

# MODELOS Y MÉTODOS DE LA INVESTIGACIÓN OPERATIVA

## Resolviendo el "Traveling Salesman Problem"

Arnaud García

January 19, 2024

### 1 Datos de nuestro TSP

En esta primera sección del trabajo nuestro objetivo es exponer cuál es la motivación detrás de la elección del problema que hemos hecho. Explicar cómo hemos conseguido los datos que usaremos durante el trabajo, y proporcionar una serie de figuras para que el lector pueda comprender mejor el contexto del problema con el que lidiamos antes de afrontarlo. Es importante mencionar, que el problema con el que trabajaremos a lo largo de este trabajo es un problema real.

Toni Medina, mi suegro, es una persona a la que tengo mucho aprecio. Toni lleva más de un año trabajando a tiempo parcial como repartidor de café en la empresa Cafés Mateu. Cuando me enteré de que el trabajo final de esta asignatura sería sobre el problema del viajante de comercio (*Traveling Salesman Problem* en inglés, con siglas TSP), pensé que era una oportunidad ideal para afrontar un problema real, tratar resolverlo con los conocimientos adquiridos durante el curso, y de paso, ayudar a mi suegro.

No son pocas las veces que Toni pierde mucho tiempo en sus rutas de reparto. Lo cual, hace que tenga menos tiempo para llevar a cabo el resto de tareas que tiene. Por eso, le pedí a Toni que me indicara algunos de los bares a los que reparte de forma habitual, es decir, que me compartiera bares donde es posible que tenga que ir en una jornada normal de reparto. Los bares que Toni me envió, junto a su localización, son:

- El Pica Tapas, L'Hospitalet de Llobregat.
- Bar Tropical, Barcelona.
- Bar Chicho, Barcelona.

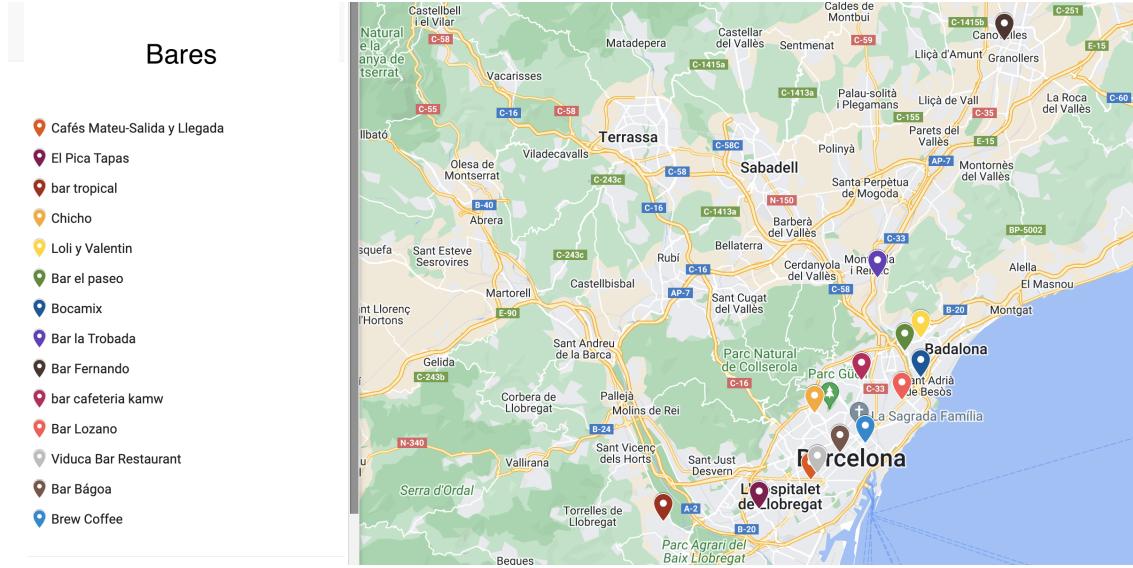


Figure 1: Mapa con los bares donde Toni debe entregar café.

- Marisquería Loli y Valentin, Santa Coloma de Gramanet.
- Bar el Paseo, Santa Coloma de Gramanet.
- Bocamix, Sant Adrià del Besòs.
- Bar la Trobada, Montcada i Reixac.
- Bar Fernando, Canovelles.
- Bar Cafetería Kamw, Barcelona.
- Bar Lozano, Barcelona.
- Bar Restaurant Viduca, Barcelona.
- Bar Bágoa, Barcelona.
- Brew Coffee, Barcelona.

Estos son los bares donde Toni debe ir a repartir, la salida y la llegada la debe hacer desde las oficinas de Cafés Mateu, que se encuentran en Barcelona. Así pues, tenemos un total de 14 lugares por donde Toni debe pasar una sola vez (excepto por Cafés Mateu, de donde debe iniciar la ruta y donde debe finalizarla) en el menor tiempo posible. El lector puede ver estos bares situados en el mapa en la Figura 1.

Podemos observar que la mayoría de bares donde hay que hacer el reparto están en Barcelona, otros se encuentran en las cercanías de esta ciudad. Cuando se trata de tráfico urbano está

	Tiempo estimado entre bares.													
	0-Cafés Mateu	1-Pica Tapas	2-Bar Tropical	3-Bar Chicho	4-Marisquería Loli y Valentín	5-Bar el Paseo	6-Bocamix	7-Bar la Trobada	8-Bar Fernando	9-Bar Kamw	10-Bar Lozano	11-Bar Restaurant Víduca	12-Bar Bágua	13-Brew Coffee
0-Cafés Mateu		19	25	21	36	33	37	39	56	35	35	6	17	25
1-Pica Tapas	19		18	20	32	28	32	36	45	32	32	29	35	39
2-Bar Tropical	25	18		28	38	33	37	40	50	39	37	36	41	42
3-Bar Chicho	21	20	28		22	18	23	27	37	24	21	28	18	27
4-Marisquería Loli y Valentín	36	32	38	22		9	16	22	36	26	15	48	38	34
5-Bar el Paseo	33	28	33	18	9		8	16	28	17	10	44	33	30
6-Bocamix	37	32	37	23	16	8		21	32	22	10	44	33	30
7-Bar la Trobada	39	36	40	27	22	16	21		29	23	24	53	47	42
8-Bar Fernando	56	45	50	37	36	28	32	29		31	29	63	53	48
9-Bar Kamw	35	32	39	24	26	17	22	23	31		20	42	32	28
10-Bar Lozano	35	32	37	21	15	10	10	24	29	20		38	27	25
11-Bar Restaurant Víduca	6	29	36	28	48	44	44	53	63	42	38		12	19
12-Bar Bágua	17	35	41	18	38	33	33	47	53	32	27	12		14
13-Brew Coffee	25	39	42	27	34	30	30	42	48	28	25	19	14	

Figure 2: Tabla con los tiempos estimados entre bares.

claro que las distancias son relativas. Un repartidor puede tardar más para ir a un punto más cercano, que a otro más alejado, si la ruta para ir al punto más cercano requiere pasar por calles congestionadas o con un mayor tráfico. Como el objetivo de este estudio es que Toni lleve a cabo su ruta en el menor tiempo posible, he pensado que sería más interesante usar como matriz de costes del problema, una matriz con los tiempos que se tarda para ir de un bar a otro. Para estimar el tiempo que hay entre bares he usado *Google Maps*. He fijado en *Google Maps* el día 14 de Noviembre de 2023, a las 9:30h. Es decir, he fijado un día laborable, en horario laborable, para obtener unas estimaciones de tiempo realistas. Al fijar dichos parámetros de día y hora, *Google Maps* nos devolvía una ventana de tiempo (el tiempo aproximado que se tarda en ir de un bar a otro). Si la ventana de tiempo que nos daba la aplicación era, por ejemplo,  $t_1 - t_2$ , nos quedábamos con  $\lceil(t_1 + t_2)/2\rceil$ . El lector puede observar la tabla resultante de este proceso para recoger los datos en la Figura 2. Todos los tiempos están en minutos.

Cabe destacar que, como en este trabajo se nos pide estudiar un problema TSP simétrico, el tiempo para ir de un bar cualquiera a otro es el mismo que el tiempo que se tarda en hacer el camino inverso. Está claro que esta suposición no es cierta en realidad.

En la tabla de la Figura 2, aparecen los bares numerados, esta será la numeración que usaremos durante todo el trabajo. Antes de pasar a la siguiente sección, es interesante traducir nuestro problema al lenguaje matemático que usaremos a lo largo de toda la memoria. En este problema estamos considerando un grafo completo  $G = (V, A)$ . Donde  $V = 0, 1, 2, \dots, 13$  son los nodos (tenemos un total de  $14 = |V|$ ), en nuestro caso los bares. Y  $A$  son las aristas que conectan los bares. En la Figura 3 el lector puede ver el grafo completo que representa el problema al que nos enfrentamos. En las siguientes secciones encontraremos soluciones

Bares en los que entregar café y tiempos para ir de uno al otro

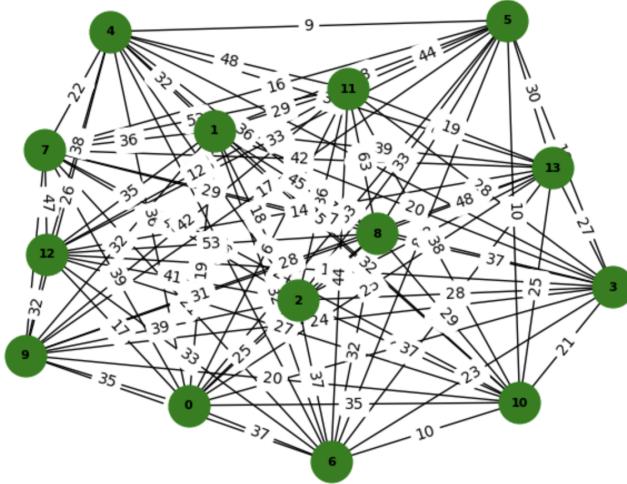


Figure 3: Grafo completo con los 14 nodos representando los bares, y las aristas entre bares con el tiempo qye se tarda en ir de uno a otro.

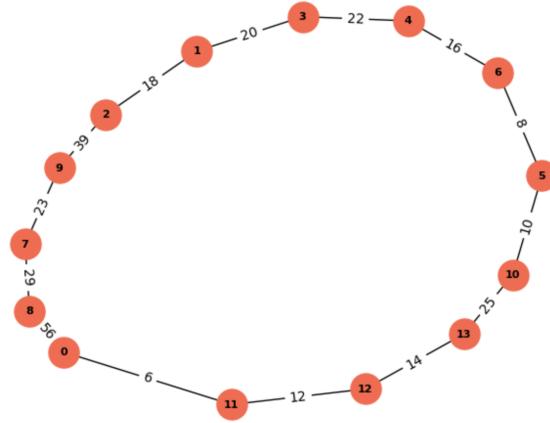
factibles para nuestro problema y trataremos de encontrar la solución óptima.

## 2 Obteniendo una cota superior

En esta sección programaremos una heurística para conseguir una solución factible de nuestro TSP. Esta solución factible nos proporcionará una cota superior de nuestro problema. Esta sección estará compuesta de dos partes, una primera parte en la que implementaremos el método heurístico *Nearest Neighbors* y hallaremos una solución factible de nuestro problema. Y una segunda parte en la que implementaremos el método *Tabu Search* con el objetivo de mejorar la solución factible obtenida previamente con *Nearest Neighbors*. Cabe destacar que las implementaciones de estos métodos se han hecho con *Python*. Al final del trabajo el lector encontrará un anexo con el *Notebook* del código de *Python*. Es decir, en vez de adjuntar en este documento pantallazos de los diferentes códigos usados, el lector puede ir directamente al primer anexo del trabajo y encontrar el notebook completo con comentarios añadidos y el código.

La heurística que he escogido para encontrar una solución factible del problema es *Nearest Neighbors*. Es una heurística muy sencilla de entender y de implementar. Básicamente en esta heurística lo que hacemos es, empezar por un nodo de nuestro grafo, pongamos  $b_i$  (haciendo referencia al bar  $i$ -ésimo). Entonces nos movemos al bar más cercano, es decir nos movemos al nodo  $b_{i+1}$  tal que  $c_{b_i, b_{i+1}} = \min\{c_{b_i, k} : k \in V \setminus \{b_i\}\}$ . Donde  $c_{i,j}$  representan los costes de la matriz de costes. Ahora nos encontramos en el nodo  $b_{i+1}$ , y lo que haremos será movernos al

Solución proporcionada por Nearest Neighbors

Figure 4: Grafo con el tour obtenido tras usar *Nearest Neighbors*.

bar más cercano, y así sucesivamente. En forma de algoritmo, este proceso explicado sería:

1. Inicialización:  $M = \{b_0\}$ ,  $i = 0$ ,  $T = \emptyset$  ( $T$  representa el tour).
2. Mientras  $M \neq V$  o  $T$  no haya visitado todos los nodos. Ecogemos  $b_{i+1} \in V \setminus M$  tal que  $c_{b_i, b_{i+1}} = \min\{c_{b_i, k} : k \in V \setminus M\}$ . Y entonces actualizamos  $M = M \cup \{b_{i+1}\}$ ,  $T = T \cup \{(b_i, b_{i+1})\}$ .
3. Finalmente obtenemos un tour, que es una solución factible de nuestro problema,  $T = T \cup \{(b_{13}, b_0)\}$ .

Hemos implementado esta heurística con *Python*. El lector puede ver el código completo con comentarios añadidos en el Anexo de este trabajo. En el *Nearest Neighbors* que hemos implementado hemos elegido como nodo inicial el nodo 0, es decir, el nodo correspondiente a Cafés Mateu. Después de haber llevado a cabo esta heurística la solución factible que hemos obtenido es el tour formado por las aristas

$$T = \{(0, 11), (11, 12), (12, 13), (13, 10), (10, 5), (5, 6), (6, 4), (4, 3), (3, 1), (1, 2), (2, 9), (9, 7), (7, 8), (8, 0)\}.$$

Este es el tour que corresponde a visitar los bares en el siguiente orden: Cafés Mateu, Bar Restaurant Viduca, Bar Bágua, Brew Coffee, Bar Lozano, Bar el Paseo, Bocamix, Marisquería Loli y Valentin, Bar Chicho, Pica Tapas, Bar Tropical, Bar Kamw, Bar la Trobada, Bar Fernando, Cafés Mateu. El coste de hacer esta ruta es de 298 minutos. El lector puede encontrar este tour en forma de grafo en la Figura 4.

En la segunda parte de esta sección nuestro objetivo es mejorar la solución factible hallada en la primera parte de la sección. Para ello he implementado el método *Tabu Search*. Así pues, lo primero es explicar como funciona este método. Empezamos con una solución inicial

que viene dada con una heurística constructiva, en este caso la solución es la que hemos expuesto anteriormente y que se encuentra en la Figura 4, esta solución será guardada como la mejor hasta la fecha al inicio. Lo que haremos será generar un vecindario completo de esta solución inicial. Entonces elegiremos el mejor de los vecinos que no se encuentre en la lista tabú, en términos del vecino con menor coste. Declararemos el mejor vecino como tabú, es decir lo incluiremos en la lista tabú, de esta forma evitaremos que el algoritmo vuelva a considerar ese vecino más adelante. Entonces comprobaremos si el mejor vecino es mejor solución que la actual, si lo es actualizaremos la mejor solución hallada hasta la fecha. Seguidamente, comprobaremos si se cumplen los criterios de parada, en nuestro caso usaremos un número máximo de iteraciones, y si no se cumple volveremos al paso de generar un vecindario completo de vecinos.

Hay que definir diferentes actores necesarios en este método. Uno de ellos es la tenencia tabú (*tabu tenure* en inglés) que es la duración durante la cual un movimiento se mantendrá tabú. Siguiendo las indicaciones que se dan en el material del curso tomamos como tenencia tabú  $t = 7$ . Lo siguiente que hay que definir es el movimiento en nuestro método *Tabu search*. Para crear nuevas soluciones, se intercambia el orden en que se visitan dos nodos en una solución potencial, esto es el movimiento. En nuestro caso definiremos el movimiento como el intercambio en el orden en el que se visitan dos nodos consecutivos.

Para que el lector pueda entender mejor el método que vamos a usar y los pasos que vamos a seguir, resumimos todo lo explicado de forma esquemática. En el método *Tabu Search* lo que hacemos es:

1. Generar una solución inicial  $x$  y inicializarla como mejor solución hallada  $x^* = x$ . Este paso, en nuestro caso, ya está cubierto. Hemos generado una solución inicial via *Nearest Neighbors*.
2. Generar un vecindario completo de la solución inicial, denotémoslo por  $N(x)$ . En nuestro caso este vecindario lo generaremos usando el movimiento descrito con anterioridad.
3. Elegir  $x' \in N(x)$ , tal que  $x'$  no está en la lista tabú y que  $x'$  es la mejor en términos de tener el menor coste.
4. Añadimos  $x'$  a la lista tabú.
5. Comprobamos si la  $x'$  es mejor solución que  $x^*$ . Si es así, actualizamos la mejor solución hallada:  $x^* = x'$ . En cualquier caso actualizamos  $x = x'$ .
6. Comprobamos si se cumple el criterio de parada. Si es así PARAMOS, y nos quedamos con  $x^*$  como solución mejorada. Si no es así, volvemos al paso 2.

