PRÁCTICA DE OPTIMIZACIÓN ENTERA Y COMBINATORIA

Máster interuniversitario en Estadística e Investigación Operativa (MESIO UPC-UB)

Arnau Mercader Luque

Objetivo de la práctica

El objetivo de esta práctica es la resolución del problema del viajante (TSP por sus siglas en inglés) simétrico sobre un grafo completo $G = (V, E)$ con $n = 12$ nodos.

El problema se resolverá mediante diferentes técnicas. Por un lado obtendremos cotas superiores mediante una heurística y por otro lado obtendremos cotas inferiores asociadas a relajaciones del problema.

Idea del TSP

El avance de la optimización ha dependido, entre otros, de los esfuerzos por aportar respuestas al problema del viajante o TSP (**Traveling Salesman Problem**), siendo calificado como una referencia "obligada" para validar métodos de resolución no polinomial, o en términos científicos, problemas "NP-hard".

En términos generales, este problema representa un agente (viajante) para el que se desea responder la siguiente pregunta. ¿ Cómo debe elaborar su itinerario de forma que visite cada ciudad (nodo en nuestra instancia) exactamente una vez, regrese al punto de origen y el coste total sea mínimo ?

Datos del problema

A continuación se puede ver el valor de las distancias (costes) c_{ij}, con $i, j = 1, ..., n$ de mi instancia(**datos 9**). A continuación se muestra la instancia a estudiar y a la vez podemos ver la simetria de esta, es decir, $c_{ij} = c_{ji}$.

0	135	584	318	628	845	372	728	579	746	376	42
135	0	858	819	802	676	830	731	895	116	870	58
584	858	0	960	465	362	200	825	819	296	67	578
318	819	960	0	842	56	180	120	551	859	126	876
628	802	465	842	0	84	32	516	496	725	961	142
845	676	362	56	84	0	842	422	834	999	962	165
372	830	200	180	32	842	0	840	150	327	135	687
728	731	825	120	516	422	840	0	601	592	405	904
579	895	819	551	496	834	150	601	0	336	871	577
746	116	296	859	725	999	327	592	336	0	68	664
376	870	67	126	961	962	135	405	871	68	0	780
42	58	578	876	142	165	687	904	577	664	780	0

Obtención de una cota superior

Como heurística para generar una solución factible para el TSP se usará el algoritmo del vecino más cercano y como mejora de este el intercambio 2 a 2 o técnica 2-opt, que consiste en reemplazar la conexión de dos aristas con otras produciendo un camino nuevo más corto.

Explicación del vecino más cercano

El problema del TSP se resuelve cuando se obtiene un ciclo hamiltoniano, ya que por definición consiste en construir un camino dentro de un grafo, donde se visitan todos los vértices del grafo una sola vez y el último vértice visitado es adyacente al primero, es decir, se crea un circuito cerrado que acaba y empieza en el mismo sitio. Esta heurística se basa en la idea de moverse de una ciudad (nodo) a la siguiente, de tal forma que, de todas las opciones posibles, la ciudad elegida sea la más cercana a donde se encuentra el viajero, es decir, seleccionando aristas de bajo coste. El problema principal es que es una heurística miope, es decir, en una iteración escoge la mejor opción que tiene disponible, sin ver que esto puede obligarle a tomar malas decisiones posteriormente. Al final del proceso probablemente quedarán vértices cuya conexión obligará a introducir aristas de coste elevado. La heurística produce una solución factible, pero no nos garantiza que esta sea la solución óptima.

Pasos del algoritmo para V vértices:

i) Inicialización

- Seleccionar un vértice j al azar
- Hacer $t = j$ y $W = V \setminus \{j\}$

ii) Mientras ($W \neq \emptyset$)

- Tomar j de $W | c_{tj} = min[c_{ti}, i \in W]$
- Conectar t a j
- Hacer $W = W \setminus \{j\}$ y $t = j$

Al final añadir la conexión del último vértice añadido al inicial para crear un circuito hamiltoniano.

Explicación de la mejora 2-opt

Dada una solución con la heurística del vecino más cercano se puede mejorar el circuito. Basta con eliminar dos aristas que se cruzan y reconectar los dos caminos resultantes mediante aristas que no se corten con tal de reducir el coste del ciclo final. El principal problema es que no podemos predecir cuántas iteraciones necesitará la heurística de mejora y que una heurística más simple, incluso una ruta seleccionada al azar, podria ser óptima. Sin embargo, al tener una instancia pequeña, construiremos la heurística de mejora empezando la inicialización con todos los vértices para ver distintos resultados. De esta manera podremos elegir la mejor.

Pasos del algoritmo de mejora:

- Dado un circuito hamiltoniano (heurística inicial) con una ruta asociada.
- (1) Ver si podemos modificar la ruta con los dos nodos asociados al siguiente paso de la ruta. De tal manera que modificamos la ruta 2 a 2, de aqui su nombre 2-opt.
- Si la ruta modificada mejora el coste, cambiar la ruta.
- Si la ruta no mejora el coste, ir al siguiente paso de ruta y volver a aplicar (1).

