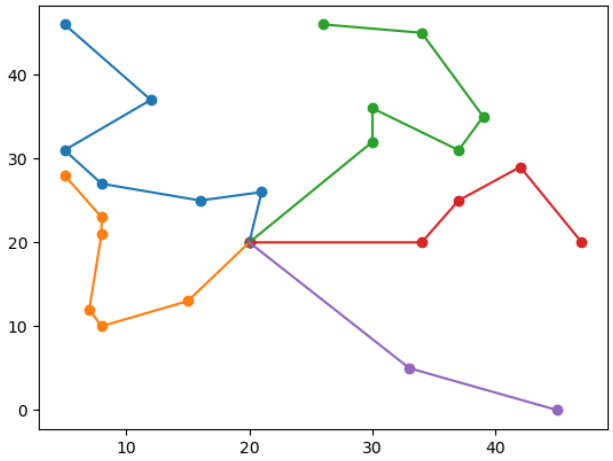
- 25 cities:



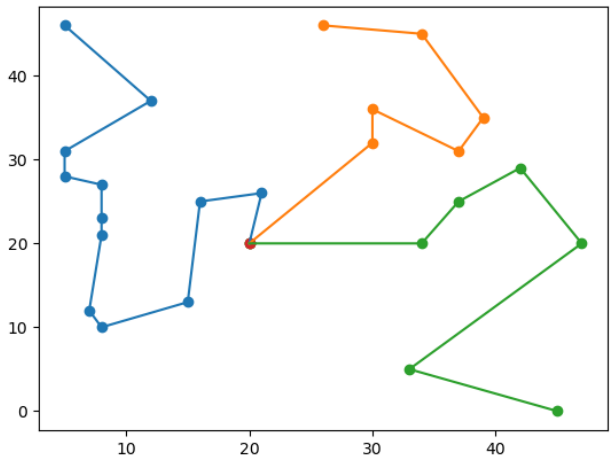
4.2.3 Increasing the maximum distance D

This is for a different input village space. Referring to the maximum distance a truck can travel. Obtained from the time constraint.

- D = 65



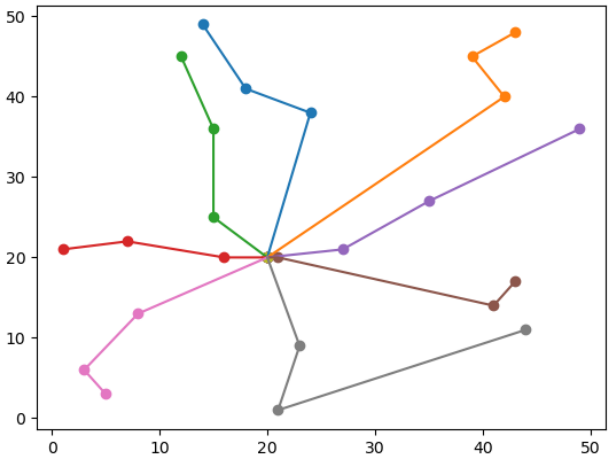
- D = 110



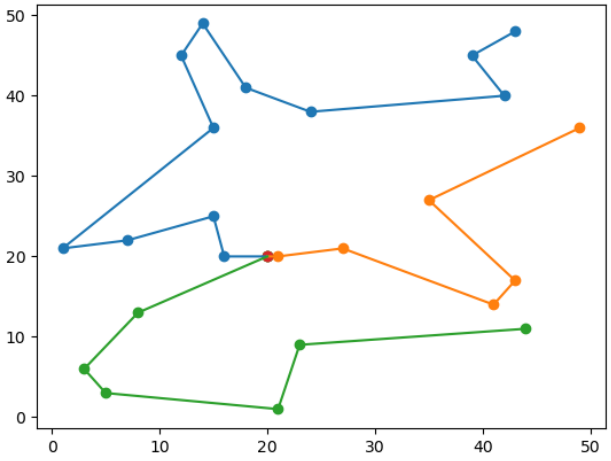
4.2.4 Increasing the maximum villages per truck

This is for a different input village space.