De nuevo, este método ha sido implementado en *Python*. El lector puede ver el código entero con comentarios en el *notebook* anexado al final del trabajo. La solución mejorada que hemos

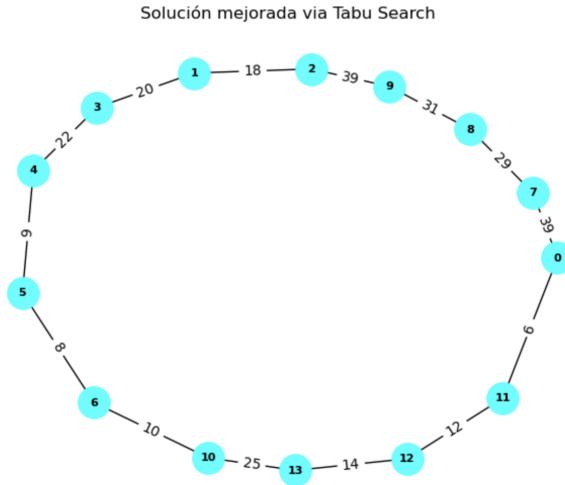


Figure 5: Grafo con el tour obtenido tras usar *Tabu Search*.

obtenido después de usar *Tabu search* es la solución formada por las aristas

$$T' = \{(0, 11), (11, 12), (12, 13), (13, 10), (10, 6), (6, 5), (5, 4), (4, 3), (3, 1), (1, 2), (2, 9), (9, 8), (8, 7), (7, 0)\}$$

Este es el tour que corresponde en visitar los bares en el siguiente orden: Cafés Mateu, Bar Restaurant Vídua, Bar Bágua, Brew Coffee, Bar Lozano, Bocamix, Bar el Paseo, Marisquería Loli y Valentín, Bar Chicho, Pica Tapas, Bar Tropical, Bar Kamw, Bar Fernando, Bar la Trobada, Cafés Mateu. El grafo correspondiente a este tour se expone en la Figura 5.

El coste de esta solución es de 282 minutos, que, en efecto, mejora el coste de 298 minutos de la primera solución hallada vía *Nearest Neighbors*. Hemos obtenido una mejora de  $298 - 282 = 16$  minutos, un cuarto de hora que Toni podrá aprovechar para llevar a cabo otras tareas, o descansar.

### 3 Obteniendo cotas inferiores

Consideramos la siguiente formulación matemática,  $(P)$ , para el TSP simétrico:

$$\begin{aligned}
& \min \sum_{(i,j) \in A} c_{ij} x_{ij} \\
\text{s.t. } & \sum_{j \in \delta(i)} x_{ij} = 2 \quad \forall i \in V \\
& \sum_{(i,j) \in A, i,j \in W} x_{ij} \leq |W| - 1, \quad \forall W \subset V, 2 \leq |W| \leq \frac{n}{2} \\
& x_{ij} \in \{0, 1\} \quad \forall (i, j) \in A
\end{aligned}$$

Donde, por orden de arriba abajo tenemos: la función objetivo, la restricción correspondiente al grado de los nodos (asegura que para cada nodo entre exactamente una arista y salga exactamente una arista), la restricción correspondiente a los subtours (*Subtour Elimination Constraints* en inglés, con siglas SEC) que asegura que no se generen soluciones con subtours, y finalmente la restricción del dominio de la variable  $x_{ij}$ , que debe ser binaria.

El objetivo en esta sección es encontrar la mejor cota inferior posible, o incluso, la solución óptima del problema. Para ello vamos a seguir el siguiente proceso iterativamente:

1. Resolver la relajación lineal del problema ( $P$ ), reemplazando las restricciones  $x_{ij} \in \{0, 1\}$  por  $0 \leq x_{ij} \leq 1$ , e ignorando las *Subtour Elimination Constraints*. Denotemos por  $x^*$  la solución obtenida.
2. Identificar una SEC violada por  $x^*$  aplicando un algoritmo de separación adecuado. Añadir la SEC violada identificada a la relajación lineal y resolver la nueva relajación lineal. Este procedimiento lo podemos repetir hasta 3 veces (según lo que se pide en el enunciado del trabajo).
3. Si no hemos encontrado ninguna solución entera, lo que haremos será identificar alguna desigualdad válida para el TSP (diferente a las SEC) violada por  $x^*$ . Añadir esta desigualdad válida violada a la relajación lineal y resolvérla. Este procedimiento puede ser repetido hasta dos veces, según el enunciado del trabajo.
4. Si no hemos encontrado ninguna solución entera, lo que haremos será identificar un corte de Gomory violado por  $x^*$ . Añadiremos este corte de Gomory a la relajación lineal y la resolvéremos.
5. Finalmente, si todavía no hemos encontrado solución entera, aplicaremos el procedimiento *branch-and-bound* des de la última relajación lineal que hemos obtenido.

Cabe destacar que las diferentes relajaciones lineales se han resuelto usando *AMPL-Gurobi*. El lector puede encontrar los archivos *.mod*, *.run*, *.dat* usados para resolver las diferentes relajaciones lineales anexados al final del trabajo.

```

ampl: include stsp.run
Gurobi 10.0.1: Gurobi 10.0.1: optimal solution; objective 220
13 simplex iterations
TotalTime = 220

x [*,*]
:   1   2   3   4   5   6   7   8   9   10  11  12  13  14  :=
1   0   0   0   0   0   0   0   0   0   0   0   1   0   0
2   0   0   1   0   0   0   0   0   0   0   0   0   0   0   0
3   0   1   0   0   0   0   0   0   0   0   0   0   0   0   0
4   0   0   0   0   0   0   0   0   0   0   1   0   0   0   0
5   0   0   0   0   0   1   0   0   0   0   0   0   0   0   0
6   0   0   0   0   1   0   0   0   0   0   0   0   0   0   0
7   0   0   0   0   0   0   0   0   0   0   0   1   0   0   0
8   0   0   0   0   0   0   0   0   0   1   0   0   0   0   0
9   0   0   0   0   0   0   0   1   0   0   0   0   0   0   0
10  0   0   0   1   0   0   0   0   0   0   0   0   0   0   0
11  0   0   0   0   0   0   1   0   0   0   0   0   0   0   0
12  1   0   0   0   0   0   0   0   0   0   0   0   0   0   0
13  0   0   0   0   0   0   0   0   0   0   0   0   0   0   1
14  0   0   0   0   0   0   0   0   0   0   0   0   1   0   0
;

```

Figure 6: Output de *AMPL* al resolver la primera relajación lineal.

Empezamos resolviendo la primera relajación lineal, la solución que nos devuelve *AMPL* se puede ver en la Figura 6. Obsérvese que la numeración de los nodos en *AMPL* es diferente, hay un decalaje de una unidad. Es decir, en *AMPL* el nodo 1 corresponde al nodo 0, Cafés Mateu. El nodo 2 corresponde al 1, Pica tapas, etc. Obtenemos una solución con un coste de 220 minutos. Esta es la "peor" cota inferior que vamos a encontrar para nuestro problema. Es decir, es la cota que más alejada va estar, por debajo, de la solución óptima de nuestro problema. Sea  $x^{opt}$  la solución óptima (es decir, la solución factible de mínimo coste) de nuestro problema, teniendo en cuenta los resultados obtenidos en la sección anterior, ahora mismo tenemos que

$$220 \leq f(x^{opt}) \leq 282,$$

donde  $f(x^o)$  denota el coste de la solución óptima  $x^{opt}$ .

Tratamos de buscar ahora SEC que sean violadas por las solución de la primera relajación lineal. Si representamos el grafo resultante de la solución obtenemos el grafo de la Figura 7. En este grafo el lector puede observar fácilmente los subtours. Vemos que representando manualmente la solución obtenida obtenemos siete pares de nodos conectados entre sí. Claramente, estos siete pares de nodos son siete subconjuntos de nodos que están violando una SEC.

De esta forma, sin necesidad de aplicar un algoritmo de separación, podemos observar que las siguientes SEC se violan:

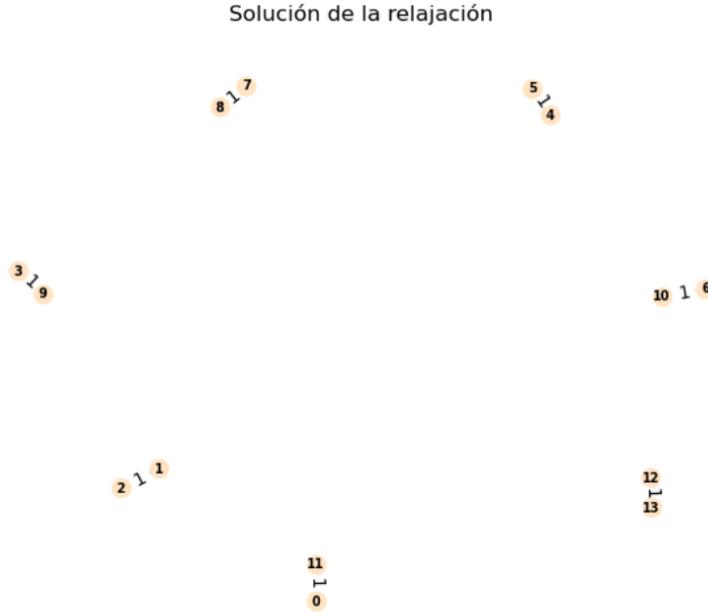


Figure 7: Grafo correspondiente a la solución de la primera relajación lineal.

- $W_1 = \{0, 11\}$ ,  $1 + 1 = 2 \not\leq |W_1| - 1 = 2 - 1 = 1$ .
- $W_2 = \{1, 2\}$ ,  $1 + 1 = 2 \not\leq |W_2| - 1 = 2 - 1 = 1$ .
- $W_3 = \{7, 8\}$ ,  $1 + 1 = 2 \not\leq |W_3| - 1 = 2 - 1 = 1$ .
- $W_4 = \{4, 5\}$ ,  $1 + 1 = 2 \not\leq |W_4| - 1 = 2 - 1 = 1$ .
- $W_5 = \{3, 9\}$ ,  $1 + 1 = 2 \not\leq |W_5| - 1 = 2 - 1 = 1$ .
- $W_6 = \{6, 10\}$ ,  $1 + 1 = 2 \not\leq |W_6| - 1 = 2 - 1 = 1$ .
- $W_7 = \{12, 13\}$ ,  $1 + 1 = 2 \not\leq |W_7| - 1 = 2 - 1 = 1$ .

Así pues, añadimos estas desigualdades a nuestra relajación lineal y la volvemos a resolver. En este caso, obtenemos la solución expuesta en la Figura 8. El coste de la solución de la relajación lineal con las *Subtour Elimination Constraints* añadidas es de 245. En efecto, obtenemos una mejor cota inferior. Ahora, tenemos que el coste de la solución óptima satisface las desigualdades

$$245 \leq f(x^{opt}) \leq 282.$$

De nuevo, tratamos de encontrar SEC violadas, este vez por la nueva solución obtenida al resolver la relajación lineal con las siete desigualdades anteriores añadidas. Si representamos el

```

ampl: include stsp.run
Gurobi 10.0.1: Gurobi 10.0.1: optimal solution; objective 245
27 simplex iterations
TotalTime = 245

x [*,*]
:   1   2   3   4   5   6   7   8   9   10  11  12  13  14      :=
1   0   0   1   0   0   0   0   0   0   0   0   0   0   0   0
2   0   0   0   1   0   0   0   0   0   0   0   0   0   0   0
3   0   1   0   0   0   0   0   0   0   0   0   0   0   0   0
4   0   0   0   0   0   0   0   0   0   0   0   0   0   0   1   0
5   0   0   0   0   0   0   0   0   0   0   0   1   0   0   0   0
6   0   0   0   0   1   0   0   0   0   0   0   0   0   0   0   0
7   0   0   0   0   0   1   0   0   0   0   0   0   0   0   0   0
8   0   0   0   0   0   0   0   0   0   1   0   0   0   0   0   0
9   0   0   0   0   0   0   0   0   0   0   1   0   0   0   0   0
10  0   0   0   0   0   0   0   0   1   0   0   0   0   0   0   0
11  0   0   0   0   0   0   0   1   0   0   0   0   0   0   0   0
12  1   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
13  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   1
14  0   0   0   0   0   0   0   0   0   0   0   1   0   0   0   0
;

```

Figure 8: Output de *AMPL* al resolver la segunda relajación lineal.

grafo de la solución obtenida nos queda el grafo expuesto en la Figura 9. De nuevo, sin necesidad de usar ningún algoritmo de separación, se pueden observar subtours que claramente no satisfacen las correspondientes SEC. En este caso tenemos los siguientes subconjuntos de nodos que violan las siguientes SEC:

- $S_1 = \{0, 1, 2, 3, 11, 12, 13\}$ ,  $1 + 1 + 1 + 1 + 1 + 1 = 7 \not\leq |S_1| - 1 = 7 - 1 = 6$ .
- $S_2 = \{4, 5, 6, 10\}$ ,  $1 + 1 + 1 + 1 = 4 \not\leq |S_2| - 1 = 4 - 1 = 3$ .
- $S_3 = \{7, 8, 9\}$ ,  $1 + 1 + 1 = 3 \not\leq |S_3| - 1 = 3 - 1 = 2$ .

Así pues, añadimos a la relajación lineal actual (la inicial con las siete primeras SEC añadidas) las tres SEC violadas que hemos hallado. Resolvemos la nueva relajación lineal y obtenemos la solución expuesta en la Figura 10.

El coste de la solución actual es de 253 minutos, que, como esperábamos, es un coste peor que el de las soluciones de las relajaciones lineales anteriores (220, 245). Así pues, las desigualdades que cumplen la solución óptima de nuestro problema son:

$$253 \leq f(x^{opt}) \leq 282.$$

Ahora, representamos el grafo correspondiente a la solución actual, y obtenemos el grafo de la Figura 11. El lector puede observar que obtenemos un tour, es decir una solución factible,

Solución de la relajación añadiendo SEC

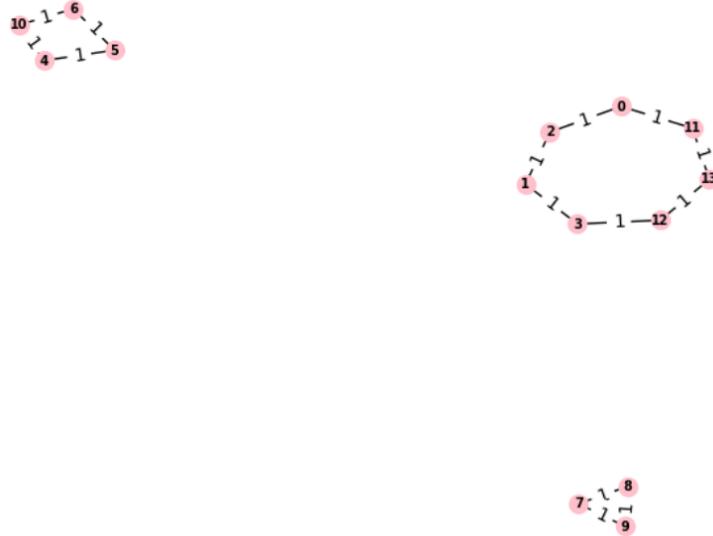


Figure 9: Grafo correspondiente a la solución de la segunda relajación lineal.

```
ampl: include stsp.run
Gurobi 10.0.1: Gurobi 10.0.1: optimal solution; objective 253
28 simplex iterations
TotalTime = 253
```

```
x [*,*]
:   1   2   3   4   5   6   7   8   9   10  11  12  13  14      :=
1  0   0   1   0 |  0   0   0   0   0   0   0   0   0   0   0
2  0   0   0   1 |  0   0   0   0   0   0   0   0   0   0   0
3  0   1   0   0 |  0   0   0   0   0   0   0   0   0   0   0
4  0   0   0   0 |  0   0   0   0   0   0   0   0   1   0   0
5  0   0   0   0 |  0   0   0   0   1   0   0   0   0   0   0
6  0   0   0   0 |  0   1   0   0   0   0   0   0   0   0   0
7  0   0   0   0 |  0   0   1   0   0   0   0   0   0   0   0
8  0   0   0   0 |  0   0   0   1   0   0   0   0   0   0   0
9  0   0   0   0 |  0   0   0   0   1   0   0   0   0   0   0
10 0   0   0   0 |  0   0   0   0   0   1   0   0   0   0   0
11 0   0   0   0 |  0   0   0   1   0   0   0   0   0   0   0
12 1   0   0   0 |  0   0   0   0   0   0   0   0   0   0   0
13 0   0   0   0 |  0   0   0   0   0   0   0   0   0   1   0
14 0   0   0   0 |  0   0   0   0   0   0   0   0   0   0   1
;
```

Figure 10: Output de *AMPL* al resolver la tercera relajación lineal.



