ALGAV

3DE

1220514

1220528

1220379

1220459

1220448

Contents

[**Domain knowledge representation** 3](#_Toc121090074)

[**Optimal solution for Deliveries Planning with an electrical truck** 3](#_Toc121090075)

[**Study of the problem complexity as well as the viability of finding the optimal solution by generating all the solutions** 7](#_Toc121090076)

[**Heuristics for the quick generation of solutions** 8](#_Toc121090077)

[**Analysis of the heuristics quality** 9](#_Toc121090078)

[**Conclusion** 9](#_Toc121090079)

# **Domain knowledge representation**

For this Project we needed an algorithm that finds the best path based on a list of warehouses. This should calculate the cost of every single delivery from a certain warehouse to another one. Based on the sum of these costs, we should be able to find the best path to do all those deliveries.

There are a few things we had to consider:

* Before doing another delivery, we have to check if we are able to get to the next warehouse with our Truck (we need to have at least 20, and also more if the next warehouse is further away).
* We should add this charging time to the total time (aka the total cost).
* We have to keep track of the warehouses where we charge the truck.
* We should also consider the unload time.

For this we need some data (or facts):

* TruckData(Truck name, Departure warehouse, Destination warehouse, Time(minutes), Energy(kWh), Additional time(minutes))
* main\_warehouse(5).
* Delivery(Id, Date, Weight, DestinationWarhehouse, TimeToLoad, TimeToRetrieve)
* TruckFeatures(Name, weight(kg), max\_load(kg), full\_charge(kWh), CurrentBattery(km), time\_to\_charge(from 20% to 80% in minutes)

For this sprint we don’t have to take into account: truck name, date, and timeToLoad.

# **Optimal solution for Deliveries Planning with an electrical truck**

Text

Description automatically generated

First, we have some code to retrieve a list of all the delivery loads, from high to low. This list represents the load that the truck is always carrying, and will always become lower because the truck is unloading at every warehouse stop.

What this code does is taking the first argument of the list (first city), and add this to the weight of the previous city (this works thanks to backtracking that is started first thing in the predicate by recalling the same predicate, at some point it will assign 0 to WeightAc when the lists are empty).

It finds the weights of the cities because it will compare the city value in the delivery facts with the one that is now first in the list. It will only take the weight of the one that matches the city.

Text

Description automatically generated

This predicate will add the truck weight to all the loads.

It does this by adding the first weight that it received as the second parameter to the truckweight, and adding this value to the list in the last parameter. The first line also recalls the same predicate, and it will do this as long as there are weights in the list. As soon as this list is empty, it will add the truckweight to the list in the last argument, which means (thanks to backtracking) that the truckweight will be the last and lowest item in the list.

Text

Description automatically generated

In this predicate we start calculating the total time and the warehouses where we’ll have to charge. For this we require a sequence of warehouse ids.

We start with getting the truckweight and current truck battery level.

After that we start calculating the total load and how it decreases by calling the sum\_weights predicate. We’ll have the list of the (total) weights from our deliveries from high to low.

The list that results from this predicate will be used as data to which the truck weight will be added in the add\_truck\_weight predicate.

Then we declare the warehouse that will function as the main warehouse. This is the warehouse from which all the trucks depart, and also the warehouse where they all arrive at the end of their trip.

After that we’ll add that warehouse to the beginning and the end of the trip by using the append predicate with as first argument a list with as first element of the list the main warehouse, and as tail of the list the list with the warehouse Ids, and the second argument is the main warehouse id again.

Now the predicate will be called that does the calculation effectively, with as argument the full warehouse list that tells us from where to where the deliveries go, the current battery level, the time (doesn’t have data but will return to us the total time this delivery path takes), a variable to save the warehouses where charging took place and the list with the total loads inclusive truck weight (see next page).

A computer screen capture

Description automatically generated with medium confidenceEvery time this predicate is called we take the first two warehouseIds out of the warehouseIds list. The other arguments contain data for the current truck battery level, the time it takes to go from w1 -> w2, the list where we stored where charges took place, the list with truck loads from which the first argument/head is being read.

First we read the facts and look for the fact that contains the data of the route from w1 to w2. From this fact we store the time it takes and the amount of energy/battery it requires.

