GRASP Terminales

Rafael Andrade Ruíz Capetillo 1

José Miguel de la Mora Álvarez 2
José Luis Lobera del Castillo 3 Samantha Licea Domínguez 4

1° de Diciembre de 2021

1 Introducción

Se conoce como optimizar a encontrar la mejor solución posible dentro de una multiplicidad de soluciones para un problema específico. Es decir, al optimizar se tratan de conseguir que se obtengan los mejores resultados posibles buscando la mejor manera de realizar una actividad. Por otra parte, las metaheurísticas son métodos para resolver problemas computacionales utilizando ciertos parámetros y que toman en cuenta la optimización. Por ello, el presente reporte busca resolver un problema dado, buscandole una aplicación de la vida real y trata de buscar la mejor solución de acuerdo a lo que se pide.

2 Planteamiento del Problema

Se tienen n estaciones de trabajo y m terminales. El costo de asignar una estación de trabajo i a una terminal j es $c_{i,j}$. Cada estación de trabajo consume o demanda w_i unidades de la capacidad de una terminal. La capacidad de una terminal j es u_j.
Objetivo:

Encontrar la asignación del mínimo costo para formar una red de conexiones entre estaciones de trabajo y terminales.
Observaciones:

Cada estación de trabajo debe ser asignada exactamente a una terminal. El costo $c_{i,j}$ se calcula usando las coordenadas de las terminales (x_j, y_j) y las estaciones de trabajo (x_i, y_i) de la siguiente forma:

$$c_{i,j} = round\sqrt{(x_i - x_j) + (y_i - y_j)}$$

Resolver: Generar 5 instancias de prueba adicionales con 100 terminales y 32 estaciones de trabajo, tomando la tabla siguiente como ejemplo (ver Tabla 2.1). Hay que

notar que las demandas de las estaciones de trabajo tienen valores generados aleatoriamente entre 10 y 15, y las capacidades de las terminales con valores generados aleatoriamente entre 1 y 6. Además, las coordenadas x y y para las estaciones de trabajp y las terminales son valores generados aleatoriamente entre 1 y 100.

Terminal (j)	Capacidad (u_j)	(x_j, y_j)
1	5	(54,28)
2	4	(28,75)
3	4	(84,44)
4	2	(67,17)
5	3	(90,41)
6	1	(68,67)
7	3	(24,79)
8	4	(38,59)
9	5	(27,86)
10	4	(07,76)
Estación de trabajo (i)	Demanda (w_i)	(x_i, y_i)
1	12	(19,76)
2	14	(50,30)
3	13	(23,79)

Tabla 2.1 Tabla que ejemplifica una instancia del problema que sirve como apoyo para la solución del problema.

3 Solución

Como se mencionó anteriormente, el problema tiene que relacionarse con algún aspecto de la vida real para mejorar su ilustración. Por ello, el problema anterior fue pensado como sigue:

Se tiene un cierto número de aviones que tienen p número de pasajeros que necesitan bajar del avión hacia las puertas de un aeropuerto. Dichas puertas soportan un determinado número de pasajeros que acaban de llegar (capacidad), por lo que los aviones tienen que llegar en una determinada posición (x,y) para cumplir con lo anterior. La tarea es asignar cuál avión va a qué puerta de forma que no se exceda el número de pasajeros por puerta y que se minimice la distancia que recorre cada avión para llegar a la puerta.

Una vez entendiendo mejor el problema y aplicándolo a lo anterior, se obtuvieron los siguientes elementos para la solución del problema.

3.1 Instancias

Para la generación de las 5 instancias que pide el problema, al ser aleatorias, se utilizó un código en R que se presenta a continuación:

```r
a<-sample(1:6,100,replace=TRUE)
x<-sample(0:100,100,replace=TRUE)
y<-sample(0:100,100,replace=TRUE)

b<-sample(10:15,32,replace=TRUE)
x1<-sample(0:100,32,replace=TRUE)
y1<-sample(0:100,32,replace=TRUE)

for (i in 1:100){

  cat(i,a[i],x[i],y[i],"\n")
}
cat("\n")
for (i in 1:32){
  cat(i,b[i],x1[i],y1[i],"\n")
}
```

Una vez compilado el código, se obtuvieron las instancias, las cuales pueden observarse con mejor detenimiento en Instancias (Ins1.txt ... Ins5.txt).

