巡回セールスマン問題に対するタブーサーチ

43M21219　是兼 慎也

1.　問題選定，背景・目的

近似的解法の実装に挑戦したかったので，今回はメタヒューリスティクスの中のタブーサーチを実装する．タブーサーチを採用した理由は，計画学研究室内で現在，誰も実装していないためである．対象とする問題は組み合わせ最適化問題の中でもっとも代表的な問題の一つである，巡回セールスマン問題（TSP）とする．

ここで，TSP について解説する．TSP とは都市の集合と各都市間の移動コストが与えられたとき，すべての都市をちょうど 1 度ずつ巡り出発点に戻る巡回路のうち，最も総移動コストが小さくなる経路を求める問題である．この問題は計算複雑性理論において NP 困難と呼ばれるクラスに属している．訪問する都市数が増加するごとに組み合わせの数が指数的に増加するため，困難な問題とされている．TSP にも様々な種類の問題が提案されているが，今回は対称セールスマン問題を対象とする．

2.　定式化

巡回セールスマン問題の定式化について記載する．

まず，集合と定数を定義する．

集合

N：都市の集合

定数

c_{ij}：都市iから都市 j$(i \neq j)$に移動する際に生じる移動コスト

決定変数

x_{ij}：都市iから都市jに移動する場合 1，それ以外の場合 0 の値をとる 0-1 決定変数

M ： 十分に大きい正の整数

次に定式化を示す．

$$\min \sum_{i}^{N} \sum_{j}^{N} c_{ij} x_{ij} \tag{1}$$

$$s.t. \ \sum_{i}^{N} x_{ij} = 1, i \ = \ 1, \dots, n \tag{2}$$

$$\sum_{j}^{N} x_{ij} = 1, j \ = \ 1, \dots, n \tag{3}$$

$$u_i + 1 - M(1 - x_{ij}) \leq u_j, \quad \forall i, j \tag{4}$$

$$1 \leq u_i \leq n - 1 \tag{5}$$

$$x_{ij} \in \{0,1\} \quad \forall i,j \tag{6}$$

(1) 式は目的関数が巡回路の距離の総和の最小化であることを示す.
(2) 式は都市iを到着するのは 1 回だけであることを示す.
また, (3) 式は都市iを出発するのは 1 回だけであることを示す.
(4)(5)式は部分巡回路を取り除くための制約である.
(6) 式は決定変数x_{ij}が離散変数条件であることを示す.

3. アルゴリズム

タブーサーチの前に，まず，その基礎となる局所探索のアルゴリズムを簡単に示す.
(1)初期解を生成し，それを暫定解とする.
(2)暫定解xの近傍$N(x)$を生成する
(3)近傍内で最良解かつ暫定解よりも良い解を暫定解とする.
(4) (2), (3)を繰り返し，解の更新がなくなれば終了
巡回セールスマン問題では都市の訪問順序によって解を表現できる．このような順列によって解表現できる場合，一般に挿入近傍や交換近傍が用いられる．このほかにも巡回セールスマン問題に対する近傍として Or-opt 近傍， λ -opt 近傍($\lambda \geq 2$)が存在する[1]．今回は実装が比較的簡単な交換近傍と 2-opt 近傍を実装した．また，局所探索法の設計において近傍内の改善解が複数個存在する場合，どの改善解に遷移するかが性能に大きく影響を与える．これは移動戦略と呼ばれる．今回はアルゴリズムの(3)に示す通り，最良移動戦略を採用している．局所探索は初期解が探索に与える影響が大きい．今回では探索性能に注目したかったので，割り振られた都市の ID 順とした.

