CSP Solver and Capacitated Vehile Routing Problem

Thinh D. Nguyen

Abstract

In this paper, we present several models for Capacitated Vehicle Routing Problem (CVRP) using Choco solver. A concise introduction to the constraint programming methods is included. Then, we construct two models for CVRP. Experimental results for each model are given in details.

Keywords: CSP, CVRP, models

1. Introduction

Constraint programming is an efficient tool to solve combinatorial optimization problems, it integrates many results in a wide range of disciplines such as artificial intelligence, operation research, algorithms, graph theory, etc. The basic idea is to state a model (including variables, constraints) that can be read by a constraint programming problem (CSP) solver. After that, the CSP solver can read input data and process to produce output for the problem. A constraint is a relation among the variables, a CSP is a set of constaints that needs to be satisfied by an assignment to variables. In particular, a CSP includes a set of variables each of which can take value from its domain, a set of constraints each of which can be a relation on some subset of variables. For examples, in the exam scheduling problem for a university, the variables could be time and place of different subjects, constraints should be capacity of each exam room (number of student in each room must not exceed the capacity of that room). There should be also constraints stating that two subjects with the same time share no student. A CSP solver can read a model for a practical problem as this one and search for an assignment that assigns values to all variables to satisfy all the constraints satated in the model.

A solver systematically searches the solution space using well-known techniques such as backtracking, branch and bound, local search heuristics or some variants of these. In some cases, it may fail to produce a solution.

Some frequent solvers integrates both search methods and inference methods. Among inference techniques, propagation ones are the most commonly used. In practice, inference helps narrow down the search space a lot.

While most of CSP problems are NP-complete, some sub-classes possesses polynomial-time algorithms. These sub-classes are characterized by their structure connecting common variables shared by two or more constraints or by languages defining those constraints. Typical examples are tree-based CSP in graph theory.

A good model can not be obtained straightforwardly by intuitive view of the problem at hand. In this paper, we want to emphasize the importance of carefully built models for a CSP through CVRP - a problem in combinatorial optimization with extensive research. [1] is a nice textbook of constraint programming.

2. Choco solver

In this section, we will introduce readers to the Choco solver through a simple example.

n-queen problem

In this problem, we will try to place n queens in a generalized nxn chessboard. A queen can attack other queens in the same row, column, diagonals. The problem asks whether it is possible to place n queens in such a way that no two queens can attack each other.

In the framework of CSP, we can model this problem as follows:

Variables: $X = \{x_i | 1 \leq i \leq n\}$

Domains: Each variable x_i has its domain $D_i = \{1, 2, ..., n\}$

Constraints: Set of constraints can be grouped into two categories

- No two queens are allowed on the same row: $\{x_i \neq x_j | i \neq j\}$
- No two queens are allowed on the same diagonal: $\{x_i \neq x_j + i - j | i, j \in [1, n], i \neq j\}$, $\{x_i \neq x_j + j - i | i, j \in [1, n], i \neq j\}$

Choco program for n-queen problem:

```
int nbQueen = 8;

CPModel m = new CPModel();
```

```
IntegerVariable[] queens = Choco.makeIntVarArray("Q", nbQueen, 1,
    nbQueen);

for (int i = 0; i < nbQueen; i++) {
   for (int j = i + 1; j < nbQueen; j++) {
      int k = j - i;
      m.addConstraint(Choco.neq(queens[i], queens[j]));
      m.addConstraint(Choco.neq(queens[i], Choco.plus(queens[j],
          k)));
      m.addConstraint(Choco.neq(queens[i], Choco.minus(queens[j],
          k)));
   }
}

CPSolver s = new CPSolver();
s.read(m);
s.solveAll();

System.out.println("Number of solutions
    found:"+s.getSolutionCount());
```

For a user manual of Choco solver, the reader is addressed to [2].

3. Using Choco to solve CVRP

3.1. Problem formulation

In this problems, there are n customers on the 2D plane. Everyday, a delivery company has to serve these customers. Each customer i informs the company about its location (x_i, y_i), its demand $demand_i$ of goods in kilograms. The company has a depot also located on the plane at (x_0, y_0). There are totally K vehicles in company's fleet, each with the maximum capacity of C kilograms. Starting at the depot, each vehicle is loaded with goods before it goes out on a tour to serve the customers on that tour before getting back to the depot.

Given these information, the company has to use some of their vehicles, and assigns each vehicle to a tour on the plane. A feasible solution is one that serves all the customers while conforms to the capacity constraint put on each vehicle. The problem asks for an optimal solution where the objective function is the total distance traveled by the fleet.

