UM-SJTU Joint Institute
Problem Solving with AI Techniques
(Ve593)

Project One
Search

Ming Xingyu 517370910224

Date: 2020/10/11

Part 1. TSP

1. Solving with CP-SAT in Ortools

- **General principle:** In this part, I solve the TSP problem as a assignment problem. I define a assignment matrix named M, where the column is the *from* city and the row is the *to* city. For example, if we have $M[i][j] = 1$ according to the solver, it means we need to travel from city $i + 1$ to city $j + 1$[1]. Also, since the solver can only deal with the integers, we times the distance by 1000 and round to a integer.
- **Added Constraints:** Apart from the general constraints that each city can only be visited once; and we can only choose one city to go to each time. I used the **AddCircuit** constraint in CP-SAT. Since we need to ensure while solving we will not meet some sub-cycle in the result we obtained and the solver can only handle the linear expression.
- **Experiment:** As required by the project, I have done the experiment of the runtime test of my code, with the input file generated by *point.py*. The results are showing below.

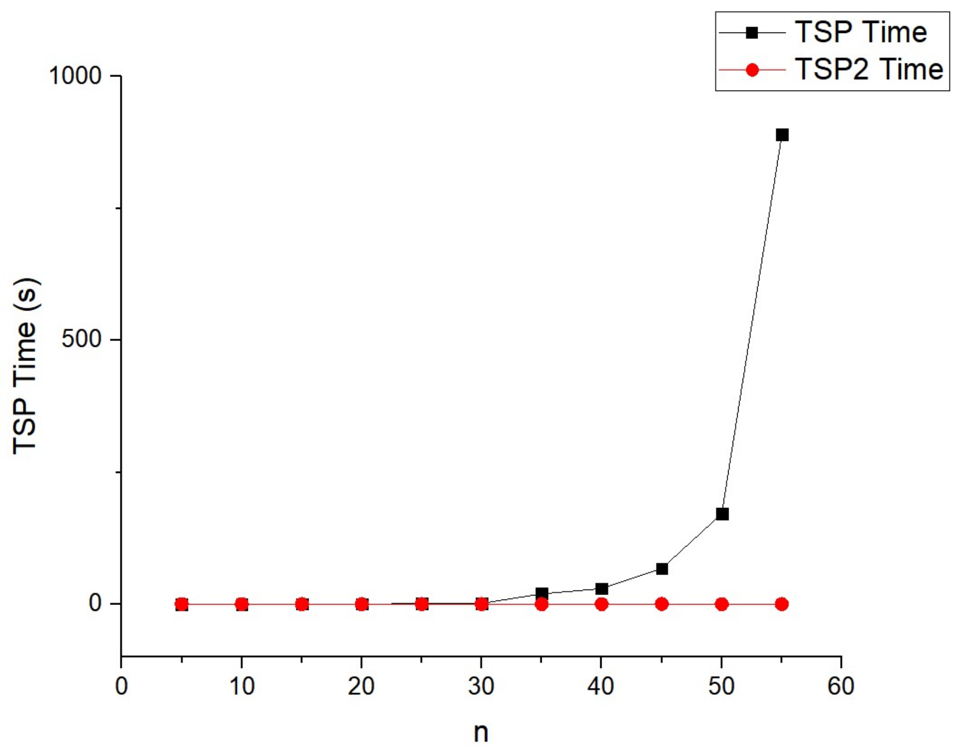

Figure 1: Time cost by TSP and TSP2.

Based on the time plot above, it can be conclude that the time cost for the code implement with CP-SAT is exponantial, while the one with Routing model is amazingly

[1]the plus one is because of the we numbering from 0.

a constant. To explain this we need to find the mathematical model of Routing model, which I failed to found online. But I guess the Routing model is a heuristic model, since some input with more number is slower than the ones with less number.

2. Solving with the constraint-solver in Ortools

Referring to the online guide of Or-tools[2], I used the the routing model to solve the problem.

