HW2

August 29, 2023

1 Read and Prepare Data

[ ]:
```python
import math
import re
import os
import random
import numpy as np
import pandas as pd
import time
from datetime import datetime
from matplotlib import pyplot as plt
```

Creating problem data from files:

[ ]:
```python
def CreateProblemData(folder, labels, prsize):
    problems = dict()
    for s in prsize:
        prdata = dict()
        for l in labels:
            f = open(f"{folder}/{l}{s}.dat","r")
            text = f.read()
            f.close()
            rows = text.split("\n")
            rows = [r.strip() for r in rows  if r.strip() != "" ]

            table = list(map(lambda x: x.split("\t"), rows  ))
            table = np.array(table)[:,1]
            table = list(map(lambda x: int(x), table  ))
            prdata[l] = table
        prdata = pd.DataFrame(prdata)
        problems[s] = prdata
    return(problems)
```

[ ]:
```python
probs = CreateProblemData("Data", ["x","y","dem"], [51,76,101])
```

2 Used Functions

2.1 Basic Functions

Solution Representation Discrete value representation is used to store solutions.

```
[ ]: x = [1,2,3,4]
```

Euclidean Distance and Weightet Distance Matrix

```
[ ]: def dist(x,y = 0):

         return np.sqrt( np.sum((x - y)**2) )
```

```
[ ]: def CreateWeightedDistMatrix(AllProblems, ProbSize):
         ProbData = AllProblems[ProbSize]

         dist = np.sqrt(\
             np.power(
             np.diag (ProbData["x"] ) @ np.matrix( ProbSize*ProbSize *[1] ).
      ↪reshape(ProbSize,ProbSize) -\
             np.matrix( ProbSize*ProbSize *[1] ).reshape(ProbSize,ProbSize) @ np.
      ↪diag (ProbData["x"] ), 2) +\
             \
             np.power(\
             np.diag (ProbData["y"] ) @ np.matrix( ProbSize*ProbSize *[1] ).
      ↪reshape(ProbSize,ProbSize) -\
             np.matrix( ProbSize*ProbSize *[1] ).reshape(ProbSize,ProbSize) @ np.
      ↪diag (ProbData["y"] ), 2)\
                       )
         res = np.diag(ProbData["dem"]) @ dist
         res = res + np.diag(ProbSize * [float('nan')])
         return np.around(res,2)
```

```
[ ]: W = CreateWeightedDistMatrix(probs,51)
```

Assignment Function

```
[ ]: W[5,:]
```

```
[ ]: array([284.97, 477.21, 601.04, 357.4 , 433.42,    nan, 280.37, 306.47,
            578.25, 674.88, 371.29, 306.47, 462.45, 175.03, 585.45, 536.78,
            420.56, 247.52, 593.78, 639.93, 702.16, 395.41, 190.07, 236.78,
            282.94, 371.29, 153.94, 505.45, 629.23, 715.01, 462.45, 289.5 ,
            759.12, 720.45, 748.19, 806.02, 464.32, 456.16, 844.54, 748.19,
            543.2 , 629.   , 396.87, 565.11, 699.48, 231.22, 263.91, 152.05,
            561.26, 618.81, 193.83])
```

```
[ ]: np.argmin(W[5,:])
```

```
[ ]: 5
```

```
[ ]: def AssignCustomers(AllProblems, ProbSize, WeightedDists, x):
         ProbData = AllProblems[ProbSize]
         custs = set(range(ProbSize)).difference(set(x))
         assignments = { key : list() for key in x }
         z = 0
         for c in custs:
             i = np.argmin( WeightedDists[c,x])
             assignments[x[i]].append(c)
             z = z + WeightedDists[c,x[i]]
         return (z, assignments)
```

