Universida de Vigo

Trabajo de Fin de Máster

Optimal Trees

Programación en R de problemas de búsqueda de árboles óptimos

Alumno Fontenla Cadavid, Manuel
Universidade de Santiago de Compostela

Directoras Lorenzo Freire, Silvia María
Universidade da Coruña

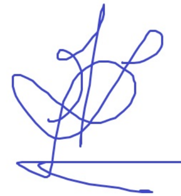
Carpente Rodríguez, María Luisa
Universidade da Coruña

Máster en Técnicas Estadísticas
Curso 2013 - 2014

El presente documento, que tiene como título **"Optimal Trees: Programación en R de problemas de búsqueda de árboles óptimos"**, constituye la memoria del Trabajo de Fin de Máster realizado por el alumno Manuel Fontenla Cadavid para el Máster en Técnicas Estadísticas. Ha sido realizado bajo la dirección de Silvia María Lorenzo Freire y María Luisa Carpente Rodríguez, quienes lo consideran terminado y dan su conformidad para que sea presentado.

Firmado:

Silvia María Lorenzo Freire



María Luisa Carpente Rodríguez

En A Coruña, a 03 de septiembre de 2014.

Índice general

1. Introducción **5**
1.1. Objetivos y estructura 6
1.2. Herramientas utilizadas 7
1.3. Instrucciones 9

2. Optrees: búsqueda de árboles óptimos **11**
2.1. Introducción 11
2.1.1. Definiciones comunes 11
2.1.2. Llevando grafos a R 15
2.2. El problema del árbol de coste mínimo 20
2.2.1. Algoritmo de Prim 21
2.2.2. Algoritmo de Kruskal 23
2.2.3. Algoritmo de Borůvka 25
2.2.4. getMinimumSpanningTree 27
2.3. El problema de la arborescencia de coste mínimo 28
2.3.1. Algoritmo de Edmonds 29
2.3.2. getMinimumArborescence 33
2.4. El problema del árbol (o arborescencia) del camino más corto 35
2.4.1. Algoritmo de Dijkstra 36
2.4.2. Algoritmo de Bellman-Ford 38
2.4.3. getShortestPathTree 40
2.5. El problema del árbol del corte minimal 41
2.5.1. Algoritmo de Gusfield 42
2.5.2. getMinimumCutTree 46

3. Cooptrees: cooperación en árboles óptimos **47**
3.1. Introducción 47
3.1.1. Definiciones comunes 48
3.2. Cooperación en problemas de árboles de coste mínimo 50
3.2.1. La regla de Bird 51
3.2.2. La regla de Dutta-Kar 51

3.2.3. La regla de Kar 52
3.2.4. La regla ERO 54
3.2.5. mstCooperative 57
3.3. Cooperación en problemas de arborescencias de coste mínimo 59
3.3.1. La regla de Bird 60
3.3.2. La regla ERO 60
3.3.3. maCooperative 62

4. OptreesGUI: aplicación web para optrees **65**
4.1. Introducción 65
4.2. Desarrollando optreesGUI 66
4.2.1. Estructura de la web 66
4.2.2. Componentes de la web 67
4.3. Manual de usuario 73

5. Conclusiones y trabajo futuro **79**

Anexos **81**

A. Ejemplos Algoritmos **81**
A.1. Algoritmo de Prim 83
A.2. Algoritmo de Kruskal 85
A.3. Algoritmo de Borůvka 87
A.4. Algoritmo de Edmonds 89
A.5. Algoritmo de Dijkstra 93
A.6. Algoritmo de Bellman-Ford 95
A.7. Algoritmo de Gusfield 97

B. Manuales Paquetes **101**
Package ‘optrees’ 103
Package ‘cooptrees’ 135

Bibliografía **153**

Capítulo 1

Introducción

"A hombros de gigantes"

Cuando navegamos por internet, al consultar una ruta en el navegador GPS, al buscar billetes de avión, incluso al abrir un grifo o encender una simple bombilla. En todos esos actos cotidianos interviene, de una u otra forma, la teoría de grafos y la determinación de árboles óptimos. Convertidos en estructuras fundamentales para el día a día de individuos, organizaciones y empresas; su extensa repercusión e influencia es un gran aliciente para el estudio de los problemas que permiten determinar caminos y rutas óptimas entre los nodos y arcos de un grafo.

Los orígenes de la teoría de grafos se remontan a 1736, año en el que Leonhard Euler publicó su afamado trabajo sobre los siete puentes de Königsberg. Con su obra presentó los antecedentes de toda una rama de conocimiento que carecería de nombre hasta la aparición del término "grafo" en la segunda mitad del siglo XIX. Su primer uso en un sentido similar al actual data de 1878, en un artículo publicado en la revista Nature. Unos años antes, otro término relacionado había irrumpido en los círculos académicos inaugurando uno de los apartados de mayor desarrollo dentro del propio ámbito de la teoría de grafos: el de los árboles.

El concepto de árbol, entendido como un grafo conectado sin ciclos, se presentó en sociedad de la mano de Gustav Kirchhoff, en cuyos trabajos sobre corrientes en circuitos eléctricos aparecía de forma implícita. Conceptos similares eran también recurso habitual en el campo de la química molecular con los escritos de James Joseph Sylvester o de Arthur Cayley. Es precisamente a este último al que se le atribuye el mérito de acuñar el término "árbol". Un término cuyo uso permaneció inicialmente vinculado a los campos que lo vieron nacer, no siendo hasta bien entrado el siglo XX cuando su utilidad empezaría a multiplicarse, dando pié, entre otros, al inicio de los problemas de búsqueda de árboles óptimos que aquí nos ocupan.

Durante los últimos 50 años la teoría de grafos ha experimentado su mayor crecimiento. Ello es debido, en buena medida, al impulso ejercido por el desarrollo de las modernas redes de

transportes y telecomunicaciones. Es justo en ese sector en donde destacan estructuras como los árboles, dada su utilidad para el estudio de estas u otras redes, como pueden ser redes eléctricas, de suministros o de distribución. No sólo eso, sus aplicaciones en la vida real se extienden por numerosos ámbitos y su uso también es recurso habitual a modo de subrutina de otros algoritmos, conformando uno de los campos más estudiados en teoría computacional.

De todo lo anterior da buena fe la vasta literatura existente y el hecho de que sus algoritmos han sido programados en multitud de lenguajes, existiendo herramientas y utilidades para mostrar su funcionamiento. A pesar de ello, existe un vacío importante en uno de los lenguajes de programación que ha ganado más popularidad en los últimos años: R. Aunque está orientado principalmente a la Estadística, R cuenta con varias librerías relacionadas con el área de la Investigación Operativa, algunas de ellas centradas precisamente en el estudio de redes y grafos. De lo que todavía carece es de un paquete común sobre árboles y arborescencias y los problemas relacionados con la búsqueda de los mismos.

Dicha ausencia ya justificaría de por sí la construcción de un paquete de R propio. Más aún cuando a ella se le une la escasez de código abierto disponible sobre algunos problemas concretos, como el de las arborescencias de coste mínimo o el de los árboles de corte minimal. Y no solo eso, la falta de herramientas informáticas sobre ellos que sirvan para indagar en algunos de los aspectos más novedosos de la teoría, como el reparto de costes o los juegos cooperativos asociados, terminan de configurar una oportunidad que parece posible aprovechar.

Este trabajo pretende ser un intento de afrontar dicha oportunidad con la construcción de un entorno que sirva de base para el estudio y resolución de problemas relacionados con la rama de la teoría de grafos que hemos convenido en llamar árboles óptimos. En concreto, con sus cuatro problemas más conocidos: árboles de coste mínimo, arborescencias de coste mínimo, árboles del camino más corto y árboles del corte minimal.

Que podamos plantearnos semejante objetivo solo es posible cuando uno se encuentra ante un campo en el que tantos otros han sentado antes sólidas bases sobre las que trabajar. Este proyecto bebe de todos ellos, a quienes se intentará dar el debido reconocimiento, ya que sin sus algoritmos y fórmulas nada de lo aquí programado podría ser posible. Si este proyecto se ha de dedicar a alguien es a todos y cada uno de ellos.

1.1. Objetivos y estructura

El punto de partida y primer objetivo de este trabajo será la programación en R de algoritmos para la resolución de los principales problemas referidos a la búsqueda de árboles óptimos en grafos. Además, se pretenden explorar aspectos cooperativos asociados a algunos de estos problemas con un segundo paquete y dotar a nuestro trabajo de una interfaz web que permita trabajar con ellos sin necesidad de recurrir a la consola de R. Para cumplir con todo lo anterior dividiremos nuestro esfuerzo en tres grandes bloques cuya estructura repasamos a continuación.

La creación del paquete de R `optrees` (abreviatura de "Optimal trees") constituye el núcleo central del proyecto. Con él pretendemos dotar al lenguaje de una librería capaz de trabajar con grafos y de funciones que permitan encontrar árboles óptimos dentro de ellos. Es a `optrees` al que está dedicado el Capítulo 2 de este documento, en el que echaremos un vistazo a la teoría y daremos buena cuenta de los principales algoritmos para su implementación en R.

En las páginas del capítulo repasaremos cada uno de los cuatro principales problemas y sus algoritmos, propondremos una versión en pseudocódigo de los mismos y resumiremos brevemente las interioridades de su traslado a R. En el proceso evitaremos extendernos demasiado, remitiendo al propio código y a la documentación del paquete para repasar su implementación con más detalle.

Una vez tengamos implementadas funciones capaces de obtener soluciones para los problemas de búsqueda de árboles óptimos, es interesante tratar de responder a la cuestión de cómo repartir los costes que conlleva el resultado. En la literatura se han propuesto diferentes reglas de reparto o reglas de asignación de costes que pretenden dar respuesta a esa pregunta. En el Capítulo 3 de este documento veremos alguna de estas reglas definidas para problemas de árboles de coste mínimo y de arborescencias de coste mínimo, explicando los pasos que hemos dado para su implementación en nuestro segundo paquete de R: `cooptrees`.

Terminado lo anterior y dado lo visual de los problemas aquí estudiados, plantearemos el desarrollo de una interfaz gráfica (`optreesGUI`) que permita manejar los paquetes `optrees` y `cooptrees` sin necesidad de acudir a la consola de R. Esto será posible gracias a nuevas librerías de R como Shiny y al uso de tecnologías web como HTML, CSS y JavaScript. Dedicaremos el capítulo 4 a explicar la utilización de las mismas para construir una aplicación que permita trabajar con los problemas de búsqueda de árboles óptimos directamente desde la web.

1.2. Herramientas utilizadas

En nuestra introducción ya hemos hablado de la que será la herramienta principal de nuestro trabajo: **R**[1]. Se trata de un lenguaje y entorno de programación, desarrollado originalmente para la computación estadística y su representación gráfica, que supone una alternativa libre al lenguaje de programación S. Creado por Ross Ihaka y Robert Gentleman de la Universidad de Auckland en Nueva Zelanda, el desarrollo de R como proyecto GNU corre a cargo del R Development Core Team. Su código fuente, escrito en C, Fortran y el propio R, es completamente accesible y se distribuye libremente bajo licencia GPL.

El hecho de que sea software libre es una de las ventajas indudables de R, pero existen otras que tienen que ver con su sintaxis sencilla y su alta capacidad de extensión. Esto le permite servir a todo tipo de propósitos, entre ellos a la Investigación Operativa. R no deja de ser un lenguaje de programación capaz de tratar eficientemente estructuras de datos como matrices, vectores, tablas o listas; por lo que ofrece posibilidades interesantes para su uso en teoría de grafos.

Al igual que recurrimos a ciertas especificaciones o pautas al redactar, a la hora de escribir código en un lenguaje de programación es positiva la adopción de ciertas normas o reglas de estilo a las que conviene atenerse. Cuestiones tales como la elección de nombres de variables o funciones, el uso de mayúsculas o símbolos, o la propia presentación de los comentarios; añaden coherencia al código y facilitan su legibilidad. A eso ayudan guías de estilo como la propuesta por la comunidad de usuarios de `R` en Google[2], basada en la más extensa "R Coding Convention"[3] y que, a falta de una oficial, será a la que recurramos.

El de la teoría de grafos es un ámbito poco explorado entre las librerías o paquetes de R. De entre lo que está disponible a día de hoy, el mejor representante lo constituye el paquete **igraph**[4], una potente librería programada en C para análisis de grafos y redes que cuenta con su propia versión para R y que tiene entre sus filas algunas funciones relacionadas con los problemas que nos ocupan. Con todo, la inmensa mayoría del código de `igraph` no está escrito en R y no satisface el nivel de interactividad buscado. Sí son valiosas sus funciones gráficas, que serán a las que recurriremos cuando necesitemos representar grafos con nuestros paquetes.

Eso sí, representar grafos en R tiene sus limitaciones, no permitiendo grandes alardes gráficos ni dotarlos de interacción. Para proporcionar una solución a esa deficiencia contaremos con la ayuda de **Shiny**[5], un paquete que proporciona un entorno para la construcción de aplicaciones web utilizando R. Introducido en noviembre de 2012 por el equipo de **RStudio**, Shiny permite la creación de webs interactivas en las que mostrar los resultados proporcionados por el código de R. Pero además, y de ahí su verdadera utilidad para nuestro caso, constituye una herramienta efectiva para la vinculación de datos entre código R y tecnologías web como **HTML**, **CSS** y **JavaScript**, proporcionando un potencial enorme para el desarrollo de una interfaz propia.

Es con HTML y CSS con lo que crearemos una interfaz web a la que dotaremos de dinamismo gracias a JavaScript. En este proceso nos será de gran utilidad **D3.js**[6], una librería utilizada para la visualización de datos en sitios web mediante el uso combinado de JavaScript, HTML5, CSS3, y también **SVG**. SVG es un formato de imágenes de vectores basado en XML y utilizado para la representación de gráficos en dos dimensiones, que incluye soporte para la animación e interacción con los mismos. La potencia combinada de todas estas herramientas será lo que nos permita construir no solo un código utilizable desde la consola de R sino toda una interfaz web que facilite trabajar con todos los algoritmos y funciones programados en nuestros paquetes `optrees` y `cooptrees` sin necesidad de conocimientos previos de programación.

Aunque todas las herramientas que hemos mencionado tienen un potencial enorme para ayudarnos en nuestro objetivo, tienen un par de problemas importantes que es importante mencionar: no existen vínculos entre muchas de ellas y no están inicialmente preparadas para trabajar con grafos. Buena parte del trabajo dedicado a este proyecto constituirá todo un desafío en forma de diseñar y programar rutinas que permitan enlazar las distintas herramientas hasta conformar, no solo paquetes que puedan ser utilizado desde R, sino también una interfaz en forma de página web que permita analizar grafos y árboles óptimos.

1.3. Instrucciones

Tres son los paquetes que se desarrollan a lo largo de este Trabajo de Fin de Máster: `optrees`, `cooptrees` y `optreesGUI`. En las siguientes páginas, dedicadas a recopilar algoritmos y fórmulas implementadas y a explicar el trabajo llevado a cabo, haremos continua referencia a sus funciones y pondremos ejemplos de su funcionamiento. Pero antes conviene tener los tres instalados.

Los paquetes `optrees` y `cooptrees` han sido subidos a los repositorios de CRAN y pueden ser instalados como una librería más de R. La interfaz web `optreesgui` cuenta con una versión accesible desde internet en la dirección optrees.net. Además se proporcionan los archivos binarios de instalación y el código fuente de los tres paquetes en el CD que acompaña a este documento.

En las siguientes instrucciones se especifican los pasos necesarios para instalar y ejecutar los paquetes por diferentes métodos.

- `optrees`: paquete de R sobre problemas de búsqueda de árboles óptimos que implementa algoritmos para la solución de los cuatro principales problemas de la materia: árboles de coste mínimo, arborescencias de coste mínimo, árboles del camino más corto y árboles de corte minimal.

  - Su instalación en R se puede realizar directamente desde CRAN:

```
# Instalar y cargar optrees desde CRAN
install.packages('optrees')
library(optrees)
```

    O copiando el archivo `optrees_1.0.zip` al directorio de trabajo y ejecutando:

```
# Requiere tener instaladas previamente las dependencias
install.packages('igraph')
# Instalar y cargar optrees desde el directorio de trabajo
install.packages('optrees_1.0.zip', repos = NULL)
library(optrees)
```

  - Su código fuente y su manual en formato PDF están disponibles para consulta en las carpetas correspondientes del CD adjunto.

- `cooptrees`: paquete de R complementario a `optrees` que incorpora a los problemas de árboles de coste mínimo y arborescencias de coste mínimo algunas de las reglas de reparto o asignación de costes más conocidas.

  - Su instalación en R se puede realizar directamente desde CRAN:

```
# Instalar y cargar cooptrees desde CRAN
install.packages('cooptrees')
library(cooptrees)
```

O copiando el archivo `cooptrees_1.0.zip` al directorio de trabajo y ejecutando:

```
# Requiere tener instaladas previamente las dependencias
install.packages('igraph')
install.packages('optrees')
# Instalar y cargar cooptrees desde el directorio de trabajo
install.packages('cooptrees_1.0.zip', repos = NULL)
library(cooptrees)
```

  - Su código fuente y su manual en formato PDF están disponibles para consulta en las carpetas correspondientes del CD adjunto.

- `optreesGUI`: Aplicación web basada en Shiny que permite controlar los paquetes `optrees` y `cooptrees` directamente desde un navegador web sin necesidad de acudir a la consola de R. Para su correcto funcionamiento se recomienda utilizar el navegador Google Chrome debido a su mejor compatibilidad con alguna de las librerías de javascript utilizadas.

  - Existe una versión en funcionamiento en la web http://optrees.net.

    O también se puede recurrir a su ejecución de forma local desde R, descomprimiendo el archivo `optreesgui.tar.gz` en el directorio de trabajo y ejecutando:

```
# Requiere tener Shiny previamente instalado
install.packages('shiny')
library(shiny)
# Ejecutar aplicación web
runApp('optreesgui')
```

  - Su código fuente está disponible para consulta en la carpeta correspondiente del CD adjunto.

Capítulo 2

Optrees: búsqueda de árboles óptimos

"-Empieza por el principio -dijo el Rey con gravedad
-y sigue hasta llegar al final; allí te paras."

- Lewis Carroll, Alicia en el país de las maravillas.

2.1. Introducción

El primer gran objetivo del proyecto es la programación en R de un paquete sobre árboles óptimos y los principales problemas referidos a la búsqueda de los mismos. Ya hemos visto como la falta de software específico y la inexistencia de un entorno cohesionado sobre la materia, salvo quizás lo ofrecido por `igraph`, genera una oportunidad para la construcción de nuestro propio paquete, al que denominaremos `optrees`. De él nos ocuparemos en este capítulo, en el que estudiaremos los diferentes problemas de búsqueda de árboles óptimos y varios algoritmos propuestos para solucionarlos, detallando y explicando su implementación en R.

Pero antes de entrar en materia necesitamos recopilar algunas definiciones comunes a los cuatro problemas. La teoría de grafos y su particular terminología exige ya de por sí un apartado como este, a lo que se le une nuestra intención de programar sobre ello en R, constituyendo un aliciente adicional para detenernos a repasar términos y conceptos que tendrán traducción directa en el código que desarrollemos.

2.1.1. Definiciones comunes

Un **grafo** es una estructura matemática formada por dos conjuntos de elementos: nodos y arcos. Los **nodos** (también denominados vértices[1]) son la unidad fundamental del grafo y se representan como puntos o circunferencias. En ellos inciden los **arcos** (o aristas[2]), enlaces que

[1]En la literatura anglosajona es más habitual el uso del término vértices.
[2]En ocasiones se recurre al uso del término arista para referirse a arcos no dirigidos.

indican la existencia de una conexión entre nodos y se representan con una línea entre ellos. La cantidad de nodos que contenga un grafo indicará su **orden**, mientras que su **tamaño** vendrá dado por el número de arcos presentes.

Definición 1: *Un grafo G es un par (V, A) donde:*

1. *V es el conjunto de nodos del grafo,*
2. *A es el conjunto de arcos del grafo.*

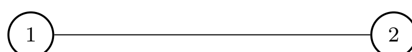


Fig. 2.1.: Grafo G con dos nodos, $V = \{1, 2\}$, y un arco que los une, $A = \{(1, 2)\}$.

Cada arco del grafo tiene asociado un par de nodos a los que une y que constituyen sus **puntos finales**. Los arcos pueden ser **no orientados**, si permiten ir de un nodo a otro indistintamente, de forma que $(i, j) = (j, i)$; u **orientados**, en el caso de que solo permitan ir desde un nodo origen i a un nodo final j, por lo que $(i, j) \neq (j, i)$. La diferenciación es clave puesto que, como veremos más adelante, nos enfrentaremos a problemas distintos según el tipo de grafo con el que nos encontremos: **grafo no dirigido**, en el que ningún arco es orientado; o **grafo dirigido**, en el que todos sus arcos son orientados. Estos últimos también reciben el nombre de **digrafos**.

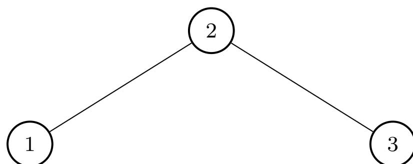


Fig. 2.2.: Grafo no dirigido.

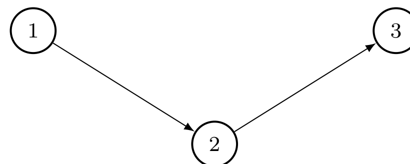


Fig. 2.3.: Grafo dirigido.

Con lo anterior ya se puede intuir que las relaciones entre arcos y nodos son una cuestión central en el ámbito de la teoría de grafos, por lo que es importante tener en cuenta una serie de definiciones para referirnos a sus conexiones. Así, empezaremos por hablar de un **paseo** cuando nos encontremos con una secuencia de nodos y arcos que empieza y termina con un nodo, en la que cada nodo es **incidente** al arco que le precede y al que le sigue en la secuencia, y los nodos que preceden y siguen a un arco son sus puntos finales. Un paseo se dice cerrado si su primer y último nodo es el mismo, siendo abierto en caso contrario. Asimismo, hablaremos de **recorrido o sendero** cuando estemos ante un paseo en el que todos los arcos son distintos, de **cadena** si además no se repite ningún nodo y de **circuito** si esta es cerrada. Utilizaremos el término **camino** para referirnos a una cadena en la que todos los arcos tienen la misma orientación, de

forma que el nodo final de un arco es el origen del siguiente. Y si este camino lo forma más de un nodo y además es cerrado tendremos un **ciclo**.

Se dice que dos nodos son **nodos conectados** cuando existe un paseo desde uno a otro. Uno o varios nodos conectados pueden formar una **componente**, que es un subconjunto de nodos y arcos del grafo en el que todos sus nodos están conectados unos a otros por caminos y ninguno de ellos está conectado a nodos adicionales del grafo. Estaremos ante un **grafo conexo** cuando este solo tenga una única componente, que además será el grafo entero.

Cuando hablemos de un **corte** en un grafo nos estaremos refiriendo a una partición de sus nodos en dos conjuntos disjuntos que pueden ser unidos por al menos un arco. El **conjunto de corte** lo forman los arcos cuyos puntos extremos se encuentran en diferentes conjuntos de la partición, arcos sobre los que se dice que están cruzando el corte y cuya suma indica el tamaño del conjunto de corte. Un corte característico es el **corte** $s - t$, que es aquel que requiere que el nodo fuente s y el nodo sumidero t se encuentren en particiones distintas.

En nuestro caso, el punto de partida de los problemas que vamos a estudiar es un **grafo conectado o conexo**, aquel en el que es posible formar un camino desde uno de sus nodos hasta cualquier otro. Dicho grafo, además, será un **grafo con pesos**, en el que cada arco tenga asignado un número denominado comúnmente como **peso** con el que se pueden representar costes, beneficios, distancias, flujos, tiempos, etc.

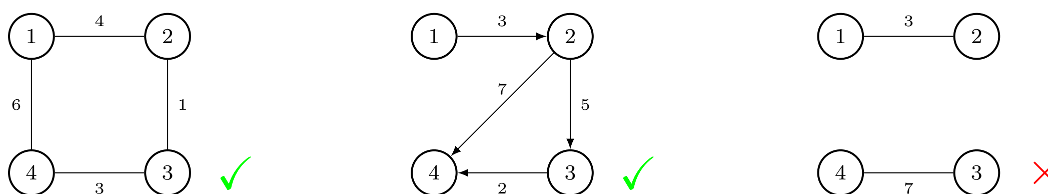


Fig. 2.4.: Ejemplo de dos grafos válidos para problemas de árboles óptimos y uno no válido. Este último no lo es al no ser conexo, ya que, por ejemplo, no es posible encontrar un camino desde el nodo 1 al nodo 3.

El objetivo común a todos los problemas de árboles óptimos pasa por encontrar un **subgrafo**, es decir, un grafo cuyo conjunto de nodos y arcos sean subconjuntos de los del grafo original. En concreto, buscamos un **subgrafo de expansión**, que es un subgrafo con el mismo conjunto de nodos que el grafo de partida, y uno muy particular, denominado árbol de expansión.

Un **árbol de expansión** es un subgrafo de expansión que además es un árbol. Un **árbol** es un grafo no dirigido y conexo que no tiene ciclos, de forma que cualquier par de nodos está conectado por un único camino. En ocasiones tendremos grafos dirigidos, por lo que, en esos casos, no hablaremos de árboles sino de **arborescencias**. Una arborescencia es un grafo dirigido y conexo que no tiene ciclos, de forma que cualquier par de nodos está conectado por un único camino.

Definición 2: *Un árbol T es un grafo no dirigido que satisface las siguientes condiciones:*

1. *T es conexo y no tiene ciclos.*
2. *Cualquier par de nodos de T están conectados por un único camino.*
3. *Si se añade un arco a T se forma un ciclo.*
4. *Si se elimina un arco de T éste deja de ser conexo.*

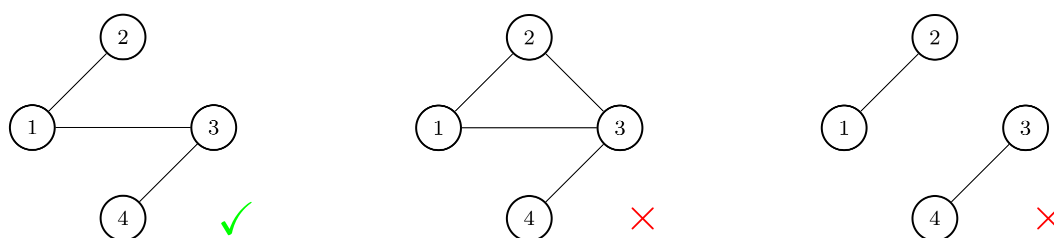


Fig. 2.5.: El primer grafo es un árbol, el segundo no (contiene un ciclo) y el tercero tampoco (no es conexo).

Definición 3: *Una arborescencia T' es un grafo dirigido que sería un árbol si se ignorasen las direcciones de sus arcos. Por esta razón se le denomina también árbol dirigido.*

Definición 4: *Un árbol de expansión asociado a un grafo $G = (V, A)$ no dirigido y conexo es un subgrafo $G^T = (V^T, A^T)$ que a su vez es un árbol e incluye los $|V|$ nodos de G y un subconjunto de $|V| - 1$ arcos de A, es decir, $V^T = V$ y $A^T \subset A$.*

Definición 5: *Una arborescencia de expansión asociada a un grafo $G = (V, A)$ dirigido y conexo es un subgrafo $G^{T'} = (V^{T'}, A^{T'})$ que a su vez es una arborescencia e incluye los $|V|$ nodos de G y un subconjunto de $|V| - 1$ arcos de A, es decir, $V^{T'} = V$ y $A^{T'} \subset A$.*

Fig. 2.6.: En rojo ejemplos de un árbol de expansión y una arborescencia de expansión sobre sus grafos originales.

Todas estas definiciones, sin ser un repaso exhaustivo a la extensa terminología de la teoría, suponen una base común suficiente para enfrentarnos a los problemas que nos ocuparán de

aquí en adelante. Como ya hemos visto, en estos problemas los datos de partida los conformarán siempre un conjunto de nodos y un conjunto de arcos que forman un grafo conexo con pesos, que puede ser dirigido o no; y el objetivo será la formación de un árbol o arborescencia de expansión a partir del mismo. Los detalles específicos se reservan para cada tipo de problema al que nos enfrentemos, de forma que los árboles resultantes pueden ser de distinto tipo:

- **Árbol de expansión de coste mínimo**: Dado un grafo no dirigido, conexo y con pesos, un árbol de expansión de coste mínimo será un árbol de expansión tal que la suma de los pesos de sus arcos es menor o igual que la suma de los pesos de los arcos de cualquier otro árbol de expansión asociado al grafo.
- **Arborescencia de coste mínimo**: Dado un grafo dirigido, conexo y con pesos, una arborescencia de coste mínimo será una arborescencia de expansión que verifica que la suma de los pesos de sus arcos es menor o igual que la suma de los pesos de los arcos de cualquier otra arborescencia de expansión asociada al grafo.
- **Árbol (o arborescencia) del camino más corto**: Dado un grafo dirigido o no, conexo y con pesos, un árbol (o arborescencia) del camino más corto es aquel que permite ir de un nodo fuente a cualquier otro mediante caminos en los que la suma de los pesos de los arcos que los forman sea mínima.
- **Árbol del corte minimal**: Dado un grafo no dirigido, conexo y con pesos, un árbol del corte minimal es un árbol de expansión que representa los cortes $i - j$ mínimos, es decir, todos aquellos cortes $i - j$ cuya suma de los arcos que los forman sea menor o igual a la suma de cualquier otro corte posible para todo par de nodos i y j presentes en el grafo.

2.1.2. Llevando grafos a R

En el apartado anterior hemos visto como un grafo G se define por un par (V, A) que contiene, respectivamente, su conjunto de nodos y su conjunto de arcos. También hemos señalado que, al margen de otros parámetros iniciales requeridos específicamente por alguno de ellos, los problemas de árboles óptimos tienen un grafo como dato principal de entrada. Como tal ha de ser tratado en nuestro código, por lo que la forma en cómo traslademos a R la estructura de un grafo es el primer aspecto clave a determinar.

