Modelling and Solving the Vehicle Routing Problem (VRP)

Giosuè Cotugno Davide Pruscini

Academic Year 2020/2021

1 Introduction

In this report we will analyse the vehicle routing problem (VRP) which is part of the family of NP-complete algorithms, it has several variants, each with a different task.

The issue investigated throughout this report in particular is the CVRP.

The problem is to find routes for each vehicle, which will have a given ability and should serve all customers within a map via a single trip.

This must start at the depot and end at the same point, during the journey the load of each individual vehicle must not exceed its capacity and each trip will result in a cost: the objective is to minimise the total distance of all trips travelled and to minimise the number of vehicles used.

VRP is a topic of high interest at both academic and business level: logistics companies in particular, need to optimise the routes they travel on a daily basis, thereby reducing overall costs.

In literature we can find different approaches to solving this problem, each with its own advantages and disadvantages.

One example is Constraint Programming, this solution allows to define a declarative model in a simple way by exploiting the rules of the language, so as to obtain a minimum path finding algorithm that follow the restrictions of the formal definition.

In order to acquire better results, a number of operations have been performed on the datasets which focus mainly on the ordering of nodes within the graphs; heuristics search have then been used to specify to the solver how to process the variables defined in the model.

2 Modelling VRP in CP

This section will formally describe the rules imposed in the Vehicle Routing Problem by constraints and decision variables, and will analyse the data structure to be worked on.

2.1 Input data and any potential data processing

After reviewing the 11 datasets provided, it was possible to define a common input to the model defined as follows:

- **Name** represents the name of the file with the extension txt, containing the data.
- **locX** is a list of abscissae representing the X-axis coordinates of each customer except for the last one which indicates the X-coordinate of the depot.
- **locY** is a list of ordinates representing the Y-axis coordinates of each customer except the last one which indicates the Y-coordinate of the depot.
- **Demand** is a list containing the request of each customer, associated with the coordinates at the same location.
- **NumVehicles** is the number of vehicles available to satisfy customer demands.
- **Capacity** is a list containing the capacities of each vehicle.

From the list of coordinates we calculate a matrix called **distance**, of dimension $N + 1 \times N + 1$ where N represents the number of customers and 1 is the depot point, that represents the correlation of these coordinates with the strongly connected graph.

Other potential data elaborations to benefit from could be sorting the list of capacities in a descending order and sorting the matrix according to distances and/or the weight of the requests of each node, so as to contribute to the minimisation of the objective function by exploiting particular heuristics.

2.2 Decision Variable

In a Constraint programming language there are two types of data: the first resembles the constant variables of classical programming and, in this case, they are valued by the single input file; the second is called the decision variable and this type allows us to find all the possible solutions to our problem while respecting the restrictions provided by the constraints.

In particular, for this model, 4 decision variables were required, each with a specific task, before defining them it is essential to describe the sets on which they work:

- $LOAD = 0..max(Capacity)$ is the range of partial capacity of each vehicles.
- $VEHICLE = 1..NumVehicles$ is the range of avaible vehicles.
- $VEHICLE\_SYM = 1..NumVehicles–1$ is the range of avaible vehicles except the last.
- $CUSTOMER = 1..N$ is the range of customers nodes.
- $START\_NODES = (N + 1)..(N + NumVehicles)$ is the range of the start nodes; these are indexed after customer nodes.
- $START\_NODES\_PREPATH = (N + 2)..(N + NumVehicles)$ is the range of the start nodes except the first one.
- $END\_NODES = (N + NumVehicle + 1)..(N + (2NumVehicles))$ is the range of the end nodes; these are indexed after start nodes.
- $END\_NODES\_PATH = (N+NumVehicles+1)..(N+(2NumVehicles)-1)$ is the range of the end nodes except the last.
- $NODES = 1..(N + (2NumVehicles))$ is the set of all nodes.

Having these data sets it is now possible to formally define the decision variables used in the model:

- $path \in NODES$ represents the chosen path, in each position i we find its successor.
- $prePath \in NODES$ contains at each position i the predecessor of the node contained in *path* in position i.
- $vehicleRoute \in VEHICLE$ tells us, at each position i, which vehicle has visited that node defined in *path* in the same index.
- $demandEachTruck \in LOAD$ represents in position i the partial load of each vehicle based on the value of $vehicleRoute$ the index i.

2.3 Constraints

Having defined the constant variables, the data sets and the decision variables, it is now possible to describe the constraints to be imposed on them in order to obtain the desired results, some of which are declared redundant to avoid too many implicit constraints, and one which aims to break a symmetry in order to slightly reduce the search domain, as shown in [1].

We can divide the constraints that allow to find a route in 4 main categories:

- $circuit(path)$ e $circuit(prePath)$ are two global constraints that build a circuit between the elements of the individual lists by assigning j to the element x_i, where j represents the successor of the index i.

