Calculation of shortest paths with time windows and capacities for pharmacy delivery

Diogo José de Sousa Machado[1[up201706832]]
Eduardo Carreira Ribeiro[1[up201705421]]

Faculty of Engineering of the University of Porto - FEUP
https://sigarra.up.pt/feup/pt/web_page.inicial

Abstract. This project was done for the subject Logic Programming, in the 3rd Year of the Master in Informatics and Computing Engineering at FEUP.
The theme of our project resides on the need to program the schedule of a distribution company. The company operates from 10 AM to 10 PM, and needs to make a delivery to every pharmacy, based on their needs. Each pharmacy needs a different product quantity and also has a set schedule in which it is available to receive the deliveries. For these deliveries, the company has delivery trucks (a finite amount), and each truck has a specific capacity. Each truck leaves the company HQ to deliver the products to a set of pharmacies, and then returns to the central. Furthermore, the locations of the pharmacies and the central are known, as well as the distance/time between each pair of locations.
The objective of this project was to develop a Prolog application, using the ***clpfd*** library and its capacity to apply restrictions, that could generate the schedule of the distribution company, while minimizing both the number of trucks used and the overall distance traveled by the trucks.

Disclaimer: Our problem corresponds to an instance of the Vehicle Routing Problem with Time Windows (**VRP-TW**). Unfortunately, due to lack of time and the difficulty of this project/theme, we were not able to fully implement a solution for the problem. Instead, our solution is a modification of the Traveling Salesman Problem with Time Windows (**TSP-TW**). It allocates each delivery to a vehicle, considering vehicle capacities and delivery quantities. However, instead of generating a route for each vehicle, only one route is generated, containing all pharmacies and respecting their time windows, while minimizing the total travelling time. Throughout the main portion of this article, we will describe our implementation. However, in the end, we will also present a model for the VRP-TW problem, using restrictions, that would solve this problem completely.
Given the difficulty of this problem (especially compared to the other themes proposed in the subject), we hope our grade can still be good, because, as stated, our solution takes almost everything into account.

Keywords: PLR · Sicstus · Prolog · Shortest path · Traveling Salesman · Time Windows · Vehicle Routing Problem.

1 Introduction

As we have already described, the problem presented consists of generating a schedule for a delivery company, that has a limited number of trucks at its disposal. The schedule needs to take into account not only the time windows of all pharmacies (that is, the time interval during which they can receive the delivery), but also the limited capacities of the trucks and the delivery quantity each pharmacy demands. The optimal solution is the one that minimizes the total time spent by the trucks on trips, and the number of different trucks used.
It was also already referred that this problem is an instance of the Vehicle Routing Problem with Time Windows (**VRP-TW**). Our work was based on a number of research papers and academic articles about this problem and other related algorithms, which will be mentioned in the “references” section.
Lastly, and also already stated before, we remind that the solution implemented and presented doesn't completely solve the whole problem (only generates one route instead of various routes for all vehicles), but is very close to do so, and takes every aspect of the problem into account. Two models for a complete solution of the problem will be presented in the “annex” section.

The structure of the article will be the following:

– **Problem Description** - Detailed description of the problem/theme presented, and explanation of the input data that program takes
– **Approach** - Section presenting our approach to the problem. Will be divided in 4 sub-sections:
  • **Decision Variables** - Explanation of the decision variables and their respective domains, and their meaning in the context of the problem.
  • **Constraints** - Explanation of all the constraints used in our implementation.
  • **Evaluation Function** - Explanation of the evaluation function implemented and how it helps generate the best solution for the problem.
  • **Search Strategy** - Explanation of the labeling strategy used when generating a solution for the problem, including heuristics for the ordering of variables and values.
– **Solution Presentation** - How the program presents the solution to the user, in text format.
– **Results** - Calculation and presentation of important performance data of the program when solving instances with different dimensions, analyzing the results. Different labeling strategies will be tested and their results compared and presented.
– **Conclusions and Future Work** - Conclusion of the importance of this problem/theme, results obtained, and explanation of future work that could be done.
– **References** - Articles and papers that were part of our research during the development of this project.
– **Annex** - Presentation of the source code developed, input text files, results, and other important aspects of the project. Two models for solving the VRP-TW problem using restrictions will be presented in this section.

2 Problem Description

The decision problem in question consists in the elaboration of a schedule for a Pharmaceutical Products Distribution company, such that all scheduled orders can be delivered to their respective pharmacies, with the minimum possible cost to the company.
The distributor operates between 10AM and 10PM (10:00 – 22:00), and each pharmacy has its own time window, during which it is available to receive its respective order.
Each order has a volume of products that must be delivered, and each vehicle from the distributing company has a fixed maximum quantity of products that it can carry at once, at any given time.
The company headquarters (HQ), as well as each of the pharmacies, is associated to a location, and the travel time between the various points must be considered when building the schedule. In addition, the delivery of the order at a given pharmacy has a duration of 30 minutes.
All this data is supplied in .txt files, that should be properly formatted.

The main priority is to guarantee that all orders are successfully delivered. Having guaranteed that, the quality of the solution should be measured considering the number of used vehicles and the total distance travelled by the vehicles, the goal being that these values are minimized.
However, as we stated, the program will output a single route, with the allocation of pharmacies to the vehicles being made based on the total capacity of the trucks.

3 Approach

3.1 Decision Variables

The decision variables are grouped in 3 different lists:

- One list, with length equal to the number of locals (that is, the number of pharmacies plus the delivery central). **The variable in position i (indexing starts at 1) represents the next local, in the route, after visiting local i**. That is, if the first variable of the list has value 4, that means that after leaving the central (local number 1; pharmacy IDs start at 2), the next local to be visited is pharmacy number 4. The domain of each element of the list is an interval between 1 and the total number of locals.

- Another list, with length equal to the number of locals plus 1, representing the start times of the delivery/activity in each local/pharmacy. **The variable in position i represents the time (in minutes; in the output it is converted to a more readable format) at which service begins at local i**. The length of the list is the number of locals plus 1, because the central has two start time values: one for when we leave the central (first position of the list; always 10 AM), and the other one for when we arrive at the central (last position of the list). The domain of each element of the list is an interval representing the time window of that local, being that those time windows are specified to the program as input data.

- And one last list, with length equal to the number of pharmacies/deliveries, representing which vehicle is in charge of doing each delivery. **The variable in position i represents the vehicle assign to the delivery to be made in pharmacy i (because pharmacy IDs start at 2, the respective vehicle is in the first position of the list, and so on)**. The domain of each element of the list is an interval between 1 and the total number of vehicles.

3.2 Constraints

The first constraint applied is the predicate ***cumulatives/3***, which is used to associate each delivery to a vehicle. Each vehicle will correspond to a machine, with resources limit equal to the maximum truck capacity of that vehicle (done by the predicate **generateMachines/3**). Each delivery will correspond to a task, with start time equal to 1, duration equal to 1, resource quantity equal to the quantity demand of that pharmacy and the machine assigned connected to the respective variable of the delivery vehicles list described in the last section (done by the predicate ***generateTasks/5***). The cumulatives constraint is called using an upper bound. Because all tasks start and end at the same time, and the constraint does not let that, for any instant, the resources of the tasks assigned to a machine surpass the resource limit of that machine, the sum of resources of the deliveries assigned to a certain vehicle never surpasses the maximum capacity of that vehicle.

The predicate ***generate_constraints/6*** is then called in order to implement constraints regarding the route that will be generated and the start times of each delivery.

The distances matrix (received by the program as input data) has information about the times needed to travel from and to each pair of locals. Element xy of the matrix corresponds to the time need to travel from local x to local y. The first ***element/3*** restriction is used in order to get/constrain the cost/time of the trip between the current local and its sucessor (in variable **NewCost**). The nth1 predicate is used to get the start time of the service in the current local (in variable **Time**). The next constraint is used to constrain the start time of the service in the local's sucessor:

- If the current local is the central (Counter is 1), then no delivery is done there, so the start time of the next local has to be equal or higher to the sum of the start time in the current local, plus the time of the trip from the current local to the sucessor. The start time of the destination can be higher because if the time window of the sucessor hasn't "opened" yet, the truck can wait for that to happen.

$$Time + NewCost \;\; \# =< \;\; StartTimeDestination \tag{1}$$

– If the current local is a pharmacy (Counter is 2 or higher), then a delivery was done there. Because we know each delivery takes up 30 minutes, the constraint applied is:

$$Time + NewCost + 30 \;\; \# =< \;\; StartTimeDestination \tag{2}$$

After constraining/calculating the service start time of the local's sucessor, we need to constrain the equivalent variable in the start times list. The following constraint:

$$\begin{gathered}(Local\# = 1\#/\backslash TimeLocal\# = NumTimes)\\ \#\backslash/\\ (Local\#\backslash = 1\#/\backslash TimeLocal\# = Local)\end{gathered} \tag{3}$$

is applied in order to know what is the position in the list that corresponds to that variable. If the local's sucessor is 1 (meaning that the next local is the central), then we want to constrain the last variable of the list, that corresponds to the time at which we arrive at the central. If the local's sucessor is different than 1 (meaning that the next local is a pharmacy), then we want to constrain the equivalent position in the list, that is the start time of the delivery in the sucessor.

After having the position of the start times list that we need to constrain, we simply use the ***element/3*** constraint

$$element(TimeLocal, StartTimesList, StartTimeDestination) \tag{4}$$

The constraint presented are applied to every local, which successfully constraints and keeps the integrity of the start times of each local, respecting their time windows and the travel times between each pair of locals.

The constraints ***sum/3*** and ***nvalue/2*** are used to, respectively, constrain the sum of all travel times and the number of vehicles used, to be later used in the evaluation function.

The last constraint applied in our implementation is the constraint ***circuit/1***, which takes the list of successors and constrains each variable in it, so that the values correspond to a Hamiltonian Circuit (that is, a route is generated beginning in the central, passing in each pharmacy once and returning to the central).

3.3 Evaluation Function

The objective of the program is to find a solution that minimizes the total travel time of the trucks and the number of trucks used for the deliveries. Therefore, we needed an evaluation function that allowed that to happen.