3.2. Two models for CVRP

Model 1

Variables:

For each k, i, j: $\mathit{flow}\,[k]\,[i]\,[j]$ decides if on its assigned tour, vehicle k visits customer j after customer i

For each k, i: $\mathit{cumulativeDemand}\,[k]\,[i]$ represents the cumulative sum of customers' demands up to customer i on the tour of vehicle k

Domains: Every flow variable has its domain as $\{0, 1\}$. Others have $[0, C]$ as their domains.

Constraints:

Each tour is a cycle: for each k,
$\sum_{j=0}^{N} \mathit{flow}\,[k]\,[i]\,[j] = \sum_{j=0}^{N} \mathit{flow}\,[k]\,[j]\,[i]$, for each i
$\sum_{i=0}^{N} \mathit{flow}\,[k]\,[0]\,[i] = 1$

All the fleet start at depot: for each k,
$\mathit{cumulativeDemand}\,[k]\,[i] = 0$

These constraints requires the values of the cumulative variables to be according to the tours: for each k,

For every $i \neq j$ where $j \neq 0$, if $\mathit{flow}\,[k]\,[i]\,[j] = 1$, then $\mathit{cumulativeDemand}\,[k]\,[j] = \mathit{cumulativeDemand}\,[k]\,[i] + \mathit{demand}\,[j]$

Each customer is visited by exactly one vehicle:
$\sum_{k=0}^{K-1} \sum_{j=0}^{N} \mathit{flow}\,[k]\,[i]\,[j] = 1$

Model 2

To improve on the number of variables, this model uses two-dimensional array of variables in stead of three-dimensional array as in previous model. In particular, we place all the tour into one two-dimensional flow. In the previous model, the inefficiency comes from allocating for each tour a separate 2D flow. The flow property for each customer stays the same. But, instead of K constraints each of which stating that the out-flow of the depot w.r.t to each vehicle does not exceed 1, we now have only one constraint stating that the out-flow of the depot, in total, does not exceed K.