Part 2. TimeTabling

- **General principle:** Since we need to print at most l solutions according to the project requirement, it is intuitively to define the callback functions, which will be called as soon as a feasible solution is found. However, since the problem can be seperated into two, which are, firstly, assign the courses to instructors and secondly assign the lecture to a proper day. It is trivial that these two problems are independent, and the solution can be any combination of them. Hence, we define two model and two callback functions respectively. The following is the callback functions with explanations I defined.

```python
        # define the callback for arranging instructors to courses
class Solution1Register(cp_model.CpSolverSolutionCallback):  # save each feasible solution to Sol1
    def __init__(self, variables, l, instr, cour, Sol1):  # initialization of the object
        ...

    def on_solution_callback(self):  # function called during each successful search
        if (self.__solution_count >= self.__l):
            self.StopSearch()  # solutions enough
            return
        sol = []
        self.__solution_count += 1
        for c in self.__cour:
            for i in self.__instr:
                if (self.BooleanValue(self.__M[i - 1][c - 1])):  # if instructor i is assigned to course c
                    sol.append((c, i))
        self.__Sol1.append(sol)
```

[2] https://developers.google.cn/optimization/routing/tsp

```python

    def solution_count(self):  # number of solutions
        """..."""


# define the callback for arranging courses to days
class Solution2Register(cp_model.CpSolverSolutionCallback):  # save each feasible solution to Sol2
    def __init__(self, variables, l, cour, Sol2):  # initialization of the object
        ...

    def on_solution_callback(self):  # function called during each successful search
        if (self.__solution_count >= self.__l):
            self.StopSearch()  # solutions enough
            return
        sol = []
        self.__solution_count += 1
        for _d in range(5):
            for c in self.__cour:
                if (self.BooleanValue(self.__M[_d][c - 1])):  # if course c is arranged on day _d+1
                    sol.append((_d + 1, c))
        self.__Sol2.append(sol)

    def solution_count(self):  # number of solutions
        ...
```

With $Sol1$ and $Sol2$ defined in the main function, I can save the arrangement of the two problems in the list.

- **Model of the problem:** The model of the two problems can be expressed in the following mathematical expressions.
  - Notation:
    Assignment Matrix,**M1** and **M2**, where $M_{ij} = 1$ means course i is assigned to instructor j/day j
    instructor list **I**
    course list **CA**

the course list of each instructor's ability is $\mathbf{C}$, where courses that instructor i can teach is in $C[i]$
the day list $\mathbf{D}$.

- **Model1:**
  Find all feasible $M1$, w.r.t.

  $M1[c][i] = 0$, if $c \notin C[i]$

  $\forall c \in \mathbf{CA}$, $\sum_{i=0}^{|\mathbf{I}|-1} M1[c][i] = 1$

  $\forall i \in \mathbf{I}$, $\sum_{c=0}^{|\mathbf{CA}|-1} M1[c][i] <= mC$

- **Model2:** Find all feasible $M2$, w.r.t.

  $\forall c \in \mathbf{CA}$, $\sum_{d=0}^{|\mathbf{D}|-1} M2[c][d] <= mD$

  $\forall d \in \mathbf{D}$, $\sum_{c=0}^{|\mathbf{CA}|-1} M2[c][d] = mL$

The code for my model is shown as following.

```python
"""define model1"""
    model1 = cp_model.CpModel()
    Sol1 = []
    M1 = []
    for i in instr:
        M_v = []
        for c in cour:
            if (c not in C[i - 1]):  # if instructor i cannot teach course c
                # M_v.append(model1.NewIntVar(0,0,'C%i, In%i'%(c,i)))
                M_v.append(0)
            else:
                M_v.append(model1.NewBoolVar('C%i, In%i' % (c, i)))
        M1.append(M_v)

    """add constrains to model1"""
    # each course can only be taught by one instructor
    for c in cour:
```

```
        model1.Add(cp_model.LinearExpr.Sum(M1[i - 1][c - 1] for i in instr) == 1)

    # each instructor can teach at most mC courses
    for i in instr:
        model1.Add(cp_model.LinearExpr.Sum(M1[i - 1]) <= mC)

    """define model2"""
    model2 = cp_model.CpModel()
    Sol2 = []
    M2 = []
    for d in days:
        M_v = []
        for c in cour:
            M_v.append(model2.NewBoolVar('Day%i, C%i' % (d, c)))
        M2.append(M_v)

    """add constrains to model2"""
    # each day can have at most mD lectures
    for _d in range(5):
        model2.Add(cp_model.LinearExpr.Sum(M2[_d]) <= mD)

    # each course has exactly mL lectures
    for c in cour:
        model2.Add(cp_model.LinearExpr.Sum(M2[_d][c - 1] for _d in range(5)) == mL)
```

