Progetto AMOD
Single-Source Capacitated Facility Location Problem (SSCFLP)

Anno accademico 2022/2023

Francesco Bernardini

Matricola: 0338264

Indice

Introduzione

Il Single-Source Capacitated Facility Location Problem (SSCFLP) è un problema dove ogni cliente deve essere assegnato a una facility che fornisce la sua intera domanda.
Inoltre, la domanda totale dei clienti assegnati a una facility non può superare la capacità di quest'ultima ed in caso di utilizzo di una facility, per uno o più clienti, bisogna pagare un suo costo di setup.
L'obiettivo in questo problema è minimizzare il costo totale di apertura delle facility e il costo totale di trasporto dei beni dalle facility allocate ai clienti ad esse assegnate.

Problema di localizzazione SSCFLP

Abbiamo la seguente funzione obiettivo:

$$min \sum_{u \in S} f_u x_u + \sum_{u \in S} \sum_{v \in D} c_{uv} y_{uv}$$

I seguenti vincoli:

$$(I) \qquad \sum_{u \in S} y_{uv} = 1 \qquad \forall\, v \in D$$

$$(II) \qquad \sum_{v \in D} d_v y_{Av} \leq k_A x_A \qquad \forall\, A \in S$$

Indichiamo con:

$$fu = costo\ di\ attivazione\ della\ facility\ u$$

$$c_{uv} = costo\ di\ trasporto\ dei\ beni\ dalla\ facility\ u\ al\ cliente\ v$$

$$d_v = la\ domanda\ del\ cliente\ v\ assegnata\ alla\ facility\ u$$

$$k_A = la\ capacità\ della\ facility\ A$$

Inoltre:

$$x_u = \begin{cases} 1, & se\ facility\ u\ allocata \\ 0, & altrimenti \end{cases} \qquad \forall\, u \in S$$

$$y_{uv} = \begin{cases} 1, & se\ cliente\ v\ viene\ servito\ dalla\ facility\ u \\ 0, & altrimenti \end{cases} \qquad \forall\, (u, v) \in SxD$$

Euristico 1

In questa prima euristica ogni cliente si collega alla facility con cui ha costo di trasporto minore se quest'ultima non ha ancora superato la capacità di cui dispone.
Se il cliente trovasse la facility con costo di trasporto minore inutilizzabile (cioè, che ha superato la sua capacità o che comunque non può soddisfare la richiesta), il cliente tenterà di collegarsi con la seconda facility con cui ha costo di trasporto minore e così via.
Questa soluzione minimizza il più possibile i costi di trasporto totali tra facility e clienti.

Euristico 2

Questa euristica funziona nel seguente modo:
Prendiamo metà delle facility disponibili e cerchiamo di associare tutti i clienti in una delle strutture del nuovo sottogruppo.
Qualora si superassero tutte le capacità delle facility nel sottogruppo allora si ne prenderà una nuova (si sceglie la facility con capacità più elevata per semplicità) e si inizierà ad associare i clienti a questa nuova facility.
Questa soluzione è una via di mezzo tra le altre due euristiche implementate e non si focalizza sul minimizzare principalmente uno dei due tipi di costi.

Euristico 3

In questa terza euristica si prende la facility con capacità più alta e si iniziano ad assegnare tutti i clienti a lei fino al raggiungimento della sua capacità. Poi si prenderà la seconda facility con capacità più elevata e a lei si inizieranno ad assegnare i clienti e così via fino a esaurimento clienti.
Così facendo si minimizzano il numero di facility che attiviamo e di conseguenza si punta a minimizzare i costi di setup.