- $\forall i \in END\_NODES\_PATH, path_i = i - NumVehicles + 1$ with which we can express that at the end of the journey of a certain vehicle the journey of the next one will have to begin.
- $path_{N+(2NumVehicles)} = N+1$, allows the last dummy depot to be assigned the first element of *start node* in order to complete the circuit.
- $\forall i \in NODES, path_{prePath_i} = i$, specifies the relationship that must exist between $path$ and $prePath$ indicating that the successor of the predecessor of i must be i.
- $\forall i \in NODES, prePath_{path_i} = i$, similar to the previous one.
- $\forall i \in START\_NODES\_PREPATH, prePath_i = i + NumVehicles - 1$ with which we are able to express that the predecessor of one *start node* must be a *end node*.
- $prePath_{N+1} = N + (2NumVehicles)$, where we state that the predecessor of the first *start node* must be the last *end node*.

Constraints aiming to enforce rules on the load capacity of each vehicle:

- $\forall i \in START\_NODES, vechileRoute_i = i - N$, ensures that each vehicle will have a unique *start node*.
- $\forall i \in END\_NODES, vehicleRoute_i = i - N - NumVehicles$, ensures that each vehicle will have a unique *end node*.
- $\forall i \in CUSTOMER, vehicleRoute_{path_i} = vehicleRoute_i$, which specifies that the next vehicle will be defined by the previous one in use, thus allowing us to specify that the vehicle cannot change until it arrives at a next *start node*.
- $\forall i \in CUSTOMER, vehicleRoute_{prePath_i} = vehicleRoute_i$, similar to the previous one.

The constraints that manage the allocation of a vehicle to a particular customer:

- $\forall i \in STAR\_NODES, demandEachTruck_i = 0$, so as to have a partial load for each vehicle at the time of its departure equal to 0.
- $\forall i \in START\_NODES, demandEachTruck_i = demandEachTruck_{path_i}$, ensures that when all vehicles arrive at the first customer the partial load is 0.
- $\forall i \in CUSTOMER, demandEachTruck_i + demand_i = demandEachTruck_{path_i}$, where the partial loads for each vehicle will be updated according to the route it takes, the update will only take place when the vehicle moves on to the next customer.

- $\forall i \in CUSTOMER, demandEachTruck_i \leq Capacity_{vehicleRoute_i}$, checks that the part load of each vehicle is always less than or equal to its capacity.
- $\forall i \in VEHICLE, demandEachTruck_{i+N+NumVehicles} \leq Capacity_i$, check that the total load of each vehicle is less than or equal to its capacity.

Finally, we identified the constraint that allows us to reduce the search domain by breaking a symmetry:

- $\forall i \in VEHICLES\_SYM, path_{N+1} = N+NumVehicles+i \Rightarrow path_{N+(i+1)} = N + NumVehicles + (i+1)$, where the next vehicle will only be available if the previous one is used in the solution.

2.4 Objective Function

The first required task is minimising the distances, in order to do this we simply added all distances according to the path set by constraints, and then instructed the solver to minimise this sum.

The first objective function was then defined as follows:

$$min\ dist = \sum_{i=1}^{NODES} distance_{i,path_i}$$

Furthermore, we attempted to minimise also the number of vehicles used in the routes together with the distances, this operation was already implicitly done by our model after sorting the capacity list in a diminishing order and the constraint that breaks the symmetries, it was however decided to make it explicit by creating a multi-objective function. To obtain this, we opted for a linear scaling method in which both functions are multiplied by factors $ALPHA$ and $BETA$ whose value gives more weight to one than the other.

In order to normalise the values of the two functions, it was necessary to obtain the respective maxima, where for the number of vehicles it was known ($NumVehicles$), while for the maximum distance ($MaxDistance$), it was necessary to use a function to calculate an approximate value.

Having obtained this information, it was then possible to normalise the two functions to values in the interval $[0, 1]$.

The objective function obtained is therefore as follows:

$$minZ = \alpha \frac{dist}{MaxDistance} + \beta \frac{usedVehicles}{NumVehicles}$$

Where both coefficients, when set to 0.5, gave the best results; $usedVehicles$ represents the number of vehicles actually used to solve the route.

3 Experimental Study

In order to obtain good results, it was not enough to define the model in MiniZinc, since the use of the original data sets on it constituted very complex graphs,

hence it was decided to create a python script capable of ordering them according to different criteria and allowing the model to be run directly from the console thanks to the pymzn library.

3.1 Experimental Setup

The tests were carried out on a notebook with the following hardware and software characteristics:

- Windows 10, version 20H2, build SO 19042.746
- MiniZinc to FlatZinc converter, version 2.5.3, build 220798393
- Gecode solver, version 6.3.0
- Python 3.8.3, external libraries specified in the requirements.txt
- Intel(R) Core(TM) i7-8750H CPU @2.20GHz
- 16GB di RAM@2667MHz

3.2 Sort Strategy

The sorting possibilities provided by the script are four: distances, demands, customers and cusdenoc (customers, demands and no crossing [2]).

