IE 517 Homework 1

Emre Kara

August 29, 2023

1 Problem Definition

Generalized Assignment Problem (GAP) is an extended version of Assignment problem in which each agent has some capacity and each task assignment has some size and cost varying among agents and tasks. The objective of the problem is to minimize total cost while assigning each task to an agent. Given m = of agents and n = of tasks, the problem has n^m solutions which grows exponentially as the number of agents increases which means the problem requires more sophisticated approaches than complete enumeration.

In this work, a construction and an improvement heuristic algorithms are applied on three different minimization problem sets each having different sizes. Optimal objective values are known for these problems. In the end, results of algorithms and known optimal values are compared to have an idea about performances of algorithms.

2 Solution Approaches

The solution approach is consisted of tho phases: First, the construction heuristic algorithm is applied to obtain a solution from scratch and then, starting from this solution, the improvement heuristic algorithm is used with aim of finding better solutions. Also, CPU times are measured in order to evaluate time dependent performances of algorithms.

Solutions to the problems are represented as **discrete values**. Binary representation of solutions would have n^m length which would be impractical to use with large numbers and as there is no ordered relation, it is not possible to use permutation representation. Because of these two reasons, discrete value representation is chosen. The formation of the representation is designed as a set of pairs whose first element is the agent and the second element is another set which have tasks assigned to that agent. Also, indices starts from 0 due to the indexing convention of the program used.

Ex: For $m = 3$ and $n = 9$

$X = \{(0, \{1, 5, 6\}), (1, \{0, 2, 7, 8\}), (2, \{3, 4\})\}$

2.1 Construction Heuristic Solution

An algorithm similar to the Construction Heuristic Algorithm for Knapsack Problem (Knapsack Algorithm) was adopted as a solution. In Knapsack Algorithm, utility/cost ratios are used in utility maximization problem. In this study, instead maximizing the utility, the cost is being minimized. Hence, there is not any similar ratio metric. However, instead of division, multiplication can be used. We are trying to avoid high costs - let us say operational costs - as well as high sizes - let us say capacity costs. As we avoid both of these two "bad things", we can use product of these two as a combining cost or a negative value (**OpCost** $\times$ **CapCost**). The algorithm works as making assignments with smaller negative values early then the others. In case of equality, assignments are sorted randomly among themselves.

Pseudo-code of the algorithm:

Algorithm 1: Negative Value Construction Heuristic

$NegValue =: OpCost \times CapCost$;
$M =:$ All possible assignments;
$X =: \emptyset$;
Sort M w.r.t $NegValue$, sort randomly those which have equal values;
while $M \neq \emptyset$ **do**
    if M_1 *is feasible* **then**
        $X =: X \cup M_1$;
    end
    $M =: M - M_1$
end
return X

2.2 Improvement Heuristic Solution

As an improvement heuristic, local search algorithm is used. Starting from a solution, best solution in a predefined neighborhood is chosen until there is no better solution in the neighborhood of the current solution. **1-1 exchange** neighborhood structure is adopted for the algorithm. Similar to the construction heuristic, random selection is used as a tie-breaker when more than one neighbor yields best improvement equally.

Pseudo-code of the algorithm:

```
Algorithm 2: Negative Value Construction Heuristic
X_0 : Initial Solution;
X* =: X_0;
repeat
    S =: ∅;
    for (i,k)and(j,l) in X* do
        ΔZ =: OpCost(i,l) + OpCost(j,k) − OpCost(i,k) − OpCost(j,l);
        if Swap (i,k), (j,l) is feasible and ΔZ < 0 then
            X =: X* − {(i,k),(j,l)};
            X =: X ∪ {(i,l),(j,k)};
            S =: S ∪ X
        end
    end
    if S ≠ ∅ then
        X* =: Best Solution in S;
    else
        return X*;
    end
until No better solution in N(X*);
```

3 Problem Sets and Solutions

Proposed two algorithms are applied on three problems sets. Best solution obtained for both construction and improvement heuristic was on problem 1 with 11.19% and 6.62% gap respectively. Improvement heuristic resulted in an acceptable outcome with 13.79% at worst on problem 3 while construction heuristic resulted in 27.4% at worst on problem 3 which can be evaluated as unsatisfactory. It can be seen that as problem size increases, algorithms perform worse. However, yet the performance of improvement heuristic seems satisfactory at some point.