Implementazioni

Classe main

writeFile()

Qui abbiamo la funzione per scrivere il file. In argomento è passata la lista che contiene tutti i dati e nel ciclo for si scrive ogni riga del file csv che viene creato per contenere i dati.

```
def writeFile(lis):

    file = open("dati.csv","w")
    num_file = int(len(lis)/9)
    x = 0
    file.write("File;F.O. SSCFLP;Tempo SSCFLP;F.O. Euristica 1;Tempo Euristica
1;F.O. Euristica 2;Tempo Euristica 2;F.O. Euristica 3;Tempo Euristica 3\n")
    for i in range(num_file):
        file.write(str(lis[x]))
        file.write(";")
        file.write(str(lis[x+1]))
        file.write(";")
        file.write(str(lis[x+2]))
        file.write(";")
        file.write(str(lis[x+3]))
        file.write(";")
        file.write(str(lis[x+4]))
        file.write(";")
        file.write(str(lis[x+5]))
        file.write(";")
        file.write(str(lis[x+6]))
        file.write(";")
        file.write(str(lis[x+7]))
        file.write(";")
        file.write(str(lis[x+8]))
        file.write("\n")
        file.flush()
        x = x + 9
    file.close()
```

createIstances()

In questa funzione vengono estratti i valori dai file, messi nelle loro opportune liste e poi dati alle classi che implementano gli algoritmi e che ci ritorneranno i valori delle funzioni obiettivo.
Inoltre prima di chiamare le istanze degli algoritmi e subito dopo la loro fine viene chiamata la funzione time(). Così che per ogni valore della funzione obiettivo si ha il tempo che è stato impiegato per poter ottenere quella soluzione.
Infine poi si chiama la funzione writeFile() che creerà il file csv che contiene ogni dato ricavato.

```
def createIstances():
    dir = os.listdir('./Yang_Istances')
    lis = []

    for path in dir:

        num_path = path
        path = os.listdir('./Yang_Istances/'+path)
        for ist in path:
            S = 0  # inizializzo il numero di possibili fornitori
            D = 0  # inizializzo il numero di clienti
            capacita = []
            costo_setup = []
            domande = []
            costi = []
            lis.append(ist)
            file = open("./Yang_Istances/" + num_path + "/" + ist, "r")
            riga = file.readline()
            S, D = riga.split()  # con split io leggo di una riga i due numeri
saltando gli spazi

            for i in range(0, int(S)):  # in S righe ho, di ogni deposito, costo
di setup e capacita'
                riga = file.readline()
                cap, set = riga.split()
                capacita.append(int(cap))
                costo_setup.append(int(set))

            riga = file.readline()  # in una riga ho tutte le domande dei
clienti
            for i in range(0, int(D)):
                domande.append(int(riga.split()[i]))

            for i in range(0, int(S)):  # in S righe ho per ogni deposito D
costi di ogni cliente con quel deposito
                riga = file.readline()
                list = []
                for j in range(0, int(D)):
                    list.append(int(riga.split()[j]))
                costi.append(list)

            start = time.time() #prendo il tempo in cui inizio ad eseguire il
modelo con Gurobi
            istance = SSCFLP(int(S),int(D),capacita,costo_setup,domande,costi)
#creo l'istanza del modello
            fo = istance.creazione()
            end = time.time() #prendo il tempo in cui termino di eseguire il
modello con Gurobi
            current = end - start
            lis.append(fo)
            lis.append(round(current,2))

            start = time.time()#prendo il tempo in cui inizio ad eseguire la
```

```
prima euristica
            euri = Euristico(int(S),int(D),capacita,costo_setup,domande,costi)
            fo = euri.creazione()
            end = time.time() #prendo il tempo in cui termino l'esecuzione della
prima euristica
            current = end - start
            lis.append(fo)
            lis.append(round(current, 2))

            start = time.time()#prendo il tempo in cui inizio ad eseguire la
seconda euristica
            euri2 = Euristico2(int(S),int(D),capacita,costo_setup,domande,costi)
            fo = euri2.creazione()
            end = time.time() #prendo il tempo in cui termino l'esecuzione della
seconda euristica
            current = end - start
            lis.append(fo)
            lis.append(round(current, 2))

            start = time.time()#prendo il tempo in cui inizio ad eseguire la
terza euristica
            euri3 = Euristico3(int(S),int(D),capacita,costo_setup,domande,costi)
            fo = euri3.creazione()
            end = time.time() #prendo il tempo in cui termino l'esecuzione della
terza euristica
            current = end - start
            lis.append(fo)
            lis.append(round(current, 2))

    writeFile(lis)
```