The first accounts for choosing the node furthest from the depot and then sorting the rest in a descending order; the second sorts the various nodes by descending request weight, customers sorts the nodes within the graph using a greedy technique, thus finding a sequence of nodes that constitutes a Hamiltonian path of minimum distance, obtainable in polynomial time.

Finally, cusdenoc behaves as the customers method with the addition of some controls on the capacity of vehicles aiming to minimise the number of incidences between the arcs as well.

In all types of sorting the coordinate pairs within locX and loxY were then reordered following the calculated path, in the same way the requests of each customer were also sorted to maintain consistency between the data. Other sorting methods were also tested, taking into account different aspects:

- Distances related to the demands of the arrival nodes through a coefficient;
- Distances related to the decrease in the number of incidences between the various arcs;
- Distances related to vehicle capacity, if this is exceeded the algorithm returns to search for nodes near the depot;

All of the methods presented make the model run successfully, however, they have not led to improvements and for this reason they are not all present within the python script.

The variant that gave us the best results, proposed in the next section, was customers, followed immediately by cusdenoc.

3.3 Search Strategy

Three heuristics were defined within the model and applied to some decision variables.

Specifically, the first_fail technique was used for the variables path, vehicleRoute and demandEachTruck, the method of choosing the values of the first two variables in the domain was indomain_split, while for the domain of the remaining variable, indomain_random was used. With first_fail, we specified to the solver to solve the most complex problems first, i.e. those with a smaller domain size which will generally be the most constrained.

In this case, therefore, the solver first aims at calculating the nodes of the depot and then the paths, which specify the order in which the customers visit.

The solver, during the execution of the model, will choose for the variables path and vehicleRoute a value in the first half of their domain; having already sorted the graph in the pre-processing phase we have that the values closest to the index of which we are choosing the successor will also be the closest.

The use of the indomain_min technique was not chosen because the minimum value will not necessarily lead to better results. As far as the domain specification for the variable demandEachTruck is concerned, it was decided not to specify the initial values that it may take, but to have these assigned randomly through the indomain_random specification, which made it possible to improve slightly in terms of failures.

We used also restart_luby for the path variable, with a scale parameter of $N \cdot 4$.

It is useful for eliminate the huge variance in solver performance and find a greater number of solutions.

Finally, using the Gecode solver, it was possible to use an annotation on path to mimic the LNS strategy, called relax_and_reconstruct, which is useful for fixing variables in a solution and searching for acceptable values for the remainder, specified by a percentage.

3.4 Experimental Results

Developing good model is time-consuming, which is why several models were produced with different approaches.

One of them stored the routes on a matrix and in each row it was possible to find the path of each individual vehicle, as shown in [3].

Despite its simplicity in implementation, it proved to be underperforming due to the size of the matrix, especially with very large data sets.

As suggested in the project description and in the [4] article, it was decided to abandon the use of a matrix in favour of arrays as data structures, to which the constraints described in the aforementioned sections were applied.

The results obtained with the last configuration described will be shown below, broken down by sorting type and each table will have within it the number of failures, the minimum distance obtained and the number of vehicles used for all the data sets available. To get an idea of good results to be achieved,

the same files were tested on another model using the OR-Tools library [5], which only outputs the distance of the route and the number of vehicles used, without considering the number of failures.

	no sorting		**distances**		**demands**	
data	*failures*	*solutions*	*failures*	*solutions*	*failures*	*solutions*
pr01	3.843.238	2.899.220	3.836.559	2.560.408	3.876.971	2.612.447
pr02	2.759.399	5.831.680	2.846.409	5.800.719	2.505.250	5.923.441
pr03	2.924.707	11.370.310	3.155.815	10.788.008	2.803.401	10.204.960
pr04	3.054.403	14.393.158	2.968.282	13.072.064	2.435.815	13.090.665
pr05	3.394.556	16.081.224	2.859.032	14.843.060	2.403.089	13.920.150
pr06	2.551.564	20.189.913	1.751.004	19.786.356	1.668.931	18.774.158
pr07	2.152.015	4.973.660	3.197.999	5.251.265	2.599.945	5.061.273
pr08	2.836.709	10.499.901	3.635.664	10.141.624	2.886.481	10.323.993
pr09	3.274.708	17.168.153	2.689.909	15.255.011	2.439.521	15.262.090
pr10	2.657.442	20.464.186	2.317.503	20.017.820	1.825.142	19.784.621
pr11	2.227.601	2.970.217	3.034.102	2.624.962	2.832.509	2.713.727