	Construction Heuristic				Improvement Heuristic			
Instance	**Solution (jobs assigned to each agent)**	**z**	**% Gap from z***	**CPU-Time (s)**	**Solution (jobs assigned to each agent)**	**z**	**% Gap from z***	**CPU-Time (s)**
5-25	{(0, {7, 9, 10, 11, 14, 15, 19}), (1, {1, 2, 4, 5, 12, 16, 24}), (2, {0, 6, 18, 21, 23}), (3, {3, 20, 13}), (4, {8, 17, 22})}	487	11.19	0.001	{(0, {7, 9, 10, 11, 14, 15, 19}), (1, {1, 2, 4, 5, 12, 13, 16}), (2, {0, 18, 20, 22, 23}), (3, {24, 3, 6}), (4, {8, 17, 21})}	467	6.62	0.046875
8-40	{(0, {2, 36, 13, 15, 19, 27}), (1, {1, 29, 17, 7}), (2, {16, 25, 28, 5}), (3, {26}), (4, {35, 3, 4, 6, 38, 11}), (5, {32, 8, 12, 22, 23, 24}), (6, {34, 37, 39, 9, 10, 31}), (7, {0, 33, 14, 18, 20, 21, 30})}	766	18.58	0.03125	{(0, {4, 36, 13, 15, 17, 27}), (1, {16, 1, 26, 29}), (2, {25, 2, 19, 6}), (3, {7}), (4, {33, 3, 35, 38, 11, 20}), (5, {32, 5, 12, 22, 23, 24}), (6, {34, 37, 39, 8, 9, 31}), (7, {0, 10, 14, 18, 21, 28, 30})}	693	7.28	0.1875
10-50	{(0, {36, 38, 42, 43, 23}), (1, {0, 4, 11, 15, 19, 26, 29}), (2, {2, 3, 8, 12, 16, 18, 20, 22, 27}), (3, {1, 5, 13, 45, 48, 28}), (4, {10, 46, 14, 21, 25}), (5, {6, 31}), (6, {33, 44, 49}), (7, {24}), (8, {9, 34, 35, 47}), (9, {32, 37, 7, 39, 40, 41, 17, 30})}	730	27.4	0.046875	{(0, {36, 47, 23, 24, 28}), (1, {0, 4, 43, 15, 19, 25, 26}), (2, {2, 35, 8, 12, 16, 18, 20, 22, 27}), (3, {1, 38, 45, 13, 49, 21}), (4, {33, 3, 10, 14, 48}), (5, {6, 31}), (6, {42, 44, 46}), (7, {5}), (8, {9, 34, 11, 29}), (9, {32, 37, 7, 39, 40, 41, 17, 30})}	652	13.79	0.328125

4 Python Code

4.1 Read and Prepare Data

[1]:
```
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
```

Reading from txt file and saving it into *rows* variable as a list of rows

[2]:
```
f = open("gap-data-3instances.txt","r")
text = f.read()
f.close()
rows = text.split("\n")
```

[3]:
```
len(re.findall("Problem", text))
# Counting occurences of problem word: 3 Occurences
```

[3]:
```
3
```

Fİnding beginning and ending rows of related subsections of problems

[4]:
```
prrows = map(lambda x: re.search("Problem", x) !=  None, rows)
prbegin = np.where(list(prrows))[0]; prbegin
```