つぎにタブーサーチのアルゴリズムを簡単に示す.
(1)初期解を生成する．タブーリストTを初期化する.
(2)近傍$N(x)$を生成し，$N(x) \setminus (\{x\} \cup T)$の中で最良の解$x'$に遷移する.
(3)解の遷移と同時に，遷移の記録をタブーリストに保存する.
(4)局所最適化に至っても，一時的な改悪を許し，タブーでない解へ遷移することで探索を続ける.
(5)指定の終了条件を満たせば，探索を終了する.
タブーサーチにおいて．タブーリストに解の遷移をどのように記憶させるかによって，探索性能は変化する．生成した順列によって表される解を直接記憶させた場合，次に生成した解がタブーリストに含まれるかを確認する際に時間がかかってしまう．したがって，タブーリストには近傍操作の特徴を記憶することが多い．このような特徴は属性と呼ぶ．交換近傍の属性については近傍操作の前後で変化する変数と変数の順列における位置をタブーリストによって記憶する．例えば，解{5,3,4,1,2}から近傍解{3,5,4,1,2}に遷移する場合,

左から一番目の要素と左から二番目の要素が交換されているため，タブーリストには{1:5, 2:3}が記録される．一方，2-opt 近傍については交換する辺を禁止辺として記憶した．2-opt 近傍を用いた探索においては「過去の探索で得られた最良解よりも良い解が得られる場合は，その近傍操作がタブーリストに含まれていたとしても例外的に許可する」という特別探索基準を設けた．

4.　シミュレーション結果

近似解法である局所探索，タブーサーチの結果を比較する．作成したプログラムは付録に記載する．また，数理最適ソルバーである gurobi を用いて厳密解を生成し，近似解と比較する．

つぎに，問題設定について示す．都市数が 30 個の問題を対象とする．都市iの座標(x_i, y_i)とする．与える問題は 0<=xi, yi<=100 の範囲の乱数で作成する．また各都市間の距離は座標間のマンハッタン距離で生成する．

以下には開発環境を示す．

CPU : Intel(R)core(TM)i7-2600
実装 RAM ; 16GB
使用言語 : python 3.7.6
ソフトウェア : gurobi 9.1.2

次に局所探索，タブーサーチのパラメータについて記載する．近傍操作の繰り返し回数は最大 100 回とした．また，タブーリストの記憶を保持しておく期間であるタブー期間は 15 とした．

数値実験の結果を表 1 に示す．

表 1．近似解法の解

	局所探索	タブーサーチ
交換近傍	754	734
2-opt 近傍	594	574

まず，交換近傍を適用した局所探索の結果は 754 であった．タブーサーチの結果は 734 であった．2-opt 近傍を用いた局所探索では 594，タブーサーチでは 574 という結果が得られた．また，gurobi から算出された厳密解は 548 であった．

