
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.

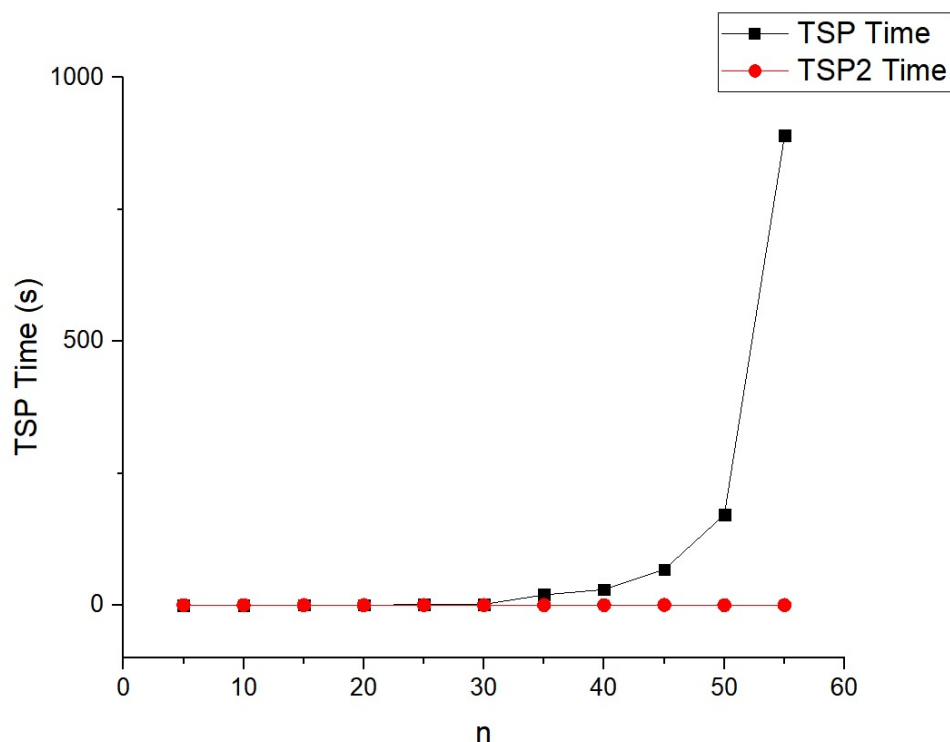


Figure 1: Time cost by TSP and TSP2.

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

¹the 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 find 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-tools², 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 separated 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.

```

1         # define the callback for arranging instructors to courses
2 class Solution1Register(cp_model.CpSolverSolutionCallback): # save
3     ↪ each feasible solution to Sol1
4     def __init__(self, variables, l, instr, cour, Sol1): #
5         ↪ initialization of the object
6         ...
7
8     def on_solution_callback(self): # function called during each
9         ↪ successful search
10        if (self.__solution_count >= self.__l):
11            self.StopSearch() # solutions enough
12            return
13        sol = []
14        self.__solution_count += 1
15        for c in self.__cour:
16            for i in self.__instr:
17                if (self.BooleanValue(self.__M[i - 1][c - 1])): # if
18                    ↪ instructor i is assigned to course c
19                    sol.append((c, i))
20        self.__Sol1.append(sol)

```

²<https://developers.google.cn/optimization/routing/tsp>

```

17
18     def solution_count(self): # number of solutions
19         """..."""
20
21
22     # define the callback for arranging courses to days
23     class Solution2Register(cp_model.CpSolverSolutionCallback): # save
24         ↪ each feasible solution to Sol2
25         def __init__(self, variables, l, cour, Sol2): # initialization of
26             ↪ the object
27             ...
28
29     def on_solution_callback(self): # function called during each
30         ↪ successful search
31         if (self.__solution_count >= self.__l):
32             self.StopSearch() # solutions enough
33             return
34         sol = []
35         self.__solution_count += 1
36         for _d in range(5):
37             for c in self.__cour:
38                 if (self.BooleanValue(self.__M[_d][c - 1])): # if
39                     ↪ course c is arranged on day _d+1
40                     sol.append((_d + 1, c))
41         self.__Sol2.append(sol)
42
43     def solution_count(self): # number of solutions
44         ...

```

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.