The evaluation function that we opted to use is:

TimeCost + (10 x DifferentVehicles)

TimeCost being the sum of all travel times and DifferentVehicles being the number of different vehicles used. The value is constrained into a variable **Cost**, which is then minimized in the labelling call.

3.4 Search Strategy

The labeling strategy consists of minimizing the **Cost** variable, to find the solution that best fits the goal of the problem.

About the search strategies, we found that we obtained the best results with the ***ff*** strategy for variable sorting (with *ffc* also being a viable alternative), and the ***bisect*** strategy for variable selection (with the default *step* also being a viable alternative).

4 Solution Presentation

The program reads the input data from text files, that should be properly formatted. The predicate read/2 allows the reading of atoms from files and can therefore be used to obtain the data. Therefore, we have built the predicate ***readFiles/7***, which receives the files and saves the extracted data into lists.

About the output data, the program prints the results to the console, formatted in a way that makes it easy for the user to read the data from the screen. This is made by the predicate ***print_ output/5***, which presents the final data from the program, contained in the lists that are returned by the main predicate.

5 Results

To analyze the obtained results, we measured the solve time, the number of backtracks and the number of constraints created by the program.

Dataset	**# Pharmacies**	**# Vehicles**	**Time**	**Backtracks**	**Constraints Created**
Small	5	3 (all used)	0.0s	87	96
Medium	9	5 (all used)	0.3s	1763	156
Large	10	6 (5 used)	1.53s	82637	171

As it's clear to see, for small sets of data, the problem is solved almost instantaneously. However, as the size of the data grows, the execution time starts to grow exponentially. The same can be said for the number of backtracks. However, the number of constraints created grows linearly with the number of pharmacies and vehicles, not being affected by the search strategy.

The next table shows the difference between our chosen strategies, and the rest, using the *medium* dataset. The reason that no other strategies are included, is that they could not reach the solution in realistic time.

Search Strategies	**Execution Time**	**Backtracks**	**Constraints Created**
ff	0.03s	1708	156
ff, bisect	0.03s	1763	156
ffc	0.03s	1707	156
ffc, bisect	0.03s	1762	156
min	3.97s	172191	156
min, bisect	5.48s	175967	156

As we can see, all of the strategies create the same number of constraints, but our strategies perform much better in terms of execution time and backtracks.

Finally, we show the direct comparison between our strategies for the *large* data:

Search Strategies	**Execution Time**	**Backtracks**	**Constraints Created**
ff	3.82s	178088	171
ff, bisect	3.38s	178201	171
ffc	3.60s	178091	171
ffc, bisect	3.35s	178204	171

6 Conclusions and Future Work

While unfortunately we were not able to fully implement a complete solution for the problem, our program still contemplates every aspect of the problem and generates a valid solution given the input data (pharmacy time windows, maximum capacity of each truck, number of trucks and pharmacies, delivery quantity for each pharmacy, trip times between the locals, etc). We believe that our take is still very positive, successfully solving a (still) very difficult problem.

As future work, the full resolution of the problem could be implemented, which would generate a different route for each vehicle. Two solutions for the whole problem will be described in the end of the “annex” section, that we would’ve implemented if we had more time. One solution is more practical, making use of some built-in constraint predicates of the ***SICStus PLR*** library, and the other one is more theoretical, explaining in detail every aspect of the problem and how we can model it.

References

Apart from the consultation and study of the PowerPoints and other material of the PLOG subject, our research included the following articles and papers:

– http://www.msc-les.org/proceedings/emss/2009/EMSS2009_Vol_1_105.pdf
– http://www.bernabe.dorronsoro.es/vrp/data/articles/VRPTW.pdf
– https://www.researchgate.net/publication/220297462_Solving_Vehicle_Routing_Problems_Using_Constraint_Programming_and_Lagrangean_Relaxation_in_a_Metaheuristics_Framework
– https://www.semanticscholar.org/paper/Solving-Set-Partitioning-Problems-with-Constraint-Mueller/a33c8f9205aa1f31fd22c62b08b8b24411d3e5e7
– http://cognitive-robotics17.csail.mit.edu/docs/tutorials/Tutorial10_Multi_vehicle_Routing_with_Time_Windows.pdf
– http://egon.cheme.cmu.edu/ewo/docs/EWO_seminar_van_Hoeve.pdf
– https://ozgurakgun.github.io/ModRef2017/files/ModRef2017_MTSP.pdf
– https://pdfs.semanticscholar.org/e14a/d53dd65244bccff68fbce31e5e9301c12981.pdf
– https://www.business2community.com/strategy/solving-the-multiple-traveling-salesman-problem-with-constraints-0406913
– http://www.hakank.org/sicstus/tsp.pl

Annex

For this project, we opted to base our approach in small increments, first considering only the travel times between the locals (regular TSP), then considering the time windows of the pharmacies (TSP with Time Windows), and finally considering time windows for the pharmacies while allocating each delivery to a vehicle (TSP with Time Windows, while allocating each delivery to a vehicle). Because of that, the developed code is split into 3 parts, each one corresponding to each iteration of the process. Currently the program is using the **part 3** (TSP with Time Windows, while allocating each delivery to a vehicle) to calculate the output.
Part 4 of this process would be the resolution of the Vehicle Routing Problem with Time Windows. Even though we were not able to fully implement this part, an attempt to do so was made, and shouldn't be too far from a possible correct implementation. The code for that will also be presented at the end of this section.

Possible VRP-TW Restriction-Based Models for solving the problem

As referred previously throughout the article, we will now present two possible approaches that solve the complete presented problem, one more practical and the other more theoretical, based on algorithms for the Vehicle Routing Problem with Time Windows.

- **Practical Solution for the presented problem**

This practical model is based on the use of the constraint ***cumulatives/3*** not only for assigning each delivery to a vehicle, but also to constrain the start times of deliveries that are done by same vehicle.
The domain variables used are the start times list and the delivery vehicles list, both described in section 3.1.

The first part is equal to what it was done in our implementation, described in 3.2. Each delivery corresponds to a task, each vehicle corresponds to a machine, and a first call to the ***cumulatives/3*** predicate assigns each delivery to a vehicle, taking into account the quantity demands of each delivery and the maximum capacity of each vehicle.
The second part of this solution is based on the use, again, of the ***cumulatives/3*** constraint, this time to constrain the start time of each delivery, based on the start times of other deliveries on the same vehicle. This time, each delivery still corresponds to a task, but the start time is connected to the respective domain variable (of the start time of that delivery), the duration is 1, and the resources consumed by each task is 1. Each vehicle still corresponds to a machine, but the resources limit for each one of them is 1, which forces the tasks to be sequential in each machine (which is what we want). What this does is schedule the deliveries for each vehicle to be done in a sequential order, but we need to add some

additional time constraints to the start time of each delivery in order to respect the travel times between places, and the 30 minute duration of each delivery.

In order to do that, for each pair of deliveries, we use the following reified constraint:

$$\begin{gathered}(StartTimeA \ \ \# < StartTimeB \ \ \#/\backslash \ \ VehicleA \ \ \# = VehicleB) \\ \#{<}{=}{>} \ \ Z\end{gathered} \tag{5}$$

It translates to: if delivery A will start before delivery B and both will be done by the same vehicle, then variable Z will be set to 1; otherwise it will be set to 0. It can be a way to, using constraints, do something similar to an if condition.

So, if Z is set to 1, then we want to constrain the start time of delivery B:

$$Z \ \ \#{>}{=} \ \ (StartTimeB \ \ \#{>}{=} StartTimeA + 30 + TravelTimeAB) \tag{6}$$

It translates to: if Z is set to 1 then the start time of delivery B needs to be at least higher than the start time of delivery A, plus the 30 minute delivery at pharmacy A, plus the travel time between the two locals.

Finally, we only need to constrain the start time of the deliveries made in pharmacies that are the first or last local in a vehicle's route. For the first delivery of each vehicle, the following constraint has to stand:

$$StartTimeDelivery \ \ \#{>}{=} \ \ (600 + TravelTimeCentralDelivery) \tag{7}$$

It translates to: the start time of that delivery needs to be equal or higher than the central's opening (10 AM = 600 minutes), plus the travel time between the central and the pharmacy where the delivery is being made.

For the last delivery of each vehicle:

$$(StartTimeDelivery + 30 + TravelTimeDeliveryCentral) \ \ \# ={<}1320 \tag{8}$$

It translates to: the start time of that delivery, plus the 30 minute delivery time plus the travel time between the delivery and the central needs to be lower or equal than the central's closing hour (10 PM = 1320 minutes).

The labeling operation can then be called, minimizing both the total travel time and the number of used vehicles.

• Another Solution (more theoretical) for the presented problem

The theoretical model that will be presented was based in the model presented in this article:
http://www.msc-les.org/proceedings/emss/2009/EMSS2009_Vol_1_105.pdf

This model will use the following notation:

- **C = C1 . . . C** are the pharmacies to serve;
- **M = M1 . . . Mm** are the available vehicles;
- **Qm = Qm1 . . . Qmm** are the vehicles capacities;
- **V = V1 . . . V(n + 2m)** are the visits, with domain 1 .. m (each vehicle);
- Two sublists of **V**, **F** and **L**, are defined as the vehicles departure and arrival nodes/pharmacies;
- For each visit **Vi**, there is the predecessor of that visit **Pi**, and its sucessor, **Si**;
- **Ri** is the amount of product to deliver in visit **i**;
- After every visit, **Qi** is the quantity of product in the vehicle serving the visit;
- In terms of costs, **tij** represents the travel time from visit **i** to visit **j**;
- **Ti** represents the accumulated time (start time of the service) for visit **i**.