Figure 11: Grafo correspondiente a la solución de la tercera relajación lineal.

que a demás es una solución entera! Así pues, estamos ante la solución óptima de nuestro problema. De esta forma, tenemos que

$$f(x^{opt}) = 253$$

La mejor ruta que Toni puede hacer, es decir la ruta óptima, es la que recorre los bares en el siguiente orden: Cafés Mateu, Bar Restaurant Vidiuca, Bar Bágoa, Brew Coffe, Bar Kamw, Bar Fernando, Bar la Trobada, Marisquería Loli y Valentin, Bar el Paseo, Bocamix, Bar Lozano, Bar Chicho, Pica Tapas, Bar Tropical y de vuelta a Cafés Mateu. El lector puede encontrar esta ruta marcada en el mapa en la Figura 12. Es una ruta que tiene un coste de 253 minutos. Respecto a la solución obtenida contructivamente en la primera sección via *Nearest Neighbors* hemos conseguido una mejora de  $298 - 253 = 45$  minutos. Es decir, un ahorro de 45 minutos que, de bien seguro, Toni agradecerá mucho y podrá usar con otros fines.

Finalmente es interesante hacer una valoración sobre el procedimiento que hemos llevado a cabo para conseguir la solución óptima de nuestro TSP. Creo que el procedimiento llevado a cabo durante el trabajo ha sido claro. En la sección anterior, aplicando el método heurístico *Nearest Neighbors* para la resolución del TSP hemos conseguido una solución factible para nuestro problema (con coste de 298 minutos). Es decir, hemos conseguido una ruta en la cual Toni solo pasa una vez por cada bar y regresa al final de la ruta al punto de partida.

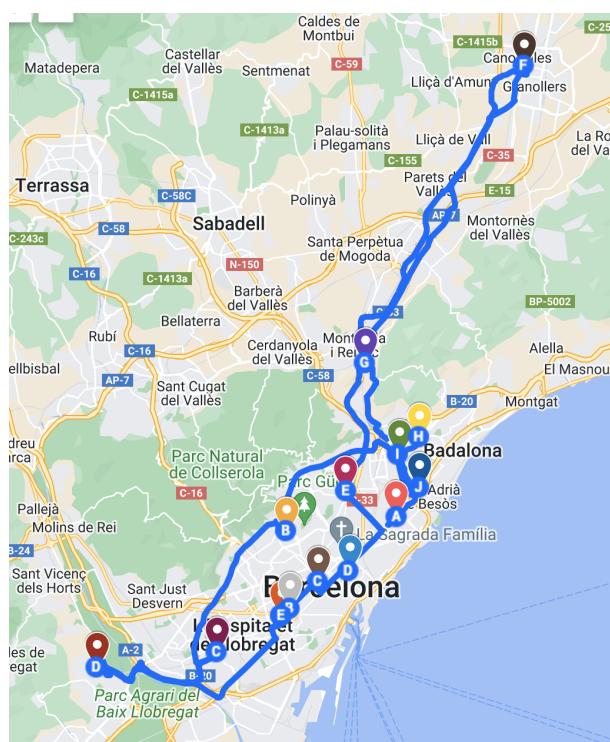


Figure 12: Mapa de la ruta de reparto óptima.

Ahora bien, está claro que esta solución no es la mejor posible, pues esta construida via una heurística. Lo siguiente que hemos hecho es mejorar esta solución via el método *Tabu Search*, obteniendo una solución con coste de 282 minutos. Esta solución, con este coste, es la que usaremos como cota superior en nuestro problema.

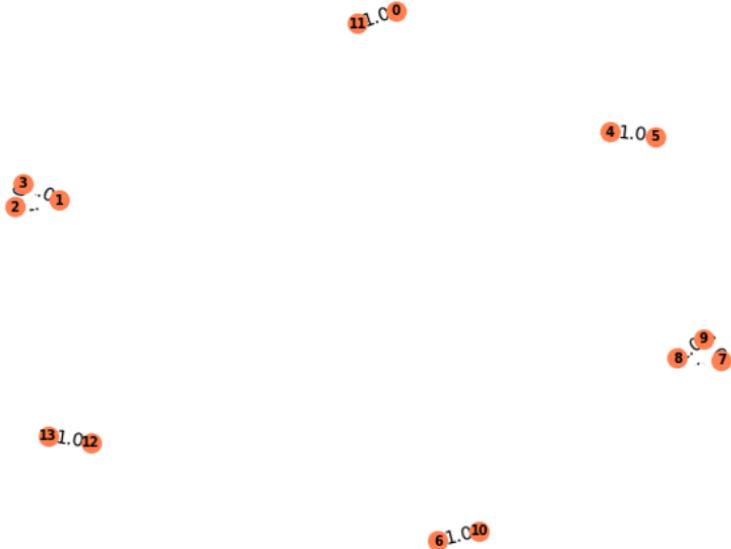
En esta segunda parte del problema, el camino que nos ha llevado a la solución óptima ha sido también muy claro. Primeramente hemos resuelto la relajación lineal de nuestro problema TSP, eliminando las SEC constraints y tomando  $0 \leq x_{ij} \leq 1$ . Con esto hemos obtenido una solución con un coste de 220 minutos. Este es el mínimo coste que puede tener nuestra solución óptima, es decir, en el mejor de los casos la solución óptima a nuestro problema tendría ese coste. No obstante, al representar esta solución nos damos cuenta de que hay subtours (se puede ver claramente en la Figura 7). Así pues, lo que hacemos es volver a resolver la relajación lineal esta vez con las SEC violadas añadidas. De esta forma nos aseguramos que los subtours que nos aparecían en la primera solución no vuelvan a aparecer. Entonces, obtenemos una solución con un coste de 245 minutos. En efecto, al haber añadido restricciones, el coste de la nueva solución es peor que la de la anterior, y esta solución pasa a ser la mejor cota inferior que tenemos para nuestro problema (en tanto que está más cerca del coste de la solución óptima). Al representar esta solución, véase Figura 9, podemos observar de nuevo, a simple vista, que nuestra solución presenta subtours. Así pues, añadimos las SEC violadas de nuevo y volvemos a resolver. Esta vez obtenemos una solución con coste 253 minutos, y que al representarla resulta ser un tour, a demás la solución es entera tal y como se puede ver en el output devuelto por *AMPL* en Figura 10. Así pues, hemos obtenido la ruta óptima.

## 4 Explorando un poco más allá

El lector puede observar, que de los diferentes pasos del proceso que debíamos seguir en la sección anterior, solo nos han hecho falta los dos primeros pasos para encontrar la solución óptima del problema. Se podría decir que hemos tenido suerte con nuestro problema. No obstante, con el objetivo de usar más herramientas de las que hemos visto durante el curso, hemos añadido esta sección, en la que no añadiremos todas las SEC violadas que podemos ver a simple vista, de esta forma obtendremos otras soluciones de las respectivas relajaciones lineales que nos darán más juego para usar los conceptos adquiridos en el curso. En esta sección todas las relajaciones lineales serán resueltas con *AMPL-Gurobi* como en el apartado anterior. A diferencia del apartado anterior, no añadiremos los outputs de *AMPL*, lo que añadiremos será directamente el grafo correspondiente a la solución. Consideramos que añadir todos los outputs implicaría un gran número de figuras que haría crecer mucho el número de páginas, y pensamos que esto no compensa teniendo en cuenta lo que aportan estos outputs. Además, el lector ha podido ver como son estos outputs en la sección anterior.

El primer paso, tal y como en la sección anterior, es resolver la relajación lineal inicial. La

Solución de la relajación añadiendo más SEC

Figure 13: Grafo correspondiente a la solución de la relajación lineal añadiendo la SEC violada por  $W_1$ .

solución es la que se puede ver en la Figura 6, el grafo correspondiente a esta solución se encuentra en la Figura 7, y el coste de esta solución es de 220. Esta es la peor cota inferior posible, tal y como comentamos en la sección anterior. Esta vez, en vez de añadir todas las SEC que se ven claramente violadas por los siete pares de nodos (ver Figura 7), lo que hacemos es añadir únicamente la siguiente SEC:

- $W_1 = \{1, 2\}$ ,  $1 + 1 = 2 \not\leq |W_1| - 1 = 2 - 1 = 1$ .

Añadimos la anterior desigualdad y resolvemos la relajación lineal restante. La solución obtenida tiene un coste de 227 minutos. Así pues, la nueva cota inferior es 227, y tenemos (usando la misma notación que en la sección anterior)

$$227 \leq f(x^{opt}) \leq 282.$$

Cabe destacar que la solución obtenida es una solución entera. El grafo correspondiente a solución que se puede ver en la Figura 13. Podemos observar, de nuevo a simple vista, que hay diferentes subtours que no satisfacen las SEC. En este caso añadimos únicamente (a pesar de que se pueden ver otros subconjuntos que violan SEC) la siguiente SEC:

- $W_2 = \{4, 5\}$ ,  $1 + 1 = 2 \not\leq |W_2| - 1 = 2 - 1 = 1$ .



Figure 14: Grafo correspondiente a la solución de la relajación lineal añadiendo la SEC violada por  $W_2$ .

Añadimos la anterior desigualdad a nuestro problema y resolvemos la relajación lineal restante. De nuevo, obtenemos una solución entera. Esta vez obtenemos una solución con un coste de 231 minutos. Luego tenemos que

$$231 \leq f(x^{opt}) \leq 282.$$

El grafo construido a partir de esta solución se puede ver en la Figura 14. De nuevo se pueden apreciar a simple vista los subconjuntos de nodos que violan alguna SEC. Añadimos únicamente la siguiente SEC:

- $W_3 = \{12, 13\}$ ,  $1 + 1 = 2 \not\leq |W_3| - 1 = 2 - 1 = 1$ .

Añadimos la anterior desigualdad a nuestro problema y resolvemos la relajación lineal. Obtenemos ahora una solución con un coste de 243 minutos, por lo tanto podemos actualizar nuestra mejor cota inferior para nuestro problema:

$$243 \leq f(x^{opt}) \leq 282.$$

El lector puede observar el grafo correspondiente a la solución en la Figura 15. Se puede observar que la solución obtenida en este caso no es entera, es la primera solución no entera que hemos obtenido en el proceso. Se puede observar a simple vista que hay un subtour.

## Solución de la relajación añadiendo más SEC

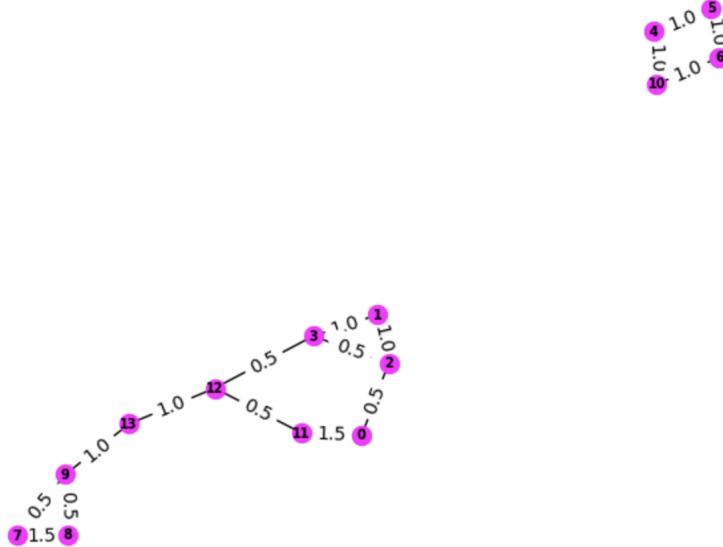


Figure 15: Grafo correspondiente a la solución de la relajación lineal añadiendo la SEC violada por  $W_3$ .

Pero esta vez, es solo un subtour el que se puede apreciar a simple vista. No obstante, usando el algoritmo heurístico de separación visto durante el curso podemos observar los dos pares de nodos  $\{0, 11\}$ ,  $\{7, 8\}$  violan una SEC cada uno. Esto lo vemos porque las dos aristas que unen estos dos pares de nodo tienen un peso de 1.5, es decir, mayor que 1. Por lo tanto, según el método heurístico de separación los subconjuntos  $\{0, 11\}$ ,  $\{7, 8\}$  violan sus respectivas SEC. Esto tiene todo el sentido, pues los dos pares de nodos  $\{0, 11\}$ ,  $\{7, 8\}$ , son de los que al inicio violaban una SEC, y que no hemos añadido al problema (para no solucionar el problema demasiado rápido) por lo tanto tienen que seguir incumpliendo la restricción. Ahora, añadimos la SEC violada por el subconjunto de nodos formando el subtour que se puede apreciar a simple vista. La SEC en cuestión es:

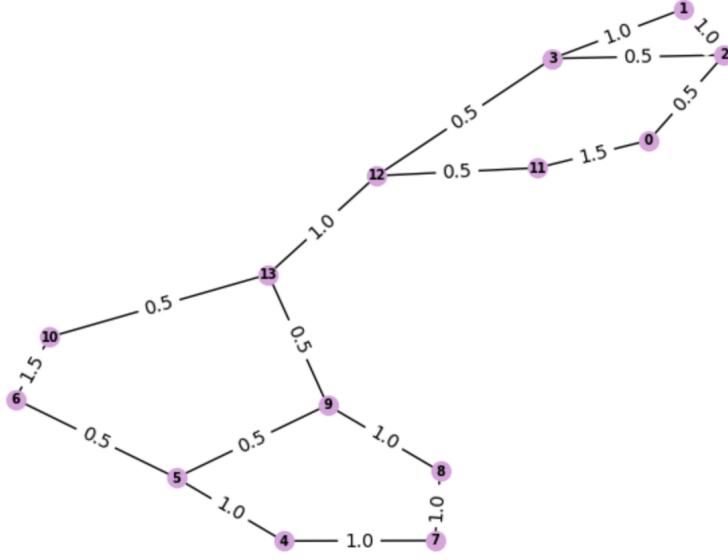
- $W_4 = \{10, 4, 5, 6\}$ ,  $1 + 1 + 1 + 1 = 4 \not\leq |W_4| - 1 = 4 - 1 = 3$ .

Añadimos la anterior desigualdad a nuestro problema, resolvemos la relajación lineal resultante. Obtenemos una solución con un coste de 247.5, por lo tanto actualizamos la cota inferior de nuestro problema:

$$247.5 < f(x^{opt}) < 282.$$

El grafo correspondiente a la solución obtenida se encuentra en la Figura 16. En este caso ya no podemos observar ningún subtour a simple vista. No obstante, usando el algoritmo heurístico de separación, y por los argumentos anteriormente usados, sabemos que los pares

Solución de la relajación añadiendo más SEC

Figure 16: Grafo correspondiente a la solución de la relajación lineal añadiendo la SEC violada por  $W_4$ .

de nodos  $\{0, 11\}$ ,  $\{6, 10\}$  no satisfacen sus respectivas SEC (las aristas que unen esos pares de nodo tienen un coste de  $1.5 > 1$ ). La solución obtenida, es de nuevo no entera. Lo que vamos a hacer en este caso es usar otro tipo de desigualdad válida, que no va a ser una SEC. Vamos a añadir una *2-match inequality*. Recordemos que la *2-match inequality* es de la forma

$$x(E(H)) + \sum_{i=1}^{2k+1} x(E(T_i)) \leq |H| + \lfloor \frac{2k+1}{2} \rfloor, \quad (1)$$

donde  $2k+1$  es el número de dientes que tenemos, los dientes se denotan con  $T_i$ , y  $H$  denota el mango.

Manualmente, hemos podido observar que la siguiente *2-match inequality* se ve violada por nuestro problema:

- $H = \{4, 5, 6\}$ ,  $T_1 = \{4, 7\}$ ,  $T_2 = \{5, 9\}$ ,  $T_3 = \{6, 10\}$ . Y, usando (1), es fácil ver que  $1 + 0.5 + 1 + 0.5 + 1.5 = 4.5 \not\leq 4 = 3 + 1$ .

Así pues, lo que hacemos es añadir esta desigualdad válida que no se satisface a nuestro problema, y resolver la relajación lineal obtenida. Al hacer esto obtenemos una solución con un coste de 248, por lo tanto obtenemos una mejor cota inferior, indicativo de que la *2-match*

Solución de la relajación añadiendo 2-match inequality

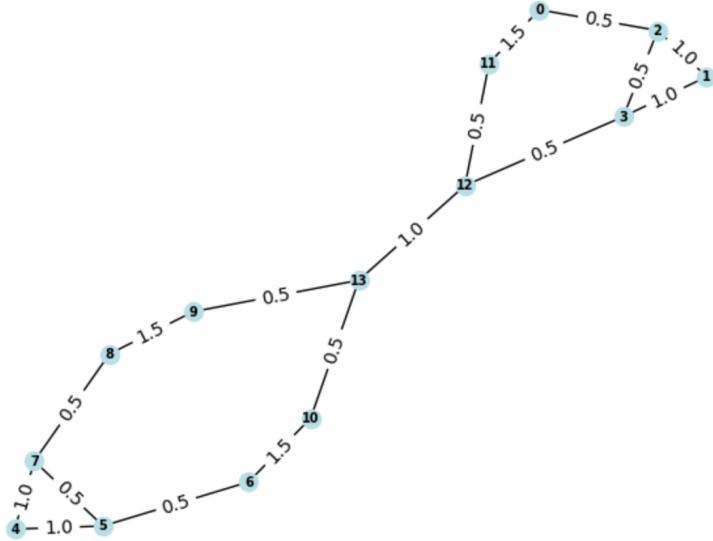


Figure 17: Grafo correspondiente a la solución de la relajación lineal añadiendo la *2-match inequality* violada.

