Universidade Vigo

UNIVERSIDADE DA CORUÑA

Traballo Fin de Master

El problema del cartero chino

Jesús Yordá Pérez

2014 – 15

Máster Interuniversitario en Técnicas Estadísticas

MASTER EN TÉCNICAS ESTADÍSTICAS

Traballo Fin de Master

El problema del cartero chino

Jesús Yordá Pérez

septiembre de 2015

Máster Interuniversitario en Técnicas Estadísticas

Índice general

Resumen

La Investigación de Operaciones o Investigación Operativa hace uso de métodos cuantitativos como herramienta de apoyo para el proceso de toma de decisiones. Muchos de los problemas de investigación operativa pueden plantearse como problemas de programación lineal, sin embargo, pese a que el planteamiento es correcto, cuando el tamaño del problema crece el tiempo computacional se vuelve excesivo. Por esto, los matemáticos intentan desarrollar algoritmos, unas veces exactos, y otras veces heuristicos, que resuelvan estos problemas en un tiempo razonable.

En esta memoria introduciremos brevemente el problema del cartero chino con sus principales variantes, junto con los principales algoritmos para resolverlo. Por otra parte presentaremos un paquete en R capaz de resolver el problema del cartero chino y lo pondremos a prueba con datos reales cedidos por la empresa Ecourense.

Abstract

Operations Research uses quantitative methods as a support tool for decision-making process. Many of the operations research problems may arise as linear programming problems, however, although the approach to linear programming problem is correct, when the problem size increases the computational time becomes excessive. Therefore, mathematicians try to develop mathematical algorithms, sometimes accurate, sometimes heuristics, to solve these problems in a reasonable time.

In this work we introduce the problem of Chinese postman with its main variants, with major algorithms to solve it. Furthermore we will present a package in R able to solve the problem of Chinese postman and will test real data provided by the company Ecourense.

Introdución

La investigación de operaciones o investigación operativa o investigación operacional (conocida también como teoría de la toma de decisiones, programación matemática o I.O.) hace uso de métodos cuantitativos como herramienta de apoyo para el proceso de toma de decisiones, algo que antaño era mucho mas sencillo, debido al menor número de variables presentes en los problemas que se planteaban, pero que hoy en día se vuelve cada mas costoso, por el incremento del número de variables.

La Investigación de Operacione es una disciplina donde las primeras actividades formales se dieron en Inglaterra en la Segunda Guerra Mundial, cuando se encarga a un grupo de científicos ingleses, encabezados por A. P. Rowe, el diseño de herramientas cuantitativas para el apoyo a la toma de decisiones acerca de la mejor utilización de materiales bélicos. De hecho, debido a este origen, se presume que el nombre de Investigación de Operaciones fue dado porque el equipo de científicos estaba llevando a cabo la actividad de Investigar Operaciones (militares), de todos modos esto no son mas que conjeturas, dado que incluso hay quien afirma que el origen de la investigación operativa se encuentra en 1503, cuando Leonardo da Vinci participó como ingeniero en la guerra contra Prisa.

Independientemente del origen de la investigación operativa, lo que si es un hecho es que necesita abstraer la realidad hacia un modelo matemático, lo que nos permite encontrar factores comunes a los problemas reales, y con ello crear modelos que representen de una forma suficientemente aproximada la realidad. Una de los modelos mas habituales son los modelos de programación lineal, conocidos por cualquiera iniciado en esta ciencia, junto con el símplex, el cual no solo es el algoritmo mas conocido para resolver los problemas de programación lineal, si no uno de los algoritmos mas importantes dentro de las matemáticas. Sin embargo los modelos de programación lineal tienen el inconveniente de que aún que son sencillos de plantear su coste computacional termina siendo demasiado elevado cuando el número de variables crece.

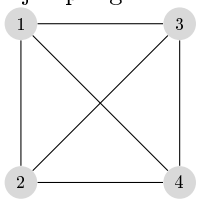
Debido al gran número de variables con las que se trabaja hoy en día los problemas de programación lineal pueden no ser la mejor opción, por eso, se intenta proponer modelos matemáticos que alcancen la solución del problema en un tiempo polinómico. Esto suena muy bien, pero alcanzar la solución exacta de un problema en un tiempo polinómico no siempre es posible (problemas NP-duros), por lo que en algunos casos sacrificaremos precisión para ganar tiempo, aplicando en estos casos algoritmos heurísticos, los cuales nos aportaran una solución razonablemente buena en un tiempo razonablemente corto.

Capítulo 1

Conceptos previos.

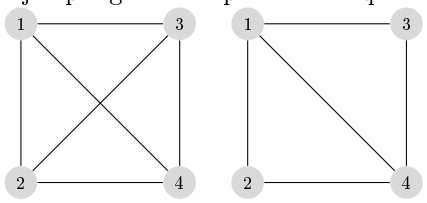
Definición 1. Una red o grafo es un par $G = (V, A)$, donde V el conjunto finito de vértices o nodos y $A \subset V \times V$ es un conjunto de pares ordenado de V, conocido como arcos. El conjunto $A \subset V \times V = \{(i, j)i, j \in V\}$ se corresponde con el conjunto de todos los arcos que podrían formar a partir del conjunto de nodos V.

Ejemplo grafo:



Definición 2. Un grafo es completo si para cada par de nodos hay un arco que los une. De esta forma, el grafo $G = (V, V \times V)$ es un grafo completo.

Ejemplo grafo completo a la izquierda y grafo no completo a la derecha:



Definición 3. Un arco dirigido o arco orientado, es un arco con una dirección determinada (gráficamente lo representaremos con una flecha).De esta forma, en un arco dirigido (i, j) el nodo i es el nodo origen y el nodo j es el nodo fin. Durante este trabajo cuando digamos solo arco nos referiremos a arco no dirigido.

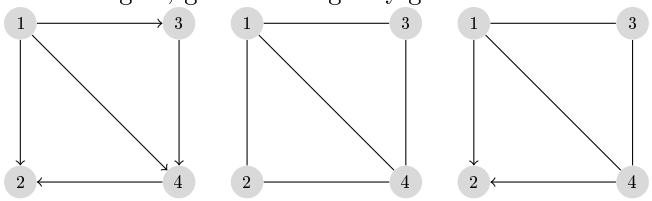
Ejemplo arco dirigido con origen en el nodo 1 y fin en el nodo 2:

Definición 4. Una arista o arco no dirigido es un arco tal que $(i, j) = (j, i)$.

Representaciones equivalentes de aristas entre los nodos 1 y 2:

Definición 5. Un grafo dirigido (también conocido como grafo orientado o digrafo) es un grafo en el que todos los arcos son arcos dirigidos. Si todos los arcos son aristas, hablaremos de un grafo no dirigido. Un grafo mixto es un grafo que puede contener tanto arcos dirigidos como no dirigidos. Los grafos dirigidos y no dirigidos son casos particulares de los grafos mixtos.

Grafo dirigido, grafo no dirigido y grafo mixto:



Definición 6. Un grafo ponderado (o grafo con pesos) es un grafo en el que cada arco tiene un peso asociado. Los pesos pueden representar costes, tiempos, capacidades, beneficios, etc (esto se utiliza mucho en problemas de optimización tales como el problema del árbol de coste minimal, problema del camino mas corto, problemas de flujo, planificación de proyectos, problemas de rutas, etc)

Definición 7. En un gafo no dirigido el grado de un nodo es el número de aristas incidentes en él. Un nodo es par si su grado es par e impar si, por el contrario , su grado es impar.

Un grafo no dirigido es par si todos sus nodos son de grado par

Definición 8. En un grafo dirigido el grado de entrada de un nodo viene dado por el número de arcos en los que ese nodo es el nodo fin, mientras que el grado de salida de dicho nodo es el número de arcos en los que este nodo es el nodo origen. Un nodo es simétrico si tiene el mismo grado de entrada que de salida.

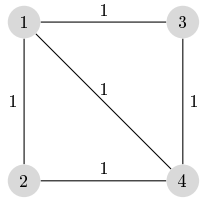
Un grafo no dirigido es simétrico si todos sus nodos son simétricos

Definición 9. E un grafo mixto, el grado de un nodo es el número de arcos, tanto dirigidos como no dirigidos, incidentes en él.

Un grafo mixto es par si todos sus nodos son pares.

Un grafo mixto es simétrico si todos sus nodos son simétricos, es decir. si coincide el número de arcos de los que ese nodo es el nodo fin con el número de arcos de los que ese nodo es el nodo origen.

Definición 10. Un grafo planar es aquel en el que el grado de todos los nodos es menor o igual a tres y todos los costes son iguales entre si.

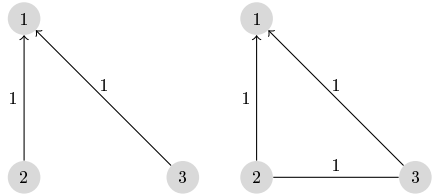


Definición 11. Un arco es incidente en un nodo si dicho nodo esta conectado con otro mediante ese arco.

Definición 12. Dos arcos denominan adyacentes si tienen un nodo en común.

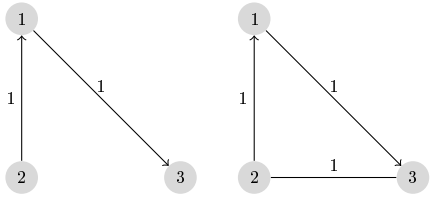
Definición 13. Una cadena entre dos nodos es una secuencia de arcos entre ellos, cumpliendo que cada uno es adyacente con el anterior. Un circuito es una cadena cerrada.

Cadena y circuito:



Definición 14. Un camino entre dos nodos es una cadena en la que todos los arcos tienen la misma orientación, es decir, el nodo final de un arco es el nodo origen del siguiente. Un ciclo es un camino cerrado.

Camino y ciclo:



Definición 15. Un tour es un ciclo que atraviesa cada arco al menos una vez.

Un tour euleriano es un tour que atraviesa cada arco exactamente una vez.

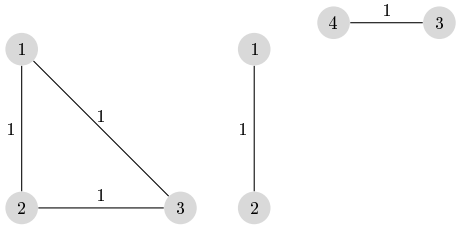
Un grafo es euleriano si contiene un tour euleriano. Esto significa que un grafo es euleriano si es posible recorrerlo sin pasar dos veces por el mismo arco, empezando y terminando el recorrido en el mismo nodo.

Un grafo es semi-euleriano si admite una camino que contiene todas sus aristas una única vez.

Definición 16. El coste de un camino, ciclo o tour es la suma de los costes de los arcos dirigidos y los no dirigidos que atraviesa.

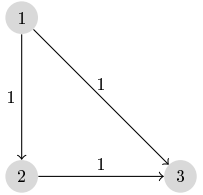
Definición 17. Un grafo no dirigido es conexo si cualquier par de nodos está conectado por un camino, es decir, para cualquier par de nodos i y j existe un camino de i hasta j.

Grafo no dirigido conexo y grafo no dirigido no conexo:



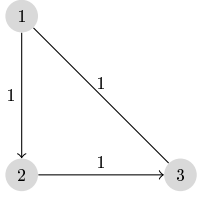
Definición 18. Un grafo dirigido o mixto es débilmente conexo si al considerar los arcos dirigidos como no dirigidos el grafo no dirigido subyacente es conexo.

Grafo débilmente conexo:



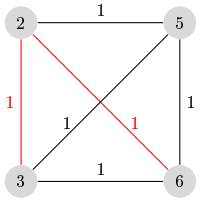
Definición 19. Un grafo dirigido o mixto es fuertemente conexo si para cualquier par de nodos i y j existe un camino de i a j y un camino de j a i.

Grafo fuertemente conexo:



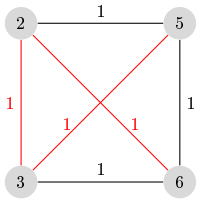
Definición 20. Un árbol sobre el grafo no dirigido $G = (V, A)$ es una colección de aristas, de modo que forman un subgrafo conexo sin ciclos.

Árbol representado en rojo:



Definición 21. Un árbol de expansión sobre el grafo no dirigido $G = (V, A)$ es una colección de aristas, de modo que forman un árbol y cubren todos los vértices. Un árbol de expansión mínima sobre G es aquel que tiene un coste menor o igual que cualquier otro árbol de expansión sobre G.

Árbol de expansión representado en rojo:



Capítulo 2

Problemas previos.

2.1. Problema del camino mas corto.

Sea $G = (V, E)$ un grafo, y sean $a, b \in V$, el problema del camino de coste mínimo que empieza en a y termina en b. El nombre viene dado porque el planteamiento clásico busca el camino que parte de a y termina en b recorriendo la menor distancia, y el nombre se mantiene aún que en muchos casos los costes de los arcos y/o aristas no sean distancias. Podemos plantear el problema como un problema de programación lineal, siendo c_{ij} el coste de la arista o arco que va de i a j, y x_{ij} el número de veces que recorremos ese arco o arista. Este problema se puede expresar como un problema de flujo de coste mínimo entre a y b, teniendo a una oferta de 1 y b una demanda de 1. Este problema podemos plantearlo como un problema de programación lineal, siendo este planteamiento el siguiente:

$$\text{Minimizar} \sum_{i,j \in V} c_{ij} x_{ij}$$

$$\text{Sujeto a:} \sum_j x_{ij} - \sum_l x_{li} = 1 \quad \text{si} \quad i = a$$

$$\sum_j x_{ij} - \sum_l x_{li} = 0 \quad \text{si} \quad i \neq a \quad \text{y} \quad i \neq b$$

$$\sum_j x_{ij} - \sum_l x_{li} = -1 \quad \text{si} \quad i = b$$

$$x_{ij} \geq 0 \text{ y } x_{ij} \text{ natural}$$

Se trata de un problema de fácil resolución, pero de gran importancia dado que aparece en muchísimos otros problemas. Entre los algoritmos con coste polinómico mas conocidos tenemos:

- Dijstra: Se usa tanto en redes dirigidas como no dirigidas con distancias no negativas. Determina la ruta más corta entre un nodo origen y cada uno de los otros nodos en la red.
- Floyd: A diferencia del algoritmos de Dijstra se puede utilizar en el caso de distancias negativas siempre y cuando no haya ciclos de distancia total negativa. Determina la ruta más corta entre dos nodos cualesquiera de la red.
- Bellman-Ford: Al igual que el algoritmo de Floyd se puede usar con costes negativos si no tenemos ciclos de coste negativo. La diferencia es que es mas lento, pero si hay ciclos de coste negativo los localiza.

Según las circunstancias puede ser mas interesante aplicar uno o otro, como en nuestro caso el mas adecuado es el de Floyd, ya que, aún que no tendremos distancias negativas, nos beneficiará tener la distancia del camino mas corto entre todos los nodos, y por ello será el que explicaremos a continuación. La idea tras el algoritmos

es que si tenemos tres nodos a, b y c para que sea mejor ir de a a c pasando a través de b es necesario que se cumpla:

$$c_{ab} + c_{bc} < c_{ac}.$$

Cuando se cumpla esta desigualdad será mejor remplazar la ruta directa de a a c por otra pasando por b. El algoritmo se aplicaría en los siguientes $n+1$ pasos (donde n es el número de nodos):

PASO 0: Definimos una matriz de distancias C_0 con n filas y columnas. Donde confluye la fila i con la columna j pondremos la distancia c_{ij}. Además también se define la matriz S_0 de secuencia del nodo, inicialmente consideramos $s_{ij} = j$ para cualquier par de nodos i, j. Esto indica que del agente i al agente j se llega pasando por j.

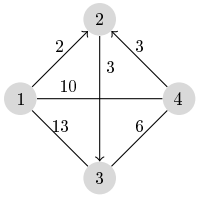
PASO K-esimo: Consideramos la fila k y la columna k como fila y columna pivote. Para cada elemento c_{ij} de la matriz C_{k-1} de la etapa anterior si se satisface que $c_{ik} + c_{kj} < c_{ij}$ se realizan los siguientes cambios:

- Definimos C_k reemplazando c_{ij} por $c_{ik} + c_{kj}$.
- Definimos S_k reemplazando s_{ij} por k.

Todos estos pasos los podemos resumir en el siguiente pseudocódigo:

```
hacer k=1 hasta n
    hacer i=1 hasta n
        hacer j=1 hasta n
            C(i,j)=min(c(i,j),c(i,k)+c(k,j))
            si c(i,j)>c(i,k)+c(k,j)\{
            S(i,j)=k
```

Veamos un breve ejemplo en el que aplicaremos el algoritmo de Floyd al siguiente grafo de 4 nodos:



En este caso C_0 sería:

C_0	1	2	3	4
1	-	2	13	10
2	∞	-	3	∞
3	13	∞	-	6
4	10	3	6	-

Y S_0:

S_0	1	2	3	4
1	-	2	3	4
2	1	-	3	4
3	1	2	-	4
4	1	2	3	-

Etapa 1, elijemos los pivotes y vemos en que elementos hay que hacer cambios:

C_0	1	2	3	4
1	-	2	13	10
2	∞	-	3	∞
3	13	$\boxed{\infty}$	-	6
4	10	3	6	-

S_0	1	2	3	4
1	-	2	3	4
2	1	-	3	4
3	1	$\boxed{2}$	-	4
4	1	2	3	-

Calculamos C_1 y S_1:

C_1	1	2	3	4
1	-	2	13	10
2	∞	-	3	∞
3	13	15	-	6
4	10	3	6	-

S_1	1	2	3	4
1	-	2	3	4
2	1	-	3	4
3	1	1	-	4
4	1	2	3	-

Etapa 2:

C_1	1	2	3	4
1	-	2	[13]	10
2	∞	-	3	∞
3	13	15	-	6
4	10	3	6	-

S_1	1	2	3	4
1	-	2	[3]	4
2	1	-	3	4
3	1	1	-	4
4	1	2	3	-

Calculamos C_2 e S_2:

C_2	1	2	3	4
1	-	2	5	10
2	∞	-	3	∞
3	13	15	-	6
4	10	3	6	-

S_2	1	2	3	4
1	-	2	2	4
2	1	-	3	4
3	1	1	-	4
4	1	2	3	-

Etapa 3:

C_2	1	2	3	4
1	-	2	5	10
2	[∞]	-	3	[∞]
3	13	15	-	6
4	10	3	6	-

S_2	1	2	3	4
1	-	2	2	4
2	[1]	-	3	[4]
3	1	1	-	4
4	1	2	3	-

Calculamos C_3 y S_3:

C_3	1	2	3	4
1	-	2	5	10
2	16	-	3	9
3	13	15	-	6
4	10	3	6	-

S_3	1	2	3	4
1	-	2	2	4
2	3	-	3	3
3	1	1	-	4
4	1	2	3	-

Etapa 4:

C_3	1	2	3	4
1	-	2	5	10
2	16	-	3	9
3	13	[15]	-	6
4	10	3	6	-

S_3	1	2	3	4
1	-	2	2	4
2	3	-	3	3
3	1	[1]	-	4
4	1	2	3	-

Calculamos C_4 y S_4:

C_4	1	2	3	4
1	-	2	5	10
2	16	-	3	9
3	13	9	-	6
4	10	3	6	-

S_4	1	2	3	4
1	-	2	2	4
2	3	-	3	3
3	1	4	-	4
4	1	2	3	-

Con lo que ya damos por terminado el algoritmo. Con estas tablas podemos calcular el camino mas corto entre cualquier par de nodos. Por ejemplo, el camino mas corto desde el nodo 1 al 2 tiene un coste 2 unidades (tal como vemos en la tabla C_4) y es el arco $(1, 2)$, dado que como vemos en la tabla S_4 para ir de 1 a 2 tenemos que pasar por 2, o lo que es lo mismo, vamos directamente de 1 a 2. Sin embargo para ir desde el nodo 2 al 1, tal como vemos en S_4, tenemos que pasar por el nodo 3, y para ir del nodo 3 al nodo 1 tenemos que pasar por 3, por lo que el camino mas corto entre el nodo 2 y el 1 es $M = \{(2, 3), (3, 1)\}$ con un coste de 16 unidades (como nos indica S_4).