2.2 Move Operators

```
[ ]: def OneOneExchangeGiven(AllProblems, ProbSize, WeightedDists, Solution,␣
      ↪newCust, newFac):

         x = Solution[0].copy()
         newX = set(x).difference(set([newCust])).union(set([newFac]))
         newX = list(newX)

         assignments = AssignCustomers(AllProblems, ProbSize, WeightedDists, newX)

         res = [newX] + (list(assignments))
         return res
```

```
[ ]: def OneOneExchangeRand(AllProblems, ProbSize, WeightedDists, Solution):

         x = sol[0].copy()
         custs = set(range(ProbSize)).difference(x)

         newFac = random.sample(custs,1)
         newCust = random.sample(x,1)

         return OneOneExchangeGiven(AllProblems, ProbSize, WeightedDists, Solution,␣
      ↪newCust, newFac)
```

```
[ ]: def OneOneExchangeBest(AllProblems, ProbSize, WeightedDists, Solution):

         x = set(Solution[0])
         custs = set(range(ProbSize)).difference(x)

         BestZ = Solution[1]
         BestSolution = list()
```

```
    for c in custs:
        for f in x:
            newFac = c
            newCust = f
            newSolution = OneOneExchangeGiven(AllProblems, ProbSize,␣
↪WeightedDists, Solution, newCust, newFac)
            if newSolution[1] < BestZ:
                BestSolution = newSolution
                BestZ = newSolution[1]

    if BestZ == Solution[1]:
        print("Local Search is complete!")
        return Solution

    return BestSolution
```

```
[ ]: def ManyManyExchangeGiven(AllProblems, ProbSize, WeightedDists, Solution,␣
↪newCusts, newFacs):

    x = Solution[0].copy()

    newX = set(x).difference(set(newCusts)).union(set(newFacs))
    newX = list(newX)

    assignments = AssignCustomers(AllProblems, ProbSize, WeightedDists, newX)

    res = [newX] + (list(assignments))
    return res
```

```
[ ]: def ManyManyExchangeRand(AllProblems, ProbSize, WeightedDists, Solution, n):

    x = Solution[0].copy()
    custs = set(range(ProbSize)).difference(x)

    newFacs = random.sample(set(custs),n)
    newCusts = random.sample(set(x),n)

    return ManyManyExchangeGiven(AllProblems, ProbSize, WeightedDists,␣
↪Solution, newCusts, newFacs)
```

2.3 Construction Heuristic

```
[ ]: def SortMatchings(CostMatrix):
    # Function to sort assignments with respect to a given cost matrix
    SortedValues = list(np.squeeze(np.array(np.sort(CostMatrix).reshape(1,-1))))
    SortedValues = list(np.unique(SortedValues))
```

```
    Matchings = list(map(lambda x:
                     np.random.permutation( # If there are even assignments, sort␣
↪them randomly
                         np.squeeze(
                             np.stack(
                                 np.where(CostMatrix==x), 1)
                         ).
                         reshape(-1,2)
                     )
                     ,
                     SortedValues ))

    Matchings = np.stack(np.concatenate(Matchings))
    return Matchings
```

[ ]:
```
def MinWeightedDistConstructionHeuristic(AllProblems, ProbSize, p,␣
↪WeightedDists):
    sortedDistances = SortMatchings(WeightedDists)
    x = set()
    i = 0
    while len(x) < p:
        x.add(sortedDistances[i][1])
        i += 1
    x = list(x)
    assignments = AssignCustomers(AllProblems, ProbSize, WeightedDists, x)
    res = [x] + (list(assignments))
    return res
```

2.4 Local Search

[ ]:
```
def OneOneExchangeLocalSearch(AllProblems, ProbSize, WeightedDists, Solution):
    Sol = Solution.copy()
    newSol =  Solution.copy()
    while True:
        newSol = OneOneExchangeBest(AllProblems, ProbSize, WeightedDists, Sol).
↪copy() # Call move operator
        if Sol[1] <= newSol[1]: # If no better solution can be found
            return Sol
        Sol = newSol.copy()

    return Sol
```

2.5 Simulated Annealing

[ ]:
```
def SetInitialTemp(AllProblems, ProbSize, p, WeightedDists,P0, n = 20):
    delta = 0
    for i in range(n):
        xfirst = list(random.sample(range(ProbSize),p))
        solfirst = [xfirst] + list(AssignCustomers(AllProblems, ProbSize,␣
↪WeightedDists,xfirst ))

        sollast = OneOneExchangeRand(AllProblems, ProbSize,␣
↪WeightedDists,solfirst)
        delta += abs(solfirst[1] - sollast[1])

    return - (delta/n) / np.log(P0)
```

[ ]:
```
def SA(AllProblems, ProbSize, p, WeightedDists, Solution, Parameters):
    P0, ip, r, sf, fp  = Parameters
    T = SetInitialTemp(AllProblems,ProbSize,p,WeightedDists,P0,25)
    Sol = Solution
    SolStar = Solution
    terct = 0
    L = sf * p * (ProbSize - p)
    it = 0
    while(terct < 5):
        print(f"T: {T}")
        j = 0
        count = 0
        while count < L:
            it += 1
            count += 1
            SolPrime = OneOneExchangeRand(AllProblems, ProbSize,␣
↪WeightedDists,Sol)
            delta = SolPrime[1] - Sol[1]
            if delta <= 0:
                Sol = SolPrime
                SolStar = SolPrime
                j += 1
            else:
                R = random.random()
                if np.exp(-delta/T) >= R:
                    Sol = SolPrime
                    j += 1

        # Update Temperature
        if j/L <= fp:
            terct += 1
            T = T*r
```

```
        else:
            terct = 0
            if j/L <= ip:
                T = T*r
            else:
                T = T*0.5
    return [SolStar, it]
```