The constraints applied can be separated in various sections:

- **Difference constraints**: the following constraints force predecessors and successors to contain no repetitions. Thus, one pharmacy can have one and only one predecessor and successor.

$$\forall i, j \in V, i < j : p_i \neq p_j$$
$$\forall i, j \in V, i < j : s_i \neq s_j$$

- **Coherence constraints**: the following constraints connect the successor and predecessor as follows: the first one says that i is the successor of its predecessor, and the second one says that i is the predecessor of its successor.

$$\forall i \in V - F : s_{p_i} = i$$
$$\forall i \in V - F : p_{s_i} = i$$

- **Path constraints**: used to assure a route is visited by a single vehicle. Thus, the vehicle assigned to i must be the same as those assigned to its predecessor and successor.

$$\forall i \in V - F : v_i = v_{p_i}$$
$$\forall i \in V - L : v_i = v_{s_i}$$

- **Capacity constraints**: the constraints count the product quantity delivered in a route. The first constraint says the product quantity accumulated after visiting pharmacy i is the addition of the quantity accumulated in the predecessor plus the quantity delivered in i. The second is similar but using the successor.

$$q_i \geq 0$$
$$r_i \neq 0$$
$$\forall i \in V - F : q_i = q_{p_i} + r_i$$
$$q_i = q_{s_i} - r_{s_i} \quad \forall i \in V - L$$

– **Time constraints**: the constraints bound the accumulated time spent by a vehicle visiting pharmacy i. This time is, at least, the accumulated time in the predecessor of i, plus the 30-minute delivery at the predecessor, plus the travel time between the predecessor and i. The second constraint applies the same logic to the successor of i.

$$\begin{aligned} T_i &\geq 0 \\ T_i &\geq T_{p_i} + 30 + t_{p_i,i} \quad \forall i \in V - F \\ T_i &\geq T_{s_i} - 30 - t_{i,s_i} \quad \forall i \in V - L \end{aligned}$$

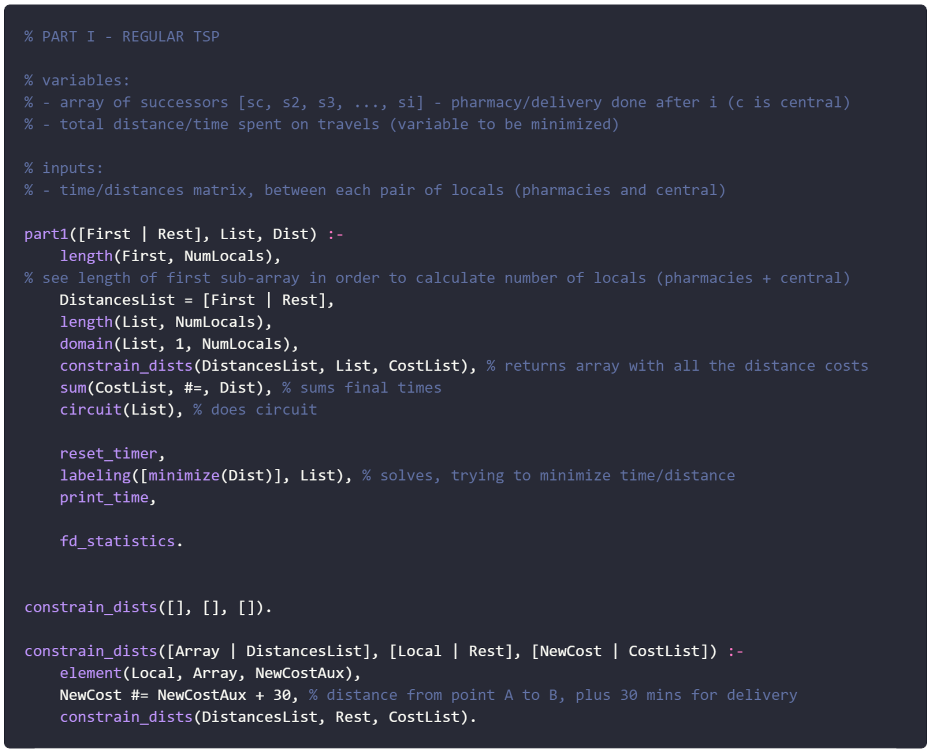
The values that we want to minimize are, as always, the total travel time and the number of different vehicles used:

$$\begin{aligned} TravelTime &= \sum_{i \in V-F} t_{p_i,i} \\ TravelTime &= \sum_{i \in V-L} t_{i,s_i} \end{aligned}$$

After applying these constraints, the labelling operation can then be called, generating the solution.

Relevant pictures and screenshots

In this section, the code used in this project is displayed, along with comments that make the interpretation easier.

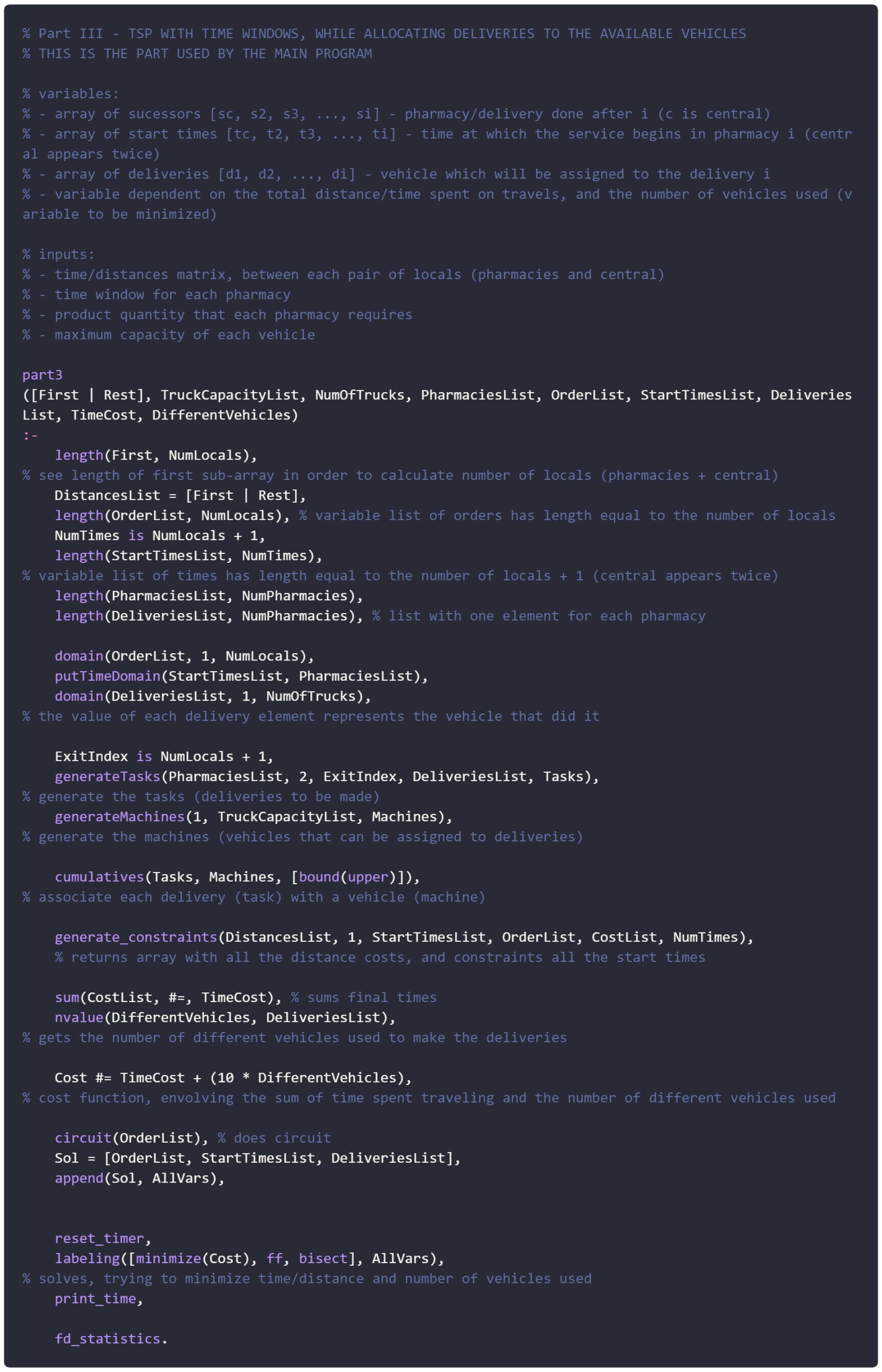


```
% PART I - REGULAR TSP

% variables:
% - array of successors [sc, s2, s3, ..., si] - pharmacy/delivery done after i (c is central)
% - total distance/time spent on travels (variable to be minimized)

% inputs:
% - time/distances matrix, between each pair of locals (pharmacies and central)

part1([First | Rest], List, Dist) :-
    length(First, NumLocals),
% see length of first sub-array in order to calculate number of locals (pharmacies + central)
    DistancesList = [First | Rest],
    length(List, NumLocals),
    domain(List, 1, NumLocals),
    constrain_dists(DistancesList, List, CostList), % returns array with all the distance costs
    sum(CostList, #=, Dist), % sums final times
    circuit(List), % does circuit

    reset_timer,
    labeling([minimize(Dist)], List), % solves, trying to minimize time/distance
    print_time,

    fd_statistics.


constrain_dists([], [], []).

constrain_dists([Array | DistancesList], [Local | Rest], [NewCost | CostList]) :-
    element(Local, Array, NewCostAux),
    NewCost #= NewCostAux + 30, % distance from point A to B, plus 30 mins for delivery
    constrain_dists(DistancesList, Rest, CostList).
```