2.2. Problema del árbol de expansión mínima.

Tal como indica el nombre este problema consiste en encontrar el árbol de expansión mínima del grafo G. Una de las formas de plantear este problema es dando un nodo fuente y queriendo conectar todos los demás directa o indirectamente a el con el menor coste posible. Sobra decir que este problema solo tiene solución si se trata de un grafo conexo. Se trata de un problema con multitud de aplicaciones, tanto por las aplicaciones practicas que tiene, como por ser un subproblema de otros problemas. Podemos plantear el problema como un problema de programación lineal, siendo c_k el coste de la arista k, y x_k la variable que nos indica si usamos o no esa arista (toma el valor 1 si sí la usamos y 0 si no la usamos). Denotemos por $M(X)$ el conjunto de aristas incidentes en algún nodo de X. El planteamiento sería el siguiente:

$$\text{Minimizar} \sum_{i,j \in V} c_k x_k$$

$$\text{Sujeto a:} \sum x_k = |V| - 1$$

$$\sum_{k \in M(X)} x_k = |M| - 1 \quad \forall X \subset V$$

$$x_{ij} \geq 0 \text{ binaria}$$

Entre los algoritmos con coste polinómico para resolver este problema destacan el algoritmo de Prim, el algoritmo de Kruskal, el algoritmo reverse-delete o el algoritmo de Boruvka. Nosotros veremos solo el de Prim, por lo fácil que es de implementar en un ordenador. Los pasos a seguir para el algoritmo de Prim son los siguientes:

Paso 1: Definimos el nodo de partida i (Que será nuestro nodo fuente) y el conjunto $X = \{i\}$, y comenzamos con nuestro árbol sin ninguna arista $T = \emptyset$.

Paso 2: Buscamos el nodo de $V \backslash X$ mas cercano a X, lo denotemos por j.

Paso 3: Añadimos la arista mas corta que une j con X a T y añadimos j a X.

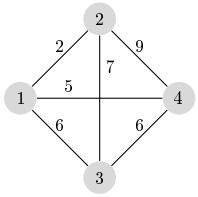
Paso 4: Si $X = V$ damos por terminado el algoritmo, en caso contrario volvemos al paso 2.

Todo esto se puede resumir en el siguiente pseudocódigo:

```
X={1}
T =NULL

hacer i=2 hasta n
    K=mas_proximo(X) !K es el nodo en V/X mas próximo a los nodos de X
    c=mas_corta(X,K) !c es la arista de menor coste que una X y k
    X=(X,K)
    T=(T,c)
```

Veamos un ejemplo en el que calculamos el árbol de expansión mínimo sobre el siguiente ejemplo:

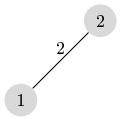


Con la siguiente matriz de costes:

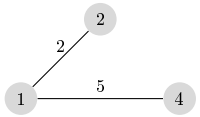
costes	1	2	3	4
1	-	2	6	5
2	2	-	7	9
3	6	7	-	6
4	5	9	6	-

Empecemos con el paso 1.Elegimos un nodo fuente arbitrariamente, por ejemplo el nodo 1. Comenzamos con $X = \{1\}$ y $T = \emptyset$.

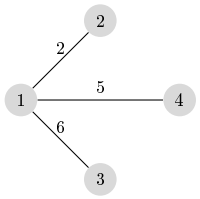
Pasamos al paso 2, y vemos que el nodo de $V \backslash X$ mas cercano a X es el $j = 2$ mediante la arista $(1, 2)$. En el paso 3 denotamos $X = \{1, 2\}$ y añadimos la arista al árbol obteniendo $T = \{(1, 2)\}$. Por lo que nuestro árbol sería:



Vemos que aún no tenemos $X = V$, por lo que repetimos el proceso. Ahora tenemos que el nodo de $V \backslash X$ mas cercano a X es el $j = 4$ mediante la arista $(1, 4)$, por lo que obtenemos $X = \{1, 2, 4\}$ y $T = \{(1, 2), (1, 4)\}$, y nuestro árbol quedaría:



Nuevamente aún no tenemos $X = V$, por lo que repetimos el proceso. Ahora tenemos que el nodo de $V \backslash X$ mas cercano a X es el $j = 3$, con dos posibles opciones, la arista $(1, 3)$ y la arista $(3, 4)$, ambas con coste 6. Nosotros elegimos la arista $(1, 3)$ por lo que obtenemos $X = \{1, 2, 3, 4\}$ y $T = \{(1, 2), (1, 4), (1, 3)\}$, y nuestro árbol quedaría



Podemos ver que ahora $X = V$, por lo que damos por terminado el algoritmo, siendo nuestro árbol de expansión mínima $T = \{(1, 2), (1, 4), (1, 3)\}$.

2.3. El problema del transporte.

El planteamiento clásico de este problema consiste en llevar un determinado número de unidades de un producto de un conjunto de almacenes (nodos oferta) a un conjunto de tiendas (nodos demanda). Cada almacén tiene una determinada cantidad de producto y cada tienda demanda una cantidad fija. El coste de trasladar una unidad del producto de cada almacén a cada tienda es conocido. Nuestro objetivo es satisfacer la demanda con el mínimo coste. Existen dos variantes del problema, que sea equilibrado (la demanda es la misma que la oferta) o que no lo sea. Nosotros solo veremos como resolver el caso equilibrado, dado que es el único que necesitaremos en este proyecto, aún que en el caso de que el problema no fuera equilibrado podríamos añadir un nodo oferta o un nodo demanda ficticio para hacerlo equilibrado. También existe una versión mas completa, que nosotros no veremos dado que no necesitamos para este proyecto, que se da cuando en el problema transporte cabe la posibilidad de enviar el producto a través de nodos intermedios antes de llegar al punto de destino, y es conocido como el problema del transbordo.

Para el problema del transporte nuestras variables de decisión serían x_{ij}, las cuales denotan la cantidad de unidades que enviaremos del nodo oferta i-ésimo al nodo demanda j-ésimo. Además usaremos los parámetros c_{ij}, el cual indica el coste de enviar una unidad de producto del nodo oferta i-ésimo al nodo demanda j-ésimo, a_i, que indica la oferta del nodo oferta i-ésimo, y b_i, que indica la demanda del nodo demanda i-ésimo. Por otro lado n denotará en número de nodos oferta y m en número de nodos demanda. Veamos como sería el problema de programación lineal asociado:

$$\text{Minimizar} \sum_{i=1}^{n} \sum_{j=1}^{m} c_{ij} x_{ij}$$

$$\text{Sujeto a: } \sum_{j=1}^{m} x_{ij} = a_i \ \forall \ i \in \{1, \ldots, n\}$$

$$\sum_{i=1}^{n} x_{ij} = b_j \ \forall \ j \in \{1, \ldots, m\}$$

$$x_{ij} \geq 0 \text{ Entera}$$

Como planteamiento es correcto, pero en el caso de tener muchos nodos oferta y demanda se volvería muy costoso, por eso nosotros usaremos una versión modificada del simplex. Para ello necesitaremos partir de una solución factible y en sucesivas iteraciones realizaremos cambios en la elección de las variables básicas que darán lugar a mejoras en la función objetivo hasta que lleguemos a la solución óptima. Para que exista esta solución factible y solución óptima necesitaremos que el problema sea equilibrado, como será siempre nuestro caso, y por lo tanto podremos aplicar nuestro algoritmo y tendremos asegurada la existencia de solución óptima.

Como ya vimos, los únicos datos que necesitamos son las ofertas y demandas de cada nodo, y los costes para enviar una unidad del nodo oferta i al nodo demanda j. Para representarlos nos valdremos de una tabla como la siguiente:

Desde-hacia	D_1	D_2	$\ldots$	D_n	
S_1	c_{11}	c_{12}	$\ldots$	c_{1n}	a_1
S_2	c_{21}	c_{22}	$\ldots$	c_{2n}	a_2
$\ldots$	$\ldots$	$\ldots$	$\ldots$	$\ldots$	$\ldots$
S_n	c_{n1}	c_{n2}	$\ldots$	c_{nn}	a_n
	b_1	b_2	$\ldots$	b_n	

Para calcular la solución factible disponemos de multitud de algoritmos, como pueden ser el algoritmo de la esquina noroeste, el algoritmo del menor coste o el algoritmo de la aproximación de Vogel. Cualquiera de ellos nos puede ser útil, veamos como aplicar el algoritmo de la esquina noroeste, que aún que no nos dará la solución factible de menor corte si es uno de los mas fáciles de aplicar. Para aplicar el algoritmo lo único que haremos será asignar los productos de manera factible empezando por la esquina superior izquierda la (esquina noroeste) de la matriz. tal como vemos en el siguiente ejemplo:

Desde-hacia	D_1	D_2	D_3	D_4	
S_1	8	6	10	9	35
S_2	9	12	13	7	50
S_3	14	9	16	5	40
	45	20	30	30	

Comenzamos asignando toda la oferta del nodo S_1, para ello la vamos asignando por orden a los nodos demanda, por lo que asignamos 35 unidades a D_1, asignando así toda la oferta de S_1 y aún nos quedarán 10 unidades de demanda en D_1. Acto seguido asignamos la oferta del nodo S_2, como D_1 todavía demanda 10 unidades empezaremos asignándole 10 unidades a D_1, como todavía nos quedan 40 unidades por asignar, le asignaremos 20 unidades a D_2 (la demanda de D_2) y las 20 unidades restantes a D_3, quedándonos todavía 10 unidades de demanda en D_3. Por último asignaremos la oferta de S_3, empezaremos asignando 10 unidades a D_3 y las 30 unidades restantes a D_4.

Desde-hacia	D_1	D_2	D_3	D_4	
S_1	8	6	10	9	
	35				35
S_2	9	12	13	7	
	10	**20**	**20**		50
S_3	14	9	16	5	
			10	**30**	40
	45	20	30	30	

Con lo que el coste de esta asignación es 1180, que no tiene porque ser una solución óptima, solo es una solución factible.

Una vez que tengamos la solución factible, calculada con este o con cualquier otro algoritmo, pasamos a calcular la solución óptima. Para ello asignaremos un valor u_i a cada fila y un valor v_j a cada columna de modo que para las variables básicas se cumpla que $c_{ij} - (u_i + v_j) = 0$. A partir de estos valores (u, v) calcularemos $c_{ij} - (u_i + v_j)$ para cada variable no básica. $c_{ij} - (u_i + v_j)$ nos indicará el cambio que se produciría en el coste si introducimos la variable x_{ij} como variable básica. Una vez calculados los valores $c_{ij} - (u_i + v_j)$ para todas las variables tendremos dos posibles opciones:

- Todos los $c_{ij} - (u_i + v_j)$ son mayores o iguales que cero, o lo que es lo mismo, no existe ninguna variable no básica que mejore la solución, y por lo tanto nuestra solución es la óptima.

- Existe algún $c_{ij} - (u_i + v_j)$ negativo, y por lo tanto podemos mejorar la solución factible incluyendo alguna de dichas variables x_{ij} como variable básica.

En caso de poder mejorar la solución factible tendremos que elegir que variable x_{ij} introducimos como variable básica, para ello lo mas lógico será tomar la que aporte una mayor mejora a nuestra solución, o lo que es lo mismo, la que tenga un menor $c_{ij} - (u_i + v_j)$. Por cada unidad que añadamos de x_{ij} el coste variará $c_{ij} - (u_i + v_j)$ unidades.

Para poder incorporar la variable x_{ij} a la base necesitamos encontrar un loop que involucre varias variables básicas. Recordemos que un loop es una secuencia de al menos cuatro celdas que verifica:

- Dos celdas consecutivas están en la misma fila o columna.

- No hay mas de dos celdas consecutivas en la misma fila o columna.

- La última celda tiene la fila o la columna en común con la primera.

La idea es asignar el máximo valor a la variable x_{ij} sin salirnos de la factibilidad, por eso construimos el loop, así podemos añadir unidades a la variable x_{ij} simplemente moviendo unidades de una variable a otra. Veamos como sobre el ejemplo anterior:

Desde-hacia	D_1	D_2	D_3	D_4	
S_1	8	6	10	9	
	35				35
S_2	9	12	13	7	
	10	**20**	**20**		50
S_3	14	9	16	5	
			10	**30**	40
	45	20	30	30	

Comencemos definiendo $u_1 = 0$, y como sabemos que en las variables básicas $c_{ij} = u_i + v_j$ calcularemos los demás u_i y v_j, obteniendo:

Desde-hacia	D_1		D_2		D_3		D_4		
S_1		8		6		10		9	$u_1 = 0$
	35								35
S_2		9		12		13		7	$u_2 = 1$
	10		**20**		**20**				50
S_3		14		9		16		5	$u_3 = 4$
					10		**30**		40
	45	$v_1 = 8$	20	$v_2 = 11$	30	$v_3 = 12$	30	$v_4 = 1$	

Ahora calculamos $c_{ij} - u_i - v_j$ para las variables no básicas, obteniendo:

Desde-hacia	D_1		D_2		D_3		D_4		
S_1		8		6		10		9	$u_1 = 0$
	35			-5		-2		8	35
S_2		9		12		13		7	$u_2 = 1$
	10		**20**		**20**			5	50
S_3		14		9		16		5	$u_3 = 4$
		2		-6	**10**		**30**		40
	45	$v_1 = 8$	20	$v_2 = 11$	30	$v_3 = 12$	30	$v_4 = 1$	

Si alguno de los $c_{ij} - u_i - v_j$ asociados a las variables no básicas (marcados en azul en la tabla) es negativo, hay posibilidades de mejorar la función objetivo, o lo que es lo mismo, disminuir el coste del transporte. En este caso tenemos varios $c_{ij} - u_i - v_j$ negativos, por lo que escogeremos el menor de ellos (que es el que nos aportará una mayor mejoría). Por lo que en este caso nos interesará meter la variable x_{32} en la base, asignándole una cantidad que enviaremos del nodo oferta 3 al nodo demanda 2. Construiremos un loop en la variable x_{32}, que representaremos en rojo en la tabla:

Desde-hacia	D_1		D_2		D_3		D_4		
S_1		8		6		10		9	$u_1 = 0$
	35			-5		-2		8	35
S_2		9		12		13		7	$u_2 = 1$
	10		**20**	•	**20**	•		5	50
S_3		14		9		16		5	$u_3 = 4$
		2		-6	**10**	•	**30**		40
	45	$v_1 = 8$	20	$v_2 = 11$	30	$v_3 = 12$	30	$v_4 = 1$	

nos interesa asignarle el mayor valor a la celda x_{32} sin salirnos de la factibilidad, para lo que le asignaremos el menor valor de las variables básicas involucradas en el loop en las que disminuiremos el valor de las unidades transportadas (en este caso 10 unidades). Hacemos dicho cambio en las variables básicas y volvemos a calcular los u_i y los v_j, así como los nuevos coeficientes $c_{ij} - u_i - v_j$ para las variables no básicas. Una vez hecho esto repetimos el paso anterior, con lo que obtenemos:

Desde-hacia	D_1		D_2		D_3		D_4		
S_1		8		6		10		9	$u_1 = 0$
	35	•		-5		-2		2	35
S_2		9		12		13		7	$u_2 = 1$
	10	•	**10**	•	**30**			-1	50
S_3		14		9		16		5	$u_3 = -2$
		8	**10**			6	**30**		40
	45	$v_1 = 8$	20	$v_2 = 11$	30	$v_3 = 12$	30	$v_4 = 7$	

Vemos que seguimos teniendo $c_{ij} - u_i - v_j$ negativos, por lo que escogemos el menor de ellos y repetimos el paso anterior. En este caso queremos meter x_{12} en la base, y la máxima cantidad que podemos asignarle sin salirnos de la factibilidad es 10 (el mínimo de las cantidades asignadas a las variables básicas del loop a las que les disminuiremos sus cantidades asociadas). Realizamos los cambios en x_{12} y volvemos a calcular los valores de u_i y v_j, así como los $c_{ij} - u_i - v_j$ asociados a las variables no básicas. Con lo que obtenemos:

Desde-hacia	D_1		D_2		D_3		D_4		
S_1		8		6		10		9	$u_1 = 0$
	25	•	**10**			-2		7	35
S_2		9		12		13		7	$u_2 = 1$
	20	•		5	**30**	•		4	50
S_3		14		9		16		5	$u_3 = 3$
		3	**10**			1	**30**		40
	45	$v_1 = 8$	20	$v_2 = 6$	30	$v_3 = 12$	30	$v_4 = 2$	

Nuevamente vemos que tenemos $c_{ij} - u_i - v_j$ negativos, por lo que tenemos que repetir el paso anterior. Ahora vamos a introducir x_{13} en la base. Consideramos el menor de los valores de las celdas básicas involucradas en el loop en las que disminuyen el número de unidades transportadas, obteniendo 25. Hacemos dicho cambio en las variables básicas y recalculamos los coeficientes, con lo que obtenemos:

Desde-hacia	D_1		D_2		D_3		D_4		
S_1		8		6		10		9	$u_1 = 0$
		2	**10**		**25**			7	35
S_2		9		12		13		7	$u_2 = 3$
	45			3	**5**			2	50
S_3		14		9		16		5	$u_3 = 3$
		5	**10**			3	**30**		40
	45	$v_1 = 6$	20	$v_2 = 6$	30	$v_3 = 10$	30	$v_4 = 2$	

Vemos que no tenemos ningún $c_{ij} - u_i - v_j$ negativo, por lo que es imposible mejorar esta solución (debemos tener en cuenta que en caso de tener algún $c_{ij} - u_i - v_j = 0$ esta no sería la única solución óptima).Si nos interesa saber el coste de esta solución tenemos dos opciones, una es calcularlo con la función objetivo, dado que sabemos cuanto mandamos desde cada nodo oferta a cada nodo demanda, en este caso el coste que obtenemos es 1020 unidades. La otra opción es partir del coste de la solución factible que obtuvimos por la regla de la esquina noroeste y ver cuanto mejoramos en cada paso, dado que en cada paso mejoramos $c_{ij} - u_i - v_j$ unidades por cada unidad que añadimos a x_{ij} cuando la metemos en la base. En nuestro caso partimos de un coste de 1180, en el primer paso $c_{32} - u_3 - v_2 = -6$ y le añadimos 10 unidades a la variable x_{32}, por lo que mejoramos nuestro coste en 60 unidades. En el segundo paso $c_{12} - u_1 - v_2 = -5$ y le añadimos 10 unidades a la variable x_{12}, por lo que mejoramos nuestro coste en 50 unidades. En el tercer paso $c_{13} - u_1 - v_3 = -2$ y le añadimos 25 unidades a la variable x_{13}, por lo que mejoramos nuestro coste en 50 unidades. Por lo tanto mejoramos en 160 unidades el coste de la solución factible inicial, por lo que nuestro nuevo coste es 1180-160=1020.

2.4. Problema del flujo de mínimo coste.

Clásicamente este problema se plantea como una red en la que queremos hacer circular el flujo con menor costo entre los nodos oferta y demanda, de modo que el flujo que sale de un nodo oferta no supere su oferta y el flujo que llega a un nodo demanda igual su demanda. Se trata de un problema muy similar al problema del transporte, de hecho es un caso particular del problema del transbordo en el que los nodos de transbordo tienen capacidad infinita. Antes de intentar resolver el problema debemos tener en cuenta que es necesario que los nodos oferta y demanda se encuentren en un grafo conexo. Para plantear el problema necesitaremos la variable x_{ij}, la cual nos indica el flujo a través de la arista o arco que parte del nodo i y llega al nodo j. a_i, que indica la oferta del nodo oferta i-ésimo, y b_i, que indica la demanda del nodo demanda i-ésimo. Su problema de programación lineal asociado es:

$$\text{Minimizar} \sum_{i=1}^{n} \sum_{i=j}^{m} c_{ij} x_{ij}$$

$$\text{Sujeto a:} \sum_{j=1}^{m} x_{ij} = a_i \ \forall \ i \in \{1, \ldots, n\}$$

$$\sum_{i=1}^{n} x_{ij} = b_j \ \forall \ j \in \{1, \ldots, m\}$$

$$x_{ij} \geq 0 \text{ Entera}$$

2.5. Problema de acoplamiento perfecto.

En un grafo completo el problema de acoplamiento perfecto intenta agrupar los nodos en grupos de dos, de forma que la suma de los costes de los arcos que unen cada pareja de nodos sea lo menor posible (o lo mayor posible). En caso de querer resolver el problema de acoplamiento perfecto de mínimo coste el problema de programación lineal asociado será:

$$\text{Minimizar} \sum_{i=1}^{n} \sum_{j=1}^{m} c_{ij} x_{ij}$$

$$\text{Sujeto a:} \sum_{j=1}^{i-1} x_{ij} + \sum_{j=i+1}^{n} x_{ji} = 1 \ \forall \ i \in \{1, \ldots, n\}$$

$$x_{ij} \in \{0, 1\}$$

Donde c_{ij} denota el coste de la arista que une los nodos i y j, y x_{ij} es la variable que toma el valor 1 si emparejamos los nodos i y j y 0 en caso contrario.