Inizializzazione classi

Questi valori sono passati ogni volta che un'istanza di uno degli algoritmi viene creata. (Questa inizializzazione è uguale per tutte e 4 le classi degli algoritmi e per questo la funzione è riportata qui una sola volta anche se presente in tutte le classi).
Qui di seguito abbiamo il significato delle varie variabili che troviamo:

- S è il numero di facility.
- D è il numero di clienti.
- capacita è una lista contenente tutte le capacità delle facility.
- costo_setup è una lista contenente tutti i costi di setup delle facility.
- domande è una lista contenente tutte le domande dei clienti.
- costi è una lista contenente tutti i costi di trasporto tra clienti e facility.

```
def __init__(self, S, D, capacita, costo_setup, domande, costi):
    self.S = S
    self.D = D
    self.capacita = capacita
    self.costo_setup = costo_setup
    self.domande = domande
    self.costi = costi
```

Classe SSCFLP

creazione()

In questa funzione si utilizza Gurobi. L'algoritmo si svolge nel modo seguente:

- Inizializzazione del modello.
- Definizione delle variabili che compongono il modello.
- Definizione dei vincoli del modello.
- Definizione della funzione obiettivo.
- Definizione dell'obiettivo che si ha con questo modello e cioè minimizzazione.
- Infine si esegue il modello con la chiamata della funzione optimize() e poi si ritorna il valore della funzione obiettivo.

```
def creazione(self):

    model = gp.Model("SSCFLP")

    # definisco le variabili
    x = []
    for i in range(0, self.S):
        x.append(model.addVar(vtype=GRB.BINARY, name="x[%s]" % i))

    y = []
    for i in range(0, self.S):
        lista = []
        for j in range(0, self.D):
            lista.append(model.addVar(vtype=GRB.BINARY, name="y[%s][%s]" % (i,
j)))
        y.append(lista)

    # i vincoli
    # Vincoli sulla capacità
    # Sommatoria su v (yuv)*dv <= ku*xu
    model.addConstrs((sum(y[u][v] * self.domande[v] for v in range(0, self.D))
<= self.capacita[u] * x[u] for u in
                     range(0, self.S)),
                    "capacity")

    # Vincoli sul soddisfacimento della domanda dei clienti
    # Sommatoria su u (yuv) == 1
    model.addConstrs((sum(y[u][v] for u in range(0, self.S)) == 1 for v in
range(0, self.D)), "demand")
    # ancora mancano

    # aggiorno il modello
    model.update()

    # definisco la funzione obiettivo
    fo1 = sum(x[i] * self.costo_setup[i] for i in range(0, self.S))
    for i in range(0, self.S):
        fo2 = sum(y[i][j] * self.costi[i][j] for j in range(0, self.D))
#problema nel doppio ciclo
    fo = 0
    fo += fo1
    fo += fo2

    model.setObjective(fo, GRB.MINIMIZE)

    model.update()

    model.optimize()
```

```
    if model.status == GRB.OPTIMAL:
        return model.ObjVal
```

Classe Euristico

creazione()