図 1，計算結果

数値実験の解の変遷を fig.1 に示す．

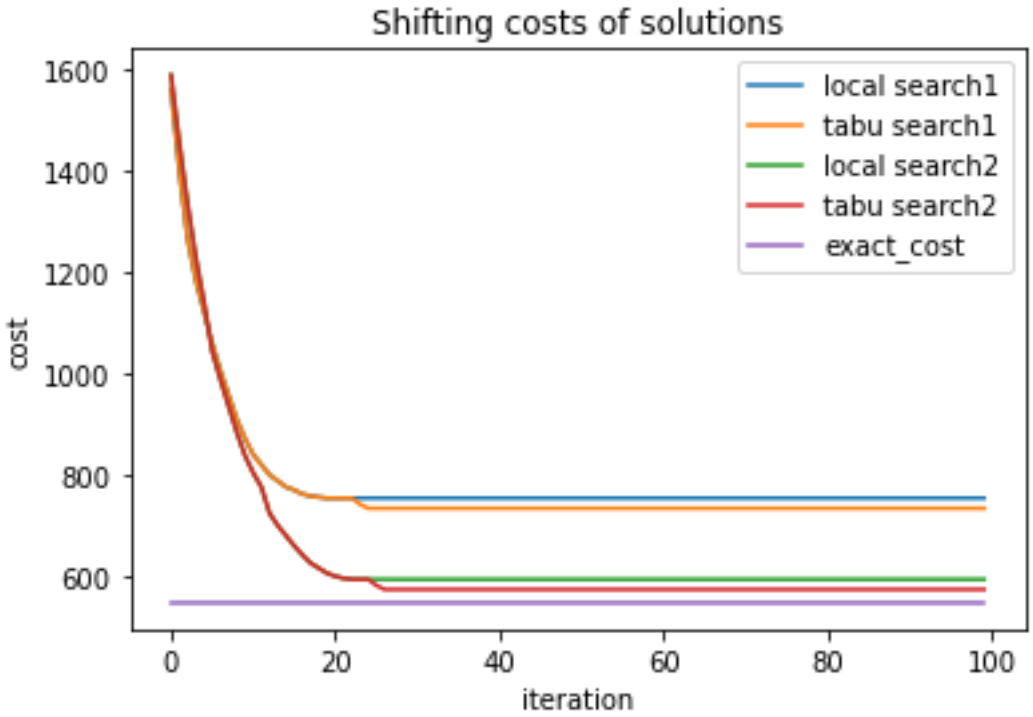

fig.1 の青線は交換近傍を用いた局所探索のイテレーションごとのコスト，オレンジ線は交換近傍を用いたタブーサーチのイテレーションごとのコスト，緑線は 2-opt 近傍を用いた局所探索のイテレーションごとのコスト，赤線は 2-opt 近傍を用いたタブーサーチのイテレーションごとのコスト，紫線は厳密解の値を示したものである．

5.　考察とまとめ

まず，図 1 より，交換近傍と 2-opt 近傍どちらの場合でも局所探索よりもタブーサーチを用いたほうがより良い解が得られたことがわかる．これは局所解に陥ったのちに，タブーリストを参照しながら改悪解にも遷移した結果，局所解を脱出した結果だと考えられる．

次に，交換近傍と 2-opt 近傍を比較する．探索序盤において，交換近傍を用いたほうが 2-opt 近傍よりも若干ではあるが速くコストの値が減少していたが，繰り返し数が 10 回あたりを境に 2-opt 近傍を用いたほうがコストの値が小さくなった．これは，交換近傍を用いた場合一度の近傍操作で変更される辺の数が 4 個であるのに対して，2-opt 近傍を用いた場合では一度の近傍操作で変更される辺の数が 2 であるためであると考えられる．1 度に 4 辺を交換する交換近傍のほうが序盤は有利であったが，探索を続けていくほど解の多様性の面で 2 辺を交換する 2-opt 近傍のほうが有利となったと考えられる．

最も小さい値を得たのは 2-opt 近傍を適用したタブーサーチであったが，厳密解と比較すると差がある．この点については，長期メモリの一種である頻度メモリなどを実装することにより改善される可能性があると考えられる．

以下，今回のレポートをまとめる．

- 交換近傍を用いた局所探索とタブーサーチ，2-opt 近傍を適用した局所探索とタブーサーチを実装し，厳密解と比較した．
- いずれの近傍の場合でもタブーサーチを用いたほうが良い値が得られた．
- TSP において，交換近傍と 2-opt 近傍を比較すると，2opt 近傍を適用したほうが適している．