3.2 Código

El algoritmo que se pidió para resolver el presente problema fue el denominado GRASP. Como se sabe, este algoritmo puede entenderse como un conjunto de procedimientos de búsqueda voraz, aleatorizados y adaptativos. El algoritmo se divide en dos partes principales:

- **Fase Constructiva:** genera una solución factible
- **Fase de Mejora Local:** mejora la solución localmente.

Una vez ejecutadas las fases, la solución se guarda y se hace otra iteración nueva, guardando siempre la mejor solución encontrada al momento.
Por ello, para encontrar la mejor solución de el problema, se procedió a generar el código en Python, el cual se muestra a continuación:

```python
import random
import math
from copy import deepcopy
import statistics as st

class Terminal:
    def __init__(self, numero, demanda, posx, posy):
        self.numero = int(numero)
        self.demanda = int(demanda)
        self.posx = int(posx)
        self.posy = int(posy)
        self.distancias = {}  # numero de estacion: distancia
```

```
13
14      def getNumero(self):
15          return self.numero
16
17      def getDemanda(self):
18          return self.demanda
19
20      def getPosx(self):
21          return self.posx
22
23      def getPosy(self):
24          return self.posy
25
26      def __str__(self):
27          return "{}: {}, ({},{}), {}".format(self.numero, self.demanda,
        self.posx, self.posy, self.distancias)
28
29  class Estacion:
30      def __init__(self, numero, capacidad, posx, posy):
31          self.numero = int(numero)
32          self.capacidad = int(capacidad)
33          self.posx = int(posx)
34          self.posy = int(posy)
35
36      def getNumero(self):
37          return self.numero
38
39      def getCapacidad(self):
40          return self.capacidad
41
42      def getPosx(self):
43          return self.posx
44
45      def getPosy(self):
46          return self.posy
47
48      def __str__(self):
49          return "{}: {}, ({},{})".format(self.numero, self.capacidad, self
        .posx, self.posy)
50
51  def calcularDistancia(terminal, estacion):
52      return round(math.sqrt( ((estacion.getPosx() - terminal.getPosx()) **
         2) + ((estacion.getPosy() - terminal.getPosy()) ** 2)))
53
54  for fileIdx in range(1,6):
55      print('INSTANCIA {}:'.format(fileIdx))
56
57      f = [line.split(' ')[:4] for line in open('./Instancias/Ins{}.txt'.
        format(fileIdx)).readlines()]
58
59      terminalesLst = f[:100]
60      estacionesLst = f[101:]
61
62
```

```
63      def getEstacion(numero, estaciones):
64          for estacion in estaciones:
65              if estacion.numero == numero:
66                  return estacion
67
68      def getDistanciaTotal(d):
69          distanciaTotal = 0
70          for valor in d.values():
71              distanciaTotal += valor[1]
72          return distanciaTotal
73
74      def getMejorResultado(resultados):
75          distancias = [x[1] for x in resultados]
76          return min(distancias)
77
78      def getPeorResultado(resultados):
79          distancias = [x[1] for x in resultados]
80          return max(distancias)
81
82      def getDesviacion(resultados):
83          distancias = [x[1] for x in resultados]
84          return st.stdev(distancias)
85
86      def GRASP(terminales, estaciones):
87          resultado = {} # NumeroDeTerminal : NumeroDeEstacion, distancia
88
89          # GREEDY
90
91          for terminal in terminales:
92              i = 0
93              while getEstacion(terminal.distancias[i][0], estaciones).
    getCapacidad() < terminal.getDemanda():
94                  i += 1
95                  if i >= len(terminal.distancias): break
96
97              if i >= len(terminal.distancias): i -= 1
98
99              getEstacion(terminal.distancias[i][0], estaciones).capacidad
    -= terminal.demanda
100             distancia = terminal.distancias[i][1]
101             resultado[terminal.numero] = [getEstacion(terminal.distancias
    [i][0], estaciones).numero, distancia]
102
103         # LOCAL SEARCH
104
105         return resultado
106
107     def algoritmo(terminales, estaciones, iteraciones=100):
108         resultadosFinales = []
109         # Calcular la distancia de cada terminal a cada estacion n*m *
    nlogn
110         for terminal in terminales:
111             for estacion in estaciones:
112                 terminal.distancias[estacion.getNumero()] =
```

```
    calcularDistancia(terminal, estacion)
113             terminal.distancias = sorted(terminal.distancias.items(), key
    =lambda x: x[1])
114
115
116         distanciasSum = 0
117
118         for _ in range(iteraciones):
119             copyTerminales, copyEstaciones = deepcopy(terminales),
    deepcopy(estaciones)
120             random.shuffle(copyTerminales)
121             random.shuffle(copyEstaciones)
122
123             resultado = GRASP(copyTerminales, copyEstaciones)
124             distanciaTotal = getDistanciaTotal(resultado)
125             distanciasSum += distanciaTotal
126             resultadosFinales.append([resultado, distanciaTotal])
127
128             '''
129             for x,y in resultado.items():
130                 print(x,y)
131             print()
132             '''
133
134
135         print('Media:', distanciasSum/iteraciones)
136         print('Mejor:', getMejorResultado(resultadosFinales))
137         print('Peor:', getPeorResultado(resultadosFinales))
138         print('Desviacion:', getDesviacion(resultadosFinales))
139
140     terminalesOriginal = [Terminal(terminal[0], terminal[1], terminal[2],
     terminal[3]) for terminal in terminalesLst]
141     estacionesOriginal = [Estacion(estacion[0], estacion[1], estacion[2],
     estacion[3]) for estacion in estacionesLst]
142     algoritmo(terminalesOriginal, estacionesOriginal)
```