Capítulo 3

Problemas de rutas.

Los problemas de rutas son problemas de Optimización Combinatoria, que consisten en encontrar la solución óptima entre un número finito o infinito numerable de soluciones. Estos problemas se suelen plantear como la búsqueda de la ruta óptima que atraviesa total o parcialmente las aristas y/o arcos dirigidos de un grafo dado. La motivación para la búsqueda de esta ruta suele plantearse como una serie de clientes que demandan un servicio y necesitamos la ruta mas corta para satisfacer dicha demanda. La importancia de estos problemas se debe a la gran cantidad de casos reales en los que se pueden aplicar, tales como repartos de mercancías, recogida de basuras, transporte de pasajeros o la limpieza de las calles; pero no solo en la logística y distribución, si no también en otras situaciones como la producción de circuitos electrónicos integrados o la organización de tareas. Debido a que cada día las empresas trabajan a una mayor escala, cada vez es mas necesaria la aplicación de problemas de rutas, dado que el ahorro se hace cada vez más y más patente. Sin embargo, en casi todos los casos tendremos una dificultad añadida, y es que en la mayoría de los casos no los podremos modelizar como problemas sencillos ya que cada uno de ellos tendrá sus características propias y tendremos que usar una metodología específica adaptada a las características propias del problema.

Además debemos tener cuidado con no confundir los problemas de ruta con los problemas de caminos, en estos últimos solo nos interesa escoger el camino óptimo que une dos nodos, sin embargo en los problemas de rutas nos interesa construír un ciclo (que será nuestra ruta) tal que recorra ciertos nodos y/o arcos. De todas formas estos problemas están relacionados y muchas veces tendremos que resolver un problema de caminos para poder resolver un problema de rutas.

Dentro de los problemas de ruta existen multitud de tipos, en la siguiente tabla veremos los principales, así como los nombres que tradicionalmente se le dio a cada caso:

Demanda	Restricciones de capacidad	nombre del problema
Arcos	No	Una componente conexa, problema del cartero chino (CPP)
		Varias componentes conexas, problema del cartero rural (RPP)
	Si	Problema de los m-carteros(m-CPP)
Nodos	No	Viajante de comercio (TSP)
	Si	Problemas de rutas de vehículos (VRP)

Como vemos en la tabla tenemos dos grandes casos, las rutas por arcos y las rutas por nodos. En los primeros nuestro objetivo es recorrer todos o parte de los arcos, mientras que en los segundos es pasar por todos o parte de los nodos. Estudiar todos los casos de problemas de rutas puede resultar muy extenso, sería mas bien tarea de una tesis que de un proyecto de fin de máster, por ello nosotros nos centraremos solo en los problemas de rutas por arcos, en concreto en el problema del cartero chino, aun que también veremos brevemente la idea general de los problemas de rutas por nodos.

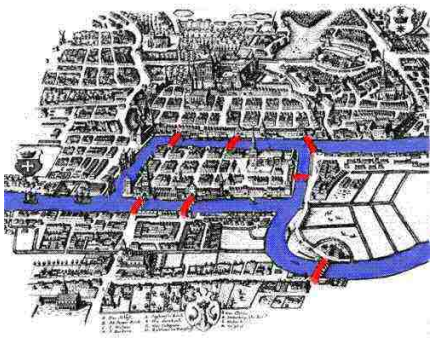


Figura 3.1: Puentes de Königsberg.

La primera referencia que tenemos sobre los problemas de rutas por arcos data de XVIII y es conocido como el problema de los puentes de Königsberg. El problema de los puentes de Königsberg nace de la discusión entre los habitantes de la ciudad sobre la posibilidad de visitar los siete puentes de la ciudad pasando por cada uno una única vez, cuya distribución sobre el rió Pregel se puede ver en la figura 3.1, y su grafo asociado en la figura 3.2. Leonhard Euler fue el primero en plantearlo matemáticamente, y posteriormente en 1736 demostró en su publicación "Solutio problematis ad geometriam situs pertonentis" que, como los propios habitantes de la ciudad suponían, esto no era posible. Para ello Euler demostró que si hay dos nodos de grado impar en un grafo, es posible encontrar un camino que atraviese todos los puentes exactamente una vez, empezando en uno de esos nodos y terminando en el otro. Si no hay nodos de grado impar, tal camino existe partiendo desde cualquier nodo y terminando en el mismo. En honor a Euler los tours que recorren todos los arcos y aristas una sola vez se denominan tours eulerianos y los grafos en los que existen tours eulerianos se denominan grafos eulerianos.

Teorema 3.1. *Un grafo conexo es semi-euleriano si y sólo si no existen más de dos nodos de grado impar, es decir, existen 0 o 2 nodos de grado impar.*

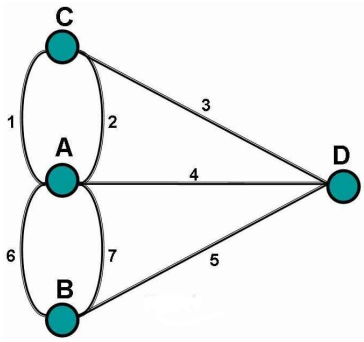


Figura 3.2: Representacion del grafo de los puentes.

Un problema parecido al de los puentes de Königsberg se plantea hoy en día los niños cuando juegan a la firma del diablo, un juego que consiste en realizar un dibujo sin levantar el lápiz del papel, y que fácilmente se puede extrapolar a problemas de rutas por arcos. En la figura 3.3 podemos ver alguno de los dibujos que suelen usar los niños, y que como es obvio son mucho mas sencillos que los que se suelen plantear los investigadores operativos, dado que los de los niños se pueden resolver simplemente por ensayo error.

Sin embargo en el problema de los puentes Königsberg, o en el juego de la firma del diablo, solo se pide la existencia de un camino, no es necesario que sea óptimo, que será una de las cuestiones que trataremos a lo largo

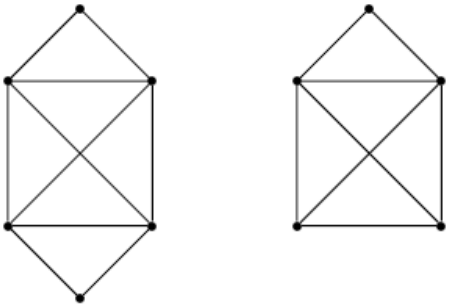


Figura 3.3: Ejemplos la firma del diablo.

de las siguientes secciones, donde analizaremos en detalle las diferentes variantes que nos podemos encontrar en los problemas de rutas por arcos.

Los problemas de rutas por nodos son posteriores a los problemas de rutas por arcos, quizás para encontrar la primera referencia sobre los problemas de rutas por nodos debamos remontarnos a 1857, cuando el irlandés William Rowan Hamilton y el británico Thomas Penynhgton Kirkman inventaron el "Icosian Game", que consistía en recorrer los 20 puntos del tablero y volver al punto de origen. En el fondo el objetivo del juego era encontrar un ciclo que recorra una y solo una vez todos los nodos del grafo formado por las aristas de un icosaedro, de hecho por este juego se denominó este tipo de ciclos como ciclos hamiltonianos. Debemos tener en cuenta que, dado que las aristas no tenían costes asociados, el juego pretendía encontrar un ciclo, no el ciclo óptimo. El juego se comercializó en diferentes formatos, en la figura 3.4 podemos ver uno de ellos, y en la figura 3.5 podemos ver una de las posibles soluciones.

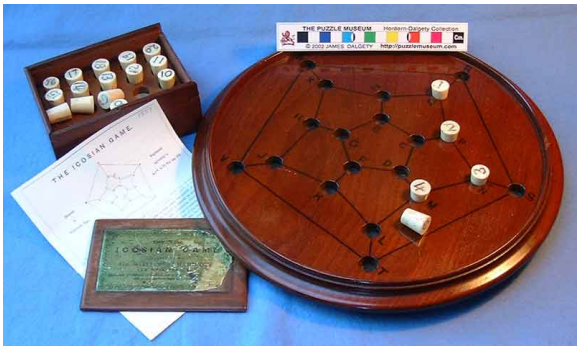


Figura 3.4: Uno de los modelos del Icosian Game.

El TSP es uno de los problemas más estudiados, no solo en el campo de los problemas de rutas por nodos, si no en la investigación operativa en general. El origen de este problema no está del todo claro, sin embargo el colectivo científico parece estar de acuerdo con que surgió entre 1931 y 1932. El problema intenta, partiendo de un nodo origen, recorrer la totalidad (o parte) de los nodos y volver al nodo origen con el mínimo coste. A día de hoy no se ha demostrado que el problema se pueda resolver o no en tiempo polinómico, por lo que pese a que los algoritmos exactos son extremadamente lentos, no podemos afirmar que no exista un algoritmo que lo resuelva en un tiempo razonable. De hecho, el Instituto Clay de Matemáticas ha ofrecido una recompensa de un millón de dólares al que descubra un algoritmo que resuelva el TSP en tiempo polinómico o demuestre la no existencia del mismo.

Los algoritmos exactos que podríamos usar para resolver el TSP serían demasiado lentos, ya que se reducirían o bien a resolver el problema de programación lineal asociado (lo cual con muchos nodos se vuelve tremendamente

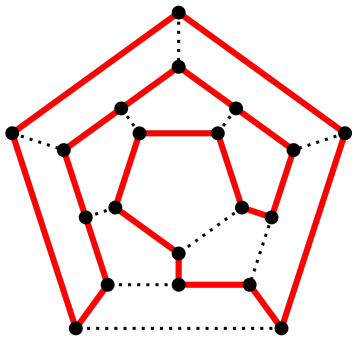


Figura 3.5: Posible solución para el Icosian Game.

costoso por la gran cantidad de variables con la que nos encontraríamos) bien sea por planos de corte o por ramificación y acotación, o calcular todos los posibles ciclos y compararlos, que nuevamente sería imposible de realizar en un tiempo razonable. Sin embargo los realmente interesantes son los algoritmos heurísticos, entre los más conocidos estaría el algoritmo del árbol (con un ratio de error de 2), el algoritmo de Christofides (con un ratio de error de 1.5), o incluso, aún que mucho menos eficiente que los anteriores, el algoritmo del vecino más próximo (solo interesante por su fácil implementación). Por último solo podríamos comentar el algoritmo del intercambio, el cual no calcula el ciclo más corto, si no que mejora ciclos ya existentes, para ello lo único que hace es intercambiar pares de arcos por alternativas de menor coste.

Debemos tener en cuenta que existe una pequeña variante del TSP, el problema Gráfico del Viajante (Graphical Traveling Salesman Problem, GTSP), introducida por Fleischmann (1985, 1988), Cornuèjols, Fonlupt y Naddef (1985) en la que el grafo no tiene por qué ser completo y el vehículo puede tener que visitar más de una vez cada nodo.

El último tipo importante de problemas de rutas por nodos es el VRP. Históricamente el VRP está íntimamente relacionado con el TSP, de echo podemos plantear el VRP como una generalización del TSP. El origen de este problema data de hace más de 50 años, y la primera referencia en la literatura científica fue publicada por Dantzig y Ramser (1959), en su artículo "The Truck Dispatching Problem". El planteamiento es similar al que tenemos en el TSP, la única diferencia es que ahora en lugar de construír un ciclo que recorra todos los nodos tendremos que construír un número determinado de ciclos de forma que entre todos recorran todos los nodos. Este problema es más difícil que el TSP, dado que se hace necesario particionar el conjunto de clientes para que éstos puedan ser atendidos por los vehículos y después determinar el orden de servicio de cada cliente.

Este problema si se ha demostrado que es NP-duro, por lo que dispondremos de diferentes algoritmos heurísticos para cada una de las múltiples variantes del problema, entre las que podemos mencionar restricciones de tiempo o distancia, permitir demandas compartidas, rutas de recogida y reparto o disponer de múltiples depósitos.

Todos los problemas descritos hasta el momento son casos particulares del problema general de rutas (GRP), que es aquel en el que la demanda se da tanto en nodos como en arcos. Este problema fue introducido por Orloff (1974) y tal como demostraron Lenstra y Rinnooy-Kan (1976) es NP-duro. En el trabajo sobre el RPP de Ghiani y Laporte (2000), ya aparecen referencias al GRP, así como en los trabajos de Theis (2005) y Reinelt y Theis (2005, 2008). Sin embargo, también tenemos autores que lo estudiaron extensamente como Corberán y Sanchis (1994, 1998) y Letchford (1997, 1999). Corberán, Letchford y Sanchis (2001) han desarrollado un algoritmo de planos de corte, que hasta el momento ha dado buenos resultados computacionales.

3.1. El problema del cartero chino.

El problema del cartero chino, también conocido como problema del circuito del cartero, el problema de los correos o problema de la inspección y selección de rutas, es el primer problema de rutas por arcos en el que se plantea la posibilidad de construir un ciclo euleriano con coste óptimo. Fue planteado originalmente por el

matemático chino Kwan Mei-Ko (en algunas bibliografías lo vemos escrito como Guan Mei-Ko) en un articulo de un diario chino en 1960 y traducido al ingles en 1962 ("Graphic Programming using odd and even points"). Debido a su autor, Alan Goldman sugirió llamarlo "problema del cartero chino". Lo que Mei-Ko planteaba era el problema al que se enfrenta el cartero para repartir la correspondencia recorriendo la menor distancia posible, que matemáticamente consiste en encontrar un tour en el grafo de longitud mínima, sin embargo, el problema original dio lugar a multitud de variantes, que estudiaremos en los siguientes apartados.

3.1.1. El problema del cartero chino en un grafo no dirigido (CPP).

Este es el problema original planteado por Mei-Ko. Para que este problema tenga solución finita tenemos que trabajar con un grafo conexo con costes asociados a sus aristas no negativos, dado que si no fuera conexo sería imposible construir un tour, y si una arista tuviera un coste negativo, la recorreríamos infinitas veces para minimizar el coste del ciclo. A partir de ahora supondremos que nos encontramos siempre en estas condiciones. Sea $G(V, A)$ un grafo conexo no dirigido con costes no negativos asociados a sus aristas. Nuestro objetivo es encontrar un tour de coste mínimo (que no tiene porque ser único). Antes de ver como resolver este problema veamos unos cuantos teoremas previos.

Teoremas previos.

Teorema 3.2. *Un grafo $G = (V, A)$ conexo y no dirigido, contiene un tour euleriano si y solo si el grafo es par*

Demostración. Supongamos que tenemos un grafo $G = (V, A)$ conexo y no dirigido para el que existe un tour Euleriano, cada vez que este tour "entre" en un nodo por una arista tendrá que "salir" de el por una arista diferente (dado que al ser un tour debemos atravesar cada arista exactamente una vez), por lo que cada nodo tendrá que tener un número de aristas par incidentes en el.

Para demostrar que si el grafo es par entonces contiene un tour euleriano nos bastará con ver el algoritmo de Fleury o el de Hierholzer que estudiaremos mas adelante en está sección, los cuales construyen un tour euleriano en cualquier grafo conexo de grado par. □

Teorema 3.3. *Un grafo conexo $G = (V, A)$ es euleriano si y solo si admite una descomposición en ciclos disjuntos.*

Demostración. Si el grafo $G = (V, A)$ es euleriano contiene un tour euleriano T, con lo que ya tendríamos una descomposición en tours disjuntos, formada por el propio tour T.

Supongamos ahora que tenemos una descomposición C en ciclos disjuntos. Sean $C_1, C_2, \ldots, C_n$ ciclos disjuntos tales que $G = \cup_{i\in\{1\ldots n\}} C_i$. Como vimos en el teorema 3.2 todos los nodos tienen grado par en cada uno de los ciclos, y por lo tanto, dado que son disjuntos, el grado de cualquier nodo será la suma de su grado en cada uno de los ciclos C_i, con lo que concluimos tienen grado par en el grafo. Como todos los nodos son de grado par podemos afirmar que el grafo es euleriano. □

Teorema 3.4. *En un grafo no dirigido $G = (V, A)$, la suma de los grados de todos sus nodos es igual al doble de aristas del grafo o lo que es lo mismo $\sum_{i\in V} d(i) = 2|A|$*

Demostración. Al efectuar la suma de los grados de un grafo G, las aristas se cuentan dos veces, pues cada una de ellas está determinada por dos nodos, de donde se concluye que el sumatorio debe ser par. Veamos esto de un modo un poco mas matemático, para lo que seguiremos un proceso de inducción. Supongamos que tenemos un grafo no dirigido $G = (V, A)$ con $A = \emptyset$, es trivial que se cumple el teorema, dado que el grado de todos sus nodos es 0. Veamos ahora que si a un grafo con n aristas para el que se cumple el teorema, y le añadimos 1 arista más también se cumple el teorema. Sea $G_n = (V', A')$ un grafo con $|A| = n$. Definamos el grafo $G_{n+1} = (V'', A'')$ donde $V'' = V$ y $A'' = A' \cup (j, k)$ siendo j y k dos nodos arbitrarios de V'. Como en G_n se cumple el teorema sabemos que $\sum_{i\in V'} d(i) = 2|A'|$, y es fácil ver que $\sum_{i\in V''} d(i) = \left(\sum_{i\in V''} d(i)\right) + 2 = 2|A'| + 2 = 2|A''|$ □

Teorema 3.5. *En un grafo G no dirigido se verifica que el número de nodos de grado impar siempre es par.*

Demostración. Sean $V_1 = \{i \in V / d(i) \equiv 1 \mod(2)\}$ y $V_2 = \{i \in V / d(i) \equiv 0 \mod(2)\}$. Es trivial que $V_1 \cap V_2 = \emptyset$ y $V_1 \cup V_2 = V$, y por lo tanto:

$$2|A| = \sum_{i \in V} d(i) = \sum_{i \in V_1} d(i) + \sum_{j \in V_2} d(j).$$

Luego por ser $2|A|$ y $\sum_{i \in V_2} d(i)$ ambos pares se tiene que $\sum_{j \in V_1} d(j)$ también es par. Como cada sumando de $\sum_{j \in V_1} d(j)$ es impar nos permite concluir que el número de sumandos debe ser par, por lo que el número de nodos de grado impar es par. □

Una vez visto estos teorema dividiremos este problema en dos posibles casos, que el grafo sea par, en cuyo caso tendremos un tour euleriano, o que no lo sea, y en este caso lo transformaremos en un grafo par duplicando aristas y construiremos un tour euleriano sobre el grafo modificado, lo que nos dará el tour de coste mínimo sobre el grafo original. Cabe destacar que cuando tengamos un tour euleriano, ese tour tendrá el mínimo coste, el cual coincidirá con la suma de los costes de todas las aristas del grafo.

Algoritmos para el CPP de grado par.

Veamos ahora los dos algoritmos mas conocidos para calcular tours eulerianos en el CPP de grado par. Tal como ya hemos comentado, en el caso del CPP de grado par, es lo mismo calcular un tour de mínimo coste que calcular un tour euleriano.

Algoritmo de Fleury.

El primer algoritmo que veremos el algoritmo de Fleury. Se trata de un algoritmo de fácil comprensión, y con un coste $O(|A|)$. Se puede resumir en los siguientes pasos:

paso 1: Partimos de un nodo inicial que denotaremos por v_0, atravesaremos cualquier arista incidente en el, de modo que al eliminarla sigamos teniendo un grafo conexo salvo nodos aislados, y la eliminamos. Al nodo llegada mediante esa arista le llamaremos v_1.

Paso 2: Si aún quedan aristas sin eliminar tomamos el nodo v_1 como nodo inicial (y lo denotamos por v_0, sustituyendo así el anterior nodo inicial) y volvemos al paso 1, en caso contrario damos por terminado el algoritmo, y nuestro tour serán las aristas en el orden en el que las fuimos eliminando.

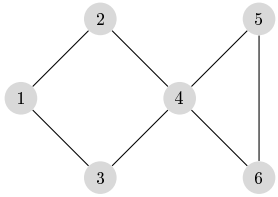
Si en lugar de un tour lo que queremos es determinar un camino que recorra todas las aristas una sola vez, para lo que sería necesario que fuera un grafo semi-euleriano, podemos aplicar exactamente el mismo algoritmo, simplemente empezando en un nodo de grado impar (de haberlo, si no empezaríamos en un nodo cualquiera).

Pseudocódigo de Fleury.