*inequality* ha sido añadida de forma correcta a nuestro problema. Tenemos entonces ahora

$$248 \leq f(x^{opt}) \leq 282.$$

El grafo que corresponde a la solución obtenida tras añadir la *2-match inequality* se encuentra en la Figura 17. La solución obtenida es no entera. De nuevo el lector puede observar que hay aristas con coste mayor que uno, lo cual implica que los pares de nodos conectados por estas aristas incumplen sus respectivas SEC.

No obstante, en vez de añadir las SEC que vemos que no se satisfacen gracias al algoritmo heurístico de separación, lo que vamos a hacer es añadir otra desigualdad válida violada por nuestra solución. En este caso añadiremos una desigualdad peine (*comb inequality* en inglés). Recordemos que las desigualdades peine son de la forma:

$$x(E(H)) + \sum_{i=1}^{2k+1} x(E(T_i)) \leq |H| + \sum_{i=1}^{2k+1} (|T_i| - 1) - (k + 1). \quad (2)$$

Manualmente, hemos podido ver que la siguiente desigualdad peine no se satisface por nuestra solución:

Solución de la relajación añadiendo comb inequality

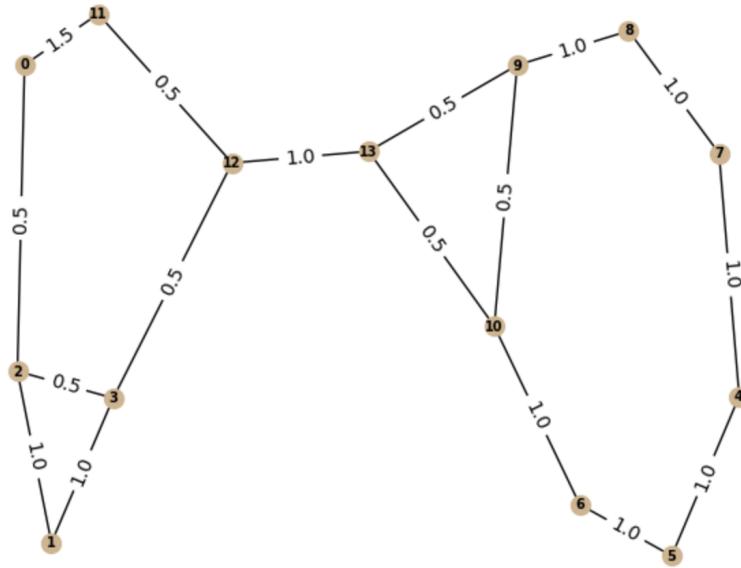


Figure 18: Grafo correspondiente a la solución de la relajación lineal añadiendo la desigualdad peine violada.

- $H = \{5, 6, 7\}$ ,  $T_1 = \{6, 10\}$ ,  $T_2 = \{5, 4\}$ ,  $T_3 = \{7, 8, 9\}$ . Y, usando (2), es fácil ver que  $0.5 + 0.5 + 1.5 + 1 + 0.5 + 1.5 = 5.5 \not\leq 5 = 3 + 1 + 1 + 2 - 2$ .

Así pues, añadimos la anterior desigualdad peine y resolvemos la relajación lineal restante. Obtenemos una solución que tiene un coste de 248. Así pues, en este caso no mejoramos la cota inferior de nuestro problema.

El grafo correspondiente a la solución de la relajación lineal se encuentra en la Figura 18. Vemos que la solución es de nuevo no entera. No obstante parece ser que obtenemos más valores enteros que en la solución anterior.

Lo que vamos a hacer ahora, es usar el algoritmo de separación heurístico visto en el curso para añadir todas las SEC violadas por nuestra solución de la relajación lineal que podamos. Este es un algoritmo muy útil y que no hemos tenido la oportunidad de usar aún, pues nuestro problema nos ha devuelto soluciones enteras con subtours donde la aplicación de este algoritmo carece de sentido. Podemos ver rápidamente que los subconjuntos de nodos  $W_5 = \{1, 2, 3\}$ ,  $W_6 = \{0, 11\}$ ,  $W_7 = \{10, 6, 5, 4, 7, 8, 9\}$  violan sus respectivas SEC:

- $W_5 = \{1, 2, 3\}$ ,  $1 + 1 + 0.5 = 2.5 \not\leq |W_5| - 1 = 3 - 1 = 2$ .
- $W_6 = \{0, 11\}$ ,  $1.5 \not\leq |W_6| - 1 = 2 - 1 = 1$ .



Figure 19: Grafo correspondiente a la solución de la relajación lineal añadiendo las SEC violadas.

- $W_7 = \{10, 6, 5, 4, 7, 8, 9\}$ ,  $1 + 1 + 1 + 1 + 1 + 1 + 0.5 = 6.5 \not\leq |W_7| - 1 = 7 - 1 = 6$ .

Añadimos las correspondientes SEC a nuestro problema y resolvemos la relajación lineal usando AMPL. Obtenemos esta vez una solución con un coste de 249 minutos, así pues esta vez obtenemos una mejor cota inferior que la anterior. Tenemos ahora

$$249 \leq f(x^{opt}) \leq 282.$$

El grafo correspondiente a la solución obtenida se encuentra en la Figura 19. Vemos que hemos vuelto a obtener una solución entera en la cual podemos observar claramente los subtours a simple vista. Así pues, no nos queda otra que añadir los subconjuntos de nodos formando los subtours que podemos observar.

Es decir, añadimos las siguientes SEC a nuestro problema:

- $W_8 = \{6, 10\}$ ,  $1 + 1 = 2 \not\leq |W_8| - 1 = 2 - 1 = 1$ .
- $W_9 = \{7, 8\}$ ,  $1 + 1 = 2 \not\leq |W_9| - 1 = 2 - 1 = 1$ .

Y resolvemos la relajación lineal resultante. Obtenemos un tour! Así pues hemos obtenido la solución óptima! El coste de esta solución es de 253, en efecto, como la solución óptima que obtenimos en el apartado anterior.

## Solución de la relajación añadiendo más SEC

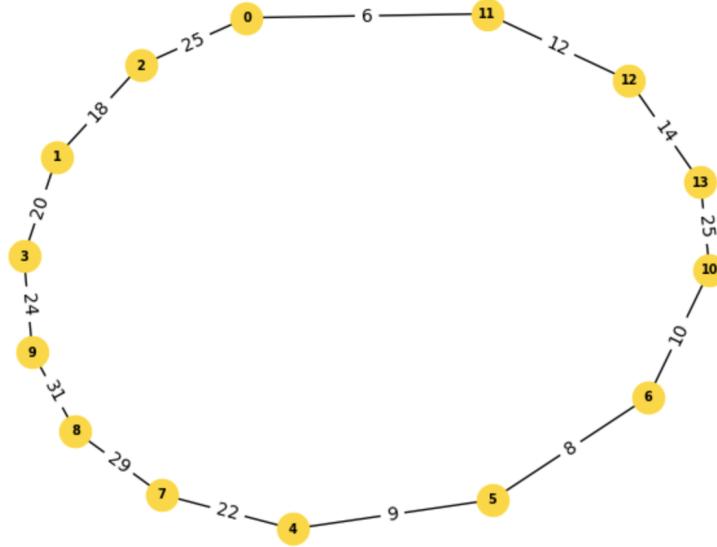


Figure 20: Grafo correspondiente a la solución óptima obtenida.

El grafo correspondiente a la solución obtenida se puede ver en la Figura 20, en este grafo hemos añadido los correspondientes costes temporales de ir de un bar a otro). La lectora puede observar que este tour es distinto al obtenido en la sección anterior expuesto en la Figura 11. No obstante, el coste es el mismo, ambos son soluciones factibles, y ambas soluciones son óptimas, por lo tanto hemos encontrado una solución óptima alternativa.

Así pues, Toni tiene dos mejores rutas posibles para llevar a cabo su reparto de café. Una de ellas es la que hemos expuesto en el apartado anterior, la que hemos encontrado ahora es una ruta en la que debe visitar los bares en el siguiente orden: Cafés Mateu, Bar Restaurant Viduca, Bar Bágoa, Brew Coffee, Bar Lozano, Bocamix, Bar el Paseo, Marisquería Loli y Valentín, Bar la Trobada, Bar Fernando, Bar Kamw, Bar Chicho, Pica tapas, Bar Tropical y de vuelta a Cafés Mateu a estacionar la furgoneta de reparto.

En esta última sección, hemos dado más vueltas sobre nuestro problema con el objetivo de usar otros conceptos vistos durante el curso. Durante todo el proceso la lógica ha sido clara, resolver la relajación lineal y entonces añadir algún corte (desigualdad válida violada por nuestra solución de la relajación). Al hacer esto, a cada paso, hemos ido obteniendo mejores cotas inferiores, que cada vez se acercaban más al coste de la solución óptima (que sabíamos de antemano que era 253, pues en el anterior apartado habíamos resuelto el problema). Esta exploración más profunda del problema me ha resultado muy interesante ya que he podido observar la evolución de las soluciones de las relajaciones lineales a medida que se

iban añadiendo restricciones. También, he podido ver que otras desigualdades válidas tales como las desigualdades peine o como las *2-match inequalities* son muy útiles para conseguir mejores cotas inferiores y acercarnos más a la solución óptima de nuestro problema.

## 5 Apéndices

Finalmente, comentar que la lectora podrá encontrar a continuación un Anexo con el código de python usado para todo el trabajo, y después de esto un Anexo con los archivos .mod, .dat, .run de AMPL usados durante el trabajo.

Sobre el notebook de python añadido: he pensado que para ser totalmente transparente era mejor añadir todo el notebook, con el código que he usado para generar los grafos que se pueden ver en la memoria incluido. No obstante el código importante (implementación de Nearest Neighbours e implementación de Tabu Search) se encuentran al inicio del notebook.

Sobre los documentos de AMPL añadidos. He decidido añadir los diferentes documentos usados en AMPL para ser totalmente transparente sobre esta parte del trabajo también. Como comentarios: en la sección 3 del trabajo he usado los primeros documentos adjuntados. Se puede ver en ellos que he trabajado con la matriz completa. No obstante, en la sección 4 del trabajo, tras descubrir como trabajar de forma más óptima con la matriz triangular superior decidí hacerlo de esta manera.

# TSP

January 19, 2024

## 1 Resolviendo el *Traveling Salesman Problem* (ANEXO)

Autor: Arnau Garcia Fernandez

En este notebook se programarán las diferentes funciones y el código necesario para resolver el Traveling Salesman Problem en el Final Project de la asignatura de Modelos y Métodos de la Investigación Operativa.

Primeramente definimos la matriz de costes. Hay que tener en cuenta que estamos trabajando con un TSP simétrico.

```
[1]: import numpy as np  
bares_time = np.array([  
    [0,19,25,21,36,33,37,39,56,35,35,6,17,25],  
    [19,0,18,20,32,28,32,36,45,32,32,29,35,39],  
    [25,18,0,28,38,33,37,40,50,39,37,36,41,42],  
    [21,20,28,0,22,18,23,27,37,24,21,28,18,27],  
    [36,32,38,22,0,9,16,22,36,26,15,48,38,34],  
    [33,28,33,18,9,0,8,16,28,17,10,44,33,30],  
    [37,32,37,23,16,8,0,21,32,22,10,44,33,30],  
    [39,36,40,27,22,16,21,0,29,23,24,53,47,42],  
    [56,45,50,37,36,28,32,29,0,31,29,63,53,48],  
    [35,32,39,24,26,17,22,23,31,0,20,42,32,28],  
    [35,32,37,21,15,10,10,24,29,20,0,38,27,25],  
    [6,29,36,28,48,44,44,53,63,42,38,0,12,19],  
    [17,35,41,18,38,33,33,47,53,32,27,12,0,14],  
    [25,39,42,27,34,30,30,42,48,28,25,19,14,0]  
])
```

Definimos una función que dada la matriz de costes y un tour nos calcule el coste de ese tour. Es decir, esta función será usada como función objetivo y será útil durante todo el trabajo.

```
[2]: def cost(mat,tour):  
    suma=0  
    for i in range(len(tour)-1):  
        suma += mat[tour[i],tour[i+1]]  
    return(suma)
```

Ahora vamos a definir una función para implementar el método heurístico de *nearest neighbors*.

```
[7]: def nearest_neighbor(bares_time):
    n = len(bares_time)
    #Inicialmente no hemos visitado ningún bar
    bares_visit = [False] * n
    #Inicializamos tour
    tour = []

    # Elegimos empezar des de Cafés Mateu (donde se produce la salida y la llegada del reparto)
    bar_actual = 0
    tour.append(bar_actual)
    bares_visit[bar_actual] = True

    while(len(tour) < n):
        # Buscamos el bar no visitado más cercano (con menor coste)
        #Fijamos en la matriz la fila del bar actual, recorremos todas las columnas
        #excepto las que sean de bares ya visitados, y nos quedamos con el mínimo.
        nearest_neighbour = min((i for i in range(n) if not bares_visit[i]),
                                 key=lambda bar: bares_time[bar][bar])

        # Nos movemos al bar más cercano
        bar_actual = nearest_neighbour
        tour.append(bar_actual)
        bares_visit[bar_actual] = True

    # Acabámos el tour en Cafés mateu
    tour.append(tour[0])

    return(tour)

result_tour = nearest_neighbor(bares_time)

print("Orden del Tour resultante con Nearest Neighbors:", result_tour)
print("Coste del Tour resultante con Nearest Neighbors:", cost(bares_time,result_tour))
```

Orden del Tour resultante con Nearest Neighbors: [0, 11, 12, 13, 10, 5, 6, 4, 3, 1, 2, 9, 7, 8, 0]  
Coste del Tour resultante con Nearest Neighbors: 298

Nuestro objetivo ahora es implementar el método de Tabu Search para obtener una mejor solución factible para nuestro STSP. Para entender bien como funciona el método Tabu Search hemos usado las slides de clase y <https://riunet.upv.es/handle/10251/167142>. Vamos a implementar un tabu search con intercambio del orden de visita de nodos consecutivos en la solución proporcionada por la heurística. Primero, vamos a definir una función que dado un tour y un índice, nos intercambie

el orden en que se visitan el nodo del índice dado y el siguiente.

```
[8]: def generador_sol_vecinas(tour,k):
    sol_vecina = tour.copy()
    if(k==len(tour)-2):
        sol_vecina[k]=tour[0]
        sol_vecina[0]=tour[k]
        sol_vecina[len(tour)-1]=sol_vecina[0]
    elif(k==0):
        sol_vecina[k]=tour[k+1]
        sol_vecina[k+1]=tour[k]
        sol_vecina[len(tour)-1]=sol_vecina[k]
    else:
        sol_vecina[k]=tour[k+1]
        sol_vecina[k+1]=tour[k]
    return(sol_vecina)
```

Definimos nuestra función para implementar Tabu Search. Como argumentos de la función tenemos la solución que nos proporciona la heurística, la matriz de costes, las iteraciones máximas que queremos usar en nuestra implementación de Tabu Search y la tabu “tenure”. En esta función lo que haremos será calcular el espacio de soluciones vecinas, crear una lista con las soluciones vecinas que no están en la “tabu list”, quedarnos con la mejor solución (la de mínimo coste de la función objetivo), comprobar si está solución es mejor que la que teníamos y si es así, guardarla. La función devuelve la mejor solución encontrada en todo el proceso.

```
[9]: def tabu_search(tour_inicial, mat, iter_max, tabu_tenure):
    sol_actual = tour_inicial.copy()
    best_tour = tour_inicial.copy()
    tabu_list = []

    for i in range(iter_max):
        sol_vecina=[]
        sol_vecina_not_tabu=[]
        #Calculamos espacio de sol vecinas, eliminando las que estén en la tabú
        ↵list
        for j in range(len(sol_actual)-1):
            sol_vecina.append(generador_sol_vecinas(sol_actual,j))
            if(sol_vecina[j] not in tabu_list):
                sol_vecina_not_tabu.append(generador_sol_vecinas(sol_actual,j))
        #Buscamos en el espacio de sol vecinas la de minimo coste
        sol = sol_vecina_not_tabu[0].copy()
        for j in range(len(sol_vecina_not_tabu)):
            if(cost(mat, sol_vecina_not_tabu[j])<cost(mat,sol)):
                sol = sol_vecina_not_tabu[j].copy()

        # Como ya sabemos que no va estar en la tabu list, solo comprobamos el
        ↵move value
        if(cost(mat,sol) < cost(mat, sol_actual)):
```

```

        sol_actual = sol.copy()
        tabu_list.append(sol)
        if(len(tabu_list) > tabu_tenure): #eliminamos elementos de la tabu
            ↪tenure cuando sea necesario
            tabu_list.pop(0)

        # Actualizar mejor sol hallada hasta la fecha
        if(cost(mat, sol_actual) < cost(mat, best_tour)):
            best_tour = sol_actual.copy()

    return(best_tour)

```

Usamos como tabu tenure 7 (usando lo que vimos en las slides del curso), y pondremos un máximo de 100 iteraciones.

*Observación:* probamos con diferentes iteraciones máximas, para 1000, 10000 el resultado era el mismo que para 100. Así pues, decidimos dejar como máximas iteraciones 100.