We’ll calculate the right amount of energy it requires to go from w1 to w2 by multiplying the energy we received from the fact with the current load divided by 11800 (which is the mass of the truck plus its cargo).

After this we’ll check if the current battery of the truck is lower than the energy required to go from w1 to w2. If this is the case we’ll read the data from the truck to see what is the time that is required to charge to 80%, this time will be used later to calculate the total time. Then we’ll calculate the new energy (after charging) by taking the energy of the truck and subtract the energy from it that was required to go from w1 to w2. Finally we’ll add the current warehouse to the list of warehouses where charging took place.

If this was not the case, we subtract the current battery of the truck with the energy that was required to go from w1 to w2, assign 0 charging time to the TCharge variable and set the list that stores all the charges to the same/previous value.

After all this is done, the same cost predicate will be called again, and again, which means backtracking takes place until only one warehouse remains.

Then the rest of the code will be executed:

We calculate the max total weight of the truck which is truckweight plus maxload values of the truckdata. The subtraction of this value with the current load (=WT) will be stored in the WT1 value and will be used later in the calculation of the total time.

After this the time to charge will be divided by 48, and the result of this will be multiplied by the result of 48 subtracted by the current battery amount.

Finally, we retrieve the value of the main warehouse again. We’ll check if the destination warehouse doesn’t equal this warehouse, and if that’s the case, we’ll retrieve the unload data and add this the total time together with the time that is required to go from w1 to w2, the WT1 value we calculated earlier by dividing the MaxTotalWeight with the current load, the value of the time it took for the other warehouse paths and the TCharge3 value we also calculated earlier.

If the destination warehouse is the same, we don’t have to unload anymore and only have to calculate the time it takes us to go there. So we do the same calculation for time, but without the unload time.

Text

Description automatically generated

The only thing that’s left is the predicate that will find all the possible routes and calculate the costs/times of these. The one that has the smallest value at the time of verifying will always be stored and written. The result of this is the best path possible.

# **Study of the problem complexity as well as the viability of finding the optimal solution by generating all the solutions**

*By generating all the solutions and calculating the optimal solution, it is very realistic that we’ll manage to find the optimal solution.*

*But as explained in the first part, we have to take a lot of things into account, which we now will go much deeper into:*

The problem is that we have a logistic planning situation where we have a delivery guy that delivers packages to a lot of different warehouses, from one city to another, and this has to be done in the fastest way.

All the trucks will always leave Matasinhos with the total mass or total load of all the deliveries it carries and fully charged. As every Truck departs and arrives in Matasinhos, **no deliveries will be made to this main warehouse.**

There are direct connections between all the cities, so in fact there are roads between every single warehouse.

The wait time in a warehouse depends on several factors. One of those is the unload time, and another one is the time it takes a truck to charge its batteries up to 80%. This second one depends on the energy that will be spent on the next journey (so from warehouse 2 to warehouse 3, aka the next warehouse in the list).

Furthermore, we also have to check when we arrive at the next warehouses, that we arrive with at least 20% of trucks’ max energy.

Another thing to consider is that a truck should be able to be charged while it is still unloading, as this is way more productive, efficient, and effective.

This problem has the n! complexity based on the number of warehouses that are included in the delivery path (so the amount of deliveries to different warehouses).