[4]:
```
array([11, 30, 55], dtype=int64)
```

Trimming problem data and omitting empty rows

[5]:
```
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() !=␣
 ↪"" ]
```

Funciton to create datasets for problems

[6]:
```
def CreateProblem(obj):
    zstar = int(obj[0][obj[0].find("=") + 1  : obj[0].find("(")])
    m,n = obj[1].split(" ")
    m,n = int(m), int(n)
    OpCost = np.matrix([ list(map( int, r.split(" ") )) for r in obj[2:2+m] ])
    CapCost = np.matrix([ list(map( int, r.split(" ") )) for r in obj[2+m:2+2*m]␣
 ↪])
    Capacity = np.array(list(map( int, obj[2+2*m].split(" ") )))

    return{"zstar":zstar, "m":m, "n":n, "OpCost":OpCost, "CapCost":CapCost,␣
 ↪"Cap": Capacity}
```

[7]:
```
ProbDataSets = list(map(CreateProblem, ProblemData))
```

In solutions, indexings of agents and jobs are as following:

agents: 0, 1, . . . , m-1
jobs: 0, 1, . . . , n-1

This approach is used due to Python's indexing convention.

4.2 Algorithms

4.2.1 Construction Heuristic

Solution is constructed adding one by one assignments starting from those which yields smallest $OpCost \times CapCost$ valeus (NegValues). If two or more assignments have equal negative values, they are sorted randomly within. This is the only randomness in the algorithm.

[8]:
```
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
```

[9]:

```python
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))
```

4.2.2 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:

[10]:
```python
def OneOneExchangeMove(ProbData, Solution):

    # Get Problem Data Variables
    zstar = ProbData["zstar"]
    m = ProbData["m"]
    n = ProbData["n"]
    OpCost = ProbData["OpCost"]
    CapCost = ProbData["CapCost"]
    Cap = ProbData["Cap"]

    # Get Solution Variables
    X = Solution[0].copy()
    Z = Solution[1].copy()
    Crem = Solution[2].copy()

    # Define Algorithm Variables
    selected = list() # To select unordered pairs of agents
    BestDeltaOpCost = 0 # Local Z*
    BestSwap = list() # Local X*

    for agent1 in X.keys():
        selected.append(agent1)
        for agent2 in X.keys():
            if agent2 in selected: continue
            for job1 in X[agent1]:
                for job2 in X[agent2]:

                    DeltaOpCost = OpCost[agent1,job2] \
                    + OpCost[agent2,job1] \
                    - OpCost[agent1,job1] \
                    - OpCost[agent2,job2]

                    CRemAgent1New = Crem[agent1] \
                    + CapCost[agent1,job1] \
                    - CapCost[agent1,job2]

                    CRemAgent2New = Crem[agent2] \
                    + CapCost[agent2,job2] \
                    - CapCost[agent2,job1]
```

```
                if ((CRemAgent1New > 0) and (CRemAgent2New > 0)): # If␣
↪feasible

                    if BestDeltaOpCost > DeltaOpCost: # If Z is better than␣
↪best Z* found so far
                        BestDeltaOpCost = DeltaOpCost
                        BestSwap = [[(agent1,job1),(agent2,job2)]]

                    elif ((DeltaOpCost < 0) and (BestDeltaOpCost ==␣
↪DeltaOpCost)): # In case of equality
                        BestSwap.append([(agent1,job1),(agent2,job2)])

    if BestSwap == []:
        # Print message if the algorithm is complete
        print("Local Search is complete!")
        return None

    # In case of equality select one randomly
    BestSwap = random.sample(BestSwap,1)[0]

    # Remove old pair and add new pair from and to the selected agent 1
    X[BestSwap[0][0]] = np.union1d( np.setdiff1d( X[BestSwap[0][0]],
                                                   BestSwap[0][1] ),
                                    BestSwap[1][1] )

    # Remove old pair and add new pair from and to the selected agent 2
    X[BestSwap[1][0]] = np.union1d( np.setdiff1d( X[BestSwap[1][0]],
                                                   BestSwap[1][1] ),
                                    BestSwap[0][1] )

    Z = Z + BestDeltaOpCost # Update Z

    # Update Crem for selected agent 1
    Crem[BestSwap[0][0]] = Crem[BestSwap[0][0]] \
    + CapCost[BestSwap[0][0],BestSwap[0][1]] \
    - CapCost[BestSwap[0][0],BestSwap[1][1]]

    # Update Crem for selected agent 2
    Crem[BestSwap[1][0]] = Crem[BestSwap[1][0]] \
    + CapCost[BestSwap[1][0],BestSwap[1][1]] \
    - CapCost[BestSwap[1][0],BestSwap[0][1]]

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

[11]:
```python
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
```

4.3 Problem 1

[12]:
```python
slns = list() # All solutions will be stored in this list
zstar = ProbDataSets[0]["zstar"]
```

4.3.1 Construction Heuristic

[13]:
```python
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.")
```

