SOLVING VEHICLE ROUTING PROBLEM WITH CONSTRAINT PROGRAMMING

MARIUSZ ZAKRZEWSKI

CONTENTS

[1] *University of Bologna, Department of Computer Science and Engineering*

1 INTRODUCTION

The document describes the experimental study for solving vehicle routing problem with Constraint Programming. For experiments with the problem two CP solvers were used: Minizinc[1] and Choco[4]. Different heuristics were used to experiment with the model including Large Neighbourhood Search.

In the first chapter both Constraint Programming and vehicle routing problem are explained. After that, the model which describes problem is explained, after that performed experiments and their results are presented.

2 BACKGROUND

2.1 Constraint Programming

Constraint Programming is a paradigm for solving decision problems. It enables to describe the decision problem to be solved - in a declarative way and use third-party solver to solve the problem. In this paradigm, solving the problem is achieved by describing it declaratively and providing the model to the solver. [5]

The decision problem is defined by a set of variables, each with it's domain, a set of constraints between these variables and an objective to be satisfied. Optionally we can define heuristics to enable better variable or value selector, restarting policies, and more. A constraint is a relation between one or more variables of a model. It defines conditions over these variables that must be respected in a solution.

The solver works in the following way: it selects a variable, then it assigns the variable a value from it's domain, then solver checks constraints to shrink the remaining search space as much as possible (a process called "propagation"), and repeats. When it appears that legal solution cannot be constructed from the remaining search space (in other words it is not possible to assign a value to variable without breaking any constraints), the solver performs a "backtrack" - the solver comes back to the most recently instantiated variable that still has alternative values and assigns next value from its domain to it.

In this process, the solver constructs a search tree. If the search tree was explored exhaustedly - the solver either finds all legal solutions to the problem, or proves that problem is unsatisfiable.

Definition 1. A Constraint Satisfaction Problem is a triple <X,D,C> where:

- X is a set of decision variables $\{X_1, \ldots, X_n\}$;
- D is a set of domains $\{D_1, \ldots, D_n\}$ for X:
- D_i is a set of possible values for X_i
- C is a set of constraints $\{C_1, \ldots, C_m\}$:
- C_i is a relation over $X_j, \ldots, X_k$ $C_i(X_j, \ldots, X_k)$
- C_i the set of combination of allowed values $C_i \subseteq D(X_j) \times \cdots \times D(X_k)$

Constraint Programming can also be applied for optimization problems - maximizing or minimizing an objective function value. In this case the solver performs a backtracking search, as described before, but when a first solution is found (or any better solution) - it posts another constraint for the objective function - stating that its value must be greater (for maximizing, or less for minimizing) than just found solution. In this case, if the search tree was explored exhaustedly - the solver either finds the optimal solution to the problem, or proves that problem is unsatisfiable.

GAC (Generalized Arc Consistency) is a form of inference which detects inconsistent partial assignment.

Definition 2. A constraint C is GAC if and only if: for all X i in $\{X_1, \ldots, X_k\}$, for all $v \in D(X_i)$, v belongs to a support.

A support is every allowed tuple for a given constraint.

GAC constraint gives a set of tuples which are allowed combination in the partial assignment.

Another form of detecting inconsistent partial assignment is BC (Bounds Consistency). BC is defined only for totally ordered domains. BC concludes that since the constraint is valid for min(X) and max(X) for a given domain X, it must be also valid for a range [min(X), max(X)].

Definition 3. Constraint $C(X_1 ,..., X_k)$ is BC if and only if: $\forall X_i$ in $\{X_1,..., X_k\}$, $\min(X_i)$ and $\max(X_i)$ belong to a bound support.

BC is weaker than GC in the meaning that there are cases where GAC will detect inconsistency and BC will not. On the other hand, achieving BC is often cheaper than achieving GAC.

Constraint programming solver, as described above, is the most basic form of search. Apart from that, many heuristics can be used to alter the search process through the solution space, including: order of variable selection, order of value selection, randomizing decisions, learning from past solutions or failures or restarting the search tree construction, and more. Those heuristics might become very important when solving a big problem - which cannot be searched exhaustedly in a reasonable time on the available hardware setup. In such cases - the goal is not to find all solutions, or the optimal solution, or prove that solution does not exist, but the goal is to find a good solution in a given time. For this task - the search space must not be searched sequentially, and heuristcs like restarting and randomization might be used.