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

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

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

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

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

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

The code for my model is shown as following.

```

1  """define model1"""
2      model1 = cp_model.CpModel()
3      Sol1 = []
4      M1 = []
5      for i in instr:
6          M_v = []
7          for c in cour:
8              if (c not in C[i - 1]): # if instructor i cannot teach
9                  ↪ course c
10                     # M_v.append(model1.NewIntVar(0,0, 'C%i, In%i'%(c,i)))
11                     M_v.append(0)
12             else:
13                 M_v.append(model1.NewBoolVar('C%i, In%i' % (c, i)))
14             M1.append(M_v)
15
16 """add constrains to model1"""
17 # each course can only be taught by one instructor
18 for c in cour:

```

```

18         model1.Add(cp_model.LinearExpr.Sum(M1[i - 1][c - 1] for i in
        ↪ instr) == 1)
19
20     # each instructor can teach at most mC courses
21     for i in instr:
22         model1.Add(cp_model.LinearExpr.Sum(M1[i - 1]) <= mC)
23
24     """define model2"""
25     model2 = cp_model.CpModel()
26     Sol2 = []
27     M2 = []
28     for d in days:
29         M_v = []
30         for c in cour:
31             M_v.append(model2.NewBoolVar('Day%i, C%i' % (d, c)))
32         M2.append(M_v)
33
34     """add constraints to model2"""
35     # each day can have at most mD lectures
36     for _d in range(5):
37         model2.Add(cp_model.LinearExpr.Sum(M2[_d]) <= mD)
38
39     # each course has exactly mL lectures
40     for c in cour:
41         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.

```

1 def printsol(Sol1,Sol2,l,mL):
2     num=min(l,len(Sol1)*len(Sol2))
3     n=0;
4     for i in range(len(Sol1)):
5         for j in range(len(Sol2)):
6             if (n==num):break
7             print("Solution %i"%n)

```

```

8      n+=1
9      for s in Sol1[i]:
10         print("Course %i, Instructor %i,"%s,end=" ")
11         d=0
12         for t in Sol2[j]:
13             if(t[1]==s[0]):
14                 print(t[0],end="")
15                 d+=1
16                 if (d==mL):
17                     print(end="")
18                     break
19                 print(", ",end=" ")
20         print()

```

Appendix

TSP.py

```
1  import sys
2  from ortools.sat.python import cp_model
3  import time
4
5  def str2pos(s):
6      p=[0,0]
7      for i in range(len(s)):
8          if(s[i]==","):
9              p[0]=int(s[0:i])
10             if(s[len(s)-1]=="\n"):
11                 p[1]=int(s[i+1:len(s)-1])
12             else:
13                 p[1]=int(s[i+1:len(s)])
14     return p
15
16
17 def distance(p1,p2):
18     x=pow(p1[0]-p2[0],2)
19     y=pow(p1[1]-p2[1],2)
20     dis=pow(x+y,0.5)
21     return round(dis*1000)
22
23
24 def main():
25     filename="in11.txt"
26     with open(filename, "r") as f:
27         data = f.readline()
28         CityCount = int(data)
29         pos_str = []
30         with open(filename, "r") as f:
31             for line in f.readlines():
32                 pos_str.append(line)
33         pos = []
34         for i in pos_str[1:]:
35             pos.append(str2pos(i))
```



```

36     dis_mat = []
37     for i in pos:
38         for j in pos:
39             dis_mat.append(int(distance(i, j)))
40     #define model
41     model = cp_model.CpModel()
42     M = [model.NewIntVar(0, 1, '%i to %i' % (i, j)) for i in
43           ↪ range(CityCount) for j in range(CityCount)]
44     arc = []
45     for i in range(CityCount):
46         for j in range(CityCount):
47             if (j == i): continue
48             arc.append([i, j, M[i * CityCount + j]])
49     #add constrains
50     for i in range(CityCount):
51         model.Add(cp_model.LinearExpr.Sum(
52             [M[i * CityCount + j] for j in range(CityCount)]) == 1) # each
53             ↪ time only one city can be visited
54     for j in range(CityCount):
55         model.Add(cp_model.LinearExpr.Sum(
56             [M[i * CityCount + j] for i in range(CityCount)]) == 1) # each
57             ↪ city can only be visit once
58     model.AddCircuit(arc)
59     model.Minimize(cp_model.LinearExpr.ScalProd(M, dis_mat))
60     # Create solver
61     solver = cp_model.CpSolver()
62     # Solve model
63     status = solver.Solve(model)
64     print(solver.ObjectiveValue())
65     print("1, ",end="")
66     seq=1
67     j=0
68     while(seq!=CityCount):
69         for i in range(CityCount):
70             if (solver.Value(M[j * CityCount + i])==1):
71                 print(i+1,end="")
72                 seq += 1

```

```

72         if(seq!=CityCount):print(", ",end="")
73         j=i
74
75
76 if __name__=='__main__':
77     time_start = time.time() #start timing
78     main()
79     time_end = time.time() #stop timing
80     time_c = time_end - time_start
81     print()
82     print('time cost', time_c, 's')

```

TSP2.py

