Lab 3 – CVRP:

Submitters: Inbar Lev Tov 316327246, Gal Gibly 315093435

Note that the assignment was done with python version 3.11.0

Introduction:

We treated the CVRP problem as if it was a real life situation:

- We have a universe that contains a specific **number of cities**.
- Each city has a certain amount of people.
- Which means – in terms of resources – each city **demands** a different quantity of food.
- The chain of supply relies only on **one warehouse – city 0**.
- That warehouse dispatches vehicles that can carry a certain amount of food each time (**capacity**).
- The **cost** for the warehouse depends on the journey each vehicle takes, as does the number of vehicles dispatched in the first place.

The question remains: how do we maintain both a good chain of supply (doesn't create a shortage and truly provides each city with what it needs), but also an effective one (lowest cost possible)…?

Representation:

We start with reading the data concerning each problem and, once we understand what we have in terms of parameters, represent the entire problem in a way that would help us achieve an optimal solution.

The text files presented to us contained:

- The number of cities we have to deal with.
- Capacity for each vehicle.
- 2D coordination of each city.
- Demand for each city.

Our representation of the problem:

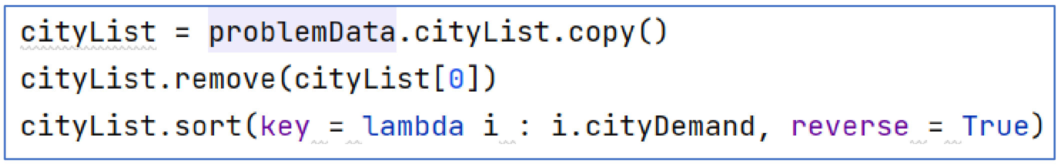
- We created an entity for each city – even city 0, which is considered to be the warehouse from which the vehicles go on their journey and then converge back to.
- Each city contain:
  - Serial number – functions as the identity of the city, later to be recognized as part of a specific vehicle journey.
  - X and Y coordination – their combination functions as the location of the city in our "universe".
  - Demand – how much does the city needs, in terms of resources.
- Aside from that, we also created a way to look at the problem globally – an entity that would function as the whole universe:
  - We stored all cities in a single list, and registered the number of cities and vehicle capacity as global parameters.
  - Since we now have all the cities stored, we can calculate the distance matrix for the entire universe.
  - Each entry in that matrix is the Euclidian distance between every 2 cities.
    $$distance = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$
  - That matrix would later help us calculate the cost for each journey the vehicles take.
  - We declared our warehouse to be city 0, as defined by the file.
  - Our last parameters to be set is the number of vehicles used overall (0), as does the possible cost (0 as well).
  - Those parameters would change as our solutions progresses, and would reach optimum at the end.

Multi Stage Heuristic:

We treat this multi stage heuristic as a way to get our base optimal solution – what's the worst way of organizing the cities, so that our solution would cost too much money and we would dispatch too many vehicles.

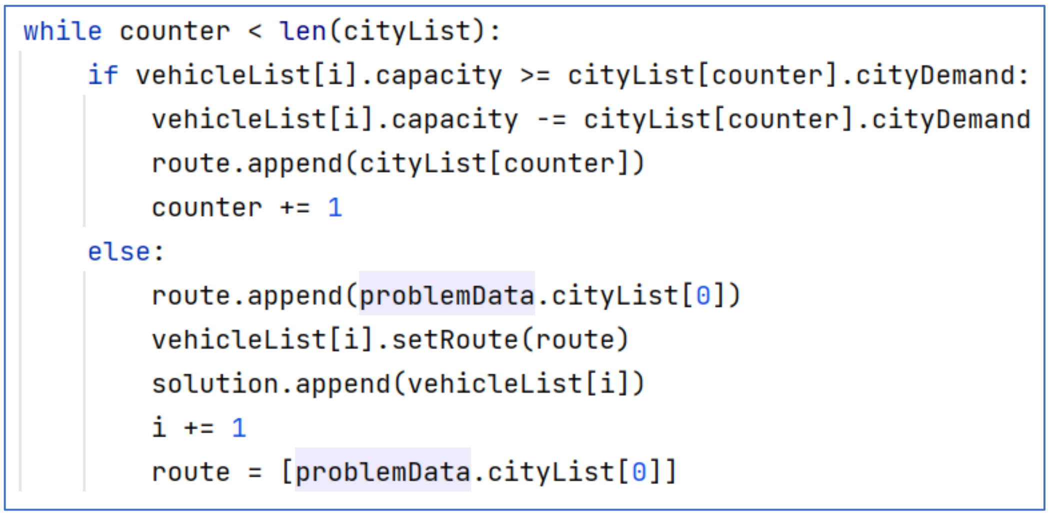
After careful examination of the data, we chose to go with maximum demand:

- Establishing the neighborhood:
  - We sort the cities based on demand (biggest to lowest) – complexity of $O(n \cdot \log(n))$.
  - We also remove city 0 because it has the lowest demand – 0.



```python
cityList = problemData.cityList.copy()
cityList.remove(cityList[0])
cityList.sort(key = lambda i : i.cityDemand, reverse = True)
```

- Assembling our fleet of vehicles:
  - We go for the worst case – a vehicle for each city.
  - We create that amount in advance and then only use some of them.
- Dispatch:
  - We dispatch a vehicle each time, and whenever it can't supply any more of the demands, it gets back to the warehouse.
  - We can't go to the same city twice – if a vehicle doesn't contain enough supplies for the city, it gives up and another vehicle takes its place.



```python
while counter < len(cityList):
    if vehicleList[i].capacity >= cityList[counter].cityDemand:
        vehicleList[i].capacity -= cityList[counter].cityDemand
        route.append(cityList[counter])
        counter += 1
    else:
        route.append(problemData.cityList[0])
        vehicleList[i].setRoute(route)
        solution.append(vehicleList[i])
        i += 1
        route = [problemData.cityList[0]]
```

- Since we already have a sorted list, we only go about the list of cities in only one way – forward. Complexity of $O(n)$.
- Final complexity would be $O(n \cdot \log(n))$, as the sorting takes most of the effort.

We would later see that for most CVRP problems, going to the most demanding city each time produces us much more expansive results, as we are bound to lose a lot more resources much faster every time. That would also cost us not just in terms of distance, but also in terms of vehicles – dispatching way more vehicles than the optimal solution is a waste.

The good thing is – now that we have the worst possible bound – from here, we can only improve.

Meta Heuristics:

As you're about to see in the next few pages, we implemented iterated local search (ILS) for all of them – each time using a different heuristic – a different approach – in order to try and solve the CVRP problem.

For each of those problems we also performed a safety check using the Ackley function:

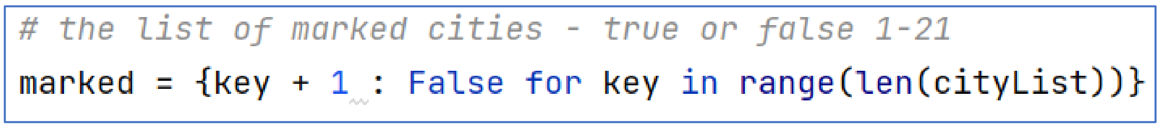
- Much less cites than any of the other problems presented to us.
- Much smaller search space – less options in terms of exploration.

Once we received a logical solution to that function, we had more confidence our approach was correct.