Restart means stopping the current tree search, then starting a new tree search from the root node. Restarting makes sense only when coupled with randomized dynamic branching strategies ensuring that the same enumeration tree is not constructed twice.

Large Neighbourhood Search (LNS) is a strategy used to complex decision problems, and is often used in Constraint Programming. The method relies on fixing a randomly selected set of variables, and performing another search for the remaining part of the variables. The rationale behind this method is that good solutions often are highly corelated, and LNS helps to find number of "neighbour" solutions with very low cost which might result in finding next best solution. [6]

In this work, two solver were used to tackle the vehicle routing problem: Minizinc and Choco. Minizinc is an open-source project containing a formal language to describe the problem. Minizinc then translates the problem to the underlying solvers and runs them. Choco is an open-sourced library for Constraint Programming in Java.

2.2 Vehicle routing problem

Vehicle routing problem is an optimization problem in which a total commuted distance is minimized while ensuring every customer is satisfied. [2]

In the problem we have a depot (with a given location), a fleet of vehicles, each with a capacity, and a number of customers, each with a demand for a resource and a given location. Vehicle travels from depot to a number of customers in a particular order, and goes back to depot. Sum of customers demands must not exceed the vehicles capacity. The goal is to assign a route for every vehicle so that every customer is satisfied while minimizing the objecting function.

The objective function is a weighted sum of total distance and number of vehicles being used.

This work solves the problem without time-windowing.

3 MODELLING VRP IN CP

In this chapter, the model used for solving vehicle routing problem is described along with few examples of problem instances with found solutions.

The first step in solving the problem is loading input. Below, there is a list of variables consisting of problem input along with explanations.

N Number of customers.

NUMVEHICLES Number of vehicles available. In the final solution, not every vehicle has to be used.

CAPACITIES Array of each vehicle capacity; i-th value in the array is the i-th vehicle capacity.

DEMAND Array of each customer demand; i-th value in the array is the i-th customer demand.

LOCX Array of each customer x coordinate location; i-th value in the array is the i-th customer location

LOCY Same as locX, but regarding the y coordinate

DEPOTX X coordintate of location of the depot.

DEPOTY Same as DepotX, but regarding the y coordinate.

After loading the input data, the models variables are defined. The solution consists of two array variables:

ORDER An order in which customers are visited. i-th value is the order in which i-th customer is visited.

VEHICLE_TO_CUSTOMER Which of vehicle will handle given customer. i-th value is the vehicle which will handle i-th customer. vehicle_to_customer[i] = j <-> i-th customer will be handled by j-th vehicle

Variable *order* is an array of *n* variables taking values from 1 to *n*. Because each customer has to be visited exactly once, the *alldifferent* constraint has been posted on variable *order*.

Variable *vehicle_to_customer* is an array of *n* variables taking values from 1 to *NumVehicles*. It is possible that a given value v: 1 < v < n will not appear in the *vehicle_to_customer* array. Such a situation means that given vehicle *v* is not used in particular solution. Therefore is is also possible to minimize the number of vehicles used in a solution.

Customers are visited in order with respect to the *order* variable by consecutive vehicles. This means that customers $c_1, c_2, \dots, c_k$ are visited by a given vehicle *v* if and only if vehicle_to_customer[c_1] = v and vehicle_to_customer[c_2] = v, ..., vehicle_to_customer[c_k] = v. Customers $c_1, c_2, \dots, c_k$ are visited in order given by the *order* array.

Below, the example problem instance is given along with solution found by solver. Interpretation of the solution is presented as a graph and explained.