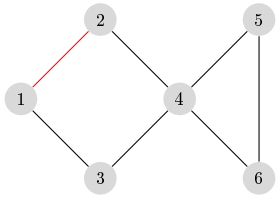
```
ruta=NULL
V="vector conjunto nodo"
n="número de aristas (teniendo en cuenta multiplicidades)"
A="matiz de costes de las aristas"
M="matriz multiplicidades aristas"
nodo=1
hacer i=1 hasta n
    "buscamos k de modo que M[nodo,k] distinto de 0, y al eliminar la arista (nodo,k) el
    grafo sea conexo salvo nodos aislados"
    ruta=(ruta,k)
    nodo=k
    M[nodo,k]=M[nodo,k]-1
    M[k,nodo]=M[k,nodo]-1
```

Ejemplo algoritmo de Fleury.

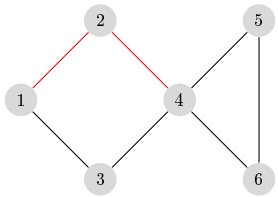
Veamos un ejemplo de como aplicar este algoritmo, para ello definiremos el grafo $G = (V, A)$ de la figura, que podremos resolver en pocos pasos y podremos ilustrar claramente los detalles a los que tendremos que prestar atención.



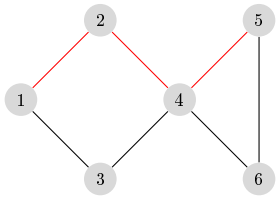
Nótese que no le asociamos un coste a cada arista, dado que al construir un tour euleriano los costes no nos influyen, y se procedería exactamente del mismo modo si tuviéramos costes asociados. Para indicar las aristas que eliminamos, en lugar de borrarlas del grafo las marcaremos en rojo. Lo primero que tenemos que hacer es definir el nodo inicial v_0, en este ejemplo, al ser inventado, no sabemos cual es el punto origen para las rutas, y aunque lo supiéramos podríamos elegir un nodo cualquiera como v_0 dado que el coste del ciclo sería el mismo. En este caso definiremos $v_0 = 1$. Una vez que conocemos el nodo inicial debemos elegir que arista eliminamos, al tener dos aristas incidentes en el nodo 1 podemos eliminar la arista $(1, 2)$ o la arista $(1, 3)$, como en los dos casos el grafo sigue siendo conexo podemos elegir una cualquiera. Nosotros eliminaremos la arista $(1, 2)$, o lo que es lo mismo, tomaremos $v_1 = 2$, y al hacerlo estaremos como en la figura.



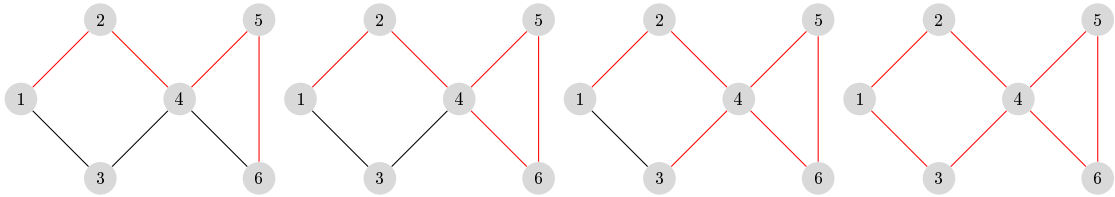
Vemos que quedan aristas por eliminar, por lo tanto definimos $v_0 = 2$, y repetimos el proceso. Ahora solo tenemos una arista incidente en el nodo 2, por lo que eliminamos la arista $(2, 4)$ y tomaremos $v_1 = 4$.



Como todavía no podemos dar por terminado el algoritmo, dado que aún nos quedan aristas por eliminar, denotamos $v_0 = 4$. En este punto ya tenemos que tener cuidado, hay tres aristas que tienen como nodo origen el nodo 4, sin embargo si eliminásemos la arista $(4, 3)$ el grafo resultante dejaría de ser conexo salvo nodos aislados, por lo tanto solo podemos eliminar las aristas $(4, 5)$ o $(4, 6)$. Nosotros decidimos eliminar la arista $(4, 5)$.



A partir de aquí el algoritmo se vuelve trivial, dado que en cada paso solo podremos eliminar una arista, por lo tanto no detallaremos los siguientes pasos, aunque si representaremos gráficamente los pasos que fuimos siguiendo.



Una vez hecho esto ya hemos eliminado todos las aristas, por lo que nuestro tour euleriano serán las aristas en el mismo orden que las eliminamos, o lo que es lo mismo, $C = \{(1, 2), (2, 4), (4, 5), (5, 6), (6, 4), (4, 3), (3, 1)\}$, que como ya comentamos coincide con el tour de mínimo coste que recorre todas las aristas.

Algoritmo de Hierholzer.

El algoritmo de Fleury tiene un inconveniente, pese a que es fácil de entender, y que su coste de $O(|A|)$ sea mas que asumible, es difícil determinar si al eliminar la arista el grafo sigue siendo conexo salvo nodos aislados. Por ello, y basándonos en el teorema 3.3 podemos pensar que el algoritmo de Fleury no es el único algoritmo del que disponemos, de hecho tenemos otra opción, quizás menos visual, pero si mas fácil de programar, y con el mismo coste computacional, por lo que podemos decir que es mas efectivo, es más, el mas efectivo a día de hoy. Se trata del algoritmo de Hierholzer, del que existen pequeñas variaciones propuestas por otros autores. Pese a que la definición original no tiene la estructura a la que estamos acostumbrados, podríamos resumirlo como sigue:

Paso 1: Empezando por el nodo de inicio v_0 construimos un ciclo C_1 atravesando las aristas adyacentes al mismo tiempo que las eliminamos, hasta que el ciclo termine (en el v). Si con esto conseguimos un ciclo euleriano damos por terminado el algoritmo, y nuestro ciclo será $C = C_1$.

Paso 2: Tomamos como nodo inicial cualquier nodo v que esté en el ciclo C_1 y todavía tenga aristas incidentes sin eliminar, construimos un ciclo C_2 que empiece en dicho nodo.

Paso 3: Fusionamos los dos ciclos, y el ciclo resultante lo denotamos como C_1. Para fusionar los ciclos lo único que hacemos es tomar dos aristas $a_1, a_2 \in C_1$ incidentes en el nodo v, y colocamos el ciclo C_2 ente dichas aristas.

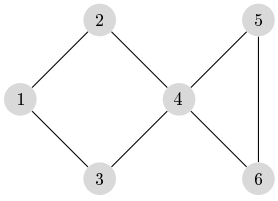
Paso 4: Si ya eliminamos todas las aristas damos por terminado el algoritmo, y nuestro ciclo será $C = C_1$., en caso contrario volvemos al paso 2.

Pseudocódigo algoritmo de Hierholzer.

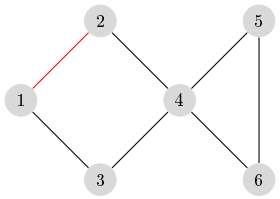
```
ruta=NULL
V="vector conjunto nodo"
A="matiz de costes de las aristas"
M="matriz multiplicidades aristas"
while "Algún elemento de M sea distinto de 0"
    nodo="cualquier nodo que no sea nodo aislado al eliminar las aristas usadas en ruta"
    ruta1=NULL
    while "nodo no es un nodo aislado"
        "buscamos k de modo que M[nodo,k] distinto de 0"
        ruta1=(ruta1,k)
        nodo=k
        M[nodo,k]=M[nodo,k]-1
        M[k,nodo]=M[k,nodo]-1
    ruta="fusionamos ruta y ruta1"
```

Ejemplo algoritmo de Hierholzer.

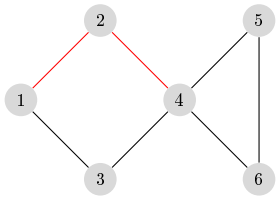
Para el ejemplo de este algoritmo usaremos el mismo grafo $G = (V, A)$ que ya hemos utilizado para el algoritmo de Fleury, que si recordamos es el de la figura.



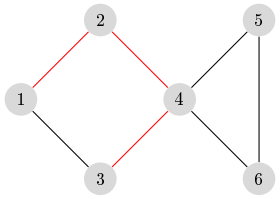
Al igual que en el algoritmo de Fleury, lo primero que tenemos que hacer es definir el nodo inicial v_0,que también definiremos como $v_0 = 1$. Una vez que conocemos el nodo inicial debemos elegir que arista eliminamos, en este caso tenemos dos aristas adyacente en el nodo 1, por lo tanto, o eliminar la arista $(1, 2)$ o la arista $(1, 3)$, nosotros eliminaremos la arista $(1, 2)$.



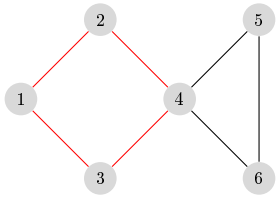
Una vez hecho esto tenemos que eliminar una arista incidente en el nodo 2, que dado que es la única que nos queda tendrá que ser la arista $(2, 4)$.



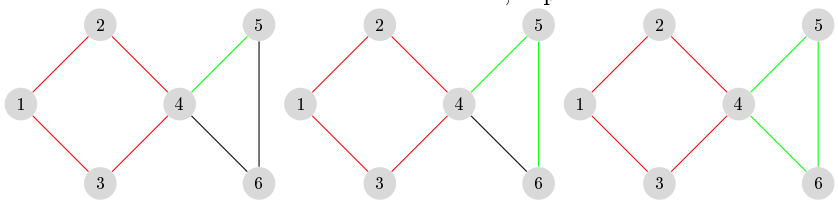
Partiendo del nodo 4 tendríamos tres opciones, que serían las aristas $(4, 3)$, $(4, 5)$ y la $(4, 6)$, y el algoritmo funcionaría con cualquiera de ellas. Sin embargo nosotros eliminaremos la arista $(4, 3)$ para forzar que el primer ciclo que obtengamos no sea un tour (dado que con las otras opciones ya obtendríamos un tour).



Ahora ya solo tenemos una opción, eliminar la arista $(3, 1)$, con lo que obtendríamos un ciclo, el ciclo $C_1 = \{(1, 2), (2, 4), (4, 3), (3, 1)\}$, y vemos que no tenemos mas aristas adyacentes al nodo 1.



Llegados a este punto tenemos que empezar a construir otro ciclo, para el que tomaremos como nodo inicio $V_0 = 4$. El proceso sería análogo el seguido en el ciclo anterior, por lo que solo lo representaremos gráficamente. Para evitar confundir este ciclo con el anterior, representaremos las aristas que eliminamos en verde.



Por lo tanto tendremos dos ciclos, el ciclo $C_1 = \{(1, 2), (2, 4), (4, 3), (3, 1)\}$ y el ciclo que acabamos de construir $C_2 = \{(4, 5), (5, 6), (6, 4)\}$, y al fusionarlos obtenemos el ciclo $C_1 = \{(1, 2), (2, 4), (4, 5), (5, 6), (6, 4), (4, 3), (3, 1)\}$. Vemos que ya no nos quedan aristas por eliminar, por lo tanto este ciclo es el tour euleriano que buscábamos, con lo que $C = C_1$.

CPP no es de grado par

Sin embargo, en la mayoría de casos reales no tendremos un grafo par, y aún así será igual de interesante encontrar un tour de coste mínimo, aún que tal como vimos en el teorema 3.2 el grafo no será euleriano.

Cuando Mei-Ko añadió la cuestión de minimizar la longitud del tour en 1960, su método se basaba en el teorema 3.5 y en que el menor tour se obtendrá duplicando aristas en el grafo original para transformarlo en

un grafo euleriano (o lo que es lo mismo, en un grafo par) y aplicando los algoritmos descritos en la sección anterior. Sin embargo debemos tener que las aristas que dupliquemos las tendremos que recorrer sin realizar ningún servicio, lo que se conoce generalmente como "deadheading", nuestro objetivo será que el coste de las aristas con "deadheading" sea lo menor posible. Para ello el proceso en el que se obtendrá el grafo par con menor coste se consigue conectando cada nodo de grado impar con otro único nodo impar mediante un camino, y duplicando esas aristas, realizándolo de modo óptimo para minimizar el gasto en "deadheading". Es sabido desde el trabajo de Edmonds y Johnson que la "conexión" de mínimo coste se puede calcular resolviendo el problema de emparejamiento. Mei-Ko no dijo esto explícitamente, sin embargo, este resultado se puede deducir de su trabajo, en el que lo primero que hizo fue determinar que las siguientes condiciones son necesarias para que una solución factible sea óptima:

I. No hay redundancia, es decir cada nodo no se duplica mas de una vez.

II. El coste de cada arista añadida en cada ciclo no excede la mitad del coste del ciclo.

Para demostrar la suficiencia de este resultado, demostró que todas las soluciones factibles que cumplan estas dos propiedades tendrán el mismo coste (Mei-Ko solo lo probo con costes unitarios, pero la demostración general sigue el mismo procedimiento). Aún que no sea el tema de nuestro trabajo, podemos observar que la segunda propiedad es básica en la prueba de Christofides de su aproximación-3/2 del problema del viajante(TSP).

El artículo "Matching, Euler Tours and the Chinese Postman"de Edmonds y Johnson es uno de los mas importantes en el campo de las rutas por arcos, y en él los autores plantean el problema de programación lineal asociado como sigue:

$$\text{minimizar} \sum_{(i,j)\in A} c_{ij}x_{ij}$$

sujeto a

$$\sum_{(i,j)\in\delta(S)} x_{ij} \geq 1, \ (S \subset V \text{ tiene un número impar de nodos de grado impar})$$

$$x_{ij} \geq 0, \ \{(i,j) \in A\}$$

$$x_{ij} \text{ es natural}, \ \{(i,j) \in A\}$$

Siendo $x_{ij}(i < j)$ el número de copias de la arista (i,j) que introducimos en el grafo para hacerlo euleriano, y tomando $\delta(S) = (i,j) : \{i \in S, j \in V\backslash S \text{ o } i \in V\backslash S, j \in S\}$ para cualquier subconjunto no vacío S de V.

Resolver este problema de programación entera puede hacerse demasiado costoso, sin embargo, se puede resolver en tiempo polinómico tal como demostró Edmonds (1965). Podemos construir un tour usando el algoritmo de Edmonds y Johnson (1965), que no es mas que una adaptación del algoritmo blossom para el problema del emparejamiento.

Algoritmo para el CPP de grado impar

El algoritmo mas conocido para transformar un grafo de grado impar en un grafo de grado par es el algoritmo de Edmonds y Johnson, que se puede resumir en los siguientes pasos:

Paso 1: Separamos los nodos de grado impar, que denotaremos por $V' \subseteq V$.

Paso 2: Para cada par de nodos calculamos el camino mas corto entre ellos usando el algoritmo de Floyd (o cualquier otro algoritmo para calcular el camino mas corto), con lo que construiremos el grafo completo $G' = (V', A')$, donde el coste de cada arista de A' es la distancia mas corta entre los nodos que une en el grafo G.

Paso 3: En G' resolvemos el problema del acoplamiento perfecto de mínimo coste usando el algoritmo de Edmonds, y denotaremos por M la solución óptima. Añadimos a G las aristas artificiales correspondientes a las aristas de M en G.

Paso 4: Se puede ver que el grafo que obtenemos es par, por lo que podemos calcular un tour usando tanto el algoritmo de Fleury como el de Hierholzer, o cualquier otro algoritmo que conociésemos.

Sin embargo cabe mencionar que para aplicar el algoritmo de Edmonds y Johnson tenemos que resolver un problema de acoplamiento perfecto, Grötschel y Holland (1985) implementaron un algoritmo de planos de corte

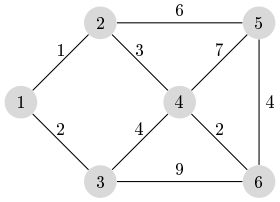
que demostró ser tan eficiente como los algoritmos existentes de tipo combinatorio, por lo que el algoritmo de Edmonds y Johnson no es el único que podemos utilizar, aunque en este documento solo estudiaremos éste.

Pseudocódigo algoritmo de Edmonds y Johnson.

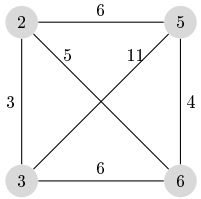
```
V="vector conjunto nodos"
n="número de aristas (teniendo en cuenta multiplicidades)"
A="matiz de costes de las aristas"
M="matriz multiplicidades aristas"
V1="nodos de grado impar"
A1="arístas que unen los nodos de V1 y cuyo coste es igual al camino mas corto que los une"
"resolver el problema emparejamiento en G1=(V1,A1)
si i y j forman una pareja
    para cada (a,b) en el camino mas corto de i a j
        M[a,b]=M[a,b]+1
        M[b,a]=M[b,a]+1
"aplicar Fleury o Hierholzer"
```

Ejemplo algoritmo de Edmonds y Johnson.

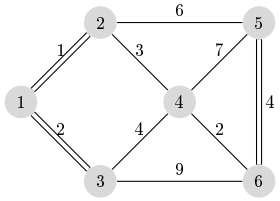
Empezaremos definiendo un grafo en el que aplicaremos el algoritmo. Nótese que a diferencia de los ejemplos de los algoritmos de Fleury y de Hierholzer, en este caso si hemos asociado costes a cada arista, dado que en este los costes si son importantes.



Podemos ver que este grafo no es par, de hecho tenemos 4 nodos de grado par, que son los nodos 2,3,5 y 6. Una vez que sabemos cuales son los nodos de grado impar construimos el grafo completo $G' = (V', A')$, formado por los nodos de grado impar de G y cuyas aristas tienen un coste asociado igual al camino mas corto entre los nodos que unen. Téngase en cuenta que aunque en este caso se puede calcular el camino mas corto a "ojo", normalmente tendremos que recurrir a alguno de los algoritmos para el camino mas corto. El grafo que obtenemos es el siguiente:



Una vez que tenemos el grafo G', lo único que tenemos que hacer es calcular el emparejamiento perfecto, nuevamente podemos calcularlo a "ojo", sin embargo en un problema de mayor tamaño tendríamos que aplicar algún algoritmo. En este caso el emparejamiento perfecto es juntar los nodo 2 y 3 y por otra parte los nodos 5 y 6, o lo que es lo mismo, el emparejamiento perfecto es $M = \{(2,3),(5,6)\}$. Tras hacerlo duplicamos las aristas que forman el camino mas corto entre cada uno de los nodos emparejados. Representaremos en color azul las aristas que son duplicadas.



Este ya sería un grafo de grado par, por lo que podríamos calcular un tour euleriano siguiendo cualquiera de los algoritmos ya mencionados.

3.1.2. El problema del cartero chino en un grafo dirigido (DCPP)

Sea $G(V, A)$ un grafo dirigido con costes no negativos asociados a los arcos dirigidos de A, aún que propiamente solo necesitaríamos que no hubiera ciclos de coste negativo. Para que exista solución es necesario y suficiente que el grafo sea fuertemente conexo. Esta condición fue descrita formalmente por Ford y Fulkerson en 1962, aún que ya era conocida mucho antes, Koning habló del tema en su libro en 1936.

En 1965 Edmonds demostró que el problema se podría resolver en un tiempo polinómico. El proceso para resolver el problema fue planteado simultáneamente por Edmonds y Johnson (1973), Orloff (1974) y Beltrami y Bodin (1974).

Teorema 3.6. *Un grafo G fuertemente conexo y dirigido, contiene un ciclo eureliano si y solo si es un grafo simétrico.*

Si el grafo es simétrico podemos usar el algoritmo de Fleury o el de Hierholzer, al igual que en el CPP. En caso de no serlo tendremos que construir un grafo simétrico, para ello tendremos que duplicar arcos dirigidos.Denotemos por $d^-(i)$ el número de arcos dirigidos cuyo nodo fin es el nodo i, y como $d^+(i)$ el numero de arcos dirigidos cuyo nodo origen es el nodo i. Denotemos por I el conjunto de nodos tales que $D(i) = d^-(i) - d^+(i) > 0$ y por J los nodos tales que $d^-(i) - d^+(i) < 0$. Los nodos $i \in I$ son considerados nodos oferta con oferta $D(i)$ y los nodos $j \in J$ son considerados nodos demanda con demanda $D(j)$. Para transformar el grafo en simétrico podemos resolver el siguiente problema de programación lineal:

$$\text{Minimizar } \sum_{i \in I} \sum_{j \in J} c_{ij} x_{ij}$$

Sujeto a:

$$\sum_{j \in J} x_{i,j} = D(i), \quad (i \in I)$$

$$\sum_{i \in I} x_{i,j} = D(j), \quad (j \in J)$$

$$x_{ij} \geq 0, \quad (i \in I, j \in J)$$

La solución optima nos indica cuantas veces tenemos que duplicar cada arco dirigido, con lo que ya tendríamos un grafo simétrico, y lo resolveríamos con los mismos algoritmos que el CPP de grado par. Sin embargo, resolver un problema de programación lineal se puede hacer muy complicado, por eso se propusieron otros algoritmos.