Fig. 1. Part 1 of the implemented solution

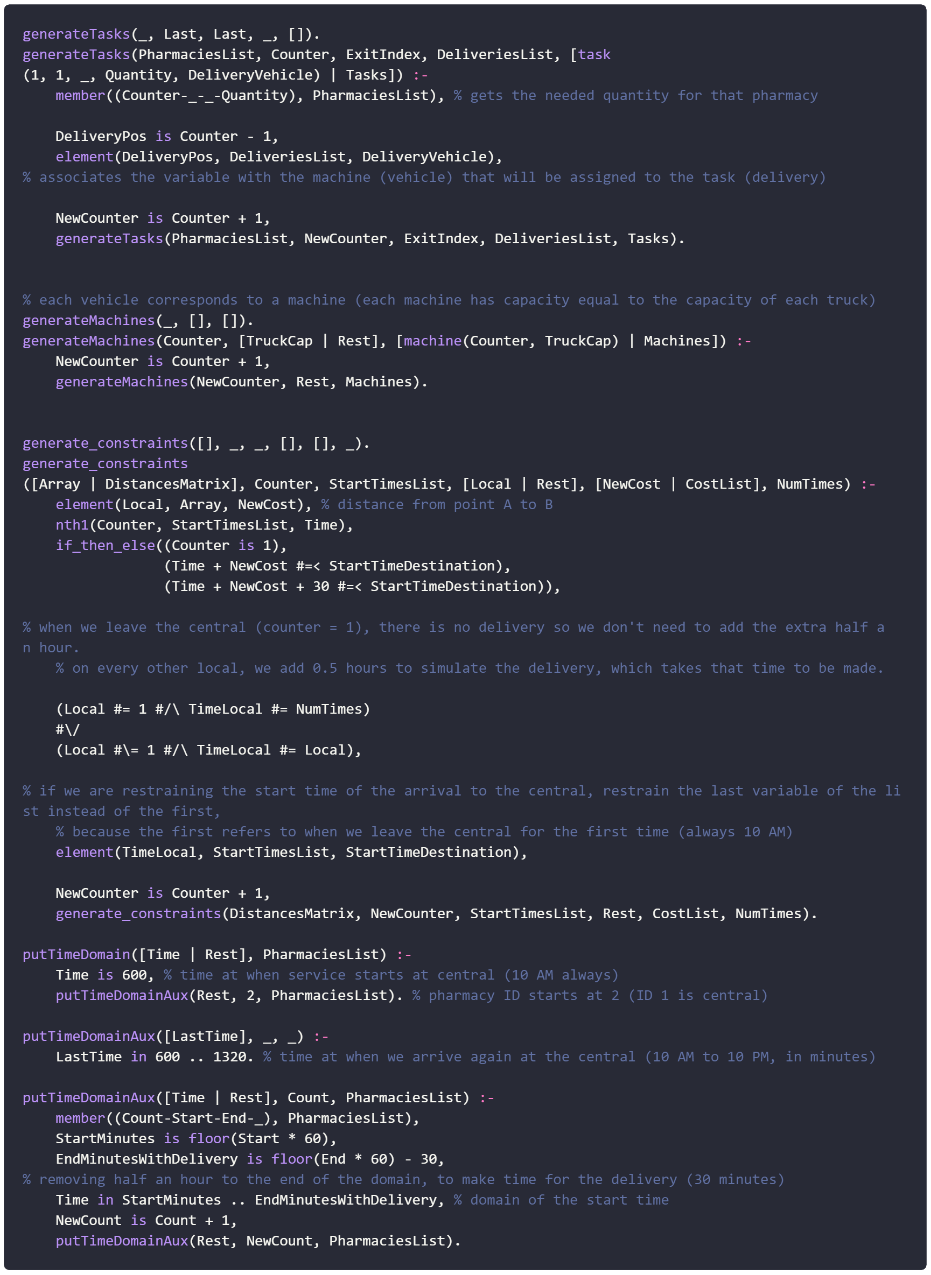
```prolog
% PART II - TSP WITH TIME WINDOWS - all times in minutes

% variables:
% - array of successors [sc, s2, s3, ..., si] - pharmacy/delivery done after i (c is central)
% - array of start times [tc, t2, t3, ..., ti] - time at which the service begins in pharmacy i
%                                                  (central appears twice)
% - total distance/time spent on travels (variable to be minimized)

% inputs:
% - time/distances matrix, between each pair of locals (pharmacies and central)
% - time window for each pharmacy

part2([First | Rest], PharmaciesList, OrderList, StartTimesList, Cost) :-
    length(First, NumLocals),
% see length of first sub-array in order to calculate number of locals (pharmacies + central)
    DistancesList = [First | Rest],
    length(OrderList, NumLocals), % variable list of orders has length equal to the number of locals
    NumTimes is NumLocals + 1,
    length(StartTimesList, NumTimes),
% variable list of times has length equal to the number of locals + 1 (central appears twice)

    domain(OrderList, 1, NumLocals),
    putTimeDomain(StartTimesList, PharmaciesList),

    generate_constraints(DistancesList, 1, StartTimesList, OrderList, CostList, NumTimes),
    % returns array with all the distance costs, and constraints all the start times

    sum(CostList, #=, Cost), % sums final times
    circuit(OrderList), % does circuit
    append(OrderList, StartTimesList, AllVars),

    reset_timer,
    labeling([minimize(Cost)], AllVars), % solves, trying to minimize time/distance
    print_time,

    fd_statistics.


generate_constraints([], _, _, [], [], _).
generate_constraints
([Array | DistancesMatrix], Counter, StartTimesList, [Local | Rest], [NewCost | CostList], NumTimes) :-
    element(Local, Array, NewCost), % distance from point A to B
    nth1(Counter, StartTimesList, Time),
    if_then_else((Counter is 1),
                 (Time + NewCost #=< StartTimeDestination),
                 (Time + NewCost + 30 #=< StartTimeDestination)),
    % when we leave the central (counter = 1), there is no delivery,
    %       so we don't need to add the extra half hour.
    % on every other local, we add 0.5 hours to simulate the delivery, which takes that time to be made.

    (Local #= 1 #/\ TimeLocal #= NumTimes)
    #\/
    (Local #\= 1 #/\ TimeLocal #= Local),
    % if we are restraining the start time of the arrival to the central,
    %        restrain the last variable of the list instead of the first,
    % because the first refers to when we leave the central for the first time (always 10 AM)
    element(TimeLocal, StartTimesList, StartTimeDestination),

    NewCounter is Counter + 1,
    generate_constraints(DistancesMatrix, NewCounter, StartTimesList, Rest, CostList, NumTimes).

putTimeDomain([Time | Rest], PharmaciesList) :-
    Time is 600, % time at when service starts at central (10 AM always)
    putTimeDomainAux(Rest, 2, PharmaciesList). % pharmacy ID starts at 2 (ID 1 is central)

putTimeDomainAux([LastTime], _, _) :-
    LastTime in 600 .. 1320. % time at when we arrive again at the central (10 AM to 10 PM, in minutes)

putTimeDomainAux([Time | Rest], Count, PharmaciesList) :-
    member((Count-Start-End-_), PharmaciesList),
    StartMinutes is floor(Start * 60),
    EndMinutesWithDelivery is floor(End * 60) - 30, % removing half an hour to the end of the domain,
                                                    %           to make time for the delivery (30 minutes)
    Time in StartMinutes .. EndMinutesWithDelivery, % domain of the start time
    NewCount is Count + 1,
    putTimeDomainAux(Rest, NewCount, PharmaciesList).
```

Fig. 2. Part 2 of the implemented solution



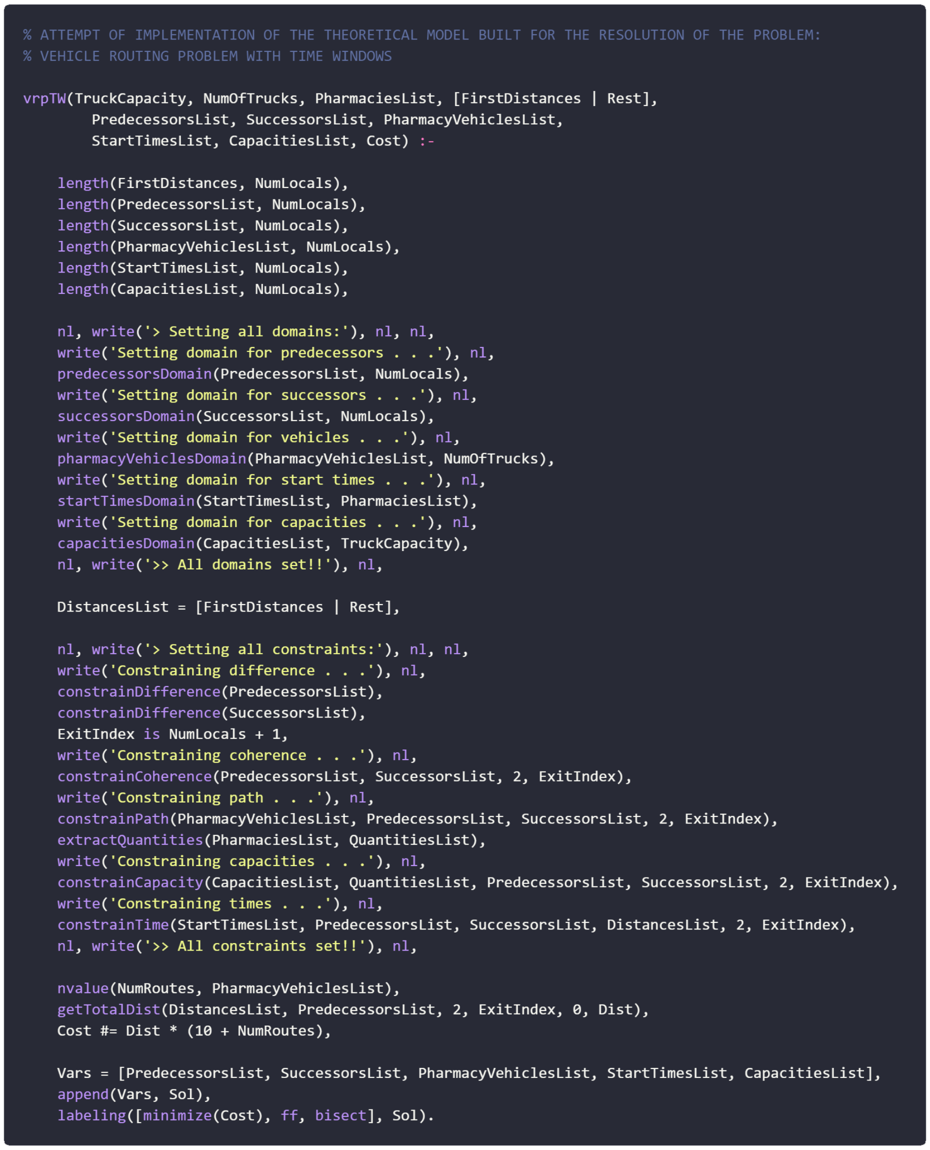
```
% Part III - TSP WITH TIME WINDOWS, WHILE ALLOCATING DELIVERIES TO THE AVAILABLE VEHICLES
% THIS IS THE PART USED BY THE MAIN PROGRAM

% variables:
% - array of sucessors [sc, s2, s3, ..., si] - pharmacy/delivery done after i (c is central)
% - array of start times [tc, t2, t3, ..., ti] - time at which the service begins in pharmacy i (centr
al appears twice)
% - array of deliveries [d1, d2, ..., di] - vehicle which will be assigned to the delivery i
% - variable dependent on the total distance/time spent on travels, and the number of vehicles used (v
ariable to be minimized)

% inputs:
% - time/distances matrix, between each pair of locals (pharmacies and central)
% - time window for each pharmacy
% - product quantity that each pharmacy requires
% - maximum capacity of each vehicle

part3
([First | Rest], TruckCapacityList, NumOfTrucks, PharmaciesList, OrderList, StartTimesList, Deliveries
List, TimeCost, DifferentVehicles)
:-
    length(First, NumLocals),
% see length of first sub-array in order to calculate number of locals (pharmacies + central)
    DistancesList = [First | Rest],
    length(OrderList, NumLocals), % variable list of orders has length equal to the number of locals
    NumTimes is NumLocals + 1,
    length(StartTimesList, NumTimes),
% variable list of times has length equal to the number of locals + 1 (central appears twice)
    length(PharmaciesList, NumPharmacies),
    length(DeliveriesList, NumPharmacies), % list with one element for each pharmacy

    domain(OrderList, 1, NumLocals),
    putTimeDomain(StartTimesList, PharmaciesList),
    domain(DeliveriesList, 1, NumOfTrucks),
% the value of each delivery element represents the vehicle that did it

    ExitIndex is NumLocals + 1,
    generateTasks(PharmaciesList, 2, ExitIndex, DeliveriesList, Tasks),
% generate the tasks (deliveries to be made)
    generateMachines(1, TruckCapacityList, Machines),
% generate the machines (vehicles that can be assigned to deliveries)

    cumulatives(Tasks, Machines, [bound(upper)]),
% associate each delivery (task) with a vehicle (machine)

    generate_constraints(DistancesList, 1, StartTimesList, OrderList, CostList, NumTimes),
    % returns array with all the distance costs, and constraints all the start times

    sum(CostList, #=, TimeCost), % sums final times
    nvalue(DifferentVehicles, DeliveriesList),
% gets the number of different vehicles used to make the deliveries

    Cost #= TimeCost + (10 * DifferentVehicles),
% cost function, envolving the sum of time spent traveling and the number of different vehicles used

    circuit(OrderList), % does circuit
    Sol = [OrderList, StartTimesList, DeliveriesList],
    append(Sol, AllVars),


    reset_timer,
    labeling([minimize(Cost), ff, bisect], AllVars),
% solves, trying to minimize time/distance and number of vehicles used
    print_time,

    fd_statistics.
```