En nuestra propuesta, que intenta respetar la formulación clásica de la teoría a la vez que pretende facilitar su interrelación con otras librerías como `igraph`, un grafo se introduce en R mediante un par de objetos: un vector de nodos y una matriz con la lista de arcos. El vector contendrá cada uno de los nodos identificados con números entre 1 y $|V|$; mientras que la matriz de arcos estará formada por $|A|$ filas de 3 columnas, de las que las dos primeras indican los nodos que une cada arco y la tercera contiene un valor numérico que representa su peso.

Siguiendo este esquema, si queremos introducir los datos de un grafo como el de la figura 2.7,

con 4 nodos y 5 arcos con sus correspondientes pesos, recurriríamos al código que le sigue a continuación:

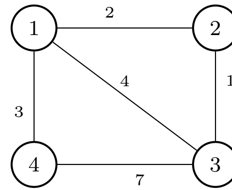


Fig. 2.7.: Grafo con $V = \{1, 2, 3, 4\}$, $A = \{(1, 2), (1, 3), (1, 4), (2, 3), (3, 4)\}$ y pesos $(2, 4, 3, 1, 7)$.

```
8  # Nodos y lista de arcos de un grafo
9  nodes <- 1:4
10 arcs <- matrix(c(1,2,2, 1,3,4, 1,4,3, 2,3,1, 3,4,7), ncol = 3, byrow = TRUE)
```

La anterior no es la única representación de un grafo a la que recurriremos. Aunque buena parte de las funciones del paquete `optrees` utilizarán como parámetro de entrada una lista de arcos tal y como acabamos de mostrar en el anterior extracto de código, en ocasiones se hará necesario acudir a una representación matricial de los mismos. Esta se define mediante una matriz $|V| \times |V|$, denominada habitualmente como **matriz de costes**, en la que cada fila i y cada columna j representan los puntos finales de cada arco y el valor en la posición (i, j) indica el peso asociado al mismo.

De esta forma, el grafo de la figura 2.7 también podría ser introducido en R mediante las siguientes líneas de código:

```
12 # Nodos y matriz de costes de un grafo
13 nodes <- 1:4
14 Cmat <- matrix(nrow = length(nodes), ncol = length(nodes))
15 Cmat[1, ] <- c( NA,   2,   4,   3)
16 Cmat[2, ] <- c(   2, NA,   1,Inf)
17 Cmat[3, ] <- c(   4,   1, NA,   7)
18 Cmat[3, ] <- c(   3,Inf,   7, NA)
```

Nótese que en las posiciones (i, j) cuando $i = j$ el peso se indica en R con `NA`. Esto es así porque en esos casos estamos ante **bucles**, es decir, arcos en los que sus dos extremos son un mismo nodo. Dado que un árbol por definición no tiene bucles, dichos arcos no afectan al problema y por eso su peso en la matriz de costes se especifica como `NA`: “Not Available”.

Nótese también la presencia del valor infinito en alguna posición de la matriz de costes, codificado con `Inf` en R. Esto ocurre cuando en un grafo no existe arco entre algún par de nodos, como de hecho pasa en la figura 2.7, donde no hay ninguna conexión directa entre los nodos 2 y 4 y, por tanto, la posición $(2, 4)$ de la matriz de costes tiene asignado el correspondiente valor infinito.

Por último, destacar el hecho de que el grafo de la figura 2.7 es un grafo no dirigido. Eso implica que su matriz de costes será simétrica, pues es indiferente considerar un arco como (i, j) o como (j, i) cuando no tiene orientación. Esta situación es apreciable en el extracto de código de R mostrado, donde el valor en la posición (i, j) se repite en la posición (j, i). Si el grafo fuese dirigido esto no tendría por qué ser así.

En nuestra formulación, tanto la lista de arcos como la matriz de costes contienen la misma información expresada de forma distinta, pero habrá situaciones en la que nos interese una forma sobre la otra en función de las operaciones que necesitemos realizar. Por defecto, la gran mayoría de nuestras funciones utilizan la lista de arcos, por lo que obviaremos indicar esto de aquí en adelante y nos limitaremos a avisar al lector cuando el parámetro de entrada utilizado sea la matriz de costes.

Por lo demás, la relación entre lista de arcos y matriz de costes es fácilmente apreciable, existiendo más de una ocasión en la que nos convenga trasladar la información de una a otra. Para agilizar esa conversión incluimos en el paquete `optrees` dos funciones auxiliares que permiten generar una lista de arcos a partir de una matriz de costes y viceversa. En su ejecución, basta introducir como parámetros de entrada los nodos y arcos del grafo en el formato correspondiente y señalar si este es dirigido o no.

```
20 # Convertir lista de arcos en matriz de costes
21 ArcList2Cmat(nodes, arcs, directed = FALSE)
22 # Convertir matriz de costes en lista de arcos
23 Cmat2ArcList(nodes, Cmat, directed = FALSE)
```

Cualquiera que sea la forma de introducción de los datos elegida, el grafo que se pretenda evaluar con el paquete `optrees` debe cumplir con los requisitos que imponen el tipo de problemas a los que nos enfrentamos. Esto es: el grafo no puede ser vacío y tiene que ser conexo, por lo que debe haber al menos un camino que permita ir desde cada uno de los nodos a todos los demás.

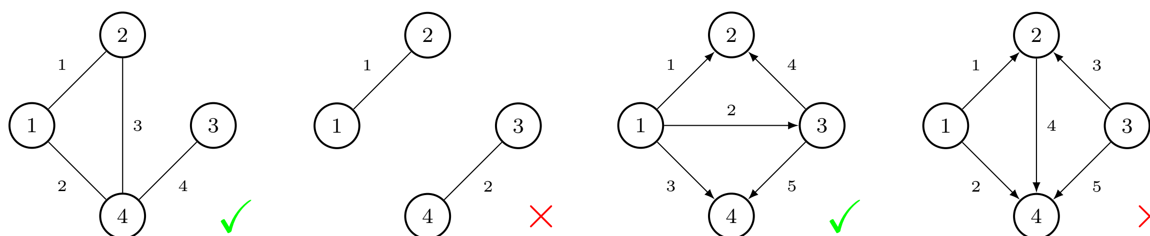


Fig. 2.8.: Ejemplos de grafos válidos y no válidos como datos de entrada de problemas de árboles óptimos.

Si el grafo no cumple con las condiciones nombradas anteriormente no será válido para problemas de árboles óptimos al ser imposible formar un árbol de expansión con sus nodos y arcos. Por ello, crearemos la función `checkGraph`, que se encargará de comprobar que un grafo introducido en R cumple todos los requisitos, confirmando que sirve como dato de entrada para los problemas.

```
25 # Comprobar la validez de un grafo para problemas de árboles óptimos
26 checkGraph(nodes, arcs, directed = FALSE)
```

Una de las comprobaciones realizadas por esta función es la relativa a la necesidad de que existan caminos que permitan ir desde cada uno de los nodos hasta todos los demás. De ello se encarga la función `searchWalk`, que devuelve el primer camino que encuentre, cualquiera que sea este, entre dos nodos indicados y es llamada reiterativamente desde `checkGraph`.

```
28 # Comprobar si existe un paseo entre nodos de un grafo
29 searchWalk(nodes, arcs, directed = FALSE, start.node = 1, end.node = 4)
```

En el caso de grafos no dirigidos existe otra forma de comprobar lo anterior y es atendiendo a sus componentes. Si los nodos de un grafo integran una única componente significa que este es conexo y cuenta con caminos que permiten ir desde cada uno de ellos hasta todos los demás.

Dicha posibilidad es solo una de las ventajas que ofrece tener una función capaz de detectar las componentes de un grafo y devolver la lista de nodos que integran las mismas. Por eso programamos la función `getComponents`, que recorre un grafo y devuelve la componente a la que pertenece cada arco y la lista con todas las componentes y los nodos que las integran.

```
31 # Obtener componentes de un grafo
32 getComponents(nodes, arcs)
```

A las anteriores funciones auxiliares se unen otras dos que pretenden reducir la complejidad del grafo introducido tanto como sea posible, lo que evitará llevar a cabo operaciones innecesarias. Por ejemplo, sabemos que a los problemas de búsqueda de árboles óptimos no les afecta la presencia de bucles (*"loops"* en inglés), por lo que estos pueden ser eliminados de los datos de entrada sin comprometer la resolución del problema. Eso hace la función `removeLoops`.

```
34 # Eliminar bucles de un grafo
35 removeLoops(arcs)
```

Pero, además, a estos problemas tampoco les afecta la existencia de **multiarcos**, arcos que tienen los mismos puntos finales que otros. Sobre ellos hemos de tener en cuenta si estamos ante un grafo dirigido o no, ya que en el primer caso se consideran multiarcos aquellos arcos que además de tener los mismos puntos finales que otros también tengan la misma orientación. Sea en uno u otro caso la operación a realizar será la misma de forma que, cuando descubra multiarcos, la función `removeMultiArcs` se encargará de reservar aquel con menor coste y eliminar el resto.

```
37 # Eliminar multiarcos de un grafo
38 removeMultiArcs(arcs, directed = FALSE)
```

La forma de tratar los grafos en R no quedaría completa sin proponer una representación de los mismos. En nuestro paquete esta tarea recae sobre la librería `igraph`, a la cual ya hemos hecho referencia. Dicha librería propone una representación al estilo tradicional y cuenta con rutinas eficientes para calcular la posición adecuada para los nodos y arcos del grafo. En nuestro código nos limitaremos a configurar el formato y aspecto de la representación con una función propia, a la que llamaremos cuando sea necesario para producir una imagen como la de la figura 2.9.

```
40 # Representar un grafo
41 repGraph(nodes, arcs)
```

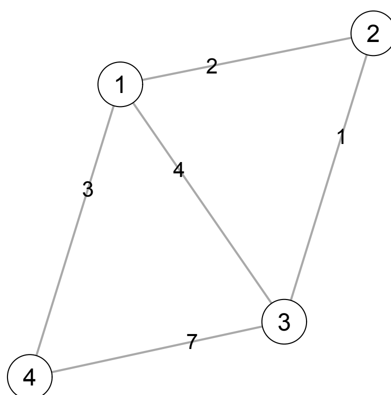


Fig. 2.9.: Representación de un grafo en el paquete `optrees` mediante la librería `igraph`.

Hasta aquí hemos presentado unas definiciones comunes a los cuatro problemas, configurado la forma con la que pretendemos que nuestro código entienda los grafos y creado unas primeras funciones auxiliares para trabajar con ellos. Lo siguiente es ya enfrentarnos a los cuatro problemas de árboles óptimos que integran la primera parte de este trabajo y cuya implementación en R constituye el núcleo principal del paquete `optrees`.

2.2. El problema del árbol de coste mínimo

El problema del árbol de coste mínimo fue formulado y resuelto por primera vez en 1926 por Otakar Borůvka[7]. Consultado sobre el diseño de una red eléctrica que permitiese una distribución eficiente de la electricidad en la región de Moravia, el matemático checo modeló la situación como un problema de árboles de coste mínimo sobre un grafo en el que las ciudades de la zona eran representadas por nodos y los arcos que las unían tenían asociado un peso equivalente a sus distancias. Borůvka propuso una solución basada en la adición repetida de conexiones entre las componentes del grafo hasta llegar a formar un árbol de expansión mínimo.

He ahí el objetivo de este primer tipo de problemas. Dado un grafo no dirigido, conexo y con pesos, se pretende encontrar un subgrafo del mismo que sea un árbol de expansión cuyo peso sea mínimo. Es decir, un subgrafo que conecte todos sus nodos y en el que la suma del peso de sus arcos sea menor o igual a la suma del peso de los arcos que forman cualquier otro árbol de expansión presente en el grafo.

Como ya indica su origen, este problema tiene utilidad directa en el diseño y estudio de redes de todo tipo, tanto eléctricas, como de telecomunicaciones, de transportes, de suministros o de ordenadores. Pero sus aplicaciones son mucho mayores e incluyen áreas tan dispares como la taxonomía, el análisis clúster, el reconocimiento de escritura, el registro y segmentación de imágenes, etc. Su uso también es recurso habitual a modo de subrutina de otros algoritmos, como de hecho ocurre, por ejemplo, en el caso del algoritmo de Christofides en su aproximación al problema del viajante.

En gran parte de estas aplicaciones es habitual que los pesos de los arcos sean considerados como costes y es por eso por lo que se suele hablar de árbol de expansión de coste mínimo o árbol de coste mínimo. Para evitar confundir al lector seguiremos esta costumbre y en este apartado utilizaremos siempre el término costes para denominar a los pesos.

Problema 1: *Dado un par* (G, C)*, donde:*

- G *es un grafo* (V, A) *no dirigido y conexo;*
- $C = (c_{ij})_{i,j \in V}$ *es una matriz de costes asociada a él, tal que:*
    - $c_{ij} \geq 0$ *cuando existe conexión directa entre los nodos* i *y* j*, en caso contrario* $c_{ij} = \infty$*;*
    - $c_{ii} = 0$ *y* $c_{ij} = c_{ji}$*, lo que implica que la matriz de costes es simétrica.*

Un ***árbol de coste mínimo*** *del problema* (G, C) *es un árbol de expansión* $G^T = (V^T, A^T)$ *que cumple que:*

- $V^T = V$ *y* $A^T \subset A$*;*
- $\sum_{(i,j) \in A^T} c_{ij} \leq \sum_{(i,j) \in A^{\tilde{T}}} c_{ij}$*, para todo árbol de expansión* $G^{\tilde{T}} = (V^{\tilde{T}}, A^{\tilde{T}})$ *asociado al problema* (G, C)*.*

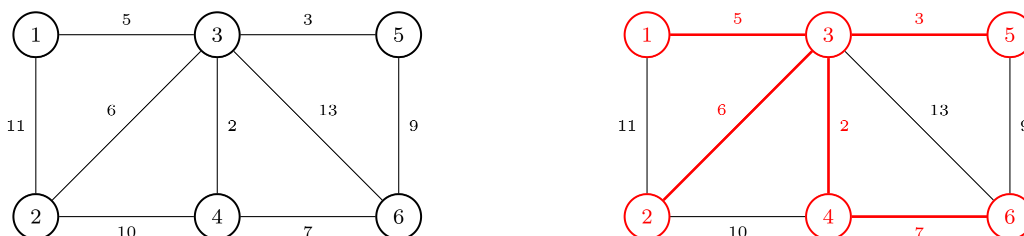


Fig. 2.10.: Un problema de árboles de coste mínimo (G, C) y su solución $G^T = (V^T, A^T)$.

Es inmediato ver que, dado que estamos hablando de un árbol de expansión, sea cual sea el grafo de partida del problema, el árbol de coste mínimo resultante tendrá siempre orden $|V|$ y tamaño $|V| - 1$. Ahora bien, si en el grafo existen arcos con un mismo coste, puede que este resultado no sea único y exista más de un árbol de expansión con igual coste mínimo. En las funciones del paquete `optrees` nos limitaremos a devolver uno de ellos.

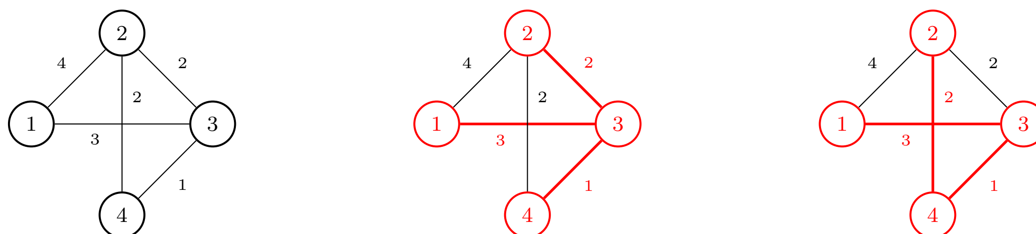


Fig. 2.11.: Problema original (G, C) y los dos posibles árboles de coste mínimo, G^{T_1} y G^{T_2}, cuyos costes asociados, en este caso 6, sabemos que siempre son iguales, es decir, $\sum_{(i,j) \in A^{T_1}} c_{ij} = \sum_{(i,j) \in A^{T_2}} c_{ij}$.

La dificultad del problema radica en determinar los pasos a seguir para encontrar un árbol de expansión dentro de un grafo no dirigido, conexo y con pesos, asegurando además que sea de coste mínimo. A lo largo de los años se han propuesto varios algoritmos para ello, destacando tres que serán los que estudiemos a continuación e implementaremos en el paquete `optrees`.

2.2.1. Algoritmo de Prim

El primero y más sencillo de los algoritmos para resolver el problema del árbol de coste mínimo es el algoritmo de Prim. Desarrollado por primera vez por el matemático checo Vojtêch Jarník, no sería hasta más tarde, en 1957, cuando aparecería publicado de forma independiente bajo la autoría del ingeniero informático estadounidense Robert C. Prim[8]. Es él quien le dio fama y por cuyo apellido es hoy conocido. Creado durante su etapa en Bell Labs, Prim trataba de abordar el problema de cómo conectar redes, ya fueran de telecomunicaciones o de transporte y distribución, mediante un número reducido o barato de conexiones.

La solución al problema propuesta por Prim se basa en la idea de ir conectando nodos secuencialmente hasta alcanzarlos a todos. Teniendo como dato de entrada un grafo no dirigido, el algoritmo empieza a construir el árbol a partir de un nodo seleccionado arbitrariamente como

punto de inicio. A continuación itera seleccionando en cada etapa el arco de menor coste (uno cualquiera si hay varios) que une un nodo del árbol con otro que aún no está en él; incorporando dicho arco y el nodo de destino al árbol. El proceso se repite hasta añadir todos los nodos, obteniendo como resultado un árbol de expansión cuyo coste será mínimo.

Algoritmo: Prim

Entrada: $G = (V, A)$; C; $s \in V$ *// grafo no dirigido, matriz de costes y nodo de inicio*
1: $A^T \leftarrow \emptyset$ *// conjunto de arcos del árbol a formar*
2: $V^T \leftarrow \{s\}$ *// conjunto de nodos del árbol a formar*
3: **mientras** $|V^T| < |V|$ **hacer**
4: seleccionar un arco $(i, j) \in A$ con $i \in V^T$ y $j \in V \setminus V^T$ cuyo c_{ij} sea mínimo
5: $A^T \leftarrow A^T \cup \{(i, j)\}$ *// añadir arco al árbol*
6: $V^T \leftarrow V^T \cup \{j\}$ *// añadir nuevo nodo al árbol*
7: **fin mientras**
Salida: $G^T = (V^T, A^T)$ *// árbol de expansión de coste mínimo*

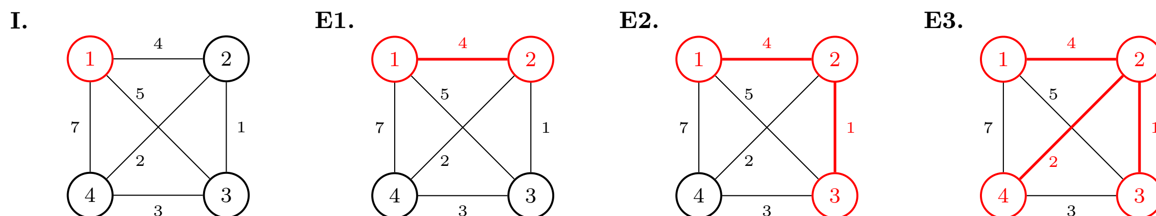


Fig. 2.12.: Ejemplo de ejecución del algoritmo de Prim detallada paso a paso en el Anexo A.1.

Para implementar los algoritmos en R utilizaremos el formato de grafo introducido en anteriores apartados, con un vector de nodos (`nodes`) y una lista de arcos (`arcs`) como parámetros de entrada. A diferencia del pseudocódigo, en el que introducíamos por un lado la lista de arcos y por otro la matriz de costes, codificamos la información de los arcos y sus pesos en un mismo objeto (`arcs`), tal y como se explica en el apartado 2.1.2 de este capítulo. Así lo haremos en todos los problemas de aquí en adelante.

```
45 # Grafo de ejemplo para problemas de árboles de coste mínimo
46 nodes <- 1:4
47 arcs <- matrix(c(1,2,4, 1,3,5, 1,4,7, 2,3,1, 2,4,2, 3,4,3),
48                ncol = 3, byrow = TRUE)
```

En el caso del algoritmo de Prim, añadiremos un parámetro adicional que servirá para indicar el nodo de partida, siendo el nodo 1 por defecto. De esta forma, el vector de nodos, la lista de arcos con sus costes y el nodo de inicio serán los datos de entrada de la función `msTreePrim` que contendrá nuestra implementación del algoritmo de Prim en R.

La función trabaja directamente con la lista de arcos, aunque modificándola ligeramente. Al tratarse de grafos no dirigidos tenemos que tener en cuenta que nuestro código ha de poder considerar que en un arco tanto podemos ir desde i hasta j como desde j hasta i. Por ese

motivo, introducimos un paso previo que duplica la lista añadiendo nuevos arcos con el mismo coste pero con sus nodos invertidos respecto a los introducidos inicialmente.

Salvado ese requisito previo, el código de R para el algoritmo de Prim es una traducción casi literal del pseudocódigo propuesto. Como resultado, la función devolverá en una lista el vector de nodos y la matriz con la lista de arcos y pesos del árbol de expansión de coste mínimo encontrado, además del número de iteraciones necesitadas para hallarlo.

```
50 # Algoritmo de Prim
51 msTreePrim(nodes, arcs)
```

Teóricamente, el algoritmo de Prim determina un árbol de expansión de coste mínimo con complejidad $O(|V|^2)$. El bucle es siempre ejecutado $|V|$ veces, y durante cada una de las interacciones la búsqueda de un arco puede ser hecha en un máximo de $|V| - |V^T|$ veces. En R las búsquedas y comparaciones entre elementos de vectores y matrices son veloces, lo que permite mantener nuestra implementación con resultados casi instantáneos por muy grande que sea el problema.

2.2.2. Algoritmo de Kruskal

No hay que ir demasiado lejos para encontrarnos con un segundo algoritmo capaz de resolver el problema de los árboles de coste mínimo. Su creación se debe a Joseph Kruskal, compañero de Prim en Bell Labs, quien lo propuso en 1956[9]. Conocido desde entonces por su apellido, la idea principal del algoritmo de Kruskal se resume en conectar arcos, y no nodos, secuencialmente. De esta forma hace innecesaria la elección de un nodo de partida.

Empezando con el conjunto inicial de arcos, el algoritmo de Kruskal selecciona aquel con el menor coste, uno cualquiera si hay varios, y lo incorpora al árbol. A partir de ahí repite dicho paso en cada etapa, asegurándose de que no se producen ciclos, hasta que logra añadir el número suficiente de arcos para formar un árbol de expansión que será de coste mínimo.

Algoritmo: Kruskal

Entrada: $G = (V, A)$; C — *// grafo no dirigido y matriz de costes*
1: $A^T \leftarrow \emptyset$ — *// conjunto de arcos del árbol a formar*
2: $M \leftarrow V$ — *// conjunto inicial de componentes*
3: **mientras** $|A^T| < (|V| - 1)$ **hacer**
4: seleccionar un arco $(i, j) \in A$ cuyo c_{ij} sea mínimo
5: **si** $M_i \neq M_j$ **entonces** — *// comprobar que no forman ciclo*
6: $A^T \leftarrow A^T \cup \{(i, j)\}$ — *// añadir arco al árbol*
7: $M_k \leftarrow M_i$ para todo $k \in \{1, ..., |M|\}$ tal que $M_k = M_j$ — *// unir componentes*
8: **fin si**
9: $A \leftarrow A \setminus \{(i, j)\}$ — *// eliminar arco de la lista*
10: **fin mientras**
11: $V^T \leftarrow V$ — *// añadir conjunto de nodos del árbol*
Salida: $G^T = (V^T, A^T)$ — *// árbol de expansión de coste mínimo*

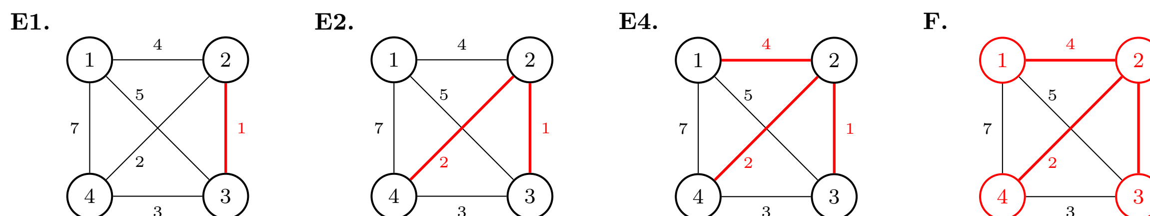


Fig. 2.13.: Ejemplo de ejecución del algoritmo de Kruskal detallada paso a paso en el Anexo A.2.

En el pseudocódigo podemos ver como reducimos la comprobación necesaria para asegurarnos de que no se producen ciclos llevando cuenta de las componentes en las que se divide el árbol. Eso es así porque los ciclos se forman al añadir un arco que une dos nodos de una misma componente. De esta forma, podemos establecer inicialmente que cada nodo se corresponda con una componente distinta y que estás se vayan uniendo conforme incorporemos arcos al árbol hasta que solo quede una. En cada iteración nos aseguraremos de que el arco a añadir no une nodos de una misma componente, eliminando el arco seleccionado para evitar volver a comprobarlo.

No siendo tan directo de programar como el de Prim, el algoritmo de Kruskal puede ser portado a R de forma muy similar al pseudocódigo presentado. De nuevo tendremos el vector de nodos (`nodes`) y la lista de arcos con los costes (`arcs`) como parámetros de entrada, aunque esta vez no hará falta indicar un nodo de inicio. También aquí hay un paso previo que nos permitirá ahorrar comprobaciones y reducir el tiempo de ejecución. Se encuentra al inicio de la función `msTreeKruskal` y consiste en ordenar de inicio los arcos de menor a mayor en función de su coste, de forma que al iterar lo hagamos uno a uno, evitando realizar el proceso de búsqueda del arco de menor coste en cada iteración del bucle principal.

Dicho bucle se repite hasta que el número de arcos del árbol determina que este es un árbol de expansión. Al finalizar, la función devuelve en una lista el vector de nodos y la matriz con los arcos y los costes del árbol de expansión de coste mínimo encontrado, así como el número de ejecuciones del bucle principal necesitadas para encontrarlo.

```
53 # Algoritmo de Kruskal
54 msTreeKruskal(nodes, arcs)
```

La teoría dice que el algoritmo de Kruskal es más rápido que el de Prim y nuestra implementación en R así lo confirma. Con una complejidad teórica de $O(A \log A)$, aún requiriendo en ocasiones más iteraciones que el de Prim, el algoritmo de Kruskal obtiene un árbol de expansión de coste mínimo en un tiempo sensiblemente inferior. Con todo, esto solo se hace apreciable a partir de grafos con un orden superior a cientos de nodos y un tamaño de varios miles de arcos.

2.2.3. Algoritmo de Borůvka

Para el último algoritmo que vamos a implementar en `R` sobre árboles de coste mínimo toca volver a sus orígenes. Otakar Borůvka[7] no solo fue el primero en modelizar este tipo de problemas, sino que además propuso un algoritmo para su resolución, siendo uno de las más recurridos todavía hoy. Aunque el algoritmo fue independientemente descubierto en varias ocasiones con posterioridad, en la actualidad se le conoce universalmente como el algoritmo de Borůvka.

Frente a los de Prim y Kruskal, el algoritmo de Borůvka utiliza una aproximación diferente basada en la conexión de componentes. El método empieza considerando a cada nodo del grafo como una componente distinta y continúa seleccionando todos los arcos de menor coste que inciden en cada una de ellas, para posteriormente reformular las componentes resultantes. A partir de ahí, repite el proceso buscando en cada etapa los arcos de coste mínimo que inciden en cada componente, evitando que se formen ciclos, hasta que solo queda una. Esta única componente constituirá un árbol de expansión que además será de coste mínimo.

Al igual que los anteriores, el algoritmo de Borůvka se puede explicar más formalmente en pseudocódigo, aunque en este caso es algo más complejo. Se inicializa de manera similar y también cuenta con un bucle "mientras" principal, pero para realizar todas las comprobaciones requiere de dos bucles adicionales que operan en cada componente y en cada nodo.

Algoritmo: Borůvka

Entrada: $G = (V, A)$; C — *// grafo no dirigido y matriz de costes*

1: $A^T \leftarrow \emptyset$ — *// conjunto de arcos del árbol a formar*
2: $M \leftarrow V$ — *// conjunto inicial de componentes*
3: **mientras** $|A^T| < (|V| - 1)$ **hacer**
4: $\quad S \leftarrow \emptyset$ — *// conjunto temporales de arcos*
5: $\quad$ **para** cada componente $k \in M$ **hacer**
6: $\quad\quad S' \leftarrow \emptyset$ — *// arcos seleccionados en cada componente*
7: $\quad\quad$ **para** cada nodo $i \in V$ tal que $M_i = k$ **hacer**
8: $\quad\quad\quad$ seleccionar un arco $(i, j) \in A$ con $j \in V$ tal que $M_j \neq k$ cuyo c_{ij} sea mínimo
9: $\quad\quad\quad$ **si** existe el arco (i, j) **entonces**
10: $\quad\quad\quad\quad S' \leftarrow S' \cup \{(i, j)\}$ — *// guardar arcos que unan nodos de componentes distintas*
11: $\quad\quad\quad$ **fin si**
12: $\quad\quad$ **fin para**
13: $\quad\quad$ seleccionar arco $(i, j) \in S'$ cuyo c_{ij} sea mínimo — *// escoger arco de entre los guardados*
14: $\quad\quad S \leftarrow S \cup \{(i, j)\}$ — *// almacenar arcos temporalmente*
15: $\quad$ **fin para**
16: $\quad$ **para** cada arco $(i, j) \in S$ **hacer**
17: $\quad\quad A^T \leftarrow A^T \cup \{(i, j)\}$ — *// añadir arco al árbol*
18: $\quad\quad M_l \leftarrow M_i$ para todo $l \in \{1, ..., |M|\}$ tal que $M_l = M_j$ — *// unir componentes*
19: $\quad$ **fin para**
20: **fin mientras**
21: $V^T \leftarrow V$ — *// añadir conjunto de nodos al árbol*

Salida: $G^T = (V^T, A^T)$ — *// árbol de expansión de coste mínimo*

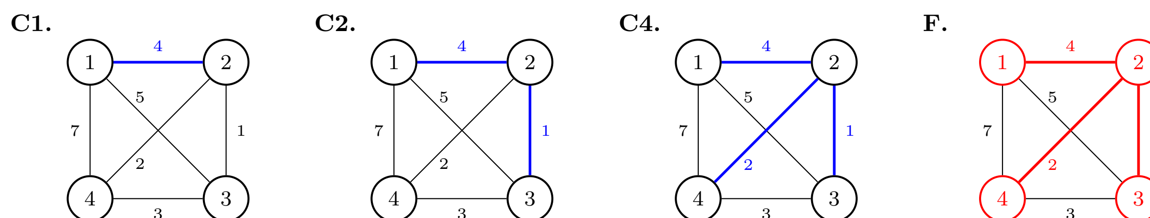


Fig. 2.14.: Ejemplo de ejecución del algoritmo de Borůvka detallada paso a paso en el Anexo A.3.