Al final obtenemos una ruta inferior o igual al coste de la heurística inicial. El procedimiento iterativo usado es el siguiente:

```
# iniciamos la ruta a la obtenida mediante la heurística
ruta anterior = ruta heurística

# doble bucle
for (k in 1:(N-1)){
  for(h in (k+1):N){
# Comparamos distancias
if ((ruta anterior[[k]][1] != ruta anterior[[h]][1]) AND
    (ruta anterior[[k]][1] != ruta anterior[[h]][2])){
# Aplicamos cambio
ruta nueva[[k]] <- c(ruta anterior[[k]][1], ruta anterior[[h]][1])
ruta nueva[[h]] <- c(ruta anterior[[k]][2], ruta anterior[[h]][2])
}
if ( costes(ruta modificada) < costes(ruta anterior) ) {ruta anterior = ruta modificada}}
return (ruta mejorada) }
```

Resultados

La mejor cota superior obtenida es **1781** y se obtiene cuando iniciamos la ruta en el **nodo 10**. Esta cota es mucho mejor que la obtenida mediante otros nodos, que no baja de 2000. Respecto a la ruta inicial mediante la heurística mejoramos la cota en **317 (2098-1781)**.Al abordar el problema empezando con todos los nodos, esto nos permite ver distintas soluciones factibles y elegir de entre todas la mejor. Al mirar solo los dos nodos adyacentes, para algunas iteraciones el cambio no proporciona mejora, y la mejora hace crecer los costes. Existen otras técnicas que permiten ver más allá que un cambio 2-opt pero incrementan la complejidad de la mejora. En la práctica, se dice que un cambio 3-opt incrementa por 3 el tiempo gastado con la mejora 2-opt, sin embargo su cota será más próxima al óptimo. Por lo que la construcción de heurísticas, como con la bondad de ajuste en los modelos estadísticos, supone un compromiso entre eficiencia y complejidad.

A continuación se muestran los siguientes resultados obtenidos para cada nodo inicial $i = 1, ..., 12$.

- Costes mejorados aplicando 2-opt
- Ruta final al aplicar 2-opt
- Ruta inicial al aplicar la heurística
- Coste asociado a la heurística inicial

```
------------------------
RUTA DESDE NODO 1
------------------------
Costes mejorados al ir aplicando opt-2
2023
Ruta final
1 12 2 10 11 3 7 5 6 4 8 9 1
Ruta realizada con vecino más cercano:
1 12 2 10 11 3 7 5 6 4 8 9 1
Coste de la ruta inicial:
2023
------------------------
RUTA DESDE NODO 2
------------------------
Costes mejorados al ir aplicando opt-2
2756 2536 2186 2023
Ruta final
```

```
2 12 1 9 8 4 6 5 7 3 11 10 2
Ruta realizada con vecino más cercano:
2 12 1 4 6 5 7 11 3 10 9 8 2
Coste de la ruta inicial:
2756
------------------------
RUTA DESDE NODO 3
------------------------
Costes mejorados al ir aplicando opt-2
2417 2225 2005
Ruta final
3 11 8 4 6 5 7 9 1 12 2 10 3
Ruta realizada con vecino más cercano:
3 11 10 2 12 1 4 6 5 7 9 8 3
Coste de la ruta inicial:
2417
------------------------
RUTA DESDE NODO 4
------------------------
Costes mejorados al ir aplicando opt-2
2186 2023
Ruta final
4 6 5 7 3 11 10 2 12 1 9 8 4
Ruta realizada con vecino más cercano:
4 6 5 7 11 3 10 2 12 1 9 8 4
Coste de la ruta inicial:
2186
------------------------
RUTA DESDE NODO 5
------------------------
Costes mejorados al ir aplicando opt-2
2639 2476
Ruta final
5 7 3 11 10 2 12 1 4 6 8 9 5
Ruta realizada con vecino más cercano:
5 7 11 3 10 2 12 1 4 6 8 9 5
Coste de la ruta inicial:
2639
------------------------
RUTA DESDE NODO 6
------------------------
Costes mejorados al ir aplicando opt-2
2485 2395 2263
Ruta final
6 4 8 11 3 10 2 1 9 7 5 12 6
Ruta realizada con vecino más cercano:
6 4 8 11 3 7 5 12 1 2 10 9 6
Coste de la ruta inicial:
2485
------------------------
RUTA DESDE NODO 7
------------------------
Costes mejorados al ir aplicando opt-2
2005
```

```
Ruta final
7 5 6 4 8 11 3 10 2 12 1 9 7
Ruta realizada con vecino más cercano:
7 5 6 4 8 11 3 10 2 12 1 9 7
Coste de la ruta inicial:
2005
------------------------
RUTA DESDE NODO 8
------------------------
Costes mejorados al ir aplicando opt-2
2186 2023
Ruta final
8 4 6 5 7 3 11 10 2 12 1 9 8
Ruta realizada con vecino más cercano:
8 4 6 5 7 11 3 10 2 12 1 9 8
Coste de la ruta inicial:
2186
------------------------
RUTA DESDE NODO 9
------------------------
Costes mejorados al ir aplicando opt-2
2005
Ruta final
9 7 5 6 4 8 11 3 10 2 12 1 9
Ruta realizada con vecino más cercano:
9 7 5 6 4 8 11 3 10 2 12 1 9
Coste de la ruta inicial:
2005
------------------------
RUTA DESDE NODO 10
------------------------
Costes mejorados al ir aplicando opt-2
2098 1781
Ruta final
10 11 3 7 9 8 4 6 5 12 1 2 10
Ruta realizada con vecino más cercano:
10 11 3 7 5 6 4 8 9 12 1 2 10
Coste de la ruta inicial:
2098
------------------------
RUTA DESDE NODO 11
------------------------
Costes mejorados al ir aplicando opt-2
2817 2772 2748
Ruta final
11 3 1 12 2 9 7 5 6 4 8 10 11
Ruta realizada con vecino más cercano:
11 3 7 5 6 4 8 10 2 12 1 9 11
Coste de la ruta inicial:
2817
------------------------
RUTA DESDE NODO 12
------------------------
Costes mejorados al ir aplicando opt-2
```

```
2098 2023
Ruta final
12 1 9 8 4 6 5 7 3 11 10 2 12
Ruta realizada con vecino más cercano:
12 1 2 10 11 3 7 5 6 4 8 9 12
Coste de la ruta inicial:
2098
```