Fig. 3. Part 3 of the implemented solution (the one currently in use) - main predicate



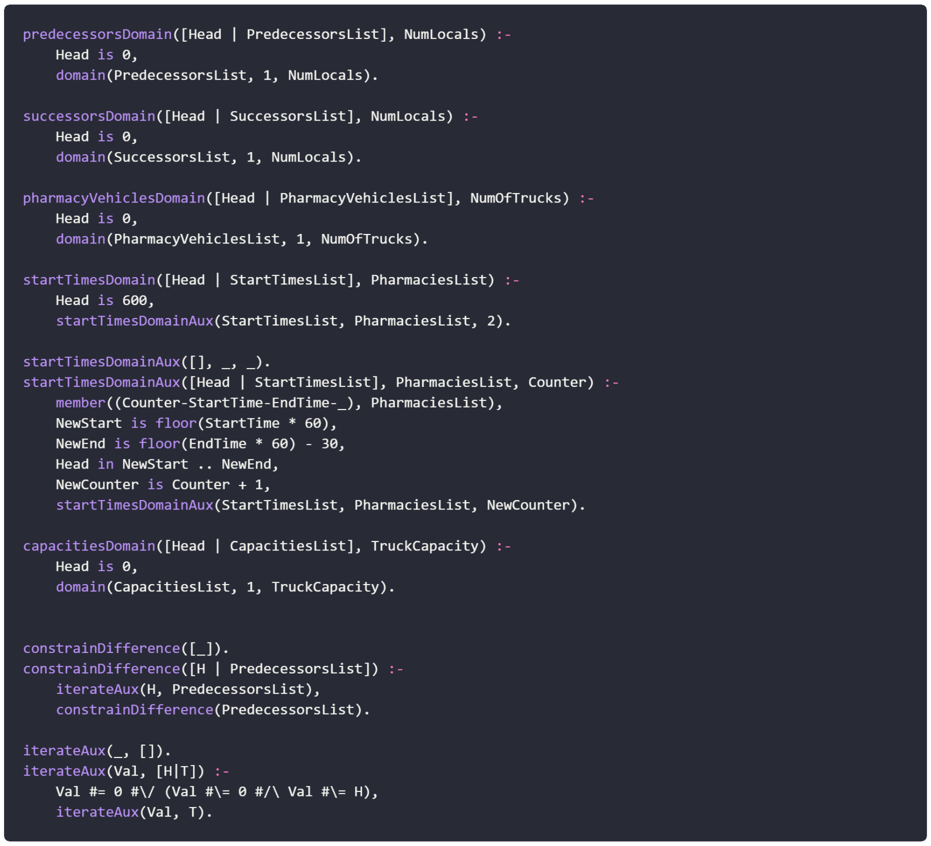
```
generateTasks(_, Last, Last, _, []).
generateTasks(PharmaciesList, Counter, ExitIndex, DeliveriesList, [task
(1, 1, _, Quantity, DeliveryVehicle) | Tasks]) :-
    member((Counter-_-_-Quantity), PharmaciesList), % gets the needed quantity for that pharmacy

    DeliveryPos is Counter - 1,
    element(DeliveryPos, DeliveriesList, DeliveryVehicle),
% associates the variable with the machine (vehicle) that will be assigned to the task (delivery)

    NewCounter is Counter + 1,
    generateTasks(PharmaciesList, NewCounter, ExitIndex, DeliveriesList, Tasks).


% each vehicle corresponds to a machine (each machine has capacity equal to the capacity of each truck)
generateMachines(_, [], []).
generateMachines(Counter, [TruckCap | Rest], [machine(Counter, TruckCap) | Machines]) :-
    NewCounter is Counter + 1,
    generateMachines(NewCounter, Rest, Machines).


generate_constraints([], _, _, [], [], _).
generate_constraints
([Array | DistancesMatrix], Counter, StartTimesList, [Local | Rest], [NewCost | CostList], NumTimes) :-
    element(Local, Array, NewCost), % distance from point A to B
    nth1(Counter, StartTimesList, Time),
    if_then_else((Counter is 1),
                 (Time + NewCost #=< StartTimeDestination),
                 (Time + NewCost + 30 #=< StartTimeDestination)),

% when we leave the central (counter = 1), there is no delivery so we don't need to add the extra half a
n hour.
    % on every other local, we add 0.5 hours to simulate the delivery, which takes that time to be made.

    (Local #= 1 #/\ TimeLocal #= NumTimes)
    #\/
    (Local #\= 1 #/\ TimeLocal #= Local),

% if we are restraining the start time of the arrival to the central, restrain the last variable of the li
st instead of the first,
    % because the first refers to when we leave the central for the first time (always 10 AM)
    element(TimeLocal, StartTimesList, StartTimeDestination),

    NewCounter is Counter + 1,
    generate_constraints(DistancesMatrix, NewCounter, StartTimesList, Rest, CostList, NumTimes).

putTimeDomain([Time | Rest], PharmaciesList) :-
    Time is 600, % time at when service starts at central (10 AM always)
    putTimeDomainAux(Rest, 2, PharmaciesList). % pharmacy ID starts at 2 (ID 1 is central)

putTimeDomainAux([LastTime], _, _) :-
    LastTime in 600 .. 1320. % time at when we arrive again at the central (10 AM to 10 PM, in minutes)

putTimeDomainAux([Time | Rest], Count, PharmaciesList) :-
    member((Count-Start-End-_), PharmaciesList),
    StartMinutes is floor(Start * 60),
    EndMinutesWithDelivery is floor(End * 60) - 30,
% removing half an hour to the end of the domain, to make time for the delivery (30 minutes)
    Time in StartMinutes .. EndMinutesWithDelivery, % domain of the start time
    NewCount is Count + 1,
    putTimeDomainAux(Rest, NewCount, PharmaciesList).
```

Fig. 4. Part 3 of the implemented solution (the one currently in use) - constraint predicates



```prolog
% ATTEMPT OF IMPLEMENTATION OF THE THEORETICAL MODEL BUILT FOR THE RESOLUTION OF THE PROBLEM:
% VEHICLE ROUTING PROBLEM WITH TIME WINDOWS

vrpTW(TruckCapacity, NumOfTrucks, PharmaciesList, [FirstDistances | Rest],
        PredecessorsList, SuccessorsList, PharmacyVehiclesList,
        StartTimesList, CapacitiesList, Cost) :-

    length(FirstDistances, NumLocals),
    length(PredecessorsList, NumLocals),
    length(SuccessorsList, NumLocals),
    length(PharmacyVehiclesList, NumLocals),
    length(StartTimesList, NumLocals),
    length(CapacitiesList, NumLocals),

    nl, write('> Setting all domains:'), nl, nl,
    write('Setting domain for predecessors . . .'), nl,
    predecessorsDomain(PredecessorsList, NumLocals),
    write('Setting domain for successors . . .'), nl,
    successorsDomain(SuccessorsList, NumLocals),
    write('Setting domain for vehicles . . .'), nl,
    pharmacyVehiclesDomain(PharmacyVehiclesList, NumOfTrucks),
    write('Setting domain for start times . . .'), nl,
    startTimesDomain(StartTimesList, PharmaciesList),
    write('Setting domain for capacities . . .'), nl,
    capacitiesDomain(CapacitiesList, TruckCapacity),
    nl, write('>> All domains set!!'), nl,

    DistancesList = [FirstDistances | Rest],

    nl, write('> Setting all constraints:'), nl, nl,
    write('Constraining difference . . .'), nl,
    constrainDifference(PredecessorsList),
    constrainDifference(SuccessorsList),
    ExitIndex is NumLocals + 1,
    write('Constraining coherence . . .'), nl,
    constrainCoherence(PredecessorsList, SuccessorsList, 2, ExitIndex),
    write('Constraining path . . .'), nl,
    constrainPath(PharmacyVehiclesList, PredecessorsList, SuccessorsList, 2, ExitIndex),
    extractQuantities(PharmaciesList, QuantitiesList),
    write('Constraining capacities . . .'), nl,
    constrainCapacity(CapacitiesList, QuantitiesList, PredecessorsList, SuccessorsList, 2, ExitIndex),
    write('Constraining times . . .'), nl,
    constrainTime(StartTimesList, PredecessorsList, SuccessorsList, DistancesList, 2, ExitIndex),
    nl, write('>> All constraints set!!'), nl,

    nvalue(NumRoutes, PharmacyVehiclesList),
    getTotalDist(DistancesList, PredecessorsList, 2, ExitIndex, 0, Dist),
    Cost #= Dist * (10 + NumRoutes),

    Vars = [PredecessorsList, SuccessorsList, PharmacyVehiclesList, StartTimesList, CapacitiesList],
    append(Vars, Sol),
    labeling([minimize(Cost), ff, bisect], Sol).
```