- **Solution Printer:** In order to print at most *solutions*, in the callback functions, I set a limit of solutions as l. Since if we have l solution for one model and 1 solution for another, we can have l solutions after combination. To implement the combination step, I simply use several for loops and some condition statement. The whole solution printer function is shown below.

```
def printsol(Sol1,Sol2,l,mL):
    num=min(l,len(Sol1)*len(Sol2))
    n=0;
    for i in range(len(Sol1)):
        for j in range(len(Sol2)):
            if (n==num):break
            print("Solution %i"%n)
```

```
        n+=1
        for s in Sol1[i]:
            print("Course %i, Instructor %i,"%s,end=" ")
            d=0
            for t in Sol2[j]:
                if(t[1]==s[0]):
                    print(t[0],end="")
                    d+=1
                    if (d==mL):
                        print(end="")
                        break
                    print(",",end=" ")
            print()
```

Appendix

TSP.py

```
import sys
from ortools.sat.python import cp_model
import time

def str2pos(s):
    p=[0,0]
    for i in range(len(s)):
        if(s[i]==","):
            p[0]=int(s[0:i])
            if(s[len(s)-1]=="\n"):
                p[1]=int(s[i+1:len(s)-1])
            else:
                p[1]=int(s[i+1:len(s)])
    return p


def distance(p1,p2):
    x=pow(p1[0]-p2[0],2)
    y=pow(p1[1]-p2[1],2)
    dis=pow(x+y,0.5)
    return round(dis*1000)


def main():
    filename="in11.txt"
    with open(filename, "r") as f:
        data = f.readline()
    CityCount = int(data)
    pos_str = []
    with open(filename, "r") as f:
        for line in f.readlines():
            pos_str.append(line)
    pos = []
    for i in pos_str[1:]:
        pos.append(str2pos(i))
```

```
    dis_mat = []
    for i in pos:
        for j in pos:
            dis_mat.append(int(distance(i, j)))
    #define model
    model = cp_model.CpModel()
    M = [model.NewIntVar(0, 1, '%i to %i' % (i, j)) for i in range(CityCount) for j in range(CityCount)]
    arc = []
    for i in range(CityCount):
        for j in range(CityCount):
            if (j == i): continue
            arc.append([i, j, M[i * CityCount + j]])

    #add constrains
    for i in range(CityCount):
        model.Add(cp_model.LinearExpr.Sum(
            [M[i * CityCount + j] for j in range(CityCount)]) == 1)  # each time only one city can be visited
    for j in range(CityCount):
        model.Add(cp_model.LinearExpr.Sum(
            [M[i * CityCount + j] for i in range(CityCount)]) == 1)  # each city can only be visit once
    model.AddCircuit(arc)
    model.Minimize(cp_model.LinearExpr.ScalProd(M, dis_mat))
    # Create solver
    solver = cp_model.CpSolver()

    # Solve model
    status = solver.Solve(model)
    print(solver.ObjectiveValue())
    print("1, ",end="")
    seq=1
    j=0
    while(seq!=CityCount):
        for i in range(CityCount):
            if (solver.Value(M[j * CityCount + i])==1):
                print(i+1,end="")
                seq += 1
```

```
                if(seq!=CityCount):print(", ",end="")
                j=i


if __name__=='__main__':
    time_start = time.time()  #start timing
    main()
    time_end = time.time()  #stop timing
    time_c = time_end - time_start
    print()
    print('time cost', time_c, 's')
```