Table 1: In the table we can see the results of no sorting, sorting by distances and sorting by demands with time limit of 5 minutes.

	customers		**cusdenoc**	
data	*failures*	*solutions*	*failures*	*solutions*
pr01	4.159.969	1.155.013	4.933.239	1.159.086
pr02	2.642.750	1.973.833	2.798.672	2.051.087
pr03	3.062.269	3.534.452	3.265.395	3.651.405
pr04	3.045.163	4.368.161	3.159.436	4.425.259
pr05	2.577.861	4.954.532	3.017.761	4.788.492
pr06	2.524.671	5.356.315	2.116.763	5.255.293
pr07	2.859.892	1.420.009	3.267.837	1.730.439
pr08	3.194.910	3.104.148	3.721.276	3.365.231
pr09	2.870.159	4.059.828	3.101.866	4.271.915
pr10	2.421.488	5.313.343	2.553.970	5.713.083
pr11	2.618.694	1.180.222	3.001.077	1.184.295

Table 2: In this table we can see the results of sorting by customers and sorting by cusdenoc with time limit of 5 minutes.

Used vehicles		
data	*MiniZinc*	*OR-Tools*
pr01	3	3
pr02	6	7
pr03	9	10
pr04	14	14
pr05	19	19
pr06	20	20
pr07	4	5
pr08	11	11
pr09	15	16
pr10	20	20
pr11	3	3

Table 3: In this table we can see the number of vehicles used for each dataset, on left MiniZinc results remain the same for all the tests executed, on right OR-Tools results.

Best results				
	restart_luby		**OR-Tools**	
data	*failures*	*solutions*	*solutions*	*difference %*
pr01	1.088	939.967	954.597	1,55
pr02	9.796	1.529.770	1.543.838	0,92
pr03	6.029	2.707.912	2.511.420	-7,25
pr04	41.564	2.702.692	2.948.516	9,09
pr05	11.126	3.751.914	3.665.138	-2,31
pr06	86.660	4.204.914	4.076.057	-3,06
pr07	6.475	1.176.926	1.258.928	6,96
pr08	4.392	2.427.033	2.288.203	-5,72
pr09	48.999	3.320.849	3.345.540	0,74
pr10	105.088	4.282.774	4.163.476	-2,78
pr11	1.687	942.632	916.219	-2,80

Table 4: In this table we can see the results of our model with customers sorting + restart_luby and the results of OR-Tools, it showed also the difference in percentage of the distances gained for each data set.

As we can see from the Table 4, all our best results were obtained by sorting according to the customers criterion with luby restart, these deviated at most 10% from those obtained with the model based on OR-Tools.

It can also be seen that without any kind of ordering or using sorting by demands or distances we got the worst results, meanwhile using customers and cusdenoc we got an important improvement in term of distance.

Considering instead the number of vehicles used in all orders compared to those based on OR-Tools, it can be seen that in some cases the model proposed in this report uses a lower number of vehicles, which may be due to too strong restrictions on the number of vehicles.

Thanks to luby restart we got the biggest step in term of distance, in fact we were able to overlap, in majority of the case, the solutions got by the model based on the OR-tools library.

Finally, using a restart inside our model we escaped from the heavy tail behavior, allowing the solver to analyze more subtree then a search without restart can do.

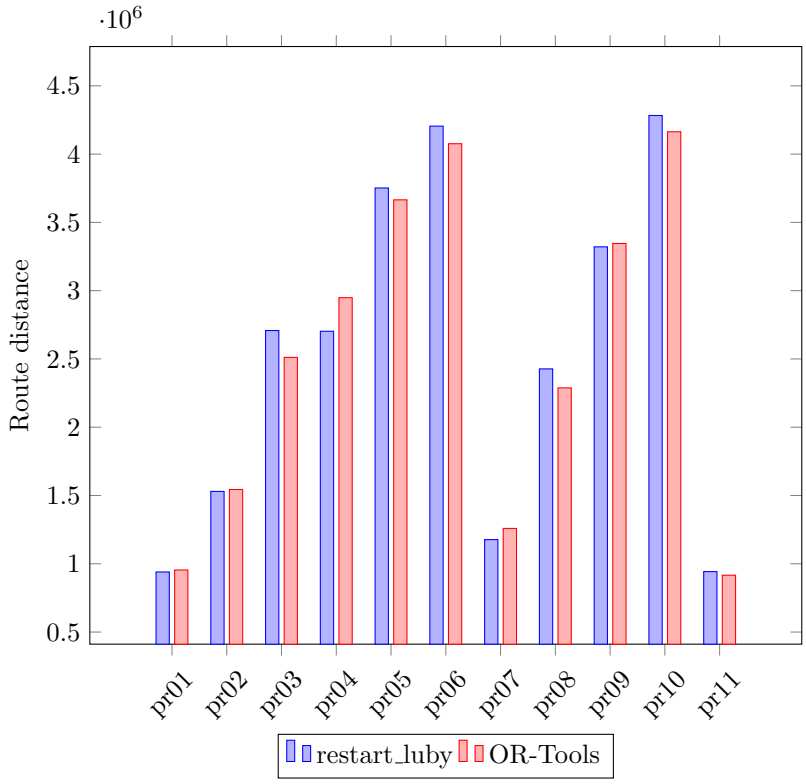

Figure 1: In this bar chart we can see the difference between customers sorting with restart_luby and OR-Tools library.