Viene fatta prima una fase di inizializzazione delle liste utilizzate per la capacità allocata e le variabili x[i] e y[i][j].
Poi inizia un ciclo for dove per ogni cliente si vede la facility che ha il costo minore e se questa ha la capacità disponibile per poter servire il cliente. Trovata la facility più vantaggiosa si aggiorna la variabile x[index] e y[index][i] e la capacità allocata della facility di numero index.
Infine, per il calcolo della funzione obiettivo si andranno a moltiplicare tutte le variabili x[i] con il loro costo di setup, così che incideranno nel calcolo solo le facility che sono state utilizzate. Con lo stesso ragionamento moltiplicheremo tutte le variabili y[i][j] così che nella funzione obiettivo andranno sommati solo i valori dei costi di trasporto tra clienti e facility che sono stati scelti dall'algoritmo euristico.

```
def creazione(self):

    capacita_allocata = []
    for i in range(0,self.D):
        capacita_allocata.append(0)

    x = []
    for i in range(0, self.S):
        x.append(0)

    y = []
    for i in range(0, self.S):
        lista = []
        for j in range(0, self.D):
            lista.append(0)
        y.append(lista)

    for i in range(0, self.D):
        costo = 10000000
        index = 0
        for j in range(0, self.S):
            if self.costi[j][i] < costo and
(capacita_allocata[j]+self.domande[i]) < self.capacita[j]:
                costo = self.costi[j][i]
                index = j
        capacita_allocata[index] += self.domande[i]
        x[index] = 1
        y[index][i] = 1

    fo = 0
    for i in range(0, self.S):
        fo += x[i] * self.costo_setup[i]

    for i in range(0, self.S):
        for j in range(0, self.D):
            fo += y[i][j] * self.costi[i][j]

    return float(fo)
```

Classe Euristico2

creazione()

Anche qui abbiamo una prima fase di inizializzazione delle varie liste. Adesso teniamo conto anche di quali sono le facility inizializzate e nella variabile num_intero abbiamo il numero intero di metà delle facility.
Per ogni cliente controlliamo innanzitutto se non sono state inizializzate almeno metà delle facility, se questo ancora non è avvenuto prendiamo per il cliente preso in considerazione la facility con cui ha un costo di trasporto minore. In caso abbiamo inizializzato almeno la metà delle facility allora cerchiamo di allocare il cliente nella facility presente in lista con costo di trasporto più conveniente e ovviamente con la capacità disponibile.
Qualora tutte le facility presenti in lista non possano soddisfare la domanda del cliente allora se ne inserirà una nuova tra quelle non inizializzate così da soddisfare il cliente.
Infine, anche qua la funzione obiettivo è calcolata moltiplicando tutte le variabili x[i] con il loro costo di setup e tutte le variabili y[i][j] con i loro costi di trasporto così che nella funzione obiettivo andranno sommati solo i valori dei costi delle facility inizializzate e le coppie facility-cliente che sono stati scelte dall'algoritmo euristico.

```
def creazione(self):

    capacita_allocata = []
    for i in range(0,self.D):
        capacita_allocata.append(0)

    x = []
    for i in range(0, self.S):
        x.append(0)

    y = []
    for i in range(0, self.S):
        lista = []
        for j in range(0, self.D):
            lista.append(0)
        y.append(lista)

    facilAlloc = 0
    list_facil = []
    num_intero = int(self.S / 2)

    for i in range(0, self.D):
        costo = 10000000
        index = self.S+1
        if facilAlloc < num_intero:
            for j in range(0, self.S):
                if self.costi[j][i] < costo and
(capacita_allocata[j]+self.domande[i]) < self.capacita[j]:
                    costo = self.costi[j][i]
                    index = j
            if index not in list_facil:
                facilAlloc += 1
                list_facil.append(index)

        else:
            for j in range(0, facilAlloc):
                if self.costi[list_facil[j]][i] < costo and
(capacita_allocata[list_facil[j]]+self.domande[i]) <
self.capacita[list_facil[j]]:
                    costo = self.costi[list_facil[j]][i]
```

```
                    index = list_facil[j]
            if index == self.S+1:
                cap = 0
                for j in range(0, self.S):
                    if j not in list_facil:
                        if self.capacita[j] > cap:
                            cap = self.capacita[j]
                            index = j

                facilAlloc += 1
                list_facil.append(index)
        capacita_allocata[index] += self.domande[i]
        x[index] = 1
        y[index][i] = 1

    fo = 0

    for i in range(0, self.S):
        fo += x[i] * self.costo_setup[i]

    for i in range(0, self.S):
        for j in range(0, self.D):
            fo += y[i][j] * self.costi[i][j]

    return float(fo)
```

Classe Euristico3

creazione()