TSP2.py

```
import time
from ortools.constraint_solver import pywrapcp
from ortools.constraint_solver import routing_enums_pb2

def str2pos(s):
    p=[0,0]
    for i in range(len(s)):
        if(s[i]==","):
            p[0]=int(s[0:i])
            if(s[len(s)-1]=="\n"):
                p[1]=int(s[i+1:len(s)-1])
            else:
                p[1]=int(s[i+1:len(s)])
    return p


def distance(p1,p2):
    x=pow(p1[0]-p2[0],2)
    y=pow(p1[1]-p2[1],2)
    dis=pow(x+y,0.5)
    return round(dis*1000)


def print_solution(manager, routing, solution,CityCount):#print the solution
```

```
    print(solution.ObjectiveValue()/1000)#the optimal distance needed
    #start from city0
    print("1, ",end="")
    print("%i, " %(solution.Value(routing.NextVar(0))+1),end="")
    city=solution.Value(routing.NextVar(0))
    num=2
    while (num!=CityCount):
        print(solution.Value(routing.NextVar(city))+1,end="")
        if (num!=CityCount-1):print(", ",end="")
        else:
            print()
        city=solution.Value(routing.NextVar(city))
        num+=1

def main():
    filename="in11.txt"
    with open(filename, "r") as f:
        data = f.readline()
    CityCount = int(data)
    pos_str = []
    with open(filename, "r") as f:
        for line in f.readlines():
            pos_str.append(line)
    pos = []
    for i in pos_str[1:]:
        pos.append(str2pos(i))
    dis_mat = []
    for i in pos:
        dis_v = [];
        for j in pos:
            dis_v.append(int(distance(i, j)))
        dis_mat.append(dis_v)
    #define the model
    num_routes = 1
    depot = 0  # since the final route is a circle, we can start at any point
    manager = pywrapcp.RoutingIndexManager(CityCount, 1, depot)
    routing = pywrapcp.RoutingModel(manager)

```

```python
    def distance_callback(from_index, to_index):  # define the callback function
        from_node = manager.IndexToNode(from_index)
        to_node = manager.IndexToNode(to_index)
        return dis_mat[from_node][to_node]

    tran_callback = routing.RegisterTransitCallback(distance_callback)
    routing.SetArcCostEvaluatorOfAllVehicles(tran_callback)
    search_parameters = pywrapcp.DefaultRoutingSearchParameters()
    search_parameters.first_solution_strategy = (routing_enums_pb2.FirstSolutionStrategy.PATH_CHEAPEST_ARC)
    solution = routing.SolveWithParameters(search_parameters)
    print_solution(manager, routing, solution, CityCount)


if __name__=='__main__':
    time_start = time.time()  #start timing
    main()
    time_end = time.time()  #stop timing
    time_c = time_end - time_start
    print()
    print('time cost', time_c, 's')
```

timeTable.py

```python
import sys
from ortools.sat.python import cp_model


# define the callback for arranging instructors to courses
class Solution1Register(cp_model.CpSolverSolutionCallback):  # save each feasible solution to Sol1
    def __init__(self, variables, l, instr, cour, Sol1):  # initialization of the object
        cp_model.CpSolverSolutionCallback.__init__(self)
        self.__M = variables
        self.__l = l
        self.__instr = instr
        self.__cour = cour
```

```python
        self.__Sol1 = Sol1
        """"""
        self.__solution_count = 0

    def on_solution_callback(self):  # function called during each
    ↪ successful search
        if (self.__solution_count >= self.__l):
            self.StopSearch()  # solutions enough
            return
        sol = []
        self.__solution_count += 1
        for c in self.__cour:
            for i in self.__instr:
                if (self.BooleanValue(self.__M[i - 1][c - 1])):  # if
                ↪ instructor i is assigned to course c
                    sol.append((c, i))
        self.__Sol1.append(sol)
        """"""

    def solution_count(self):  # number of solutions
        return self.__solution_count


# define the callback for arranging courses to days
class Solution2Register(cp_model.CpSolverSolutionCallback):  # save each
↪ feasible solution to Sol2
    def __init__(self, variables, l, cour, Sol2):  # initialization of the
    ↪ object
        cp_model.CpSolverSolutionCallback.__init__(self)
        self.__M = variables
        self.__l = l
        self.__cour = cour
        self.__Sol2 = Sol2
        """"""
        self.__solution_count = 0

    def on_solution_callback(self):  # function called during each
    ↪ successful search
        if (self.__solution_count >= self.__l):
```

```
            self.StopSearch()  # solutions enough
            return
        sol = []
        self.__solution_count += 1
        for _d in range(5):
            for c in self.__cour:
                if (self.BooleanValue(self.__M[_d][c - 1])):  # if course c is arranged on day _d+1
                    sol.append((_d + 1, c))
        self.__Sol2.append(sol)
        """"""

    def solution_count(self):  # number of solutions
        return self.__solution_count


#define the solution printer
def printsol(Sol1,Sol2,l,mL):
    num=min(l,len(Sol1)*len(Sol2))
    n=0;
    for i in range(len(Sol1)):
        for j in range(len(Sol2)):
            if (n==num):break
            print("Solution %i"%n)
            n+=1
            for s in Sol1[i]:
                print("Course %i, Instructor %i,"%s,end=" ")
                d=0
                for t in Sol2[j]:
                    if(t[1]==s[0]):
                        print(t[0],end="")
                        d+=1
                        if (d==mL):
                            print(end="")
                            break
                        print(",",end=" ")
                print()