Si Kruskal ya añadía una ligera complejidad al trasladarlo a R, la propuesta del algoritmo de Borůvka dificulta aún más su programación. La función correspondiente volverá a tener el vector de nodos (`nodes`) y la lista de arcos con sus pesos (`arcs`) como parámetros de entrada. Además, al igual que el algoritmo de Prim, el de Borůvka requerirá el paso previo de duplicar los arcos de forma que nos aseguremos que se comprueban todas las posibilidades. A partir de ahí, la necesidad de recurrir a más bucles y realizar un mayor número de comprobaciones aumenta el código necesario que contiene la función `msTreeBoruvka`.

En el pseudocódigo se puede ver como el algoritmo necesita de hasta tres bucles dentro del principal, dos para comprobar cada componente y sus nodos y un tercero para añadir arcos y unir componentes. Al implementarlo en R se han hecho modificaciones, empezando por sustituir el primer bucle que itera sobre las componentes por un "while" que evita comprobar aquellas que podrían haber sido unidas a otras durante el proceso. En cada iteración seguimos comprobando para cada nodo los arcos de coste mínimo que lo unen con nodos en otras componentes, pero ya no recurrimos a un tercer bucle para elegir el arco de coste mínimo de todos ellos y añadirlo al árbol fusionando las componentes a las que pertenecen los dos nodos que une. Todo ese proceso se repite calculando el nuevo número de componentes e iterando sobre la siguiente.

Los pasos anteriores se llevan a cabb tantas veces como sea necesario hasta unir a todas las componentes en una. Cuando alcance ese punto, `msTreeBoruvka` habrá formado un árbol de expansión que será de coste mínimo. Al terminar, la función devuelve una lista con el vector de nodos y la matriz con la lista de arcos y pesos del árbol de coste mínimo encontrado, además del número de iteraciones del bucle principal necesitadas.

```
56 # Algoritmo de Boruvka
57 msTreeBoruvka(nodes, arcs)
```

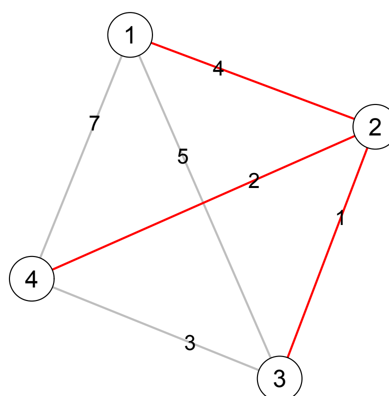
El algoritmo de Borůvka tiene una complejidad de $O(|A| \log |V|)$, siendo teóricamente al menos tan rápido como el de Kruskal. La idea es que el número de repeticiones que ha de registrar el bucle principal es mucho menor que las requeridas por Prim y Kruskal, pero dentro de él se repiten otros dos bucles que terminan por ralentizar el proceso en R. El resultado es que la versión del algoritmo de Borůvka implementada en nuestro paquete `optrees` es la más lenta de las tres llevadas a cabo para el árbol de coste mínimo.

2.2.4. getMinimumSpanningTree

Implementados los tres algoritmos, lo que haremos será agruparlos en una función general que permita llamar a cada uno según se determine en uno de sus parámetros de entrada. De esa tarea se encargará la función `getMinimumSpanningTree`. Esta comprueba que el grafo introducido, con su vector de nodos (`nodes`) y su lista de arcos con sus pesos (`arcs`), es válido y ejecuta una de las anteriores funciones según el parámetro `algorithm` especificado: `"Prim"`, `"Kruskal"` o `"Boruvka"`. El resultado puede ser seleccionado para su impresión en consola y para su representación gráfica mediante la librería `igraph`.

```
59 # Nodos y arcos del grafo
60 nodes <- 1:4
61 arcs <- matrix(c(1,2,4, 1,3,5, 1,4,7, 2,3,1, 2,4,2, 3,4,3),
62                ncol = 3, byrow = TRUE)
63 # Obtener árbol de coste mínimo
64 getMinimumSpaningTree(nodes, arcs, algorithm = "Prim")
```

```
Minimum Cost Spanning Tree
Algorithm: Prim
Stages:  3 | Time:  0
------------------------------
     ept1     ept2     weight
        1        2          4
        2        3          1
        2        4          2
------------------------------
                  Total = 7
```



2.3. El problema de la arborescencia de coste mínimo

Los problemas de arborescencias de coste mínimo parten de un planteamiento similar al de los árboles de coste mínimo estudiados en el apartado anterior, pero cuentan con la restricción añadida de que el grafo a resolver es dirigido. La consecuencia directa de dicha restricción es que la matriz de costes no tiene por qué ser simétrica, pudiendo ocurrir que el coste de conexión de un nodo i a otro j no sea igual que el coste de conexión del nodo j al nodo i. Debido a esto, los algoritmos vistos en el apartado anterior no pueden ser utilizados.

En las aplicaciones de árboles de coste mínimo asumíamos que los arcos carecen de orientación y pueden ser recorridos con el mismo coste yendo de un nodo i a otro j o viceversa, pero no siempre es así. Es fácil imaginar situaciones en las que el coste de ir de j a i sea diferente o incluso no sea posible. Esto puede ocurrir debido a circunstancias muy variadas, como puede ser la orografía del terreno, en el caso de redes de transportes o suministros; la capacidad asimétrica de la red, en el caso de redes de telecomunicaciones; o incluso decisiones normativas, comerciales o de otra índole que llevan a limitar de alguna forma el intercambio en una u otra dirección.

Dado un grafo dirigido, conexo y con pesos, el problema se formula en este caso con la selección de uno de sus nodos como fuente y con un objetivo que pasa por encontrar un subgrafo del grafo original que contenga a todos sus nodos y a $|V| - 1$ arcos de forma que exista un único camino desde la fuente a cualquier otro nodo. Dicho subgrafo recibe el nombre de arborescencia de expansión, y, para que sea de coste mínimo, la suma de los pesos de los arcos que la forman debe ser igual o menor a la de cualquier otra arborescencia de expansión contenida en el grafo.

Como ocurría en el caso de los árboles de coste mínimo, lo habitual aquí también es identificar los pesos como costes y por eso hablaremos de arborescencias de coste mínimo.

Problema 2: *Dado un par (G, C) donde:*

- *G es un grafo (V, A) dirigido y conexo, con un nodo fuente $s \in V$;*
- *$C = (c_{ij})_{i,j \in V}$ es su matriz de costes asociada, tal que:*
  - *$c_{ij} \geq 0$ cuando existe conexión directa entre los nodos i y j, en caso contrario $c_{ij} = \infty$;*
  - *$c_{ii} = 0$.*

*Una **arborescencia de coste mínimo** del problema (G, C) es una arborescencia de expansión $G^{T'} = (V^{T'}, A^{T'})$ de forma que:*

- *$V^{T'} = V$ y $A^{T'} \subset A$;*
- *$\sum_{(i,j) \in A^{T'}} c_{ij} \leq \sum_{(i,j) \in A^{\tilde{T}'}} c_{ij}$, para toda arborescencia de expansión $G^{\tilde{T}'} = (V^{\tilde{T}'}, A^{\tilde{T}'})$ asociada al problema (G, C).*

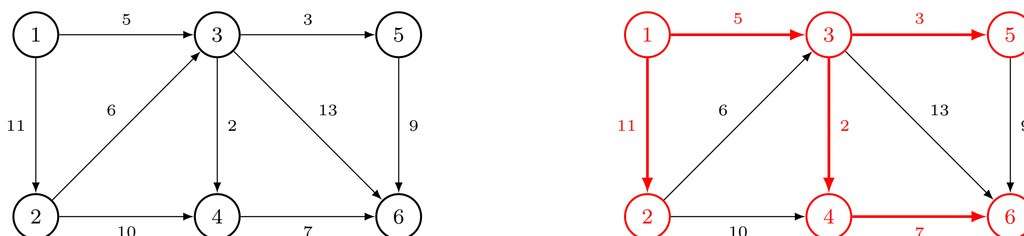


Fig. 2.15.: Un problema de árborescencias de coste mínimo (G, C) y su solución $G^{T'} = (V^{T'}, A^{T'})$.

Al igual que ocurre en el caso de los árboles de coste mínimo, es necesario señalar que en el problema de las arborescencias de coste mínimo la solución puede no ser única. En un mismo grafo podemos encontrarnos con varios conjuntos de $|V|-1$ arcos que formen una arborescencia y compartan el mismo coste mínimo. La función implementada en `optrees` se limita a devolver una de las arborescencias de expansión de coste mínimo, cualquiera que sea esta.

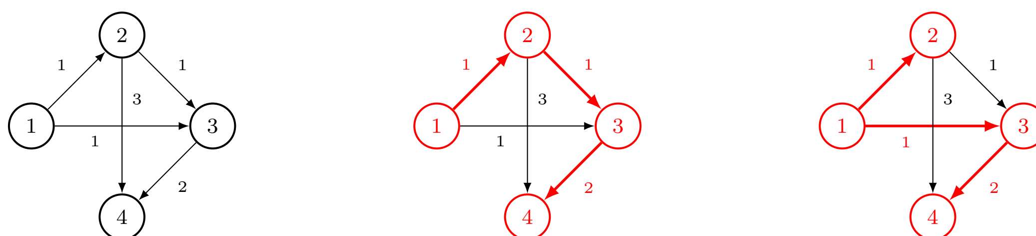


Fig. 2.16.: Problema original (G, C) y las dos posibles arborescencias de coste mínimo, $G^{T_1'}$ y $G^{T_2'}$, cuyos costes asociados, en este caso 4, sabemos que siempre son iguales, es decir, $\sum_{(i,j)\in A^{T_1'}} c_{ij} = \sum_{(i,j)\in A^{T_2'}} c_{ij}$.

2.3.1. Algoritmo de Edmonds

Para solucionar el problema de la arborescencia de coste mínimo contamos con el algoritmo de Edmonds o Chu-Liu. Desarrollado primero por Yoeng-Jin Chu y Tseng-Hong Liu en 1965, fue luego independientemente propuesto por Jack R. Edmonds en 1967[10]. Por el apellido de este último ha terminado por ser conocido a pesar de que la formulación es idéntica. En ambos casos, la idea es encontrar aquellos arcos con coste mínimo que permitan formar ciclos y contraerlos hasta que no sea posible continuar. El resultado será una arborescencia que puede ser reconstruida hasta conformar una arborescencia de expansión del grafo original.

Dado un grafo dirigido, conexo y con costes, el algoritmo de Edmonds se divide en dos grandes fases: una primera de búsqueda de una arborescencia, en la que se contraen nodos y ciclos hasta alcanzar una arborescencia de coste cero; y una segunda de reconstrucción de la arborescencia sobre el grafo original.

En la fase de búsqueda se empieza por restar a todos los arcos incidentes en cada nodo distinto de la fuente el coste del menor de ellos. Como resultado tendremos una serie de arcos con coste 0

que pueden dar lugar a una arborescencia, con lo que pararíamos el proceso, o no, en cuyo caso significa que se ha formado al menos un ciclo. En este segundo caso continuamos seleccionando uno de los ciclos formado por arcos de coste 0, da igual cuál; fusionando los nodos que lo forman y dando lugar a un nuevo grafo en el que los costes entre el nuevo nodo y el resto de los nodos vienen dados por el coste mínimo entre los nodos del ciclo y cada uno de los nodos restantes. Sobre este grafo volveremos a repetir todo el proceso hasta encontrar una arborescencia.

Una vez hayamos alcanzado una arborescencia, se inicia la fase de reconstrucción. En ella revertiremos el proceso de contracción que nos llevó hasta la arborescencia, reservando en cada etapa previa los arcos que la integran y añadiendo, con excepción de uno, los que forman el ciclo de coste 0. Al llegar al grafo original tendremos seleccionados arcos que forman una arborescencia de expansión cuyo coste además será mínimo.

Algoritmo: Edmonds

Entrada: $G = (V, A)$; C; $s \in V$ *// grafo dirigido, matriz de costes y nodo fuente*
1: $A^{T'} \leftarrow \emptyset$ *// conjunto de arcos de la arborescencia a formar*
2: $k \leftarrow 1$; $V_k \leftarrow V$; $A_k \leftarrow A$; $C_k \leftarrow C$ *// etapa inicial*
3: **mientras** $A^{T'} = \emptyset$ **hacer** *// fase de búsqueda de una arborescencia*
4: **para** cada nodo $j \in V \setminus \{s\}$ **hacer**
5: entre los arcos de A_k incidentes en j seleccionar el coste $c_{k_{ij}}$ mínimo
6: restar coste $c_{k_{ij}}$ a todos los arcos de A_k que inciden en j
7: **fin para**
8: seleccionar arcos de A_k con coste 0
9: **si** forman una arborescencia de expansión **entonces**
10: guardar arborescencia en $A^{T'}$
11: **sino**
12: seleccionar arcos de A_k con coste 0 que formen un ciclo
13: fusionar nodos y arcos del ciclo en un supernodo
14: $k \leftarrow k + 1$; $V_k \leftarrow V_{k-1}$; $A_k \leftarrow A_{k-1}$; $C_k \leftarrow C_{k-1}$ *// nueva etapa*
15: **fin si**
16: **fin mientras**
17: **mientras** $k > 1$ **hacer** *// fase de reconstrucción de la arborescencia*
18: deshacer supernodo de la etapa k y recuperar grafo, costes y ciclo de la etapa $k - 1$
19: seleccionar arcos del grafo en la etapa $k - 1$ que se correspondan con arcos en $A^{T'}$
20: seleccionar arcos del ciclo en la etapa $k - 1$ excepto el que incide en un nodo ya alcanzado
21: guardar arcos seleccionados en una nueva arborescencia en $A^{T'}$
22: $k \leftarrow k - 1$ *// comprobar etapa previa*
23: **fin mientras**
24: $V^{T'} \leftarrow V$ *// añadir conjunto de nodos al árbol*
Salida: $G^{T'} = (V^{T'}, A^{T'})$ *// arborescencia de expansión de coste mínimo*

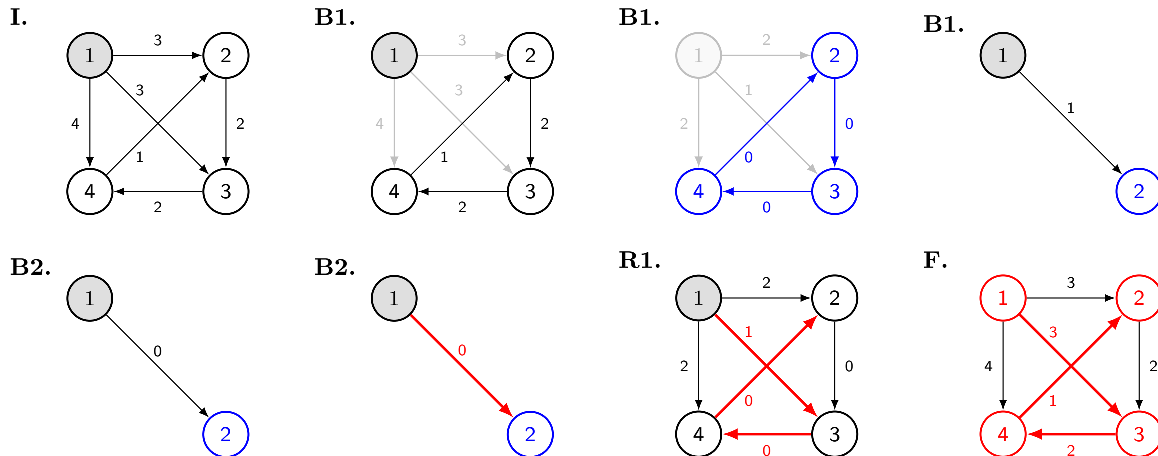


Fig. 2.17.: Ejemplo de ejecución paso a paso del algoritmo de Edmonds detallada en el Anexo A.4.

El de Edmonds es, probablemente, el algoritmo más difícil de implementar de todos los programados para el paquete `optrees`. Su elevado número de pasos, los cambios en la estructura del grafo entre etapas y la necesidad de conservar las relaciones entre nodos y arcos, complica considerablemente la tarea de programarlo. Ello explica la falta de software y código disponible sobre el mismo. Para poder portarlo a R dividiremos los pasos del algoritmo en diferentes funciones que nos permitirán trabajar más cómodamente con él. En cualquier caso, el grafo de entrada vuelve a introducirse como un vector de nodos y una matriz con la lista de arcos.

```
68 # Grafo de ejemplo para problemas de arborescencias de coste mínimo
69 nodes <- 1:4
70 arcs <- matrix(c(1,2,3, 1,3,3, 1,4,4, 2,3,2, 3,4,2, 4,2,1),
71               ncol = 3, byrow = TRUE)
```

La primera de las funciones que formarán parten de nuestra implementación del algoritmo de Edmonds sirve para seleccionar los arcos de coste mínimo que entran en cada nodo. Como parámetros de entrada toma el vector de nodos y la lista de arcos con los costes, aunque, para acelerar los cálculos, trabaja internamente sobre la matriz de costes.

```
73 # Seleccionar arcos de coste mínimo
74 getMinCostArcs(nodes, arcs)
```

La segunda función comprueba si estos arcos de coste mínimo forman o no una arborescencia. Para ello, `checkArbor` recorre el grafo en busca de caminos que permitan ir desde el nodo fuente al resto una única vez, formando una arborescencia. Al terminar devuelve `TRUE` o `FALSE` según exista una o no, y en el primer caso añade a la salida los arcos que la forman.

```
76 # Comprobar arborescencia
77 checkArbor(nodes, arcs)
```

Si en una etapa no existe arborescencia, el algoritmo de Edmonds continúa con más pasos, para lo que forma un nuevo conjunto de arcos a los que se les restan los costes mínimos de los arcos que inciden en cada nodo. De esa tarea se encarga la siguiente función, `getCheapArcs`, que vuelve a trabajar internamente sobre la matriz de costes para facilitar los cálculos y devuelve una nueva lista de arcos sobre los que continuar con el resto del proceso.

```
79 # Restar arcos de coste mínimo
80 getCheapArcs(nodes, arcs)
```

En la nueva lista de arcos nos interesan aquellos con coste 0. De su selección se encarga la función `getZeroArcs` que recurre una vez más a la matriz de costes para realizar sus operaciones.

```
82 # Seleccionar arcos de coste cero
83 getZeroArcs(nodes, arcs)
```

Si hemos llegado hasta aquí, ya sabemos que en la etapa en la que nos encontramos no existe arborescencia. Según el algoritmo de Edmonds, eso solo puede significar que en el grafo de arcos con costes reducidos haya un ciclo formado por aquellos con coste 0. Necesitamos encontrarlo e identificar los nodos y arcos que lo forman. De ello se encarga la función `searchZeroCycle`, que recorre el grafo en busca de ciclos y devuelve el primero que logre encontrar, da igual cuál sea, con los nodos y arcos que lo forman.

```
85 # Buscar un ciclo de coste cero
86 searchZeroCycle(nodes, arcs)
```

Una vez encontrado el ciclo, la tarea será contraer los nodos que lo forman en un único supernodo y formar un nuevo grafo con él y el resto de nodos y arcos. Para ello, creamos una función que se encargará de comprimir los nodos, manteniendo las correspondencias entre el supernodo y los nodos que contiene, y de seleccionar los arcos y sus nuevos costes. Hecho esto, devolverá sendos conjuntos de nodos y arcos reformulados con las correspondencias entre una y otra etapa.

```
88 # Ciclo de ejemplo
89 cycle <- c(2, 3, 4, 2)
90 # Contraer ciclo
91 compactCycle(nodes, arcs, cycle)
```

Llegados a este punto, tenemos un nuevo grafo con el que repetir los pasos de la fase de búsqueda. Las funciones anteriores representan estos pasos y forman parte de una iteración que se llevará a cabo tantas veces como sea necesaria para encontrar una arborescencia, guardando toda la información posible en cada etapa. Cuando hayamos llegado a la arborescencia, la tarea será revertir el proceso e ir formando otras en las etapas previas, desde la última hasta la primera, es decir, la que se corresponde con aquella buscada en el grafo original.

Para esta tarea de reconstrucción construimos una última función que realice el proceso de ir de un grafo con un ciclo comprimido hacia su grafo originario. Para ello introduciremos como parámetros de entrada todos los datos generados en una etapa del algoritmo y en la inmediatamente anterior, reconstruyendo esta segunda a partir de la primera. El proceso requiere atender a las correspondencias entre nodos y seleccionar los arcos adecuados en cada momento. Al terminar, la función devolverá los datos de la etapa anterior con la arborescencia encontrada.

```
93 # Reconstruir arborescencia
94 stepbackArbor(before, after)
```

De esta forma, todas estas funciones resumen los pasos del algoritmo de Edmonds, pero las mismas deben ser aplicadas en conjunto para hacerlo funcionar. A ello se dedicará una nueva función encargada de llamarlas en el orden adecuado y cuantas veces sea necesario. Esta es la función principal del código con el que implementaremos el algoritmo de Edmonds en R. Aquella que, dado el conjunto de nodos y arcos que forman un grafo e indicado un nodo fuente, aplica los pasos del algoritmo y encuentra y devuelve la lista de arcos que forman una arborescencia de expansión de coste mínimo.

```
96 # Algoritmo de Edmonds
97 msArborEdmonds(nodes, arcs, source.node = 1)
```

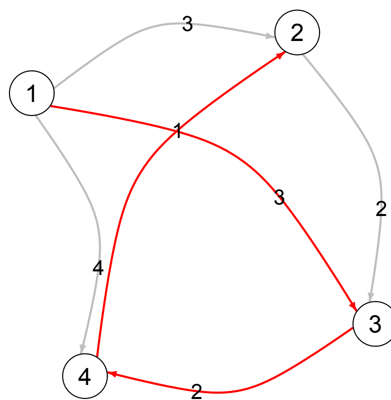
El tiempo computacional teórico del algoritmo de Edmonds es $O(AV)$, lo que debería asegurar una rápida ejecución incluso con grafos de gran orden y tamaño. No obstante, la gran cantidad de pasos requeridos, con más de un bucle en alguna de las funciones y numerosas comprobaciones a realizar, pesa en la velocidad de nuestra implementación en R.

2.3.2. getMinimumArborescence

Al igual que en el caso de los problemas de árboles de coste mínimo, una vez hemos implementado el algoritmo para la búsqueda de arborescencias de coste mínimo, construimos una función general que se encargue de aplicarlo. Esta función es `getMinimumArborescence`, que recibe los nodos (`nodes`) y arcos (`arcs`) de un grafo, junto a un nodo fuente (`source.node`), como parámetros de entrada y devuelve una arborescencia de coste mínimo como resultado. Además permite seleccionar opciones para mostrarlo en consola, representarlo gráficamente o devolver toda la información de cada una de las etapas.

```
99  # Nodos y arcos del grafo
100 nodes <- 1:4
101 arcs <- matrix(c(1,2,3, 1,3,3, 1,4,4, 2,3,2, 3,4,2, 4,2,1),
102                ncol = 3, byrow = TRUE)
103 # Obtener arborescencia de coste mínimo
104 getMinimumArborescence(nodes, arcs, algorithm="Edmonds")
```

```
Minimum cost spanning arborescence
Algorithm: Edmonds
Stages:  2 | Time:  0
------------------------------
     head      tail     weight
        1         3          3
        3         4          2
        4         2          1
------------------------------
                  Total = 6
```



2.4. El problema del árbol (o arborescencia) del camino más corto

El tercer problema es, en realidad, una extensión del problema más reducido del camino más corto en el que el objetivo es encontrar un camino entre un par de nodos de un grafo de tal forma que la suma de los pesos de los arcos que lo constituyen sea mínima. En el caso de árbol (o arborescencia) del camino más corto, lo que se pretende es ir un paso más allá y completar un árbol o arborescencia de expansión que contenga a todos los caminos más cortos desde un nodo de partida a todos los demás.

En términos más formales, dado un grafo, dirigido o no, conexo y con pesos, y fijado uno de sus nodos como fuente, un árbol o arborescencia del camino más corto será un árbol de expansión o una arborescencia de expansión tal que la suma de los pesos de los arcos que forman el camino desde el nodo fuente hasta cualquier otro nodo sea la más pequeña de todos los caminos posibles entre ambos nodos en el grafo original.

A diferencia de lo que ocurría en los problemas anteriores, en este problema del camino más corto se suelen relacionar los pesos de los arcos con distancias y así lo haremos de aquí en adelante. En este caso, además, el problema es equivalente tanto en grafos no dirigidos como en grafos dirigidos. En el primer caso formaremos un árbol de expansión, mientras que en el segundo formaremos una arborescencia de expansión. Dado que la formulación del problema y los algoritmos utilizados son iguales en ambos casos, utilizaremos la denominación común de árbol del camino más corto, englobando en ella tanto árboles como arborescencias.

Problema 3: *Sea un par (G, D) donde:*

- *G es un grafo (V, A), dirigido o no, conexo y con un nodo fuente $s \in V$;*
- *$D = (d_{ij})_{i,j \in V}$ es la matriz de distancias asociada a él, tal que:*
  - *d_{ij} es la distancia, positiva o negativa, cuando existe conexión directa entre los nodos i y j, en caso contrario $d_{ij} = \infty$;*
  - *$d_{ii} = 0$, y, en el caso de grafos no dirigidos, $d_{ij} = d_{ji}$.*

*Un **árbol del camino más corto** del problema (G, D) es un par (G^T, W), tal que:*

- *$G^T = (V^T, A^T)$ es un árbol de expansión, y, por tanto, verifica que $V^T = V$ y $A^T \subset A$.*
- *W es un vector asociado a G^T que contiene las distancias desde el nodo fuente al resto de nodos, de tal forma que w_{si} representa la suma de las distancias en D de los arcos de A^T asociados a un camino que conecta el nodo fuente s al nodo $i \in V^T \setminus \{s\}$. Esta suma será menor o igual a la suma de las distancias en D de los arcos de A que formen cualquier otro camino capaz de conectar a dichos nodos en el grafo original.*

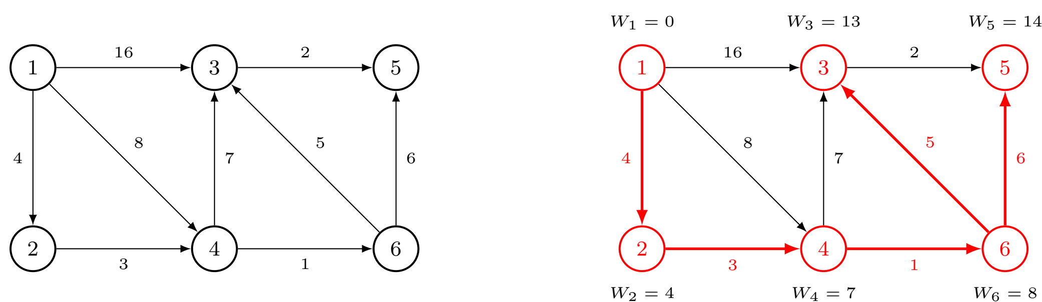


Fig. 2.18.: Un problema de árboles del camino más corto (G, D) y su solución (G^T, W).

Al igual que ocurre con los árboles y arborescencias de coste mínimo, la solución para el problema de los árboles del camino más corto no tiene por qué ser única, aunque el resultado siempre tendrá orden $|V|$ y tamaño $|V| - 1$. La diferencia con los otros problemas es que aquí no influye que los arcos sean dirigidos y las distancias pueden ser positivas o negativas. Eso sí, de contar con valores negativos solo funcionará uno de los algoritmos, y en grafos que presenten ciclos negativos no necesariamente se llegará a formar un árbol.

Si las aplicaciones de los árboles de coste mínimo son múltiples, las de los árboles del camino más corto no se quedan atrás. Los algoritmos que solucionan este problema son a menudo utilizados para la determinación de todo tipo de rutas, como pueden ser de transportes o de telecomunicaciones. Su uso también es un recurso habitual para establecer secuencias óptimas de objetivos y para la planificación de proyectos, así como para el estudio de conexiones en redes sociales. Cabe destacar además el desempeño fundamental de sus algoritmos en varios campos en los que también son utilizados como subrutinas de otros más complejos.

2.4.1. Algoritmo de Dijkstra

La primera solución que estudiaremos para resolver el problema del árbol del camino más corto fue concebida por Edsger Wybe Dijkstra en 1956 y publicada en 1959[11]. Dado un grafo dirigido o no dirigido con distancias no negativas, su algoritmo permite alcanzar un árbol del camino más corto. Para ello explora recursivamente todos los caminos más cortos que unen el nodo fuente con los demás nodos hasta que logra conectarlos formando un árbol o una arborescencia. Existen varias implementaciones del algoritmo al margen de la original de Dijkstra. De hecho, la que utilizaremos en el paquete `optrees` modifica un poco su funcionamiento para adaptarlo a nuestras necesidades.