参考資料

[1]橋本英樹，野々部宏司，“入門タブー探索法”，オペレーションズ・リサーチ 2013 年 12 月号, p703-707

付録

＜作成したコード＞

```
# -*- coding: utf-8 -*-
"""
Created on Wed Nov 17 19:07:48 2021
@author: korekane1998
tabu-search for TSP
"""
import math
import random
import time
import gurobipy as gp
from matplotlib import pyplot as plt
import numpy as np

class Instance:
    def __init__(self,n):
        random.seed(11)
        self.n = n
        self.points = []
        for i in range(n):
            self.points.append([random.randint(0,100),random.randint(0,100)])
```

```
        self.cost = [[0] * n for i in range(n)]
        for i in range(n):
            for j in range(i+1,n):
                #マンハッタン距離
                self.cost[i][j] = math.fabs(self.points[i][0]-self.points[j][0]) +
math.fabs(self.points[i][1]-self.points[j][1])
                self.cost[j][i] = math.fabs(self.points[i][0]-self.points[j][0]) +
math.fabs(self.points[i][1]-self.points[j][1])
                # #ユークリッド距離
                # self.cost[i][j] = math.sqrt(math.fabs(self.points[i][0]-
self.points[j][0])**2 + math.fabs(self.points[i][1]-self.points[j][1])**2)
                # self.cost[j][i] = math.sqrt(math.fabs(self.points[i][0]-
self.points[j][0])**2 + math.fabs(self.points[i][1]-self.points[j][1])**2)

class Solve:
    #昇順
    def initial_route1(self,instance):
        route = list(range(instance.n))
        return route
    #ランダム
    def initial_route2(self,instance):
        route = list(range(instance.n))
        random.shuffle(route)
        return route
    #貪欲法
    def initial_route3(self,instance):
        route = [0]
        for i in range(instance.n-1):
            copy_list = instance.cost[route[-1]].copy()
            for j in route:
                copy_list[j] = 9999
            index = copy_list.index(min(copy_list))
            route.append(index)
        return route
    def cal_cost(self,route,instance):
        sum_cost = 0
```

```
        n = instance.n
        for i in range(n - 1):
            sum_cost += instance.cost[route[i]][route[i + 1]]
        sum_cost += instance.cost[route[n - 1]][route[0]]
        return sum_cost
    def get_cost(self, index_i, index_j, route, instance):
        if(index_i == instance.n):
            index_i = 0
        if(index_j == instance.n):
            index_j = 0
        return instance.cost[route[index_i]][route[index_j]]
    #交換近傍
    def cal_neighbor1(self, index_i, index_j, route, instance):
        n = instance.n
        ### i,j:都市の ID
        ### index_i,index_j:route 上の都市 i,j のインデクス
        delta = 0
        ### 隣接する場合
        if(index_i == index_j - 1 or index_i == index_j + n - 1):
            delta -= (self.get_cost(index_i - 1, index_i, route, instance) +
self.get_cost(index_j, index_j + 1, route, instance))
            delta += (self.get_cost(index_i - 1, index_j, route, instance) +
self.get_cost(index_i, index_j + 1, route, instance))
        elif(index_i == index_j + 1 or index_j == index_i + n - 1):
            delta -= (self.get_cost(index_j - 1, index_j, route, instance) +
self.get_cost(index_i, index_i + 1, route, instance))
            delta += (self.get_cost(index_j - 1, index_i, route, instance) +
self.get_cost(index_j, index_i + 1, route, instance))
        ### 隣接しない場合
        else:
            delta -= (self.get_cost(index_i - 1, index_i, route, instance) +
self.get_cost(index_i, index_i + 1, route, instance))
            delta += (self.get_cost(index_j - 1, index_i, route, instance) +
self.get_cost(index_i, index_j + 1, route, instance))
            delta -= (self.get_cost(index_j - 1, index_j, route, instance) +
self.get_cost(index_j, index_j + 1, route, instance))
```

```
                delta += (self.get_cost(index_i - 1, index_j, route, instance) +
self.get_cost(index_j, index_i + 1, route, instance))
        return delta
    #交換近傍を用いた近傍探索
    def local_search1(self, ini_route, iteration, instance):
        n = instance.n
        best = 999999
        local = self.cal_cost(ini_route,instance)
        best = min(best, local)
        #### search
        route = ini_route.copy()
        best_route = ini_route.copy()
        neighbors = dict()
        iteration_values = [0]*iteration
        for k in range(iteration):
            #近傍解を列挙
            for index_i in range(n):
                for index_j in range(index_i + 1, n):
                    neighbors[str(index_i)+','+str(index_j)] =
self.cal_neighbor1(index_i, index_j, route, instance)