Código R programado

```
# algoritmo vecino más cercano + 2-opt
############################

# Función mejorada de list() para añadir valores
expandingList <- function(capacity = 20) {
  buffer <- vector('list', capacity)
  length <- 0

  methods <- list()

  methods$double.size <- function() {
    buffer <<- c(buffer, vector('list', capacity))
    capacity <<- capacity * 2
  }

  methods$add <- function(val) {
    if(length == capacity) {
      methods$double.size()
    }

    length <<- length + 1
    buffer[[length]] <<- val
  }

  methods$as.list <- function() {
    b <- buffer[0:length]
    return(b)
  }

  methods
}


# Buscamos el circuito hamiltoniano de coste mínimo (NN algorithm)

A <- distance
N <- 12; # 12 ciudades (nodos)

# ponemos valor grande en valores i=j
# (el mínimo lo buscaría allí sino)
for (i in 1:N){A[i,i] = 10e6}
```

```
# output's
output_matrix <- list()
output_costs <- list()
output_ruta <- list()
output_mejora_2opt <- expandingList()


for (step in 1:N){
A <- as.matrix(distance)
objective_list <- list()

i = step # vertice azar (de 1 a N)
initial = step

 # ruta añadida
 order <- i

 out1 <- c()
 out2 <- c()
 matrix_distance <- matrix(0,12,12)

# vecino más cercano (NN algorithm)
# lo mismo es vacíar el conjunto W que añadir n-1 nodos al circuito
while( length(order) < N ) {

degree <- TRUE

while(degree) {



j <- which.min(A[i,]) # distancia + cercana de nodo i

# sino está en la ruta
if (!(j %in% order)){

# conectamos nodos
matrix_distance[i,j] <- 1
matrix_distance[j,i] <- 1

# añadimos nodo
order <- c(order, j)

# añadimos coste
out2 <- c(out2,distance[i,j])

# buscar otro nodo
degree <- FALSE
i = j # conexión nodos

# donde nos dirigimos
out1 <- c(out1,j)
```

```
}
# nodo ya visitado (está en ruta)
else {A[i,j] <- 10e6}

}
}

# volvemos al nodo inicial
 matrix_distance[initial,j] <- 1
 matrix_distance[j,initial] <- 1
 out2 <- c(out2,distance[j,initial])
 out1 <- c(out1,initial) # volvemos al inicio
 out1 <- c(initial,out1) # de inicio a ruta
 output_matrix[[step]] <- matrix_distance
 output_ruta[[step]] <- out1
 output_costs[[step]] <- sum(out2)

 ### ALGORITMO DE MEJORA

 order <- as.vector(out1)

 adjacency_list <- list() # create adjacency list
 k <- 1

 for(k in 1:(N-1)){
   adjacency_list[[k]] <- c(order[k],order[k+1]) # guardamos ruta en nuevo objeto
 }

 adjacency_list[[N]] <- c(order[N], order[1])
 adjacency_list

 objective <- function(rute, costs_dist){

   cost <- 0
   for(y in 1:length(rute)){
     cost <- cost + costs_dist[rute[[y]][1], rute[[y]][2]]
     #calcula la suma total de costes del circuito realizado.
   }

   cost
 }



 ADJACENCY_LIST <- adjacency_list
 OBJECTIVE_LIST <- objective(adjacency_list, distance )

 for (k in 1:(N-1)){
   for(h in (k+1):N){

     # Comparamos la lista de distancias
     if ((ADJACENCY_LIST[[k]][1] != ADJACENCY_LIST[[h]][1]) &&
         (ADJACENCY_LIST[[k]][1] !=  ADJACENCY_LIST[[h]][2])){
```

```
      # Miramos el cambio 2-opt
      ADJACENCY_LIST[[k]] <- c(adjacency_list[[k]][1], adjacency_list[[h]][1])
      ADJACENCY_LIST[[h]] <- c(adjacency_list[[k]][2], adjacency_list[[h]][2])

    }
    if ( objective(ADJACENCY_LIST, distance ) < objective(adjacency_list, distance ) ) {
     #Con coste inferior, modificamos ruta y calculamos costes
      OBJECTIVE_LIST <- c( OBJECTIVE_LIST, objective(ADJACENCY_LIST, distance ))

      sub_order <- order[which(order==
                                adjacency_list[[k]][2]):which(order==adjacency_list[[h]][1])]
      ##Guardamos rutas modificadas

      position <- NULL

      for(i in sub_order){
        position <- c(position, which(order == i)) #posiciones que ocupan
      }

      rev_position <- rev(position) #giramos ruta => cambiamos ruta
      ORDER <- order

      for (t in 1:length(position)){
        ORDER[position[t]] <- order[rev_position[t]]
      }
      order <- ORDER
      orderout <- c(paste("Step",step,sep="|"),"=",order)
      output_mejora_2opt$add(orderout)
      # Valores de todas las rutas mejoradas en lista mejorada

      #reasignación para calcular costes
      for(k in 1:(N-1)){
        adjacency_list[[k]] <- c(order[k],order[k+1])
      }
      adjacency_list[[N]] <- c(order[N], order[1])
    }

    ADJACENCY_LIST <- adjacency_list
  }
 }

 cat("------------------------\n")
 cat("RUTA DESDE NODO",step,"\n")
 cat("------------------------\n")
 cat("Costes mejorados al ir aplicando opt-2 \n")
 cat(OBJECTIVE_LIST,"\n")
 cat("Ruta final","\n")
 cat(order,"\n")
 cat("Ruta inicial realizada con vecino más cercano: \n")
 cat(unlist(output_ruta[step]),"\n")

}
```