Dado un grafo conexo, dirigido o no dirigido, con distancias no negativas; el algoritmo de Dijkstra empieza por marcar el nodo fuente e inicializar a 0 o a infinito las distancias acumuladas de cada nodo. Posteriormente itera, mientras queden nodos sin marcar, seleccionando en cada etapa el arco con la suma más pequeña de la distancia acumulada por uno de los nodos marcados más la distancia del arco que lo une a uno de los nodos no marcados, guardando este último nodo como marcado y actualizando su distancia acumulada con la suma antes calculada. Al final del

proceso tendremos un árbol de expansión del camino más corto.

Algoritmo: Dijkstra

Entrada: $G = (V, A)$; D; $s \in V$ *// grafo, matriz de distancias y nodo fuente*

1: $A^T \leftarrow \emptyset$ *// conjunto de arcos del árbol a formar*

2: $V^T \leftarrow \{s\}$ *// conjunto de nodos del árbol a formar*

3: $W_{1:|V|} \leftarrow 0$ *// distancias desde s a cada nodo*

4: **mientras** $|V^T| < |V|$ **hacer**

5: seleccionar arcos $(i, j) \in A$ con $i \in V^T$ y $j \in V \setminus V^T$

6: elegir un arco (i, j) de los anteriores tal que $d_{ij} + W_i$ sea mínimo

7: $A^T \leftarrow A^T \cup \{(i, j)\}$ *// añadir arco al árbol*

8: $V^T \leftarrow V^T \cup \{j\}$ *// añadir nodo al árbol*

9: $W_j \leftarrow d_{ij} + W_j$ *// actualizar distancia del nodo añadido*

10: **fin mientras**

Salida: $G^T = (V^T, A^T)$; W *// árbol del camino más corto y distancias*

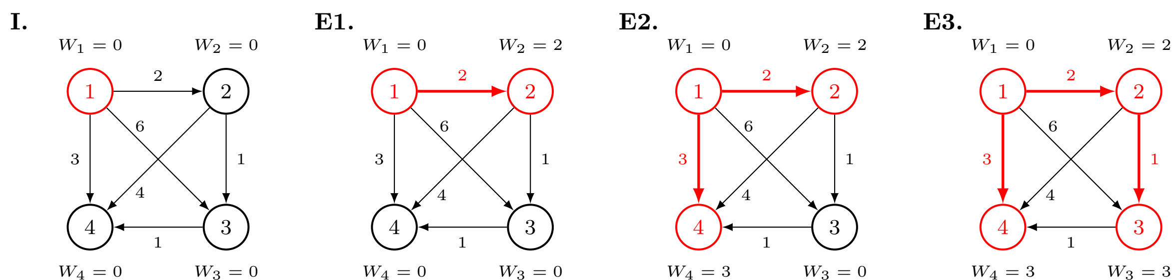


Fig. 2.19.: Ejemplo de ejecución del algoritmo de Dijkstra detallada paso a paso en el Anexo A.5.

La implementación del algoritmo tal y como está escrita en pseudocódigo no es casual y facilita portarlo a R. Es necesario, eso sí, indicar si el grafo que introducimos en la función es o no dirigido, pues en este segundo caso deberemos realizar el paso previo de duplicar los arcos. El resto de datos del grafo de entrada lo constituyen el conjunto de nodos y arcos, los cuales se introducen en R de la misma forma que hemos visto hasta ahora.

```
108 # Grafo de ejemplo para problemas de árboles del camino más corto
109 nodes <- 1:4
110 arcs <- matrix(c(1,2,2, 1,3,6, 1,4,4, 2,3,1, 2,4,4, 3,4,1),
111               ncol = 3, byrow = TRUE)
```

Por lo demás, las instrucciones del algoritmo de Dijkstra se pueden trasladar directamente manteniendo el uso de un único bucle, lo que permite evitar aumentar la complejidad del problema. El código queda recogido en una nueva función que devuelve el conjunto de nodos y la lista de arcos del árbol del camino más corto encontrado, así como los pasos requeridos para obtenerlo y el vector con las distancias desde el nodo fuente.

```
113 # Algoritmo de Dijkstra
114 spTreeDijkstra(nodes, arcs, source.node = 1, directed = FALSE)
```

La propuesta original de Dijkstra funciona en un tiempo computacional teórico de $O(|V|^2)$. Con el tiempo se han desarrollado mejores implementaciones que requieren una cola de prioridad y reducen la complejidad del algoritmo, pero complican portar el código a R. En nuestro caso, aunque con diferencias, mantenemos un formato muy similar al modelo original de Dijkstra, que es de por sí lo suficientemente rápido como para obtener resultados por debajo del segundo en grafos con cientos de nodos y miles de arcos.

2.4.2. Algoritmo de Bellman-Ford

El segundo algoritmo para el problema del árbol de camino más corto es el conocido como algoritmo de Bellman-Ford. Propuesto independientemente por Lester Ford Jr. en 1956[12] y por Richard Bellman en 1958[13], viene a solventar la imposibilidad del algoritmo de Dijkstra de lidiar con grafos en los que algunos arcos cuentan con pesos negativos. El de Bellman-Ford es capaz de detectar ciclos negativos y avisar de su existencia evitando entrar en un bucle en el que siempre se puede mejorar el camino más corto. Al igual que el de Dijkstra, se basa en un mecanismo en el que la aproximación a la distancia entre nodos se va mejorando sucesivamente hasta que se alcanza una solución óptima, pero se diferencia en la forma de llevarlo a cabo.

El algoritmo de Bellman-Ford empieza inicializando a 0 la distancia asociada a la fuente y a infinito las distancias acumuladas del resto de nodos. Posteriormente, lleva a cabo el proceso de relajación iterando en todos los nodos distintos de la fuente, comprobando en cada arco que sale de él si la distancia hacia otro nodo es mayor que la suma de la distancia acumulada del primero más el peso del arco que los une, en caso afirmativo establece este resultado como la distancia acumulada por el nodo de llegada y guarda el nodo de partida como su predecesor.

Al final termina comprobando si se han encontrado ciclos negativos, revisando arco por arco si se puede reducir la distancia acumulada del nodo de llegada. Si no se topa con uno, el resultado final es un árbol de expansión del camino más corto.

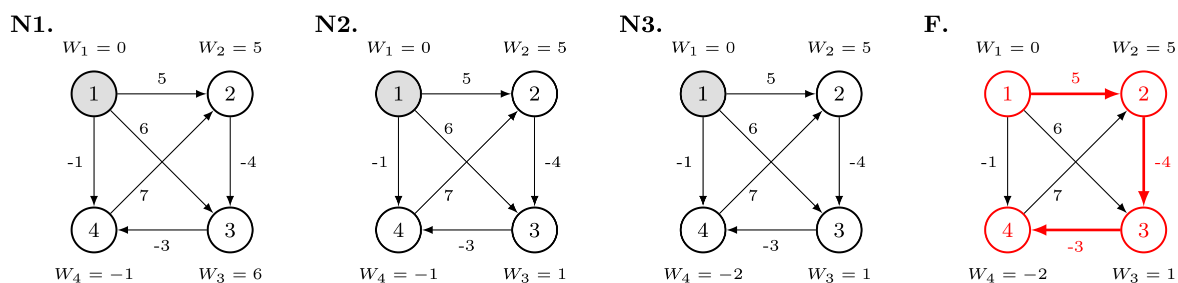


Fig. 2.20.: Ejemplo de ejecución del algoritmo de Bellman-Ford detallada paso a paso en el Anexo A.6.

Algoritmo: Bellman-Ford

Entrada: $G = (V, A)$; D; $s \in V$ *// grafo, matriz de distancias y nodo fuente*

1: $A^T \leftarrow \emptyset$ *// conjunto de arcos del árbol a formar*
2: $W_{1:|V|} \leftarrow \infty$; $W_s \leftarrow 0$ *// distancias asociadas a cada nodo y a la fuente*
3: $P \leftarrow \emptyset$ *// conjunto de nodos predecesores inicialmente vacío*
4: **para** cada nodo $i \in V \setminus \{s\}$ **hacer**
5: **para** cada arco $(i, j) \in A$ **hacer**
6: **si** $W_j > W_i + d_{ij}$ **entonces** *// proceso de relajación*
7: $W_j \leftarrow W_i + d_{ij}$ *// actualizar distancia acumulada*
8: $P_j \leftarrow i$ *// guardar predecesor*
9: **fin si**
10: **fin para**
11: **fin para**
12: **para** cada arco $(i, j) \in A$ **hacer**
13: **si** $W_j > W_i + d_{ij}$ **entonces** *// comprobación de ciclos negativos*
14: no hay solución
15: **fin si**
16: **fin para**
17: **para** cada nodo $j \in V \setminus \{s\}$ **hacer** *// construir árbol*
18: $i \leftarrow P_j$ *// recuperar predecesor*
19: seleccionar arco (i, j) de A
20: $A^T \leftarrow A^T \cup \{(i, j)\}$ *// añadir arco al árbol*
21: **fin para**
22: $V^T \leftarrow V$ *// añadir conjunto de nodos al árbol*

Salida: $G^T = (V^T, A^T)$; W *// árbol del camino más corto y distancias*

El proceso de programar el algoritmo de Bellman-Ford en R es equivalente al pseudocódigo aquí desarrollado, aunque con ligeros añadidos. En este caso, también es necesario introducir un parámetro de entrada que permita señalar si estamos ante un grafo dirigido o no, para duplicar los arcos ante esta segunda situación. La inicialización, el procedimiento de relajación y la comprobación de ciclos negativos se pueden trasladar directamente; pero al final es necesario incorporar un bucle adicional que reconstruya el árbol del camino más corto a partir del conjunto de nodos y los predecesores que hemos ido almacenando durante el proceso de relajación.

```
116 # Algoritmo de Bellman-Ford
117 spTreeBellmanFord(nodes, arcs, source.node = 1, directed = FALSE)
```

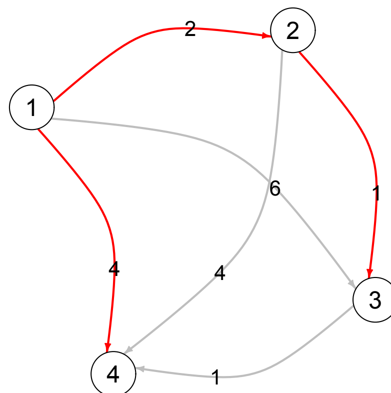
Este último bucle ralentiza un poco nuestra función. Teóricamente el algoritmo de Bellman-Ford tiene una complejidad computacional de $O(|V||A|)$. Es, por tanto, más lento que el algoritmo de Dijkstra, siendo este el precio a pagar por permitir trabajar con pesos negativos. Esa menor celeridad se mantiene en nuestro código, aunque la diferencia es prácticamente inapreciable.

2.4.3. getShortestPathTree

Programados los algoritmos, volvemos a recurrir a una función general que reciba los datos, llame al algoritmo indicado y prepare una salida para la consola de R y una representación gráfica. En los problemas del árbol del camino más corto ese trabajo está reservado a la función `getShortestPathTree`, que contará de nuevo con opciones entre los parámetros de entrada para elegir si se imprime o no en consola o si se representa gráficamente gracias a `igraph`.

```
119 # Nodos y arcos del grafo
120 nodes <- 1:4
121 arcs <- matrix(c(1,2,2, 1,3,6, 1,4,4, 2,3,1, 2,4,4, 3,4,1),
122                ncol = 3, byrow = TRUE)
123 # Obtener árbol del camino más corto
124 getShortestPathTree(nodes, arcs, algorithm = "Dijkstra")
```

```
Shortest path tree
Algorithm: Dijkstra
Stages:  3 | Time:  0
------------------------------
     head     tail    weight
        1        2         2
        2        3         1
        1        4         4
------------------------------
                   Total = 7
Distances from source:
------------------------------
   source     node  distance
        1        2         2
        1        3         3
        1        4         4
------------------------------
```



2.5. El problema del árbol del corte minimal

El último de los problemas de árboles óptimos sobre el que trataremos de implementar algoritmos de solución en R es el problema del árbol del corte minimal. Este árbol fue introducido por R. E. Gomory y T. C. Hu en 1961[14], por lo que a menudo es identificado como el árbol de Gomory-Hu. Ambos propusieron un algoritmo para determinarlo, construyendo a partir de un grafo dado uno nuevo con los mismos nodos pero en el que cada arco representa un corte mínimo del grafo original. Para explicar esto es necesario precisar un poco más algunas definiciones.

Al inicio de este capítulo ya hemos visto como un corte en un grafo es una partición de sus nodos en dos conjuntos disjuntos que pueden ser unidos por al menos un arco. De esta forma, dado un grafo (V, A), un corte entre dos nodos $i, j \in V$ es una partición del conjunto de nodos de V, representada por el conjunto $\{S_1, S_2\}$, tal que $i \in S_1$ y $j \in S_2$. La capacidad de este corte $\{S_1, S_2\}$ viene dada por la suma de las capacidades (así denominaremos a los pesos en este problema) correspondientes a los arcos que tienen un nodo en el conjunto S_1 y otro en el conjunto S_2, es decir, $w(S_1, S_2) = \sum_{k \in S_1, l \in S_2} w_{kl}$. El corte mínimo entre dos nodos i y j será el corte entre i y j que tenga menor capacidad.

Teniendo en cuenta lo anterior, dado un grafo no dirigido, conexo y con pesos, un árbol del corte minimal será un árbol que represente los cortes $i - j$ mínimos para todo par de nodos i y j presentes en el grafo. Al ser así, el árbol del corte minimal que se obtiene como solución al problema cumplirá con la propiedad de conservación del corte mínimo, según la cual un corte mínimo entre dos nodos en el árbol será también un corte mínimo entre esos dos mismos nodos en el grafo original. No solo eso, el árbol del corte minimal también verificará la propiedad, vinculada a la anterior como veremos más adelante, de conservación del flujo máximo, que establece que el flujo máximo entre cualquier par de nodos del árbol coincidirá con el flujo máximo de dichos nodos en el grafo original.

Problema 4: *Sea un par (G, Z) donde:*

- *G es un grafo (V, A) no dirigido y conexo;*
- *$Z = (z_{ij})_{i,j \in V}$ es su matriz de capacidades asociada, tal que:*
    - *$z_{ij} \geq 0$ denota la capacidad máxima del arco (i, j) cuando existe conexión directa entre los nodos i y j, en caso contrario $z_{ij} = \infty$;*
    - *$z_{ii} = 0$ y $z_{ij} = z_{ji}$, lo que implica que la matriz de capacidades es simétrica.*

*Un **árbol del corte minimal** asociado al problema (G, Z) es un par (G^T, Z^T), tal que:*

- *$G^T = (V^T, A^T)$ es un árbol con $V^T = V$;*
- *$Z^T = (z^T_{ij})_{i,j \in V}$ es la matriz asociada al árbol G^T, que representa los cortes mínimos entre pares de nodos, de forma que, si un arco $(i, j) \in A^T$, z^T_{ij} será la capacidad del corte $i - j$ mínimo en el par (G, Z) original.*

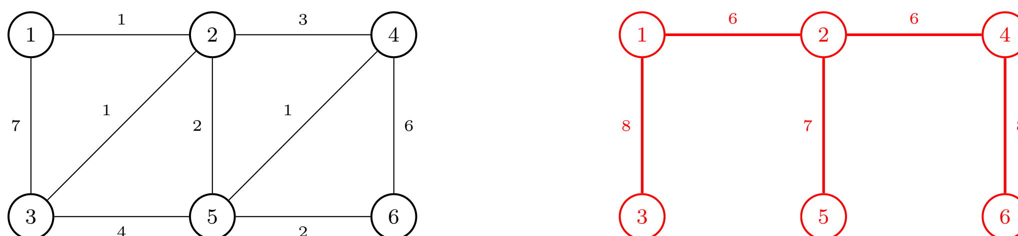


Fig. 2.21.: Un problema de árboles del corte minimal (G, Z) y una solución (G^T, Z^T).

Al igual que los otros tres problemas, las aplicaciones del problema del árbol del corte minimal son múltiples y variadas. Entre sus principales utilidades están aquellas relacionadas con el problema de determinar la mejor agrupación de nodos cuando se requiere una división del grafo. Así es utilizado en computación y redes de ordenadores, permitiendo determinar la capacidad de estas y gestionar los cortes de las mismas. También se recurre a él en la gestión de organizaciones, para determinar, por ejemplo, la división adecuada entre departamentos. E incluso en biología, donde se utiliza para estudiar la calidad de los enlaces entre proteínas.

2.5.1. Algoritmo de Gusfield

Aunque el algoritmo más conocido para obtener un árbol del corte minimal es el de Gomory-Hu, su formulación complica enormemente el programarlo en R. Al igual que en el caso de las arborescencias de coste mínimo, nos encontramos con la necesidad de comprimir nodos y mantener correspondencias entre ellos, elevando la complejidad del código necesario. Por fortuna, existen propuestas alternativas, como la de Dan Gusfield, que en 1990 publicó[15] un nuevo algoritmo para encontrar un árbol del corte minimal con una implementación más accesible.

La principal ventaja del algoritmo de Gusfield es, precisamente, evitar el requisito de comprimir nodos. El algoritmo arranca construyendo un árbol con el primer nodo como nodo único, e itera posteriormente añadiendo cada vez uno nuevo de acuerdo al orden $2, 3, ..., |V|$. En cada iteración la duda radica en escoger a qué nodo del árbol se une cada nuevo nodo k cuando existe más de una posibilidad. Para determinarlo repetimos el siguiente proceso:

- Buscamos el arco (i, j) de menor peso dentro del árbol. Este arco tendrá el valor del corte mínimo entre los nodos i y j en el grafo original.
- Borramos dicho arco del árbol formando dos componentes, una con el nodo i (T_1) y otra con el nodo j (T_2). Con ello sabremos también los nodos que pertenecen a la misma componente que i (S_1) y los que pertenecen a la misma componente que j (S_2) en el corte $i - j$ del grafo original.
- Si el nodo k pertenece a la componente S_1 en el grafo original, entonces fijamos como nuevo árbol a comprobar la componente T_1 del árbol. En caso contrario fijamos como nuevo árbol a comprobar la componente T_2 del árbol.
- Repetimos los pasos anteriores hasta que nos quedemos con un árbol con un único nodo.

Este será el nodo del árbol al que tenemos que unir el nuevo nodo k. El peso del arco que los una será igual al corte mínimo entre los dos nodos en el grafo original.

Todo este proceso se realiza hasta completar un árbol con todos los nodos del grafo y un nuevo conjunto de arcos. Este será un árbol del corte minimal en el que cada arco (i, j) se corresponda con la capacidad del corte $i - j$ mínimo en el grafo original.

Algoritmo: Gusfield

Entrada: $G = (V, A)$; Z *// grafo no dirigido y capacidades*

1: $V^T \leftarrow \{1\}$ *// empezar árbol por el primer nodo*
2: $A^T \leftarrow \emptyset$ *// conjunto de arcos del árbol a formar*
3: $Z^T_{|V|\times|V|} \leftarrow \infty$ *// matriz de capacidades a formar*
4: **para** cada nodo $i \in V \setminus 1$ **hacer**
5: $V^T_k \leftarrow V^T$; $A^T_k \leftarrow A^T$
6: **mientras** $|V^T_k| > 1$ **hacer** *// determinar un nodo del árbol*
7: borrar un arco $(i, j) \in A^T_k$ cuyo $z_{i,j}$ sea mínimo
8: determinar componentes T_1 y T_2 de V^T_k y A^T_k tal que $i \in T_1$ y $j \in T_2$
9: obtener corte mínimo entre i y j en el grafo $G = (V, A)$ original
10: determinar componentes S_1 y S_2 de V^T y A^T tal que $i \in S_1$ y $j \in S_2$
11: **si** nodo $i \in S_1$ **entonces** *// fijar nuevo árbol*
12: $V^T_k \leftarrow$ nodos de la componente T_1
13: $A^T_k \leftarrow$ arcos de la componente T_1
14: **sino**
15: $V^T_k \leftarrow$ nodos de la componente T_2
16: $A^T_k \leftarrow$ arcos de la componente T_2
17: **fin si**
18: **fin mientras**
19: añadir arco (i, k), con $k \in V^T_k$, al árbol A^T
20: obtener corte mínimo entre i y k en el grafo $G = (V, A)$ original
21: $z^T_{ik} \leftarrow$ capacidad del corte mínimo $i - k$
22: $V^T \leftarrow V^T \cup \{i\}$ *// añadir nodo al árbol*
23: **fin para**

Salida: $G^T = (V^T, A^T)$; Z^T *// árbol del corte minimal y capacidades*

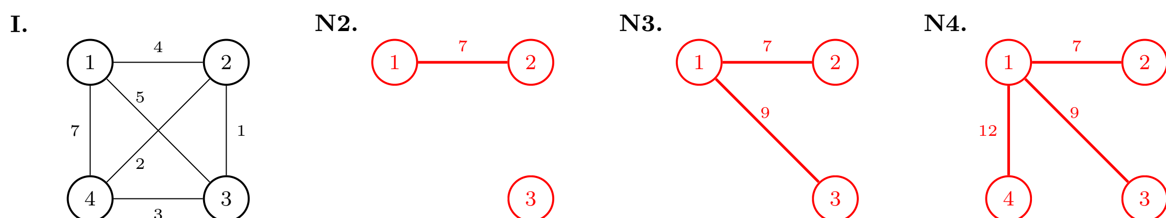


Fig. 2.22.: Ejemplo de ejecución del algoritmo de Gusfield detallada paso a paso en el Anexo A.7.

Este planteamiento del algoritmo de Gusfield pone más fáciles las cosas a la hora de obtener una solución para el problema del árbol del corte minimal, y, de paso, da pie a la creación de funciones adicionales relacionadas con el corte mínimo y el flujo máximo que pueden ser utilizadas en otro tipo de problemas. Tal y como vemos en el pseudocódigo, necesitamos implementar una función capaz de obtener el corte mínimo de un grafo y su capacidad. Esto se puede lograr mediante la aplicación de algoritmos para determinar el flujo máximo de una red y el recurso al teorema del flujo-máximo corte mínimo.

El **teorema del flujo-máximo corte-mínimo**[16] especifica que la capacidad del corte mínimo entre dos nodos i y j es equivalente a calcular el flujo máximo entre ellos. Es decir, al tratar de determinar el flujo máximo que circula por una red obtendremos también el corte mínimo de la misma. Es por esto por lo que, al implementar el algoritmo de Gusfield, tenemos que plantearnos antes el problema del flujo máximo y un algoritmo de solución del mismo.

Problema 5: *Dado un problema (G, W), un problema de flujo entre dos nodos i y j (donde uno de los nodos será la fuente y el otro el sumidero), viene dado por un par (G, f), en el que f es una función $f : A \to \mathbb{R}^+$ que verifica las siguientes condiciones:*

- $f_{kl} \leq w_{kl}$ *para todo* $(k, l) \in A$.
- *Ley de conservación del flujo: el flujo total que entra en un nodo por medio de los caminos que van de la fuente al sumidero es igual al flujo total que sale de dicho nodo (esta propiedad se verifica para todos los nodos de V que no sean la fuente o el sumidero).*

*El **flujo máximo** entre dos nodos i y j viene dado por el $\sum_{(k,l) \in A} f_{kl}$ máximo.*

Aunque existen varios algoritmos para obtener una solución del problema de flujo máximo, en `optrees` nos limitaremos a programar uno de los más conocidos: el algoritmo de Ford-Fulkerson[16]. La idea del mismo es enviar flujo recursivamente entre el nodo fuente y el nodo sumidero a través de caminos cuyos arcos tengan capacidad disponible. Eventualmente algunos de estos arcos se verán saturados y será imposible encontrar un nuevo camino, quedando el grafo dividido en dos componentes, una con el nodo fuente y otra con el nodo sumidero. Aquellos arcos del grafo que unan nodos de componentes distintas serán los que formen el conjunto de corte mínimo, y la suma de sus capacidades determinará el flujo máximo de la red.

Para implementar el algoritmo de Ford-Fulkerson en R desarrollaremos antes un par de funciones adicionales. La primera de ellas será la encargada de atravesar el grafo y obtener un camino con capacidad disponible entre dos nodos dados. Para ello, a la habitual lista de arcos, con tres columnas indicando los puntos finales y el peso de cada uno (en este caso su capacidad), le añadimos una cuarta que indica el flujo que ya circula por ellos. A partir de ahí, la función utiliza una estrategia de búsqueda en profundidad para alcanzar desde la fuente el nodo marcado como sumidero escogiendo a cada paso el arco con mayor capacidad disponible. Como resultado devuelve una lista con el conjunto de nodos y arcos que forman el camino encontrado.

```
133 # Buscar camino de flujo máximo entre nodos
134 searchFlowPath(nodes, arcs)
```

Tras encontrar sucesivos caminos con capacidad disponible en el grafo e ir saturándolos hasta que sea imposible conectar nodo fuente y sumidero, necesitaremos llamar a una segunda función. Esta se encarga de revisar el nuevo conjunto de arcos mediante una estrategia de búsqueda en anchura hasta que logra determinar el conjunto de corte que divide el grafo en dos particiones. Al terminar devuelve dos vectores con los nodos de una y otra partición.

```
136 # Encontrar conjunto de corte s-t
137 findstCut(nodes, arcs, s = 1, t = 4)
```

Las dos funciones anteriores son las utilizadas por `maxFlowFordFulkerson` para determinar el flujo máximo de una red de nodos. Para ello llama recursivamente a `searchFlowPath`, determinando un camino entre los nodos fuente y sumidero. Posteriormente, resta las capacidades de los arcos del camino y repite el proceso hasta que satura los suficientes arcos como para que sea imposible encontrar uno nuevo. Al final utiliza la función `findstCut` para determinar las dos particiones del grafo, una con el nodo fuente y otra con el nodo sumidero, y extrae la suma de las capacidades de aquellos arcos con un nodo en cada partición para calcular el flujo máximo.

```
139 # Determinar flujo máximo de la red
140 maxFlowFordFulkerson(nodes, arcs, source.node = 1, sink.node = 4)
```

Aplicando el teorema del flujo-máximo corte-mínimo, sabemos que la máxima cantidad de flujo que puede ir desde un nodo fuente hasta un nodo sumidero es igual al peso de los arcos que forman el corte mínimo entre ellos. Con la implementación del algoritmo de Ford-Fulkerson, al calcular el flujo máximo ya determinamos un conjunto de corte que se corresponde con este corte mínimo. De esta forma, a la nueva función `findMinCut` le basta con recurrir a la función `maxFlowFordFulkerson` para determinar que arcos forman el conjunto de corte mínimo.

```
142 # Determinar conjunto de corte mínimo
143 findMinCut(nodes, arcs, source.node = 1, sink.node = 4)
```

Llegados a este punto, ya estamos en posición de implementar el algoritmo de Gusfield en R. Para ello trasladamos el pseudocódigo antes especificado, manteniendo los dos bucles principales y utilizando las funciones anteriores para determinar y obtener los cortes y flujos máximos necesarios. El resultado es una función capaz de construir un árbol del corte minimal a partir de los nodos y arcos de un grafo dado.

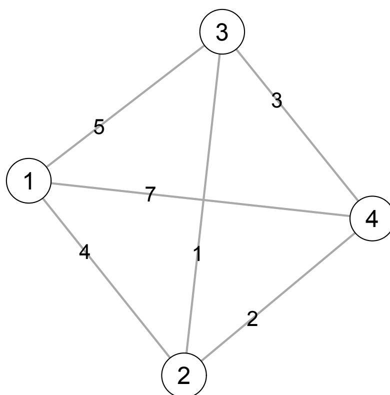
```
145 # Algoritmo de Gusfield
146 ghTreeGusfield(nodes, arcs)
```

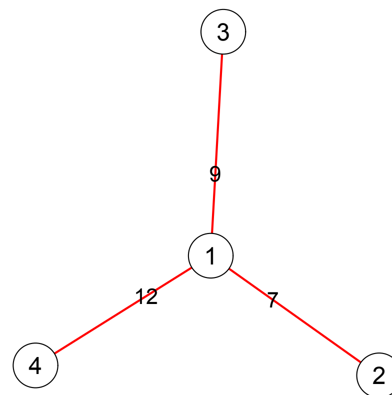
2.5.2. getMinimumCutTree

Tal y como hemos hecho en los otros tres, en el problema del árbol del corte minimal volvemos a crear una función general que se encargue de llevar a cabo la ejecución del algoritmo implementado. Esta es `getMinimumCutTree`, la cual recibe los nodos y arcos que conforman el grafo a estudiar y devuelve un resultado que puede ser seleccionado para su impresión en consola y su representación gráfica mediante el paquete `igraph`.

```
148 # Nodos y arcos del grafo
149 nodes <- 1:4
150 arcs <- matrix(c(1,2,4, 1,3,5, 1,4,7, 2,3,1, 2,4,2, 3,4,3),
151                byrow = TRUE, ncol = 3)
152 # Obtener árbol de corte minimal
153 getMinimumCutTree(nodes, arcs)
```

```
Minimum cut tree (Gomory-Hu)
Algorithm: Gusfield
Steps:  4 | Time:  0.03
------------------------------
     ept1     ept2    weight
        1        2         7
        1        3         9
        1        4        12
------------------------------
```





Capítulo 3

Cooptrees: cooperación en árboles óptimos

"-¿Sabes sumar? -le preguntó la Reina blanca-. ¿Cuánto es uno más uno más uno más uno más uno más uno más uno?
-No lo sé -dijo Alicia-. Perdí la cuenta.
-No sabe hacer una adición -le interrumpió la Reina Roja."

- Lewis Carroll, Alicia a través del espejo.

3.1. Introducción

En el capítulo anterior hemos visto que, tanto en el caso de los árboles de expansión de coste mínimo como en las arborescencias de expansión de coste mínimo, es habitual considerar los pesos de los arcos como costes. En ambos problemas nos podemos encontrar, además, con situaciones en las que un grupo de agentes, que se identifican como nodos de un grafo y que en este contexto denotaremos por N[1], debe conectarse a otro nodo conocido como fuente y que se denotará por 0, que es el que va a proporcionar o suministrar un determinado servicio a los agentes.