3.3. Choco program for CVRP

We only provide the program corresponding the more efficient model 2:

```java
import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;

import choco.Choco;
import choco.Options;
import choco.cp.model.CPModel;
import choco.cp.solver.CPSolver;
import choco.kernel.model.variables.integer.IntegerExpressionVariable;
import choco.kernel.model.variables.integer.IntegerVariable;
//import choco.kernel.model.variables.real.RealExpressionVariable;
//import choco.kernel.model.variables.real.RealVariable;

public class ChocoCVRPFast {
    static int n, k, capacity;
    static int[] x, y, demand;
    static final int INF = 1000000;

    static CPModel m;
    static IntegerVariable[][] flow;
    static IntegerVariable distSum;

    public static void readData() throws FileNotFoundException {
        Scanner sc = new Scanner(new File("cvrp9.txt"));

        // Number of customers
        n = sc.nextInt();
        System.out.println("Number of customers: "+n);
        // Number of vehicles
        k = sc.nextInt();
        System.out.println("Number of vehicles: "+k);
        // Capacity
        capacity = sc.nextInt();
        System.out.println("Capacity: "+capacity);

        // Read coordinates and demand of each customer
        x = new int [n + 1];
        y = new int [n + 1];
        demand = new int [n + 1];
        for (int i = 0; i <= n; i++) {
           System.out.print("Customer "+i+" ");
           int ii = sc.nextInt(); if (ii != i)
System.out.println("Input file "+i+"th customer");
           x[i] = sc.nextInt();
           System.out.print(x[i]+" ");
           y[i] = sc.nextInt();
           System.out.print(y[i]+" ");
           demand[i] = sc.nextInt();
           System.out.println(demand[i]+" ");
        }

        sc.close();
     }

     public static int dist(int i, int j) {
        return (int) ( Math.round(1000 *
Math.sqrt((x[i]-x[j])*(x[i]-x[j])+(y[i]-y[j])*(y[i]-y[j]))) );
     }

     public static void stateModel() {
        // Create the model
        m = new CPModel();

        // Create the variables
        flow = new IntegerVariable [n + 1] [n + 1];
        for (int i = 0; i < n + 1; i++) {
           for (int j = 0; j < n + 1; j++) {
              flow[i][j] = Choco.makeIntVar("flow_"+i+"_"+j, 0,
1);
           }
        }

        // Transpose of flow
        IntegerVariable[][] flowCol = new IntegerVariable [n +
        1] [n + 1];
        for (int i = 0; i < n + 1; i++) {
           for (int j = 0; j < n + 1; j++) {
              flowCol[i][j] = flow[j][i];
           }
        }

        IntegerVariable[] cumulativeDemand = new IntegerVariable[n + 1];
        for (int i = 0; i < n + 1; i++)
           cumulativeDemand[i] = Choco.makeIntVar("cumulativeDemand_"+i, 0, capacity);

        // Post constraints
        // Each route is a cycle
        for (int i = 0; i < n + 1; i++) {
           m.addConstraint(Choco.eq(Choco.sum(flow[i]), Choco.sum(flowCol[i])));
        }
        m.addConstraint(Choco.leq(Choco.sum(flow[0]), k));

        // Start at depot
        m.addConstraint(Choco.eq(cumulativeDemand[0], 0));

        // Each customer visited by one vehicle
        for (int i = 1; i < n + 1; i++) {
           m.addConstraint(Choco.eq(Choco.sum(flow[i]), 1));
        }

        // Cumulative sum according to route
        for (int i = 0; i < n + 1; i++)
           for (int j = 1; j < n + 1; j++)
              m.addConstraint(Choco.implies(Choco.eq(flow[i][j], 1), Choco.eq(cumulativeDemand[j], Choco.plus(cumulativeDemand[i], demand[j]))));

        // Objective function
        distSum = Choco.makeIntVar("distSum", 0, INF, Options.V_OBJECTIVE);
        IntegerExpressionVariable[] distSumI = new
```

```
IntegerExpressionVariable [n + 1];
    for (int i = 0; i < n + 1; i++) {
      IntegerExpressionVariable[] coeff = new
IntegerExpressionVariable [n + 1];
      for (int j = 0; j < n + 1; j++)
        coeff[j] = Choco.mult(dist(i, j), flow[i][j]);

      distSumI[i] = Choco.sum(coeff);
    }
    m.addConstraint(Choco.eq(distSum, Choco.sum(distSumI)));
  }

  public static void solve() {
    // Create the solver
    CPSolver s = new CPSolver();
    s.read(m);
    //s.setTimeLimit(1000000);
    s.minimize(s.getVar(distSum), true);

    // Print the solution found
    boolean[] visited = new boolean[n + 1]; visited[0] =
true;
    for (int kk = 0; kk < k; kk++) {
      System.out.print("Route "+kk+": ");

      int currentVertex = 0;
      System.out.print(currentVertex+" ");
      do {
        for (int i = 0; i < n + 1; i++)
          if (s.getVar(flow[currentVertex][i]).getVal()
== 1 && ((i == 0) || visited[i] == false)) {
            System.out.print(i+" ");
            currentVertex = i; visited[i] = true;
            break;
          }
      } while (currentVertex != 0);

      System.out.println();
    }

```

```java
        for (int i = 0; i < n + 1; i++) {
            for (int j = 0; j < n + 1; j++) {
                System.out.print(s.getVar(flow[i][j]).getVal()+" ");
            }
            System.out.println();
        }

        System.out.println(s.isFeasible());
        System.out.println(s.getVar(flow[0][0]).getVal());
        System.out.println(s.getVar(distSum).getVal()*0.001);
    }

    public static void main(String[] args) throws FileNotFoundException {
        readData();
        stateModel();
        solve();
    }
}
```

4. Experimental results

The author has tested the two Choco programs on 9 instances with increasing difficulty. The running time ranges from seconds to hours, according to the instance's difficulty. If one only considers whether a model to be inefficient on an instance, then, only with instance 9, model 1 can not produce output in reasonable time. For that instance, one has to use model 2.

The input data of the smallest instance is provided here:

```
Input 1:
6 2 13
0 0 0 0
1 4 6 4
2 6 7 2
3 8 5 5
4 10 10 7
5 3 10 3
6 2 1 2
```

In this instance, we have 6 customers, 2 vehicles each with capacity of 13. The remaining information is related to customers' co-ordinates and demands.

5. Conclusion

There have been intensive ongoing research on combinatorial optimization problems. These works yield numerous solvers ranging from linear programming, interger linear programming, constraint programming, hybrid solvers, etc. In this paper, we only consider the CSP ones and use Choco as a realistic tool to cope with CVRP.

References

[1] Francesca Rossi, Peter van Beek, Toby Walsh. *Handbook of Constraint Programming*, 1st ed., 2006

[2] Charles Prud'homme, Jean-Guillaume Fages, Xavier Lorca. *Choco Documentation*, 2017