|  |  |  |  |
| --- | --- | --- | --- |
| Nr. Of Delivery Warehouses | N. of solutions | List with the warehouse sequence for the  deliveries | Time to generate a solution |
| 5 | 120 | L = [8, 3, 1, 11, 9], | 0.030s |
| 6 | 720 | L = [17, 8, 3, 1, 11, 9], | 0.111s |
| 7 | 5040 | L = [17, 1, 14, 3, 8, 11, 9], | 0.250s |
| 8 | 40320 | L = [17, 8, 3, 12, 1, 14, 11, 9], | 0.520s |
| 9 | 362,880 | L = [17, 8, 3, 12, 6, 14, 1, 11, 9], | 2.648s |
| 10 | 3,628,800 | L = [17, 8, 3, 12, 6, 14, 1, 11, 13, 9], | 26.873s |
| 11 | 39,916,800 | L = [17, 2, 12, 6, 14, 1, 3, 8, 11, 13, 9], | 322.464s |
| 12 | 479,001,600 | L = [17, 2, 12, 6, 14, 1, 3, 8, 11, 13, 9, 4], | Took too long |
| 13 | 6,227,020,800 | L = [17, 2, 12, 6, 14, 1, 3, 8, 11, 13, 9, 4, 15], | Took too long |
| 14 | 87,178,291,200 | L = [17, 2, 12, 6, 14, 1, 3, 8, 11, 13, 9, 4, 15, 7], | Took too long |
| 15 | 1.307674368ᴇ12 | L = [17, 2, 12, 6, 14, 1, 3, 8, 11, 13, 9, 4, 15, 7, 10], | Took too long |
| 16 | 2.0922789888ᴇ13 | L = [17, 2, 12, 6, 14, 1, 3, 8, 11, 13, 9, 4, 15, 7, 10, 16], | Took too long |

# **Heuristics for the quick generation of solutions**

We need to apply heuristics to our solution, since the time needed to calculate with too many warehouses gets too great. We must apply three different heuristics:

- One that chooses minimum time to travel to from the current warehouse

- One that chooses largest mass to unload, so we have to spend less of time and energy in order to do the rest of the deliveries

- One that combines both

If we have for example warehouses 1, 9, 3, 8 and 11, so first we have five options to choose from, since we are starting from warehouse 5. We compare the times which are in this case: 141 minutes to warehouse 1, 48 minutes to warehouse 3, 29 minutes to warehouse 8, 48 minutes to warehouse 9 and 53 minutes to warehouse 11, so in this case we choose warehouse 8. Next, we need to choose from the warehouses we haven’t yet visited, them being 1, 3, 9, and 11, and again we do the same thing and choose the one with shortest time to drive. Now we compare the times: 134 minutes to warehouse 1, 32 minutes to warehouse 3, 69 minutes to warehouse 9 and 53 minutes to warehouse 11, so now we choose the warehouse 3. We continue this until we have delivered all the deliveries and return to the warehouse 5, them we calculate all the times and get the total time, which is going to be 513 minutes. The route we drive is changing depending on the deliveries we need to make. This example doesn’t take into account the facts that alter the time and energy spent, since they don’t matter with this solution. Using this heuristic is a lot less time consuming than the optimal solution, since we only need to compare the times between current warehouse to the warehouses we haven’t yet visited, so that would make 16 warehouses to compare at most assuming we’re leaving from the warehouse 5 and we have a delivery to make to all the warehouses.

Now we take the warehouses from the previous example and apply the second heuristic and choose the order of warehouses to visit, by the weight that gets unloaded at each warehouse. Again we start from warehouse 5, but instead of times we now choose the warehouse based on the weight of the delivery: 200kg to warehouse 1, 100kg to warehouse 3, 120kg to warehouse 8, 150kg to warehouse 9 and 300kg to warehouse 11. In this case we visit the warehouses in order: 11, 1, 9, 8, 3 and then we return to the warehouse 5. Now because we drop the heaviest mass first, the time needed and energy consumed will be lower for the rest of the deliveries. Using this heuristic again is lot less time consuming than the optimal solution, since we need to compare far lower number of elements at the same time.

# **Analysis of the heuristics quality**

/

# **Conclusion**

With our team we kind of managed to realize a solution to the given problem.

We have a solution that takes charging into account, so that trucks can’t go to another warehouse without charging (so less battery than the route requires isn’t allowed).

Our solution also checks all the possibilities and makes sure that it returns the delivery ids sequence that has the lowest cost/time amount.

Our solution also implements the requirement that there is main warehouse where trucks depart from and will arrive at, at the end of the day. This is realized by adding that id (5 – Matasinhos) to the list with all the Ids.

Furthermore, the solution also shows us at which specific warehouses we have to charge, so that we have an idea where the truck will have to wait a bit longer.

Even though the tasks weren’t fully implemented and not everything could be done, we’re sure that we can improve it even more in the next sprint to optimize and improve our solution.

Finally, a full code explanation was given in [*Study of the problem complexity as well as the viability of finding the optimal solution by generating all the solutions*.](#_Study_of_the)