Optimisation Problems Collection

SOICT - HUST (IT3052E)

Manh Duc, Tran Vu

Contents

1 Introduction

This is a collection of solvers developed during the optimisation course at SOICT - HUST (IT3052E). Note that these implementations are drawn from various external resources due to the complexity of constraint programming.

2 Capacitated Vehicle Routing Problem

2.1 Problem Statement

A fleet of K identical trucks having capacity Q need to be scheduled to deliver pepsi packages from a central depot 0 to clients 1,2,. . . ,n. Each client i requests d[i] packages. The distance from location i to location j is c[i,j], $0 \leq i, j \leq n$. A delivery solution is a set of routes: each truck is associated with a route, starting from depot, visiting some clients and returning to the depot for delivering requested pepsi packages.

Constraints:

- Each client must be visited exactly once
- Total package load per truck cannot exceed capacity Q

Objective: Minimize total travel distance

2.2 I/O Format

Input:

- Line 1: n,K,Q ($2 \leq n \leq 12, 1 \leq K \leq 5, 1 \leq Q \leq 50$)
- Line 2: d[1],...,d[n] ($1 \leq d[i] \leq 10$)
- Line i+3 (i=0,. . . ,n): the ith row of the distance matrix c ($1 \leq c[i, j] \leq 30$)

Output: Minimal total travel distance.

E.g.

Input:

```
4 2 15
7 7 11 2
0 12 12 11 14
14 0 11 14 14
14 10 0 11 12
10 14 12 0 13
10 13 14 11 0
```

Output:

```
70
```

2.3 Solution Approach

Using CP, we need to formulate the constraints. For this problem, the constraints are:

2.3.1 Client visitation constraint

Each client is visited exactly once.

```
for j in range(1, n + 1):
    model.Add(sum(x[i,j,k] for i in range(n + 1) for k in
        range(K) if i != j) == 1)
```

2.3.2 Capacity constraint

Total number of packages requested by clients cannot exceed its capacity.

```
for k in range(K):
    model.Add(sum(demands[j] * sum(x[i,j,k] for i in range(n
         + 1) if i != j) for j in range(1, n + 1)) <= Q)
```

2.3.3 Flow conservation

For each node, the sent package must be exhaustive. In other words, the number of incoming packages must be equal to the outgoing packages.

```
for k in range(K):
    for j in range(n + 1):
        model.Add(sum(x[i,j,k] for i in range(n + 1) if i !=
             j) == sum(x[j,i,k] for i in range(n + 1) if i !=
             j))
```

2.3.4 Subtour elimination

No subtour (circular routes that aren't the solution)

```
    u = {}
    for i in range(n + 1):
        for k in range(K):
            u[i,k] = model.NewIntVar(0, n, f'u_{i}_{k}')

    for i in range(1, n + 1):
        for j in range(1, n + 1):
            for k in range(K):
                if i != j:
                    model.Add(u[i,k] - u[j,k] + n * x[i,j,k]
                         <= n - 1)
```

2.3.5 Objective function

The objective of this problem is to minimise the total distance

```
total_dist = sum(dist[i][j] * x[i,j,k] for i in range(n + 1)
     for j in range(n + 1) for k in range(K) if i != j)
model.Minimize(total_dist)
```

3 Balanced Courses Assisgnment Problem

3.1 Problem Statement

At the beginning of the semester, the head of a computer science department D have to assign courses to teachers in a balanced way. The department D has m teachers T=1,2,...,m and n courses C=1,2,...,n. Each teacher $t \in T$ has a preference list which is a list of courses he/she can teach depending on his/her specialization. We have known a list of pairs of conflicting two courses that cannot be assigned to the same teacher as these courses have been already scheduled in the same slot of the timetable. The load of a teacher is the number of courses assigned to her/him. How to assign n courses to m teacher such that each course assigned to a teacher is in his/her preference list, no two conflicting courses are assigned to the same teacher, and the maximal load is minimal.

3.2 I/O Format

Input:

- Line 1: contains two integer m and n ($1 \leq m \leq 10, 1 \leq n \leq 30$)
- Line i+1: contains an positive integer k and k positive integers indicating the courses that teacher i can teach ($\forall i = 1, \ldots, m$)
- Line m+2: contains an integer k
- Line i+m+2: contains two integer i and j indicating two conflicting courses ($\forall i = 1, \ldots, k$)

Output: The output contains a unique number which is the maximal load of the teachers in the solution found and the value -1 if not solution found.

E.g.