Anche qui inizializziamo le varie liste e teniamo conto delle facility che sono state allocate.
Siccome l'obiettivo è minimizzare le facility da inizializzare allora prendiamo quelle con capacità più alta.
Viene scelta la facility che ha la capacità più elevata e si iniziano ad assegnare i clienti a lei fino a che essa non potrà più soddisfare la richiesta di un cliente, li allora si prenderà tra le facility che ancora non sono state allocate quella con capacità più alta e così via.
Infine, viene sempre calcolata la funzione obiettivo allo stesso modo degli altri due algoritmi euristici.

```
def creazione(self):

    capacita_allocata = []
    for i in range(0,self.D):
        capacita_allocata.append(0)

    x = []
    for i in range(0, self.S):
        x.append(0)

    y = []
    for i in range(0, self.S):
        lista = []
        for j in range(0, self.D):
            lista.append(0)
        y.append(lista)

    list_facil = []
    index = 0
    cap = 0

    for j in range(0, self.S):
        if self.capacita[j] > cap:
            cap = self.capacita[j]
            index = j
    list_facil.append(index)   #inserisco il primo facility con capacità più
grande
    facilAlloc = 1

    for i in range(0, self.D):
        costo = 10000000
        index = self.S+1
        for j in range(0, facilAlloc):
            if self.costi[list_facil[j]][i] < costo and
(capacita_allocata[list_facil[j]]+self.domande[i]) <
self.capacita[list_facil[j]]:
                costo = self.costi[list_facil[j]][i]
                index = list_facil[j]
        if index == self.S+1:
            cap = 0
            for j in range(0, self.S):
                if j not in list_facil:
                    if self.capacita[j] > cap:
                        cap = self.capacita[j]
                        index = j
            facilAlloc += 1
            list_facil.append(index)
```

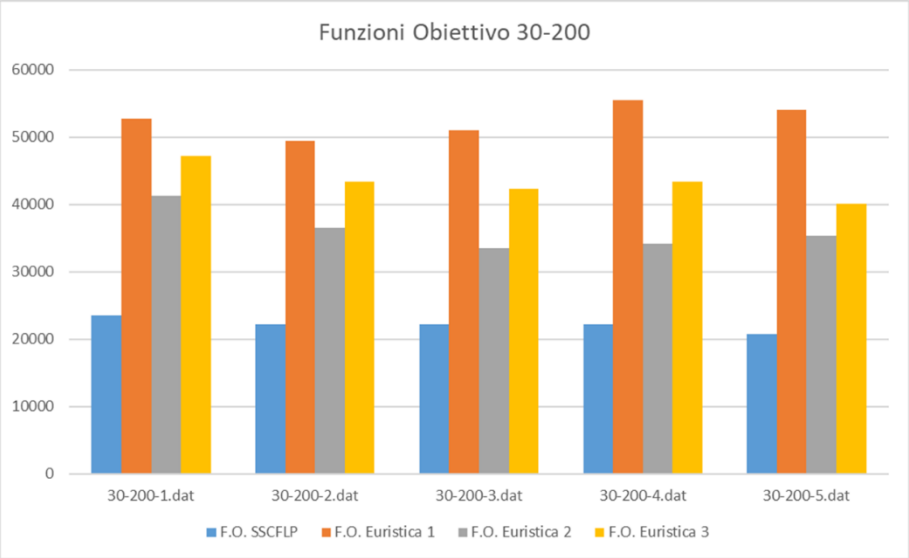
```
            capacita_allocata[index] += self.domande[i]
            x[index] = 1
            y[index][i] = 1

    fo = 0
    for i in range(0, self.S):
        fo += x[i] * self.costo_setup[i]

    for i in range(0, self.S):
        for j in range(0, self.D):
            fo += y[i][j] * self.costi[i][j]

    return float(fo)
```

Risultati ottenuti

Qui di seguito abbiamo i vari risultati ottenuti utilizzando i quattro algoritmi sui vari file di istanze di SSCFLP.