- maximum villages = 5

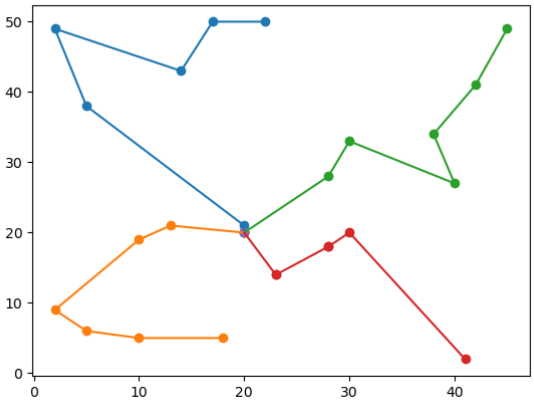


- maximum villages = 17

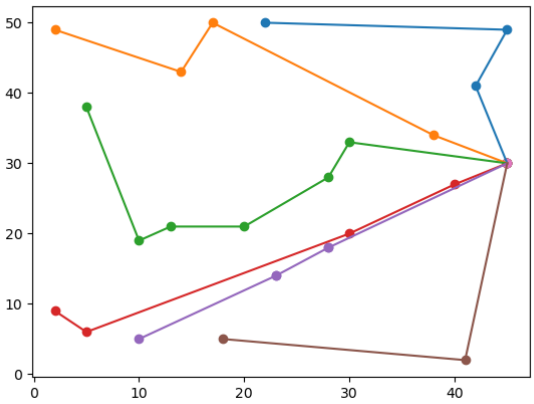


4.2.5 Changing Hub position for fixed village space

- hub = (20,20)



- hub = (45,30)



5 Looking ahead: What if I have even more trucks?

5.1 Extra trucks entails healthier people

Suppose we have a hugs surplus of trucks. In this case, minimising the number of trucks is not ideal as even though we may work under the constraints, having more trucks means more routes and shorter routes(by the pigeonhole principle).This means that people get their rations even faster as few villages in

the route leading up to a certain village have been accommodated in a different route, hence wait time for ever body decreases.
We have developed a solution for such a case too which works on similar principles as the last algorithm but here we tell the program how many trucks we have and it gives us the most efficient list of routes.

5.2 Algorithm

Since no. of trucks is not to be minimised, the only active constraint we need to follow is the distance one(so that people get fed properly); Keeping the no. of trucks greater than the minimum no. of trucks reported form the first algorithm take care of this.
In the following algorithm. the term 'sectors' is synonymous to 'truck' or 'trucks'. Each truck feeds one sector.

Essence: This algorithm focuses on dividing the entire set of places into several sectors, each of which can be individually solved by a TSP algorithm. To achieve this, we rely on the weight of each sector. If we can minimise the variance in this weight, we can distribute the workload more evenly among the trucks.

Classes:

- Hub: A class to represent the hub. Used to display the hub on the screen.
- Place: A class to represent each place. Contains information such as position of the place, weight of the place and the places neighbouring this place. Also used to display the place on the screen.
- Sector: A class to represent each sector. Contains a list of places in the sector.
- SortedQueue: A queue which can accept elements as key-value pairs and sorts them based on their value(increasing order).

Functions: note: 'place' is synonymous with 'village'.

- get_point_angle: Finds the angle made by each point/place with respect to the positive x axis. This angle is between – pi and pi.
- get_points_sorted_by_angle_from_positive_x_axis: Gets a sorted list of points/places based on the angle from get_point_angle.
- set_neighbouring_places_for_nodes: Sets the neighbouring places for each place using information from get_points_sorted_by_angle_from_positive_x_axis.
- move_place_between_sectors: Moves a place from one sector to another. Note that this can be done only for places on the boundry of each sector and for sectors right next to each other. If places were allowed to be

shifted with no such constraints, the effectiveness of the algorithm would increase, but the time complexity of the algorithm would increase as well.

- get_variance_of_sector_weights: Finds the variance in the weights of all sectors. Here variance refers to the statistical variance. ¡insert formula for variance here.¿
- get_current_sectors_weight: Returns the current sector weights as a list.
- generate_possible_place_switches_sorted_on_varance_of_sector_weights: Generates possible list of places which can be moved between sectors. As the list is generated, it is sorted on get_variance_of_sector_weights. Optimisation has been included to reduce the time complexity here.
- act_on_first_task_from_possible_place_switches: Performs the first action from generate_possible_place_switches_sorted_on_variance_of_sector_weights. This is the action which can reduce the variance of weights between sectors the most.
- get_distance_matrix_from_points: Gets the distance matrix from the list of points. This is used to solve each individual sector via the dynamic programming solution of TSP.
- solve_by_tsp: Used to run dynamic programming solution of TSP on a sector. Uses get_distance_matrix_from_points. Refer to python_tsp for implementation of the dynamic programming solution of TSP.
- solve_all_tsp: Calls solve_by_tsp for each sector and gets the solution, which it then displays on the screen.