Input:

```
4 12
5 1 3 5 10 12
5 9 3 4 8 12
6 1 2 3 4 9 7
7 1 2 3 5 6 10 11
25
1 2
1 3
1 5
2 4
2 5
2 6
3 5
3 7
3 10
4 6
4 9
5 6
5 7
5 8
6 8
6 9
7 8
7 10
7 11
8 9
8 11
8 12
9 12
10 11
11 12
```

Output:

```
3
```

3.3 Solution Approach

The problem can be solved using CP. The constraints are:

3.3.1 Preferences constraint

Each course must be assigned to a teacher who has it in his/her preference list.

```
for course in range(1, n+1):
    allowed_teachers = [t for t in range(1, m+1) if course
        in preferences[t]]
    for teacher in range(1, m+1):
        if teacher not in allowed_teachers:
            model.Add(course_teacher[course] != teacher)
```

3.3.2 Conflict constraint

Two conflicting courses cannot be assigned to the same teacher.

```
for c1, c2 in conflicting
    model.Add(course_teacher[c1] != course_teacher[c2])
```

3.3.3 Boolean constraint

The main part of the problem is to assign each course to a teacher. This can be done using a boolean variable. We can define them as boolean constraints where if a teach is selected, the course is assigned to him/her. In other words, if the course is assigned to a teacher, the boolean variable is true.

```
for teacher in range(1, m+1):
    teaches_vars = []
    for course in range(1, n+1):
        teaches = model.NewBoolVar(f'teaches_{teacher}_{
            course}')
        model.Add(course_teacher[course] == teacher).
            OnlyEnforceIf(teaches)
        model.Add(course_teacher[course] != teacher).
            OnlyEnforceIf(teaches.Not())
        teaches_vars.append(teaches)
    model.Add(teacher_load[teacher] == sum(teaches_vars))
    model.Add(max_load >= teacher_load[teacher])
```

3.3.4 Objective function

The objective of this problem is to minimise the maximal load of the teachers.

```
model.Minimize(max_load)
```

4 N-Queens Problem

4.1 Problem Statement

Find a solution to place n queens on a chess board such that no two queens attack each other. A solution is represented by a sequence of variables: x[1], x[2], . . ., x[n] in which x[i] is the row of the queen on column i (i = 1, . . ., n)

4.2 I/O Format

Input:

- Line 1: contains a positive integer n ($10 \leq n \leq 10000$)

Output:

- Line 1: write n
- Line 2: write x[1], x[2], . . ., x[n] (after each value is a SPACE character)

E.g.

Input:	**Output:**
10	10 1 3 6 8 10 5 9 2 4 7

4.3 Solution Approach

The problem can be solved using CP. The constraints are:

4.3.1 Row constraint

Each row must have exactly one queen.

```
model.AddAllDifferent(x)
```

4.3.2 Diagonal constraint

No two queens can be on the same diagonal.

```
for i in range(n):
    for j in range(i + 1, n):
        model.Add(x[i] - x[j] != i - j)  # Upward diagonal
        model.Add(x[i] - x[j] != j - i)  # Downward diagonal
```

4.3.3 Objective function

The objective of this problem is to find a solution.

```
solver = cp_model.CpSolver()
status = solver.Solve(model)
```

5 Traveling Salesman Problem

5.1 Problem Statement

There are n cities 1, 2, ..., n. The travel distance from city i to city j is c(i, j), for i, j = 1, 2, ..., n. A person departs from city 1, visits each city 2, 3, ..., n exactly once and comes back to city 1. Find the itinerary for that person so that the total travel distance is minimal. A solution is represented by a sequence of n variables $x[1], x[2], \ldots, x[n]$ in which the tour is: $x[1] \rightarrow x[2] \rightarrow \cdots \rightarrow x[n] \rightarrow x[1]$.

5.2 I/O Format

Input:

- Line 1: a positive integer n ($1 \leq n \leq 200$)
- Line i+1 (i = 1, . . ., n): contains the ith row of the distance matrix x (elements are separated by a SPACE character)

Output:

- Line 1: write the value n
- Line 2: write $x[1], x[2], ..., x[n]$ (after each element, there is a SPACE character)

E.g.

Input:

```
4
0 1 1 9
1 0 9 3
1 9 0 2
9 3 2 0
```

Output:

```
4
1 3 4 2
```

5.3 Solution Approach

The problem can be solved using CP. We thus need to formulate the constraints as follows:

5.3.1 Each city must be visited exactly once

- `arcs`: a list of tuples `(i, j, lit)` where lit is a boolean variable indicating whether the person goes from city i to city j.

```
# Create variables for successor of each city
succ = [model.NewIntVar(0, n-1, f'succ_{i}') for i in range(
    n)]

# Create circuit
for i in range(n):
    for j in range(n):
        if i != j:
            lit = model.NewBoolVar(f'arc_{i}_{j}')
            model.Add(succ[i] == j).OnlyEnforceIf(lit)
            arcs.append((i, j, lit))
model.AddCircuit(arcs)
```

5.3.2 Objective function

The objective of this problem is to minimise the total distance.

- `distance_terms`: a list of variables representing the distance between each pair of cities.

```
# Calculate total distance
total_distance = model.NewIntVar(0, sum(max(row) for row in
    dist), 'total_distance')

for i in range(n):
    dist_var = model.NewIntVar(0, max(max(row) for row in
        dist), f'dist_{i}')
    distance_terms.append(dist_var)
    model.AddElement(succ[i], dist[i], dist_var)

model.Add(total_distance == sum(distance_terms))
model.Minimize(total_distance)
```