Example problem instance:

```
Demand = [7, 4, 8, 2, 5, 3];
NumVehicles = 4;
Capacities = [8 8 8 8];
```

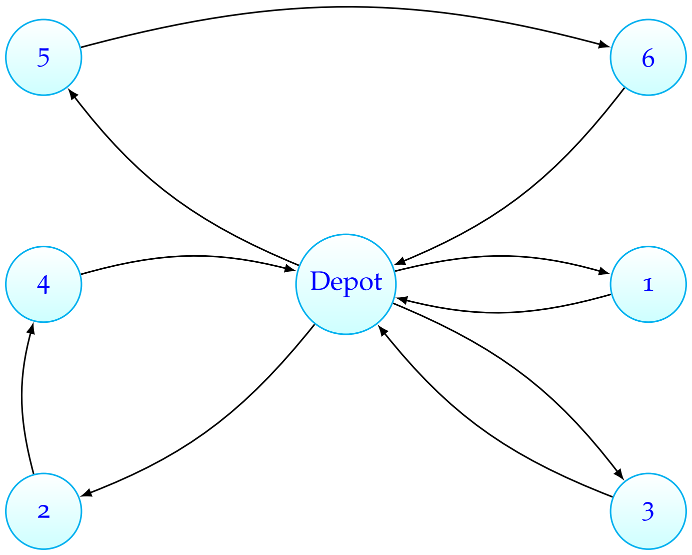
Solution found by solver:

```
vehicle_to_customer = [4, 2, 3, 2, 1, 1]
order = [5, 6, 1, 3, 2, 4]
total_distance = 106
objective_function = 40106
used_vehicles = 4
```

Array variable *vehicle_to_customer* represents which vehicle will go to given customer. *vehicle_to_customer[1]* in the example is equal to 4, which means that customer 1 will be handled by vehicle 4. *vehicle_to_customer[3]* = 3 means that the customer 3 will be handled by vehicle 3. As we can see, customers 5 and 6 will be handled by vehicle 1, and customers 2 and 4 will be handled by vehicle 2.

Now, to interpret the order of visiting of customers, for a given vehicle we look at the *order* array variable. Customers 5 and 6 are visited by the same vehicle, and because *order[*x_1*]* = 5, *order[*x_2*]* = 6, and $x_1 < x_2$ the 5th customer will be visited before 6th customer. In the same way: *order[6]* = 4, and *order[5]* = 2, because 5 < 6, therefore customer 2 will be visited before customer 4.

Below, there is a visual interpretion of obtained solution.



To enforce that no vehicle exceeds its capacity while visiting its customer, for every vehicle v, the sum of demands of its customer is calculated, and constraint that aforementioned sum is lower or equal than the vehicles capacity is posted: $\forall(v \in 1, 2, \ldots \mathrm{NumVehicles})(\sum_{c}^{customers(v)} demand(c) <= capacity(v))$ where $customers(v) = \{\, i \,|\, 1 < i < n \; and \; vehicle\_to\_customer[i] = v\}$

Each customer is visited by exactly one vehicle and therefore it is ensured that no customer will be visited more than once, also every customer is visited exactly once.

To calculate the objective function value, it is necessary to calculate total distance of all vehicles and to calculate number of vehicles used. Below, there is an explanation how objective function is calculated.

Variable array *vehicle_to_customer* holds identifiers of all used vehicles in a given solution. Therefore number of distinct values if equal to the number of vehicles used. Variable *used_vehicles* means number of vehicles being used in a given solution. It is calculated in the following way: *nvalue(used_vehicles, vehicle_to_customer)*, where *nvalue* requires that the number of distinct values provided as second argument is equal to the first argument.

Calculating total distance is achieved in by iterating over the *order* variable. If two subsequent customers are handled by the same vehicle - the distance between those two customers is added to the final result. If they are handled by different vehicle - the distance between depot to the first customer plus the distance between second customer to the depot is added to the final result.

4 EXPERIMENTAL STUDY

4.1 Experimental Setup

In experiments two projects were used to run implemented models: Minizinc and Choco. Every experiment was run on a personal laptop with an economical hardware components. The hardware used for experiments was a 8th generation Intel Core i5 with 8 Gigabytes of RAM (Random Access Memory). Because of how Java Virtual Machine performs runtime optimizations[3], every experiment using Choco has been proceeded by a 20 seconds warmup of JVM.

To test performance of implemented model and used heuristics 10 problem instances were available, ranging from n = 47 customers, to n = 247 customers.