File 30-200

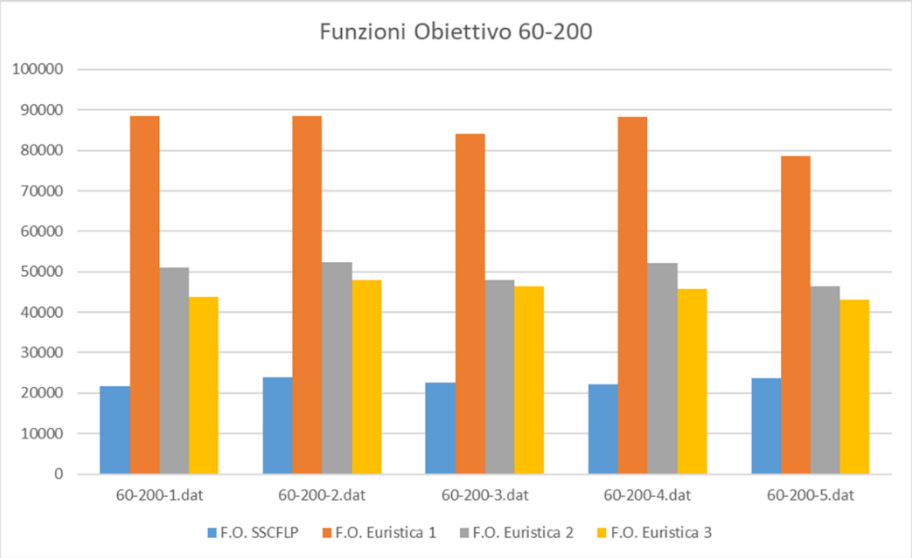
Qui abbiamo i tempi di esecuzione e le funzioni obiettivo ottenute con i file che hanno 30 facility e 200 clienti.



File	Tempo SSCFLP	Tempo Euristica 1	Tempo Euristica 2	Tempo Euristica 3
30-200-1.dat	4,72	0	0,01	0
30-200-2.dat	6,07	0,01	0,01	0,01
30-200-3.dat	0,94	0,01	0,01	0,01
30-200-4.dat	6,11	0	0,01	0,01
30-200-5.dat	1,72	0	0	0

File 60-200

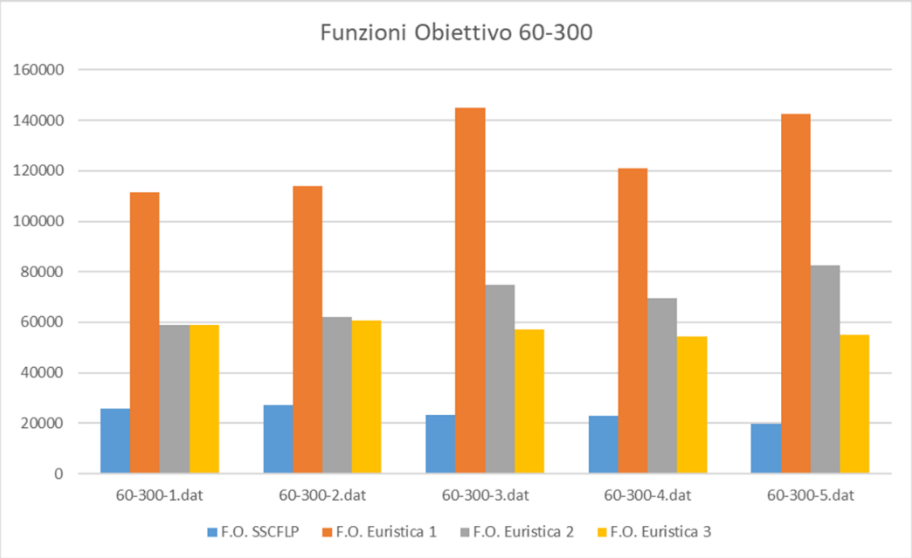
Qui abbiamo i tempi di esecuzione e le funzioni obiettivo ottenute con i file che hanno 60 facility e 200 clienti.