Tabu Search:

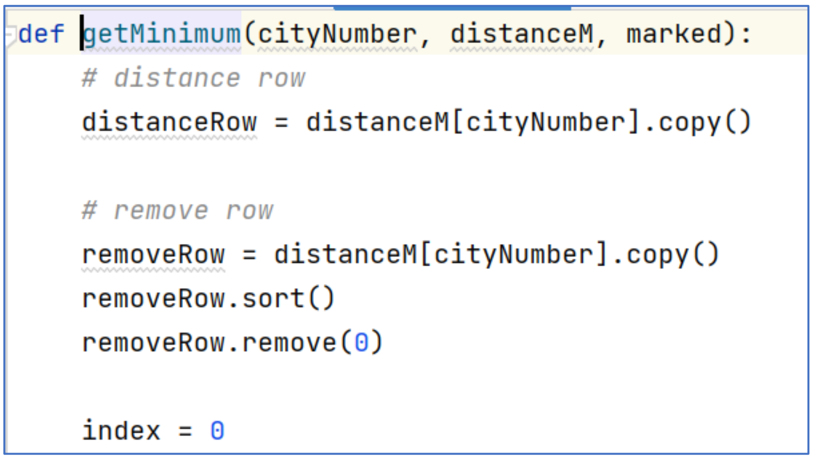
For the first meta heuristic, we used tabu search. Unlike with the multi stage heuristic, here the establishment of the neighborhood is done locally – and therefore only relies on the best neighbor you have "right now":

- We initialize the list of marked cities – cities that have already been visited and supplied upon.

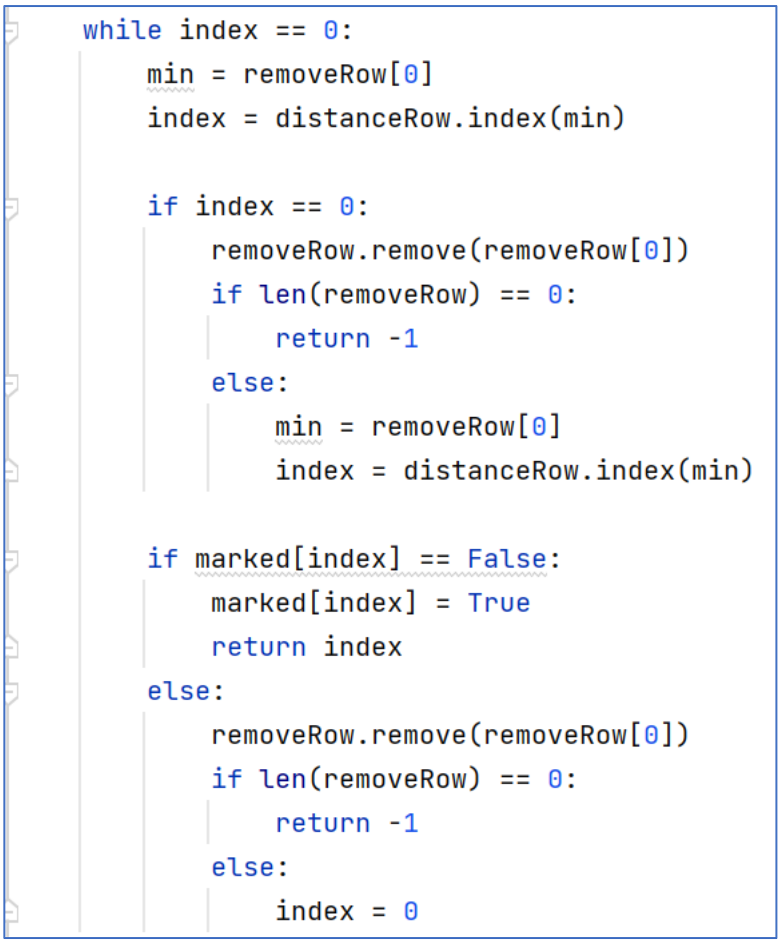


```python
# the list of marked cities - true or false 1-21
marked = {key + 1 : False for key in range(len(cityList))}
```

- For the first city, as does every other city afterward, we call the get minimum function that would find us both the nearest city and the newest city (hasn't been visited upon).



```python
def getMinimum(cityNumber, distanceM, marked):
    # distance row
    distanceRow = distanceM[cityNumber].copy()

    # remove row
    removeRow = distanceM[cityNumber].copy()
    removeRow.sort()
    removeRow.remove(0)

    index = 0
```