Fig. 5. Proposed implementation for the VRP-TW (part 4) - main predicate

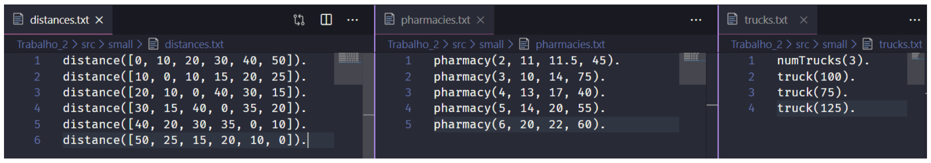


```
predecessorsDomain([Head | PredecessorsList], NumLocals) :-
    Head is 0,
    domain(PredecessorsList, 1, NumLocals).

successorsDomain([Head | SuccessorsList], NumLocals) :-
    Head is 0,
    domain(SuccessorsList, 1, NumLocals).

pharmacyVehiclesDomain([Head | PharmacyVehiclesList], NumOfTrucks) :-
    Head is 0,
    domain(PharmacyVehiclesList, 1, NumOfTrucks).

startTimesDomain([Head | StartTimesList], PharmaciesList) :-
    Head is 600,
    startTimesDomainAux(StartTimesList, PharmaciesList, 2).

startTimesDomainAux([], _, _).
startTimesDomainAux([Head | StartTimesList], PharmaciesList, Counter) :-
    member((Counter-StartTime-EndTime-_), PharmaciesList),
    NewStart is floor(StartTime * 60),
    NewEnd is floor(EndTime * 60) - 30,
    Head in NewStart .. NewEnd,
    NewCounter is Counter + 1,
    startTimesDomainAux(StartTimesList, PharmaciesList, NewCounter).

capacitiesDomain([Head | CapacitiesList], TruckCapacity) :-
    Head is 0,
    domain(CapacitiesList, 1, TruckCapacity).


constrainDifference([_]).
constrainDifference([H | PredecessorsList]) :-
    iterateAux(H, PredecessorsList),
    constrainDifference(PredecessorsList).

iterateAux(_, []).
iterateAux(Val, [H|T]) :-
    Val #= 0 #\/ (Val #\= 0 #/\ Val #\= H),
    iterateAux(Val, T).
```

Fig. 6. Proposed implementation for the VRP-TW (part 4) - variable domains

```
constrainCoherence(_, _, Last, Last).
constrainCoherence(PredecessorsList, SuccessorsList, Counter, ExitIndex) :-
    element(Counter, PredecessorsList, Pi1),
    element(Pi1, SuccessorsList, Si1),
    Si1 #= 0 #\/ (Si1 #\= 0 #/\ Si1 #= Counter),

    element(Counter, SuccessorsList, Si2),
    element(Si2, PredecessorsList, Pi2),
    Pi2 #= 0 #\/ (Pi2 #\= 0 #/\ Pi2 #= Counter),

    NewCounter is Counter + 1,
    constrainCoherence(PredecessorsList, SuccessorsList, NewCounter, ExitIndex).


constrainPath(_, _, _, Last, Last).
constrainPath(PharmacyVehiclesList, PredecessorsList, SuccessorsList, Counter, ExitIndex) :-
    element(Counter, PharmacyVehiclesList, Vi),

    element(Counter, PredecessorsList, Pi),
    element(Pi, PharmacyVehiclesList, Vpi),
    Pi #= 0 #\/ (Vpi #\= 0 #/\ Vpi #= Vi),

    element(Counter, SuccessorsList, Si),
    element(Si, PharmacyVehiclesList, Vsi),
    Si #= 0 #\/ (Vsi #\= 0 #/\ Vsi #= Vi),

    NewCounter is Counter + 1,
    constrainPath(PharmacyVehiclesList, PredecessorsList, SuccessorsList, NewCounter, ExitIndex).


extractQuantities([], []).
extractQuantities([(_-_-_-Quantity) | Tail], [Quantity | QuantitiesList]) :-
    extractQuantities(Tail, QuantitiesList).

constrainCapacity(_, _, _, _, Last, Last).
constrainCapacity(CapacitiesList, QuantitiesList, PredecessorsList, SuccessorsList, Counter, ExitIndex) :-
    nth1(Counter, QuantitiesList, Ri),
    element(Counter, CapacitiesList, Qi),

    element(Counter, PredecessorsList, Pi),
    element(Pi, CapacitiesList, Qpi),
    Qi #= Qpi + Ri,

    element(Counter, SuccessorsList, Si),
    element(Si, CapacitiesList, Qsi),
    element(Si, QuantitiesList, Rsi),
    Si #= 0 #\/ (Si #\= 0 #/\ Qsi #= Qi + Rsi),

    NewCounter is Counter + 1,
    constrainCapacity
(CapacitiesList, QuantitiesList, PredecessorsList, SuccessorsList, NewCounter, ExitIndex).


getDistance(DistancesList, ID1, ID2, Dist) :-
    nth1(ID1, DistancesList, List),
    element(ID2, List, Dist).

constrainTime(_, _, _, _, Last, Last).
constrainTime(StartTimesList, PredecessorsList, SuccessorsList, DistancesList, Counter, ExitIndex) :-
    element(Counter, StartTimesList, Ti),

    element(Counter, PredecessorsList, Pi),
    element(Pi, StartTimesList, Tpi),
    getDistance(DistancesList, Counter, Pi, DistIPi),
    Ti #>= Tpi + DistIPi + 30,

    element(Counter, SuccessorsList, Si),
    element(Si, StartTimesList, Tsi),
    getDistance(DistancesList, Counter, Si, DistISi),
    Ti #=< Tsi - DistISi - 30,

    NewCounter is Counter + 1,
    constrainTime(StartTimesList, PredecessorsList, SuccessorsList, DistancesList, NewCounter, ExitIndex).

getTotalDist(_, _, Last, Last, Dist, Dist).
getTotalDist(DistancesList, PredecessorsList, Counter, ExitIndex, Dist, FinalDist) :-
    element(Counter, PredecessorsList, Pi),
    getDistance(Counter, Pi, Increment),
    NewDist #= Dist + Increment,
    NewCounter is Counter + 1,
    getTotalDist(DistancesList, PredecessorsList, NewCounter, ExitIndex, NewDist, FinalDist).
```

Fig. 7. Proposed implementation for the VRP-TW (part 4) - variable constraints

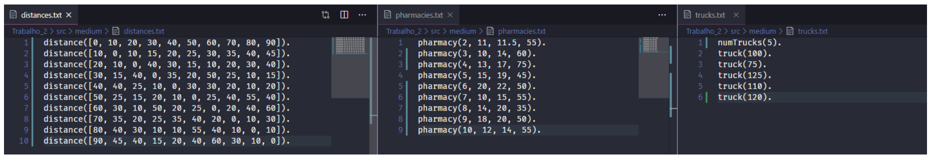

distances.txt
Trabalho_2 > src > small > distances.txt
pharmacies.txt
Trabalho_2 > src > small > pharmacies.txt
trucks.txt
Trabalho_2 > src > small > trucks.txt

```
distance([0, 10, 20, 30, 40, 50]).
distance([10, 0, 10, 15, 20, 25]).
distance([20, 10, 0, 40, 30, 15]).
distance([30, 15, 40, 0, 35, 20]).
distance([40, 20, 30, 35, 0, 10]).
distance([50, 25, 15, 20, 10, 0]).
```

```
pharmacy(2, 11, 11.5, 45).
pharmacy(3, 10, 14, 75).
pharmacy(4, 13, 17, 40).
pharmacy(5, 14, 20, 55).
pharmacy(6, 20, 22, 60).
```

```
numTrucks(3).
truck(100).
truck(75).
truck(125).
```

Fig. 8. Data used for the small example (5 pharmacies, 3 vehicles)