Ya hemos visto como el objetivo principal de los dos problemas es obtener un árbol o arborescencia de expansión de coste mínimo; pero, una vez obtenido este, surge la cuestión, no menos relevante, de determinar cómo han de repartirse el coste asociado a la solución obtenida. Los algoritmos implementados en el paquete `optrees` son capaces de determinar un árbol o una arborescencia, pero nada dicen del posible reparto del coste asociado entre los diferentes nodos a los que conecta. En situaciones como las que consideraremos ahora, en las que tenemos varios agentes conectados a una fuente, hemos de ir un paso más allá y atender a la cooperación en este tipo de problemas.

[1] La notación cambia aquí ligeramente respecto al capítulo anterior. El conjunto de nodos del grafo se denotará ahora por $N_0 = N \cup 0$, donde $N = \{1, 2, ..., n\}$ es el conjunto de agentes y 0 es la fuente. Además, los problemas de árboles de coste mínimo y los problemas de arborescencias de coste mínimo se denotarán indistintamente por el par (N_0, C), siendo C la matriz de costes asociada.

De eso nos encargaremos en este tercer capítulo. En las próximas líneas estudiaremos los juegos cooperativos asociados a los problemas del árbol de coste mínimo y los problemas de la arborescencia de coste mínimo y repasaremos las principales reglas de reparto y asignación de costes propuestas en la literatura. El objetivo es programar en R un conjunto de funciones que permitan obtener los juegos y las reglas de reparto asociados a cada problema, integrando un segundo paquete denominado `cooptrees`.

Pero antes conviene detenerse a definir qué entendemos por juego cooperativo y por una regla de reparto o asignación de costes.

3.1.1. Definiciones comunes

Una vez obtenido un árbol o una arborescencia de coste mínimo, una regla de reparto, o regla de asignación de costes, trata de responder a la pregunta de cómo han de dividirse los costes entre los agentes implicados.

Definición 6: *Una regla de asignación de costes es una función ψ que asigna a cada problema de árboles o arborescencias de coste mínimo (N_0, C) el vector $\psi(N_0, C) \in \mathbb{R}^N$ tal que*

$$\sum_{i \in N} \psi_i(N_0, C) = m(N_0, C),$$

donde $\psi_i(N_0, C)$ es el coste asignado al agente i y $m(N_0, C)$ es el coste mínimo de conexión de los agentes de N a la fuente.

Nótese que, además, $m(N_0, C)$ coincide con el coste asociado al árbol o arborescencia de coste mínimo, sin depender de cuál haya sido considerado.

A la hora de calcular las reglas de reparto que vamos a estudiar podemos recurrir a dos estrategias. Por un lado, tenemos la opción de seguir los pasos resultantes de aplicar alguno de los algoritmos vistos en el capítulo anterior. Y, por otro, está la posibilidad de utilizar la teoría de juegos. Para esta última estrategia será necesario definir antes lo que entendemos por juegos cooperativos, que son aquellos que tratan de modelar situaciones en los que el conjunto de agentes pueden negociar entre ellos para beneficiarse de la cooperación. De ellos, nos centraremos en los juegos con utilidad transferible (juegos TU), en los que la utilidad puede repartirse de cualquier modo entre los distintos agentes. Nos interesa, además, la expresión del juego en su forma característica.

Definición 7: *Un juego cooperativo con utilidad transferible (juego TU) en forma característica viene dado por (N, v), donde*

- *$N = \{1, \ldots, n\}$ es el conjunto de jugadores;*
- *v es la función característica, que asocia a cada coalición $S \subset N$ un número real $v(S)$, que representa el valor de la coalición S. Además, se asume que $v(\emptyset) = 0$.*

Cuando utilicemos esta estrategia, lo que haremos será obtener un juego cooperativo de costes asociado a un problema de árboles o arborescencias de coste mínimo y calcular a partir de él un reparto. Estos juegos nos servirán, además, a la hora de representar gráficamente las diferentes reglas de reparto que implementemos en el paquete `cooptrees`. Para ello utilizaremos grafos de ejemplo en los que contamos con una fuente a la que se conectan tres agentes y sobre cuyo juego cooperativo asociado calcularemos y representaremos el conjunto de imputaciones y el núcleo.

Definición 8: *Sea (N, v) un juego TU , su conjunto de imputaciones (costes) se define como:*

$$I(N,v) = \left\{ x \in \mathbb{R}^n : x_i \leq v(\{i\}) \ \forall i \in N, \ \sum_{i \in N} x_i = v(N) \right\};$$

y contiene los repartos de los jugadores de forma que se distribuya entre ellos lo que pagarían si todos cooperan y además no se defrauda a ningún jugador.

Definición 9: *El núcleo de un juego (N, v) (costes) viene dado por:*

$$C(N,v) = \left\{ x \in I(N,v) : \sum_{i \in S} x_i \leq v(S), \ \textit{para todo } S \subset N, \ S \neq \emptyset \right\};$$

y representa las asignaciones seleccionadas de forma que no se defrauda a ninguna coalición.

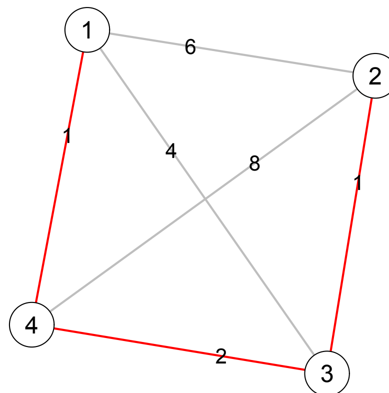
De llevar a cabo esto último se encargará un script de R adicional incorporado al paquete `cooptrees` que contiene una serie de funciones programadas específicamente para el cálculo y representación del conjunto de imputaciones y del núcleo en juegos de tres jugadores. Estas funciones, que no se detallan en este documento por razones de extensión, tienen como base la librería TUGlab[17] de Matlab y están contenidas en el archivo `coop_games.R` del paquete.

3.2. Cooperación en problemas de árboles de coste mínimo

Para abordar el tema de la cooperación en problemas de árboles de coste mínimo tomaremos un ejemplo en el que el nodo 1 se corresponde con la fuente y los nodos 2, 3 y 4 con los diferentes agentes[2]. Introducimos los datos en R y obtenemos un árbol de coste mínimo del problema con la función correspondiente implementada en el paquete `optrees`.

```
157 # Problema de árboles de coste mínimo
158 nodes <- c(1, 2, 3, 4)
159 arcs <- matrix(c(1,2,6, 1,3,4, 1,4,1, 2,3,1, 2,4,8, 3,4,2),
160                byrow = TRUE, ncol = 3)
161 # Solución
162 getMinimumSpanningTree(nodes, arcs, algorithm = "Prim")
```

```
Minimum Cost Spanning Tree
Algorithm: Prim
Stages:  3 | Time:  0
------------------------------
     ept1     ept2    weight
        1        4         1
        4        3         2
        3        2         1
------------------------------
                  Total = 4
```



Como ya hemos dicho, nos interesa obtener el reparto de la suma de costes de los arcos del árbol encontrado entre los diferentes agentes (nodos 2, 3 y 4) que se unen a la fuente (nodo 1). Para ello hay muchas reglas propuestas en la literatura relativa a los problemas de árboles de mínimo coste. Aquí se implementan las 4 reglas más conocidas: Bird, Dutta-Kar, Kar y ERO.

[2]Aunque en la teoría es habitual identificar al nodo fuente con el 0 y a los agentes entre 1 y n, la implementación de `optrees` nos obliga a identificar a los nodos a partir del 1, tal y como hemos hecho.

3.2.1. La regla de Bird

La primera de las reglas de asignación de costes sobre el problema de árboles de coste mínimo que implementaremos en R es la regla de Bird[18]. Esta basa el cómputo del reparto en los pasos del algoritmo de Prim, de forma que cada agente se conecta secuencialmente a la fuente siguiendo dicho algoritmo y paga su correspondiente coste de conexión a su predecesor inmediato.

Regla 1: *Dado un problema de árboles de coste mínimo (N_0, C), la regla de Bird para la asignación de costes se define como:*

$$B_i(N_0, C) = c_{i^0 i}$$

donde i^0 representa el predecesor inmediato de i.

De acuerdo a esta regla, cada jugador paga, por tanto, el coste de un único arco del árbol. En concreto, paga el coste de aquel arco que le conecta, directa o indirectamente, a la fuente. Esto facilita mucho la programación en R, pues solo será necesario recopilar los costes de los distintos arcos del árbol de coste mínimo obtenido y asignárselos al nuevo agente que conectan. El código está implementado en la función `mstBird`, que devuelve como resultado la lista de agentes y el coste que ha de pagar cada uno.

```
164 # Regla de Bird
165 mstBird(nodes, arcs)
```

```
     agent cost
[1,]     4    1
[2,]     3    2
[3,]     2    1
```

Es necesario señalar aquí que la regla de Bird, tal y como la hemos definido, depende del árbol de coste mínimo que hayamos escogido en el algoritmo de Prim. Dado que nuestra implementación del algoritmo de Prim en el paquete `optrees` se limita a encontrar un único árbol de coste mínimo, aunque haya varios, la regla de Bird solo se calcula sobre él.

3.2.2. La regla de Dutta-Kar

Una segunda regla de reparto para los problemas de árboles de coste mínimo es la regla de Dutta-Kar[19], la cual vuelve a recurrir al algoritmo de Prim para repartir los costes entre los distintos agentes. De esta forma, los agentes se conectan otra vez secuencialmente a la fuente según dicho algoritmo, pero, en esta ocasión, se reparten los costes de forma que al ser conectado cada agente escoge el mínimo coste entre el que le correspondería y el de su sucesor inmediato.

Regla 2: *Dado un problema de árboles de coste mínimo (N_0, C) y suponiendo que los agentes se conectan de acuerdo al algoritmo de Prim en el orden $1, 2, \ldots, n$, la regla de Dutta-Kar reparte los costes mediante las siguientes etapas:*

- ***Etapa 1:*** $t_1 = c_{01}$
- ***Etapa k,*** $2 \leq k \leq n$***:*** $t_k = max\{t_{k-1}, c_{k-1k}\}$; $DK_{k-1}(N_0, C) = min\{t_{k-1}, c_{k-1k}\}$

El proceso continúa hasta la etapa n*, en la que* $DK_n(N_0, C) = t_n$*.*

Se puede ver como la formulación de la regla de Dutta-Kar obliga a mantener almacenado en todo momento un coste pendiente (t_k), que será el no escogido por cada agente al incorporarse al árbol. Cuando se conecte un nuevo agente podrá escoger el coste mínimo entre el coste pendiente y el del siguiente arco que parte de él. Todo este proceso lo llevaremos a cabo mediante la función `mstDuttaKar`, que devolverá la lista de agentes con el coste asignado a cada uno de ellos.

```
167 # Regla de Dutta-Kar
168 mstDuttaKar(nodes, arcs)
```

```
     agent cost
[1,]     4    1
[2,]     3    1
[3,]     2    2
```

Como ocurría con la regla de Bird, la regla de Dutta-Kar depende del árbol de coste mínimo escogido en el algoritmo de Prim. Al igual que entonces, en nuestro código solo está previsto el cálculo para el único árbol que obtiene la función `msTreePrim` del paquete `optrees`.

3.2.3. La regla de Kar

La tercera de las reglas de reparto que implementaremos para los problemas de árboles de coste mínimo opta por una estrategia distinta a las anteriores y basa su cálculo en la teoría de juegos cooperativos. Mediante esta estrategia, la regla de Kar[20] se obtiene al calcular el valor de Shapley del juego pesimista asociado al problema.

Regla 3: *Dado un problema de árboles de coste mínimo* (N_0, C)*, la regla de Kar para la asignación de costes se define como:*

$$K(N_0, C) = Sh(N, v_C)$$

donde (N, v_C) *representa el juego pesimista asociado al problema.*

El juego cooperativo asociado a un problema de árboles de mínimo coste es un juego TU según el cual cada coalición pagaría su coste mínimo de conexión asumiendo que los agentes que no pertenecen a la coalición no están presentes. Este juego se conoce como el juego pesimista[18].

Definición 10: *Dado un problema de árboles de mínimo coste* (N_0, C)*, el juego pesimista asociado es un par* (N, v_C) *tal que*

$$v_c(S) = m(S_0, C)$$

para toda coalición $S \subset N$*.*

Para poder obtener el juego pesimista en R programamos la función `mstPessimistic`. Esta función comprueba todas las posibles coaliciones que pueden acordar los agentes presentes en el problema y obtiene el valor correspondiente al árbol de coste mínimo de cada una. El resultado que devuelve es la función característica del juego pesimista.

```
170 # Juego pesimista asociado a un problema de árboles de coste mínimo
171 mstPessimistic(nodes, arcs)
```

```
$coalitions
[1] "1"     "2"     "3"     "1,2"   "1,3"   "2,3"   "1,2,3"

$values
[1] 6 4 1 5 7 3 4
```

Como hemos dicho antes, nos interesa calcular el valor de Shapley[21] del juego pesimista. Este se obtiene promediando los vectores de contribuciones marginales asociados a todos los posibles órdenes de los jugadores.

Definición 11: *Dado un juego TU (N,v), el valor de Shapley está dado por*

$$Sh_i(N,v) = \sum_{S \subset N \setminus \{i\}} \frac{s!(n-s-1)!}{n!}(v(S \cup \{i\}) - v(S))$$

para todo $i \in N$.

Para calcular el valor de Shapley programamos una nueva función (`shapleyValue`), con la que generamos todas las posibles coaliciones de jugadores y comprobamos todos los órdenes hasta construir la matriz de contribuciones marginales. Promediando el valor para cada jugador, la función obtiene el valor de Shapley, el cual devuelve junto a la matriz de contribuciones.

```
173 # Valor de Shapley del juego pesimista
174 shapleyValue(n = 3, v = mstPessimistic(nodes, arcs)$values)
```

```
$contributions
     [,1] [,2] [,3]
[1,]    6   -1   -1
[2,]    6   -3    1
[3,]    1    4   -1
[4,]    1    4   -1
[5,]    6   -3    1
[6,]    1    2    1

$value
[1] 3.5 0.5 0.0
```

Es importante recordar aquí que el cálculo del valor de Shapley se complica exponencialmente a medida que aumenta el número de jugadores. En el caso de nuestra implementación, la función `shapleyValue` es capaz de obtener rápidamente el valor para juegos de menos de 10 jugadores. A partir de dicho tamaño, el tiempo de ejecución aumenta considerablemente, por lo que se incluye un aviso en esos casos y se solicita una confirmación adicional para continuar con las operaciones.

Con esas dos funciones ya somos capaces de obtener el juego pesimista asociado a un problema de árboles de mínimo coste y de calcular su valor de Shapley. Con ello, estamos en posición de obtener el reparto propuesto la regla de Kar. De esta tarea se encargará la función `mstKar`, la cual llama a las anteriores y devuelve el correspondiente reparto de costes entre los agentes presentes en el problema.

```
176 # Regla de Kar
177 mstKar(nodes, arcs)
```

```
     agent cost
[1,]     2  3.5
[2,]     3  0.5
[3,]     4  0.0
```

3.2.4. La regla ERO

La última de las reglas de asignación de costes que vamos a implementar para los problemas de árboles de coste mínimo es la regla ERO. Bajo ese nombre apareció por primera vez en 1994[22], aunque posteriormente sería propuesta bajo el nombre de "Folk rule"[23]. En este trabajo la denominaremos por su nombre originario, que es regla ERO (Equal Remaining Obligation).

Esta regla puede ser obtenida de varias formas distintas. Nosotros nos centraremos en el estudio de tres de ellas: a partir del algoritmo de Kruskal, a partir del juego optimista y a partir del juego pesimista. En el primer caso se hará uso de uno de los algoritmos utilizados para la obtención del árbol de mínimo coste, mientras que en el segundo y tercer caso se recurrirá a la teoría de juegos cooperativos, ya que, en ambos casos, la regla ERO se puede obtener como el valor de Shapley de un juego cooperativo.

Regla 4: *Dado un problema de árboles de coste mínimo* (N_0, C)*, la regla ERO para la asignación de costes se define como:*

$$ERO(N_0, C) = Sh(N, v_C^+) = Sh(N, v_C^*)$$

donde (N, v_C^+) *es el juego optimista asociado al problema y* (N, v_C^*) *el juego pesimista asociado a la forma irreducible de uno de los árboles de coste mínimo.*

En el juego optimista asociado a un problema de árboles de coste mínimo[24] cada coalición obtiene el mínimo coste de conexión asumiendo que los agentes que no pertenecen a la coalición ya están conectados.

Definición 12: *Dado un problema de árboles de coste mínimo* (N_0, C)*, el juego optimista asociado es un par* (N, v_C^+) *tal que*

$$v_C^+(S) = m\left(S_0, C^{+(N\setminus S)}\right) \text{ para todo } S \subset N,$$

con $c_{ij}^{+(N\setminus S)} = c_{ij}$ *para todos los agentes* $i, j \in S$ *y* $c_{i0}^{+(N\setminus S)} = min_{j\in(N\setminus S)_0}\ c_{ij}$ *para todo* $i \in S$*.*

Al igual que el juego pesimista, el juego optimista también ha sido implementado en el paquete `cooptrees` mediante la función `mstOptimistic`. Dicha función devuelve las coaliciones y la función característica del juego optimista asociado a un problema de árboles de coste mínimo.

```
179 # Juego optimista asociado a un problema de árboles de coste mínimo
180 mstOptimistic(nodes, arcs)
```

```
$coalitions
[1] "1"     "2"     "3"     "1,2"   "1,3"   "2,3"   "1,2,3"

$values
[1] 1 1 1 3 2 2 4
```

Por otro lado, también implementamos en nuestro paquete una función para obtener la forma irreducible de un problema de árboles de coste mínimo[18], que no depende del árbol de coste mínimo que consideremos.

Definición 13: *Dado un problema de árboles de coste mínimo* (N_0, C)*, se define su forma irreducible como el grafo minimal* $(N_0, C^*) = (N_0, C^{(G^T, C^T)})$ *asociado a cualquiera de sus árboles de coste mínimo* (G^T, C^T)*.*

Definición 14: *Dado un problema de árboles de coste mínimo* (N_0, C) *y un árbol de coste mínimo* (G^T, C^T)*, el grafo minimal* $(N_0, C^{(G^T, C^T)})$ *se obtiene reduciendo el coste de un arco* (i, j) *al máximo coste del arco que hay en el único camino de* (G^T, C^T) *que une los dos nodos del arco* (i, j)*.*

La función `mstIrreducible` se encarga de realizar los cálculos necesarios y devuelve el conjunto de arcos que forman el grafo que constituye la forma irreducible del problema. Dado que, en este caso, lo que nos interesa es el juego pesimista asociado a dicha forma irreducible, guarda-

mos los arcos en un objeto y lo introducimos como parámetro en la función `mstPessimistic` implementada en su momento.

```
182 # Forma irreducible de un problema de árboles de coste mínimo
183 arcsIrr <- mstIrreducible(nodes, arcs)
184 # Juego pesimista asociado
185 mstPessimistic(nodes, arcsIrr)
```

```
$coalitions
[1] "1"     "2"     "3"     "1,2"   "1,3"   "2,3"   "1,2,3"

$values
[1] 2 2 1 3 3 3 4
```

Sobre estos dos juegos, el optimista y el pesimista asociado a la forma irreducible, se podría calcular ahora el valor de Shapley, comprobando que en ambos casos obtenemos el mismo reparto de costes, el cual coincide con la regla ERO.

```
187 # Valor de Shapley del juego optimista
188 shapleyValue(n = 3, v = mstOptimistic(nodes, arcs)$values)
```

```
$contributions
     [,1] [,2] [,3]
[1,]    1    2    1
[2,]    1    2    1
[3,]    2    1    1
[4,]    2    1    1
[5,]    1    2    1
[6,]    2    1    1

$value
[1] 1.5 1.5 1.0
```

```
190 # Valor de Shapley del juego pesimista asociado a la forma irreducible
191 shapleyValue(n = 3, v = mstPessimistic(nodes, arcsIrr)$values)
```

```
$contributions
     [,1] [,2] [,3]
[1,]    2    1    1
[2,]    2    1    1
[3,]    1    2    1
[4,]    1    2    1
[5,]    2    1    1
[6,]    1    2    1

$value
[1] 1.5 1.5 1.0
```

El problema de este método es el ya mencionado relativo al aumento de complejidad del cálculo del valor de Shapley en juegos con muchos jugadores. Ante esta situación, parece más recomendable intentar obtener la regla ERO por otras vías, recurriendo al uso de alguno de los algoritmos utilizados para la obtención de árboles de coste mínimo. A diferencia de Bird y Dutta-Kar, este procedimiento cuenta además con la ventaja de no depender del árbol elegido.

Para aplicar este método, podemos optar por seguir los pasos del algoritmo de Kruskal. La idea es que los agentes empiecen con obligaciones de pago asociadas iguales a 1 y que estos se vayan conectando secuencialmente de acuerdo al orden determinado por el algoritmo. En cada iteración solo estarán obligados a pagar aquellos agentes que se beneficien de la conexión, dividiéndose entre ellos las obligaciones a partes iguales, de forma que $O_i(S) = \frac{1}{|S|} \forall S \subset N$. Una vez que un agente se haya conectado a la fuente, su obligación pasa a ser cero.

Para implementarlo en R programamos la función `mstEROKruskal` que realiza todo el proceso descrito anteriormente y devuelve el reparto de costes entre los agentes según la regla de ERO.

```
193 # Regla ERO con algoritmo de Kruskal
194 mstEROKruskal(nodes, arcs)
```

```
     agent cost
[1,]     2  1.5
[2,]     3  1.5
[3,]     4  1.0
```

3.2.5. mstCooperative

Todas las reglas anteriores se recopilan en `mstCooperative`, una de las funciones principales del paquete `cooptrees`. Esta función recibe como parámetros de entrada los nodos y arcos de un grafo y, tras obtener un árbol de coste mínimo, llama al resto de funciones que hemos visto en el presente capítulo para calcular los juegos asociados al problema y las diferentes reglas de reparto. Los resultados se imprimen directamente en consola, mostrando los principales juegos cooperativos y las cuatro reglas cuyo cálculo hemos implementado en el paquete.

Además, en el caso de un problema con tres jugadores como el de nuestro ejemplo, la función hace uso del script adicional del paquete que hemos desarrollado para representar el conjunto de imputaciones y el núcleo del juego pesimista. Sobre ellos representa los repartos de costes obtenidos de acuerdo a las cuatro reglas.

```
196 # Nodos, conexiones y costes del problema
197 nodes <- c(1, 2, 3, 4)
198 arcs <- matrix(c(1,2,6, 1,3,4, 1,4,1, 2,3,1, 2,4,8, 3,4,2),
199                byrow = TRUE, ncol = 3)
200 # Cooperación en un problema de árboles de coste mínimo
201 mstCooperative(nodes, arcs)
```

```
 Minimum Cost Spanning Tree
 Algorithm: Prim
 Stages:  3 | Time:  0
 ------------------------------
      ept1     ept2    weight
         1        4         1
         4        3         2
         3        2         1
 ------------------------------
                  Total = 4

 Cooperative games:
 ---------------------------
     1 2 3 1,2 1,3 2,3 1,2,3
Pes. 6 4 1   5   7   3     4
Opt. 1 1 1   3   2   2     4
Irr. 2 2 1   3   3   3     4
 ---------------------------

 Allocation rules:
 ---------------------------
  Bird D-K Kar ERO
2    1   2 3.5 1.5
3    2   1 0.5 1.5
4    1   1 0.0 1.0
 ---------------------------
   Total cost = 4
```

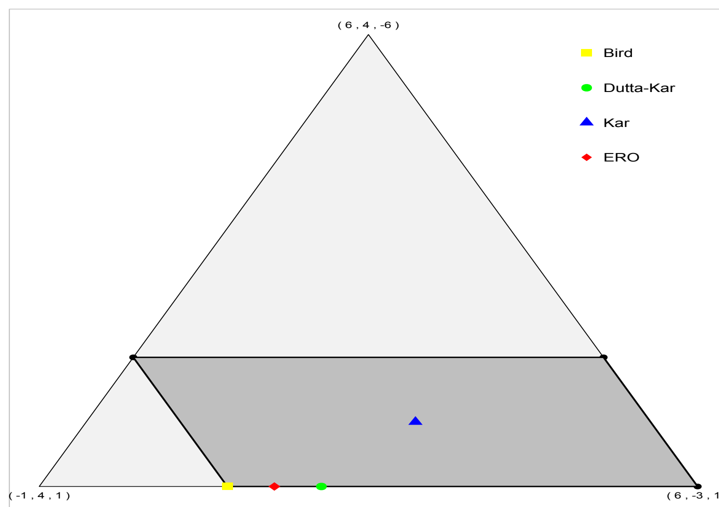


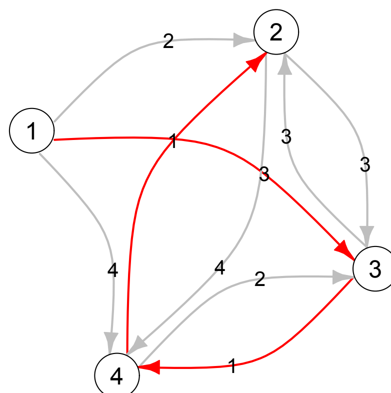
Fig. 3.1.: Conjunto de imputaciones y núcleo del juego pesimista asociado a un problema de árboles de coste mínimo, con las reglas de reparto sobreimpresas.

3.3. Cooperación en problemas de arborescencias de coste mínimo

Al igual que hicimos con los árboles de coste mínimo, para explicar el tema de la cooperación y reparto de costes en los problemas de arborescencias de coste mínimo contamos con un grafo de ejemplo. En él, una vez más, el nodo 1 se corresponde con la fuente y los nodos 2, 3 y 4 con los agentes que se quieren conectar a ella. El problema lo podemos resolver en R con la función implementada para ello en el paquete `optrees`.

```
205 # Problema de arborescencias de coste mínimo
206 nodes <- c(1,2,3,4)
207 arcs <- matrix(c(1,2,2, 1,3,3, 1,4,4, 2,3,3, 2,4,4, 3,2,3,
208                  3,4,1, 4,2,1, 4,3,2), ncol = 3, byrow = TRUE)
209 # Solución
210 getMinimumArborescence(nodes, arcs)
```

```
Minimum cost spanning arborescence
Algorithm: Edmonds
Stages:  2 | Time:  0.01
------------------------------
     head     tail    weight
        1        3         3
        3        4         1
        4        2         1
------------------------------
                  Total = 5
```



3.3.1. La regla de Bird

En los problemas de arborescencias de costes mínimo la primera regla de reparto que implementaremos es, de nuevo, la regla de Bird[25]. Esta se calcula teniendo en cuenta cuál es el nodo predecesor de cada agente en la arborescencia, es decir, el único nodo que está justo antes del agente en el camino que lo conecta a la fuente.

Regla 5: *Dado un problema de arborescencias de coste mínimo (N_0, C) y una arborescencia de coste mínimo $G^{T'}$ asociada a dicho problema, la regla de Bird para la asignación de costes se define como:*

$$B_i(N_0, C, G^{T'}) = c_{i^0 i}$$

donde i^0 representa el predecesor inmediato de i en la arborescencia $G^{T'}$.

Al igual que en los árboles de coste mínimo, implementar en R la regla de Bird para el problema de las arborescencias de coste mínimo es tan inmediato como considerar el coste de cada arco y asignárselo al nodo destino del mismo. Esa tarea la lleva a cabo la función `maBird`, que devuelve la lista de agentes con el coste asociado a cada uno.

```
212 # Regla de Bird
213 maBird(nodes, arcs)
```

```
     agent cost
[1,]     2    1
[2,]     3    3
[3,]     4    1
```

Eso sí, de nuevo aquí nos encontramos con el problema de que la regla de Bird depende de la arborescencia de coste mínimo que hayamos elegido. Y, como ocurría en el apartado anterior, tenemos una implementación del algoritmo de Edmonds en el paquete `optrees` que solo devuelve una única arborescencia de coste mínimo. Por lo tanto, aunque un problema cuente con varias arborescencias de coste mínimo, la regla de Bird solo se calcula sobre una de ellas, aquella encontrada por la función `msArborEdmonds`.

3.3.2. La regla ERO

La otra regla de reparto que implementaremos para los problemas de arborescencias de coste mínimo es la regla ERO[25]. En su cálculo recurrimos de nuevo, tal y como ya hemos hecho en otras ocasiones, a los juegos cooperativos. En este caso al juego pesimista asociado a una forma irreducible de una arborescencia de coste mínimo. La regla ERO será su valor de Shapley.

Esta es la única forma que hay hasta el momento para obtener la regla ERO en los problemas

de arborescencias de coste mínimo. A diferencia de lo que ocurre en el caso de los problemas de árboles de coste mínimo, todavía no se ha descubierto un algoritmo que nos permita calcular la regla ERO sin hacer uso del valor de Shapley.

Regla 6: *Dado un problema de arborescencias de coste mínimo* (N_0, C)*, se define la regla ERO como la regla que propone el reparto*

$$ERO(N_0, C) = Sh(N, v_{C^*})$$

donde (N, v_{C^*}) *es el juego pesimista asociado a una forma irreducible de una arborescencia de coste mínimo.*

Para obtener la forma irreducible de una arborescencia de coste mínimo debemos considerar el grafo obtenido en el último paso del algoritmo de Edmonds. En este, asignamos a cada arco incidente en un nodo distinto de la fuente el menor coste de todos los arcos incidentes en él. A continuación, proseguimos reconstruyendo el grafo etapa por etapa hasta el grafo original sumando en cada una las cantidades que fuimos descontando de cada nodo. Toda esa tarea la lleva a cabo la función `maIrreducible`, que, tomando de entrada una arborescencia y las distintas etapas necesitadas para hallarla, construye la forma irreducible y la devuelve.

```
215 # Forma irreducible de un problema de arborescencias de coste mínimo
216 maIrreducible(nodes, arcs)
```

```
      head tail weight
 [1,]    1    2      1
 [2,]    1    3      3
 [3,]    1    4      2
 [4,]    2    3      3
 [5,]    2    4      2
 [6,]    3    2      1
 [7,]    3    4      1
 [8,]    4    2      1
 [9,]    4    3      2
```

Es importante señalar que en los problemas de arborescencias de coste mínimo, en contraposición a lo que ocurre en el caso de los problemas de árboles de coste mínimo, puede haber más de una forma irreducible. Debido a la falta de algoritmos para encontrar todas ellas, en nuestro paquete construiremos únicamente una forma irreducible, que será la obtenida de acuerdo al algoritmo de Edmonds implementado en el paquete `optrees`.