Formulación 1 del problema TSP

En este apartado, consideramos la siguiente formulación para el problema:

$$(P1) \quad Min \quad \sum_{(i,j)\in E} c_{ij}\, x_{ij}$$

$$\sum_{i<j} x_{ij} + \sum_{j<i} x_{ji} = 2 \qquad i \in V \qquad (1)$$

$$\sum_{i,j\in S} x_{ij} \leq |S| - 1 \qquad S \subset V \qquad (2)$$

$$x_{ij} \in \{0,1\} \qquad (i,j) \in E \qquad (3)$$

Explicación del método

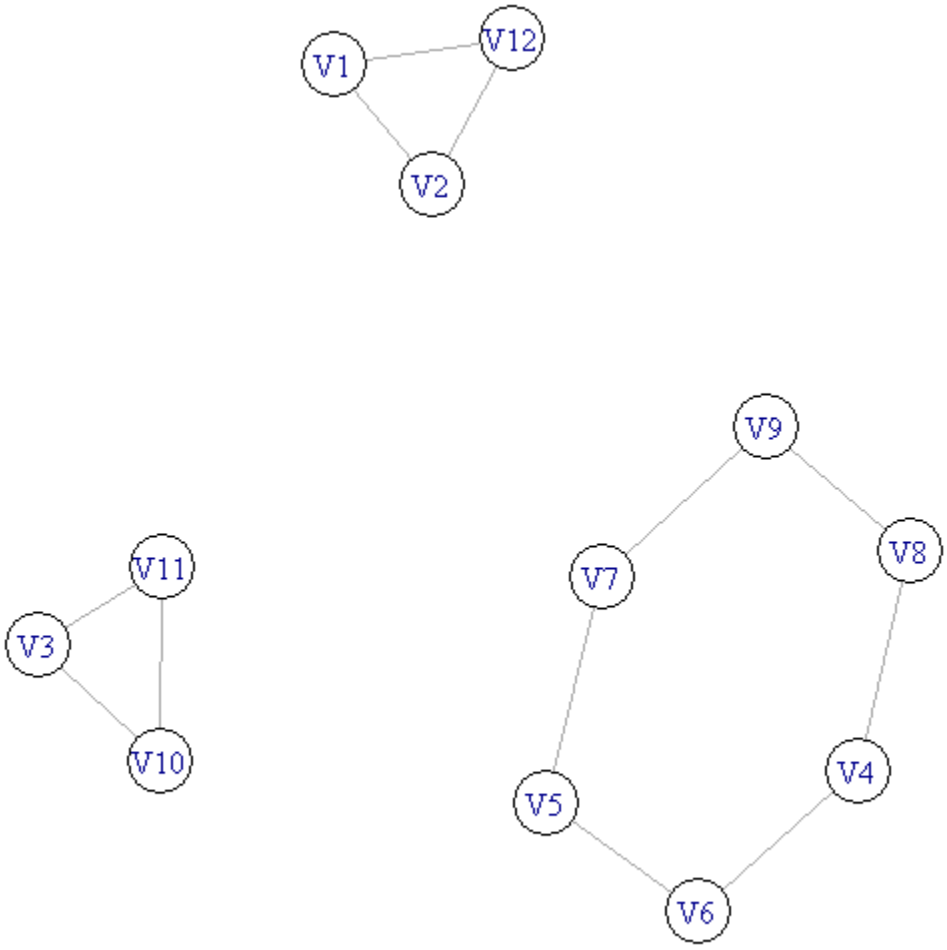
Con esta formulación del TSP, lo que se pretende es relajar el problema eliminando las restricciones (2), ya que es una restricción costosa ya que controla todos los subcircuitos que se puedan crear. Resolveremos el problema y dibujaremos el grafo generado para observar gráficamente si la solución presenta algun subcircuito. Si observamos algun subcircuito, procederemos a crear una restricción adicional para controlarlo y volveremos a resolver el problema anterior añadiendo la nueva restricción para controlar el subcircuito.