4 Conclusions

In this paper we have presented a solution for a variant of the VRP using constraint programming.

This allowed us to develop a model capable of finding solutions without too many difficulties.

In addition, the usage of different type of heuristics give us the opportunity to have at the start one solution immediately and at the end to have the best solution that we can have.

Finally, the pre-processing phase was fundamental to obtain results comparable to those obtained through the OR-Tools library.

Without any sort of ordering, as we have seen, the solutions were tripled in order of distance.

It is assumed that with an a greater amount of searching time the solver would be able to reach the smallest distance value.

References

[1] Mahé, A.; Urli, T.; Kilby, P. Fleet Size and Mix Split-Delivery Vehicle Routing. *CoRR*, abs/1612.01691, 2016. http://arxiv.org/abs/1612.01691.

[2] GeeksforGeeks. How to check if two given line segments intersect?, February 2020. https://www.geeksforgeeks.org/check-if-two-given-line-segments-intersect/.

[3] Fröhlingsdorf, W. David. Using Constraint Programming to solve the Vehicle Routing Problem with Time Windows. Master's thesis, University of Glasgow, 2018. http://www.jacktex.eu/research/material/master_thesis.pdf.

[4] Kajgård, E. Route Optimisation for Winter Road Maintenance using Constraint Modelling. Master's thesis, Uppsala Universitet, 2015. http://uu.diva-portal.org/smash/get/diva2:875504/FULLTEXT01.pdf.

[5] Google. Vehicle routing problem, June 2020. https://developers.google.com/optimization/routing/vrp.