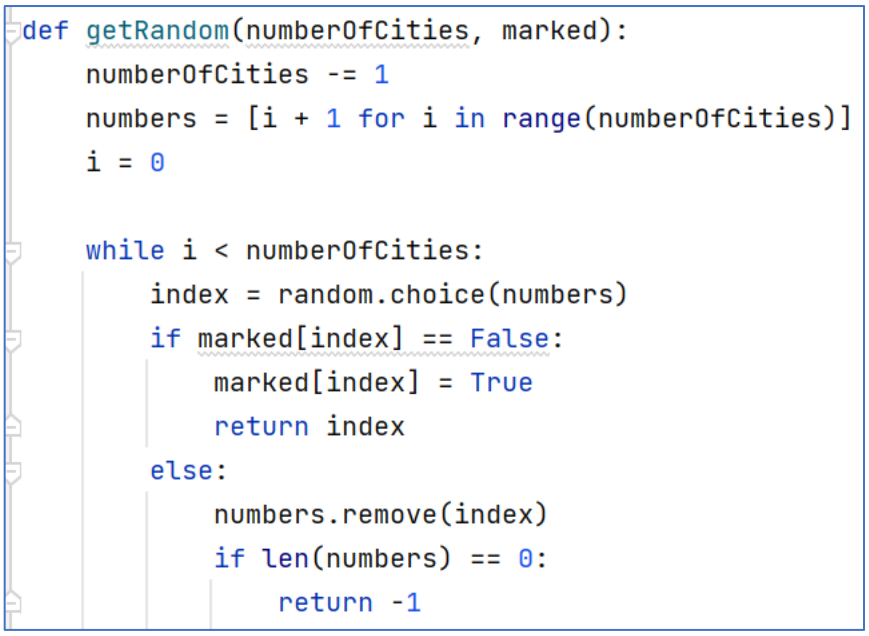
```python
    while index == 0:
        min = removeRow[0]
        index = distanceRow.index(min)

        if index == 0:
            removeRow.remove(removeRow[0])
            if len(removeRow) == 0:
                return -1
            else:
                min = removeRow[0]
                index = distanceRow.index(min)

        if marked[index] == False:
            marked[index] = True
            return index
        else:
            removeRow.remove(removeRow[0])
            if len(removeRow) == 0:
                return -1
            else:
                index = 0
```

- Also notice that the function takes into consideration a case where we accidently land on city 0 (warehouse) too soon – we don't want to shorten our journey too early.
- That's why we emphasize the idea of "best right now" – we need a city that isn't 0, hasn't been visited upon and is also the closest to us in terms of local distance.
- After we receive the best city right now, we also check that we can supply that city in the same way as was done in the maximum demand multi stage heuristic.
- If we can't supply it, we remark the city as false (unvisited), and finish the journey for that specific vehicle.
- It's also important to mention that, while finding the best neighbor right now helps us locally, it might hurt us in a global perspective – more exploration in sacrifice of optimum.
- Tabu search does face that problem, and we would later see how it affects its results in comparisons to all the other meta heuristics.
- Another problem we might mention is – it's expansive to recheck all the visited cities each time. That's why we really try to limit that check to only the possible closest city.

<u>Ant Colony:</u>

In the second meta heuristic, the vehicles are considered as "ants".

- As in nature, ants are unpredictable. There is no certain order to them, no contingency plan in place in case of an emergency.
- That is why, for the entire duration of the algorithm, the choice of the next city to be visited is entirely random.
- In the first round, we send the ants out there into the world in order to find cities that they like.
- The first one that get's back to headquarter – that city is chosen.
- Each round after that the same "mass exploration" happens – but if an ant gets back to headquarters with a city that has already been visited – we of course, as per the rules of CVRP, don't visit that city again, as it had already been supplied.



```python
def getRandom(numberOfCities, marked):
    numberOfCities -= 1
    numbers = [i + 1 for i in range(numberOfCities)]
    i = 0

    while i < numberOfCities:
        index = random.choice(numbers)
        if marked[index] == False:
            marked[index] = True
            return index
        else:
            numbers.remove(index)
            if len(numbers) == 0:
                return -1
```