            sorted_neighbors = sorted(neighbors.items(), key = lambda item : item[1])
            flag = True
            for neighbor in sorted_neighbors:
                #最もよい近傍に遷移
                nids = neighbor[0].split(',')
                nid_index_i = int(nids[0])
                nid_index_j = int(nids[1])
                delta = neighbor[1]
                if(delta < 0):
                    flag = False
                    local += delta
                    break
            #遷移がなければ終了
            if flag:
                iteration_values[k:] = [best]*(iteration-k)
                break
```

```
                #解の更新
                index_i = nid_index_i
                index_j = nid_index_j
                i = route[index_i]
                j = route[index_j]
                route.pop(index_i)
                route.insert(index_i, j)
                route.pop(index_j)
                route.insert(index_j, i)
                if(local < best):
                        best = local
                        best_route = route.copy()
                iteration_values[k] = best
                neighbors.clear()
        result = dict()
        result['tour'] = str(best_route)
        result['cost'] = best
        result['iteration_values'] = iteration_values
        return result
#タブーリスト
def tabu_append(self, tabu_length, tabu_list, opration):
        if len(tabu_list) >= tabu_length:
                tabu_list.pop(0)
        tabu_list.append(opration)
        # tabu_list.append([opration[1],opration[0]])
        return tabu_list
#交換近傍を用いたタブーサーチ
def tabu_search1(self, ini_route, iteration, instance, tabu_length):
        n = instance.n
        tabu_list = list()
        best = 999999
        local = self.cal_cost(ini_route,instance)
        best = min(best, local)
        #### search
        route = ini_route.copy()
        best_route = ini_route.copy()
```

```
            neighbors = dict()
            iteration_values = [0]*iteration
            for k in range(iteration):
                #近傍解を列挙
                for index_i in range(n):
                    for index_j in range(index_i + 1, n):
                        neighbors[str(index_i)+','+str(index_j)] =
self.cal_neighbor1(index_i, index_j, route, instance)
                sorted_neighbors = sorted(neighbors.items(), key = lambda item : item[1])
                flag = False
                for neighbor in sorted_neighbors:
                    nids = neighbor[0].split(',')
                    nid_index_i = int(nids[0])
                    nid_index_j = int(nids[1])
                    nid_i = route[nid_index_i]
                    nid_j = route[nid_index_j]
                    if [nid_i, nid_index_i, nid_j, nid_index_j] not in tabu_list:
                        #最もよい近傍に遷移
                        delta = neighbor[1]
                        local += delta
                        flag = True
                        break
                if flag:
                    #解の更新
                    index_i = nid_index_i
                    index_j = nid_index_j
                    i = route[index_i]
                    j = route[index_j]
                    route.pop(index_i)
                    route.insert(index_i, j)
                    route.pop(index_j)
                    route.insert(index_j, i)
                    #タブーリストに記憶
                    #opration : [都市 i,都市 i のインデクス,都市 j,都市 j のインデクス]
                    opration = [i, index_i, j, index_j]
                    tabu_list = self.tabu_append(tabu_length, tabu_list, opration)
```

```
                if(local < best):
                    best = local
                    best_route = route.copy()
                iteration_values[k] = best
                neighbors.clear()
            result = dict()
            result['tour'] = str(best_route)
            result['cost'] = best
            result['iteration_values'] = iteration_values
            return result
        #2-opt 近傍
        def cal_neighbor2(self, index_i, index_j, route, instance):
            n = instance.n
            ### i,j:都市の ID
            ### index_i,index_j:route 上の都市 i,j のインデクス
            ### index_i < index_j
            delta = 0
            if index_i == 0 and index_j == n - 1:
                delta = 0
            else:
                delta -= (self.get_cost(index_i - 1, index_i, route, instance) +
self.get_cost(index_j, index_j + 1, route, instance))
                delta += (self.get_cost(index_i - 1, index_j, route, instance) +