File	Tempo SSCFLP	Tempo Euristica 1	Tempo Euristica 2	Tempo Euristica 3
60-200-1.dat	14,8	0,01	0,01	0,01
60-200-2.dat	11,83	0	0	0,01
60-200-3.dat	8,78	0,01	0,01	0,01
60-200-4.dat	32,36	0,01	0,01	0
60-200-5.dat	3,26	0,01	0,01	0,01

File 60-300

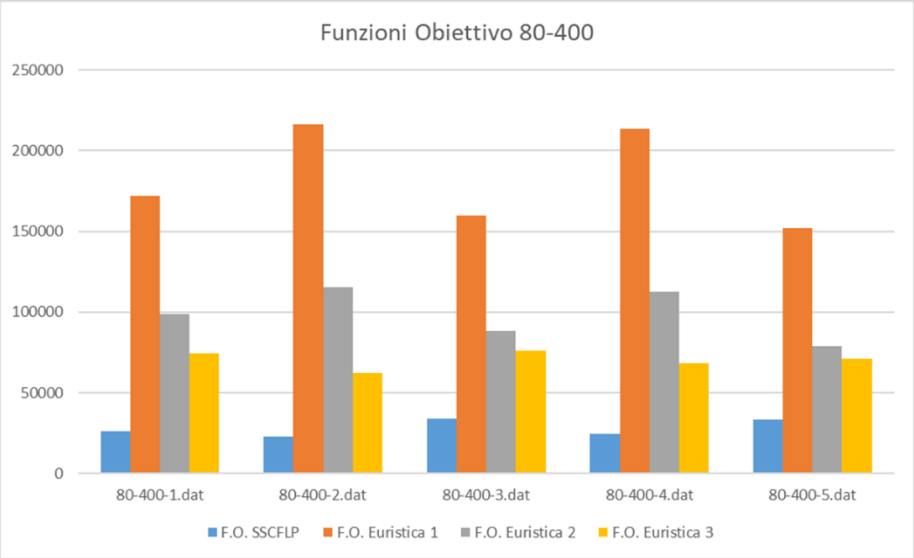
Qui abbiamo i tempi di esecuzione e le funzioni obiettivo ottenute con i file che hanno 60 facility e 300 clienti.



File	Tempo SSCFLP	Tempo Euristica 1	Tempo Euristica 2	Tempo Euristica 3
60-300-1.dat	5,55	0,01	0,02	0,02
60-300-2.dat	186,59	0,02	0,02	0,02
60-300-3.dat	7,42	0,02	0,01	0,01
60-300-4.dat	2,12	0,01	0,01	0,01
60-300-5.dat	7,95	0,01	0,01	0,01

File 80-400

Qui abbiamo i tempi di esecuzione e le funzioni obiettivo ottenute con i file che hanno 80 facility e 400 clienti.



File	Tempo SSCFLP	Tempo Euristica 1	Tempo Euristica 2	Tempo Euristica 3
80-400-1.dat	10,48	0,03	0,02	0,02
80-400-2.dat	2,13	0,01	0,02	0,01
80-400-3.dat	30,86	0,01	0,01	0,02
80-400-4.dat	73,84	0,01	0,01	0,01
80-400-5.dat	9,66	0,05	0,04	0,04

Conclusioni

Dai risultati ottenuti si può vedere come ovviamente l'utilizzo del modello di SSCFLP con Gurobi porti una soluzione nettamente migliore, essendo quest'ultima la soluzione ottima. Dall'altra parte gli algoritmi euristici hanno una velocità di esecuzione molto più rapida rispetto al modello con Gurobi.

Tra le soluzioni euristiche quella che ha le prestazioni peggiori è sicuramente l'euristica 1 dove viene assegnata ogni volta al cliente la facility con cui ha un costo di trasporto minore.

Mentre l'euristica migliore è per la maggioranza delle volte l'euristica 3 dove si punta ad utilizzare il minor numero possibile di facility, difatti questa soluzione è migliore in quasi tutte le istanze tranne nei file con 30 facility e 200 clienti, dove in quel caso l'euristica 2 è la migliore.

Da questi risultati quindi si può vedere come in queste istanze, all'aumentare di facility e clienti, minimizzare i costi di setup è sempre più vantaggioso rispetto a minimizzare i costi di trasporto.