Resolución inicial

Resolvemos nuestro problema relajado asociado a nuestra instancia de distancias y obtenemos una solución entera. La cota obtenida es **1709** y la matriz de variables x_{ij} es la siguiente:

0	1	0	0	0	0	0	0	0	0	0	1
0	0	0	0	0	0	0	0	0	0	0	1
0	0	0	0	0	0	0	0	0	1	1	0
0	0	0	0	0	1	0	1	0	0	0	0
0	0	0	0	0	1	1	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	1	0
0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0

Si graficamos nuestra solución vemos que se crean 3 subcircuitos.



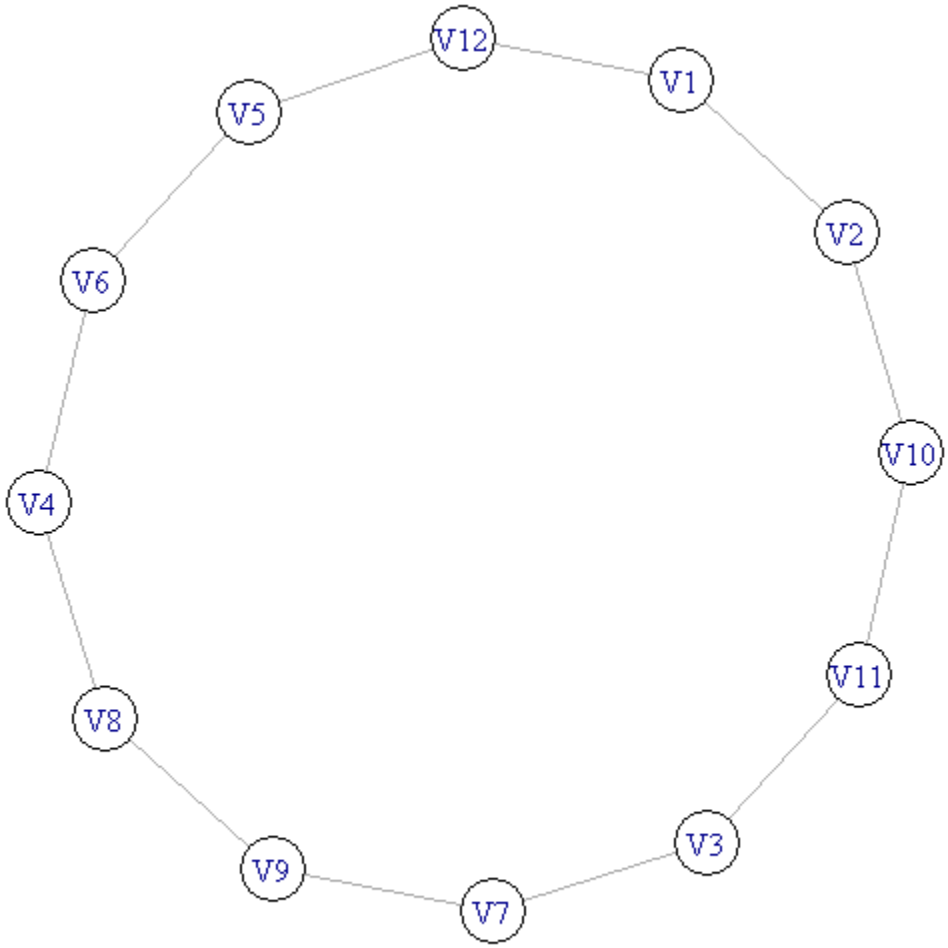
Podemos añadir la restricción asociada a los nodos 1, 2 y 12. Esta restricción tiene que tomar valor como mucho 2 ya que si juntamos dos de los nodos con un pseudonodo el flujo que recibe es 2 cuando por definición de nuestro problema, la variable de decisión x_{ij} toma valor 1. La suma de nuestras 3 variables de decisión pueden tomar como mucho valor 2 ($\leq |S| - 1$), donde en este caso $S = 3$. La restricción a añadir al problema es:

$$x_{1\ 2} + x_{1\ 12} + x_{2\ 12} <= 2$$

Resolución añadiendo restricción de subcircuito

Resolvemos nuestro problema relajado asociado a nuestra instancia de distancias añadiendo la nueva restricción y obtenemos una solución entera. La cota obtenida es **1781**, valor que coincide con la heurística mejorada si empezamos la ruta des del nodo 10. Si graficamos nuestra solución vemos que no se crean subcircuitos, por lo que tanto la heurística mejorada como la cota inferior obtenida añadiendo la restricción de subcircuito son soluciones óptimas al problema.