```
Executed in 0.0 CPU*seconds.
```

Print Solution

[14]:
```
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*")
```

```
X = {
( 0, {7, 9, 10, 11, 14, 15, 19} ),
( 1, {1, 2, 4, 5, 12, 16, 24} ),
( 2, {0, 6, 18, 21, 23} ),
( 3, {3, 20, 13} ),
( 4, {8, 17, 22} ),
}
Z = 487
Crem = [1, 3, 14, 43, 18]
Z* = 438
11.19% larger than z*
```

4.3.2 Improvement Heuristic

Starting Solution:

[15]:
```
slns[0]
```

[15]:
```
({0: [7, 9, 10, 11, 14, 15, 19],
  1: [1, 2, 4, 5, 12, 16, 24],
  2: [0, 6, 18, 21, 23],
  3: [3, 13, 20],
  4: [8, 17, 22]},
 487,
 [1, 3, 14, 43, 18])
```

[16]:
```
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.")
```

```
Local Search is complete!
Executed in 0.046875 CPU*seconds.
```

Iterations

[17]:
```
[sln[1] for sln in slns ]
```

[17]:
```
[487, 477, 469, 467]
```

Print Solution

[18]:
```
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*")
```

```
X = {
( 0, {7, 9, 10, 11, 14, 15, 19} ),
( 1, {1, 2, 4, 5, 12, 13, 16} ),
( 2, {0, 18, 20, 22, 23} ),
( 3, {24, 3, 6} ),
( 4, {8, 17, 21} ),
}
Z = 467
Crem = [1, 3, 6, 11, 10]
Z* = 438
6.62% larger than z*
```

4.4 Problem 2

[19]:
```
slns2 = list()
zstar2 = ProbDataSets[1]["zstar"]
```

4.4.1 Construction Heuristic

[20]:
```
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.")
```

```
Executed in 0.03125 CPU*seconds.
```

Print Solution

[21]:
```
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*")
```

```
X = {
( 0, {2, 36, 13, 15, 19, 27} ),
( 1, {1, 29, 17, 7} ),
( 2, {16, 25, 28, 5} ),
( 3, {26} ),
( 4, {35, 3, 4, 6, 38, 11} ),
( 5, {32, 8, 12, 22, 23, 24} ),
( 6, {34, 37, 39, 9, 10, 31} ),
( 7, {0, 33, 14, 18, 20, 21, 30} )
}
Z = 766
Crem = [10, 30, 42, 59, 12, 12, 25, 3]
Z* = 646
18.58% larger than z*
```

4.4.2 Improvement Heuristic

Starting Solution:

[22]:
```
slns2[0]
```

[22]:
```
({0: [2, 13, 15, 19, 27, 36],
  1: [1, 7, 17, 29],
  2: [5, 16, 25, 28],
  3: [26],
  4: [3, 4, 6, 11, 35, 38],
  5: [8, 12, 22, 23, 24, 32],
  6: [9, 10, 31, 34, 37, 39],
  7: [0, 14, 18, 20, 21, 30, 33]},
 766,
 [10, 30, 42, 59, 12, 12, 25, 3])
```

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:

[23]:
```
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.")
```