El algoritmo básicamente ordena primero, para todas las terminales, sus mejores estaciones dependiendo de la distancia generada previamente. Posteriormente, se barajea la lista donde se tienen tanto las estaciones como las terminales para que el GRASP cumpla con su función aleatoria, y después las terminales se van a ir sacando para intentar meterlas en su mejor estación. En caso de que no quepan en su primera mejor estación, se va a la segunda mejor, y asi sucesivamente, de forma que quepan. Ya que se le ha encontrado lugar a la mejor y están ordenadas, se regresa el resultado y se hace la suma de todas las distancias para saber la distancia total. Adicionalmente, se sacan las distancias promedio, mínima y máxima, así como la desviación estandar. Este código se repite para cada instancia, las cuales se iteran en el código a través de un for. Se podría decir que el algoritmo se xompone y explica de tres partes principales:

- **Pre-procesamiento:** Generar el diccionario que contiene las distancias de cada terminal con todas las estaciones.
- **Fase constructiva:** Generar un orden aleatorio de las terminales las cuales irán

ocupando un lugar dentro de las estaciones de forma Greedy.

- **Fase de Mejora:** Cada terminal se asignará con la mejor opción en la que quepa todavía.

4 Observaciones y Resultado final

Una vez realizadas todas las tareas necesarias para resolver el problema de las Terminales, se pudo observar que el hecho de haber ordenado las variables antes de realizar el algoritmo Greedy, se logró bajar de 500 a 200 en alguno de los casos. Esto puede considerarse como una mejora para el algoritmo que ayuda a optimizar el resultado del GRASP. El resultado final se puede visualizar en la Tabla 4.1.

INSTANCIA 1	
Media	1870.07
Mejor	1697
Peor	2086
Desviación	81.54756373746031
INSTANCIA 2	
Media	1567.64
Mejor	1420
Peor	1742
Desviación	70.43307877594579
INSTANCIA 3	
Media	1621.09
Mejor	1460
Peor	1821
Desviación	76.55355226007715
INSTANCIA 4	
Media	1189.61
Mejor	1099
Peor	1311
Desviación	46.942882766445464
INSTANCIA 5	
Media	1421.24
Mejor	1271
Peor	1582
Desviación	62.12111337760075

Tabla 4.1 En la tabla se muestran organizados los resultados que corresponden a la distancia media, la distancia mejor, la distancia peor y la desviacion estándar para el problema de Terminales.

El resultado, las instancias y los códigos pueden visualizarse completos en Github.