En esta práctica, la instancia a optimizar es pequeña y con una restricción de subcircuito solucionamos el problema, por lo que la relajación es buena. Sin embargo, con instancias más grandes, las combinaciones de subcircuito serán mayores y resolver el problema mediante relajación nos puede llevar mucho más tiempo. Veamos el circuito hamiltoniano que genera nuestra matriz de variables x_{ij}:



Código CPLEX programado

```
/*********************************************
 * OPL 12.5 Model
 * Author: fme
 * Creation Date: 11/01/2018 at 17:35:58
 *********************************************/

int n = ...;
range Cities = 1..n;

int dist[Cities][Cities] = ...;


dvar float+ x[Cities][Cities];

 minimize sum (i in Cities,j in Cities) dist[i][j]*x[i][j];

 subject to{

    forall(i in Cities)
    forall(j in Cities)
      x[i][j] >=0;

    forall(i in Cities)
    forall(j in Cities)
       x[i][j] <= 1;



    forall(i in Cities)
    sum(j in Cities: i<j) x[i][j] + sum(j in Cities: i>j) x[j][i] == 2;

    /* Restricción de subcircuito*/
   x[1][2] + x[1][12] + x[2][12] <= 2;


}

execute {
 /* writeln hace salto de línea */

writeln("//X= ",x);

}
```

Formulación 2 del problema TSP

En este apartado, consideramos la siguiente formulación para el problema:

$$\text{(P1)} \quad Min \quad \sum_{(i,j)\in E} c_{ij}\, x_{ij}$$

$$\sum_{i<j} x_{ij} + \sum_{j<i} x_{ji} = 2 \qquad i \in V \qquad (1)$$

$$x \text{ es un 1 - árbol} \qquad (2')$$

$$x_{ij} \in \{0,1\} \qquad (i,j) \in E \qquad (3)$$

Explicación general del método

Con esta formulación del TSP, lo que se pretende es relajar la restricción 2 ya que de esta manera el problema se reduce a encontrar un 1-árbol de coste mínimo. Para crear este 1-árbol en cada iteración calcularemos el árbol de expansión mínima de los vértices $2, 3, ..., n$ mediante el algoritmo de Prim y finalmente añadiremos los dos costes menores incidentes al vértice 1. Lo que queremos obtener mediante el método del subgradiente es una ruta dónde cada vértice tenga grado 2, es decir, que este conectado exactamente con 2 nodos. En cada iteración del método a implementar, se obtendrán cotas inferiores. Dicho de otra manera, lo que se hace es forzar que el 1-árbol tenga grado 2, ya que de esta manera se obtendrá un ciclo hamiltoniano, que es la definición de TSP.

Explicación algoritmo de Prim

Los pasos a seguir para conseguir un 1-árbol mediante el algoritmo de Prim son los siguientes:

- Iniciar un árbol A con 1 único vértice j.
- Definir B como los vértices sin visitar. $B \setminus \{j\}$

i) Mientras $(B \neq \emptyset)$

- Tomar k de $A | c_{tk} = min[c_{ti}, i \in B]$, t es cualquier vértice en A.
- Añadir vértice k al árbol A.
- Quitar vértice k del conjunto a visitar. $B \setminus \{k\}$

ii) Juntar el nodo 1 con los dos nodos adyacentes con coste menor.

Empezamos la ruta con aquel vértice j que tenga un nodo adyacente a él con menor coste. Descartamos el nodo 1 ya que lo añadiremos al final para formar el 1-árbol, juntado este con los 2 nodos adyacentes con coste más bajo, obligando así a que se cree un subcircuito.

Código R para calcular 1-árbol

```
prim_1tree <- function(A) {

  A <- as.matrix(A)
  n <- nrow(A)
  # quitamos la fila/columna de A

  costs <- A
  for (i in 1:n) {costs[i,i] <- 10e6; costs[i,1] <- 10e6; costs[1,i] <- 10e6}

  # donde empezar
  cost2  = 10e6
  for (i in 1:n) {
    if(cost2 > min(costs[i,])) {cost2 = min(costs[i,]);i_initial=i;}

  }


  intree <- i_initial;
  notintree <- 1:n; notintree <- notintree[! notintree %in% intree]
  number_of_edges = 0;

  number_in_tree = 1;
  number_notin_tree = n-1;


  en = intree
  out = notintree

  mst<-matrix(0,11,2)
  costs_v<-vector()
  adjacencies<-matrix(0,12,12)

  mincost=10e6;
  while(number_in_tree<n-1){
    for(i in en){
      for(j in out){
        if(costs[i,j] < mincost){
          mincost=costs[i,j];isave=i;jsave=j;


        }
      }
    }



    number_of_edges = number_of_edges + 1
    mst[number_of_edges,1] = isave;
    mst[number_of_edges,2] = jsave;

    adjacencies[isave,jsave]<-1
```

```
    adjacencies[jsave,isave]<-1

    costs_v <- c(costs,mincost)
    # actualizamos el coste mínimo
    mincost = 10e6;

    number_in_tree = number_in_tree + 1


    # actualizamos en & out

    en <- c(en,jsave);
    out <- out[! out %in% jsave];


  }

  ## juntamos el 1-T con el nodo 1

  for (i in 1:n) {A[i,i]=10e6}

  j1 = which.min(A[1,]);
  adjacencies[1,j1]<-1
  adjacencies[j1,1]<-1

  mst[number_of_edges,1] = j1;
  mst[number_of_edges,2] = 1;

  number_of_edges = number_of_edges + 1


  # añadimos distancia grande
  A[1,j1] = 10e6


  j1 = which.min(A[1,]);

  adjacencies[1,j1]<-1
  adjacencies[j1,1]<-1

  mst[number_of_edges,1] = j1;
  mst[number_of_edges,2] = 1;




  return(list(MST=mst,ADJ=adjacencies))
}
```