self.get_cost(index_i, index_j + 1, route, instance))
            return delta
        #2-opt 近傍を用いた近傍探索
        def local_search2(self, ini_route, iteration, instance):
            n = instance.n
            best = 999999
            local = self.cal_cost(ini_route,instance)
            best = min(best, local)
            #### search
            route = ini_route.copy()
            best_route = ini_route.copy()
            neighbors = dict()
            iteration_values = [0]*iteration
```

```
            for k in range(iteration):
                #近傍解を列挙
                for index_i in range(n):
                    for index_j in range(index_i + 1, n):
                        neighbors[str(index_i)+','+str(index_j)] =
self.cal_neighbor2(index_i, index_j, route, instance)
                sorted_neighbors = sorted(neighbors.items(), key = lambda item : item[1])
                flag = True
                for neighbor in sorted_neighbors:
                    #最もよい近傍に遷移
                    nids = neighbor[0].split(',')
                    nid_index_i = int(nids[0])
                    nid_index_j = int(nids[1])
                    delta = neighbor[1]
                    if(delta < 0):
                        flag = False
                        #print(delta)
                        local += delta
                        break
                #遷移がなければ終了
                if flag:
                    iteration_values[k:] = [best]*(iteration-k)
                    break
                #解の更新
                index_i = nid_index_i
                index_j = nid_index_j
                rev = route[index_i : index_j + 1]
                rev.reverse()
                route = route[:index_i] + rev + route[index_j + 1:]
                if(local < best):
                    best = local
                    best_route = route.copy()
                iteration_values[k] = best
                neighbors.clear()
            result = dict()
            result['tour'] = str(best_route)
```

```
        result['cost'] = best
        result['iteration_values'] = iteration_values
        return result
    #2-opt 近傍用タブーリスト
    def tabu_append2(self, tabu_length, tabu_list, opration):
        if len(tabu_list) >= tabu_length:
            tabu_list.pop(0)
        tabu_list.append(opration[0])
        tabu_list.append(opration[1])
        # tabu_list.append([opration[1],opration[0]])
        return tabu_list
    #2-opt 近傍を用いたタブーサーチ
    def tabu_search2(self, ini_route, iteration, instance, tabu_length):
        n = instance.n
        tabu_list = list()
        best = 999999
        local = self.cal_cost(ini_route,instance)
        best = min(best, local)
        #### search
        route = ini_route.copy()
        best_route = ini_route.copy()
        neighbors = dict()
        iteration_values = [0]*iteration
        for k in range(iteration):
            #近傍解を列挙
            for index_i in range(n):
                for index_j in range(index_i + 1, n):
                    neighbors[str(index_i)+','+str(index_j)] =
self.cal_neighbor2(index_i, index_j, route, instance)
            sorted_neighbors = sorted(neighbors.items(), key = lambda item : item[1])
            flag = False
            for neighbor in sorted_neighbors:
                nids = neighbor[0].split(',')
                nid_index_i = int(nids[0])
                nid_index_j = int(nids[1])
                in1, in2, in3, in4 = index_i-1,index_i,index_j,index_j+1
```

```
                    if not (in2 == 0) and (in4 == n):
                        if in4 == n:
                            in4 = 0
                        if not ([route[in1], route[in2]] in tabu_list)or([route[in3],
route[in4]] in tabu_list)or(neighbor[1] >= 0):
                            #最もよい近傍に遷移
                            delta = neighbor[1]
                            local += delta
                            flag = True
                            break
                if flag:
                    #解の更新
                    index_i = nid_index_i
                    index_j = nid_index_j
                    #タブーリストに記憶
                    #opration : [禁止辺]
                    if not (index_i == 0) and (index_j == n - 1):
                        opration = [[route[index_i-1], route[index_i]],[route[index_j],
route[index_j + 1]]]
                    rev = route[index_i : index_j + 1]
                    rev.reverse()