Algoritmo para hacer simétrico el DCPP

Una posible opción sería plantearlo como el problema de encontrar el plan de transporte de coste mínimo entre los nodos oferta y los nodos demanda (Liebling, 1970; Edmonds y Johnson, 1973), con un coste $O = (|A||V|^2)$.El algoritmo quedaría como sigue:

Paso 1: Calcular el camino mas corto de cada nodo oferta a cada nodo demanda usando el algoritmo de Floyd (o cualquier otro algoritmo para el camino mas corto).

Paso 2: Resolvemos el problema del transporte equilibrado asociado a los nodos oferta y demanda.

Paso 3: Añadimos al grafo $G = (V, A)$ los arcos dirigidos asociados a la solución del problema del transporte, con lo que ya tenemos un grafo simétrico.

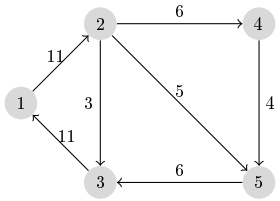
Pseudocódigo algorítmo

```
V="vector conjunto nodos"
A="matiz de costes de las arcos"
M="matriz multiplicidades arcos"
```

```
V1="nodos oferta"
V2="nodos demanda"
A1="matriz de costes camino mas corto de cada nodo oferta a los nodos demanda"
"resolver el problema transporte en A1"
t(a,b)="cantidad que transportamos del nodo a al nodo b en el problema del transporte"
M[a,b]=M[a,b]+t(a,b)
```

Ejemplo algorítmo

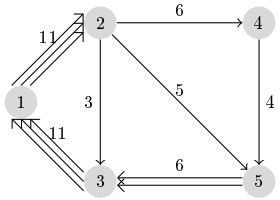
Comencemos definiendo un grafo $G = (V, A)$ dirigido, como puede ser el de la figura:



En el caso del CPP dirigido no se suele ver tan fácilmente si es fuertemente conexo, y aunque este ejemplo si cumpla esta propiedad, no está de mas recordar que siempre que nos enfrentemos a un DCPP debemos comprobar que efectivamente es fuertemente conexo. Una vez que sabemos que el grafo es fuertemente conexo, debemos comprobar cuales son los nodos oferta y cuales los nodos demanda. En este caso tenemos nodos oferta en el nodo 3, con una oferta de 1, y el nodo 5, también con una oferta de 1. Como nodo oferta solo tendremos al nodo 2, con una demanda de 2. Con lo que la tabla del problema del transporte nos quedaría:

	Nodo 2	
Nodo 3	22	1
Nodo 5	28	1
	2	

La solución a este problema del transporte es trivial, tenemos que enviar una unidad del nodo 3 al nodo 2 y otra unidad del nodo 5 al nodo 2. Por lo tanto duplicaríamos los arcos dirigidos correspondientes a la solución del problema del transporte.



Una vez llegados a este punto podemos calcular el tour usando cualquiera de los algoritmos que vimos para el CPP. En este caso el tour de mínimo coste, o al menos uno de los que tienes mínimo coste, pues no tiene porqué ser único, es $C = \{(1,2),(2,4),(4,5),(5,3),(3,1),(1,2),(2,5),(5,3),(3,1),(1,2),(2,3),(3,1)\}$, cuyo coste es 96 unidades.

3.1.3. El problema del cartero chino en un grafo mixto (MCPP)

Sea $G = (V, A)$ un grafo mixto, con A_e el conjunto de aristas, A_a el conjunto de arcos dirigidos y por lo tanto $A_e \cup A_a = A$ y $A_e \cap A_a = \emptyset$. Al igual que en los casos anteriores para que el problema tenga solución optima finita necesitaremos que en grafo sea fuertemente conexo con costes asociados no negativos a los arcos o aristas.

La primera referencia que tenemos sobre la condición necesaria y suficiente de existencia de un tour euleriano en un grafo mixto fue planteado por Ford y Fulkerson (1962), la cual decía que en cada nodo debe incidir un numero par de arcos y aristas, y que para cualquier $S \subset V$ no vacío el valor absoluto de la diferencia entre el número de arcos de S hasta $V \setminus S$ y el número de arcos de $V \setminus S$ hasta S debe ser menor o igual al número de aristas entre $V \setminus S$ y S.

A diferencia de los casos anteriores, aún cuando se cumplen las condiciones de existencia, este es un problema NP-duro, como demostró Papadimitriou (1976), incluso siendo un grafo planar o si los costes de todos los arcos y arista son iguales. Sin embargo, en el caso de que el grafo sea par si existen algoritmos polinomiales para resolver el MCPP. Christofides et al.(1984) propusieron una formulación con una variable por cada arco, dos variables por cada arista (representando el número de veces que es atravesada en cada dirección) y una variable por cada nodo. Nosotros en este trabajo veremos una formulación propuesta por Ralphs (1993).

Sea y_a el número de copias que añadiremos del arco $a \in A_a$ y A_r y A_l los conjuntos formados por arcos de orientaciones opuestas para cada arista $e \in A_e$. Denotemos por u_e^r a la primera orientación de la arista e (que tomara el valor 1 o 0 según usemos el arco con esa orientación o no) y y_e^r cada copia adicional que añadamos de dicho arco. Definiremos de forma análoga u_e^l y y_e^l. Definamos $I(i)$ el conjunto de arcos dirigidos que entran y $O(i)$ el conjunto de arcos dirigidos que salen del nodo i. Con esto nuestro problema de programación lineal sería:

$$\min \sum_{a \in A_a} c_a(y_a + 1) + \sum_{a \in A_r \cup A_l} c_e(u_e^r + y_e^r + u_e^r + y_e^l)$$

Sujeto a:

$$u_e^r + u_e^l = 1, \forall e \in A_e$$

$$x_a = 1 + y_a, \forall a \in A$$

$$x_a = u_e^r + y_e^r, \forall a \in A_r$$

$$x_a = u_e^l + y_e^l, \forall a \in A_r$$

$$\sum_{a \in O(i)} x_a - \sum_{a \in I(i)} = 0, \forall i \in V$$

$$y_a, y_e^r, y_e^l \geq 0 \text{ y enteras }, \forall e, a$$

$$u_e^r, u_e^r \in \{0, 1\}, \forall e$$

Algoritmos polinómicos

Para que exista un algoritmo polinómico necesitamos que el grafo $G = (V, A)$ sea par, sin embargo tendremos dos casos posibles, que el grafo sea simétrico, o que no lo sea.

Algoritmo 1. El grafo es par y simétrico.

En este caso la solución es bastante sencilla, podemos aplicarle el algoritmo de Hierholzer con una pequeña modificación, cuando construyamos cada uno de los ciclos, si para salir de un nodo disponemos tanto de arcos dirigidos como aristas para salir de el, usaremos un arco dirigido, dado que si usamos la arista el algoritmo puede fallar. También podríamos adaptar el algoritmo de Floyd, pero sería mas complicado que el de Hierholzer.

Algoritmo 2. El grafo es par, pero no es simétrico

Ahora tendremos que transformarlo en un grafo simétrico, para poder resolverlo como el caso en el que es par y simétrico, para lo que seguiremos los siguientes pasos:

Paso 1: Para transformar el grafo en simétrico seguiremos el proceso propuesto por Edmonds y Johnson, primero definiremos $G' = (V, A' \cup A_1 \cup A_2 \cup A_3)$ donde A' contiene los arcos dirigidos de A con sus costes originales, A_1 y A_2 contiene arcos dirigidos con dirección opuesta por cada arista de A, con costes iguales al coste de la arista y A_3 una arista con coste 0 por cada arista de A_1. Asignaremos capacidad infinita a cada arco de $A' \cup A_1 \cup A_2$, y capacidad uno a cada arista de A_3. Denotemos los nodos con $D(i) > 0$ como nodos oferta con una oferta $D(i)$, y los nodos $D(i) < 0$ como nodos demanda con una demanda $D(i)$.

Paso 2: Sea x_{ij} el flujo que atraviesa los arcos y aristas de $A' \cup A_1 \cup A_2 \cup A_3$. Definamos $G'' := G$, en el que haremos los siguientes cambios:

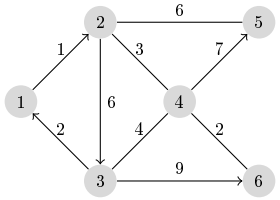
Paso 2.1: Si $x_{ij} = 1$ en una arista de A_3 entonces orientamos la arista de v_i a v_j.

Paso 2.2: Si $x_{ij} = 1$ en un arco de A', A_1 o A_2 añadimos una copia del arco correspondiente a G''.

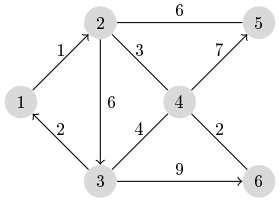
Paso 3: El grafo G'' es par y simétrico, por lo que podemos resolver el problema aplicando el algoritmo anterior.

Ejemplo del algoritmo 2

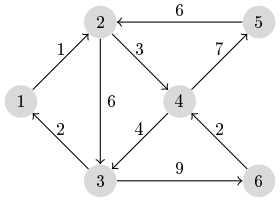
Definamos el grafo $G = (V, A)$ de la figura.



Es fácil ver que se trata de un grafo par pero no simétrico. Lo primero que hacemos es resolver el problema del flujo asociado, donde los nodos 5 y 6 son nodos oferta, ambos con oferta 1, y los nodos 3 y 4 es un nodo demanda, también ambos con demanda 1, en el grafo:



Vemos que $x_{52} = 1$, $x_{64} = 1$ y $x_{43} = 1$, siendo el flujo por los demás arcos y aristas igual a 0. Por lo que nuestro grafo resultante es:



Podemos ver que el grafo ya es par y simétrico, con lo que lo podemos resolver aplicando el algoritmo de Hierholzer adaptado al grafo mixto par y simétrico.

Algoritmos heurísticos

Quizás las dos mejores opciones para transformar el grafo original en un grafo par y simétrico son las propuestas por Edmonds y Johnson (1973), y mejorado después por Frederickson (1979) y Chistofides (1984). Llamemos a estos algoritmos *MIXED1* y *MIXED2*. Debemos tener en cuenta que estos algoritmos son efectivos por separado, pero fallan si se aplican ambos al mismo tiempo. Además Frederickson demostró que el ratio del peor caso de ambos algoritmos es 2, que además es alcanzable, sin embargo si se aplica la mejor solución entre las dos producidas por estos algoritmos este ratio es de $\frac{5}{3}$, aunque a día de hoy no se conocen ejemplos con un ratio peor a $\frac{3}{2}$. En el caso del grafo planar Frederickson presentó un algoritmo cuyo ratio en el peor caso es $\frac{3}{2}$.

Mixed 1.

Paso 1: Transformar el grafo original el grafo original en un grafo par. Para ello determinamos el conjunto V' de nodos de grado impar, y construimos el grafo completo $G' = (V', A')$, donde el coste de cada arista $(v_i.v_j)$ se calcula como la longitud de la cadena mas corta SP_{ij} entre los nodos v_i y v_j (calculada ignorando la dirección de los arcos dirigidos). Resolveremos en problema de acoplamiento mínimo entre los nodos de G' y para cada par de nodos v_i y v_j en la solución a dicho problema, añadimos a G una copia de cada arco dirigido y arista contenidas en la cadena SP_{ij}.

Paso 2: El grafo obtenido es par, por lo que podemos aplicarle al menos uno de los algoritmos polinomicos para el CPP mixto.

Mixed 2.

Paso 1: Aplicamos el algoritmo polinómico que utilizamos cuando el grafo es par pero no simétrico, obteniendo así un grafo simétrico (que en este caso no será par).

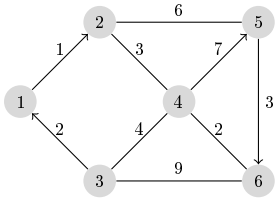
Paso 2: Con el grafo $G' = (V, A_e)$, el grafo inducido por las aristas del grafo G original, resolvemos el algoritmo de acoplamiento mínimo. Añadimos al grafo del paso 1 una copia de cada arista implicada en la solución del problema del acoplamiento, con lo que obtenemos un grafo par y simétrico.

Pseudocódigo mixed 1.

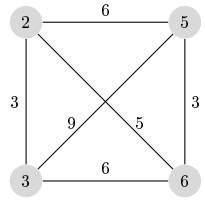
```
V="vector conjunto nodos"
n="número de aristas (teniendo en cuenta multiplicidades)"
A="matiz de costes de las arcos"
MA="matriz multiplicidades arcos"
E="matiz de costes de las aristas"
ME="matriz multiplicidades aristas"
V1="nodos de grado impar"
A1="aristas que unen los nodos de V1 y cuyo coste es igual al camino mas corto que los une
calculado ignorando la dirección de las aristas"
"resolver el problema emparejamiento en G1=(V1,A1)
si i y j forman una pareja
    para cada (a,b) en el camino mas corto de i a j (calculado ignorando la dirección de
    los arcos)
        si ME[a,b] distinto de 0
            ME[a,b]=ME[a,b]+1
            ME[b,a]=ME[b,a]+1
        si MA[a,b] distinto de 0
            MA[a,b]=MA[a,b]+1
"aplicar alguno de los algoritmos polinómicos para grafos de grado par"
```

Ejemplo Mixed 1.

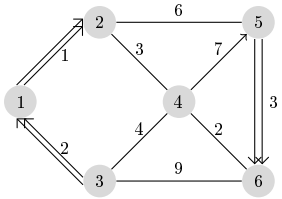
Definamos un grafo $G = (V, A)$ mixto.



Es fácil ver que no estamos en condiciones de aplicar ninguno de los algoritmos polinómicos que vimos en el apartado anterior. Los nodos de grado impar son los nodos 2, 3, 5 y 6. Construimos el grafo completo G', obteniendo:

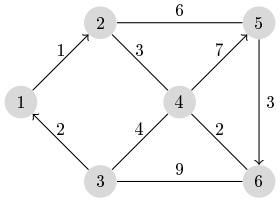


Véase que los costes de cada arista se corresponde con la longitud de la cadena mas corta en G entre los nodos que une (calculada ignorando la dirección de los arcos dirigidos). Vemos que el acoplamiento perfecto se da cuando juntamos el nodo 2 con el 3 y el nodo 5 con el 6. Duplicamos las aristas y los arcos dirigidos de las cadenas SP_{23} y SP_{56}, con lo que obtenemos:

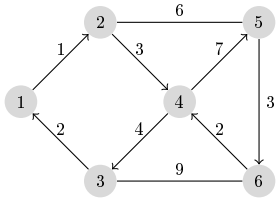


Podemos ver que nos encontramos ante un grafo par pero no simetrico, por lo que podríamos aplicarle el segundo algoritmo polinómico para el MCPP.

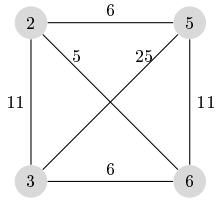
Ejemplo Mixed 2.
Usaremos el mismo grafo G que usamos en el ejemplo para ilustrar el Mixed 1.



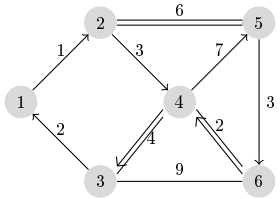
Tras aplicar nuestro algoritmo para hacer el grafo simétrico obtenemos:



Vemos que el grafo no es par, para hacerlo par resolvemos el problema del acoplamiento de mínimo coste entre los nodos de grado impar, pero usando solo las aristas del grafo original, o lo que es lo mismo, calculamos el acoplamiento de mínimo coste en el siguiente grafo:



En este caso el acoplamiento es sencillo, simplemente acoplaremos 3 con 6 y 2 con 5. Duplicamos las aristas de la solución, en este caso las aristas $(2,5)$, $(4,6)$ y $(3,4)$ y obtenemos:



Fijémonos que entre los nodos 3 y 4 y entre 4 y 6 tenemos un arco y una arista, pero aún que esta sea una situación un poco particular, al estar ante un grafo par y simetrico podemos resolverlo aplicando usando el primer algoritmo exacto.

3.2. Otros problemas

Los casos que vimos hasta ahora, pese a formaren una buena base teórica, no siempre se adaptan a la realidad, de hecho en la investigación operativa no existe ningún modelo que se adapte al 100 % a los problemas reales, por eso los matemáticos actualizan estos modelos con pequeñas variantes, que intentan explicar las variaciones mas comunes en los problemas reales. A lo largo de este apartado veremos resumidas las principales variantes que han estudiado los matemáticos en los últimos años, destacando entre ellas el problema del cartero rural, que quizás sea el que con mas frecuencia se dá en problemas reales.

3.2.1. El problema del cartero rural(RPP)

En algunos casos no nos interesa atravesar todas las aristas, o arcos, si no solo un subconjunto de ellos (que denominaremos como aristas o arcos requeridos y denotaremos por A_R). Sea el grafo $G(V, A)$ un grafo conexo

con costes asociados no negativos, en este problema nos interesa definir un ciclo que recorra todos los arcos y/o aristas de A_R al menos una vez. El RPP puede plantearse como un cartero que tiene que repartir el correo en varios pueblos, tiene que recorrer todas las calles de los pueblos, pero no todas las carreteras que los unen.

Existen varias variantes, pero no todas fueron estudiadas en profundidad. En este documento hablaremos brevemente del caso dirigido y no dirigido, y aún que el caso mixto no esta muy estudiado, si dedicaremos unas lineas a una variante conocida como el problema de la grúa.

En general se trata de un problema NP-duro, sin embargo si es un grafo estrictamente dirigido o no dirigido se puede resolver en tiempo polinómico siempre y cuando el subgrafo formado por las aristas o arcos requeridos sea fuertemente conexo.

RPP no dirigido

Este problema fue planteado por Orloff (1974) y en 1976 Lenstra y Rinnooy probaron que es NP-duro. Orloff señalo que la complejidad aumenta a medida que aumentan el número de componentes conexas, lo cual coincide con al observación de Frederickson (1979) de que existe un algoritmo recursivo exacto para el RPP cuyo coste es exponencial en el número de componentes conexas. El primer planteamiento como problema de programación lineal fue dado por Chistofides et al. (1981), pero no fue examinado en detalle, lo cual fue hecho en Corberán y Sanchis (1994), obteniendo la siguiente formulación:

$$x(\delta(S)) \geq 2, \forall S \subset V \begin{cases} \delta_R(S) = \emptyset \\ S \cap R \neq \emptyset \\ V_R \backslash S \neq \emptyset S \end{cases}$$

$$x(\delta(i)) \equiv |\delta_R(i)| (mod 2) \forall i \in V$$

$$x_a \geq 0 (\forall a \in A)$$

$$x \in Z^{|E|}$$

Siendo $\delta_R = \delta(S) \cap A_R$.

Frederickson (1979) y Christofides et al. (1981) definieron un algoritmo aproximado para resolver el RPP basado en construir el árbol de expansión mínima que conecte todas las componentes conexas, de forma similar al algoritmo de Christofides para el TSP, para ello tendremos que introducir unas cuantas definiciones. Definamos el grafo $G_R = (V_R, A_R \cup A_S)$, donde V_R son los nodos incidentes en al menos una arista de A_R y las aristas A_S unen cada par de nodos $i, j \in V_R$ y tienen un coste igual al camino mas corto entre i y j. El algoritmo consiste en:

Paso 1: Sean $G_1, G_2, \ldots, G_n$ las componentes conexas inducidas por G_R por el conjunto de aristas A_R. Definamos el grafo completo $G' = (V', A')$, donde cada nodo de V' representa una componente conexa G_i, y el coste de cada arista es la menor distancia entre ambas componentes conexas.

Paso 2: Calcular el árbol de expansión mínima de G'. Denotemos por A_{RS} el conjunto de aristas de G_R correspondientes al árbol.

Paso 3: Tomemos V_0 el conjunto de nodos de grado impar sobre el grafo $\overline{G} = (V_R, A_R \cup A_{RS})$. Sobre ellos calcular el acoplamiento de mínimo coste, y denotaremos esas aristas por A_{PM}, y las añadiremos al grafo.