```
Local Search is complete!
Executed in 0.1875 CPU*seconds.
```

Iterations

[24]:
```
[sln[1] for sln in slns2 ]
```

[24]:
```
[766, 749, 738, 729, 721, 713, 706, 704, 702, 701, 699, 695, 693]
```

Print Solution

[25]:
```
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*")
```

```
X = {
( 0, {4, 36, 13, 15, 17, 27} ),
( 1, {16, 1, 26, 29} ),
( 2, {25, 2, 19, 6} ),
( 3, {7} ),
( 4, {33, 3, 35, 38, 11, 20} ),
( 5, {32, 5, 12, 22, 23, 24} ),
( 6, {34, 37, 39, 8, 9, 31} ),
( 7, {0, 10, 14, 18, 21, 28, 30} ),
}
Z = 693
Crem = [1, 15, 2, 53, 2, 3, 11, 1]
Z* = 646
7.28% larger than z*
```

4.5 Problem 3

[26]:
```
slns3 = list()
zstar3 = ProbDataSets[2]["zstar"]
```

4.5.1 Construction Heuristic

[27]:
```
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.")
```

```
Executed in 0.046875 CPU*seconds.
```

Print Solution

[28]:
```
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*")
```

```
X = {
( 0, {36, 38, 42, 43, 23} ),
( 1, {0, 4, 11, 15, 19, 26, 29} ),
( 2, {2, 3, 8, 12, 16, 18, 20, 22, 27} ),
( 3, {1, 5, 13, 45, 48, 28} ),
( 4, {10, 46, 14, 21, 25} ),
( 5, {6, 31} ),
( 6, {33, 44, 49} ),
( 7, {24} ),
( 8, {9, 34, 35, 47} ),
( 9, {32, 37, 7, 39, 40, 41, 17, 30} ),
}
Z = 730
Crem = [25, 3, 3, 11, 40, 48, 32, 52, 22, 0]
Z* = 573
27.40% larger than z*
```

4.5.2 Improvement Heuristic

Starting Solution:

[29]:
```
slns3[0]
```

[29]:
```
({0: [23, 36, 38, 42, 43],
  1: [0, 4, 11, 15, 19, 26, 29],
  2: [2, 3, 8, 12, 16, 18, 20, 22, 27],
  3: [1, 5, 13, 28, 45, 48],
  4: [10, 14, 21, 25, 46],
  5: [6, 31],
  6: [33, 44, 49],
  7: [24],
  8: [9, 34, 35, 47],
  9: [7, 17, 30, 32, 37, 39, 40, 41]},
 730,
 [25, 3, 3, 11, 40, 48, 32, 52, 22, 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:

[30]:
```
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.")
```

```
Local Search is complete!
Executed in 0.328125 CPU*seconds.
```

Iterations

[31]:
```
[sln[1] for sln in slns3 ]
```

[31]:
```
[730, 713, 699, 692, 686, 680, 675, 668, 662, 659, 655, 653, 652]
```

Print Solution

[32]:
```
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*")
```

```
X = {
( 0, {36, 47, 23, 24, 28} ),
( 1, {0, 4, 43, 15, 19, 25, 26} ),
( 2, {2, 35, 8, 12, 16, 18, 20, 22, 27} ),
( 3, {1, 38, 45, 13, 49, 21} ),
( 4, {33, 3, 10, 14, 48} ),
( 5, {6, 31} ),
( 6, {42, 44, 46} ),
( 7, {5} ),
( 8, {9, 34, 11, 29} ),
( 9, {32, 37, 7, 39, 40, 41, 17, 30} ),
}
Z = 652
Crem = [7, 1, 1, 2, 6, 48, 15, 40, 7, 0]
Z* = 573
13.79% larger than z*
```