```

```
def main():
    l = int(sys.argv[2])
    nI = 0
    nC = 0
    mL = 0
    mD = 0
    mC = 0
    C = []
    filename=sys.argv[1]
    with open(filename, "r") as f:
        nI, nC, mL, mD, mC = [int(s) for s in f.readline().split(',')]
        for i in range(nI):
            C.append([int(c) for c in str(f.readline()).split(',')])  # instructor i can teach all courses in C[i]
    instr = list(range(1, nI + 1))  # list of instructors
    cour = list(range(1, nC + 1))  # list of courses
    days = [1, 2, 3, 4, 5]  # weekdays
    """define model1"""
    model1 = cp_model.CpModel()
    Sol1 = []
    M1 = []
    for i in instr:
        M_v = []
        for c in cour:
            if (c not in C[i - 1]):  # if instructor i cannot teach course c
                # M_v.append(model1.NewIntVar(0,0,'C%i, In%i'%(c,i)))
                M_v.append(0)
            else:
                M_v.append(model1.NewBoolVar('C%i, In%i' % (c, i)))
        M1.append(M_v)

    """add constrains to model1"""
    # each course can only be taught by one instructor
    for c in cour:
        model1.Add(cp_model.LinearExpr.Sum(M1[i - 1][c - 1] for i in instr) == 1)

    # each instructor can teach at most mC courses
```

```python
    for i in instr:
        model1.Add(cp_model.LinearExpr.Sum(M1[i - 1]) <= mC)

    """define model2"""
    model2 = cp_model.CpModel()
    Sol2 = []
    M2 = []
    for d in days:
        M_v = []
        for c in cour:
            M_v.append(model2.NewBoolVar('Day%i, C%i' % (d, c)))
        M2.append(M_v)

    """add constrains to model2"""
    # each day can have at most mD lectures
    for _d in range(5):
        model2.Add(cp_model.LinearExpr.Sum(M2[_d]) <= mD)

    # each course has exactly mL lectures
    for c in cour:
        model2.Add(cp_model.LinearExpr.Sum(M2[_d][c - 1] for _d in range(5)) == mL)

    """solve model1"""
    solver1 = cp_model.CpSolver()
    sol_reg1 = Solution1Register(M1, l, instr, cour, Sol1)
    solver1.SearchForAllSolutions(model1, sol_reg1)
    """solve model2"""
    solver2 = cp_model.CpSolver()
    sol_reg2 = Solution2Register(M2, l, cour, Sol2)
    solver2.SearchForAllSolutions(model2, sol_reg2)
    printsol(Sol1, Sol2, l, mL)


if __name__=='__main__':
    main()
```

points.py

```python
import random


def main():
    for i in range(1,14):
        num = 5
        output=""
        output+=str(num*i)
        output+="\n"
        for j in range(num*i):
            #print("%i, %i" %(random.randint(0,150),random.randint(0,150)))
            output+="%i, %i" %(random.randint(0,150),random.randint(0,150))
            output+="\n"
        print(output)
        filename="in"
        filename+=str(i)
        filename+=".txt"
        with open(filename, "w") as f:
            f.write(output)


if __name__=='__main__':
    main()
```