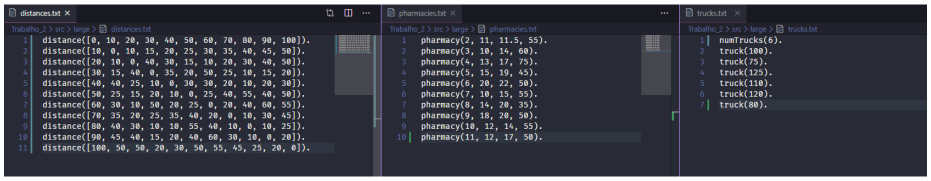

distances.txt
Trabalho_2 > src > medium > distances.txt
pharmacies.txt
Trabalho_2 > src > medium > pharmacies.txt
trucks.txt
Trabalho_2 > src > medium > trucks.txt

```
distance([0, 10, 20, 30, 40, 50, 60, 70, 80, 90]).
distance([10, 0, 10, 15, 20, 25, 30, 35, 40, 45]).
distance([20, 10, 0, 40, 30, 15, 10, 20, 30, 40]).
distance([30, 15, 40, 0, 35, 20, 50, 25, 10, 15]).
distance([40, 40, 25, 10, 0, 30, 30, 20, 10, 20]).
distance([50, 25, 15, 20, 10, 0, 25, 40, 55, 40]).
distance([60, 30, 10, 50, 20, 25, 0, 20, 40, 60]).
distance([70, 35, 20, 25, 35, 40, 20, 0, 10, 30]).
distance([80, 40, 30, 10, 10, 55, 40, 10, 0, 10]).
distance([90, 45, 40, 15, 20, 40, 60, 30, 10, 0]).
```

```
pharmacy(2, 11, 11.5, 55).
pharmacy(3, 10, 14, 60).
pharmacy(4, 13, 17, 75).
pharmacy(5, 15, 19, 45).
pharmacy(6, 20, 22, 50).
pharmacy(7, 10, 15, 55).
pharmacy(8, 14, 20, 35).
pharmacy(9, 18, 20, 50).
pharmacy(10, 12, 14, 55).
```

```
numTrucks(5).
truck(100).
truck(75).
truck(125).
truck(110).
truck(120).
```

Fig. 9. Data used for the medium example (9 pharmacies, 5 vehicles)

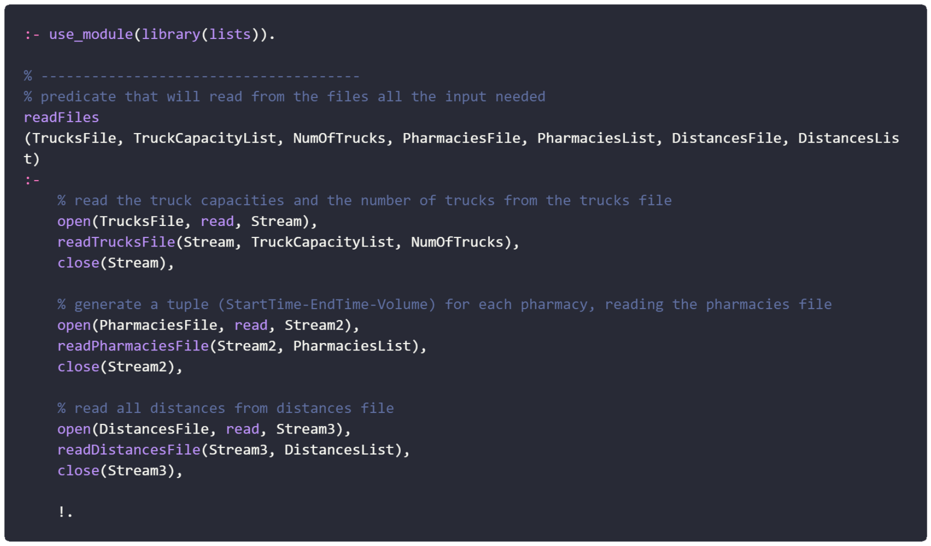

distances.txt
Trabalho_2 > src > large > distances.txt
pharmacies.txt
Trabalho_2 > src > large > pharmacies.txt
trucks.txt
Trabalho_2 > src > large > trucks.txt

```
distance([0, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100]).
distance([10, 0, 10, 15, 20, 25, 30, 35, 40, 45, 50]).
distance([20, 10, 0, 40, 30, 15, 10, 20, 30, 40, 50]).
distance([30, 15, 40, 0, 35, 20, 50, 25, 10, 15, 20]).
distance([40, 40, 25, 10, 0, 30, 30, 20, 10, 20, 30]).
distance([50, 25, 15, 20, 10, 0, 25, 40, 55, 40, 50]).
distance([60, 30, 10, 50, 20, 25, 0, 20, 40, 60, 55]).
distance([70, 35, 20, 25, 35, 40, 20, 0, 10, 30, 45]).
distance([80, 40, 30, 10, 10, 55, 40, 10, 0, 10, 25]).
distance([90, 45, 40, 15, 20, 40, 60, 30, 10, 0, 20]).
distance([100, 50, 50, 20, 30, 50, 55, 45, 25, 20, 0]).
```

```
pharmacy(2, 11, 11.5, 55).
pharmacy(3, 10, 14, 60).
pharmacy(4, 13, 17, 75).
pharmacy(5, 15, 19, 45).
pharmacy(6, 20, 22, 50).
pharmacy(7, 10, 15, 55).
pharmacy(8, 14, 20, 35).
pharmacy(9, 18, 20, 50).
pharmacy(10, 12, 14, 55).
pharmacy(11, 12, 17, 50).
```

```
numTrucks(6).
truck(100).
truck(75).
truck(125).
truck(110).
truck(120).
truck(80).
```

Fig. 10. Data used for the large example (10 pharmacies, 6 vehicles)



```
:- use_module(library(lists)).

% --------------------------------------
% predicate that will read from the files all the input needed
readFiles
(TrucksFile, TruckCapacityList, NumOfTrucks, PharmaciesFile, PharmaciesList, DistancesFile, DistancesLis
t)
:-
    % read the truck capacities and the number of trucks from the trucks file
    open(TrucksFile, read, Stream),
    readTrucksFile(Stream, TruckCapacityList, NumOfTrucks),
    close(Stream),

    % generate a tuple (StartTime-EndTime-Volume) for each pharmacy, reading the pharmacies file
    open(PharmaciesFile, read, Stream2),
    readPharmaciesFile(Stream2, PharmaciesList),
    close(Stream2),

    % read all distances from distances file
    open(DistancesFile, read, Stream3),
    readDistancesFile(Stream3, DistancesList),
    close(Stream3),

    !.
```

Fig. 11. Code for reading input from the files

```
% reads input from truck file
readTrucksFile(Stream, TruckCapacityList, NumOfTrucks) :-
    \+ at_end_of_stream(Stream),
    read(Stream, numTrucks(NumOfTrucks)),
    readTrucksFileAux(Stream, TruckCapacityList).

readTrucksFileAux(Stream, []) :-
    at_end_of_stream(Stream), !.

readTrucksFileAux(Stream, [TruckCap | Rest]) :-
    read(Stream, truck(TruckCap)),
    readTrucksFileAux(Stream, Rest).
```

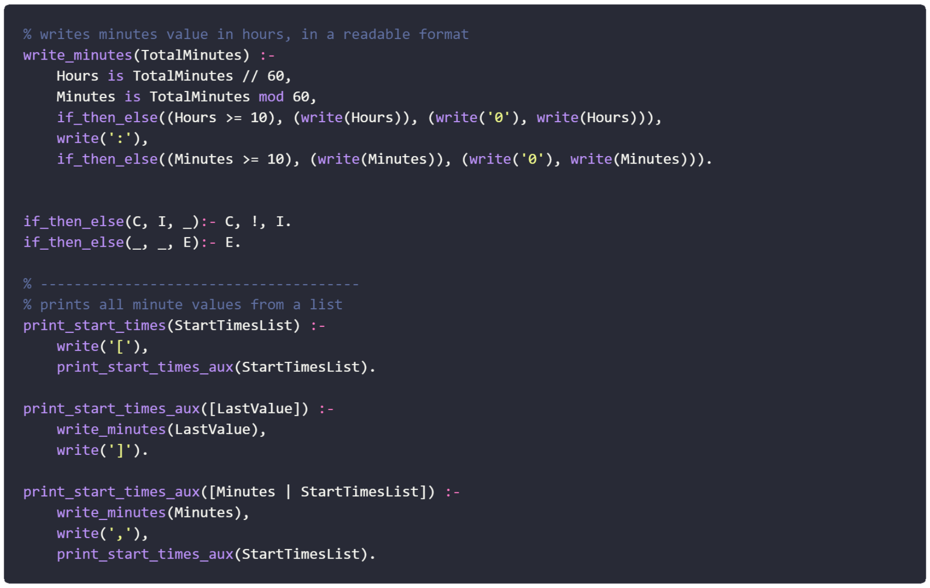
Fig. 12. Code for reading input for the trucks

```
% reads input from pharmacies file
readPharmaciesFile(Stream, []) :-
    at_end_of_stream(Stream), !.

readPharmaciesFile(Stream, [(ID-StartTime-EndTime-Volume) | Rest]) :-
    \+ at_end_of_stream(Stream),
    read(Stream, pharmacy(ID, StartTime, EndTime, Volume)),
    readPharmaciesFile(Stream, Rest).
```

Fig. 13. Code for reading input for the pharmacies

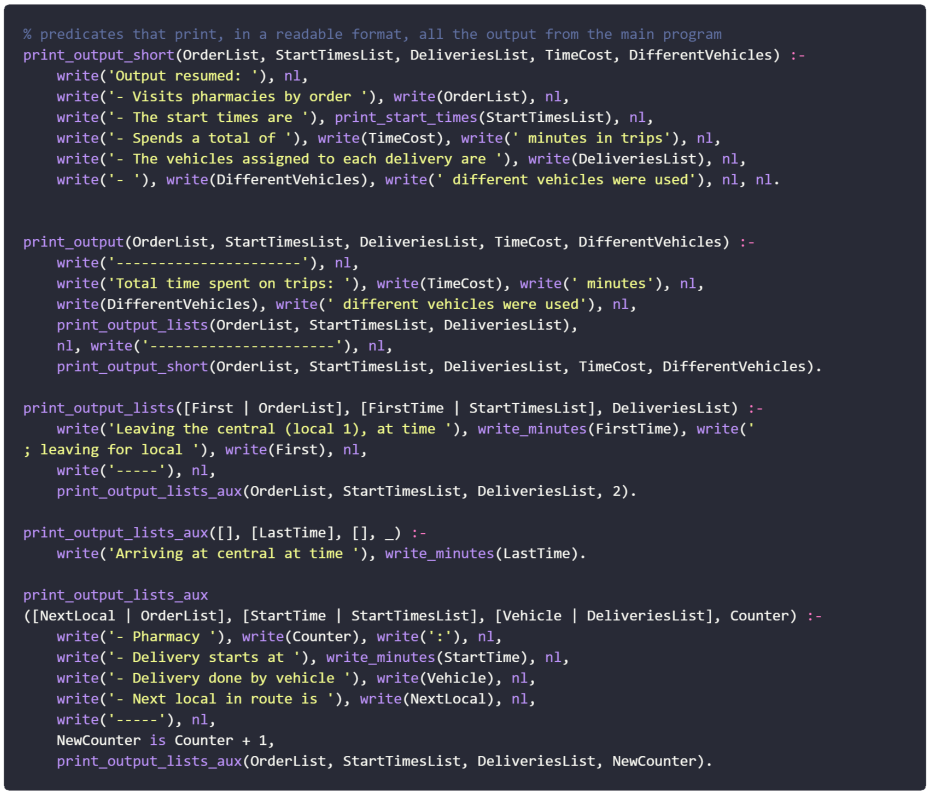
```
% reads input from distances file
readDistancesFile(Stream, []) :-
    at_end_of_stream(Stream), !.

readDistancesFile(Stream, [DistancesArray | Rest]) :-
    \+ at_end_of_stream(Stream),
    read(Stream, distance(DistancesArray)),
    readDistancesFile(Stream, Rest).
```