Paso 4: Sea el grafo $G_H = (V_R, A_R \cup A_{RS} \cup A_{PM})$ es un grafo conexo y par, por lo que podemos usar cualquiera de los algoritmos que conocemos para el CPP, con lo que obtendríamos una solución factible para el RPP.

Este algoritmo tiene un ratio de 3/2 tal como demostró Frederickson (1979), sin embargo debemos tener en cuenta que este ratio solo se cumple si la matriz cumple la desigualdad triangular. Recientemente Hertz, Laporte y Nanchen-Hugo (1999) desarrollaron una serie de algoritmos heurísticos de post-optimización para el RPP no dirigido, los cuales nos dan una solución óptima o casi óptima.

Como caso particular tenemos e caso en el que el grafo $G(V, A_R)$ sea un grafo conexo, en el cual el algoritmo nos aportaría la solución exacta.

RPP dirigido (DRPP)

Tenemos un algoritmo heurísticos propuesto también por Chistofides et al (1986) para el RPP dirigido de forma similar al caso no dirigido, con la diferencia de que en lugar de construir un árbol de expansión mínima construiremos una arborescencia entre las componentes requeridas conexas, y que en lugar de resolver el problema de emparejamiento resolvemos un problema del transporte. Definamos el grafo $G_R = (V_R, A_R \cup A_S)$, donde V_R son los nodos incidentes en al menos un arco de A_R y los arcos A_S unen cada par de nodos $i, j \in V_R$ y tienen un coste igual al camino mas corto desde i hasta j. Sean $G_1, G_2, \ldots, G_n$ las componentes conexas inducidas por G_R por el conjunto de arcos A_R. Partiendo de una componente G_i el algoritmo consiste en:

Paso 1: Calcular en G' la arborescencia de coste mínimo con raíz en G_i. Sea A^i_{RS} el conjunto de arcos correspondientes a la arborescencia.

Paso 2: En el grafo $\overline{G}^i = (V_R, A_R \cup A^i_{RS})$ identificaremos los nodos fuentes y sumideros. Sobre ellos resolvemos el problema del transporte asociado. Denotemos por A^i_T el conjunto de arcos asociados a la solución del problema (tengamos en cuenta que A^i_T contiene tantas copias de cada arco como veces aparece en la solución)

Paso 3: Sea el grafo $G_H = (V_R, A_R \cup A^i_{RS} \cup A^i_T)$ es un grafo conexo y par, por lo que podemos usar cualquiera de los algoritmos que conocemos para el DPP, con lo que obtendríamos una solución factible para el DRPP.

Variando el G_i que tomamos como raíz obtenemos una solución factible distinta. Ahora tan solo tendremos que elegir la que tenga un coste asociado menor.

Pese a que el algoritmo es similar al dado para en caso no dirigido, en este el ratio en el peor caso no esta acotado de forma general, sin embargo si tenemos una cota en función de los datos, es concreto en función del valor α que cumple que:

$$c_{ij} \leq \alpha c_{ji} \forall i, j \in V_R$$

Es fácil ver que esta α existe y es finito mientras todos los arcos tengan coste positivo.

RPP mixto

Este caso, pese a ser común, es de reciente aparición, y por lo tanto no disponemos de mucha información a cerca de él. Tan solo conocemos la aportación de Corberán, Marti y Romero (2000), por lo que en este documento no nos centraremos mucho en esta variante.

El problema de la Grúa (SCP)

El Problema de la Grúa o "Stacker Crane Problem" es un caso particular del RPP mixto, en el que los arcos requeridos coincidirán con los arcos dirigidos. El SCP fue introducido por Frederickson, Hecht y Kim (1978). El problema es NP-duro, para demostralo Frederickson, Hecht y Kim (1978) comprobaron que el problema del Viajante (TSP), que es NP-duro, puede transformarse en otra versión del SCP. Ellos además propusieron un algoritmo heurísticos cuyo ratio de error en el peor caso es $9/5$ y un coste de $O(n^3)$, por lo que se trata de un algoritmo polinómico, donde n es el numero de arcos a recorrer. El algoritmo propuesto escoge la mejor solución dada por dos algoritmos, "LARGEARCS", que da mejores resultados cuando el coste de los arcos es menor que el de las aristas usadas en la solución óptima, y "LARGEEDGE", que da mejores resultados cuando el coste de los arcos es parecido al de las aristas, teniendo en cuenta que ambos necesitan que se cumpla la desigualdad triangular. El primero algoritmo resuelve un problema de acoplamiento de mínimo coste, y después calcula el árbol de expansión de mínimo coste mientras que el segundo algoritmo primero hace una transformación del SCP en un TSP con el objetivo de poder aplicarle el conocido algoritmo de Christofides (1976) para el TSP. Zhang (1992) propuso una simplificación del algoritmo reduciendo su coste de $O(n^3)$ a $O(n^2)$, pero que empeora el ratio pasando a ser este igual a 2. Sin embargo este no el único algoritmo, Zhang y Zheng (1995) publican un artículo donde se muestra una posible transformación del SCP en un problema del viajante de comercio asimétrico, con lo que se le puede aplicar cualquier algoritmo propio de estos problemas. Recientemente disponemos de un estudio escrito por Srour y van de Velde (2011) en el que comparan la dificultad del SCP con la del problema del viajante de comercio asimétrico y un artículo de Cirasella et al. (2007), donde se realiza una comparación de heurísticos para el ATSP y los del SCP.

3.2.2. El problema de los m carteros(m-CPP)

Otra posible extensión sería si consideramos que en lugar de un cartero tenemos m carteros, o lo que es lo mismo, en lugar de encontrar un tour encontraremos m tours, todos ellos con el mismo nodo de partida, y que entre todos atraviesan todas las aristas o arcos del grafo. Se le pueden poner muchísimas restricciones distintas a este problema, capacidad de los vehículos, distancia de cada tour, tiempo necesario para cada tour etc.

Pese a que se podría plantear en cualquier situación, este problema se ha estudiado mayormente en grafos no dirigidos, y se han planteado multitud de algoritmos heurísticos. Hasta el día de hoy no hay ningún algoritmo exacto, sin embargo no esta demostrado que sea un problema NP-duro, por lo que es posible que en un futuro alguien proponga un algoritmo exacto.

Una de las versiones mas habituales (que es en la que nos centraremos en este proyecto) es aquella en la que cada arco o arista tiene asociada una cantidad no negativa q_{ij}. Disponemos de m vehículos cada uno con una capacidad Q_k (que puede ser la misma para todos los vehículos), y nuestro objetivo será recorrer todos los arcos y aristas y recoger (o dejar) todas las cantidades q_{ij} sin que ningún vehículo supere su capacidad Q_k asociada. Por eso a esta versión también se le llama problema de rutas por arcos con capacidades. Esto fue planteado por Golden y Wong (1981), aunque mucho antes la opción en la que todos los q_{ij} son estrictamente positivos fue estudiada por Christofides (1973). Es mas, el problema planteado por Golden y Wong se puede ver como un RPP con restricción de capacidades y m vehículos, mientras que la versión de Christofides de puede ver como un CPP con restricción de capacidades y m vehículos.

Se propusieron muchos algoritmos para resolver este problema, quizás uno de los mejores sera el algoritmo contruct-strike de Christofides (1973), que mas tarde fue mejorado por Pearn (1989), y el algoritmo argument-insert de Pearn (1991).

3.2.3. problema del viento(WPP)

En la vida real tenemos multitud de ejemplo en los que no es igual de costoso recorrer las aristas en un sentido que en el otro, bien porque se trata de un camino empinado, porque en un sentido hay mas trafico que en otro, etc. Para resolver este tipo de problemas nace el problema del viento, el cual debe su nombre a que no es igual de costoso ir a favor que contra el viento.

El problema del viento se define un grafo no dirigido en el que cada arista tiene dos costes asociados, uno en cada dirección, aún que cada arista solo será necesario recorrerla una vez. Nuevamente para que exista solución optima finita necesitamos que el grafo sea conexo con costes asociados en ambos sentidos no negativos, solo que en este caso será condición suficiente. El problema fue introducido por Minieka (1979) (él cual planteaba el CPP, el DCPP y el MCPP adaptados al problema del viento). Brucker (1981) y Mei-Ko (1984) demostraron que se trata de un problema NP-duro. Un buen estudio sobre el WPP puede encontrarse en la tesis de Win (1989), el cual demostró que se puede resolver en tiempo polinómico en algunos supuestos, si G es euleriano o si sus costes son equilibrados por ciclos (Fleischner, 1991).

Existe un algoritmo heurísticos, con ratio en el peor de los casos igual a 2, para resolver el WPP basado en que si todos los nodos tienen grado par lo podemos resolver en tiempo polinómico, que lo único que hace es primero convertir el grafo en un grafo par y luego aplica el algoritmo exacto para grafos pares. Posteriormente Raghavachari y Veerasamy (1999) propusieron una modificación de este algoritmo y demostraron que tiene una cota en el peor caso de $3/2$.

3.2.4. El problema del Cartero Rural con viento (WRPP)

Un caso particular del problema del viento se da cuando no tenemos que recorrer todos los arcos y/o aristas, si no solo un subconjunto, o lo que es lo mismo, cuando estamos en el caso del cartero rural. Nuevamente, para que exista solución optima finita necesitaremos que el grafo sea conexo y que los costes asociados a las aristas en ambos sentidos sean no negativos.

3.2.5. Problema jerárquico

Por último hablaremos de un problema introducido por Dror, Stern y Trudeau (1987) que también se da en la vida real, se trata del caso en el que unos arcos, o aristas, tienen diferentes prioridades, por lo que no los podremos recorrer en cualquier orden. Esto se da muy a menudo en las rutas de recogida de nieve, en las que las calles principales tienen que ser limpiadas antes de las secundarias (Stricker ,1970; Lenieux y Campagna, 1984; Alfa y Lieu, 1988; Haslam y Wright, 1991), en recogidas de basura (Bodin y Kursh, 1978) o en trabajos de extinción de incendios (Manber y Israni,1984).

Este problema es un problema NP-duro, sin embargo Dror demostró que existen algoritmos polinómicos cuando:

I. El grafo es completamente dirigido o no dirigido.

II. La relación de orden entre las clases es completa.

III. Cada clase incluye un grafo conexo.

El mismo Dror propuso un algoritmo polinómico con coste $O(k|V|^5)$ siendo k el numeró de clases. Además, en el año 2000 Ghiani y Improta demostraron que este caso particular puede resolverse como un problema de emparejamiento en un grafo auxiliar con $O(k|V|)$ nodos. Existe una variante de este problema, propuesta por Letchford y Eglese (1998), que además de separar los arcos o aristas en clases pone un tiempo limite para el servicio de cada una de las clases, y comprueba cuando la solución cumple ese tiempo limite. El mismo autor propuso un algoritmo branch-and-cut capaz de resolver problemas de tamaño pequeño o mediano.

3.3. Paquete en R.

Todos los métodos que hemos visto a lo largo de este trabajo son prácticamente imposible de aplicarlos a mano a un problema de un tamaño medio-grande, por ello, en la practica necesitaremos valernos de algún software especifico, que nos permita aplicas estos algoritmos en un tiempo razonable. Como el software mas utilizado en este máster fue R, hemos decidido programar en R un paquete que resuelve el CPP, el DCPP y el MCPP, y como caso particular el problema del cartero rural en el que los arcos y aristas requeridos formen un grafo conexo. La idea original era subir el paquete al CRAN, y en un futuro se hará, pero como todavía no ha sido posible lo tendremos que instalar desde el archivo zip. En un futuro se podrá descargar directamente desde el CRAN con el nombre solverChinPost.

3.3.1. Instalar el paquete en R

Para instalar en paquete en R será suficiente con hacer clic en la pestaña *paquetes*, y luego *instalar paquete(s) a partir de archivos zip locales...*, tal como se ve en la figura 3.6. Una vez hecho esto, tendremos que seleccionar el archivo zip en la careta en la que lo tengamos guardado y R instalará tanto el paquete solverChinPost como los paquetes necesarios para el buen funcionamiento de éste.

3.3.2. Función principal.

La función *solverChinPost* es la función principal del paquete, y es ella la que se encargará de resolver cualquiera de los problemas antes mencionados. Su estructura básica es:

$$solverChinPost(Arcs, Edges, ME, MA, mixed1, mixed2)$$

Veamos que información nos aporta cada una de estas variables.

- La variable *Arcs* nos dirá el coste de todos los arcos presentes en el grafo. Por defecto toma el valor "*nomatri*", el cual solo nos interesa en el caso de que no tengamos arcos dirigidos (lo que solo ocurre en

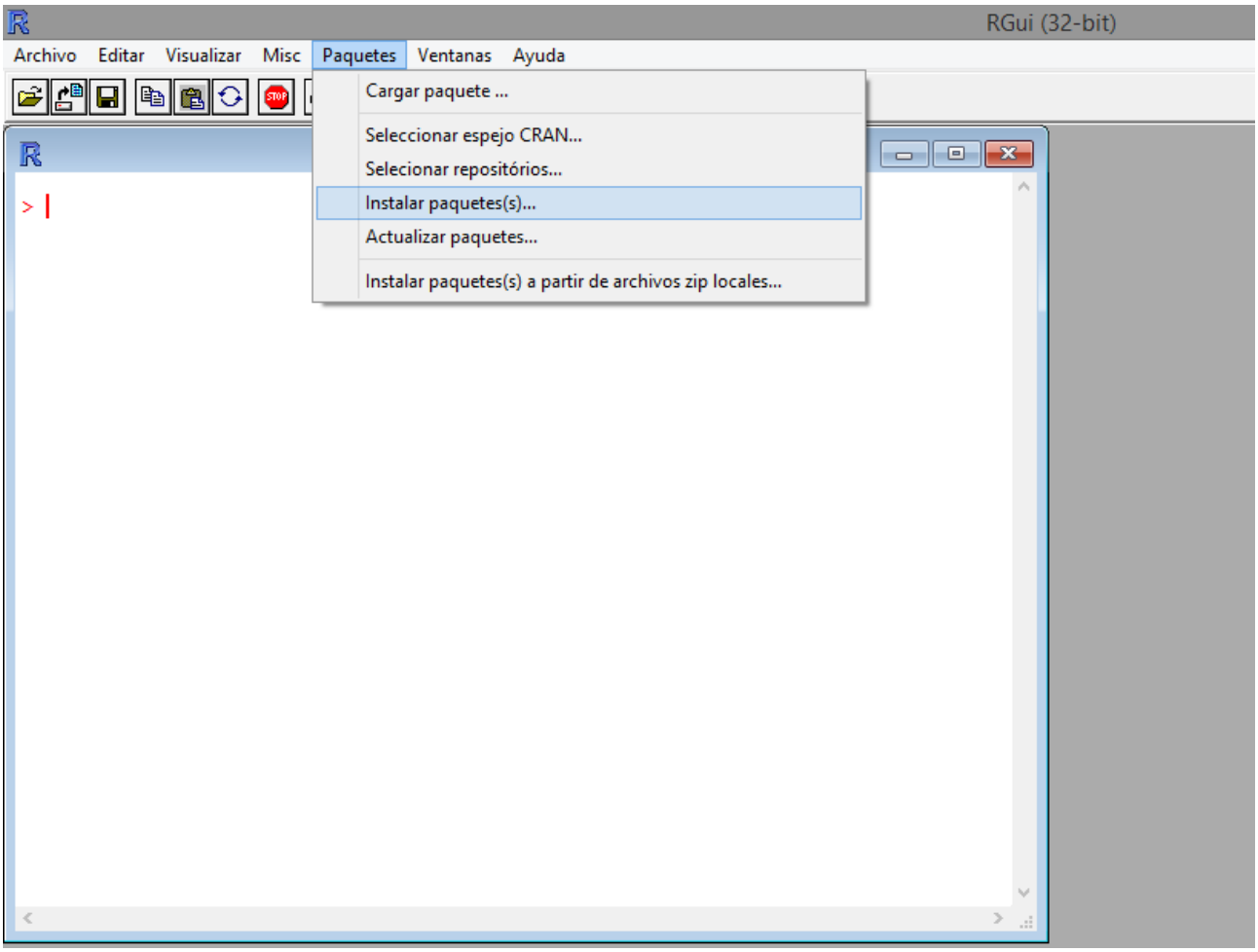

RGui (32-bit)
Archivo
Editar
Visualizar
Misc
Paquetes
Ventanas
Ayuda
Cargar paquete ...
Seleccionar espejo CRAN...
Selecionar repositórios...
Instalar paquetes(s)...
Actualizar paquetes...
Instalar paquetes(s) a partir de archivos zip locales...
>

Figura 3.6

el CPP). En el caso de que nuestro grafo si tenga arcos en la variable *Arcs* introduciremos una matriz cuadrada $A_{n\times n}$ en la que n es el número de nodos de nuestro grafo y el elemento a_{ij} indica el coste de arco que parte del nodo i y llega al nodo j, siendo este Inf en caso de que no exista dicho arco.

- La variable *Edges* nos dirá el coste de todas las aristas presentes en el grafo. Al igual que la variable *Arcs*, por defecto toma el valor "*nomatrix*", el cual solo nos interesa en el caso de que no tengamos arcos dirigidos (lo que solo ocurre en el DCPP). En el caso de que nuestro grafo si tenga aristas en la variable *Edges* introduciremos una matriz cuadrada y simétrica $E_{n\times n}$ en la que n es el número de nodos de nuestro grafo y el elemento e_{ij} indica el coste la arista que une el nodo i y el nodo j, siendo este Inf en caso de que no exista dicha arista.

- La variable MA nos indicara el número de copias que hay de cada arco dirigido, o lo que es lo mismo, cuantas veces recorreremos cada arco. Al igual que las variables anteriores por defecto toma el valor "*nomatrix*", sin embargo este valor nos puede indicar dos cosas, en caso de que la variable *Arcs* sea una matriz la función interpretará que en el grafo solo tendremos una copia de cada uno de ellos, y en caso de que no halla arcos dirigidos esta variable deberá tomar el valor "*nomatrix*" obligadamente. En caso de que tengamos arcos dirigidos y alguno de ellos se repita, tendremos que introducir en MA una matriz cuadrada $MA_{n\times n}$ en la que n es el número de nodos de nuestro grafo y el elemento ma_{ij} indica el número de copias que tenemos del arco que parte de i y llega hasta j, pudiendo ser este 0 en caso de que no tengamos arco o este no sea un arco requerido.

- La variable ME nos indicara el número de copias que hay de cada arista, o lo que es lo mismo, cuantas veces recorreremos cada arista, y su interpretación es análoga ala de la variable MA.

- Las variables *mixed*1 y *mixed*2 solo son relevantes en el caso de que el grafo sea mixto. Por defecto toman el valor "*yes*", por lo que el algoritmo aplicara tanto el algoritmo mixed1 como el algoritmo mixed2 y nos devolverá la solución con el menor coste entre ellas. En caso de que ambas tengan el mismo coste nos devolverá la solución obtenida con el mixed 1. En caso alguna de ellas tome un valor diferente a "*yes*" la función no aplicara dicho algoritmo. Téngase en cuenta que si estemos ante un grafo mixto que no podamos resolver con algún algoritmo polinómico (Un grafo que no sea par) sera necesario aplicar o bien el algoritmo mixed 1 o bien el mixed 2, por lo que si ninguna de las dos variables toma el valor "*yes*" la función no podrá resolver el problema.

La función solverChinPost nos devolverá tres variables distintas, *route*, *cost* y *solution*. *route* nos indica los nodos que vamos visitando y el orden en el que los visitaremos. *cost* nos indica el coste de la ruta. *solution* nos indicará si la solución es exacta o no en función del tipo de problema al que nos enfrentemos. En el ejemplo del siguiente apartado veremos estas variables mas en detalle.

Cabe destacar que la propia función detectará algunos de los posibles errores que pueda cometer en usuario al introducir los datos:

- Cuando el usuario no introduzca ni matriz de coste de los arcos ni de las aristas, ya que no tendríamos un grafo que resolver.

- Si alguna de las matrices de coste no fuera cuadrada, dado que en ese caso no estaríamos ante una matriz de costes. Este error es muy común cuando al definir la matriz coloquemos mal el número de columnas o de filas.

- Cuando la matriz de costes de las aristas (en caso de haberla) no sea simétrica, dado que estaríamos ante un problema del viento, el cual no es materia de este proyecto.

- Si alguno de los costes, tanto de los arcos como de las aristas fuera infinito, dado que era condición necesaria para que existiera solución optima que los costes fueran no negativos.