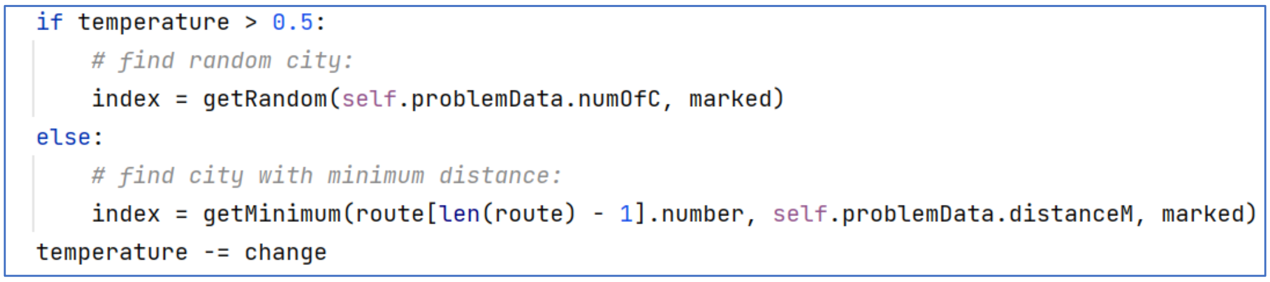
- In terms of probability to be chosen, that probability changes each time – because it very much depends upon the speed of each ant, as does it's ability to supply all the cities it has chosen.
- If an ant if fast enough but supply very large demand each time, it doesn't really matter how fast she is – her capabilities are limited.
- If an ant is very slow, most of what she has chosen has already been supplied, and so she has to make longer journeys, searching for the last "free" cities that remain.

Simulated Annealing:

Simulated annealing is the perfect combination of exploitation and exploration. It tries to balance between random choice (exploration) and the best choice (exploitation).

Our way of balancing between the two is temperature:

- In the beginning, the temperature is very high – allow much exploration as possible, because we would have time to fix it later.
- As long as the temperature is high, we choose cities based on the random algorithm.
- Once the temperature gets cold (near the end of the search), we have to try converging to the best minimal solution as possible (exploitation).
- That is why, when the temperature is low, we choose cities based on the nearest city algorithm



```
if temperature > 0.5:
    # find random city:
    index = getRandom(self.problemData.numOfC, marked)
else:
    # find city with minimum distance:
    index = getMinimum(route[len(route) - 1].number, self.problemData.distanceM, marked)
temperature -= change
```

- We of course update the temperature after every choice – so that the chance would be gradual rather than extreme.
- Later in the comparisons section, we would see how the number of cities to be chosen matters very much to simulated annealing – if we have more time to correct course if necessary, the results would be much better.
- Another thing we should mention is – what happens when the temperature is high, and the choice of city is random, and the chosen city hurts us immediately (very easy to identify, as we calculated cost after each addition)…?
- We allowed the city to be accepted under high temperature **only**, because we wanted to see a wide range of results.

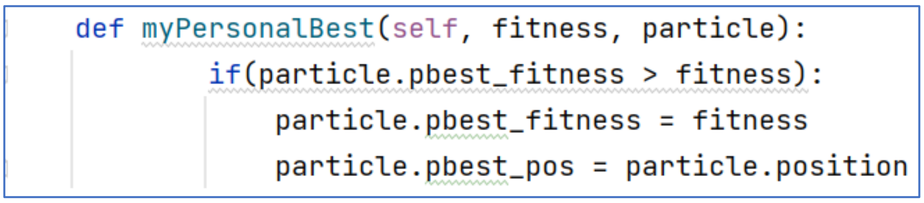
Islands:

- The island model here functions the same as with assignment 2 – the only change was in representation. If the early island model was in terms of x and y, now the gene holds an ordered list of cities to be visited.
- Since we focus on the same cities in each gene, we go back to handling permutations – and that has been explained in details in both assignment 1 and 2.
- Number of possibilities = $(number\ of\ cities)!$ Because we only visit every city once.
- Suffice to say, we allow
  - The maximum number of genes (1000).
  - The maximum number of generations (1000), to have allowed more time to reach solution as does full exploration.
  - SUS parent selection – as it has proven in the past to be most effective.
  - PMX crossover.
  - Scramble mutation on 100% scale – functions as completely random choice in every level.
- Each CVRP problem presents with a different solution in terms of time of convergence to solution – as we must take into consideration factors such as large distances between cities, as does maximum demand.

