Delivery Scheduling

Final Delivery

Problem Definition

- This optimization problem focuses on delivering packages using a cost efficient route. Three package types (fragile, normal, urgent) may be delivered from a starting point to different locations, until all packages have been dropped at their specified destiny location.

- The goal is to minimize the overall cost by : (1) Minimizing the total travelling distance, (2) Delivering the Urgent packages under the expected time and (3) Increasing the odds of delivering a fragile package without being damaged.

References

- https://www.geeksforgeeks.org/traveling-salesman-problem-using-genetic-algorithm/
- https://youtu.be/oSdPmxRCWws?si=xYNA3R9c1Iv0KT9o
- https://youtu.be/cY4HiiFHO1o?si=BtmcsZVgLRhdXiOB
- https://en.wikipedia.org/wiki/Crossover_(genetic_algorithm)

Formulation of the problem

Solution representation: sequence of nodes, numbers from 1-n, n being the total number of nodes to be visited. There should also be a value representing the percentage of packages delivered on time, and/or not broken.

Neighborhood(mutation) function: Since our solutions is represented as a set of numbered nodes, a mutation function would basically be a swap between two nodes of a solution, and that basically generates another solution, which may be better or worse. This is also known as a 2-opt optimization technique.

Crossover function: Given two permutations of the set of nodes, calling them both parents, P0 and P1, then randomly selecting gene segments of the P0, which means that the child permutation will contain the selected genes of P0 in the same position, and then the remaining genes will be transferred in the order which they appear in P1.

Hard Constraints :

- You only have one vehicle available.
- The delivery locations are specified by their coordinates.
- Routes between all delivery coordinates are available.
- The driver drives at 60km per hour and takes 0 seconds to deliver the goods.
- The cost per km is C=0.3;

Evaluation function : To evaluate the quality of the different obtained solutions we will use the following formula:

Total Cost = Total distance traveled in KM * C + Total delay minutes * C + (∑Z)*C;
Z being the random breaking cost of the different packages.

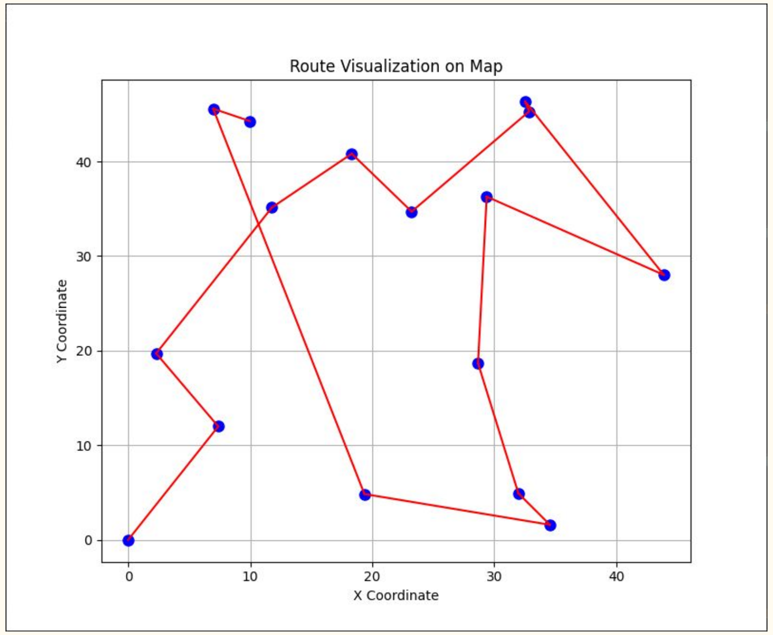
Implementation

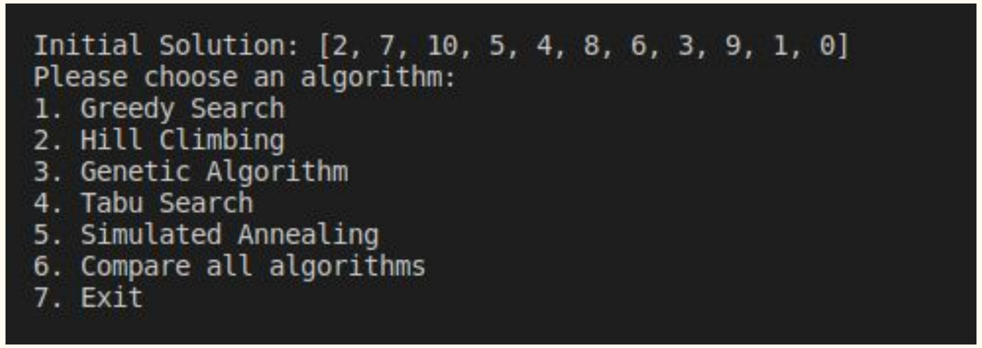
To solve the problem at our hands we defined the following functions (the names of the functions remain the names of the algorithms used):