                    route = route[:index_i] + rev + route[index_j + 1:]
                    if not (index_i == 0) and (index_j == n - 1):
                        tabu_list = self.tabu_append2(tabu_length, tabu_list, opration)
                if(local < best):
                    best = local
                    best_route = route.copy()
                iteration_values[k] = best
                neighbors.clear()
            result = dict()
            result['tour'] = str(best_route)
            result['cost'] = best
            result['iteration_values'] = iteration_values
            return result
        #Gurobi を用いた厳密解法
        def Gurobi(self,instance,iteration):
```

```
            n = instance.n
            cost = np.array(instance.cost)
            #the big M
            M = 10000
            tsp = gp.Model("traveling_salesman")
            tsp.Params.outputFlag = 0
            x = tsp.addVars(n, n, vtype=gp.GRB.BINARY, name = "x")
            u = tsp.addVars(n, name = "u")
            # Set objective
            tsp.setObjective( gp.quicksum(cost[i,j]*x[i,j] for i in range(n) for j in range(n)),
gp.GRB.MINIMIZE)
            # Assignment constraints:
            tsp.addConstrs(( gp.quicksum(x[i,j] for j in range(n)) == 1 for i in range(n) ))
            tsp.addConstrs(( gp.quicksum(x[i,j] for i in range(n)) == 1 for j in range(n) ))
            # Subtour-breaking constraints:
            tsp.addConstrs(( u[i] + 1 - u[j] <= M*(1 - x[i,j])   for i in range(n) for j in
range(1,n) ))
            # Solving the model
            tsp.optimize()
            result = dict()
            print("厳密解　コスト",tsp.ObjVal)
            result['cost'] = tsp.ObjVal
            iteration_values = [tsp.ObjVal] * iteration
            result['iteration_values'] = iteration_values
            return result
        #結果のプロット
        def
plot(self,local_result1,tabu_result1,local_result2,tabu_result2,gurobi_result,iteration):
            fig, ax = plt.subplots()
            x = list(range(iteration))
            y1 = local_result1['iteration_values']
            y2 = tabu_result1['iteration_values']
            y3 = local_result2['iteration_values']
            y4 = tabu_result2['iteration_values']
            y5 = gurobi_result['iteration_values']
            ax.set_xlabel('iteration')  # x 軸ラベル
```

```
        ax.set_ylabel('cost')　# y 軸ラベル
        ax.set_title('Shifting costs of solutions') #　グラフタイトル
        l1,l2,l3,l4,l5 = "local search1","tabu search1","local search2","tabu
search2","exact_cost"
        ax.plot(x, y1, label = l1)
        ax.plot(x, y2, label = l2)
        ax.plot(x, y3, label = l3)
        ax.plot(x, y4, label = l4)
        ax.plot(x, y5, label = l5)
        ax.legend(loc = 0)
        plt.show()

def main():
    probrem_size = 30
    tabu_length = 30
    instance = Instance(probrem_size)
    iteration = 100
    solve = Solve()
    ini_route1 = solve.initial_route1(instance)
    ini_routes = [ini_route1]
    # ini_route2 = solve.initial_route2(instance)
    # ini_route3 = solve.initial_route3(instance)
    # ini_routes = [ini_route1,ini_route2,ini_route3]
    for ini_route in ini_routes:
        t = time.time()
        local_result1 = solve.local_search1(ini_route, iteration, instance)
        print("交換近傍を用いた局所探索")
        # print(local_result1['tour'])
        print('目的関数値 :'+str(local_result1['cost']))
        print(time.time()-t,'(s)')
        print()

        t = time.time()
        tabu_result1 = solve.tabu_search1(ini_route, iteration, instance, tabu_length)
        print("交換近傍を用いたタブーサーチ")
        # print(tabu_result1['tour'])
```

```
        print('目的関数値 : '+str(tabu_result1['cost']))
        print(time.time()-t,'(s)')
        print()

        t = time.time()
        local_result2 = solve.local_search2(ini_route, iteration, instance)
        print("2-opt 近傍を用いた局所探索")
        # print(local_result2['tour'])
        print('目的関数値 : '+str(local_result2['cost']))
        print(time.time()-t,'(s)')
        print()

        t = time.time()
        tabu_result2 = solve.tabu_search2(ini_route, iteration, instance, tabu_length)
        print("2-opt 近傍を用いたタブーサーチ")
        # print(tabu_result2['tour'])
        print('目的関数値 : '+str(tabu_result2['cost']))
        print(time.time()-t,'(s)')
        print()

        gurobi_result = solve.Gurobi(instance,iteration)

solve.plot(local_result1,tabu_result1,local_result2,tabu_result2,gurobi_result,iteration)

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