Process:

1. Input the places positions (as a list) and number of sectors 's'.
2. Initialise the Hub, providing the first entry of places positions as the hub position.
3. Initialise the places via the Place class, providing the remaining entries of places positions as the place positions.
4. Run set_neighbouring_places_for_nodes for each place.
5. Divide the entire set of places into 's' sectors.
6. Initialise the sectors via the Sector class, providing the first and last place in the sector(order based on angle). The Sector class then finds out all the places it covers by checking for neighbouring classes of the first sector and repeating the process for subsequent neighbours.
7. Make a SortedQueue named possible_place_switches.
8. Empty possible_place_switches.

9. Run get_current_sectors_weight and use its return value to get weight variance from get_variance_of_sector_weights. Store this result with the key None in possible_place_switches.

10. Add the result of generate_possible_place_switches_sorted_on_varance_of_sector_weights to possible_place_switches.

11. If the first action to be performed is not None(from possible_place_switches), this means that there is a place which can be transfered from one sector to another to reduce the varaiance of sector weights.

    (a) Run act_on_first_task_from_possible_place_switches providing possible_place_switches as an argument.

    (b) Goto step 8.

12. If the first action to be performed is None(from possible_place_switches), this means that there is no place which can be transfered from one sector to another to reduce the varaiance of sector weights. Hence we have found a minima for the variance in weights.

13. Run solve_all_tsp for the current sectors.

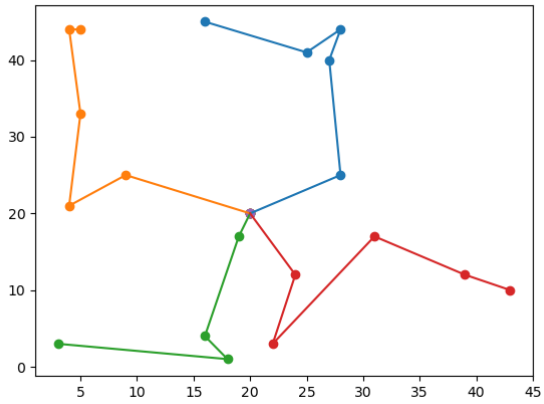
You can see this algorithm in action here: Github repository for section 5

5.3 Implementation and examples

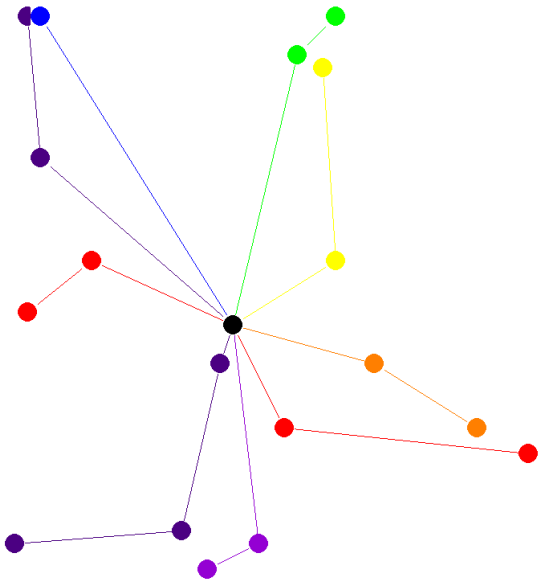
The following are examples of what happens when we increase the no. of trucks in the examples discussed in **Section 4.2**.