```

1  import time
2  from ortools.constraint_solver import pywrapcp
3  from ortools.constraint_solver import routing_enums_pb2
4
5  def str2pos(s):
6      p=[0,0]
7      for i in range(len(s)):
8          if(s[i]==" ,"):
9              p[0]=int(s[0:i])
10             if(s[len(s)-1]=="\n"):
11                 p[1]=int(s[i+1:len(s)-1])
12             else:
13                 p[1]=int(s[i+1:len(s)])
14     return p
15
16
17 def distance(p1,p2):
18     x=pow(p1[0]-p2[0],2)
19     y=pow(p1[1]-p2[1],2)
20     dis=pow(x+y,0.5)
21     return round(dis*1000)
22
23
24 def print_solution(manager, routing, solution, CityCount):#print the
    ↪ solution

```

```

25     print(solution.ObjectiveValue()/1000)#the optimal distance needed
26     #start from city0
27     print("1, ",end="")
28     print("%i, " %(solution.Value(routing.NextVar(0))+1),end="")
29     city=solution.Value(routing.NextVar(0))
30     num=2
31     while (num!=CityCount):
32         print(solution.Value(routing.NextVar(city))+1,end="")
33         if (num!=CityCount-1):print(", ",end="")
34         else:
35             print()
36             city=solution.Value(routing.NextVar(city))
37             num+=1
38
39 def main():
40     filename="in11.txt"
41     with open(filename, "r") as f:
42         data = f.readline()
43     CityCount = int(data)
44     pos_str = []
45     with open(filename, "r") as f:
46         for line in f.readlines():
47             pos_str.append(line)
48     pos = []
49     for i in pos_str[1:]:
50         pos.append(str2pos(i))
51     dis_mat = []
52     for i in pos:
53         dis_v = [];
54         for j in pos:
55             dis_v.append(int(distance(i, j)))
56         dis_mat.append(dis_v)
57     #define the model
58     num_routes = 1
59     depot = 0 # since the final route is a circle, we can start at any
60               ↪ point
61     manager = pywrapcp.RoutingIndexManager(CityCount, 1, depot)
62     routing = pywrapcp.RoutingModel(manager)

```

```

63     def distance_callback(from_index, to_index): # define the callback
        ↪ function
64         from_node = manager.IndexToNode(from_index)
65         to_node = manager.IndexToNode(to_index)
66         return dis_mat[from_node][to_node]
67
68     tran_callback = routing.RegisterTransitCallback(distance_callback)
69     routing.SetArcCostEvaluatorOfAllVehicles(tran_callback)
70     search_parameters = pywrapcp.DefaultRoutingSearchParameters()
71     search_parameters.first_solution_strategy =
        ↪ (routing_enums_pb2.FirstSolutionStrategy.PATH_CHEAPEST_ARC)
72     solution = routing.SolveWithParameters(search_parameters)
73     print_solution(manager, routing, solution, CityCount)
74
75
76 if __name__ == '__main__':
77     time_start = time.time() #start timing
78     main()
79     time_end = time.time() #stop timing
80     time_c = time_end - time_start
81     print()
82     print('time cost', time_c, 's')

```

timeTable.py

```

1  import sys
2  from ortools.sat.python import cp_model
3
4
5  # define the callback for arranging instructors to courses
6  class Solution1Register(cp_model.CpSolverSolutionCallback): # save each
        ↪ feasible solution to Sol1
7      def __init__(self, variables, l, instr, cour, Sol1): # initialization
            ↪ of the object
8          cp_model.CpSolverSolutionCallback.__init__(self)
9          self.__M = variables
10         self.__l = l
11         self.__instr = instr
12         self.__cour = cour

```

```

13     self.__Sol1 = Sol1
14     #####
15     self.__solution_count = 0
16
17     def on_solution_callback(self): # function called during each
18         ↪ successful search
19         if (self.__solution_count >= self.__l):
20             self.StopSearch() # solutions enough
21             return
22         sol = []
23         self.__solution_count += 1
24         for c in self.__cour:
25             for i in self.__instr:
26                 if (self.BooleanValue(self.__M[i - 1][c - 1])): # if
27                     ↪ instructor i is assigned to course c
28                     sol.append((c, i))
29             self.__Sol1.append(sol)
30         #####
31
32     def solution_count(self): # number of solutions
33         return self.__solution_count
34
35     # define the callback for arranging courses to days
36     class Solution2Register(cp_model.CpSolverSolutionCallback): # save each
37         ↪ feasible solution to Sol2
38         def __init__(self, variables, l, cour, Sol2): # initialization of the
39             ↪ object
40             cp_model.CpSolverSolutionCallback.__init__(self)
41             self.__M = variables
42             self.__l = l
43             self.__cour = cour
44             self.__Sol2 = Sol2
45             #####
46             self.__solution_count = 0
47
48     def on_solution_callback(self): # function called during each
49         ↪ successful search
50         if (self.__solution_count >= self.__l):

```