A Route Plots

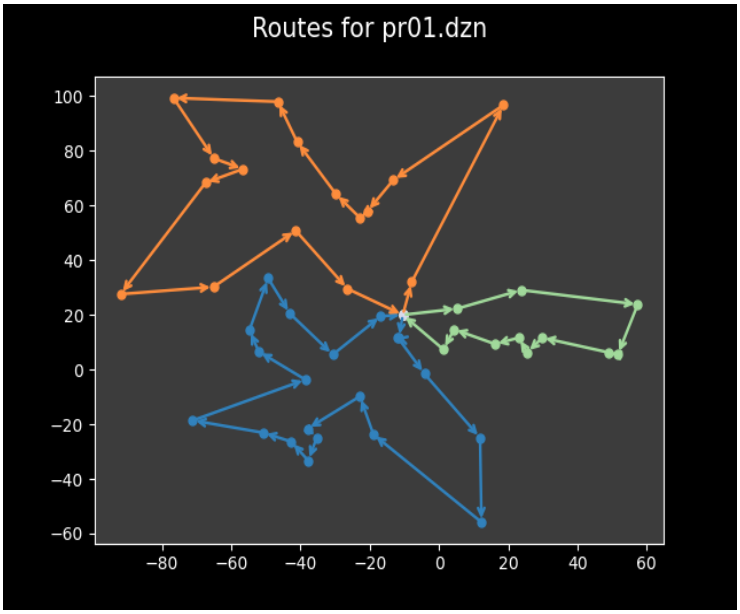

Figure 2: pr01 - sorting by customers, with luby restart

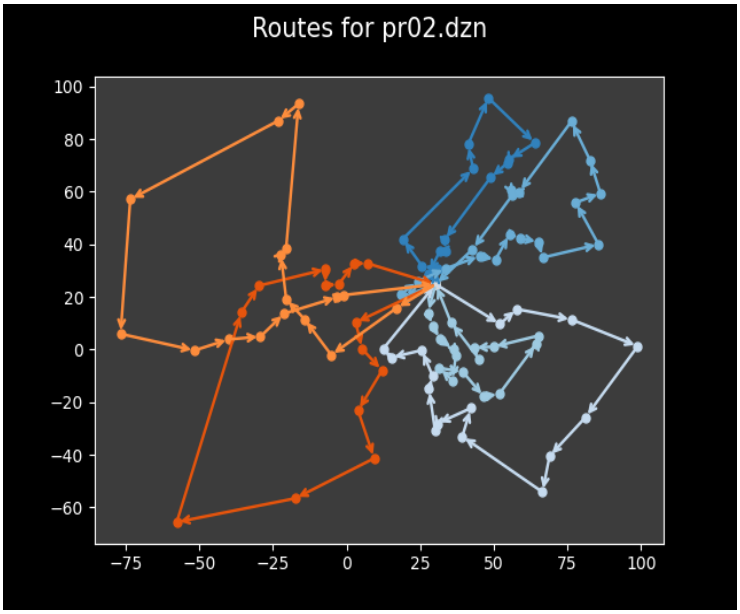

Figure 3: pr02 - sorting by customers, with luby restart

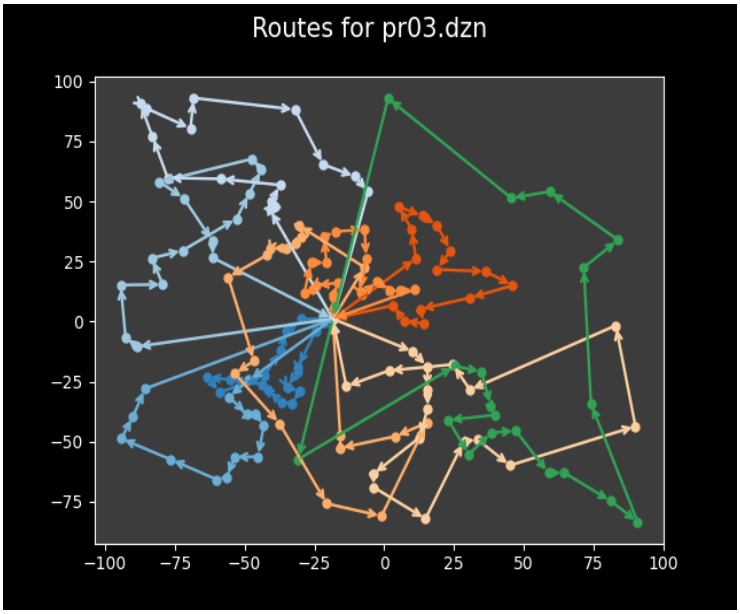

Figure 4: pr03 - sorting by customers, with luby restart

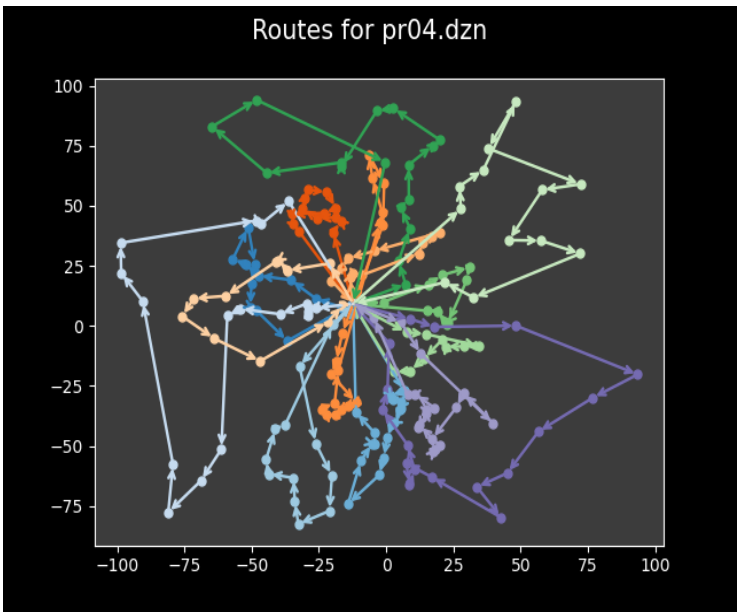