5.3.1 Increasing trucks in 4.2.2

- Initial(minimum trucks):

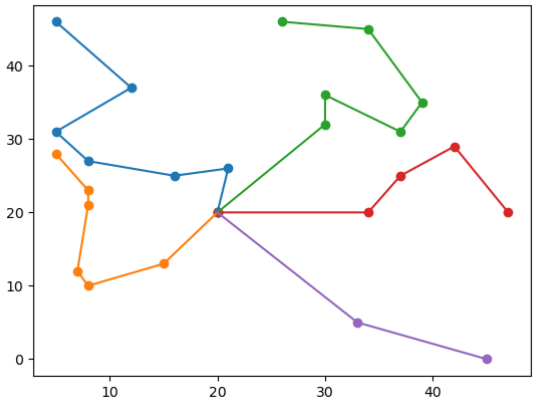


- Increasing trucks to 7:

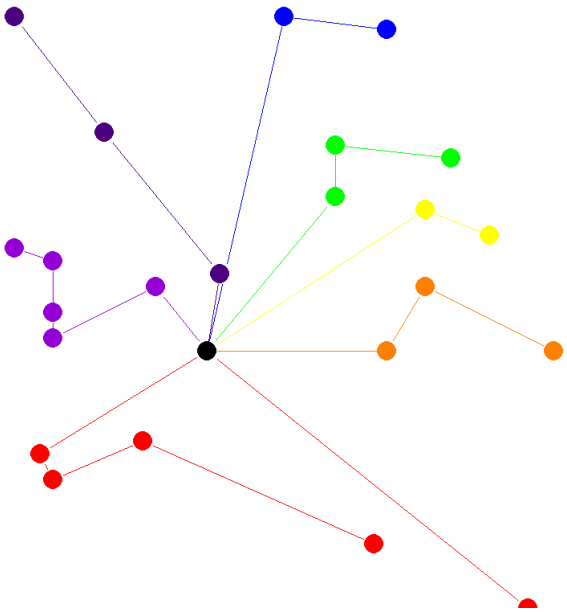


5.3.2 Increasing trucks in 4.2.3

- Initial(minimum trucks):

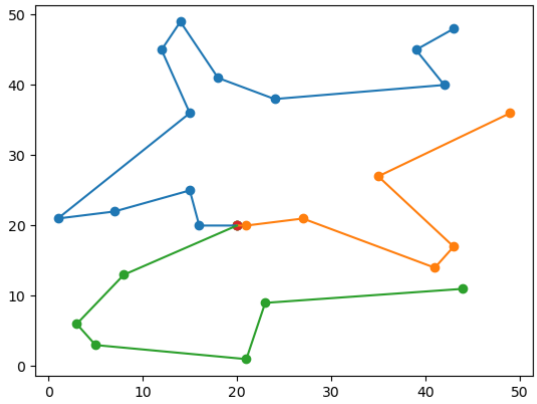


- Increasing trucks to 8:

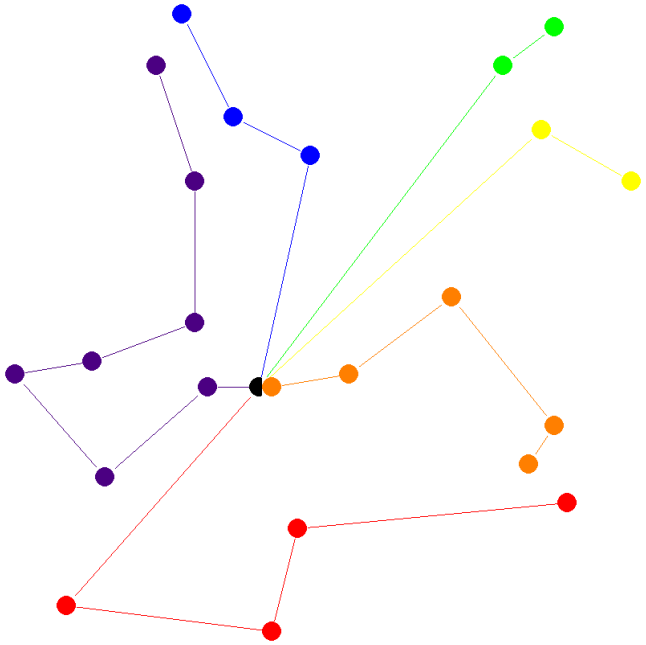


5.3.3 Increasing trucks in 4.2.4

- Initial(minimum trucks):



- Increasing trucks to 6:



6 Acknowledgments

We would like to acknowledge **Priyanshu Rai** and **Shashwat Sourav** for their valuable contribution to the project. They were a part of the ideation phase, but due to certain limitations could not participate further. Their contribution remains important.

The python implementation of standard TSP we used: https://pypi.org/project/python-tsp/