Cooperative PSO:

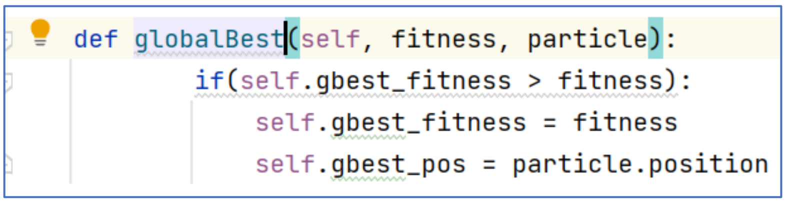
For the last meta heuristic, we play a little bit with the relationship between local "local" search, overseed by the cognitive element, and local "global" search, overseed by the social element:

- The particles want exploration – and they get it.
- The cognitive element remembers where was the best solution so far for himself.
- The social element consider all of the particles together and remembers the best solution overall.
- Its basically dealing with a wide range of random solutions that we are trying to improve, always looking both locally and globally.
- Estimating my personal best:



```
def myPersonalBest(self, fitness, particle):
        if(particle.pbest_fitness > fitness):
            particle.pbest_fitness = fitness
            particle.pbest_pos = particle.position
```

- Estimating global best:

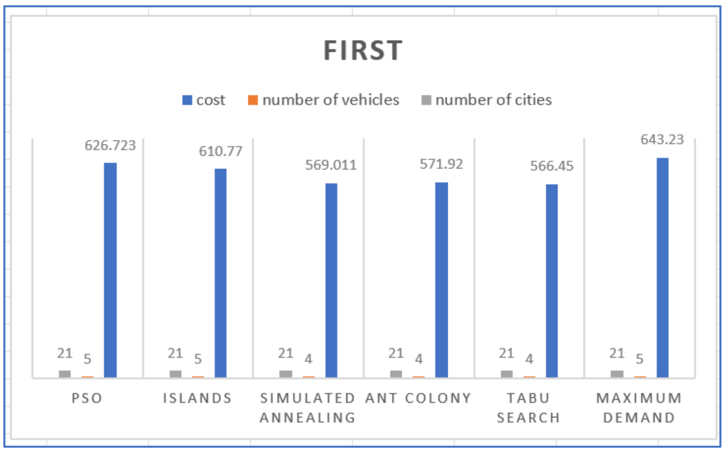


```
def globalBest(self, fitness, particle):
        if(self.gbest_fitness > fitness):
            self.gbest_fitness = fitness
            self.gbest_pos = particle.position
```

Comparisons:

In order to create a truly informative comparison, we checked the algorithms on the 3 sets of CVRP problems that had an optimal solution on the website.

Here is the graph for the first problem:

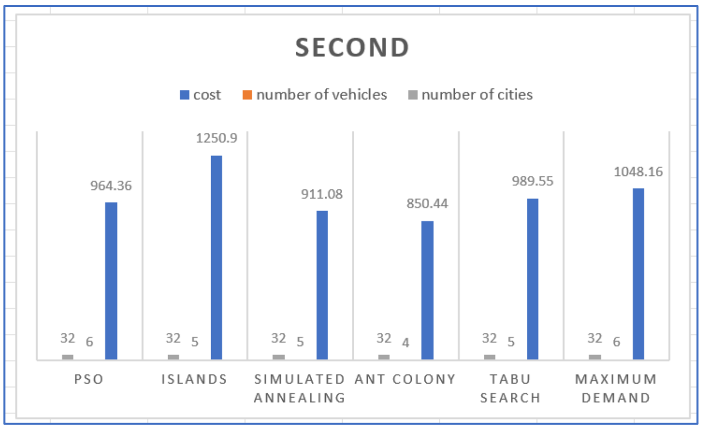


- The number of cities was always 21 because we checked all the algorithms on the same CVRP problem.
- As for the number of vehicles needed to be dispatched, please notice that some of them needed 5 vehicles, while others needed 4.
- Since the optimal solution consists of 4 vehicles and minimal cost of 375, you can automatically dismiss 3 of them.
- Also notice that they have extremely high cost – there is correlation between higher cost and the number of vehicles needed.
- Now lets look at the 3 remaining algorithms: tabu, simulated annealing and ant colony.
- Tabu obviously wins, but the cost differences between him and the other 2 are minimal at best.
- Since ant colony is random and therefore completely unreliable, we can only assume that was one lucky run, and not a common phenomenon.

- The real comparison would be between tabu ad simulated annealing – and since of them came out really close, we can just assume that the random part in simulated was very close matched to the best neighbor part of tabu search.

Tabu wins!

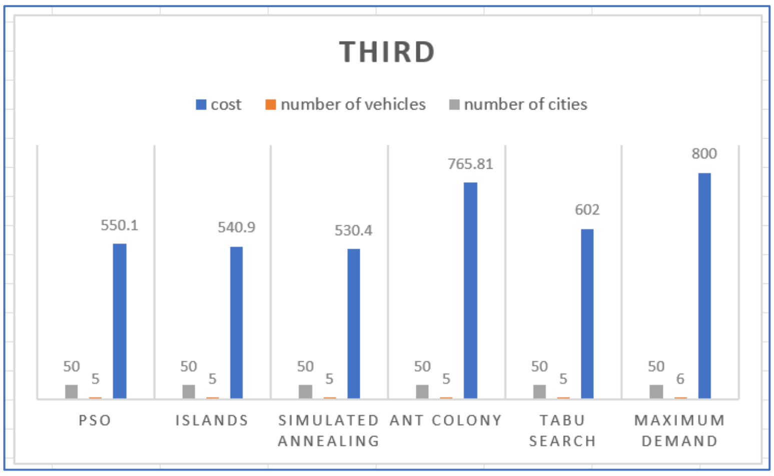
Here is the graph for the second problem:



- The number of cities was always 32 because we checked all the algorithms on the same CVRP problem.
- As you can clearly see, ant colony wins. But! Ant colony is so random that rather than looking only at the winner, we would look at all the other algorithms too.
- If we compare PSO to islands, for example, we would see that islands may have less vehicles than PSO, but the order of cities within each vehicle creates a state where even with more vehicles, PSO is more effective.
- It shows you that the inner journey of each vehicle on a "local" perspective of the vehicle itself is much more important than the number of vehicles dispatched on a "global" perspective.
- Another interested comparison is between simulated annealing and tabu search.
- While both had the same number of vehicles dispatched, simulated annealing had much better results than tabu.
- That is because, while tabu only searches on a local level, simulated annealing has the temperature factor that helps him control whether or not to receive a city that does look good now but can hurt him globally.
- In comparison with tabu, simulated is much more intolerant towards "locally good, globally bad" cities – especially towards the end of the algorithm.
- It gives us much more of a chance to self correct ourselves, and therefore simulated has better results.

Ant colony wins.

Here is the graph for the third problem:



- Unlike the first two examples, where ant colony was relatively close or even the best solution out of the 5 – here its unpredictable nature is shown by being one of the highest costs.
- All 5 meta heuristics has the same number of vehicles (which is the optimal one), and so the only thing that truly differentiates one from the other is the inner order of cities for each vehicle.
- Simulated annealing is the best, then islands, then PSO.
- All of them are relatively close in terms of costs – which means it’s a matter of 1 or 2 cities that are different for each of them.
- Here we see a clear example of simulated annealing self correcting it's course towards the end – it finally had enough time to change the outcome to its advantage.