[10]: tour\_mejorado = tabu\_search(result\_tour, bares\_time, 100, 7)

[11]: print("Tour mejorado:", tour\_mejorado)  
print("Coste del tour mejorado:", cost(bares\_time, tour\_mejorado))

Tour mejorado: [0, 11, 12, 13, 10, 6, 5, 4, 3, 1, 2, 9, 8, 7, 0]  
Coste del tour mejorado: 282

### 1.0.1 Visualización de grafos

Usaremos una librería de python muy útil, `networkx`, para imprimir grafos. La usaremos para imprimir los grafos correspondientes a las soluciones que vayamos encontrando durante el trabajo.

Primeramente definimos una función que dada una matriz de costes simétrica nos contruya un grafo completo usando los costes en dicha matriz.

[6]: import networkx as nx  
import matplotlib.pyplot as plt  
def grafo(cost\_matrix):  
 num\_nodes = len(cost\_matrix)  
  
 # Create a complete graph (all nodes connected to all nodes)
 G = nx.complete\_graph(num\_nodes)  
  
 # Add weights to the edges based on the distance matrix
 for i in range(num\_nodes):
 for j in range(i+1, num\_nodes):
 weight = cost\_matrix[i, j]
 G[i][j]['weight'] = weight
 G[j][i]['weight'] = weight

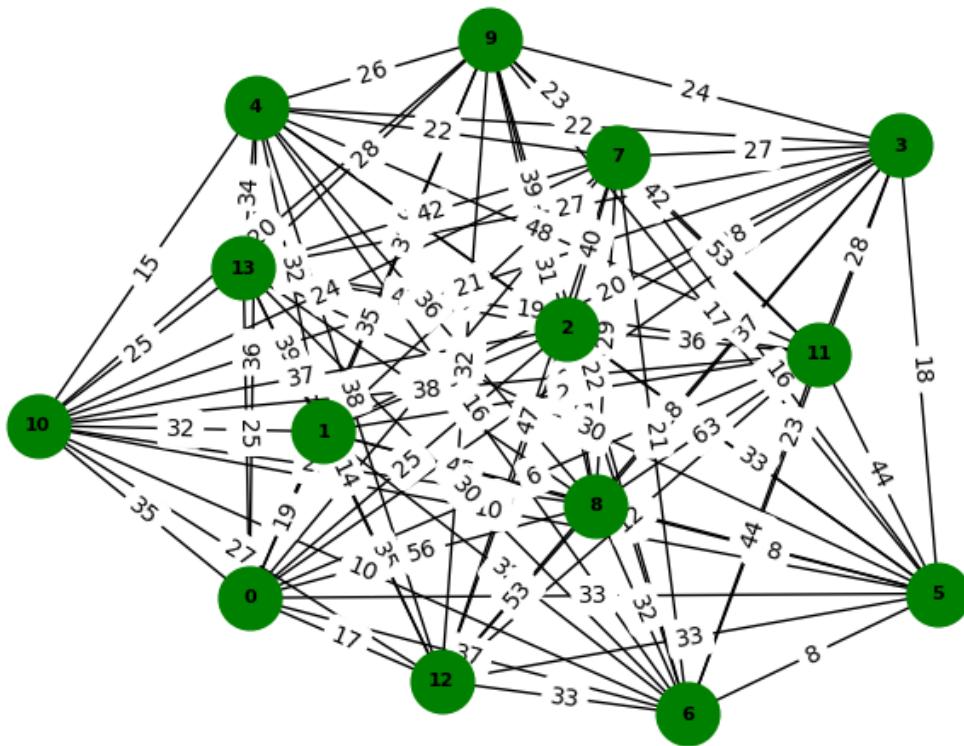
```
return G
```

A continuación, construimos un grafo completo con todos los tiempos entre bares usando la función `grafo` y imorimimos el grafo usando `matplotlib`.

```
[9]: # Construimos el grafo
weighted_graph = grafo(bares_time)

# imprimimos
pos = nx.spring_layout(weighted_graph)
nx.draw(weighted_graph, pos, with_labels=True, node_size=700, ▾
    node_color="green",
    font_size=8, font_color="black", font_weight="bold", arrowsize=10)
labels = nx.get_edge_attributes(weighted_graph, 'weight')
nx.draw_networkx_edge_labels(weighted_graph, pos, edge_labels=labels)
plt.title("Bares en los que entregar café y tiempos para ir de uno al otro")
plt.show()
```

Bares en los que entregar café y tiempos para ir de uno al otro



Ahora, definimos una función que dada una matriz de costes y una tour que sea solución del problema nos devuelva la matriz de costes asociada a la solución. Lo que haremos será inicializar

a 0 la matriz y llenar las componentes de la matriz correspondientes a los nodos conectados por la solución.

```
[14]: def new_matrix(mat, sol):
    new_mat = np.zeros((len(sol)-1, len(sol)-1), dtype=int)
    for i in range(len(sol)-1):
        new_mat[sol[i],sol[i+1]] = int(mat[sol[i],sol[i+1]])
        new_mat[sol[i+1],sol[i]] = int(mat[sol[i+1],sol[i]])
    return(new_mat)
```

Usando esta función `new_matrix`, imprimimos los grafos correspondientes a las soluciones que obtenemos. Lo que haremos será programar el código de forma que aquellas aristas con peso igual a 0 no se impriman, así podremos ver un grafo más claro con solo las aristas que conectan los nodos en el tour.

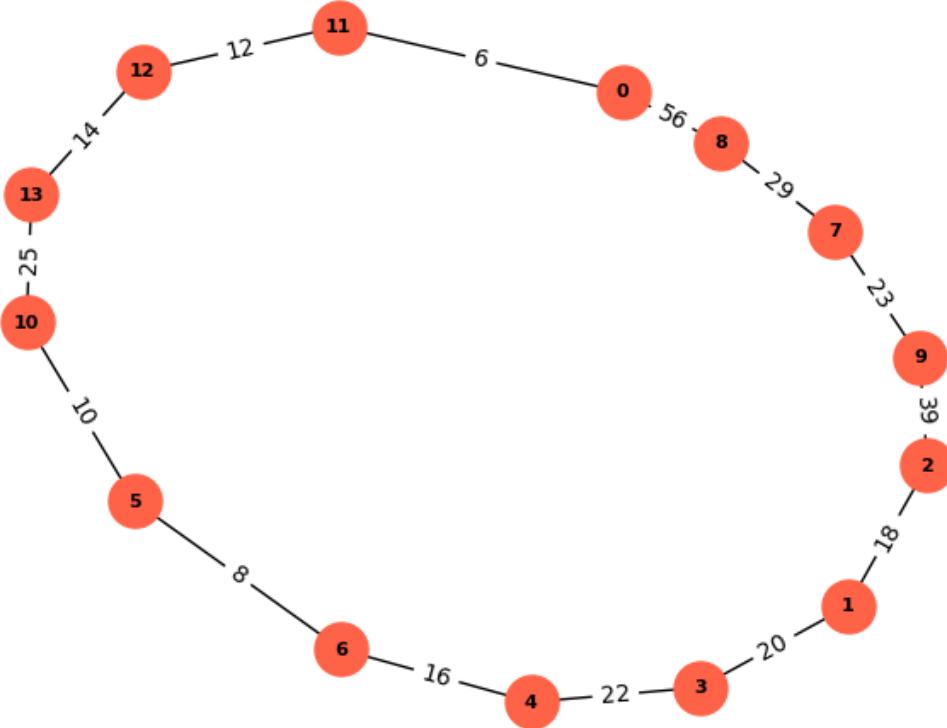
```
[20]: #Matriz de tiempos para la sol de NN
bares_sol1 = new_matrix(bares_time, result_tour)

weighted_graph = grafo(bares_sol1)

# Subgrafo con peso >0
edges_to_keep = [(u, v) for u, v, d in weighted_graph.edges(data=True) if
                  d['weight'] > 0]
subgraph = weighted_graph.edge_subgraph(edges_to_keep)

# Imprimimos subgrafo
pos = nx.spring_layout(subgraph)
nx.draw(subgraph, pos, with_labels=True, node_size=500, node_color="tomato",
        font_size=8, font_color="black", font_weight="bold", arrowsize=10)
labels = nx.get_edge_attributes(subgraph, 'weight')
nx.draw_networkx_edge_labels(subgraph, pos, edge_labels=labels)
plt.title("Solución proporcionada por Nearest Neighbors")
plt.show()
```

### Solución proporcionada por Nearest Neighbors



Volvemos a hacer lo mismo, esta vez para imprimir el grafo correspondiente a la solución mejorada con Tabu Search.

```
[21]: #Matriz de distancias para sol de tabu search
bares_sol2=new_matrix(bares_time, tour_mejorado)

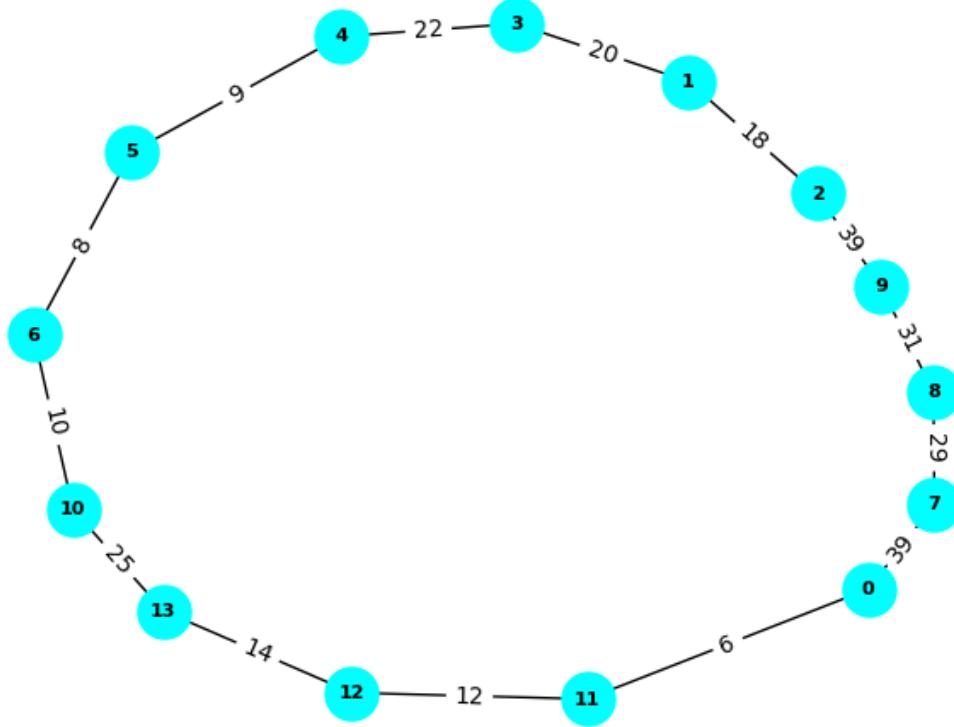
weighted_graph = grafo(bares_sol2)

# Subgrafo con peso >0
edges_to_keep = [(u, v) for u, v, d in weighted_graph.edges(data=True) if
                  d['weight'] > 0]
subgraph = weighted_graph.edge_subgraph(edges_to_keep)

# Imprimimos subgrafo
pos = nx.spring_layout(subgraph)
nx.draw(subgraph, pos, with_labels=True, node_size=500, node_color="cyan",
        font_size=8, font_color="black", font_weight="bold", arrowsize=10)
labels = nx.get_edge_attributes(subgraph, 'weight')
nx.draw_networkx_edge_labels(subgraph, pos, edge_labels=labels)
plt.title("Solución mejorada via Tabu Search ")
```

```
plt.show()
```

Solución mejorada via Tabu Search



Lo que haremos ahora es representar los grafos correspondientes a las soluciones de las relajaciones lineales. Para ello, definiremos las matrices que nos devuelve AMPL al resolver la relajación manualmente. Una vez tenemos la matriz de costes, definimos el grafo y lo imprimimos. Con estos grafos podremos usar algoritmos de separación más fácilmente, y podremos observar de forma visual el comportamiento de nuestras soluciones.

```
[22]: lr = np.zeros((14,14), dtype=int)
lr[11,0], lr[0,11]=1,1
lr[2,1], lr[1,2]=1,1
lr[1,2], lr[2,1]=1,1
lr[9,3], lr[3,9]=1,1
lr[5,4], lr[4,5]=1,1
lr[10,6], lr[6,10]=1,1
lr[8,7], lr[7,8]=1,1
lr[13,12], lr[12,13]=1,1
```

```
[23]: weighted_graph = grafo(lr)

# Subgrafo con peso >0
edges_to_keep = [(u, v) for u, v, d in weighted_graph.edges(data=True) if
    d['weight'] > 0]
subgraph = weighted_graph.edge_subgraph(edges_to_keep)

# Imprimimos subgrafo
pos = nx.spring_layout(subgraph)
nx.draw(subgraph, pos, with_labels=True, node_size=100, node_color="bisque",
        font_size=7, font_color="black", font_weight="bold", arrowsize=10)
labels = nx.get_edge_attributes(subgraph, 'weight')
nx.draw_networkx_edge_labels(subgraph, pos, edge_labels=labels)
plt.title("Solución de la relajación")
plt.show()
```

Solución de la relajación



AMPL nos devuelve como solución el grafo que acabamos de ver en el chunk anterior. El coste de esta solución es de 220. Recordemos que el coste con NN era de 298 y el coste del tour mejorado con Tabu Search era de 282. Como esperábamos, el coste de la relajación lineal es mejor que el de los métodos heurísticos. El coste de la relajación lineal nos servirá como cota inferior de la

solución óptima del problema. Los costes proporcionados por las heurísticas nos servirán como cotas superiores.

Si nos fijamos, podemos ver que vía Relajación Lineal y quitando las SEC, son todo subtours de pares de nodos. De esta forma, los pares de nodos que se encuentran emparejados son subconjuntos de nodos que violan una SEC. Así pues, los subconjuntos de nodos que violan SEC son: - {0,11} - {1,2} - {8,7} - {4,5} - {3,9} - {6,10} - {12,13}

Después de resolver la nueva relajación lineal con las SEC violadas añadidas obtenemos una nueva solución con AMPL. Lo que hacemos a continuación es dibujar el grafo correspondiente a la nueva solución, de la misma forma que hemos hecho anteriormente.

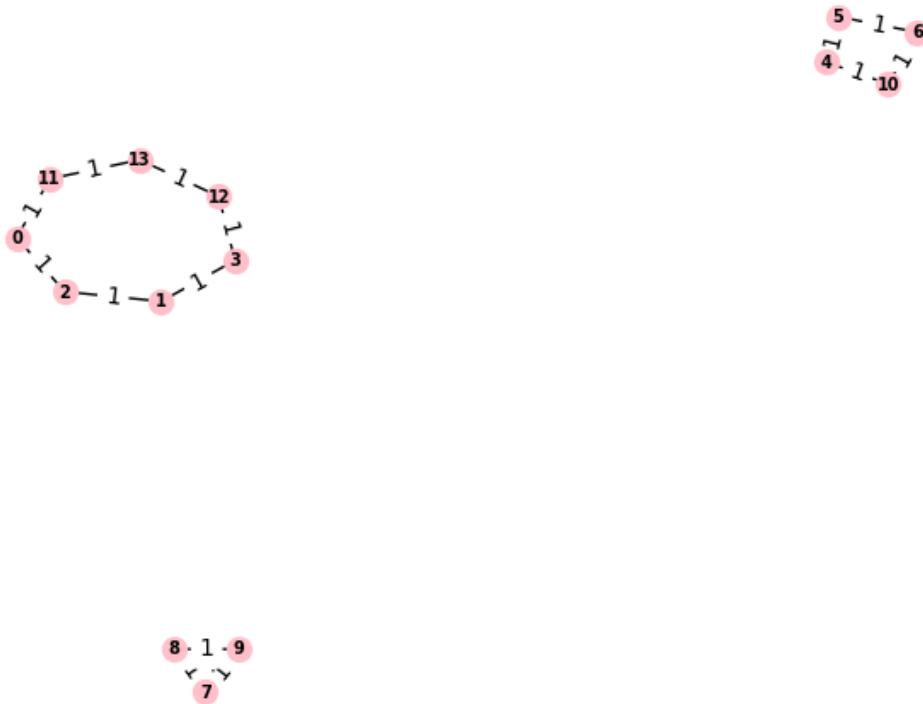
```
[24]: lr_sec1 = np.zeros((14,14), dtype=int)
lr_sec1[11,0],lr_sec1[0,11]=1,1
lr_sec1[2,1],lr_sec1[1,2]=1,1
lr_sec1[0,2],lr_sec1[2,0]=1,1
lr_sec1[1,3],lr_sec1[3,1]=1,1
lr_sec1[5,4],lr_sec1[4,5]=1,1
lr_sec1[6,5], lr_sec1[5,6]=1,1
lr_sec1[10,6], lr_sec1[6,10]=1,1
lr_sec1[9,7],lr_sec1[7,9]=1,1
lr_sec1[7,8], lr_sec1[8,7]=1,1
lr_sec1[8,9], lr_sec1[9,8]=1,1
lr_sec1[4,10], lr_sec1[10,4]=1,1
lr_sec1[13,11], lr_sec1[11,13]=1,1
lr_sec1[3,12],lr_sec1[12,3]=1,1
lr_sec1[12,13], lr_sec1[13,12]=1,1
```

```
[16]: weighted_graph = grafo(lr_sec1)

# Subgrafo con peso >0
edges_to_keep = [(u, v) for u, v, d in weighted_graph.edges(data=True) if
    ↪d['weight'] > 0]
subgraph = weighted_graph.edge_subgraph(edges_to_keep)

# Imprimimos subgrafo
pos = nx.spring_layout(subgraph)
nx.draw(subgraph, pos, with_labels=True, node_size=100, node_color="pink",
        font_size=7, font_color="black", font_weight="bold", arrowsize=10)
labels = nx.get_edge_attributes(subgraph, 'weight')
nx.draw_networkx_edge_labels(subgraph, pos, edge_labels=labels)
plt.title("Solución de la relajación añadiendo SEC")
plt.show()
```

## Solución de la relajación añadiendo SEC



El coste de esta solución es de 245.