Entre los juegos cooperativos asociados a un problema de arborescencias de coste mínimo ya hemos visto que nos interesa el juego pesimista. Para su obtención programamos la función `maPessimistic` que genera todas las posibles coaliciones de jugadores y calcula una arborescen-

cia de coste mínimo para cada una. El resultado que devuelve es la función característica del juego pesimista.

```
218 # Juego pesimista asociado a un problema de arborescencias de coste mínimo
219 maPessimistic(nodes, arcs)
```

```
$coalitions
[1] "1"     "2"     "3"     "1,2"   "1,3"   "2,3"   "1,2,3"

$values
[1] 2 3 4 5 5 4 5
```

Con estas dos funciones y la programada previamente para calcular el valor de Shapley ya estamos en posición de obtener la regla ERO. De su cálculo se encargará la función `maERO`, que llama a las anteriores obteniendo primero la forma irreducible, luego el juego pesimista asociado a la misma y, por último, el valor de Shapley de dicho juego. Al final devuelve el conjunto de agentes con los costes que ha de pagar cada uno.

```
221 # Regla ERO
222 maERO(nodes, arcs)
```

```
     agent cost
[1,]     2  1.0
[2,]     3  2.5
[3,]     4  1.5
```

3.3.3. maCooperative

Al igual que hicimos en el apartado anterior, para los problemas de arborescencias de coste mínimo creamos una función general que sirva de recopilación de las anteriores y devuelva los resultados obtenidos para las diferentes reglas, así como el juego cooperativo implementado. Esa es la función `maCooperative`, que recibe los nodos y arcos del grafo como parámetros de entrada e imprime en consola el juego pesimista y los repartos de costes obtenidos.

En esta ocasión también hemos vuelto a recurrir a un ejemplo con un nodo fuente y tres agentes para representar el conjunto de imputaciones y el núcleo del juego pesimista asociado. Sobre ellos, representamos los repartos obtenidos con las dos reglas implementadas para este problema.

```
224 # Jugadores, conexiones y costes del problema
225 nodes <- c(1, 2, 3, 4)
226 arcs <- matrix(c(1,2,2, 1,3,3, 1,4,4, 2,3,3, 2,4,4, 3,2,3,
227                  3,4,1, 4,2,1, 4,3,2), ncol = 3, byrow = TRUE)
228 # Cooperación en un problema de arborescencias de coste mínimo
229 maCooperative(nodes, arcs)
```

```
 Minimum cost spanning arborescence
 Algorithm: Edmonds
 Stages:  2 | Time:  0
 ------------------------------
      head     tail    weight
         1        3         3
         3        4         1
         4        2         1
 ------------------------------
                  Total = 5

 Cooperative games:
 ---------------------------
     1 2 3 1,2 1,3 2,3 1,2,3
Pes. 2 3 4   5   5   4     5
Irr. 1 3 2   4   3   4     5
 ---------------------------

 Allocation rules:
 ---------------------------
  Bird ERO
2    1 1.0
3    3 2.5
4    1 1.5
 ---------------------------
   Total cost = 5
```

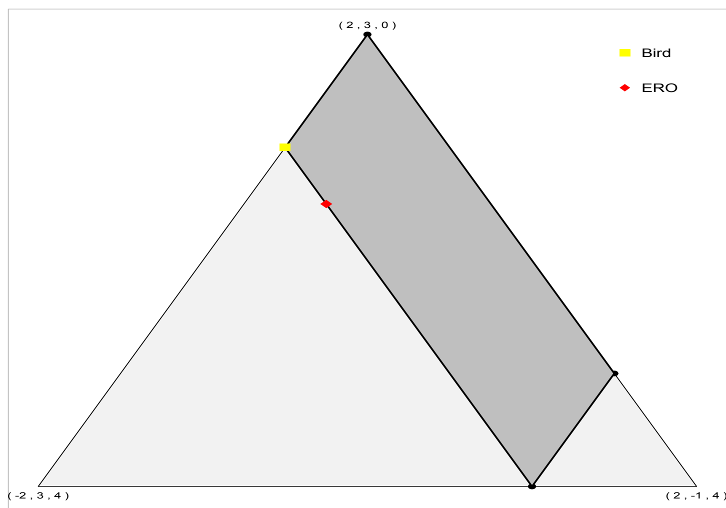


Fig. 3.2.: Conjunto de imputaciones y núcleo del juego pesimista asociado a un problema de arborescencias de coste mínimo, con las reglas de reparto sobreimpresas.

Capítulo 4

OptreesGUI: aplicación web para optrees

"Una imagen vale más que mil palabras.
Una interfaz vale más que mil imágenes."

- Ben Shneiderman.

4.1. Introducción

Los problemas de árboles óptimos, como casi todo lo relacionado con la teoría de grafos, se basan en buena medida en las estructuras gráficas que los representan. No es lo mismo ver una lista de arcos que una imagen representando a los diferentes nodos unidos por ellos. En los paquetes `optrees` y `cooptrees` hemos tratado de suplir esa ausencia con la librería `igraph`, pero esta se encuentra lejos de satisfacer el nivel de interactividad que pueda interesar a los usuarios. Usuarios que además no tienen por qué tener conocimientos de R y para los que se hace necesario pensar en una interfaz de trabajo distinta. De eso nos encargaremos en este capítulo.

Nuestro objetivo es contruir una interfaz gráfica de usuario (GUI por sus siglas en inglés) para `optrees` y `cooptrees`. Para ello haremos uso del paquete Shiny, creado en 2012 por el equipo de RStudio, que permite construir aplicaciones web interactivas directamente desde R. Shiny puede ser utilizado a un nivel inicial para crear aplicaciones simples como las que muestran en su página web, pero el problema es que ninguna de las componentes que pone a nuestra disposición cubren las necesidades del proyecto que nos ocupa. La mayoría están pensadas para el uso y representación de datos estadísticos y ninguna de ellas sirve, en principio, a un ámbito de la Investigación Operativa como es la teoría de grafos.

Afortunadamente, Shiny va mucho más allá de un mero conjunto de componentes prefabricadas y está diseñado de forma que permite desarrollar una aplicación web completa por cuenta propia utilizando HTML, CSS y JavaScript. Todo ello al mismo tiempo que mantiene el código de R como motor encargado de realizar los cálculos y operaciones. La tarea no es sencilla, pues implica modificar componentes y desarrollar otras nuevas desde cero en JavaScript, construyendo todo

su armazón en HTML y CSS. En los siguientes apartados repasaremos todos ese trabajo y explicaremos el funcionamiento de la aplicación `optreesGUI` que hemos construido.

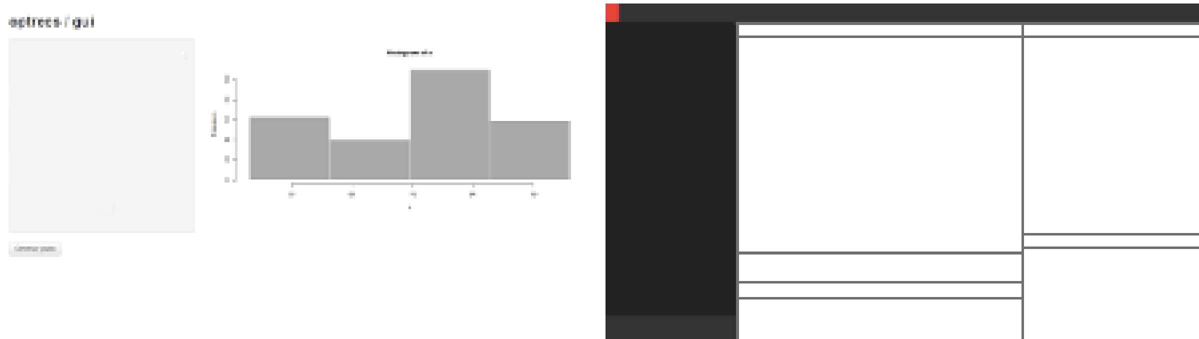
4.2. Desarrollando optreesGUI

4.2.1. Estructura de la web

Ya hemos comentado como Shiny pretende facilitar las cosas a sus usuarios proporcionando un esquema básico de aplicación. Este se basa en dos scripts de R: `server.R` y `ui.R`. El primero es el núcleo de la aplicación, hasta donde llegan los datos introducidos, en donde se procesan y ejecutan las funciones necesarias, y desde donde se envían de vuelta a la web. Su papel es fundamental y su presencia es, por tanto, innegociable. El segundo contiene el código encargado de determinar la estructura y apariencia de la aplicación web. Su uso es opcional y puede ser sustituido por archivos HTML propios que permiten cualquier nivel de personalización. Esta última opción será la utilizada aquí.

Nuestra aplicación web, `optreesGUI`, se compondrá así, principalmente, de un archivo `server.R`, encargado de ejecutar en R las funciones necesarias de los paquetes `optrees` y `cooptrees`; y de un archivo `index.html`, que constituirá el esqueleto de la web e incluirá parte del código JavaScript encargado de gestionar sus diferentes elementos.

Pero una página web es mucho más que su esqueleto en HTML. Para posicionar los distintos elementos que la forman y dotarla de un determinado diseño es necesario crear nuestra propia hoja de estilo CSS. Esta se encuentra contenida en el archivo `style.css`. Junto a la misma, se almacenan los scripts de JavaScript `optrees.js` y `cooptrees.js`, que contienen las componentes de Shiny que desarrollaremos y de las que hablaremos a continuación.

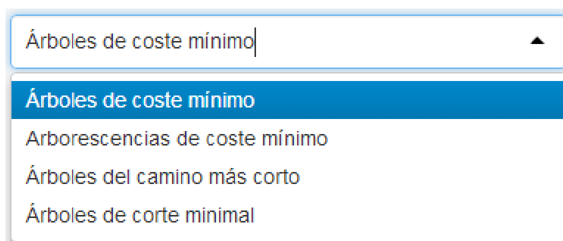


A la izquierda, el esqueleto básico de una web de Shiny. A la derecha, el esqueleto de optreesGUI.

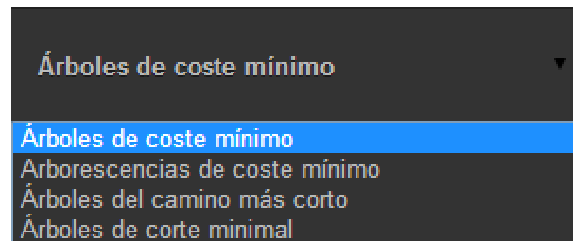
4.2.2. Componentes de la web

La web de `optreesGUI` se divide en dos secciones: un menú lateral desde donde se selecciona el problema de árboles óptimos deseado y se introducen los datos, y un panel central en donde se representa el grafo y los controles que permiten resolverlo. Además, en este último se muestra información adicional relacionada con la solución encontrada. Todos los elementos que componen estas secciones han de ser construidos en mayor o menor medida.

Empezando por el menú de selección de problema, el cual nos permitirá variar entre problemas de árboles de coste mínimo, arborescencias de coste mínimo, árboles del camino más corto y árboles del corte minimal. En este caso se podría utilizar la componente de selección proporcionada por Shiny, pero es conveniente modificar su aspecto para adaptarla al diseño de `optreesGUI`.

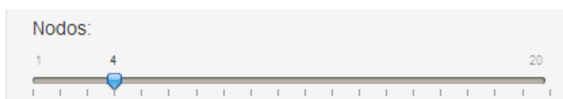


Componente original de Shiny.

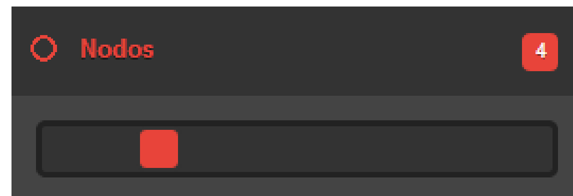


Componente desarrollada para OptreesGUI.

Una vez seleccionado el problema, los elementos principales son los botones deslizantes para introducir el número de nodos y arcos del grafo de partida. Shiny cuenta de nuevo con su propia componente para esta tarea, pero la misma no se adapta al entorno de nuestra web ni a nuestros datos. Por ello, volvemos a rediseñar las componentes y añadimos modificaciones en JavaScript que permitan ocultarlas y bloquearlas cuando sea necesario. Las diferencias entre el elemento básico de Shiny y el nuestro se pueden ver a simple vista:



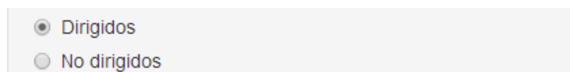
Componente original de Shiny.



Componente desarrollada para OptreesGUI.

La componente para la selección del número de arcos tiene el mismo aspecto y funcionamiento que el de nodos, pero a ella le acompañan otros elementos igual de importantes. En primer lugar, una nueva componente permite seleccionar entre el tipo de arcos que vayamos a utilizar: dirigidos o no dirigidos. En el caso de los problemas de árboles de coste mínimo y de arborescencias de

coste mínimo la opción estará fijada en no dirigidos y dirigidos, respectivamente; pero en el caso del árbol del camino más corto podremos decidir qué tipo de arco escoger. Shiny tiene su propia componente para ello, pero, una vez más, su funcionamiento y diseño debe ser adaptado.

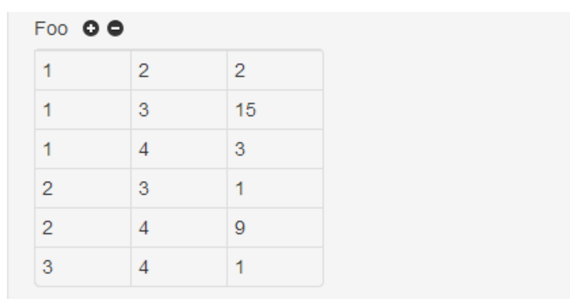


Componente original de Shiny.

Componente desarrollada para OptreesGUI.

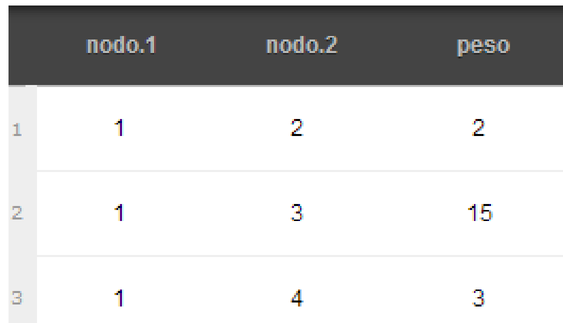
Debajo de la opción con el tipo de arcos, se extenderá una lista desplegable conforme seleccionemos el número de arcos de nuestro grafo original. Cada fila de la lista representa un arco, cuyos dos primeros elementos se corresponden con los nodos que une y el tercero con su peso. Los datos se pueden introducir una vez se haya fijado el número de nodos y arcos del grafo.

Esta componente en forma de lista ha tenido que ser construida desde cero debido a la falta de otras prefabricadas en Shiny y la no adecuación de terceras opciones[26]. La base de su funcionamiento se encuentra en el archivo `server.R`, cuyo código genera las filas de entradas de tipo númerico necesarias según la cantidad de arcos seleccionados. Este método evita tener que introducir aquí código JavaScript adicional, más allá del encargado de limitar el tipo de entrada según el problema, permitiendo recoger los datos directamente para su uso en R.



1	2	2
1	3	15
1	4	3
2	3	1
2	4	9
3	4	1

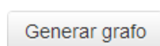
Componente de ShinyIncubator.



	nodo.1	nodo.2	peso
1	1	2	2
2	1	3	15
3	1	4	3

Componente desarrollada para OptreesGUI.

El menú lateral lo completa un botón que tiene por función generar el grafo con los datos introducidos. De nuevo, se trata de una componente de Shiny cuyo funcionamiento y aspecto ha de ser modificado para adaptarse a nuestras necesidades.



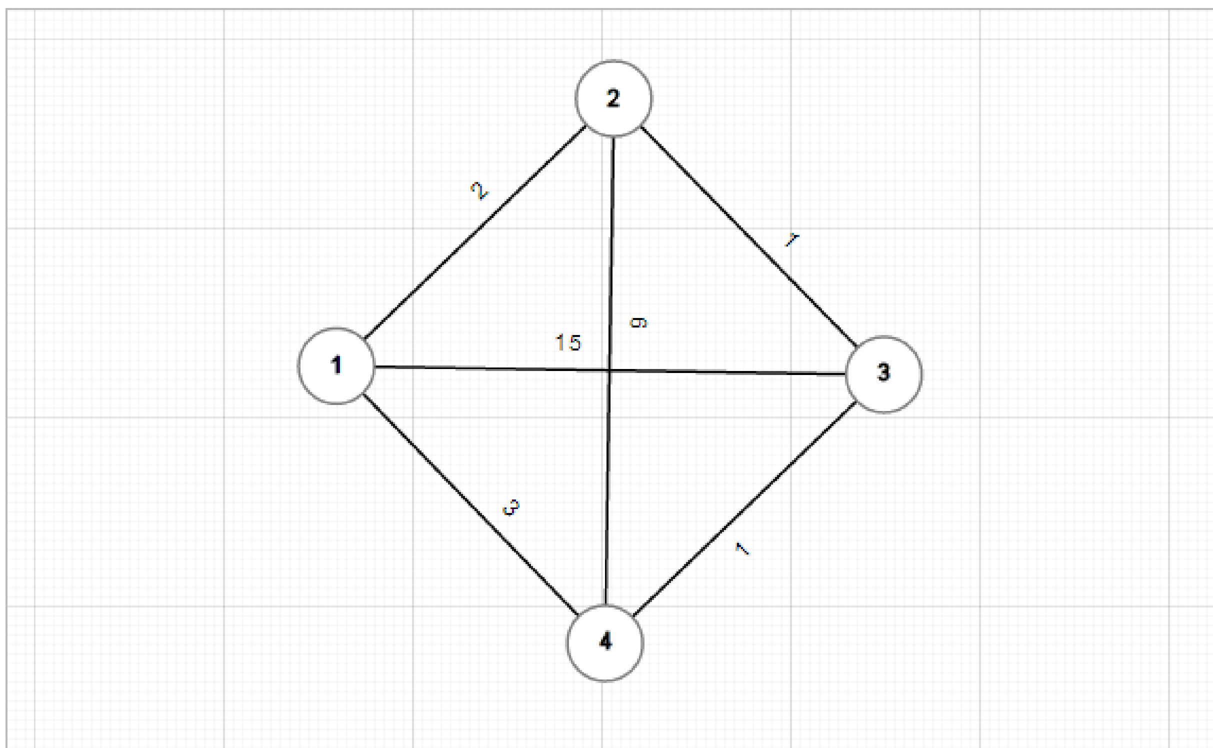
Componente original de Shiny.

Componente desarrollada para OptreesGUI.

Hasta este momento, salvo la lista de arcos, los componentes que hemos utilizado para la entrada de datos cuentan con homólogos en Shiny que podían ser modificados, por lo que, al margen de controlar ciertos aspectos de su funcionamiento, no se requería la programación en JavaScript de los mismos. Esto cambia en cuanto nos planteamos la representación del grafo y su resolución. La falta de herramientas en Shiny para este tipo de circunstancias nos obliga a crear unas propias.

En esa tarea nos servirá de gran ayuda una librería de JavaScript especialmente pensada para la manipulación de documentos basados en datos: D3.js. Esta librería cuenta entre sus herramientas con una preparada especialmente para la representación de diagramas al estilo de nodos y arcos, similares a los utilizados por los grafos, que servirá a nuestro propósito y nos permitirá dotar a nuestras representaciones de movimiento e interactividad.

Eso sí, su incorporación a la web no es inmediata. Necesitamos programar en JavaScript el código necesario para adaptar la implementación de D3.js a los problemas que estudiamos en `optrees` y a nuestra aplicación. Para ello creamos el script `optrees.js`, que se encargará de gestionar la representación de nuestro grafo de forma que en ella se reflejen los nodos con su correspondiente etiqueta, así como los arcos y el peso de cada uno de ellos. Este código estará vinculado a Shiny a través de una nueva componente desarrollada por nosotros que recoge los datos introducidos en el menú y tratados en R y los transforma en objetos de JavaScript entendibles por la librería D3.js. Una serie de funciones se encargarán posteriormente de producir la representación.



Grafo representado en optreesGUI con Shiny y D3.js.

En la captura anterior se puede ver un ejemplo de la representación lograda con la combinación de Shiny y D3.js. El resultado es un grafo interactivo que, partiendo de las posiciones iniciales determinadas por la librería D3.js, permite seleccionar y arrastrar cada uno de los nodos, fijándolos en la ubicación que se desee.

Este grafo representa los datos de entrada de uno de nuestros problemas, pero no contiene aún su solución. Para obtenerla, creamos una barra de control dotada de dos componentes, una encargada de seleccionar el algoritmo a utilizar y otra compuesta por un botón de ejecución. Ambas se modifican de manera similar a como ya hicimos anteriormente para adaptarlas a nuestra aplicación web y gestionar su comportamiento de acuerdo a nuestras necesidades, de manera que muestren los algoritmos pertinentes en cada momento y se active o desactive el botón de ejecución según diferentes situaciones.

Barra de control desarrollada para OptreesGUI.

Junto a la barra de control anterior, se incluye un panel a modo de consola que utilizaremos para enviar mensajes de aviso al usuario. Ahí informaremos sobre posibles errores que haya con los datos introducidos o con cuestiones relativas a la resolución de los diferentes problemas. Una vez más, se trata de una componente modificada a partir de la propuesta por Shiny para mostrar la impresión de datos en una consola de R.

Componente de consola desarrollada para OptreesGUI.

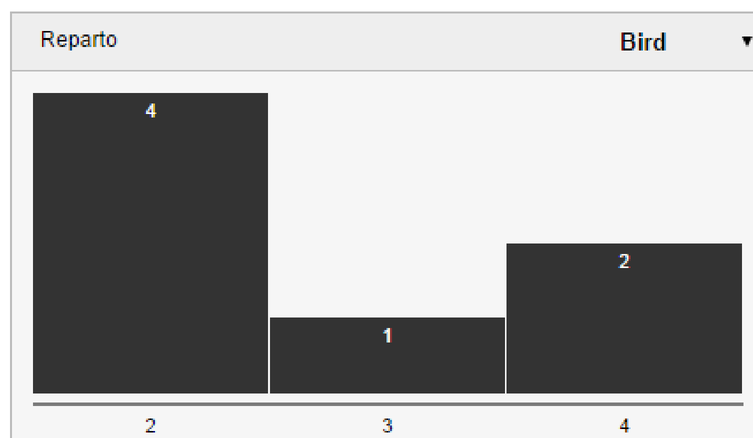
Al solicitar la resolución del problema que hayamos introducido, nuestro script `optrees.js` se encargará de marcar en el grafo los arcos que forman la solución encontrada. Dicha solución se marcará además sobre una lista de arcos presente en el lateral derecho de la web y creada gracias a una componente de Shiny disponible libremente en internet[27].

Arcos

ept1	ept2	weight
1	2	4
1	3	5
1	4	7
2	3	1
2	4	2
3	4	3

Lista de arcos con aquellos que componen la solución encontrada marcados en rojo.

En el caso de los problemas de árboles de coste mínimo y de arborescencias de coste mínimo, los resultados mostrados se completan con los obtenidos mediante el paquete `cooptrees`, para los que se reserva buena parte de la columna derecha del panel principal. Para mostrar estos volvemos a recurrir a la librería D3.js y programamos en JavaScript un script adicional denominado `cooptrees.js`. En él se crea una nueva componente de Shiny que varía según el tipo de problema que hayamos introducido. En el caso de los problemas de árboles de coste mínimo o arborescencias de coste mínimo, muestra los resultados de calcular las diferentes reglas de reparto en forma de gráficos de barra, otorgando a cada uno de los nodos distintos de la fuente (el nodo 1) el coste que ha de pagar según la regla seleccionada en la barra superior.



Reglas de reparto obtenidas con `cooptrees`.

Cuando estamos ante un problema de árboles del camino más corto o un problema de árboles del corte minimal, esta componente varía. En los primeros mostrará la lista de arcos que constituyen

el camino más corto entre los dos nodos seleccionados en la barra superior, mientra que en los segundos mostrará los arcos que forman el conjunto de corte mínimo en el grafo original entre los dos nodos escogidos.

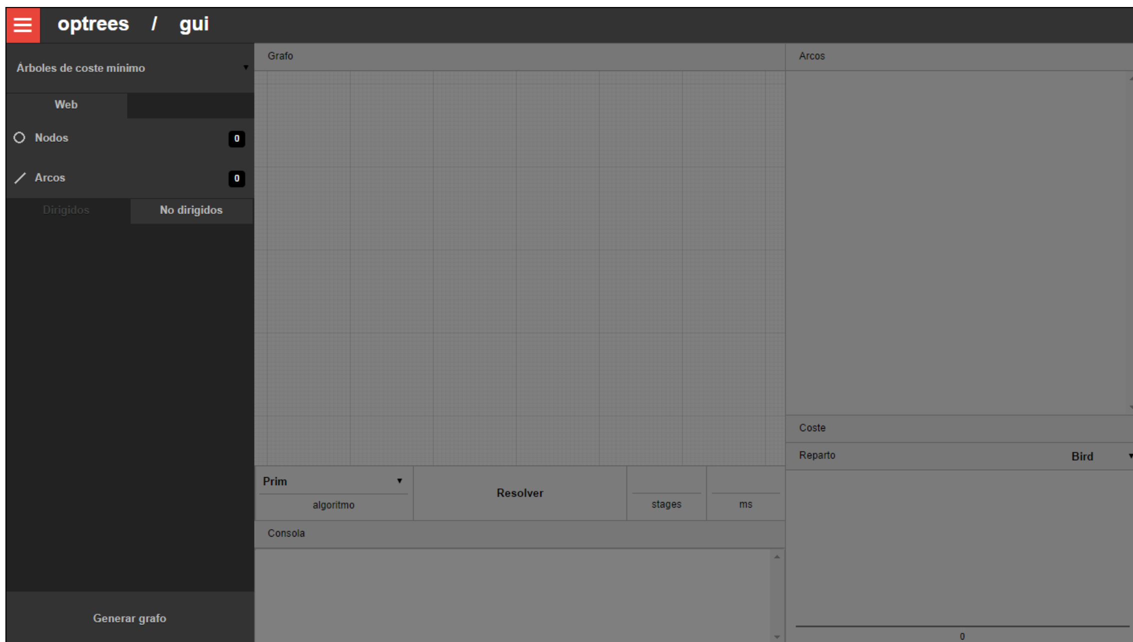
Todos los elementos anteriores completan la aplicación web `optreesGUI`, que constituye nuestra propuesta de interfaz para los paquetes `optrees` y `cooptrees`. Con ella se pueden llevar a cabo buena parte de las operaciones a realizar con ambos sin necesidad de recurrir a R en ningún momento. El límite lo pone el tamaño. Para asegurar el buen funcionamiento de la web se fija un máximo de 20 nodos y 40 arcos en los problemas introducidos. Para grafos de tamaño y orden superior sigue siendo necesario acudir a R e instalar sendos paquetes, aunque en ese tipo de problemas tiene menos utilidad hacer uso de las representaciones gráficas debido a la cantidad de elementos a mostrar.

4.3. Manual de usuario

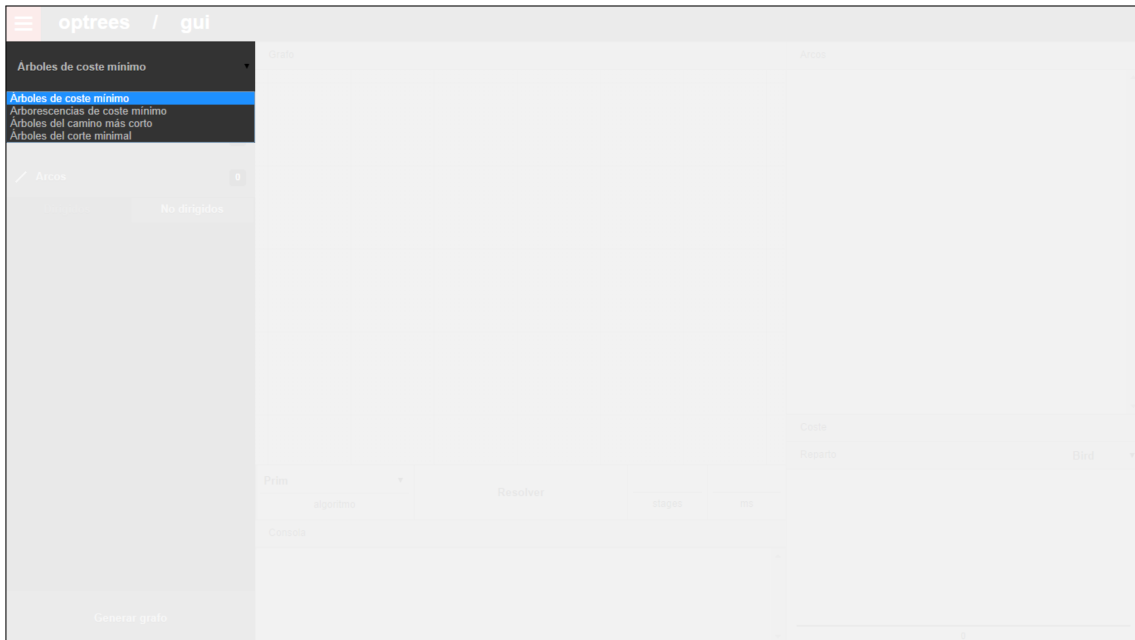
El objetivo a la hora de desarollar `optreesGUI` era facilitar el trabajo con `optrees` y `cooptrees`, proporcionando una interfaz visual que no requiera de conocimientos de R para su utilización. Los paquetes creados en los capítulos anteriores siguen constituyendo el motor del proyecto, pero `optreesGUI` simplifica sus operaciones hasta reducirlas a unos pocos clics de ratón.