4.2 Results

In the experiments, two solver were used: Minizinc (with Gecode solver) and Choco. The model has been implemented in two solver and the experiments constisted of three scenarios of used heuristics. In the first scenario only the *dom_w_deg* heurisitic was used for variable selection. The heuristics chooses the variable with the smallest value of domain size divided by weighted degree, which is the number of times it has been in a constraint that caused failure earlier in the search. In the second scenario, the *dom_w_deg* heuristic was used along with luby restart. Restarting is a heurisitic for stopping the solver and starting it again from the begging after using a number of resources (such as time of process or failing to many times). Luby restart means restarting the search after using number of resources multiplied by subsequent number in the luby sequence. In the third scenario, the *dom_w_deg* heuristic was used along with luby restart and Large Neighbourhood Search (LNS).

All of 10 instances of the problem were run in both solvers and in all three heuristics. Every setup (a triple of: problem instance, solver, used heuristics) was run for 3 minutes on aforementioned hardware setup and objective function value has been recorded after the run. The gathered results are presented in the table below. For every instance - the best objective function value is bolded.

Table 1: Results for Choco solver

Problem Instance \Heuristics used	domwdeg	domwdeg + restarts	domwdeg + restarts + lns
1	43,362	**43,142**	43,537
2	78,811	77,988	**77,966**
3	114,610	**112,712**	112,787
4	156,891	155,193	**155,015**
5	208,301	**206,810**	207,466
6	unknown	unknown	unknown
7	57,717	**56,742**	57,435
8	123,619	**122,697**	122,940
9	183,324	**181,762**	182,004
10	unknown	unknown	unknown

Table 2: Results for Minizinc (Gecode)

Problem Instance \Heuristics used	domwdeg	domwdeg + restarts	domwdeg + restarts + lns
1	43,735	43,333	*43,231*
2	198,034	*187,854*	187,734
3	215,678	205,789	*205,583*
4	218,616	218,367	*218,000*
5	*209,807*	219,568	219,491
6	unknown	unknown	unknown
7	206,992	176,650	*166,554*
8	214,573	*204,452*	204,551
9	*182,631*	222,028	222,076
10	unknown	unknown	unknown

Instances 6 and 10 did not follow the necessary conditions to be feasible. In those instances the sum of demands for all customers was larger them sum of all vehicles capacities. Both solvers were not able to detect this situation for the presented model. In all cases solver yielded no result (they were not able to prove the unfeasibility of the problem). As can be seen in above tables, the Choco solver was generally able to find better solution for a particular model. Regardless of the backend solver, used heuristics were not able to improve the result very significantly. There was however a minor improvement between only *dom_w_deg* heuristic and a combination of also restaring or LNS.

5 CONCLUSIONS

In the document Constraint Programming paradigm is explained. Also, it is exemplified on the vehicle routing problem along with description of the model used, and applied heuristics. The model was tested and experimented with. Different heuristics were used to solve vehicle routing problem and the results were presented.

REFERENCES

[1] Minizinc - constraint modelling language. https://www.minizinc.org/.

[2] Gilbert Laporte. The vehicle routing problem: An overview of exact and approximate algorithms. *European journal of operational research*, 59(3):345–358, 1992.

[3] David Lion, Adrian Chiu, Hailong Sun, Xin Zhuang, Nikola Grcevski, and Ding Yuan. Don't get caught in the cold, warm-up your {JVM}: Understand and eliminate {JVM} warm-up overhead in data-parallel systems. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 383–400, 2016.

[4] Charles Prud'homme, Jean-Guillaume Fages, and Xavier Lorca. *Choco Solver Documentation*. TASC, INRIA Rennes, LINA CNRS UMR 6241, COSLING S.A.S., 2016. URL http://www.choco-solver.org.

[5] Philippe Refalo. Impact-based search strategies for constraint programming. In *International Conference on Principles and Practice of Constraint Programming*, pages 557–571. Springer, 2004.

[6] Paul Shaw. Using constraint programming and local search methods to solve vehicle routing problems. In *International conference on principles and practice of constraint programming*, pages 417–431. Springer, 1998.