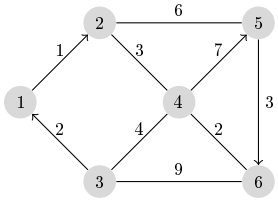
- Se considerará error que algún arco o arista tenga coste infinito, y esto puede parecer extraño, dado que cuando en la matriz de costes poníamos un infinito se daba por hecho que no había arco o arista entre los

nodos correspondientes. Sin embargo, esto entra en conflicto con las matrices de multiplicidades, que si bien no son obligatorias, podemos introducirlas (sobre todo cuando un arco o arista se deba recorrer mas de una vez), y si podría nos dará problemas si un elemento de una matriz de multiplicidades es distinto de 0 y su coste asociado es infinito.

- Por último, obtendremos un error si en un grafo mixto que no sea par tanto la variable mixed1 como variable la mixed2 son distintas de "*yes*"

3.3.3. Ejemplo función solverChinPost.

Tomemos como ejemplo el siguiente grafo:



Para resolverlo comenzaremos definiendo las matrices de coste, con lo que en R tendremos:

```
> Arcs<-matrix(c(Inf,1,Inf,Inf,Inf,Inf,Inf,Inf,Inf,Inf,Inf,Inf,2,Inf,Inf,Inf,Inf,Inf,
+              Inf,Inf,Inf,7,Inf,Inf,Inf,Inf,Inf,Inf,Inf,3,Inf,Inf,Inf,Inf,Inf,Inf),byrow=T,nrow=6)
> Arcs

     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]  Inf    1  Inf  Inf  Inf  Inf
[2,]  Inf  Inf  Inf  Inf  Inf  Inf
[3,]    2  Inf  Inf  Inf  Inf  Inf
[4,]  Inf  Inf  Inf  Inf    7  Inf
[5,]  Inf  Inf  Inf  Inf  Inf    3
[6,]  Inf  Inf  Inf  Inf  Inf  Inf
```

```
> Edges<-matrix(c(Inf,Inf,Inf,Inf,Inf,Inf,Inf,Inf,Inf,3,6,Inf,Inf,Inf,Inf,4,Inf,9,
+              Inf,3,4,Inf,Inf,2,Inf,6,Inf,Inf,Inf,Inf,Inf,Inf,9,2,Inf,Inf),byrow=T,nrow=6)
> Edges

     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]  Inf  Inf  Inf  Inf  Inf  Inf
[2,]  Inf  Inf  Inf    3    6  Inf
[3,]  Inf  Inf  Inf    4  Inf    9
[4,]  Inf    3    4  Inf  Inf    2
[5,]  Inf    6  Inf  Inf  Inf  Inf
[6,]  Inf  Inf    9    2  Inf  Inf
```

En este caso no seria necesario definir las matrices MA y ME, dado que todos los arcos y aristas son requeridos, y solo necesitamos recorrerlos una vez. Nosotros las definiremos para que se vea como se haría en el caso de querer definir las.

```
> MA<-matrix(c(0,1,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,
+              0,0,0,0,1,0,0,0,0,0,0,1,0,0,0,0,0,0),byrow=T,nrow=6)
> MA
```

```
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]    0    1    0    0    0    0
[2,]    0    0    0    0    0    0
[3,]    1    0    0    0    0    0
[4,]    0    0    0    0    1    0
[5,]    0    0    0    0    0    1
[6,]    0    0    0    0    0    0
```

```
> ME<-matrix(c(0,0,0,0,0,0,0,0,0,1,1,0,0,0,0,1,0,1,
+              0,1,1,0,0,1,0,1,0,0,0,0,0,0,1,1,0,0),byrow=T,nrow=6)
> ME
```

```
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]    0    0    0    0    0    0
[2,]    0    0    0    1    1    0
[3,]    0    0    0    1    0    1
[4,]    0    1    1    0    0    1
[5,]    0    1    0    0    0    0
[6,]    0    0    1    1    0    0
```

Una vez definidas todas las variables que necesitamos podemos proceder a utilizar la función, con lo que obtenemos:

```
> solverChinPost(Arcs=Arcs, Edges=Edges,MA=MA,ME=ME)
```

```
$route
 [1] 1 2 4 5 6 4 3 1 2 5 6 3 1

$cost
[1] 43

$solution
[1] "We do not know if it is optimal solution, because this is a odd degree MCPP."
```

Con lo que obtenemos que nuestro ciclo es $C = \{(1,2),(2,4),(4,5),(5,6),(6,4),(4,3),(3,1),(1,2),(2,5),(5,6),(6,3),(3,1)\}$. Ademas vemos que el coste es 43, y que no sabemos si la solución es óptima, al tratarse de un grafo mixto de grado impar.

El programa también realiza una representación del grafo, aún que esta solo es útil si estamos ante un grafo pequeño, , dado que si tiene muchos nodos queda una representación muy engorrosa e incluso imposible de interpretar.

Téngase en cuenta que esta función solo necesita que introduzcamos la matriz de costes de los arcos (análogo para las aristas) en el caso de que halla arcos en el grafo, si nos hubiera no sería necesario introducir dicha matriz.

3.3.4. Errores función solveChinPost.

no existe grafo

```
> A<-matrix(c(Inf,Inf,8,Inf,Inf,Inf,Inf,Inf,Inf,11,Inf,Inf,8,Inf,Inf,Inf,Inf,Inf,
+             Inf,11,Inf,Inf,2,Inf,Inf,Inf,Inf,2,Inf,7,Inf,Inf,Inf,Inf,7,Inf),byrow=T,nrow=6)
> A
```

```
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]  Inf  Inf    8  Inf  Inf  Inf
[2,]  Inf  Inf  Inf   11  Inf  Inf
[3,]    8  Inf  Inf  Inf  Inf  Inf
[4,]  Inf   11  Inf  Inf    2  Inf
[5,]  Inf  Inf  Inf    2  Inf    7
[6,]  Inf  Inf  Inf  Inf    7  Inf
```

```
> solverChinPost(MA=A)
```

```
[1] "error: There is not a graph"
```

Matrices no cuadradas.

```
> A<-matrix(c(1,1,1,1,1,1),byrow=T,nrow=2)
> A
```

```
     [,1] [,2] [,3]
[1,]    1    1    1
[2,]    1    1    1
```

```
solverChinPost(Edges=A)
```

```
[1] "error: Edges cost matrix is not squared"
```

```
> solverChinPost(Arcs=A)
```

```
[1] "error: Arcs cost matrix is not squared"
```

Matriz de costes de las aristas no simétrica.

```
> A<-matrix(c(Inf,1,6,Inf,Inf,Inf,1,Inf,Inf,11,1,Inf,8,Inf,Inf,4,Inf,9,
+             Inf,11,4,Inf,2,1,Inf,1,Inf,2,Inf,7,Inf,Inf,9,1,7,Inf),byrow=T,nrow=6)
> A
```

```
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]  Inf    1    6  Inf  Inf  Inf
[2,]    1  Inf  Inf   11    1  Inf
[3,]    8  Inf  Inf    4  Inf    9
[4,]  Inf   11    4  Inf    2    1
[5,]  Inf    1  Inf    2  Inf    7
[6,]  Inf  Inf    9    1    7  Inf
```

```
> solverChinPost(Edges=A)
```

```
[1] "error: Edges cost matrix is not simetric"
```

Algún coste negativo.

```
> A<-matrix(c(Inf,1,-8,Inf,Inf,Inf,1,Inf,Inf,11,1,Inf,-8,Inf,Inf,4,Inf,9,
+ Inf,11,4,Inf,2,1,Inf,1,Inf,2,Inf,7,Inf,Inf,9,1,7,Inf),byrow=T,nrow=6)
> A

     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]  Inf    1   -8  Inf  Inf  Inf
[2,]    1  Inf  Inf   11    1  Inf
[3,]   -8  Inf  Inf    4  Inf    9
[4,]  Inf   11    4  Inf    2    1
[5,]  Inf    1  Inf    2  Inf    7
[6,]  Inf  Inf    9    1    7  Inf

> solverChinPost(Edges=A)

[1] "error: There is one edge whit infinite cost"

> solverChinPost(Arcs=A)

[1] "error: There is one arc whit a negative cost"
```

Arista o arco con coste infinito

```
> A<-matrix(c(Inf,1,8,Inf,Inf,Inf,1,Inf,Inf,11,1,Inf,8,Inf,Inf,4,Inf,9,
+ Inf,11,4,Inf,2,1,Inf,1,Inf,2,Inf,7,Inf,Inf,9,1,7,Inf),byrow=T,nrow=6)
> A

     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]  Inf    1    8  Inf  Inf  Inf
[2,]    1  Inf  Inf   11    1  Inf
[3,]    8  Inf  Inf    4  Inf    9
[4,]  Inf   11    4  Inf    2    1
[5,]  Inf    1  Inf    2  Inf    7
[6,]  Inf  Inf    9    1    7  Inf

> M<-matrix(c(0,1,1,1,0,0,1,0,0,1,1,0,1,0,0,1,0,1,
+ 0,1,1,0,1,1,0,1,0,1,0,1,0,0,1,1,1,0),byrow=T,nrow=6)
> M

     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]    0    1    1    1    0    0
[2,]    1    0    0    1    1    0
[3,]    1    0    0    1    0    1
[4,]    0    1    1    0    1    1
[5,]    0    1    0    1    0    1
[6,]    0    0    1    1    1    0

> solverChinPost(Edges=A,ME=M)

[1] "error: There is one edge whit infinite cost"
```

mixed1 y mixed2 distitos de zes"

```
> A<-matrix(c(Inf,1,3,Inf,Inf,Inf,Inf,Inf,Inf,11,Inf,4,8,Inf,Inf,Inf,Inf,Inf,
+             Inf,Inf,Inf,Inf,Inf,1,Inf,Inf,Inf,Inf,Inf,Inf,Inf,Inf,Inf,Inf,Inf,Inf),byrow=T,nrow=6)
> A

     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]  Inf    1    3  Inf  Inf  Inf
[2,]  Inf  Inf  Inf   11  Inf    4
[3,]    8  Inf  Inf  Inf  Inf  Inf
[4,]  Inf  Inf  Inf  Inf  Inf    1
[5,]  Inf  Inf  Inf  Inf  Inf  Inf
[6,]  Inf  Inf  Inf  Inf  Inf  Inf

> E<-matrix(c(Inf,Inf,Inf,Inf,Inf,Inf,Inf,Inf,6,Inf,1,Inf,Inf,6,Inf,4,Inf,9,
+             Inf,Inf,4,Inf,2,Inf,Inf,1,Inf,2,Inf,Inf,Inf,Inf,9,Inf,Inf,Inf),byrow=T,nrow=6)
> E

     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]  Inf  Inf  Inf  Inf  Inf  Inf
[2,]  Inf  Inf    6  Inf    1  Inf
[3,]  Inf    6  Inf    4  Inf    9
[4,]  Inf  Inf    4  Inf    2  Inf
[5,]  Inf    1  Inf    2  Inf  Inf
[6,]  Inf  Inf    9  Inf  Inf  Inf

> solverChinPost(Arcs=A,Edges=E,mixed1="no",mixed2="no")

[1] "mixed1 or mixed2 must be yes, because this is a odd degree MCPP"
```

Tengase en cuenta que para que no se aplique el algoritmo mixed1 (análogo para mixed2) es suficiente con que $mixed1$ sea distinto de "yes", no es necesario que sea igual a "no". Veamos que ocurre si con las mismas matrices A y E le damos a las variables $mixed1$ y $mixed2$ distintos valores.

```
> solverChinPost(Arcs=A,Edges=E,mixed1="coche",mixed2="perro")

[1] "mixed1 or mixed2 must be yes, because this is a odd degree MCPP"
```

3.3.5. Función costrut.

Casi todas las funciones que utiliza el paquete están englobadas dentro de la funciónsolverChinPost, sin embargo tendremos una que no, y a la que llamaremos por separado. Se trata de la función *costrut*, la cual usaremos cuando ya tengamos una ruta sobre el grafo y nos interese calcular su coste. Téngase en cuenta que esta función no solo calcula el coste de los ciclos, si no también de caminos. Veamos como esta definida esta función:

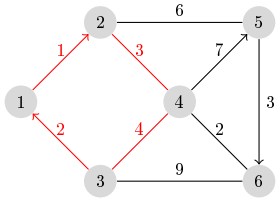
$$costrut(ruta, Arcs, Edges)$$

- En la variable *ruta* introduciremos la ruta de la que queremos calcular su coste, siendo *ruta* un vector con los nodos que recorremos y en el orden que los recorremos (exactamente el mismo formato en el que nos la devuelve la función *solverChinPost*).
- En la variable *Arcs* introduciremos la matriz de coste de los arcos, y en la variable *Edges* introduciremos la matriz de coste de las aristas, ambas de forma análoga a como lo hacíamos en la función *solverChinPost*.

La función unicamente nos devuelve la variable *cost*, la cual nos indica el coste de la ruta introducida.

3.3.6. Ejemplo función costrut.

Veamos ahora como aplicar esta función, para ello tomaremos una ruta sencilla, que seria recorrer los arcos pintados en rojo una única vez:



Para ello definiremos la ruta como:

```
> route=c(1,2,4,3,1)
```

Definiremos los arcos y aristas del grafo:

```
> Arcs<-matrix(c(Inf,1,Inf,Inf,Inf,Inf,Inf,Inf,Inf,Inf,Inf,Inf,2,Inf,Inf,Inf,Inf,Inf,
+              Inf,Inf,Inf,7,Inf,Inf,Inf,Inf,Inf,Inf,Inf,3,Inf,Inf,Inf,Inf,Inf,Inf),byrow=T,nrow=6)
> Edges<-matrix(c(Inf,Inf,Inf,Inf,Inf,Inf,Inf,Inf,Inf,3,6,Inf,Inf,Inf,Inf,4,Inf,9,
+              Inf,3,4,Inf,Inf,2,Inf,6,Inf,Inf,Inf,Inf,Inf,Inf,9,2,Inf,Inf),byrow=T,nrow=6)
```

Y le aplicaremos la función obteniendo:

```
> costrut(route=route,Arcs=Arcs,Edges=Edges)


$cost
[1] 10
```

Con lo que vemos que el coste es 10, lo cual es fácil de comprobar en la representación del grafo.

3.3.7. Análisis del paquete.

Tras comprobar el paquete con diferentes grafos, vemos que este funciona bien y no comete errores, o al menos no los cometió en ninguno de los grafos a los que se lo aplicamos. Veamos brevemente algunos de los tiempos computacionales que obtuvimos al ejecutarlo. Tengase en cuenta que todas las pruebas se hicieron con un ordenador con un procesador Intel(R) Core(TM) i5-4200U CPU @ 1.60 GHz 2.30 GHz. En la siguiente tabla se muestran tiempos necesarios:

Nº nodos	Nº arcos	Nº aristas	Tiempo	algoritmo exacto
6	0	9	0.03	si
6	14	0	0.03	si
6	4	5	0.03	si
6	4	4	0.03	no
6	5	6	0.03	no
6	4	5	0.03	no
61	0	216	0.27	si
61	52	164	0.31	si
27	0	702	0.16	si

Téngase en cuenta que estos tiempos pueden variar según el ordenador tenga otras tareas en segundo plano o no. Para analizar el paquete en detalle habríamos necesitado mas bases de datos, en concreto algunas de tamaño mayor, pero no disponíamos de ellas, por lo que solo pudimos hacer este análisis, en que obtuvimos unos resultados bastante satisfactorios.

3.4. Ejemplo problema real.

Vamos a introducir un problema real, que se da en la ciudad de Ourense y que resolveremos usando nuestro paquete de R y los datos cedidos por la empresa Ecourense. Todos hemos visto muchas veces por las calles de nuestra ciudad barredoras, las cuales recorren las calles al mismo tiempo que las limpian, las cuales suelen ser similares a la mostrada en la figura 3.7, y sobre ellas versará nuestro ejemplo. El problema planteado por Ecourense consiste en diseñar una ruta óptima para la limpieza del casco antiguo de Ourense, dado que es una zona formada por unas calles con una características diferentes al resto de calles de Ourense, siendo estas las únicas que se limpian con las barredoras a diario, y siendo la mayoría de ellas peatonales, por lo que la empresa decidió crear una ruta propia para estas calles independiente del resto de la ciudad. La zona en cuestión es un conjunto de calles que suman una distancia de 8255 metros. Sin embargo debemos tener en cuenta que todas las calles necesitan ser recorridas un mínimo de 2 veces, y en algunos casos 4 o incluso 6 veces en el caso de la calle del Paseo, con lo que realmente tenemos que limpiar 20930 metros.

Figura 3.7: Barredora de la ciudad de Ourense.

Debido a que las barredoras no tienen que respetar los sentidos de circulación de las calles, es fácil ver que estamos antes un problema del cartero chino no dirigido, lo cual es una muy buena noticia para la empresa, ya que en este caso podemos calcular la solución exacta. Dado que estamos ante un grafo con 61 nodos y 216 aristas, representar el grafo se vuelve un poco complicado, por lo que decidimos que las aristas que se repiten mas de una vez las representaremos como una única arista, usando diferentes colores según el número de veces que se repita dicha arista. Sabiendo esto decidimos representar el grafo asociado sobre el plano de la ciudad, tal

como se ve en la figura 3.8, y decidimos marcar en verde las aristas que se repitan solo dos veces, azul las que se repitan cuatro veces y rojo las que se repitan seis veces.

Al representar las aristas nos encontramos con que nuestro grafo es par (todas las aristas se repiten un número par de veces), con lo que no solo encontraremos la solución óptima, sino que está será un ciclo eureliano, por lo que nuestro operario solo tendrá que recorrer una calle cuando la esté limpiando.

Hecho esto tendríamos que calcular la matriz de costes y la matriz de multiplicidades, pero al tratarse de matrices 61×61 hemos decidido no incluirlas en este documento, por lo que las hubo que almacenar directamente en sendos archivos txt.

El hecho de que el ciclo sea eureliano aparentemente no tiene importancia, dado que lo que nosotros buscamos es que la solución sea óptima, sin embargo, en este tipo de problemas este es un hecho de gran importancia, dado que estas barredoras tienen una velocidad muy baja, y una de las cosas que menos gusta a los operarios es estar conduciéndolas sin estar barriendo, ya que tienen sensación de que no avanzan. En nuestro caso el operario, una vez que llegue a la zona a limpiar, estará todo el tiempo limpiando, con lo que además del ahorro de tiempo que supone tener la ruta óptima, conseguiremos que nuestros trabajadores se aburran menos en el trabajo, lo que se traduce en no solo mayor productividad, sino una mayor satisfacción.

Veamos ahora cual será la ruta a seguir por nuestro operario. Tras ejecutar nuestro paquete obtenemos la siguiente salida:

```
> solverChinPost(Edges=A,ME=ME)


$route
  [1]  1  2  3  2  3  2 11 12 13 12 18 19 18 19 18 21 22 21 22 21 32 33 34 51 48
 [26] 47 48 49 61 49 48 49 48 51 52 53 54 55 56 55 45 59 45 55 54 56 45 56 54 53
 [51] 46 53 52 50 49 50 52 51 34 60 34 60 34 33 60 33 32 61 32 21 18 12 11 12 11
 [76]  2  1  3  4  5  6  7  6  9 10  9 13 14 13 19 20 19 20 19 22 23 24 23 24 23
[101] 27 28 27 36 27 23 22 23 22 26 27 26 35 60 35 60 35 26 22 19 13 19 13 19 13
[126]  9 13  9 13  9  6  5  7 10 14 15 16 17 16 15 17 15 14 20 24 25 30 43 44 43
[151] 30 25 31 44 58 44 31 57 31 25 57 25 24 28 29 28 37 38 39 38 40 38 37 40 41
[176] 42 58 59 58 42 59 42 41 43 41 40 37 28 24 29 30 29 24 20 14 20 14 10 14 10
[201]  7  5  4  8  4  8  4  3  8  9  8 11  8 11  8  3  1

$cost
[1] 20930

$solution
[1] "Optimal solution, because this is a CPP."
```

Interpretemos ahora las tres variables que nos devuelve la función. La primera es *route*, que tal como ya mencionamos en el apartado anterior nos devuelve la ruta que debemos seguir, indicándonos los nodos que vamos visitando y el orden en el que lo hacemos. En este caso comenzamos por el nodo 1, luego el 2, el 3, el 2... Si esto lo queremos traducir a aristas sería muy sencillo, simplemente empezamos por la arista (1,2), continuamos con la (2,3), seguido de la (3,2), y así sucesivamente hasta recorrer todos los nodos en el orden que nos indica la variable. Tengamos en cuenta que no sabemos si nuestro operario saldrá del nodo 1 o de otro nodo distinto, sin embargo esto no es relevante, dado que al tratarse de un ciclo la solución será la misma. La segunda variable es *coste*, la cual nos indica el coste de nuestro tour, en este caso 20930 metros, que es lo que esperábamos dado que estamos ante un grafo no dirigido de grado par (unicamente recorremos las calles cuando las estamos limpiando, por lo que el coste de nuestro ciclo será el mismo que el de las calles a limpiar). La ultima variable, *solucion*, nos indica lo que ya sabíamos, que nuestra solución es exacta dado que estamos ante un problema del cartero chino ni dirigido.