Explicación método iterativo

En cada iteración calcularemos el 1-árbol (notado como **T**) mediante el algoritmo de Prim y restaremos a las distancias los multiplicadores λ_i,λ_j, de tal forma que nuestro problema se reduce a:

$$L(\lambda) = \begin{cases} \min_x \sum_{j<i} c_{ij}x_{ij} + \sum_{i=1}^{n} \lambda_i \left(2 - \sum_{j<i} x_{ji} + \sum_{i>j} x_{ij} \right) \\ \text{s.t.} \\ \quad x \in \mathbf{T} \\ \quad x_{ij} \ binary \end{cases} = \quad L(\lambda) = 2\sum_{i=1}^{n} \lambda_i \; + \begin{cases} \min_x \sum_{j<i} (c_{ij} - \lambda_i - \lambda_j)x_{ij} \\ \text{s.t.} \\ \quad x \in \mathbf{T} \\ \quad x_{ij} \ binary \end{cases}.$$

Para cada iteración el valor $L(\lambda)$ nos irá proporcionando cotas inferiores al problema. Lo que debemos hacer es encontrar el máximo de esta función (dual = $\max(L(\lambda))$) ya que será el óptimo del problema. Para el método del subgradiente actualizaremos los parámetros λ, $l\,max$, δ y α según las fórmulas e ideas proporcionadas en clase. Como cota superior (parámetro UB en la función) se pondrá el valor **1781** ya que es el mejor valor obtenido con la mejora 2-opt. A continuación se muestra el método y se comentan sus resultados.

Resultados mediante el método subgradiente

Nuestra instancia se resuelve con **7 iteraciones** mediante los siguientes parámetros:

- $\delta = 2$
- $l\,max = 2$
- $\alpha = 0.2$
- $UB = 1781$

A comentar, que en nuestra instancia, lo importante es considerar $\delta = 2$ ya que el valor de $L(\lambda)$ mejora en cada paso, por lo que nuestros parámetros $l\,max$, α no juegan un papel relevante con este valor de δ.

Código R para el método del subgradiente

```
subgradient <- function(A,delta,lmax,
                        alfa,UB,max_iter) {


outg <- list()
# parametros necesarios
N = nrow(A)
A <- as.matrix(A)
NewCosts = as.matrix(A)
LB = 0
l = 0
lambda = rep(0,12)
e = rep(1,12)
max = max_iter
iteration = 0


while ( iteration < max) {
```

```
iteration = iteration + 1
cat("\n")
cat("\n","ITERACIÓN",iteration,"\n")
cat("-----------\n")
a <- prim_1tree(NewCosts)[[2]]
X <- a
X <- as.matrix(X)

# para ver evolución 1-T (evolución gráfica)

#outg[[iteration]] <- X

# borramos obj. temporal
a <- NULL



# suma de ruta por filas
X_out <- vector()
for (i in 1:N){X_out <- c(X_out,sum(X[i,]))}

# Lagrangiano
Lagrangian = sum(as.matrix(NewCosts)*X)/2 + 2*(sum(lambda))
cat("Valor lagrangiano:",Lagrangian,"\n")

gradient = (e*2-X_out);

# gradient valor
cat("Valor del gradiente:",gradient,"\n")


#print(Lagrangian)
if(sum(gradient*gradient)==0) {
  # print(prim_tree(NewCosts)[[2]]);
  cat("Solución óptima\n")
  break

  }


# Actualización cota si es mayor
if(Lagrangian > LB) {cat("LB:",LB,"\n"); LB = Lagrangian;  l = 0;}

else{ l = l +1 ;cat("LB:",LB,"\n")}



# sino mejoramos => actualizamos valor delta
if(l==lmax) {
  delta = alfa*delta;
  l=0;
}
```

```
  cat("Valor delta",delta,"\n")
  cat("Valor gradient^2:",sum(gradient*gradient),"\n")

  # Calculo para los siguiente multiplicadores
  step1 = delta*(((UB-Lagrangian)/(sum(gradient*gradient))));

  # Actualización multiplicadores
  lambda = lambda + (as.vector(step1)*(gradient))
  cat("Multiplicadores:",round(lambda,2),"\n")

  # modificación de costes restando multiplicadores i,j
    for (i in 1:N) {
      for(j in 1:N){
        if (i == j) {NewCosts[i,j]=0
        }else { NewCosts[i,j] = A[i,j] - (lambda[i]+lambda[j])}
      }
    }


}
}
```

Veamos la evolución de nuestro 1-árbol graficando los resultados en cada iteración. Los multiplicadores λ se han redondeado a 2 decimales para una mejor visualización.