- evaluation_function (we use this function to calculate the cost of a solution);
- greedy_search (it chooses the next node to visit if the sum for the current path + next_node is the lowest)
- hill_Climbing (it looks for a better solution in the neighbourhood until it reaches the local maximum itself)
- genetic_algorithm (we implemented two versions of it, one using order-crossover and other using partially-mapped-crossover)
- tabu_search (similar to hill_Climbing but it records past nodes that were visited)
- simulated_annealing (try to escape local maximums using temperature and a cooling rate)

Approach

In our application the user starts from deciding the number of packages he wants to deliver, after that he chooses the size of the map and whether he wants to see a individual result of an algorithm or if he wants to see them all in comparison.





```
Initial Solution: [2, 7, 10, 5, 4, 8, 6, 3, 9, 1, 0]
Please choose an algorithm:
1. Greedy Search
2. Hill Climbing
3. Genetic Algorithm
4. Tabu Search
5. Simulated Annealing
6. Compare all algorithms
7. Exit
```

If the user selects option number 6, seeing all the results he can pick the best one (the one with the minimum cost) and visualize the route.

Greedy Search and Hill_Climbing

For the greedy algorithm, as base we use our evaluation function, which has in account the total distance traveled, the number of packages that were broken when delivered and the sum of “time_delivered-time_expected” of all the packages delivered out of time. So our solution iteratively picks the next node to add on the list of nodes visited, based on a prediction of what will be the cost when that next node is added.

The Hill Climbing algorithm starts with a random solution, and using the solution, it uses a function “generate_neighbours” to generate 10 neighbours for the solution, in that neighbour list, it compares its own result of evaluation function, compared to all neighbours and if a neighbour is better than the current solution, it becomes the new current solution. It follows this for a number of iterations chosen by the user, and if it can’t find a better solution within an iteration, it stops, meaning it has reached the local maxima

We have also shown the results of running these two different algorithms, but because of the limited space, we chose to place all the results in an attachments file, which can be seen by the teacher if necessary.

Genetic Algorithm

For the Genetic Algorithm we created a population of random solutions, then we take two parents from that list and used them to make a child using a crossover function, we implemented two crossover heuristics:

The order crossover (OX), in this function, the child will take the first half of nodes of the first parent and for the other half it will take the remaining nodes but in order they are in the second parent.

The partially mapped crossover (PMX), in this function, it will be selected from the first parent two crossover points forming a gene segment that will be copied to the child in the same position, then it will look for nodes that have not been copied in the correspondent segment of the second parent and for each node found (n) lookup which node (m) was copied in its place from the first parent to the child and copy n to the position that is held by m in the second parent to the child, if the place is already taken by an element k, copy n to the place taken by k in the second parent to the child. For the remaining positions fill the child with the nodes from the second parent that have not been yet copied, in order of appearance.

We have also shown the results of running these two different heuristics, but because of the limited space, we chose to place all the results in an attachments file, which can be seen by the teacher if necessary.

Tabu Search and Simulated Annealing

Tabu Search is very similar to hil_climbing, but it uses memory, which in our implementation the size of the memory(represented as a list) can be chosen by the user. Because tabu search uses memory, it can avoid revisiting solutions previously exploited, and because of that it can escape local maxima.

Another algorithm that we used that allows escaping local maximas is simulated annealing. It is a metaphor of the annealing process used in metallurgy that uses temperature to facilitate the manipulation of the metallic objects. For this algorithm, we use an initial temperature, which represents the probability of accepting a worse solution to escape local maximas, as the temperature decreases, the probability of accepting worse solutions also decreases. Because it escapes to worse solutions while the temperature decreases it may not be at the global maxima in the end, as it may get stuck when the temperature gets low, when it was in a solution that maybe wasn’t the optimal.

Conclusion

With this project, we were able to see the algorithms mentioned in theory classes more in detail, and see in fact the difference in the results provided by the different algorithms when starting with the same solution (except the genetic
ones, which generates children based on two random solution parents). Some of the conclusion we took were:

- For the hill_climbing algorithm, a larger number of iterations, in general won't influence the result very much, because when the local maxima is reached it stops.
- For the genetic algorithm, having a larger number of iterations slightly decreases the cost, as we predicted, because it keeps creating fitter and fitter generations as it goes. For this algorithm the most differentiating factor was the size of each generation, this is, the size of each set of solutions procreated by two parents. The comparison between the crossover heuristics was almost null because both of them depend on random crossovers.
- For simmulated_annealing, in general, a high temperature and small cooling rate is generally good, as it increases the exploration space, but logically, the higher the temperature, and the lower the cooling rate, the higher the time to give the final solution will be.
- And finally, for tabu search, the size of the tabu list slightly increases the performance time as it increases, and slightly reduces the cost. But, pay attention, having a tabu size way larger than the number of nodes, won't serve you better than just having it at the number of nodes.(you don't need to initialize more memory than you will need).

And this was our project, all our performance measures can be found in the "attachments file".

Thank you!