[ ]:
```
SAparams = [0.9, 0.7, 0.85, 1, 0.1] # P0, ip, r, sf, fp
```

[ ]:
```
SA(probs, 51, 4, W, sol, SAparams)
```

2.6 Variable Neighborhood Search

[ ]:
```
def VNS(AllProblems, ProbSize, p, WeightedDists, Solution, kmax):
    Sol = Solution
    SolStar = Solution
    delta = 1
    k = 0
    it = 0
    counter = 0
    while True:
        k = 0
        while k < kmax:
            SolPrime = ManyManyExchangeRand(AllProblems, ProbSize,␣
 ↪WeightedDists, Sol, k) #Shaking
            SolPrPr = OneOneExchangeLocalSearch(AllProblems, ProbSize,␣
 ↪WeightedDists, SolPrime) #LS

            if SolPrPr[1] < Sol[1]:
                Sol = SolPrPr
                k = 1
            else: k +=1
            it += 1
        delta = SolStar[1] - Sol[1]
        SolStar = Sol
        if delta <= 0: counter += 1
        if counter >= 3: break
    return SolStar
```

[ ]:
```
solVNS = VNS(probs, 51, 4, W, solVNS, 3 )
```

```
Local Search is complete!
Local Search is complete!
Local Search is complete!
Local Search is complete!
```

```
Local Search is complete!
Local Search is complete!
Local Search is complete!
Local Search is complete!
Local Search is complete!
```

[ ]:
```
solVNS
```

[ ]:
```
[[48, 3, 28, 7],
 7555.709999999999,
 {48: [4, 8, 9, 10, 14, 29, 32, 33, 37, 38, 44],
  3: [11, 12, 13, 16, 17, 18, 24, 36, 39, 40, 41, 43, 45, 46, 50],
  28: [1, 2, 15, 19, 20, 34, 35, 49],
  7: [0, 5, 6, 21, 22, 23, 25, 26, 27, 30, 31, 42, 47]}]
```

[ ]:
```
OneOneExchangeBest(probs, 51,  W, solVNS )
```

```
Local Search is complete!
```

[ ]:
```
[[48, 3, 28, 7],
 7555.709999999999,
 {48: [4, 8, 9, 10, 14, 29, 32, 33, 37, 38, 44],
  3: [11, 12, 13, 16, 17, 18, 24, 36, 39, 40, 41, 43, 45, 46, 50],
  28: [1, 2, 15, 19, 20, 34, 35, 49],
  7: [0, 5, 6, 21, 22, 23, 25, 26, 27, 30, 31, 42, 47]}]
```

[ ]:
```
sol = [[48, 3, 28, 7]] + list(AssignCustomers(probs, 51,W,[48, 3, 28, 7]))
```

[ ]:
```
OneOneExchangeLocalSearch(probs, 51,  W, sol )
```

```
7555.709999999999
7559.789999999999
```

[ ]:
```
[[48, 3, 28, 7],
 7555.709999999999,
 {48: [4, 8, 9, 10, 14, 29, 32, 33, 37, 38, 44],
  3: [11, 12, 13, 16, 17, 18, 24, 36, 39, 40, 41, 43, 45, 46, 50],
  28: [1, 2, 15, 19, 20, 34, 35, 49],
  7: [0, 5, 6, 21, 22, 23, 25, 26, 27, 30, 31, 42, 47]}]
```

[ ]:
```
OneOneExchangeBest(probs, 51, W, sol)
```

[ ]:
```
[[48, 3, 28, 7],
 7555.709999999999,
 {48: [4, 8, 9, 10, 14, 29, 32, 33, 37, 38, 44],
  3: [11, 12, 13, 16, 17, 18, 24, 36, 39, 40, 41, 43, 45, 46, 50],
  28: [1, 2, 15, 19, 20, 34, 35, 49],
  7: [0, 5, 6, 21, 22, 23, 25, 26, 27, 30, 31, 42, 47]}]
```

[ ]:
```
solVNS
```

[ ]: [1, 3, 2, 7]

3 51 Problem

3.1 p = 4

3.1.1 Initial Points

3.1.2 Simulated Annealing

3.1.3 Variable Neighborhood Search

3.2 p = 6

3.2.1 Initial Points

3.2.2 Simulated Annealing

3.2.3 Variable Neighborhood Search

3.3 p = 8

3.3.1 Initial Points

3.3.2 Simulated Annealing

3.3.3 Variable Neighborhood Search

3.4 Title

[ ]:
```
x0 = list()
```

[ ]:

[ ]:

[ ]:

[ ]:

[ ]:

[ ]:

[ ]:
```
def CostProductConstructionHeuristic(Dataset):

    # Get Variables
    zstar = Dataset["zstar"]
    m = Dataset["m"]
    n = Dataset["n"]
```

```
    OpCost = Dataset["OpCost"]
    CapCost = Dataset["CapCost"]
    Cap = Dataset["Cap"]

    # Calculate products of costs so called Negative Values
    NegValue = np.multiply(OpCost, CapCost)

    # Sort Agent-Job Matchings with respect to Negative Values
    SortedMatchings = SortMatchings(NegValue)

    Crem = Cap.copy() # Remaining Capacity
    xlist = list() # X
    z = 0 # Z
    unassigned = list(range(n)) # Unassigned Jobs

    for r in SortedMatchings:
        if  (Crem[r[0]] - CapCost[r[0],r[1]] >= 0
             and r[1] in unassigned):
            # If remaining capacity permits:
            Crem[r[0]] = Crem[r[0]] - CapCost[r[0],r[1]] # Reduce Capacity
            unassigned.remove((r[1])) # Remove job from unassigned list
            xlist.append(r) # Add mathcing to the solution
            z = z + OpCost[r[0],r[1]] #Sum up operational costs

    # Converting solution to a dictionary for convenience
    xlist = np.stack(xlist)
    x = dict()
    for i in range(m):
        x[i] = xlist[xlist[:,0] == i][:,1]
    for key in x.keys():
        x[key] = list(np.sort(x[key]))

    # Print message if any unassigned job exists
    if unassigned != []:
        print(f"These jobs are unassigned: {unassigned}")

    # Function returns X, Z and Crem
    # which can be used again in iterations
    return (x,z,list(Crem))
```

[ ]:
```
ProblemData = list((rows[prbegin[0]:prbegin[1]-1], rows[prbegin[1]:
 ↪prbegin[2]-1], rows[prbegin[2]:]))
for prob in range(3):
    ProblemData[prob] = [r.strip() for r in ProblemData[prob]  if r.strip() !=␣
 ↪"" ]
```

3.5 Improvement Heuristic

1-1 Exchange Neighborhood is used in the solution. Function used to move to the best solution in the 1-1 Exchange neighborhood of a predetermined solution is given below. If there are more than one local optima, one of them is selected randomly:

[ ]:
```
def OneOneExchangeLocalSearch(ProbData, Solutions, timelimit):
    Sols = Solutions.copy()
    t1 = datetime.now()
    while(1):
        realtime = (datetime.now() - t1).total_seconds() # Calculate duration
        newsln = OneOneExchangeMove(ProbData,Sols[-1]) # Call move operator
        if ((newsln == None) # If no better solution can be found
            or (realtime > timelimit)): # If the real time duration exceeds␣
↪given limit
            break

        Sols.append(newsln)

        if Sols.count(None) > 0: Sols.remove(None)
    return Sols
```

4 Problem 1

[ ]:
```
slns = list() # All solutions will be stored in this list
zstar = ProbDataSets[0]["zstar"]
```

4.1 Construction Heuristic

[ ]:
```
tstart = time.process_time_ns()

random.seed(11)
newsln = CostProductConstructionHeuristic(ProbDataSets[0]) # Run Algorithm
slns.append(newsln)

tend = time.process_time_ns()
print(f"Executed in {1.e-9*(tend - tstart)} CPU*seconds.")
```

Print Solution

[ ]:
```
print("X = {",end="")
for key in slns[0][0].keys():
    print(f"\n( {key}, {set([j for j in slns[-1][0][key]])} ),",end="")
print("\b\n}")
print(f"Z = {slns[0][1]}")
print(f"Crem = {slns[0][2]}")
print(f"Z* = {zstar}")
```

```
print(f"{100 * ( slns[0][1]/zstar-1):.2f}% larger than z*")
```

4.2 Improvement Heuristic

Starting Solution:

[ ]:
```
slns[0]
```

[ ]:
```
tstart = time.process_time_ns()

random.seed(12)
slns = OneOneExchangeLocalSearch(ProbDataSets[0], slns, 300) # Run algorithm␣
↪with 5 min time limit

tend = time.process_time_ns()
print(f"Executed in {1.e-9*(tend - tstart)} CPU*seconds.")
```