Para acceder a ella basta introducir la dirección web optrees.net en el navegador de Internet, preferiblemente Google Chrome, y seguir los pasos que resumimos a continuación. Con ellos se puede resolver un problema de árboles de coste mínimo, pero son homólogos a los necesarios para resolver problemas de arborescencias de coste mínimo, de árboles del camino más corto o de árboles del corte minimal.

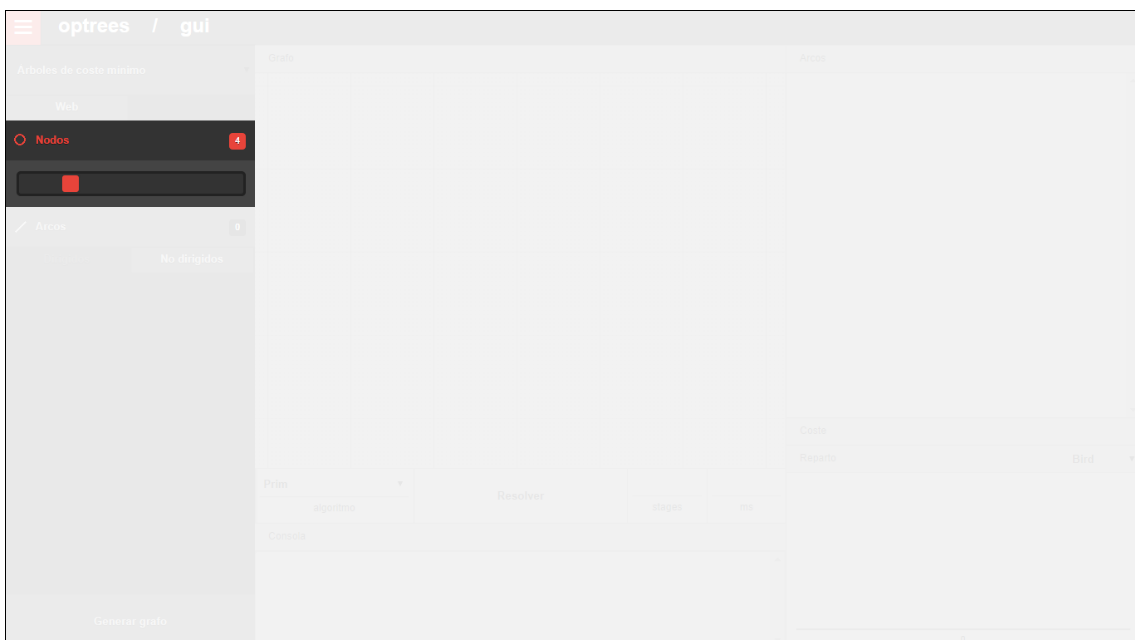
Interfaz inicial de optreesGUI.



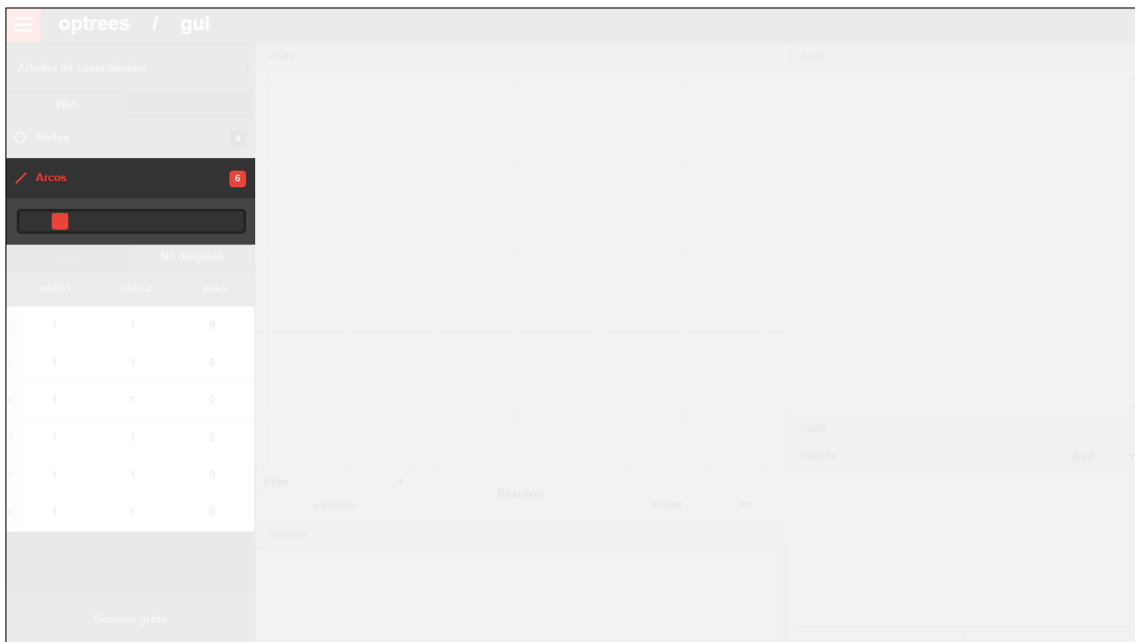
1. Seleccionar el problema.



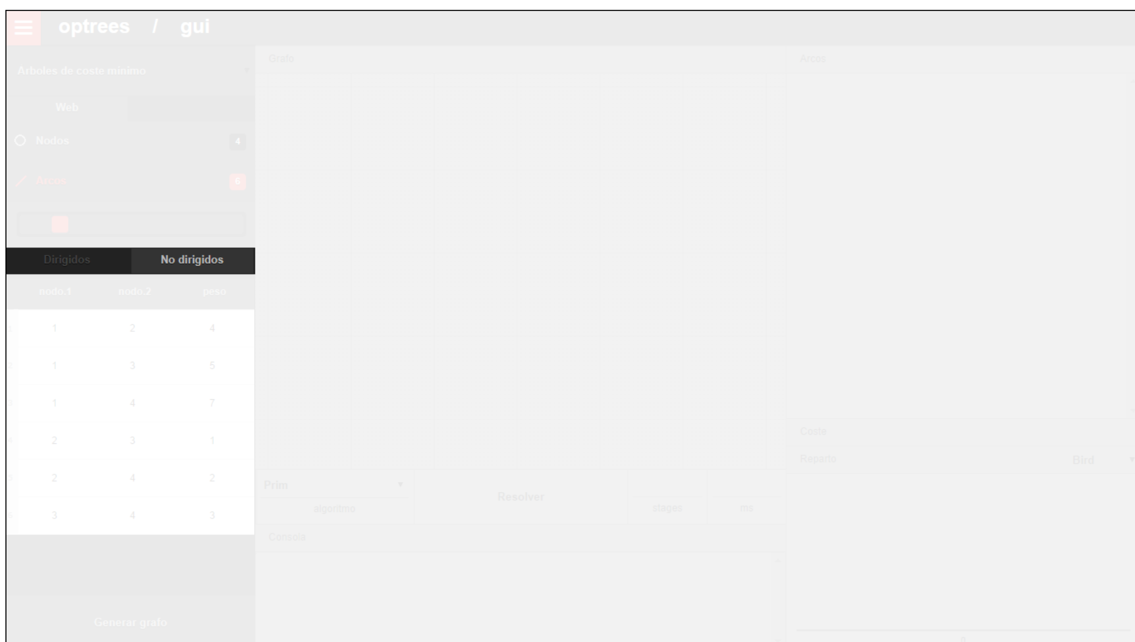
2. Indicar número de nodos.



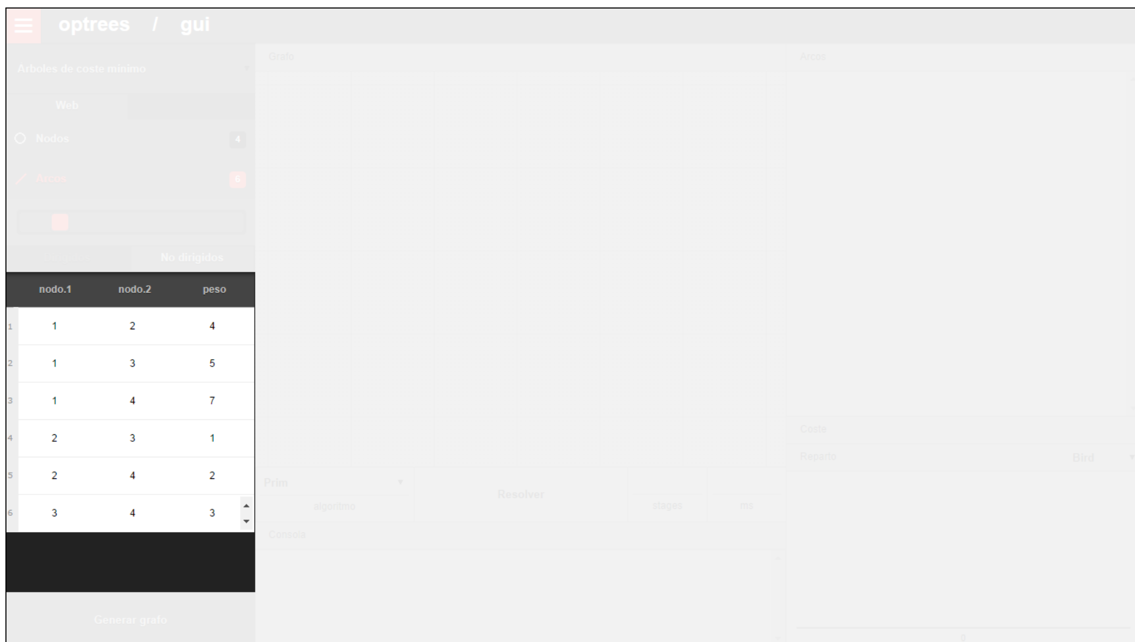
3. Indicar número de arcos.



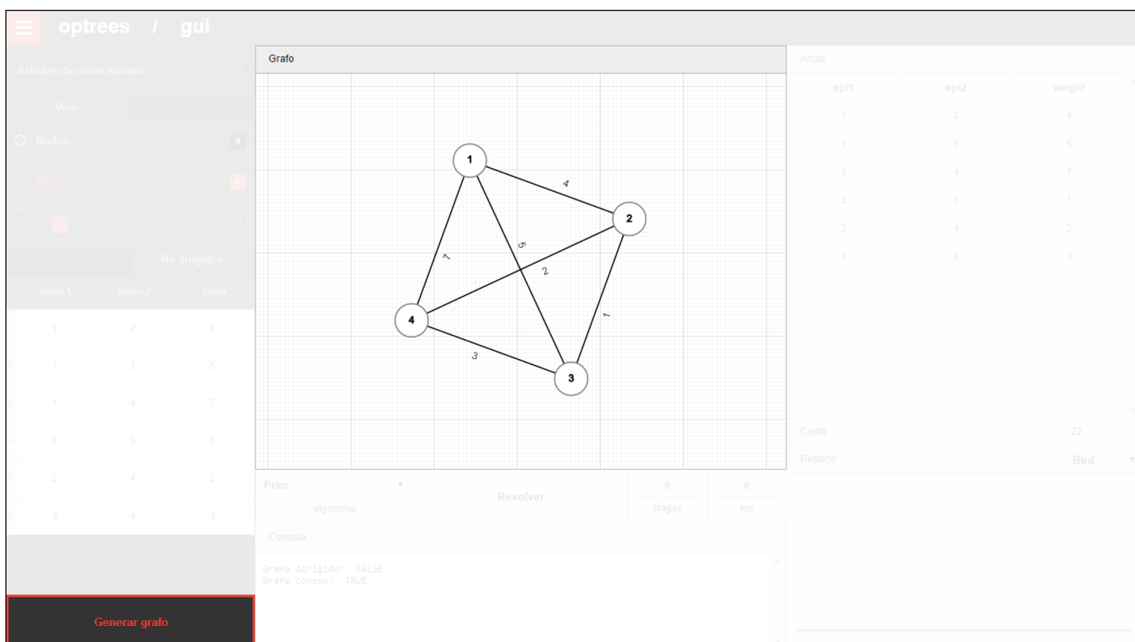
4. Seleccionar tipo de arcos (si es necesario).



5. Introducir la lista de arcos.



6. Generar grafo.



7. Seleccionar algoritmo de resolución.

8. Resolver el problema.

9. Seleccionar regla de reparto o nodos del camino a comprobar.

10. Reiniciar o editar el grafo.

Capítulo 5

Conclusiones y trabajo futuro

En las páginas de este documento hemos tratado de explicar todo el trabajo desarrollado para la implementación en R de problemas de árboles óptimos, sus principales algoritmos de solución y aspectos cooperativos relacionados con todo ellos. Hemos intentado dar buena cuenta de la teoría, especificando una notación que intenta ser lo más común posible y tratando de exponer en pseudocódigo y en fórmulas los principales algoritmos, reglas y juegos de la literatura, siendo nuestra intención definir una forma de trabajar con todos ellos en R.

Con todo, el núcleo duro del trabajo lo constituye el código de los dos paquetes: `optrees` y `cooptrees`. Las versiones `1.0` de ambos han sido subidas a los repositorios CRAN de R. Su código fuente está allí disponible para consulta, además de en el CD que acompaña a este documento, entre cuyas carpetas están contenidos todos los archivos que los forman.

El proyecto lo completa una interfaz web que permite resolver problemas de árboles óptimos: optreesGUI. Como ya hemos explicado, esta funciona sobre los paquetes `optrees` y `cooptrees` y la librería `Shiny` de R, y ha sido desarrollada con HTML, CSS y JavaScript.

La interfaz puede ser utilizada directamente, sin necesidad de abrir R o instalar nada, accediendo desde un navegador, preferiblemente Google Chrome, a la web optrees.net. Si la web no está operativa, en la carpeta con el código de optreesGUI contenida en el CD que acompaña a este documento se encuentra su código fuente, pudiendo ejecutarse una versión de la misma desde la consola de R siguiendo las instrucciones del Capítulo 1 de este documento.

Dicho lo anterior, la documentación y el código desarrollado no agota aquí todas sus posibilidades y da pie a numerosos trabajos futuros que completen y amplíen lo ya hecho. Ya sea en los paquetes o en la interfaz web, hay muchas posibles mejoras e incorporaciones que pueden ser abordadas en el futuro. Aquí nos limitamos a mencionar algunas de ellas.

- **Paquete `optrees`:**
  - Mejorar las funciones relativas a los problemas de árboles de coste mínimo y de arborescencias de coste mínimo, añadiendo la posibilidad de obtener, no solo una, sino todos las soluciones posibles.
  - Relacionado con la anterior, modificar las funciones desarrolladas para el cálculo de las reglas de Bird y Dutta-Kar de forma que puedan trabajar con situaciones en las que tenemos más de un árbol de coste mínimo (o también, en el caso de la regla de Bird, más de una arborescencia de coste mínimo).
- **Paquete `cooptrees`:**
  - Incorporar la representación gráfica para situaciones con 4 jugadores.
  - Estudiar la posibilidad de extender el paquete a los problemas de árboles del camino más corto y a los problemas del corte minimal.
  - Optimizar la función desarrollada para el cálculo del valor de Shapley, intentando reducir su tiempo de ejecución en juegos con más de 10 jugadores.
  - Crear un paquete más general de juegos cooperativos partiendo de las funciones ya desarrollado para el cálculo y representación del conjunto de imputaciones y el núcleo.
- **Interfaz `optreesGUI`:**
  - Mejorar diseño y rendimiento de la web, solucionando posibles deficiencias relativas al uso de navegadores no soportados.
  - Modificar la estructura de la web para facilitar más su uso desde dispositivos táctiles como tablets o smartphones.
  - Añadir una explicación detallada de los pasos dados por cada algoritmo para encontrar una solución a los diferentes problemas.
  - Incorporar la opción de introducir los datos del problema a través de un archivo.
  - Introducir la posibilidad de generar informes que puedan ser descargados con el problemas introducido y su solución.

Anexo A

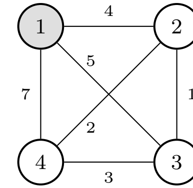
Ejemplos Algoritmos

A.1. Algoritmo de Prim

Entrada

$$V = \{1,2,3,4\}$$
$$A = \{(1,2),(2,1),(1,3),(3,1),(1,4),(4,1),(2,3),(3,2),(2,4),(4,2),(3,4),(4,3)\}\ ^*$$
$$s = 1$$

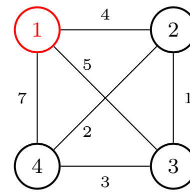
$$C = \begin{pmatrix} \mathtt{NA} & 4 & 5 & 7 \\ 4 & \mathtt{NA} & 1 & 2 \\ 5 & 1 & \mathtt{NA} & 3 \\ 6 & 2 & 3 & \mathtt{NA} \end{pmatrix}$$



* Al tratarse de un grafo no dirigido consideramos cada arco dos veces, en una y otra dirección.

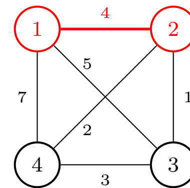
Inicio

1: $A^T \leftarrow \emptyset$
2: $V^T \leftarrow \{s\} = \{1\}$
3: **mientras** $|V^T| < |V|$ **hacer**



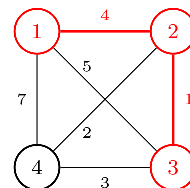
Etapa 1

4: seleccionar un arco $(i,j) \in A$ con $i \in V^T$ y $j \in V \setminus V^T$ cuyo c_{ij} sea mínimo
$\text{mín}\{c_{12}, c_{13}, c_{14}\} = c_{12} = 4$
5: $A^T \leftarrow A^T \cup \{(1,2)\} = \{(1,2)\}$
6: $V^T \leftarrow V^T \cup \{2\} = \{1,2\}$



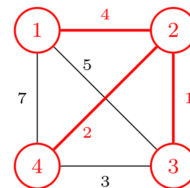
Etapa 2

4: seleccionar un arco $(i,j) \in A$ con $i \in V^T$ y $j \in V \setminus V^T$ cuyo c_{ij} sea mínimo
$\text{mín}\{c_{13}, c_{14}, c_{23}, c_{2,4}\} = c_{23} = 1$
5: $A^T \leftarrow A^T \cup \{(2,3)\} = \{(1,2),(2,3)\}$
6: $V^T \leftarrow V^T \cup \{3\} = \{1,2,3\}$



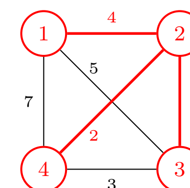
Etapa 3

4: seleccionar un arco $(i,j) \in A$ con $i \in V^T$ y $j \in V \setminus V^T$ cuyo c_{ij} sea mínimo
$\text{mín}\{c_{14}, c_{24}, c_{34}\} = c_{24} = 2$
5: $A^T \leftarrow A^T \cup \{(2,4)\} = \{(1,2),(2,3),(2,4)\}$
6: $V^T \leftarrow V^T \cup \{4\} = \{1,2,3,4\}$
7: **fin mientras** $|V^T| = |V|$



Salida

$$V^T = \{1,2,3,4\}$$
$$A^T = \{(1,2),(2,3),(2,4)\}$$

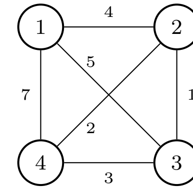


A.2. Algoritmo de Kruskal

Entrada

$$V = \{1,2,3,4\}$$
$$A = \{(1,2),(2,1),(1,3),(3,1),(1,4),(4,1),(2,3),(3,2),(2,4),(4,2),(3,4),(4,3)\}\ ^{*}$$

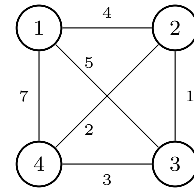
$$C = \begin{pmatrix} \texttt{NA} & 4 & 5 & 7 \\ 4 & \texttt{NA} & 1 & 2 \\ 5 & 1 & \texttt{NA} & 3 \\ 6 & 2 & 3 & \texttt{NA} \end{pmatrix}$$



* Al tratarse de un grafo no dirigido consideramos cada arco dos veces, en una y otra dirección.

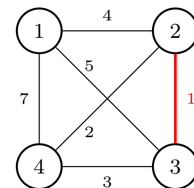
Inicio

1: $A^T \leftarrow \emptyset$
2: $M \leftarrow V = \{1,2,3,4\}$
3: **mientras** $|A^T| < (|V|-1)$ **hacer**



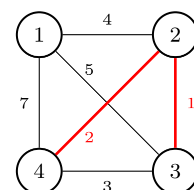
Etapa 1

4: seleccionar un arco $(i,j) \in A$ cuyo c_{ij} sea mínimo
$\quad$ $\text{mín}\{c_{12}, c_{13}, c_{14}, c_{23}, c_{24}, c_{34}\} = c_{23} = 1$
5: **si** $M_2 \neq M_3$ **entonces**
6: $\quad A^T \leftarrow A^T \cup \{(2,3)\} = \{(2,3)\}$
7: $\quad M_k \leftarrow M_i$ para todo $k \in \{1, ..., |M|\}$ tal que $M_k = M_j$
$\quad\quad M_3 \leftarrow M_2 = 2;\ M = \{1,2,2,4\}$
8: **fin si**
9: $A^T \leftarrow A^T \setminus \{(2,3)\} = \{(1,2),(1,3),(1,4),(2,4),(3,4)\}$



Etapa 2

4: seleccionar un arco $(i,j) \in A$ cuyo c_{ij} sea mínimo
$\quad$ $\text{mín}\{c_{12}, c_{13}, c_{14}, c_{24}, c_{34}\} = c_{24} = 2$
5: **si** $M_2 \neq M_4$ **entonces**
6: $\quad A^T \leftarrow A^T \cup \{(2,4)\} = \{(2,3),(2,4)\}$
7: $\quad M_k \leftarrow M_i$ para todo $k \in \{1, ..., |M|\}$ tal que $M_k = M_j$
$\quad\quad M_4 \leftarrow M_2 = 2;\ M = \{1,2,2,2\}$
8: **fin si**
9: $A^T \leftarrow A^T \setminus \{(2,4)\} = \{(1,2),(1,3),(1,4),(3,4)\}$

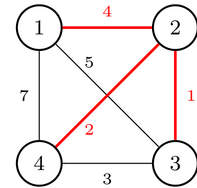


Etapa 3

4: seleccionar un arco $(i,j) \in A$ cuyo c_{ij} sea mínimo
$\quad$ $\text{mín}\{c_{12}, c_{13}, c_{14}, c_{34}\} = c_{34} = 3$
5: $M_3 = M_4$

9: $A^T \leftarrow A^T \setminus \{(3,4)\} = \{(1,2),(1,3),(1,4)\}$

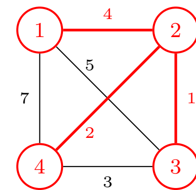
Etapa 4



4: seleccionar un arco $(i,j) \in A$ cuyo c_{ij} sea mínimo
$\min\{c_{12}, c_{13}, c_{14}\} = c_{12} = 4$
5: **si** $M_1 \neq M_2$ **entonces**
6: $A^T \leftarrow A^T \cup \{(1,2)\} = \{(2,3),(2,4),(1,2)\}$
7: $M_k \leftarrow M_i$ para todo $k \in \{1, ..., |M|\}$ tal que $M_k = M_j$
$M_2 \leftarrow M_1 = 1$; $M_3 \leftarrow M_1 = 1$; $M_4 \leftarrow M_1 = 1$; $M = \{1,1,1,1\}$
8: **fin si**
9: $A^T \leftarrow A^T \setminus \{(1,2)\} = \{(1,3),(1,4)\}$
10: **fin mientras** $|A^T| = (|V| - 1)$

11: $V^T \leftarrow V = \{1,2,3,4\}$

Salida



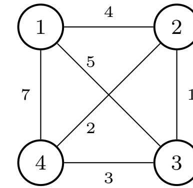
$$V^T = \{1,2,3,4\}$$
$$A^T = \{(1,2),(2,3),(2,4)\}$$

A.3. Algoritmo de Borůvka

Entrada

$$V = \{1,2,3,4\}$$
$$A = \{(1,2),(2,1),(1,3),(3,1),(1,4),(4,1),(2,3),(3,2),(2,4),(4,2),(3,4),(4,3)\} \ ^*$$

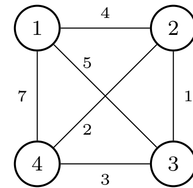
$$C = \begin{pmatrix} \mathtt{NA} & 4 & 5 & 7 \\ 4 & \mathtt{NA} & 1 & 2 \\ 5 & 1 & \mathtt{NA} & 3 \\ 6 & 2 & 3 & \mathtt{NA} \end{pmatrix}$$



* Al tratarse de un grafo no dirigido consideramos cada arco dos veces, en una y otra dirección.

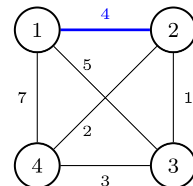
Inicio

1: $A^T \leftarrow \emptyset$
2: $M \leftarrow V = \{1,2,3,4\}$
3: **mientras** $|A^T| < (|V|-1)$ **hacer**
4: $S \leftarrow \emptyset$
5: **para** cada componente $k \in M$ **hacer**



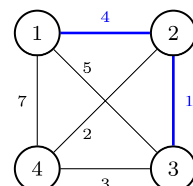
Etapa 1 - Componente 1

6: $S' \leftarrow \emptyset$
7: **para** cada nodo $i \in V$ tal que $M_i = k$ **hacer**
Nodo 1:
8: seleccionar arco $(i,j) \in A$ con $j \in V$ tal que $M_j \neq k$ cuyo c_{ij} sea mínimo
$min\{c_{12}, c_{13}, c_{14}\} = c_{12} = 4$
9: **si** existe el arco (i,j) **entonces**
10: $S' \leftarrow S' \cup \{(1,2)\} = \{(1,2)\}$
11: **fin si**
12: **fin para**
13: seleccionar arco $(i,j) \in S'$ cuyo c_{ij} sea mínimo
$min\{c_{12}\} = c_{12} = 4$
14: $S \leftarrow S \cup \{(1,2)\} = \{(1,2)\}$



Etapa 1 - Componente 2

6: $S' \leftarrow \emptyset$
7: **para** cada nodo $i \in V$ tal que $M_i = k$ **hacer**
Nodo 2:
8: seleccionar arco $(i,j) \in A$ con $j \in V$ tal que $M_j \neq k$ cuyo c_{ij} sea mínimo
$min\{c_{21}, c_{23}, c_{24}\} = c_{23} = 1$
9: **si** existe el arco (i,j) **entonces**
10: $S' \leftarrow S' \cup \{(2,3)\} = \{(2,3)\}$
11: **fin si**
12: **fin para**
13: seleccionar arco $(i,j) \in S'$ cuyo c_{ij} sea mínimo
$min\{c_{23}\} = c_{23} = 1$
14: $S \leftarrow S \cup \{(2,3)\} = \{(1,2),(2,3)\}$



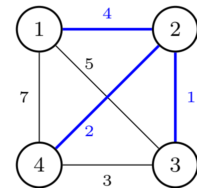
Etapa 1 - Componente 3

6: $S' \leftarrow \emptyset$
7: **para** cada nodo $i \in V$ tal que $M_i = k$ **hacer**
Nodo 3:
8: seleccionar arco $(i, j) \in A$ con $j \in V$ tal que $M_j \neq k$ cuyo c_{ij} sea mínimo
$min\{c_{31}, c_{32}, c_{34}\} = c_{32} = 1$
como $c_{32} = c_{23}$ se trata del arco $(2, 3)$ que ya está en el árbol

---

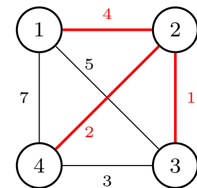
Etapa 1 - Componente 4

6: $S' \leftarrow \emptyset$
7: **para** cada nodo $i \in V$ tal que $M_i = k$ **hacer**
Nodo 4:
8: seleccionar arco $(i, j) \in A$ con $j \in V$ tal que $M_j \neq k$ cuyo c_{ij} sea mínimo
$min\{c_{41}, c_{42}, c_{43}\} = c_{42} = 2$
9: **si** existe el arco (i, j) **entonces**
19: $S' \leftarrow S' \cup \{(4, 2)\} = \{(4, 2)\}$
11: **fin si**
12: **fin para**
13: seleccionar arco $(i, j) \in S$ cuyo c_{ij} sea mínimo
$min\{c_{42}\} = c_{42} = 2$
14: $S \leftarrow S \cup \{(4, 2)\} = \{(1, 2), (2, 3), (4, 2)\}$
15: **fin para**

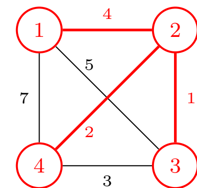


Etapa 1 - Fin

16: **para** cada arco $(i, j) \in S$ **hacer**
17: $A^T \leftarrow A^T \cup \{(i, j)\}$
$A^T = \{(1, 2), (2, 3), (4, 2)\}$
18: $M_l \leftarrow M_i$ para todo $l \in \{1, ..., |M|\}$ tal que $M_l = M_j$
$M_2 \leftarrow M_1 = 2;\ M = \{1, 2, 3, 4\}$
19: **fin para**
20: **fin mientras** $|A^T| = (|V| - 1)$

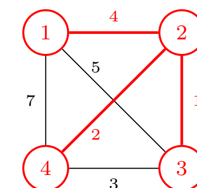


21: $V^T \leftarrow V = \{1, 2, 3, 4\}$



Salida

$V^T = \{1, 2, 3, 4\}$
$A^T = \{(1, 2), (2, 3), (2, 4)\}$

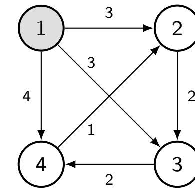


A.4. Algoritmo de Edmonds

Entrada

$$V = \{1,2,3,4\}$$
$$A = \{(1,2),(1,3),(1,4),(2,3),(3,4),(4,2)\}$$
$$s = 1$$

$$C = \begin{pmatrix} \texttt{NA} & 3 & 3 & 4 \\ \infty & \texttt{NA} & 2 & \infty \\ \infty & \infty & \texttt{NA} & 2 \\ \infty & 1 & \infty & \texttt{NA} \end{pmatrix}$$



Inicio

1: $A^{T'} \leftarrow \emptyset$

2: $k \leftarrow 1$; $V_k \leftarrow V$; $A_k \leftarrow A$; $C_k \leftarrow C$

3: **mientras** $A^{T'} = \emptyset$ **hacer**

Fase de búsqueda - Etapa 1

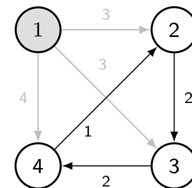
4: **para** cada nodo $j \in V \setminus \{s\}$ **hacer**

5: entre los arcos de A_k incidentes en j seleccionar el coste $c_{k_{ij}}$ mínimo

Nodo 2: $\min\{c_{12}, c_{42}\} = c_{42} = 1$

Nodo 3: $\min\{c_{13}, c_{23}\} = c_{23} = 2$

Nodo 4: $\min\{c_{14}, c_{34}\} = c_{34} = 2$



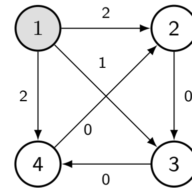
6: restar coste $c_{k_{ij}}$ a todos los arcos de A_k que inciden en j

Nodo 2: $c_{12} - c_{42} = 3 - 1 = 2$; $c_{42} - c_{42} = 1 - 1 = 0$

Nodo 3: $c_{13} - c_{23} = 3 - 2 = 1$; $c_{23} - c_{23} = 2 - 2 = 0$

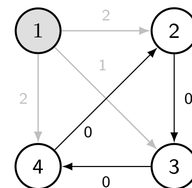
Nodo 4: $c_{14} - c_{34} = 4 - 2 = 2$; $c_{34} - c_{34} = 2 - 2 = 0$

7: **fin para**



8: seleccionar arcos de A_k con coste 0

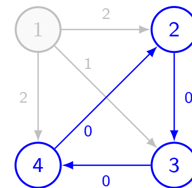
$\{(2,3),\ (3,4),\ (4,2)\}$



11: **si no** forman una arborescencia de expansión **entonces**

12: seleccionar arcos de A_k con coste 0 que formen un ciclo

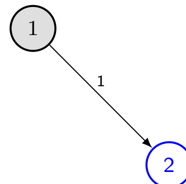
$\{(2,3),\ (3,4),\ (4,2)\}$



13: fusionar nodos y arcos del ciclo en un supernodo *

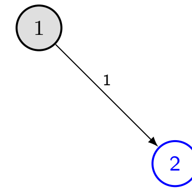
14: $k \leftarrow 1 + 1 = 2$; $V_2 \leftarrow V_1$; $A_2 \leftarrow A_1$; $C_2 \leftarrow C_1$

15: **fin si**

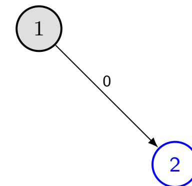


* Arcos y costes vienen dados por el coste mínimo entre los nodos del ciclo y cada uno de los nodos restantes.