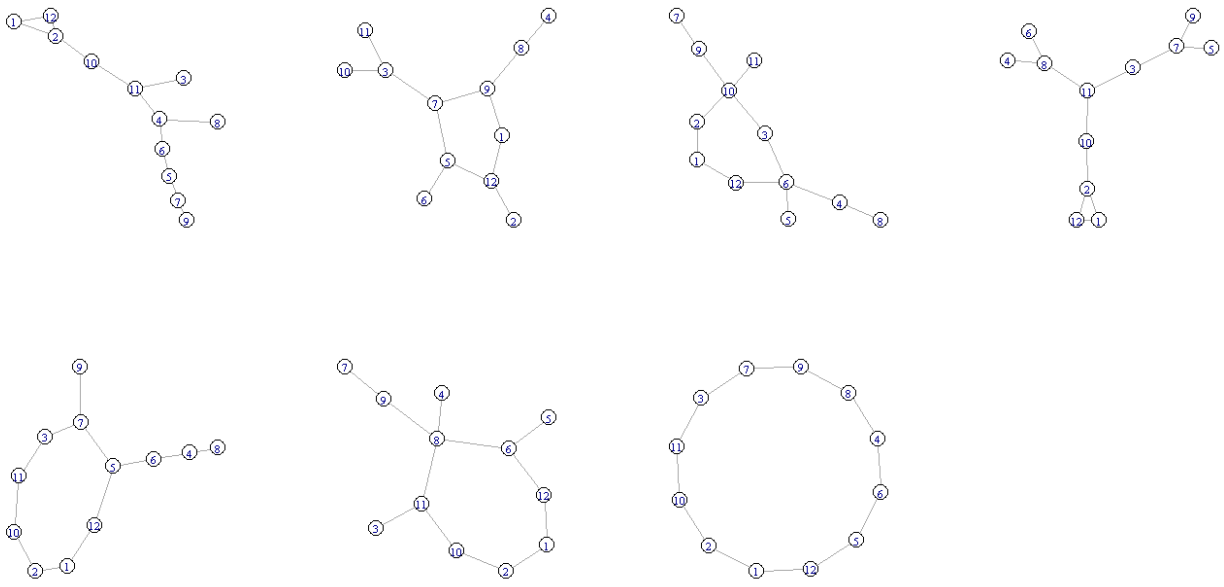
```
subgradient(A=distance,delta=2,lmax=2,alfa=0.2,UB=1781,max_iter=10)
```

```
 ITERACIÓN 1
-----------
Valor lagrangiano: 1054
Valor del gradiente: 0 -1 1 -1 0 0 0 1 1 0 -1 0
LB: 0
Valor delta 2
Valor gradient^2: 6
Multiplicadores: 0 -242.33 242.33 -242.33 0 0 0 242.33 242.33 0 -242.33 0


 ITERACIÓN 2
-----------
Valor lagrangiano: 1159.333
Valor del gradiente: 0 1 -1 1 -1 1 -1 0 -1 1 1 -1
LB: 1054
Valor delta 2
Valor gradient^2: 10
Multiplicadores: 0 -118 118 -118 -124.33 124.33 -124.33 242.33 118 124.33 -118 -124.33


 ITERACIÓN 3
-----------
Valor lagrangiano: 1308.333
Valor del gradiente: 0 0 0 0 1 -2 1 1 0 -2 1 0
LB: 1159.333
```

```
Valor delta 2
Valor gradient^2: 12
Multiplicadores: 0 -118 118 -118 -45.56 -33.22 -45.56 321.11 118 -33.22 -39.22 -124.33


 ITERACIÓN 4
-----------
Valor lagrangiano: 1617.889
Valor del gradiente: 0 -1 0 1 1 1 -1 -1 1 0 -1 0
LB: 1308.333
Valor delta 2
Valor gradient^2: 8
Multiplicadores: 0 -158.78 118 -77.22 -4.78 7.56 -86.33 280.33 158.78 -33.22 -80 -124.33


 ITERACIÓN 5
-----------
Valor lagrangiano: 1742.222
Valor del gradiente: 0 0 0 0 -1 0 -1 1 1 0 0 0
LB: 1617.889
Valor delta 2
Valor gradient^2: 4
Multiplicadores: 0 -158.78 118 -77.22 -24.17 7.56 -105.72 299.72 178.17 -33.22 -80 -124.33


 ITERACIÓN 6
-----------
Valor lagrangiano: 1758.889
Valor del gradiente: 0 0 1 1 1 -1 1 -2 0 0 -1 0
LB: 1742.222
Valor delta 2
Valor gradient^2: 10
Multiplicadores: 0 -158.78 122.42 -72.8 -19.74 3.13 -101.3 290.88 178.17 -33.22 -84.42 -124.33


 ITERACIÓN 7
-----------
Valor lagrangiano: 1781
Valor del gradiente: 0 0 0 0 0 0 0 0 0 0 0 0
Solución óptima
```

Evolución gráfica

Veamos la evolución gráfica de nuestro problema mediante el valor que va tomando nuestro 1-árbol($\mathbf{T}$).



Comentarios

Para nuestra instancia necesitamos 7 iteraciones para obtener la solución óptima. Durante las 4 primeras iteraciones, la cota que proporciona $L(\lambda)$ crece bastante rápido. Después de estas iteraciones las cotas crecen más lentamente hasta obtener **1781**, valor que ya se tenía mediante la cota superior obtenida con la mejora 2-opt de la heurística inicial. Esto nos demuestra, que aplicando el dual, la solución inicial era la óptima. Para esta instancia concreta, con pocas iteraciones hemos podido resolver el problema. El único problema del subgradiente es que tenemos que optimizar nuestros parámetros a priori ya que de estos depende la eficiencia del algoritmo. Aplicando la formulación (1) hemos tenido suerte que con una restricción "adicional" obtenemos el óptimo. Sin embargo, con instancias más grandes podríamos haber tenido que definir muchas más restricciones de subruta, restricciones bastante costosas. Es por eso, que plantear el problema como un 1-árbol puede ser mejor con instancias más grandes.