Podemos ver fácilmente que los siguientes subconjuntos de nodos violan SEC (tomamos nodos 1,2,..,14): - {11,0,2,1,3,12,13} - {9,7,8} - {10,4,5,6}

Añadimos estos subconjuntos de nodos que sabemos que violan SECs a nuestra relajación lineal y obtenemos una nueva solución. De nuevo, representamos el grafo asociado a esta nueva solución.

```
[25]: lr_sec2 = np.zeros((14,14), dtype=int)
lr_sec2[11,0], lr_sec2[0,11] = 1,1
lr_sec2[2,1], lr_sec2[1,2] = 1,1
lr_sec2[0,2], lr_sec2[2,0] = 1,1
lr_sec2[1,3], lr_sec2[3,1] = 1,1
lr_sec2[5,4], lr_sec2[4,5] = 1,1
lr_sec2[6,5], lr_sec2[5,6] = 1,1
lr_sec2[10,6], lr_sec2[6,10] = 1,1
lr_sec2[4,7], lr_sec2[7,4] = 1,1
lr_sec2[7,8], lr_sec2[8,7] = 1,1
lr_sec2[8,9], lr_sec2[9,8] = 1,1
lr_sec2[3,10], lr_sec2[10,3] = 1,1
lr_sec2[12,11], lr_sec2[11,12] = 1,1
```

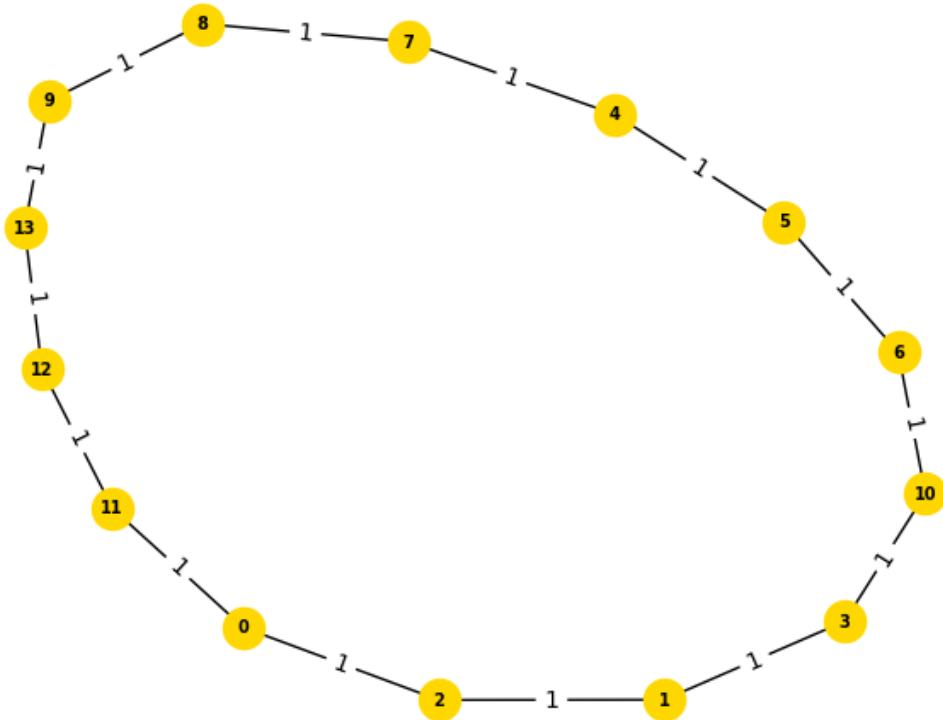
```
lr_sec2[13,12], lr_sec2[12,13]=1,1
lr_sec2[9,13], lr_sec2[13,9]=1,1
```

```
[18]: weighted_graph = grafo(lr_sec2)

# Subgrafo con peso >0
edges_to_keep = [(u, v) for u, v, d in weighted_graph.edges(data=True) if d['weight'] > 0]
subgraph = weighted_graph.edge_subgraph(edges_to_keep)

# Imprimimos subgrafo
pos = nx.spring_layout(subgraph)
nx.draw(subgraph, pos, with_labels=True, node_size=300, node_color="gold",
        font_size=7, font_color="black", font_weight="bold", arrowsize=10)
labels = nx.get_edge_attributes(subgraph, 'weight')
nx.draw_networkx_edge_labels(subgraph, pos, edge_labels=labels)
plt.title("Solución de la relajación añadiendo más SEC")
plt.show()
```

Solución de la relajación añadiendo más SEC



Es una solución entera que no presenta subtours! Así pues hemos hallado la solución óptima de

nuestro problema TSP. Si observamos los resultados que nos ha devuelto ampl vemos que esta solución tiene un coste de 253. Recordemos que la solución de la relajación lineal sin imponer las SEC que sabemos que no se satisfacen es de 220.

Dibujamos ahora el grafo con los costes correspondientes.

```
[26]: optim_tour = [0,11,12,13,9,8,7,4,5,6,10,3,1,2,0]
```

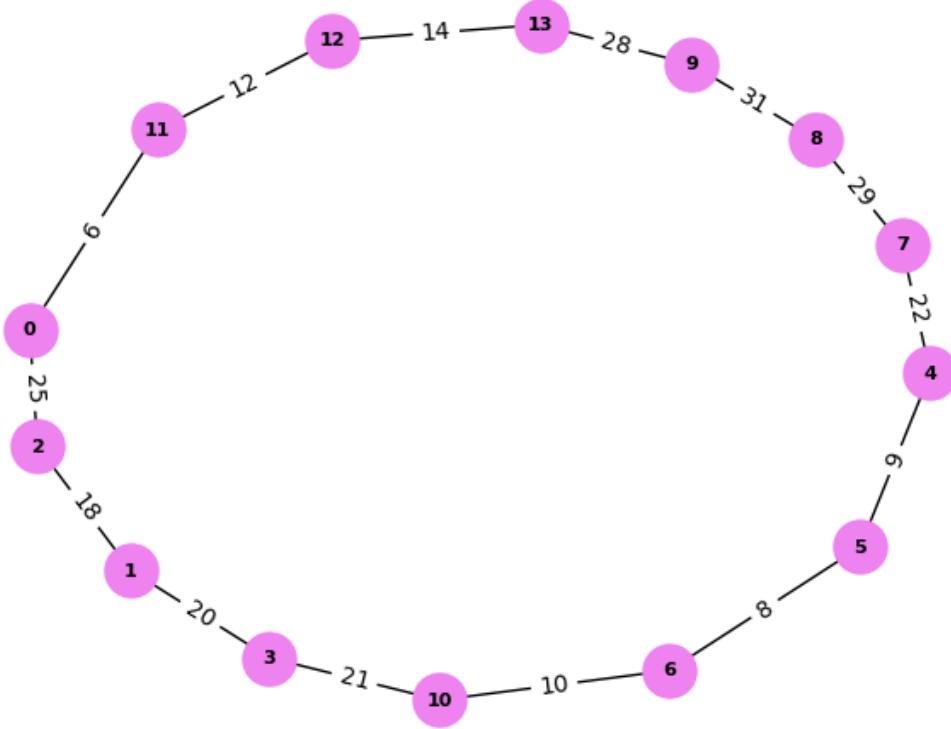
```
[28]: #Matriz de distancias para sol de tabu search
bares_optim=new_matrix(bares_time, optim_tour)

weighted_graph = grafo(bares_optim)

# Subgrafo con peso >0
edges_to_keep = [(u, v) for u, v, d in weighted_graph.edges(data=True) if
                  d['weight'] > 0]
subgraph = weighted_graph.edge_subgraph(edges_to_keep)

# Imprimimos subgrafo
pos = nx.spring_layout(subgraph)
nx.draw(subgraph, pos, with_labels=True, node_size=500, node_color="violet",
        font_size=8, font_color="black", font_weight="bold", arrowsize=10)
labels = nx.get_edge_attributes(subgraph, 'weight')
nx.draw_networkx_edge_labels(subgraph, pos, edge_labels=labels)
plt.title("Solución óptima")
plt.show()
```

### Solución óptima



Comprobamos que el coste es de 253 tal y como nos indica AMPL.

```
[21]: cost(bares_time, optim_tour)
```

```
[21]: 253
```

## 1.1 Explorando un poco más allá

En la parte que sigue vemos otras formas de explorar la solución de nuestro TSP. Lo que vamos a hacer será la siguiente exploración, resolviendo las relajaciones lineales con el solver AMPL-Gurobi.

- Primero resolvemos la relajación lineal inicial, obteniendo la solución con siete pares de subtours que hemos impreso anteriormente en este notebook. A pesar de que podemos ver a simple vista todas las SEC violada sin necesidad de usar un algoritmo de separación, solo vamos a añadir uno de los subconjuntos de nodos que viola la SEC.
- Añadimos la SEC violada por el subconjunto  $W = \{1, 2\}$ .
- Añadimos la SEC violada por el subconjunto  $W_1 = \{4, 5\}$ .
- Añadimos la SEC violada por el subconjunto  $W_2 = \{12, 13\}$ .
- Añadimos la SEC violada por el subconjunto  $W_3 = \{10, 4, 5, 6\}$ .
- Añadimos la *2-match inequality* definida por los subconjubtos  $H = \{4, 5, 6\}$ ,  $T_1 = \{4, 7\}$ ,  $T_2 = \{5, 9\}$ ,  $T_3 = \{6, 10\}$ .

- Añadimos la *comb inequality* definida por los subconjuntos  $H = \{5, 6, 7\}$ ,  $T_1 = \{6, 10\}$ ,  $T_2 = \{4, 5\}$ ,  $T_3 = \{7, 8, 9\}$ .

Lo que vamos a hacer en este notebook es definir las matrices solución que nos devuelven el solver AMPL-Gurobi y representar los respectivos grafos para cada uno de los pasos. Estos grafos servirán para ver como se comportan las soluciones.

Empezamos definiendo la matriz que nos devuelve el solver después de añadir la primera SEC (con  $W = \{1, 2\}$ ) y representando el grafo correspondiente.

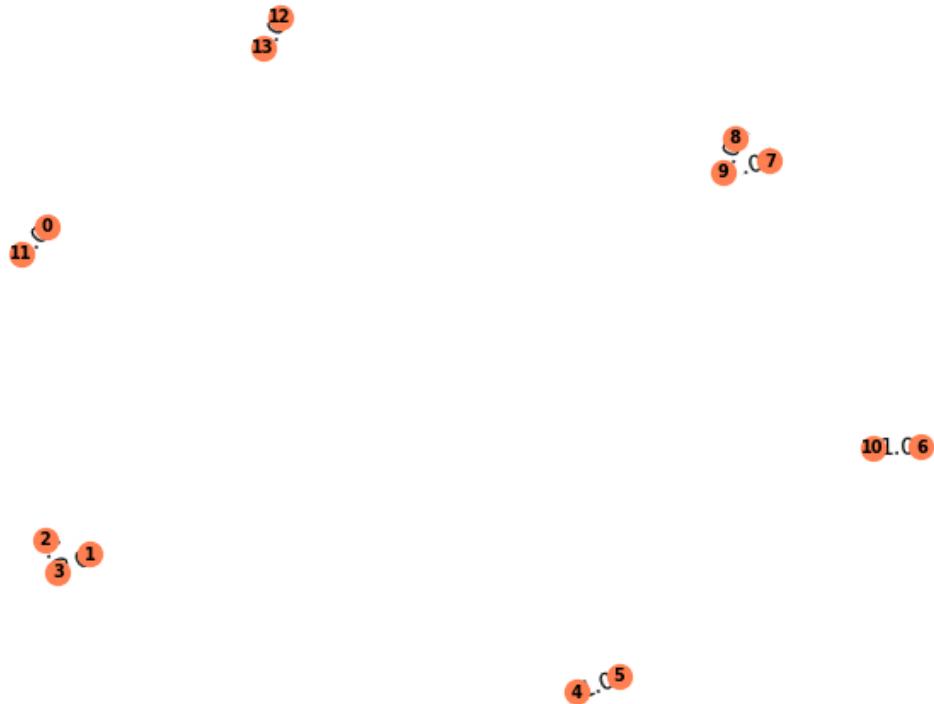
```
[22]: mat1 = np.zeros((14,14))
mat1[0,11],mat1[11,0]=1,1
mat1[1,2],mat1[2,1]=1,1
mat1[2,3],mat1[3,2]=1,1
mat1[3,1],mat1[1,3]=1,1
mat1[4,5],mat1[5,4]=1,1
mat1[5,4],mat1[4,5]=1,1
mat1[6,10],mat1[10,6]=1,1
mat1[7,9],mat1[9,7]=1,1
mat1[8,7],mat1[7,8]=1,1
mat1[9,8],mat1[8,9]=1,1
mat1[10,6],mat1[6,10]=1,1
mat1[0,11],mat1[11,0]=1,1
mat1[12,13],mat1[13,12]=1,1
mat1[12,13],mat1[13,12]=1,1
```

```
[23]: weighted_graph = grafo(mat1)

# Subgrafo con peso >0
edges_to_keep = [(u, v) for u, v, d in weighted_graph.edges(data=True) if
                  d['weight']>0]
subgraph = weighted_graph.edge_subgraph(edges_to_keep)

# Imprimimos subgrafo
pos = nx.spring_layout(subgraph)
nx.draw(subgraph, pos, with_labels=True, node_size=100, node_color="coral",
        font_size=7, font_color="black", font_weight="bold", arrowsize=10)
labels = nx.get_edge_attributes(subgraph, 'weight')
nx.draw_networkx_edge_labels(subgraph, pos, edge_labels=labels)
plt.title("Solución de la relajación añadiendo más SEC")
plt.show()
```

## Solución de la relajación añadiendo más SEC



El coste de esta solución es de 227 (el coste de la primera relajación era de 220). Claramente vemos diferentes SEC que se violan. No obstante solo añadimos la SEC violada por el subconjunto  $W_1 = \{4, 5\}$ . Resolvemos la relajación con AMPL-Gurobi. Definimos la matriz solución que nos devuelve AMPL.

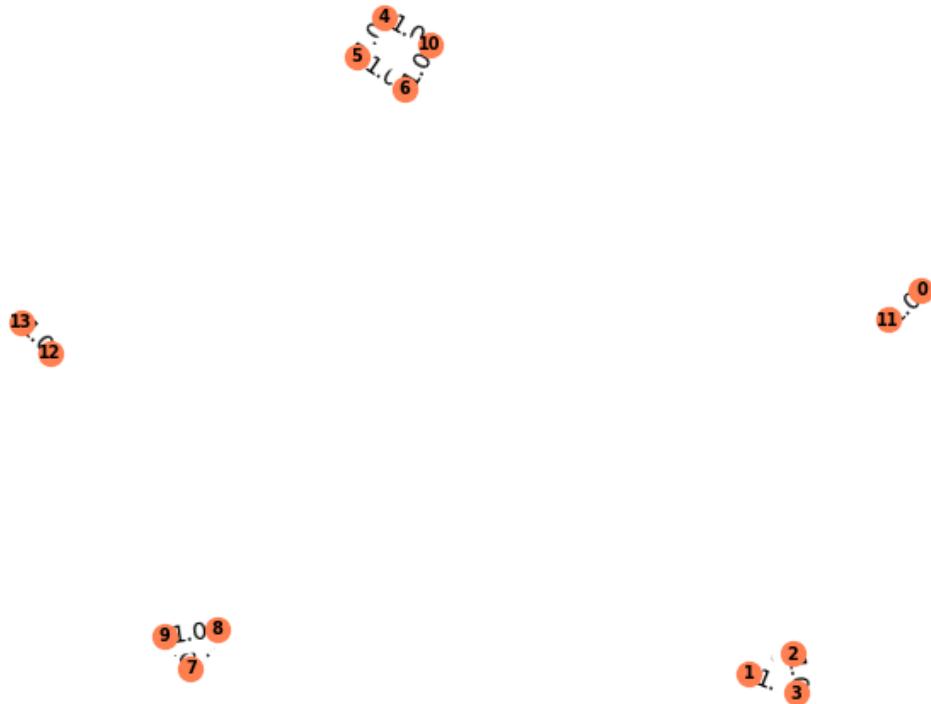
```
[24]: mat2 = np.zeros((14,14))
mat2[0,11],mat2[11,0]=1,1
mat2[1,2],mat2[2,1]=1,1
mat2[2,3],mat2[3,2]=1,1
mat2[3,1],mat2[1,3]=1,1
mat2[4,10],mat2[10,4]=1,1
mat2[5,4],mat2[4,5]=1,1
mat2[6,5],mat2[5,6]=1,1
mat2[7,9],mat2[9,7]=1,1
mat2[8,7],mat2[7,8]=1,1
mat2[9,8],mat2[8,9]=1,1
mat2[10,6],mat2[6,10]=1,1
mat2[11,0],mat2[0,11]=1,1
mat2[12,13],mat2[13,12]=1,1
mat2[13,12],mat2[12,13]=1,1
```

```
[25]: weighted_graph = grafo(mat2)

# Subgrafo con peso >0
edges_to_keep = [(u, v) for u, v, d in weighted_graph.edges(data=True) if
    d['weight']>0]
subgraph = weighted_graph.edge_subgraph(edges_to_keep)

# Imprimimos subgrafo
pos = nx.spring_layout(subgraph)
nx.draw(subgraph, pos, with_labels=True, node_size=100, node_color="coral",
        font_size=7, font_color="black", font_weight="bold", arrowsize=10)
labels = nx.get_edge_attributes(subgraph, 'weight')
nx.draw_networkx_edge_labels(subgraph, pos, edge_labels=labels)
plt.title("Solución de la relajación añadiendo más SEC")
plt.show()
```

Solución de la relajación añadiendo más SEC



Esta solución es de coste 231. Añadimos la SEC violada por el subconjunto de nodos  $W_2 = \{12, 13\}$ . Definimos la matriz de la solución que nos devuelve AMPL.