Iterations

[ ]:
```
[sln[1] for sln in slns ]
```

Print Solution

[ ]:
```
print("X = {",end="")
for key in slns[-1][0].keys():
    print(f"\n( {key}, {set([j for j in slns[-1][0][key]])} ),",end="")
print("\b\n}")
print(f"Z = {slns[-1][1]}")
print(f"Crem = {slns[-1][2]}")
print(f"Z* = {zstar}")
print(f"{100 * ( slns[-1][1]/zstar-1):.2f}% larger than z*")
```

5 Problem 2

[ ]:
```
slns2 = list()
zstar2 = ProbDataSets[1]["zstar"]
```

5.1 Construction Heuristic

[ ]:
```
tstart = time.process_time_ns()

random.seed(19)
newsln = CostProductConstructionHeuristic(ProbDataSets[1]) # Run Algorithm
slns2.append(newsln)

tend = time.process_time_ns()
print(f"Executed in {1.e-9*(tend - tstart)} CPU*seconds.")
```

Print Solution

```
[ ]: print("X = {",end="")
     for key in slns2[0][0].keys():
         print(f"\n( {key}, {set([j for j in slns2[0][0][key]])} ),",end="")
     print("\b\n}")
     print(f"Z = {slns2[0][1]}")
     print(f"Crem = {slns2[0][2]}")
     print(f"Z* = {zstar2}")
     print(f"{100 * ( slns2[0][1]/zstar2-1):.2f}% larger than z*")
```

5.2 Improvement Heuristic

Starting Solution:

```
[ ]: slns2[0]
```

1-1 Exchange Neighborhood is used in the solution. Function used for moving to the best solution in the 1-1 Exchange neighborhood of a predetermined solution is given below:

```
[ ]: tstart = time.process_time_ns()

     random.seed(12)
     slns2 = OneOneExchangeLocalSearch(ProbDataSets[1], slns2, 300) # Run algorithm␣
      ↪with 5 min time limit

     tend = time.process_time_ns()
     print(f"Executed in {1.e-9*(tend - tstart)} CPU*seconds.")
```

Iterations

```
[ ]: [sln[1] for sln in slns2 ]
```

Print Solution

```
[ ]: print("X = {",end="")
     for key in slns2[-1][0].keys():
         print(f"\n( {key}, {set([j for j in slns2[-1][0][key]])} ),",end="")
     print("\b\n}")
     print(f"Z = {slns2[-1][1]}")
     print(f"Crem = {slns2[-1][2]}")
     print(f"Z* = {zstar2}")
     print(f"{100 * ( slns2[-1][1]/zstar2-1):.2f}% larger than z*")
```

6 Problem 3

[ ]:
```
slns3 = list()
zstar3 = ProbDataSets[2]["zstar"]
```

6.1 Construction Heuristic

[ ]:
```
tstart = time.process_time_ns()

random.seed(32)
newsln = CostProductConstructionHeuristic(ProbDataSets[2]) # Run Algorithm
slns3.append(newsln)

tend = time.process_time_ns()
print(f"Executed in {1.e-9*(tend - tstart)} CPU*seconds.")
```

Print Solution

[ ]:
```
print("X = {",end="")
for key in slns3[0][0].keys():
    print(f"\n( {key}, {set([j for j in slns3[0][0][key]])} ),",end="")
print("\b\n}")
print(f"Z = {slns3[0][1]}")
print(f"Crem = {slns3[0][2]}")
print(f"Z* = {zstar3}")
print(f"{100 * ( slns3[0][1]/zstar3-1):.2f}% larger than z*")
```

6.2 Improvement Heuristic

Starting Solution:

[ ]:
```
slns3[0]
```

1-1 Exchange Neighborhood is used in the solution. Function used for moving to the best solution in the 1-1 Exchange neighborhood of a predetermined solution is given below:

[ ]:
```
tstart = time.process_time_ns()

random.seed(26)
slns3 = OneOneExchangeLocalSearch(ProbDataSets[2], slns3, 300) # Run algorithm␣
 ↪with 5 min time limit

tend = time.process_time_ns()
print(f"Executed in {1.e-9*(tend - tstart)} CPU*seconds.")
```

Iterations

[ ]:
```
[sln[1] for sln in slns3 ]
```

Print Solution

[ ]:
```
print("X = {",end="")
for key in slns3[-1][0].keys():
    print(f"\n( {key}, {set([j for j in slns3[-1][0][key]])} ),",end="")
print("\b\n}")
print(f"Z = {slns3[-1][1]}")
print(f"Crem = {slns3[-1][2]}")
print(f"Z* = {zstar3}")
print(f"{100 * ( slns3[-1][1]/zstar3-1):.2f}% larger than z*")
```