Figure 5: pr04 - sorting by customers, with luby restart

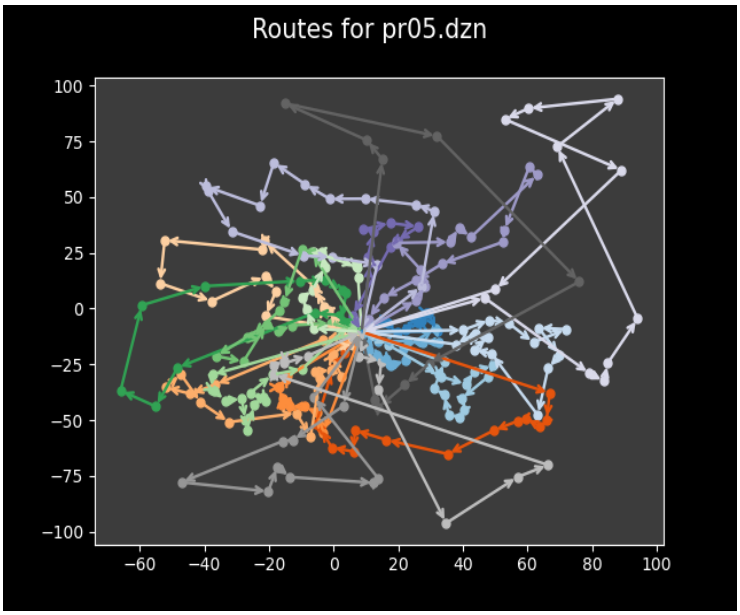

Figure 6: pr05 - sorting by customers, with luby restart

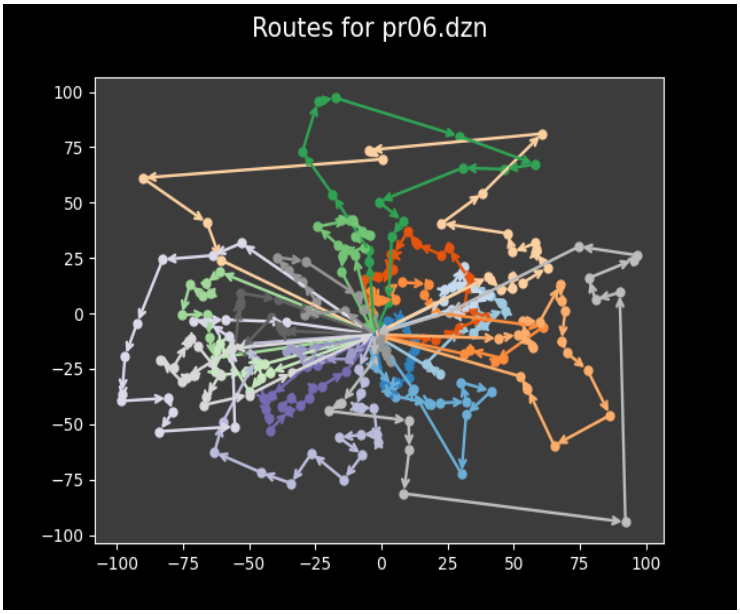

Figure 7: pr06 - sorting by customers, with luby restart

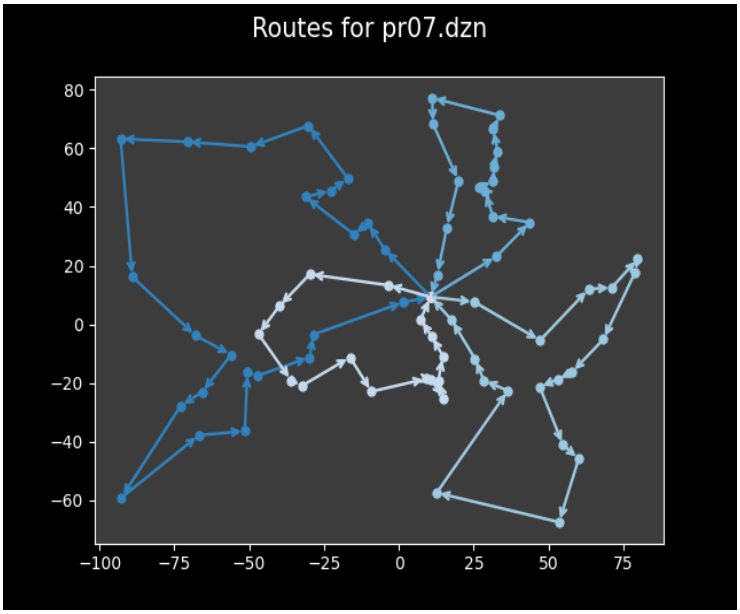

Figure 8: pr07 - sorting by customers, with luby restart

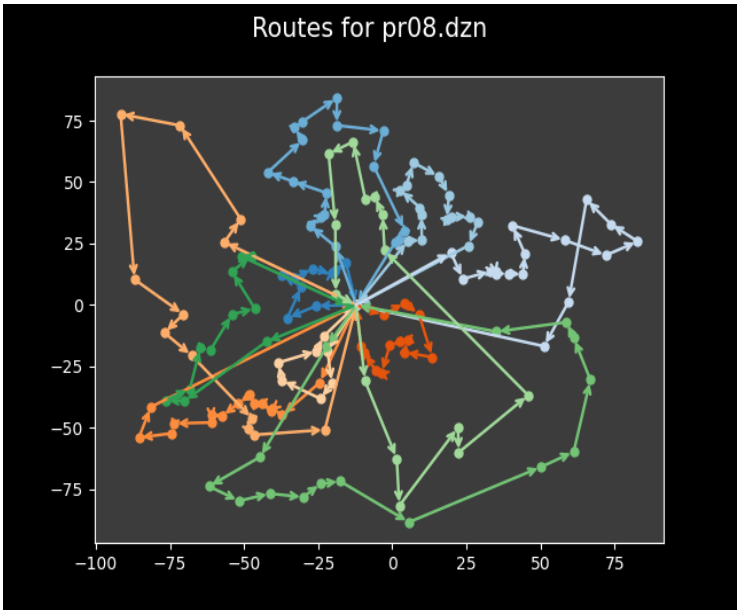