Fig. 14. Code for reading input for the distances



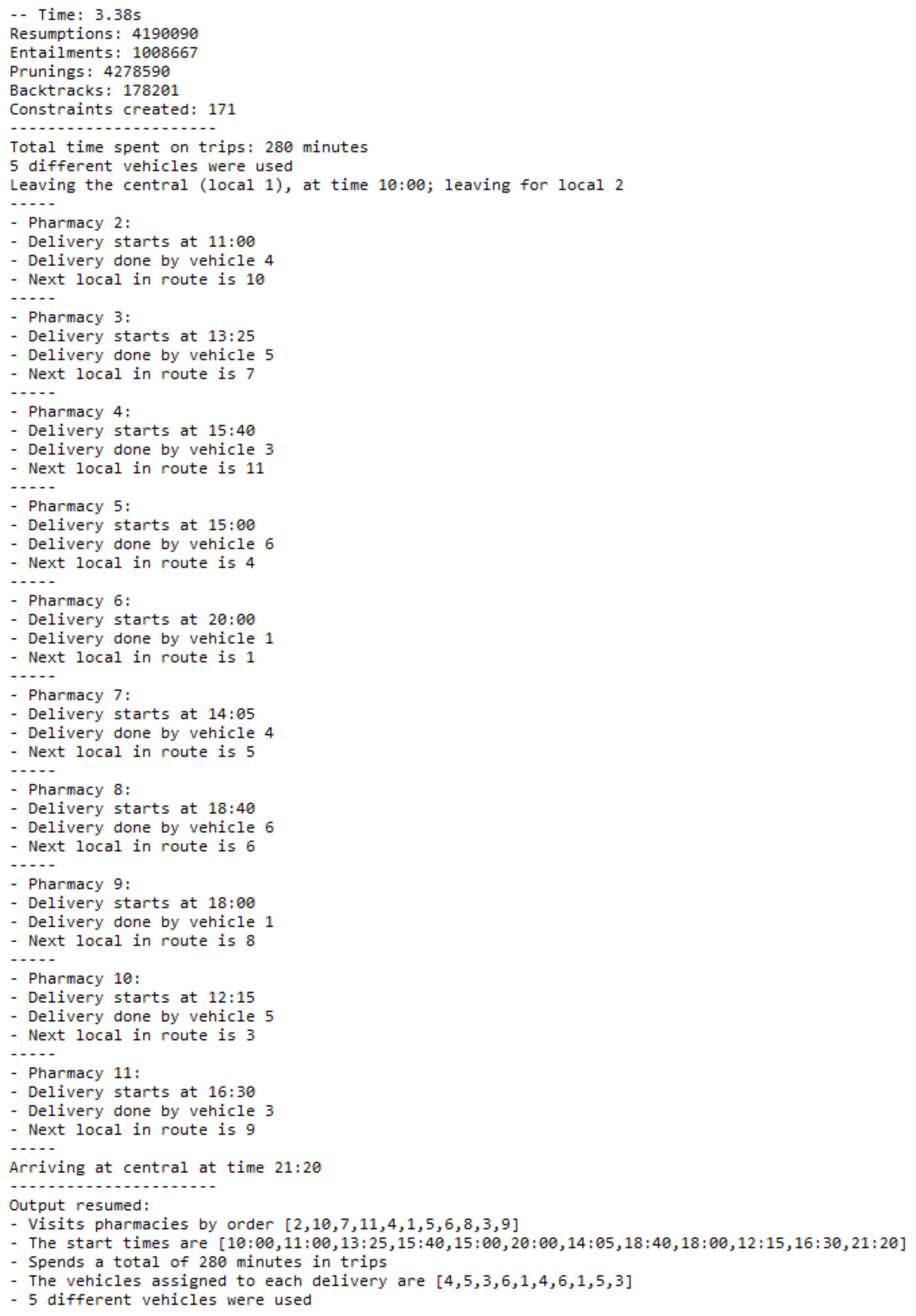
```
% writes minutes value in hours, in a readable format
write_minutes(TotalMinutes) :-
    Hours is TotalMinutes // 60,
    Minutes is TotalMinutes mod 60,
    if_then_else((Hours >= 10), (write(Hours)), (write('0'), write(Hours))),
    write(':'),
    if_then_else((Minutes >= 10), (write(Minutes)), (write('0'), write(Minutes))).


if_then_else(C, I, _):- C, !, I.
if_then_else(_, _, E):- E.

% -------------------------------------
% prints all minute values from a list
print_start_times(StartTimesList) :-
    write('['),
    print_start_times_aux(StartTimesList).

print_start_times_aux([LastValue]) :-
    write_minutes(LastValue),
    write(']').

print_start_times_aux([Minutes | StartTimesList]) :-
    write_minutes(Minutes),
    write(','),
    print_start_times_aux(StartTimesList).
```

Fig. 15. Code for reading and writing data for times



```
% predicates that print, in a readable format, all the output from the main program
print_output_short(OrderList, StartTimesList, DeliveriesList, TimeCost, DifferentVehicles) :-
    write('Output resumed: '), nl,
    write('- Visits pharmacies by order '), write(OrderList), nl,
    write('- The start times are '), print_start_times(StartTimesList), nl,
    write('- Spends a total of '), write(TimeCost), write(' minutes in trips'), nl,
    write('- The vehicles assigned to each delivery are '), write(DeliveriesList), nl,
    write('- '), write(DifferentVehicles), write(' different vehicles were used'), nl, nl.


print_output(OrderList, StartTimesList, DeliveriesList, TimeCost, DifferentVehicles) :-
    write('----------------------'), nl,
    write('Total time spent on trips: '), write(TimeCost), write(' minutes'), nl,
    write(DifferentVehicles), write(' different vehicles were used'), nl,
    print_output_lists(OrderList, StartTimesList, DeliveriesList),
    nl, write('----------------------'), nl,
    print_output_short(OrderList, StartTimesList, DeliveriesList, TimeCost, DifferentVehicles).

print_output_lists([First | OrderList], [FirstTime | StartTimesList], DeliveriesList) :-
    write('Leaving the central (local 1), at time '), write_minutes(FirstTime), write('
; leaving for local '), write(First), nl,
    write('-----'), nl,
    print_output_lists_aux(OrderList, StartTimesList, DeliveriesList, 2).

print_output_lists_aux([], [LastTime], [], _) :-
    write('Arriving at central at time '), write_minutes(LastTime).

print_output_lists_aux
([NextLocal | OrderList], [StartTime | StartTimesList], [Vehicle | DeliveriesList], Counter) :-
    write('- Pharmacy '), write(Counter), write(':'), nl,
    write('- Delivery starts at '), write_minutes(StartTime), nl,
    write('- Delivery done by vehicle '), write(Vehicle), nl,
    write('- Next local in route is '), write(NextLocal), nl,
    write('-----'), nl,
    NewCounter is Counter + 1,
    print_output_lists_aux(OrderList, StartTimesList, DeliveriesList, NewCounter).
```

Fig. 16. Code for displaying the program output



```
-- Time: 3.38s
Resumptions: 4190090
Entailments: 1008667
Prunings: 4278590
Backtracks: 178201
Constraints created: 171
----------------------
Total time spent on trips: 280 minutes
5 different vehicles were used
Leaving the central (local 1), at time 10:00; leaving for local 2
-----
- Pharmacy 2:
- Delivery starts at 11:00
- Delivery done by vehicle 4
- Next local in route is 10
-----
- Pharmacy 3:
- Delivery starts at 13:25
- Delivery done by vehicle 5
- Next local in route is 7
-----
- Pharmacy 4:
- Delivery starts at 15:40
- Delivery done by vehicle 3
- Next local in route is 11
-----
- Pharmacy 5:
- Delivery starts at 15:00
- Delivery done by vehicle 6
- Next local in route is 4
-----
- Pharmacy 6:
- Delivery starts at 20:00
- Delivery done by vehicle 1
- Next local in route is 1
-----
- Pharmacy 7:
- Delivery starts at 14:05
- Delivery done by vehicle 4
- Next local in route is 5
-----
- Pharmacy 8:
- Delivery starts at 18:40
- Delivery done by vehicle 6
- Next local in route is 6
-----
- Pharmacy 9:
- Delivery starts at 18:00
- Delivery done by vehicle 1
- Next local in route is 8
-----
- Pharmacy 10:
- Delivery starts at 12:15
- Delivery done by vehicle 5
- Next local in route is 3
-----
- Pharmacy 11:
- Delivery starts at 16:30
- Delivery done by vehicle 3
- Next local in route is 9
-----
Arriving at central at time 21:20
----------------------
Output resumed:
- Visits pharmacies by order [2,10,7,11,4,1,5,6,8,3,9]
- The start times are [10:00,11:00,13:25,15:40,15:00,20:00,14:05,18:40,18:00,12:15,16:30,21:20]
- Spends a total of 280 minutes in trips
- The vehicles assigned to each delivery are [4,5,3,6,1,4,6,1,5,3]
- 5 different vehicles were used
```

Fig. 17. The output given by our program, with the information about the achieved solution