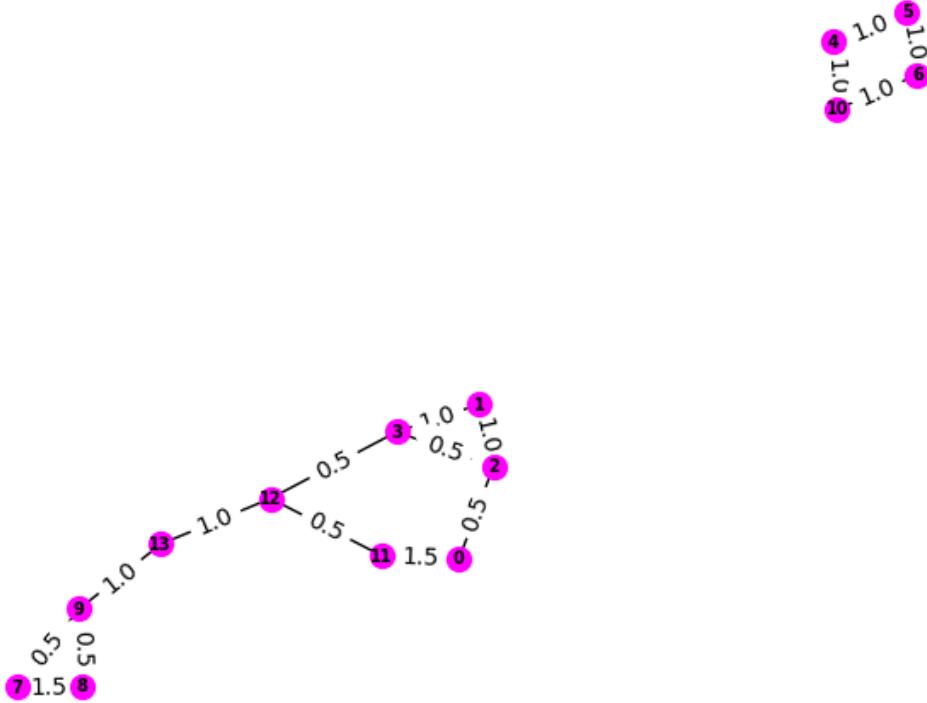
```
[50]: mat3 = np.zeros((14,14))
mat3[0,11],mat3[11,0]=1.5,1.5
mat3[1,2],mat3[2,1]=1,1
mat3[1,3],mat3[3,1]=1,1
mat3[2,0],mat3[0,2]=0.5,0.5
mat3[3,2],mat3[2,3]=0.5,0.5
mat3[4,10],mat3[10,4]=1,1
mat3[5,4],mat3[4,5]=1,1
mat3[6,5],mat3[5,6]=1,1
mat3[7,8],mat3[8,7]=1.5,1.5
mat3[7,9],mat3[9,7]=0.5,0.5
mat3[9,8],mat3[8,9]=0.5,0.5
mat3[9,13],mat3[13,9]=1,1
mat3[10,6],mat3[6,10]=1,1
mat3[11,12],mat3[12,11]=0.5,0.5
mat3[12,3],mat3[3,12]=0.5,0.5
mat3[12,13],mat3[13,12]=1,1
```

```
[55]: weighted_graph = grafo(mat3)

# Subgrafo con peso >0
edges_to_keep = [(u, v) for u, v, d in weighted_graph.edges(data=True) if
    ↪d['weight']>0]
subgraph = weighted_graph.edge_subgraph(edges_to_keep)

# Imprimimos subgrafo
pos = nx.spring_layout(subgraph)
nx.draw(subgraph, pos, with_labels=True, node_size=100, node_color="fuchsia",
        font_size=7, font_color="black", font_weight="bold", arrowsize=10)
labels = nx.get_edge_attributes(subgraph, 'weight')
nx.draw_networkx_edge_labels(subgraph, pos, edge_labels=labels)
plt.title("Solución de la relajación añadiendo más SEC")
plt.show()
```

## Solución de la relajación añadiendo más SEC



Esta solución tiene un coste de 243. Vemos que claramente el subconjunto  $W_3 = \{10, 4, 5, 6\}$ , añadimos pues la SEC violada por este subconjunto, resolvemos la relajación lineal y definimos la matriz solución que nos devuelve el solver.

Podemos observar que esta solución anterior es no entera, es la primera solución no entera que hemos obtenido. Además, se puede ver que hay nodos que están conectados por pesos mayores que uno, estos nodos son  $\{0, 11\}, \{7, 8\}$ . Usando el algoritmo de separación heurístico visto en clase, sabemos que estos subconjuntos de nodos violan una SEC. Esto tiene todo el sentido, pues los dos pares de nodos  $\{0, 11\}, \{7, 8\}$ , son de los que al inicio violaban una SEC, por lo tanto tienen que seguir incumpliendo la restricción.

```
[56]: mat4=np.zeros((14,14))
mat4[0,11],mat4[11,0]=1.5,1.5
mat4[1,2],mat4[2,1]=1,1
mat4[2,0],mat4[0,2]=0.5,0.5
mat4[2,3],mat4[3,2]=0.5,0.5
mat4[3,1],mat4[1,3]=1,1
mat4[4,5],mat4[5,4]=1,1
mat4[4,7],mat4[7,4]=1,1
mat4[5,6],mat4[6,5]=0.5,0.5
```

```

mat4[6,10],mat4[10,6]=1.5,1.5
mat4[7,8],mat4[8,7]=1,1
mat4[8,9],mat4[9,8]=1,1
mat4[9,5],mat4[5,9]=0.5,0.5
mat4[10,13],mat4[13,10]=0.5,0.5
mat4[11,12],mat4[12,11]=0.5,0.5
mat4[12,3],mat4[3,12]=0.5,0.5
mat4[12,13],mat4[13,12]=1,1
mat4[13,9],mat4[9,13]=0.5,0.5

```

```

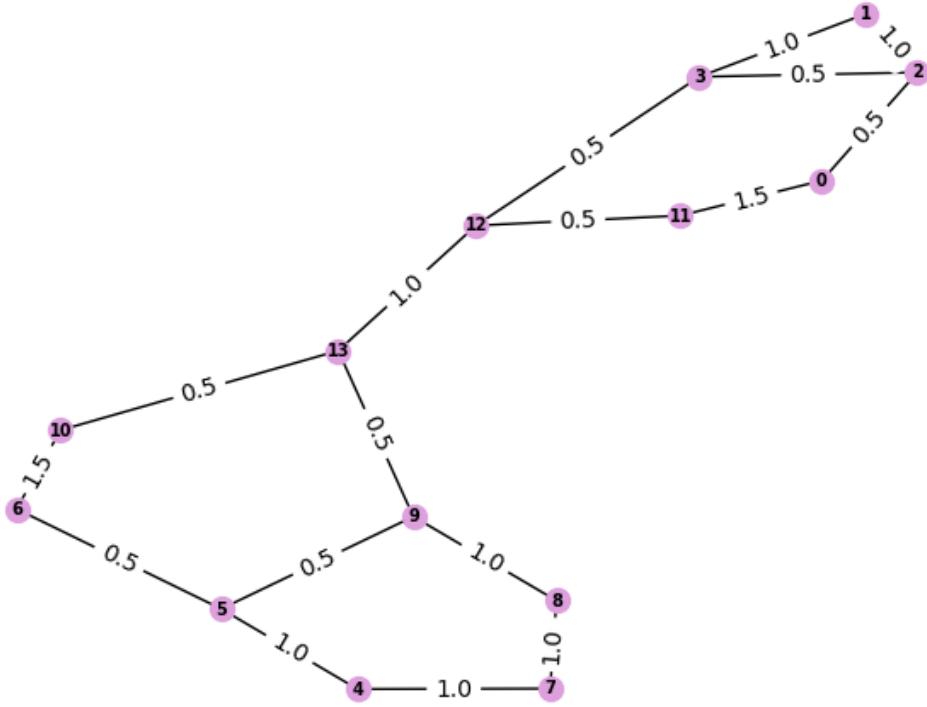
[58]: weighted_graph = grafo(mat4)

# Subgrafo con peso >0
edges_to_keep = [(u, v) for u, v, d in weighted_graph.edges(data=True) if
    ↪d['weight']>0]
subgraph = weighted_graph.edge_subgraph(edges_to_keep)

# Imprimimos subgrafo
pos = nx.spring_layout(subgraph)
nx.draw(subgraph, pos, with_labels=True, node_size=100, node_color="plum",
        font_size=7, font_color="black", font_weight="bold", arrowsize=10)
labels = nx.get_edge_attributes(subgraph, 'weight')
nx.draw_networkx_edge_labels(subgraph, pos, edge_labels=labels)
plt.title("Solución de la relajación añadiendo más SEC")
plt.show()

```

### Solución de la relajación añadiendo más SEC



El coste de esta solución es de 247.5. El lector puede observar que ahora no se ven subconjuntos que no satisfacen las SEC a simple vista. Lo que haremos ahora es añadir la 2-match inequality violada por los subconjuntos  $H = \{4, 5, 6\}$ ,  $T_1 = \{4, 7\}$ ,  $T_2 = \{5, 9\}$ ,  $T_3 = \{6, 10\}$ .

```
[59]: mat5=np.zeros((14,14))
mat5[0,11],mat5[11,0]=1.5,1.5
mat5[1,2],mat5[2,1]=1,1
mat5[2,0],mat5[0,2]=0.5,0.5
mat5[2,3],mat5[3,2]=0.5,0.5
mat5[3,1],mat5[1,3]=1,1
mat5[4,5],mat5[5,4]=1,1
mat5[4,7],mat5[7,4]=1,1
mat5[5,6],mat5[6,5]=0.5,0.5
mat5[6,10],mat5[10,6]=1.5,1.5
mat5[7,5],mat5[5,7]=0.5,0.5
mat5[8,7],mat5[7,8]=0.5,0.5
mat5[9,8],mat5[8,9]=1.5,1.5
mat5[10,13],mat5[13,10]=0.5,0.5
mat5[11,12],mat5[12,11]=0.5,0.5
mat5[12,3],mat5[3,12]=0.5,0.5
```

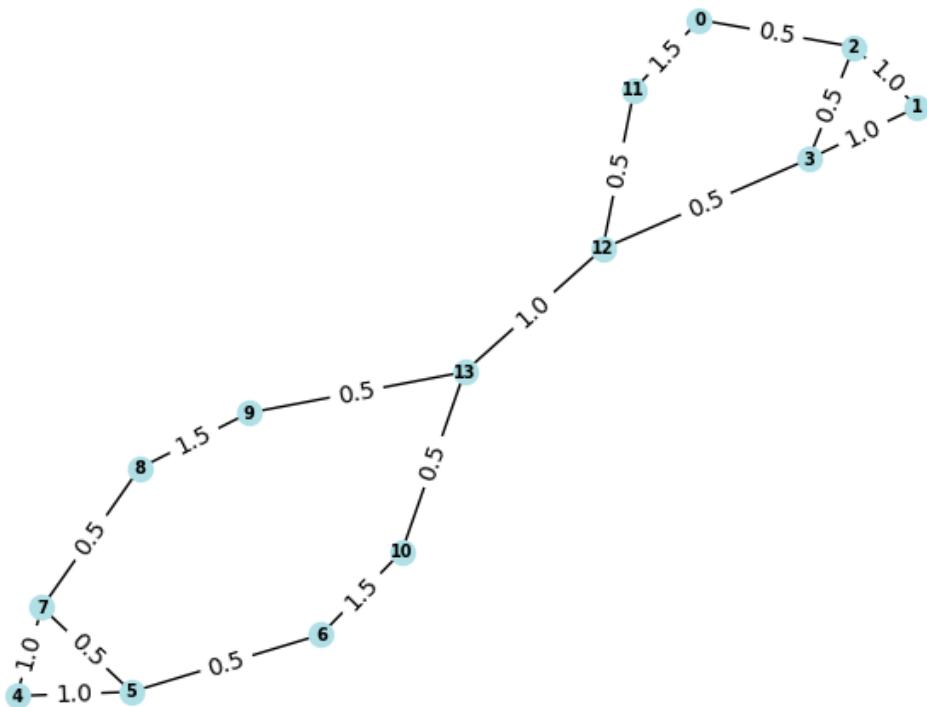
```
mat5[12,13],mat5[13,12]=1,1  
mat5[13,9],mat5[9,13]=0.5,0.5
```

```
[60]: weighted_graph = grafo(mat5)

# Subgrafo con peso >0
edges_to_keep = [(u, v) for u, v, d in weighted_graph.edges(data=True) if
                  d['weight']>0]
subgraph = weighted_graph.edge_subgraph(edges_to_keep)

# Imprimimos subgrafo
pos = nx.spring_layout(subgraph)
nx.draw(subgraph, pos, with_labels=True, node_size=100, node_color="powderblue",
        font_size=7, font_color="black", font_weight="bold", arrowsize=10)
labels = nx.get_edge_attributes(subgraph, 'weight')
nx.draw_networkx_edge_labels(subgraph, pos, edge_labels=labels)
plt.title("Solución de la relajación añadiendo 2-match inequality")
plt.show()
```

### Solución de la relajación añadiendo 2-match inequality



El coste de esta solución es de 248. Añadimos la comb inequality violada por los subconjuntos

$H = \{6, 5, 7\}$ ,  $T_1 = \{6, 10\}$ ,  $T_2 = \{5, 4\}$ ,  $T_3 = \{7, 8, 9\}$ .