Figure 9: pr08 - sorting by customers, with luby restart

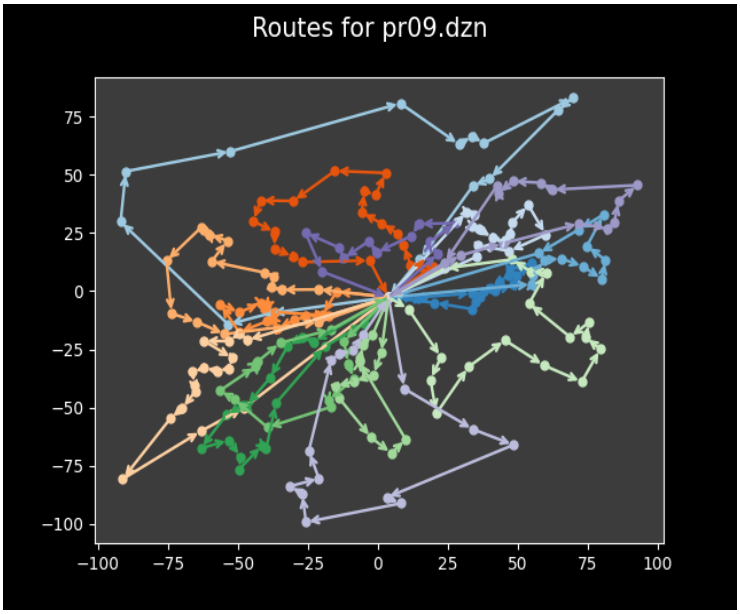

Figure 10: pr09 - sorting by customers, with luby restart

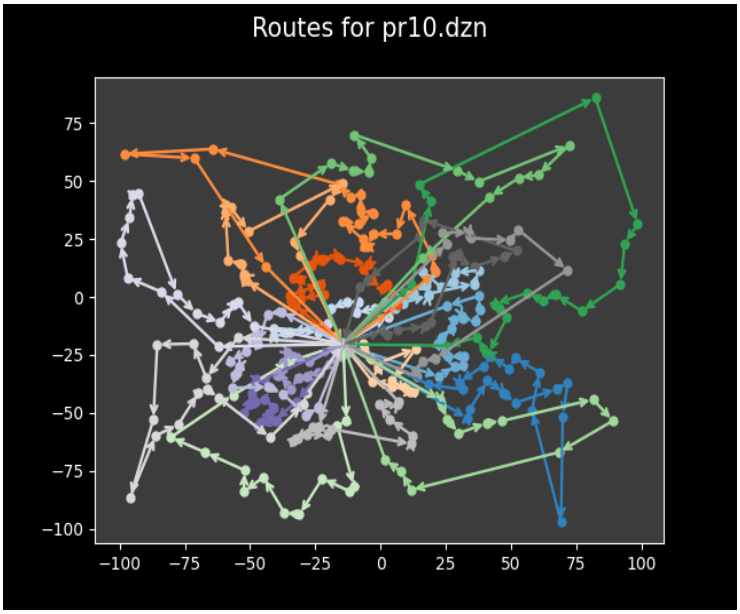

Figure 11: pr10 - sorting by customers, with luby restart

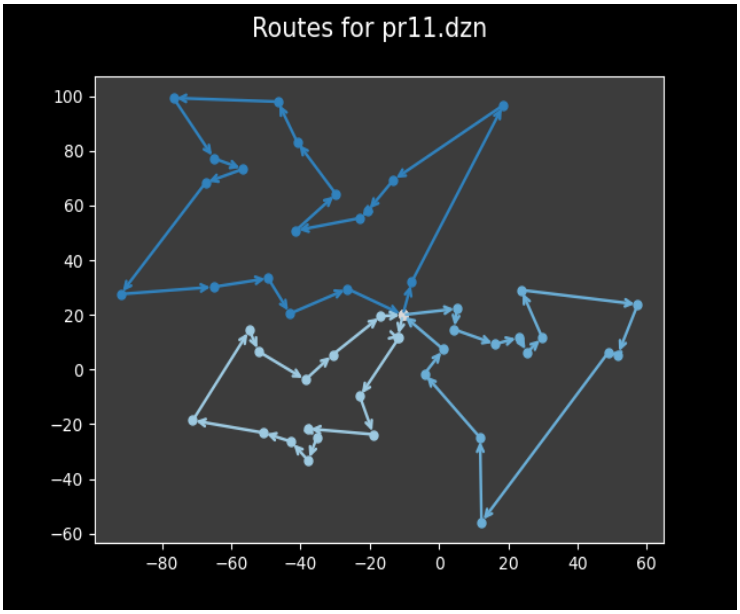

Figure 12: pr11 - sorting by customers, with luby restart