```

47         self.StopSearch()  # solutions enough
48         return
49     sol = []
50     self.__solution_count += 1
51     for _d in range(5):
52         for c in self.__cour:
53             if (self.BooleanValue(self.__M[_d][c - 1])):  # if course c
54                 ↪ is arranged on day _d+1
55                 sol.append((_d + 1, c))
56     self.__Sol2.append(sol)
57     """
58
59     def solution_count(self):  # number of solutions
60         return self.__solution_count
61
62     #define the solution printer
63     def printsol(Sol1,Sol2,l,mL):
64         num=min(l,len(Sol1)*len(Sol2))
65         n=0;
66         for i in range(len(Sol1)):
67             for j in range(len(Sol2)):
68                 if (n==num):break
69                 print("Solution %i"%n)
70                 n+=1
71                 for s in Sol1[i]:
72                     print("Course %i, Instructor %i,"%s,end=" ")
73                     d=0
74                     for t in Sol2[j]:
75                         if(t[1]==s[0]):
76                             print(t[0],end="")
77                             d+=1
78                             if (d==mL):
79                                 print(end="")
80                                 break
81                             print(", ",end=" ")
82                     print()
83
84

```

```

85 def main():
86     l = int(sys.argv[2])
87     nI = 0
88     nC = 0
89     mL = 0
90     mD = 0
91     mC = 0
92     C = []
93     filename=sys.argv[1]
94     with open(filename, "r") as f:
95         nI, nC, mL, mD, mC = [int(s) for s in f.readline().split(',')]
96         for i in range(nI):
97             C.append([int(c) for c in str(f.readline()).split(',')]) #
98             ↪ instructor i can teach all courses in C[i]
99     instr = list(range(1, nI + 1)) # list of instructors
100    cour = list(range(1, nC + 1)) # list of courses
101    days = [1, 2, 3, 4, 5] # weekdays
102    """define model1"""
103    model1 = cp_model.CpModel()
104    Sol1 = []
105    M1 = []
106    for i in instr:
107        M_v = []
108        for c in cour:
109            if (c not in C[i - 1]): # if instructor i cannot teach course
110                ↪ c
111                # M_v.append(model1.NewIntVar(0,0, 'C%i, In%i'%(c,i)))
112                M_v.append(0)
113            else:
114                M_v.append(model1.NewBoolVar('C%i, In%i' % (c, i)))
115        M1.append(M_v)
116    """add constraints to model1"""
117    # each course can only be taught by one instructor
118    for c in cour:
119        model1.Add(cp_model.LinearExpr.Sum(M1[i - 1][c - 1] for i in instr)
120            ↪ == 1)
121
122    # each instructor can teach at most mC courses

```

```

121     for i in instr:
122         model1.Add(cp_model.LinearExpr.Sum(M1[i - 1]) <= mC)
123
124     """define model2"""
125     model2 = cp_model.CpModel()
126     Sol2 = []
127     M2 = []
128     for d in days:
129         M_v = []
130         for c in cour:
131             M_v.append(model2.NewBoolVar('Day%i, C%i' % (d, c)))
132         M2.append(M_v)
133
134     """add constraints to model2"""
135     # each day can have at most mD lectures
136     for _d in range(5):
137         model2.Add(cp_model.LinearExpr.Sum(M2[_d]) <= mD)
138
139     # each course has exactly mL lectures
140     for c in cour:
141         model2.Add(cp_model.LinearExpr.Sum(M2[_d][c - 1] for _d in range(5))
142                     == mL)
143
144     """solve model1"""
145     solver1 = cp_model.CpSolver()
146     sol_reg1 = Solution1Register(M1, l, instr, cour, Sol1)
147     solver1.SearchForAllSolutions(model1, sol_reg1)
148     """solve model2"""
149     solver2 = cp_model.CpSolver()
150     sol_reg2 = Solution2Register(M2, l, cour, Sol2)
151     solver2.SearchForAllSolutions(model2, sol_reg2)
152     printsol(Sol1, Sol2, l, mL)
153
154 if __name__ == '__main__':
155     main()

```


points.py

```
1  import random
2
3
4  def main():
5      for i in range(1,14):
6          num = 5
7          output=""
8          output+=str(num*i)
9          output+="\n"
10         for j in range(num*i):
11             #print("%i, %i"
12                 ↪   %(random.randint(0,150),random.randint(0,150)))
13             output+= "%i, %i" %(random.randint(0,150),random.randint(0,150))
14             output+="\n"
15         print(output)
16         filename="in"
17         filename+=str(i)
18         filename+="txt"
19         with open(filename, "w") as f:
20             f.write(output)
21
22 if __name__=='__main__':
23     main()
```