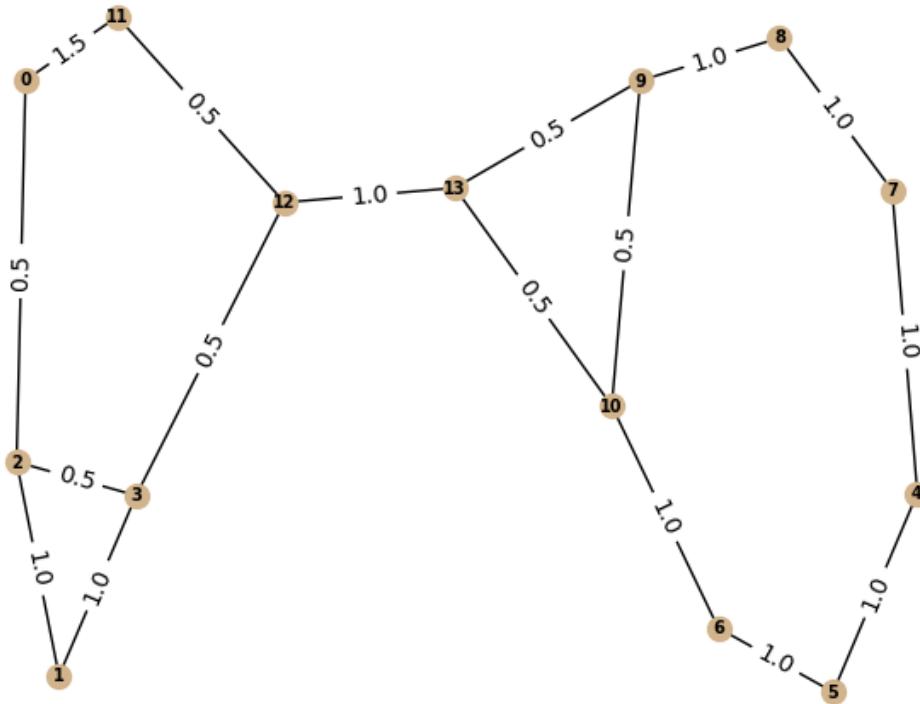
```
[61]: mat6=np.zeros((14,14))
mat6[0,11],mat6[11,0]=1.5,1.5
mat6[1,2],mat6[2,1]=1,1
mat6[2,0],mat6[0,2]=0.5,0.5
mat6[2,3],mat6[3,2]=0.5,0.5
mat6[3,1],mat6[1,3]=1,1
mat6[4,5],mat6[5,4]=1,1
mat6[5,6],mat6[6,5]=1,1
mat6[6,10],mat6[10,6]=1,1
mat6[7,4],mat6[4,7]=1,1
mat6[8,7],mat6[7,8]=1,1
mat6[9,8],mat6[8,9]=1,1
mat6[10,9],mat6[9,10]=0.5,0.5
mat6[10,13],mat6[13,10]=0.5,0.5
mat6[11,12],mat6[12,11]=0.5,0.5
mat6[12,3],mat6[3,12]=0.5,0.5
mat6[12,13],mat6[13,12]=1,1
mat6[13,9],mat6[9,13]=0.5,0.5
```

```
[62]: weighted_graph = grafo(mat6)

# Subgrafo con peso >0
edges_to_keep = [(u, v) for u, v, d in weighted_graph.edges(data=True) if d['weight']>0]
subgraph = weighted_graph.edge_subgraph(edges_to_keep)

# Imprimimos subgrafo
pos = nx.spring_layout(subgraph)
nx.draw(subgraph, pos, with_labels=True, node_size=100, node_color="tan",
        font_size=7, font_color="black", font_weight="bold", arrowsize=10)
labels = nx.get_edge_attributes(subgraph, 'weight')
nx.draw_networkx_edge_labels(subgraph, pos, edge_labels=labels)
plt.title("Solución de la relajación añadiendo comb inequality")
plt.show()
```

### Solución de la relajación añadiendo comb inequality



El coste de esta solución es de nuevo 248, pero podemos ver que obtenemos más soluciones enteras que en el paso anterior. Usamos ahora un algoritmo de separación, usamos manualmente el método de separación heurístico visto en clase, y vemos rápidamente que el subconjunto de nodos  $W_4 = \{1, 2, 3\}$  viola una SEC. Y también el subconjunto  $W_5 = \{0, 11\}$  y el  $W_6 = \{10, 6, 5, 4, 7, 8, 9\}$ . Añadimos las correspondientes SEC a nuestro problema y resolvemos la relajación lineal usando AMPL. Definimos a continuación la matriz solución que obtenemos.

```
[32]: mat7=np.zeros((14,14))
mat7[0,11],mat7[11,0]=1,1
mat7[1,2],mat7[2,1]=1,1
mat7[2,0],mat7[0,2]=1,1
mat7[3,1],mat7[1,3]=1,1
mat7[4,3],mat7[3,4]=1,1
mat7[5,4],mat7[4,5]=1,1
mat7[6,10],mat7[10,6]=1,1
mat7[7,8],mat7[8,7]=1,1
mat7[8,7],mat7[7,8]=1,1
mat7[9,5],mat7[5,9]=1,1
mat7[10,6],mat7[6,10]=1,1
mat7[11,12],mat7[12,11]=1,1
```

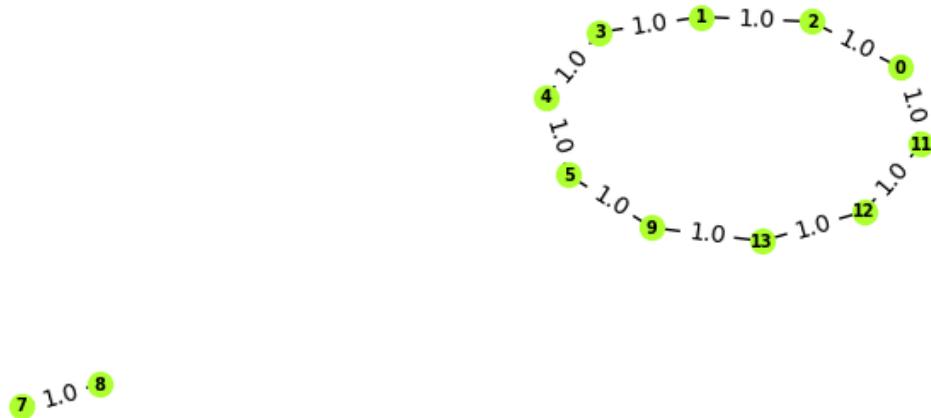
```
mat7[12,13],mat7[13,12]=1,1
mat7[13,9],mat7[9,13]=1,1
```

```
[33]: weighted_graph = grafo(mat7)

# Subgrafo con peso >0
edges_to_keep = [(u, v) for u, v, d in weighted_graph.edges(data=True) if
                  d['weight']>0]
subgraph = weighted_graph.edge_subgraph(edges_to_keep)

# Imprimimos subgrafo
pos = nx.spring_layout(subgraph)
nx.draw(subgraph, pos, with_labels=True, node_size=100,
        node_color="greenyellow",
        font_size=7, font_color="black", font_weight="bold", arrowsize=10)
labels = nx.get_edge_attributes(subgraph, 'weight')
nx.draw_networkx_edge_labels(subgraph, pos, edge_labels=labels)
plt.title("Solución de la relajación SEC con algoritmo heurístico")
plt.show()
```

Solución de la relajación SEC con algoritmo heurístico



10 6

El coste de esta solución es de 249. Vemos que claramente la solución obtenida tiene tres subtours. Así, añadimos las SEC violadas por  $W_7 = \{6, 10\}$ ,  $W_8 = \{8, 7\}$ , ya que no podemos hacer otra cosa con la solución que tenemos.

```
[12]: tour = [0,2,1,3,9,8,7,4,5,6,10,13,12,11,0]
```

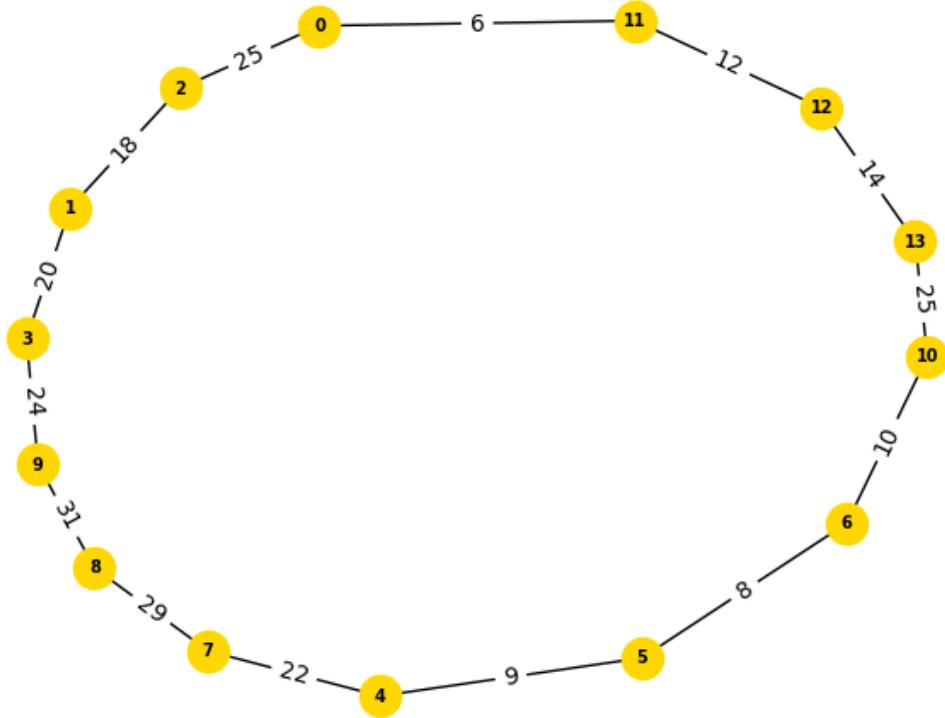
```
[15]: mat = new_matrix(bares_time,tour)

weighted_graph = grafo(mat)

# Subgrafo con peso >0
edges_to_keep = [(u, v) for u, v, d in weighted_graph.edges(data=True) if
    ↪d['weight'] > 0]
subgraph = weighted_graph.edge_subgraph(edges_to_keep)

# Imprimimos subgrafo
pos = nx.spring_layout(subgraph)
nx.draw(subgraph, pos, with_labels=True, node_size=300, node_color="gold",
        font_size=7, font_color="black", font_weight="bold", arrowsize=10)
labels = nx.get_edge_attributes(subgraph, 'weight')
nx.draw_networkx_edge_labels(subgraph, pos, edge_labels=labels)
plt.title("Solución de la relajación añadiendo más SEC")
plt.show()
```

## Solución de la relajación añadiendo más SEC



```
[17]: cost(bares_time,tour)
```

```
[17]: 253
```

Así pues hemos vuelto a obtener la solución óptima! No obstante, podemos observar que las dos soluciones óptimas encontradas son diferentes! Es decir, son soluciones óptimas alternativas a nuestro problema. Las dos son soluciones fáciles con coste mínimo pero los tours son diferentes.

```
[34]: print("Solución óptima anterior:", optim_tour, "\nSolución óptima encontrada ahora:", tour)
```

Solución óptima anterior: [0, 11, 12, 13, 9, 8, 7, 4, 5, 6, 10, 3, 1, 2, 0]  
Solución óptima encontrada ahora: [0, 2, 1, 3, 9, 8, 7, 4, 5, 6, 10, 13, 12, 11, 0]

```

param n;
set bares:=1..n;
#W,W1,W2,...,W7 son los pares de bares que violan SEC en la
#solución devuelta por la primera relajación lineal
set W;
set W2;
set W3;
set W4;
set W5;
set W6;
set W7;
#S1,S2,S3 son los subconjuntos de bares que violan SEC en
#la solución de la segunda relajación lineal
set S1;
set S2;
set S3;

```

```

param TIME{bares, bares};

#Variable x relajada
var x{i in bares, j in bares}>=0,<=1;
#Función objetivo
minimize TotalTime:
sum{i in bares, j in bares} TIME[i, j]*x[i, j];
#degree constraint
subject to degree_constraints {i in bares}:
sum{j in bares} x[i, j] = 1;
#degree constraint
subject to degree_constraints2 {i in bares}:
sum{j in bares} x[j, i] = 1;

#SEC
subject to sec:
sum{i in W, j in W: i<>j} x[i,j] <= card(W)-1;

subject to sec2:
sum{i in W2, j in W2: i<>j} x[i,j] <= card(W2)-1;

subject to sec3:
sum{i in W3, j in W3: i<>j} x[i,j] <= card(W3)-1;

subject to sec4:
sum{i in W4, j in W4: i<>j} x[i,j] <= card(W4)-1;

subject to sec5:

```

```
sum{i in W5, j in W5: i<>j} x[i,j] <= card(W5)-1;  
  
subject to sec6:  
sum{i in W6, j in W6: i<>j} x[i,j] <= card(W6)-1;  
  
subject to sec7:  
sum{i in W7, j in W7: i<>j} x[i,j] <= card(W7)-1;  
  
subject to sec8:  
sum{i in S1, j in S1: i<>j} x[i,j] <= card(S1)-1;  
  
subject to sec9:  
sum{i in S2, j in S2: i<>j} x[i,j] <= card(S2)-1;  
  
subject to sec10:  
sum{i in S3, j in S3: i<>j} x[i,j] <= card(S3)-1;
```

```
#Archivo .dat donde entramos los datos de nuestras relajaciones
lineales
param n := 14;
set W := 1,12;
set W2 := 2,3;
set W3 := 8,9;
set W4 := 5,6;
set W5 := 4,10;
set W6 := 7,11;
set W7 := 13,14;
set S1 := 14,13,12,1,2,3,4;
set S2 := 8,9,10;
set S3 := 11,5,6,7;
param TIME : 1 2 3 4 5 6 7 8 9 10 11 12 13 14 :=  
1 1000 19 25 21 36 33 37 39 56 35 35 6 17 25  
2 19 1000 18 20 32 28 32 36 45 32 32 29 35 39  
3 25 18 1000 28 38 33 37 40 50 39 37 36 41 42  
4 21 20 28 1000 22 18 23 27 37 24 21 28 18 27  
5 36 32 38 22 1000 9 16 22 36 26 15 48 38 34  
6 33 28 33 18 9 1000 8 16 28 17 10 44 33 30  
7 37 32 37 23 16 8 1000 21 32 22 10 44 33 30  
8 39 36 40 27 22 16 21 1000 29 23 24 53 47 42  
9 56 45 50 37 36 28 32 29 1000 31 29 63 53 48  
10 35 32 39 24 26 17 22 23 31 1000 20 42 32 28  
11 35 32 37 21 15 10 10 24 29 20 1000 38 27 25  
12 6 29 36 28 48 44 44 53 63 42 38 1000 12 19  
13 17 35 41 18 38 33 33 47 53 32 27 12 1000 14  
14 25 39 42 27 34 30 30 42 48 28 25 19 14 1000;
```

stsp.run

sábado, 30 de diciembre de 2023 9:49

```
reset;
model stsp.mod;
data stsp.dat;
option solver gurobi;
solve;

display TotalTime;
display x;
```

```
#Obs: en esta segunda parte resolvemos el problema usando
#la matriz triangular superior y adaptando el modelo para
#trabajar con los datos de esta forma.
#En el .run añadimos instrucciones que nos permiten ver las
#variables básicas en cada solución.
param n;
set bares:=0..n;
#W,W1,W2,...,W7 son los pares de bares que violan SEC en la
#solución devuelta por la primera relajación lineal
set W;
set W2;
set W3;
set W4;

#S1,S2,S3 SEC violadas tras añadir 2-match y comb ineq
set S1;
set S2;
set S3;
set W3b;
set W5;

#2-match ineq
set H;
set T1;
set T2;
set T3;

#comb ineq
set H1;
set T1b;
set T2b;
set T3b;

param TIME{bares, bares};

#Variable x relajada
var x{i in bares, j in bares: j>i}>=0,<=1;

#Función objetivo
minimize TotalTime:
sum{i in bares, j in bares: j>i} TIME[i, j]*x[i, j];

#degree constraints
subject to degree_constraints {i in bares}:
sum{j in bares: j < i} x[j,i] + sum{k in bares: i < k} x[i,k] = 2;

#SEC
subject to sec:
```

```

sum{i in W, j in W: i<j} x[i,j] <= card(W)-1;

subject to sec2:
sum{i in W2, j in W2: i<j} x[i,j] <= card(W2)-1;

subject to sec3:
sum{i in W3, j in W3: i<j} x[i,j] <= card(W3)-1;

subject to sec4:
sum{i in W4, j in W4: i<j} x[i,j] <= card(W4)-1;

subject to sec9:
sum{i in S1, j in S1: i<j} x[i,j] <= card(S1)-1;

subject to sec10:
sum{i in S2, j in S2: i<j} x[i,j] <= card(S2)-1;

subject to sec11:
sum{i in S3, j in S3: i<j} x[i,j] <= card(S3)-1;

subject to sec5:
sum{i in W3b, j in W3b: i<j} x[i,j] <= card(W3b)-1;

subject to sec6:
sum{i in W5, j in W5: i<j} x[i,j] <= card(W5)-1;

#2-match ineq
subject to 2match:
sum{i in H, j in H: i<j} x[i,j] + sum{i in T1, j in T1: i<j} x[i,j]
+ sum{i in T2, j in T2: i<j} x[i,j] + sum{i in T3, j in T3: i<j}
x[i,j] <= card(H) + 1;

#Comb inequality
subject to comb:
sum{i in H1, j in H1: i<j} x[i,j] + sum{i in T1b, j in T1b: i<j}
x[i,j] + sum{i in T2b, j in T2b: i<j} x[i,j] + sum{i in T3b, j in
T3b: i<j} x[i,j] <= card(H1) + card(T1b) - 1 + card(T2b) - 1 +
card(T3b) - 1 - 2;

```

```
param n := 13;
set W:= 1,2;
set W2:= 4,5;
set W3=12,13;
set W4 := 10,4,5,6;
set W3b := 6,10;
set W5 :=7,8;
set S1 := 1,2,3;
set S2 := 0,11;
set S3 := 10,6,5,4,7,8,9;
set H := 4,5,6;
set T1 := 4,7;
set T2 := 5,9;
set T3 := 6,10;
set H1:=5,6,7;
set T1b := 6,10;
set T2b :=5,4;
set T3b := 7,8,9;
param TIME: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 :=
  0 1000 19 25 21 36 33 37 39 56 35 35 6 17 25
  1 . 1000 18 20 32 28 32 36 45 32 32 29 35 39
  2 . . 1000 28 38 33 37 40 50 39 37 36 41 42
  3 . . . 1000 22 18 23 27 37 24 21 28 18 27
  4 . . . . 1000 9 16 22 36 26 15 48 38 34
  5 . . . . . 1000 8 16 28 17 10 44 33 30
  6 . . . . . . 1000 21 32 22 10 44 33 30
  7 . . . . . . . 1000 29 23 24 53 47 42
  8 . . . . . . . . 1000 31 29 63 53 48
  9 . . . . . . . . . 1000 20 42 32 28
  10 . . . . . . . . . . 1000 38 27 25
  11 . . . . . . . . . . . 1000 12 19
  12 . . . . . . . . . . . . 1000 14
  13 . . . . . . . . . . . . . 1000;
```

stsp\_LR.run

viernes, 19 de enero de 2024 12:06

```
reset;
model final.mod;
data final.dat;
option solver gurobi;
solve;

display TotalTime;
display x;
display x.sstatus;
```