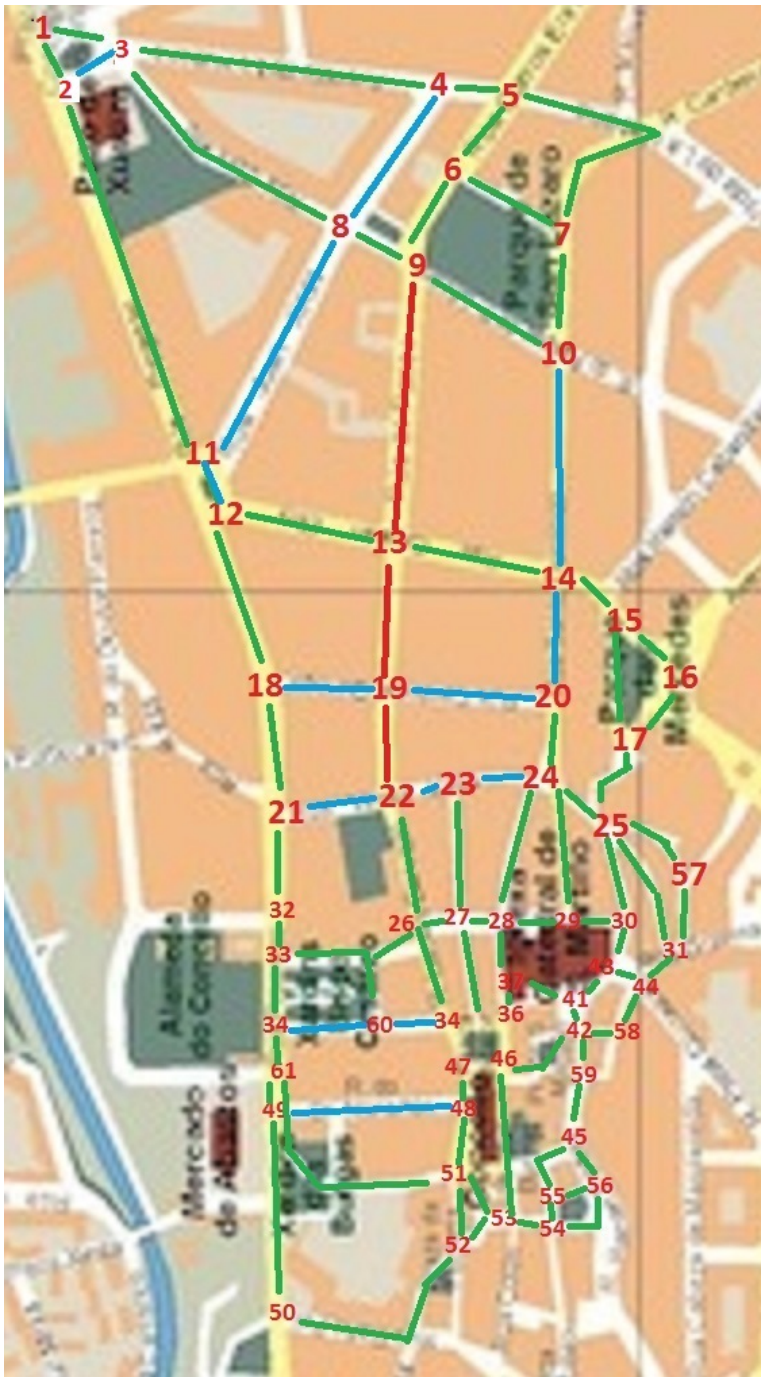


Figura 3.8: Grafo sobre el plano.

No ha sido posible comparar nuestra ruta con la seguida hasta el día de hoy por Ecourense dado que no existía una ruta preestablecida. Sin embargo, si podemos decir que nuestra ruta tiene un coste total de 20930 litros de combustible, dado que el consumo medio de una barredora es 100 litros a los 100 km, además de que ninguna de las que siguieron en algún momento pasado tiene un coste menor a la propuesta por nosotros.

Para terminar, debemos tener en cuenta que cuando se cree una ruta para un problema real los resultados teóricos no siempre se pueden aplicar a la vida real, dado que es posible que nuestro operario no pueda recorrer alguna calle (actos de vandalismo, accidentes, carreteras cortadas por obras, etc), que el coste de la arista varíe (debido a tener obstáculos en la vía, como coches mal aparcados, desperfectos en las aceras, etc), o que tenga que recorrer alguna arista distinta a estas (avería en la maquina que tenía que limpiarla, alguna emergencia que requiera limpiarla el día que no lo toca, etc). Por todos estos motivos no debemos despreciar la capacidad de nuestros operarios para improvisar y solucionar todos los posibles imprevistos que se le puedan presentar en el día a día. Además esta solución esta sujeta a posibles variaciones futuras, dado que una ciudad no es algo estático, y aún que no tenemos indicios de que alguna de estas calles vallan a desaparecer, o aparezcan otras nuevas, si es mas que posible que la demanda cambie, con lo que alguna calle la habría que recorrer menos o más veces.

3.4.1. Lineas futuras para este problema.

Si intentamos ver este problema a mayor escala podemos pensar que realmente a la hora de limpiar Ourense esta empresa esta ante un problema de los m-carteros, dado que esta empresa tiene que limpiar todas las calles y dispone de m barredoras (número que desconocemos), las cuales tienen una capacidad (el tiempo que los operarios pueden estar barriendo). Sin embargo, su problema se complicaría mucho, dado que las barredoras no pueden limpiar todas las calles de Ourense (escaleras, calles estrechas, etc), por lo que tienen que estar apoyadas por barrenderos que recorran la ciudad a pie. También debemos tener en cuenta que para las aristas que puedan ser recorridas tanto por barrenderos como por barredoras, el coste no es el mismo para ambos, en nuestro problema no se tubo en cuenta porque el coste en metros es proporcional al coste en combustible y tiempo del operario, sin embargo, sabemos que los barrenderos aún sin implicar un gasto de combustible, por norma general tienen un coste mayor a las barredoras, ya que su coste en tiempo es significativamente mayor al de las barredoras. Además, no todas las calles se limpian el mismo número de días a la semana, por lo que tendríamos que añadirle al problema de los m-carteros restricciones de tiempo, y saber que algunos tours tendrían que ser hechos por barrenderos y otros por barredoras, según los arcos que estén involucrados en dicho tour. Por otra parte tenemos que tener en cuenta que ciertas zonas ya están delimitadas simplemente por políticas de la empresa, por lo que salvo que cambiaran de opinión, en esas zonas tendríamos un problema del cartero chino, y solo resolveríamos el problema de los m-carteros en la ciudad que aún no está delimitada. Por lo que aún que nuestra ruta se mantendría invariante, dado que esa zona está delimitada por políticas de la empresa, hay otras zonas que tendríamos que resolver usando un problema mucho mas complicado que este.

3.4.2. Variaciones del problema real.

Tras resolver el problema con los datos reales, vimos que se trata de un problema demasiado sencillo, por lo que aún que es útil como ejemplo, sabemos que nos podríamos haber topado con uno mucho mas complicado, motivo por le cual haremos tres pequeñas variaciones para ver como funcionaría nuestro paquete frente al mismo problema si los datos fueran mas complicados.

Grafo dirigido de grado impar

Nuestra primera modificación consistirá en aplicarle el algoritmo a un grafo de grado impar, para ello tomaremos el mismo grafo, pero no repetiremos ninguna arista, con lo que tendremos 61 nodos y 80 aristas, y ahora solo tendríamos que limpiar 8255 metros de calles. Si nos fijamos en la figura 3.8 vemos que hay muchos nodos de grado impar, con lo que problema ya se complica un poco (recordemos que en estos casos lo primero que hay que hacer es duplicar aristas para hacer el grafo par). Usando R obtenemos:

```
> solverChinPost(Edges=A,ME=ME)
```

```
$route
  [1]  1  2  3  4  3  8  4  5  6  5  7 10  7  6  9 10 14 15 16 17 15 14 20 14 13
 [26]  9  8 11 12 13 19 20 24 23 22 19 18 21 22 26 27 23 24 25 31 44 31 57 25 30
 [51] 29 24 28 29 30 43 41 42 58 42 59 45 55 56 55 54 53 54 56 45 59 58 44 43 41
 [76] 40 37 38 39 38 40 37 28 27 36 35 26 35 60 33 32 33 34 60 34 51 52 51 48 47
[101] 46 53 52 50 49 48 49 61 32 21 18 12 11  2  3  1

$cost
[1] 9936

$solution
[1] "Optimal solution, because this is a CPP."
```

Vemos que ahora nuestra distancia es 9936, y en este caso estamos recorriendo mas distancia de la que necesitamos limpiar. Esto se debe a que al ser un grafo de grado impar tendremos que recorrer algunas aristas mas veces de las necesarias (recordemos que el primer paso consistía en transformar el grafo en un grafo de grado par).

Grafo mixto grado par

Nuestra segunda modificación consistirá en usar un grafo mixto, pero de grado par, dado que en ese caso podemos calcular la solución exacta. Para conseguir el grafo mixto de grado par lo único que haremos será tomar el grafo de nuestro problema real y ponerle dirección a algunas aristas (transformándolas así en arcos dirigidos y manteniendo el grado par del grafo). En cualquier otro problema simplemente habríamos tomado el sentido de las calles, suponiendo que la barredora tuviera que respetarlo, sin embargo, como la mayoría de las calles de Ourense que usamos en este ejemplo son peatonales y las que no lo son tienen doble sentido, tendremos que decidir de forma arbitraria a cuales le asociaremos un único sentido. Decidimos que pondremos como calles de sentido único a la calle del Progreso (sentido de norte a sur), calle Paseo (sentido sur a norte), calle Capitán Eloy (sentido oeste a este), calle Cardenal Quiroga (sentido este a oeste), y la calle Doutor Marañón (sentido este a oeste), manteniendo el número de veces que hay que recorrer cada una de las calles. Con lo que ahora tendremos los arcos dirigidos (1,2), (2,11), (11,12), (12,18), (18,21), (21,32), (32,33), (33,34), (34,61), (61,49), (49,50), (22,19), (19,13), (13,9), (12,13), (13,14), (20,19), (19,18), (49,48). Por lo que ahora tenemos 61 nodos, 164 aristas y 52 arcos dirigidos. En la figura 3.9 vemos como quedaría el grafo, manteniendo el código de colores para la multiplicidad de los arcos y aristas. Usando R obtenemos:

```
>solverChinPost(Arcs=A,Edges=E,ME=ME,MA=MA)
```

```
$route
  [1]  1  3  8 11  2  3  4  8  4  5  6  7  6  9 13 19 22 26 22 23 22 23 22 21 18 19 19 19 19 12 11  6
 [34] 13 19 22 21 18 12 11  7 10 14 13 19 20 24 29 30 43 44 58 59 58 44 43 41 43 30 29 28 37 40 41 42
 [67] 45 56 55 56 54 53 52 51 48 49 61 49 49 51 34 33 32 21 22 21 61 60 34 33 60 34 60 34 51 32 47 48
[100] 52 50 49 52 49 61 32 21 53 46 53 54 56 45 55 54 55 45 59 42 58 42 41 40 38 40 37 38 39 38 37 28
[133] 24 28 27 36 27 28 24 25 57 31 57 25 31 44 31 25 30 25 24 23 27 26 35 60 35 60 35 26 27 23 24 23
[166] 20 14 13 19 20 14 20 14 15 17 16 17 15 16 15 14 10 14 10  9 13 19 20  5  7 10  9 13 19 20  8 11
[199]  9 13 12 11  3  4  8  9 13 12 11  3  2  1  3  8 11  2  1

$cost
[1] 20930

$solution
```

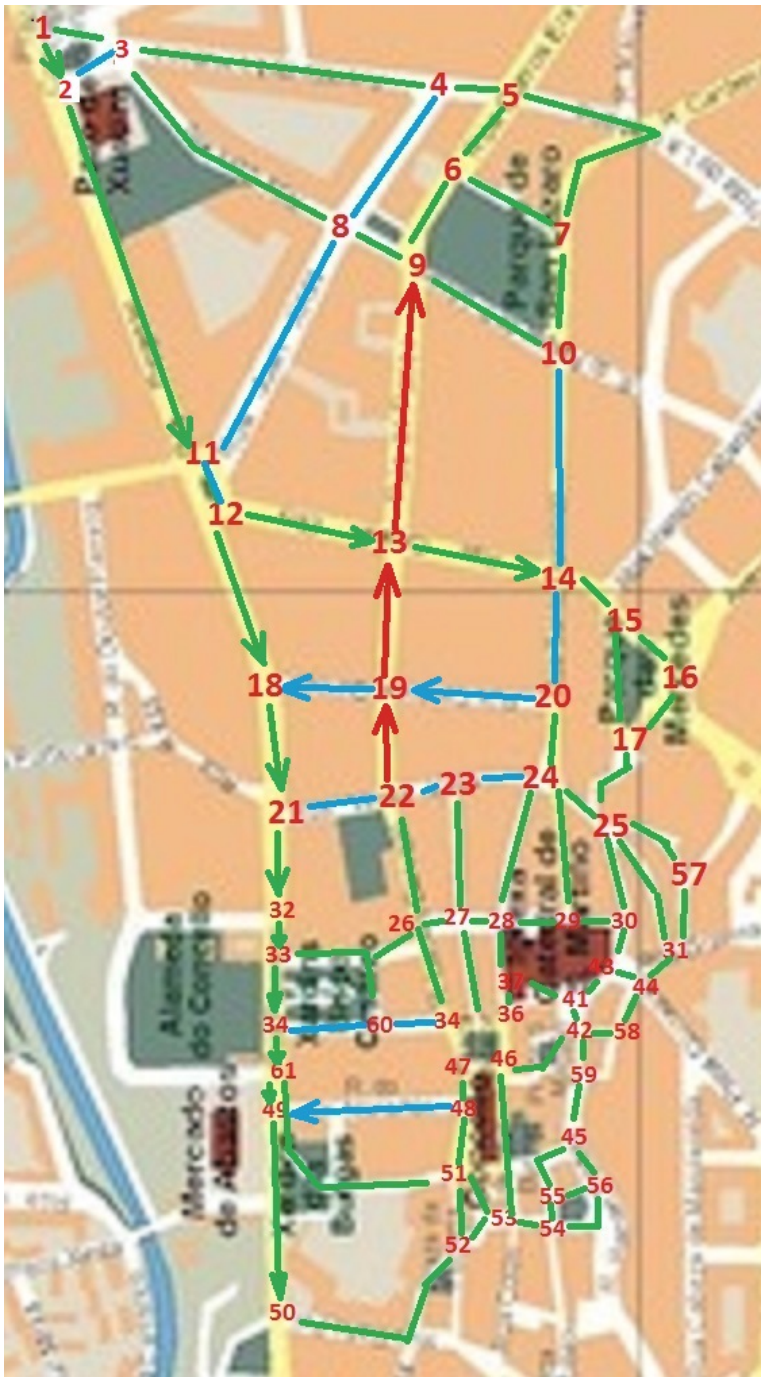


Figura 3.9: Grafo mixto sobre el plano.

```
[1] "Optimal solution, because this is a even degree MCPP."
```

Como estamos ante el MCPP de grado par hemos encontrado la solución exacta, y recorremos exactamente los 20930 metros. Véase que en este caso en particular estamos recorriendo cada arista y cada arco una sola vez, sin embargo, aunque siempre que tengamos un grafo par tendremos su solución exacta, no siempre se conseguirá sin necesidad de duplicar ningún arco ni ninguna arista.

Grafo mixto grado impar

Por último calculemos que ocurriría con un grafo mixto de orden impar, para ello tomaremos las direcciones fijadas en el apartado anterior, y resolveremos nuevamente el problema pero limpiando cada calle una sola vez. Ahora tendremos 61 nodos, 72 aristas y 18 arcos dirigidos, y solo tendremos que limpiar 8255 metros de calle. Usando R obtenemos:

```
>solverChinPost(Arcs=A,Edges=E,mixed1="yes",mixed2="no")#


$route
  [1]  1  3  4  5  7 10  7  6  5  6  9 10 14 20 14 15 17 16 15 14 13 12 11  2  3
 [26]  8  9 13 19 20 24 25 57 31 44 43 41 40 38 39 36 35 26 27 28 29 30 25 31 44
 [51] 58 59 45 55 56 55 54 53 46 47 48 49 61 32 21 22 23 24 29 30 43 41 42 58 42
 [76] 59 45 56 54 53 52 50 49 61 32 21 22 23 27 36 35 47 48 51 52 51 34 60 34 33
[101] 32 21 22 26 35 60 33 32 21 18 19 22 23 24 28 37 40 37 38 39 46 47 48 49 61
[126] 32 21 18 12 11  8  4  3  2  1

$cost
[1] 11511

$solution
[1] "We do not know if it is optimal solution, because this is a odd degree MCPP."



>solverChinPost(Arcs=A,Edges=E,mixed1="no",mixed2="yes")


$route
  [1]  1  3  4  5  7 10  7  6  5  6  9 10 14 20 14 15 17 16 15 14 13 12 11  2  3
 [26]  8  9 13 19 20 24 28 37 40 37 38 40 41 43 41 42 58 42 59 45 55 54 53 52 50
 [51] 49 61 32 21 22 26 35 60 34 33 60 33 32 21 18 19 22 23 24 29 30 43 44 58 59
 [76] 45 56 55 56 54 53 46 47 48 51 52 51 34 60 35 26 27 28 29 30 25 57 31 44 31
[101] 25 24 23 27 36 39 46 39 38 39 36 35 47 48 49 61 32 21 18 12 11  8  4  3  2
[126]  1

$cost
[1] 10635

$solution
[1] "We do not know if it is optimal solution, because this is a odd degree MCPP."
```

Y vemos que en este caso la mejor solución la obtenemos aplicándole el mixed 2, con un coste de 10635 metros, frente a los 11511 del mixed 1 (recordemos que esto no tiene porque ser así, la mejor solución puede ser una cualquiera de ellas). Véase que no sabemos si esta solución es exacta o no, sin embargo si sabemos que esta solución siempre tiene un error inferior al 50 %, y en este caso podemos afirmar que es inferior al 39'44 % ya que el coste de los arcos y aristas suma 8255, por lo que la ruta óptima tiene que tener un coste mayor a 8255 (al tratarse de un grafo impar tendremos que duplicar algún arco o arista), y dado que $\frac{11511}{8255} = 1{,}394428$ podemos afirmar que cometemos un error menor a 39'44 %.

Bibliografía

[1] Ávila, T. *Algunos problemas de rutas por arcos.* Tesis. Valencia: Universidad Politécnica de Valencia. 2014

[2] Benavent, E. y Campos, V. *Análisis heurístico para el problema del cartero rural.* Tesis. Valencia: Universidad Politécnica de Valencia. 1985

[3] Assad, A. y Golden, B.L.. *Arc Routing Methods and Applications. Handbooks of Operations Research and Management Science.* North Holland, 2000.

[4] Benavent,E., Campos, V., Corberán, Á. y Mota, E. *Problemas de rutas por arcos.* 1983.

[5] Bondy, L. y Murty,U. *Graph Theory with Applications. American Elsevier.* New York. 1976.

[6] Corberán,Á., Martí, R. y Sanchis, J.M.. *A grasp heuristic for the mixed Chinese postman problem.* European Journal of Operational Research. 2002.

[7] Corberán,Á. y Prins,C. *Recent results on arc routing problems: An annotated bibliography.* 2010.

[8] Dror,M. *Arc Routing: Theory, Solutions and Applications.* Kluwer Academic Publishers, 2000.

[9] Edmonds,J. y Johnson,E. *Matching, euler tours and chinese postman..*1973

[10] Ford, L.R. Fulkerson, D.R. *Flows in networks.*1962