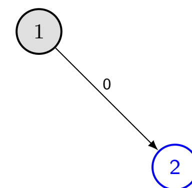
Fase de búsqueda - Etapa 2



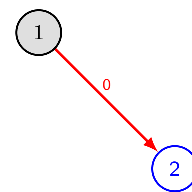
4: **para** cada nodo $j \in V \setminus \{s\}$ **hacer**
5: entre los arcos de A_k incidentes en j seleccionar el coste $c_{k_{ij}}$ mínimo
Nodo 2: $\text{mín}\{c_{12}\} = c_{12} = 1$



6: restar coste $c_{k_{ij}}$ a todos los arcos de A_k que inciden en j
Nodo 2: $c_{12} - c_{12} = 1 - 1 = 0$
7: **fin para**



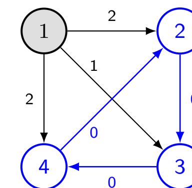
8: seleccionar arcos de A_k con coste 0
$\{(1, 2)\}$



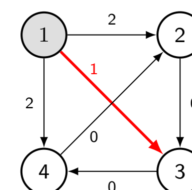
9: **si** forman una arborescencia de **expansión** entonces
10: guardar arborescencia en $A^{T'}$
15: **fin si**
16: **fin mientras** $A^{T'} \neq \emptyset$

17: **mientras** $k > 1$ **hacer**

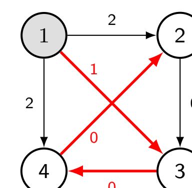
Fase de reconstrucción: Etapa 2



18: deshacer supernodo de la etapa k y recuperar grafo, costes y ciclo de la etapa $k - 1$

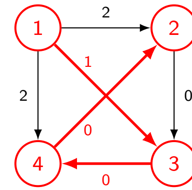


19: seleccionar arcos del grafo en la etapa $k - 1$ que se correspondan con arcos en $A^{T'}$



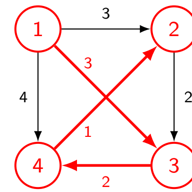
20: seleccionar arcos del ciclo en la etapa $k - 1$ excepto el que incide en un nodo ya alcanzado
21: guardar arcos seleccionados en una nueva arborescencia en $A^{T'}$
22 $k \leftarrow k - 1$
23: **fin mientras** $k = 1$

24: $V^T \leftarrow V = \{1, 2, 3, 4\}$



Salida

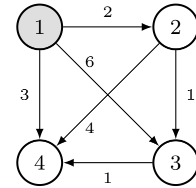
$$V^T = \{1, 2, 3, 4\}$$
$$A^T = \{(1, 2), (2, 3), (2, 4)\}$$



A.5. Algoritmo de Dijkstra

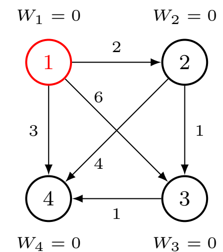
Entrada

$$\begin{aligned} V &= \{1,2,3,4\} \\ A &= \{(1,2),(1,3),(1,4), \\ &\quad (2,3),(2,4),(3,4)\} \\ s &= 1 \end{aligned} \qquad D = \begin{pmatrix} \texttt{NA} & 2 & 6 & 3 \\ \infty & \texttt{NA} & 1 & 4 \\ \infty & \infty & \texttt{NA} & 1 \\ \infty & \infty & \infty & \texttt{NA} \end{pmatrix}$$



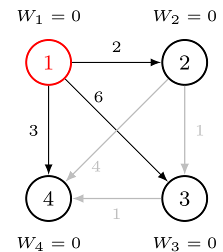
Inicio

1: $A^T \leftarrow \emptyset$
2: $V^T \leftarrow \{s\} = \{1\}$
3: $W_{1:|V|} \leftarrow 0 = \{0,0,0,0\}$
4: **mientras** $|V^T| < |V|$ **hacer**

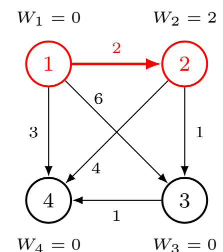


Etapa 1

5: seleccionar arcos $(i,j) \in A$ con $i \in V^T$ y $j \in V \setminus V^T$
$\{(1,2),(1,3),(1,4)\}$

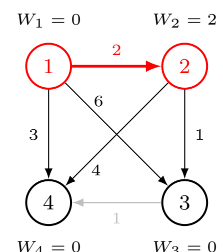


6: elegir un arco (i,j) de los anteriores tal que $d_{ij} + W_i$ sea mínimo
$\text{mín}\{d_{12} + W_1, d_{13} + W_1, d_{14} + W_1\} = d_{12} + W_1 = 2 + 0 = 2$
7: $A^T \leftarrow A^T \cup \{(i,j)\} = \{(1,2)\}$
8: $V^T \leftarrow V^T \cup \{j\} = \{1,2\}$
9: $W_j \leftarrow d_{ij} + W_i = \{0,2,0,0\}$

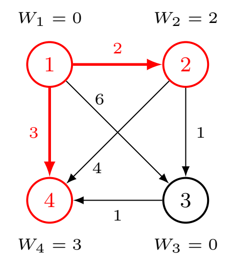


Etapa 2

5: seleccionar arcos $(i,j) \in A$ con $i \in V^T$ y $j \in V \setminus V^T$
$\{(1,3),(1,4),(2,3),(2,4)\}$

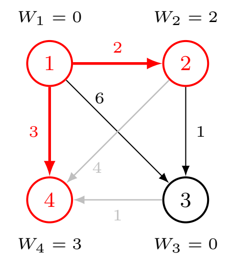


6: elegir un arco (i, j) de los anteriores tal que $d_{ij} + W_i$ sea mínimo
$\min\{d_{13} + W_1, d_{14} + W_1, d_{23} + W_2, d_{24} + W_2\} = d_{14} + W_1 = 3 + 0 = 3$
7: $A^T \leftarrow A^T \cup \{(i, j)\} = \{(1, 2), (1, 4)\}$
8: $V^T \leftarrow V^T \cup \{j\} = \{1, 2, 4\}$
9: $W_j \leftarrow d_{ij} + W_i = \{0, 2, 0, 3\}$

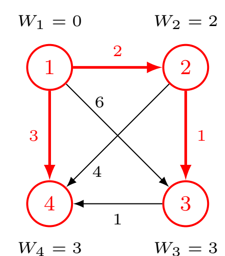


Etapa 3

5: seleccionar arcos $(i, j) \in A$ con $i \in V^T$ y $j \in V \setminus V^T$
$\{(1, 3), (2, 3)\}$

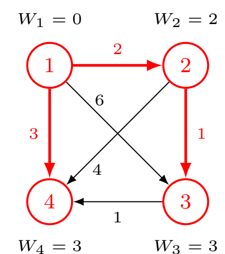


6: elegir un arco (i, j) de los anteriores tal que $d_{ij} + W_i$ sea mínimo
$\min\{d_{13} + W_1, d_{23} + W_2\} = d_{23} + W_2 = 1 + 2 = 3$
7: $A^T \leftarrow A^T \cup \{(i, j)\} = \{(1, 2), (1, 4), (2, 3)\}$
8: $V^T \leftarrow V^T \cup \{j\} = \{1, 2, 4, 3\}$
9: $W_j \leftarrow d_{ij} + W_i = \{0, 2, 3, 3\}$
10: **fin mientras** $|V^T| = |V|$



Salida

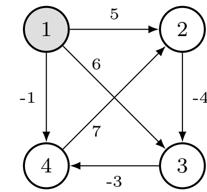
$V^T = \{1, 2, 3, 4\}$
$A^T = \{(1, 2), (2, 3), (2, 4)\}$
$W = \{0, 2, 3, 3\}$



A.6. Algoritmo de Bellman-Ford

$$V = \{1,2,3,4\}$$
$$A = \{(1,2),(1,3),(1,4),(2,3),(2,4),(3,4)\}$$
$$s = 1$$

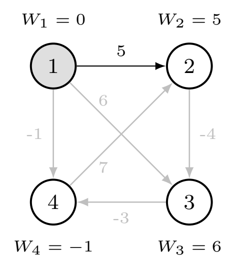
$$D = \begin{pmatrix} \texttt{NA} & 5 & 6 & -1 \\ \infty & \texttt{NA} & -4 & \infty \\ \infty & \infty & \texttt{NA} & -3 \\ \infty & 2 & \infty & \texttt{NA} \end{pmatrix}$$



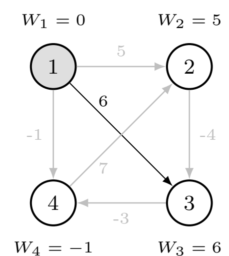
1: $A^T \leftarrow \emptyset$
2: $W_{1:|V|} \leftarrow \infty = \{\infty,\infty,\infty,\infty\}$; $W_s \leftarrow 0$; $W = \{0,\infty,\infty,\infty\}$
3: $P_{1:|V|} \leftarrow 0$
4: **para** cada nodo $i \in V \setminus \{s\}$ **hacer**

Etapa 1 - Nodo 1

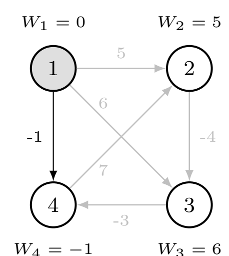
5: **para** cada arco $(i,j) \in A$ **hacer**
Arco $(1,2)$:
6: **si** $W_j > W_i + d_{ij}$ **entonces**
7: $W_2 \leftarrow W_1 + d_{12} = 0 + 5 = 5$
8: $P_2 \leftarrow 1 = \{0,1,0,0\}$
9: **fin si**



Arco $(1,3)$:
6: **si** $W_j > W_i + d_{ij}$ **entonces**
7: $W_3 \leftarrow W_1 + d_{13} = 0 + 6 = 6$
8: $P_3 \leftarrow 1 = \{0,1,1,0\}$
9: **fin si**

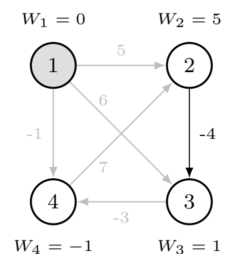


Arco $(1,4)$:
6: **si** $W_j > W_i + d_{ij}$ **entonces**
7: $W_4 \leftarrow W_1 + d_{14} = 0 + (-1) = -1$
8: $P_4 \leftarrow 1 = \{0,1,1,1\}$
9: **fin si**
10: **fin para**

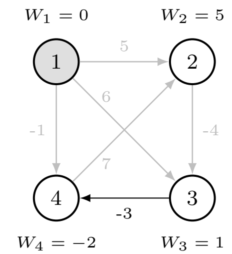


Etapa 1 - Nodo 2

5: **para** cada arco $(i,j) \in A$ **hacer**
Arco $(2,3)$:
6: **si** $W_j > W_i + d_{ij}$ **entonces**
7: $W_3 \leftarrow W_2 + d_{23} = 5 + (-4) = 1$
8: $P_3 \leftarrow 2 = \{0,1,2,1\}$
9: **fin si**
10: **fin para**

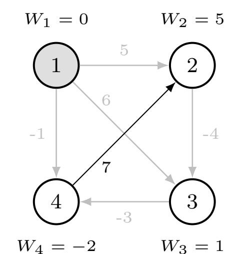


Etapa 1 - Nodo 3



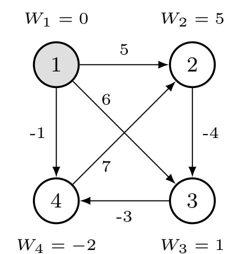
5: **para** cada arco $(i, j) \in A$ **hacer**
Arco $(3, 4)$**:**
6: **si** $W_j > W_i + d_{ij}$ **entonces**
7: $W_4 \leftarrow W_3 + d_{34} = 1 + (-3) = -2$
8: $P_4 \leftarrow 3 = \{0, 1, 2, 3\}$
9: **fin si**
10: **fin para**

Etapa 1 - Nodo 4



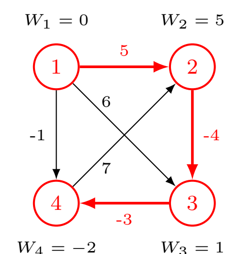
5: **para** cada arco $(i, j) \in A$ **hacer**
Arco $(4, 2)$**:**
6: **no** $W_j > W_i + d_{ij}$ **entonces**
9: **fin si**
10: **fin para**

Comprobación de ciclos negativos



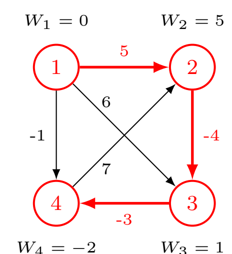
12: **para** cada arco $(i, j) \in A$ **hacer**
13: **si** $W_j > W_i + d_{ij}$ **entonces**
14: no hay solución
15: **fin si**
16: **fin para**

Fin - Construir árbol



17: **para** cada nodo $j \in V \setminus \{s\}$ **hacer**
18: $i \leftarrow P_j$
19: seleccionar arco (i, j) de A
20: $A^T \leftarrow A^T \cup \{(i, j)\}$
21: **fin para**
22: $V^T \leftarrow V$

Salida



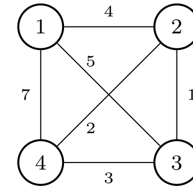
$$V^T = \{1, 2, 3, 4\}$$
$$A^T = \{(1, 2), (2, 3), (3, 4)\}$$
$$W = \{0, 5, 1, -2\}$$

A.7. Algoritmo de Gusfield

Entrada

$$V = \{1,2,3,4\}$$
$$A = \{(1,2),(2,1),(1,3),(3,1),(1,4),(4,1),(2,3),(3,2),(2,4),(4,2),(3,4),(4,3)\}\ ^*$$

$$Z = \begin{pmatrix} \texttt{NA} & 4 & 5 & 7 \\ 4 & \texttt{NA} & 1 & 2 \\ 5 & 1 & \texttt{NA} & 3 \\ 6 & 2 & 3 & \texttt{NA} \end{pmatrix}$$



* Al tratarse de un grafo no dirigido consideramos cada arco dos veces, en una y otra dirección.

Inicio

1: $V^T \leftarrow \{1\}$
2: $A^T \leftarrow \emptyset$
3: $Z^T_{|V|\times|V|} \leftarrow \infty$
4: **para** cada nodo $i \in V \setminus \{1\}$ **hacer**

Nodo 2

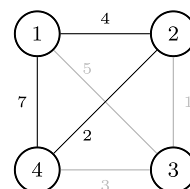
5: $V_k^T \leftarrow V^T = \{1\}$; $A_k^T \leftarrow A^T = \emptyset$

19: añadir arco (i,k), con $k \in V_k^T$, al árbol A^T
$A^T \leftarrow A^T \cup (1,2) = \{(1,2)\}$
20: obtener corte mínimo entre i y k en el grafo $G = (V,A)$ original
21: $z_{ik}^T \leftarrow$ capacidad del corte mínimo $i - k$
$z_{21}^T \leftarrow 7$
22: $V^T \leftarrow V^T \cup \{2\} = \{1,2\}$

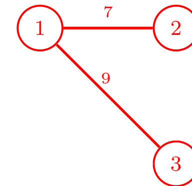
Nodo 3

5: $V_k^T \leftarrow V^T = \{1,2\}$; $A_k^T \leftarrow A^T = \{(1,2)\}$
6: **mientras** $|V_k^T| > 1$ **hacer**
7: borrar un arco $(i,j) \in A_k^T$ cuyo $z_{i,j}$ sea mínimo
$\min\{z_{12}\} = z_{12} = 7$
8: determinar componentes T_1 y T_2 de V_k^T y A_k^T tal que $i \in T_1$ y $j \in T_2$
$T_1 = \{1\}$; $T_2 = \{2\}$

9: obtener corte mínimo entre i y j en el grafo $G = (V,A)$ original
10: determinar componentes S_1 y S_2 de V^T y A^T tal que $i \in S_1$ y $j \in S_2$
$S_1 = \{1,3,4\}$; $S_2 = \{2\}$
11: **si** nodo $i \in S_1$ **entonces**
12: $V_k^T \leftarrow$ nodos de la componente T_1
$V_k^T \leftarrow \{1\}$
13: $A_k^T \leftarrow$ arcos de la componente T_1
$A_k^T \leftarrow \emptyset$
17: **fin si**
18: **fin mientras** $|V_k^T| = 1$

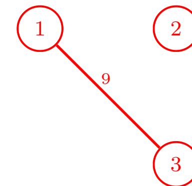


19: añadir arco (i, k), con $k \in V_k^T$, al árbol A^T
$A^T \leftarrow A^T \cup (1,3) = \{(1,2),(1,3)\}$
20: obtener corte mínimo entre i y k en el grafo $G = (V, A)$ original
21: $z_{ik}^T \leftarrow$ capacidad del corte mínimo $i - k$
$z_{31}^T \leftarrow 5 + 3 + 1 = 9$
22: $V^T \leftarrow V^T \cup \{3\} = \{1, 2, 3\}$

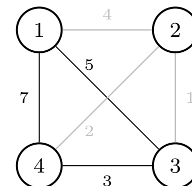


Nodo 4

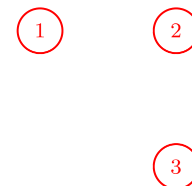
5: $V_k^T \leftarrow V^T = \{1,2,3\}$; $A_k^T \leftarrow A^T = \{(1,2),(1,3)\}$
6: **mientras** $|V_k^T| > 1$ **hacer**
7: borrar un arco $(i, j) \in A_k^T$ cuyo $z_{i,j}$ sea mínimo
$\text{mín}\{z_{12}, z_{13}\} = z_{12} = 7$
8: determinar componentes T_1 y T_2 de V_k^T y A_k^T tal que $i \in T_1$ y $j \in T_2$
$T_1 = \{1,3\}$; $T_2 = \{2\}$



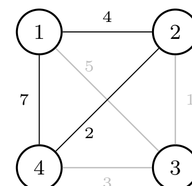
9: obtener corte mínimo entre i y j en el grafo $G = (V, A)$ original
10: determinar componentes S_1 y S_2 de V^T y A^T tal que $i \in S_1$ y $j \in S_2$
$S_1 = \{1,3,4\}$; $S_2 = \{2\}$
11: **si** nodo $i \in S_1$ **entonces**
12: $V_k^T \leftarrow$ nodos de la componente T_1
$V_k^T \leftarrow \{1,3\}$
13: $A_k^T \leftarrow$ arcos de la componente T_1
$A_k^T \leftarrow \{(1,3)\}$
17: **fin si**



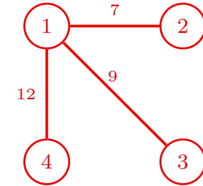
7: borrar un arco $(i, j) \in A_k^T$ cuyo $z_{i,j}$ sea mínimo
$\text{mín}\{z_{13}\} = z_{13} = 9$
8: determinar componentes T_1 y T_2 de V_k^T y A_k^T tal que $i \in T_1$ y $j \in T_2$
$T_1 = \{1\}$; $T_2 = \{3\}$



9: obtener corte mínimo entre i y j en el grafo $G = (V, A)$ original
10: determinar componentes S_1 y S_2 de V^T y A^T tal que $i \in S_1$ y $j \in S_2$
$S_1 = \{1,2,4\}$; $S_2 = \{3\}$
11: **si** nodo $i \in S_1$ **entonces**
12: $V_k^T \leftarrow$ nodos de la componente T_1
$V_k^T \leftarrow \{1\}$
13: $A_k^T \leftarrow$ arcos de la componente T_1
$A_k^T \leftarrow \emptyset$
17: **fin si**
18: **fin mientras** $|V_k^T| = 1$



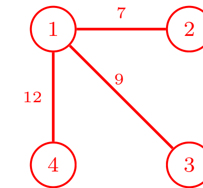
19: añadir arco (i, k), con $k \in V_k^T$, al árbol A^T
$A^T \leftarrow A^T \cup (1, 4) = \{(1, 2), (1, 3), (1, 4)\}$
20: obtener corte mínimo entre i y k en el grafo $G = (V, A)$ original
21: $z_{ik}^T \leftarrow$ capacidad del corte mínimo $i - k$
$z_{41}^T \leftarrow 7 + 2 + 3 = 12$
22: $V^T \leftarrow V^T \cup \{3\} = \{1, 2, 3, 4\}$
23: **fin para**



Salida

$$V^T = \{1, 2, 3, 4\}$$
$$A^T = \{(1, 2), (1, 3), (1, 4)\}$$

$$Z^T = \begin{pmatrix} \texttt{NA} & 7 & 9 & 12 \\ 7 & \texttt{NA} & \infty & \infty \\ 9 & \infty & \texttt{NA} & \infty \\ 12 & \infty & \infty & \texttt{NA} \end{pmatrix}$$



Anexo B

Manuales Paquetes

Package 'optrees'

September 3, 2014

Type Package

Title Optimal Trees in Weighted Graphs

Version 1.0

Date 2014-09-01

Author Manuel Fontenla [aut, cre]

Maintainer Manuel Fontenla <manu.fontenla@gmail.com>

Depends R (>= 2.7.0),igraph (>= 0.7.1)

Description Finds optimal trees in weighted graphs. In particular, this package provides solving tools for minimum cost spanning tree problems, minimum cost arborescence problems, shortest path tree problems and minimum cut tree problems.

License GPL-3

R topics documented:

optrees-package 2
ArcList2Cmat 3
checkArbor 3
checkGraph 4
Cmat2ArcList 5
compactCycle 5
findMinCut 6
findstCut 7
getCheapArcs 8
getComponents 9
getMinCostArcs 9
getMinimumArborescence 10
getMinimumCutTree 12
getMinimumSpanningTree 13
getShortestPathTree 15
getZeroArcs 17

ghTreeGusfield . . . 18
maxFlowFordFulkerson . . . 19
msArborEdmonds . . . 20
msTreeBoruvka . . . 21
msTreeKruskal . . . 22
msTreePrim . . . 23
removeLoops . . . 24
removeMultiArcs . . . 24
repGraph . . . 25
searchFlowPath . . . 26
searchWalk . . . 26
searchZeroCycle . . . 27
spTreeBellmanFord . . . 28
spTreeDijkstra . . . 29
stepbackArbor . . . 30

Index **31**

`optrees-package` *Optimal Trees in Weighted Graphs*

Description

Finds optimal trees in weighted graphs. In particular, this package provides solving tools for minimum cost spanning tree problems, minimum cost arborescence problems, shortest path tree problems and minimum cut tree problems.

Details

Package:	optrees
Type:	Package
Version:	1.0
Date:	2014-09-01
License:	GPL-3

The most important functions are getMinimumSpanningTree, getMinimumArborescence, getShortestPathTree and getMinimumCutTree. The other functions included in the package are auxiliary functions that can be used independently.

Author(s)

Manuel Fontenla <manu.fontenla@gmail.com>

`ArcList2Cmat` *Builds the cost matrix of a graph from its list of arcs*

Description

The `ArcList2Cmat` function constructs the cost matrix of a graph from a list that contains the arcs and its associated weights.

Usage

```
ArcList2Cmat(nodes, arcs, directed = TRUE)
```

Arguments

`nodes`	vector containing the nodes of the graph, identified by a number that goes from 1 to the order of the graph.
`arcs`	matrix with the list of arcs of the graph. Each row represents one arc. The first two columns contain the two endpoints of each arc and the third column contains their weights.
`directed`	logical value indicating whether the graph is directed (`TRUE`) or not (`FALSE`).

Value

`ArcList2Cmat` returns a $n \times n$ matrix that contains the weights of the arcs. It means that the element (i, j) of the matrix returns the weight of the arc (i, j). If the value of an arc (i, j) is `NA` or `Inf`, then it means this arc does not exist in the graph.

`checkArbor` *Checks if there is at least one arborescence in the graph*

Description

Given a directed graph, `checkArbor` searchs for an arborescence from the list of arcs. An arborescence is a directed graph with a source node and such that there is a unique path from the source to any other node.

Usage

```
checkArbor(nodes, arcs, source.node = 1)
```

Arguments

`nodes`	vector containing the nodes of the graph, identified by a number that goes from 1 to the order of the graph.
`arcs`	matrix with the list of arcs of the graph. Each row represents one arc. The first two columns contain the two endpoints of each arc and the third column contains their weights.
`source.node`	source node of the graph. Its default value is 1.

Value

If `checkArbor` found an arborescence it returns `TRUE`, otherwise it returns `FALSE`. If there is an arborescence the function also returns the list of arcs of the arborescence.

See Also

This function is an auxiliar function used in msArborEdmonds and getMinimumArborescence.

`checkGraph` *Checks if the graph contains at least one tree or one arborescence*

Description

The `checkGraph` function checks if it is possible to find at least one tree (or arborescence, if it is the case) in the graph. It only happens when the graph is connected and it is posible to find a walk from the source to any other node.

Usage

```
checkGraph(nodes, arcs, source.node = 1, directed = TRUE)
```

Arguments

`nodes`	vector containing the nodes of the graph, identified by a number that goes from 1 to the order of the graph.
`arcs`	matrix with the list of arcs of the graph. Each row represents one arc. The first two columns contain the two endpoints of each arc and the third column contains their weights.
`source.node`	number pointing the source node of the graph.
`directed`	logical value indicating whether the graph is directed (`TRUE`) or not (`FALSE`).

Value

`checkGraph` returns the value `TRUE` if the graph meets the requirements and `FALSE` otherwise. If the graph is not acceptable this functions also prints the reason.

Examples

```
# Graph
nodes <- 1:4
arcs <- matrix(c(1,2,2, 1,3,15, 2,3,1, 2,4,9, 3,4,1),
               byrow = TRUE, ncol = 3)
# Check graph
checkGraph(nodes, arcs)
```

`Cmat2ArcList` *Builds the list of arcs of a graph from its cost matrix*

Description

The `Cmat2ArcList` function builds the list of arcs of a graph from a cost matrix that contains the weights of all the arcs.

Usage

```
Cmat2ArcList(nodes, Cmat, directed = TRUE)
```

Arguments

`nodes`	vector containing the nodes of the graph, identified by a number that goes from 1 to the order of the graph.
`Cmat`	$n \times n$ matrix that contains the weights or costs of the arcs. Row i and column j represents the endpoints of an arc, and the value of the index ij is its weight or cost. If this value is `NA` or `Inf` means that there is no arc ij.
`directed`	logical value indicating whether the graph is directed (`TRUE`) or not (`FALSE`).

Value

`Cmat2ArcList` returns a matrix with the list of arcs of the graph. Each row represents one arc. The first two columns contain the two endpoints of each arc and the third column contains their weights.

`compactCycle` *Compacts the nodes in a cycle into a single node*

Description

Given a directed graph with a cycle, `compactCycle` compacts all the nodes in the cycle to a single node called supernode. The function uses the first and the last node of the cycle as a fusion point and obtains the costs of the incoming and outgoing arcs of the new node.

Usage

```
compactCycle(nodes, arcs, cycle)
```

Arguments

`nodes`	vector containing the nodes of the graph, identified by a number that goes from 1 to the order of the graph.
`arcs`	matrix with the list of arcs of the graph. Each row represents one arc. The first two columns contain the two endpoints of each arc and the third column contains their weights.
`cycle`	vector with the original nodes in the cycle.

Value

`compactCycle` returns the nodes and the list of arcs forming a new graph with the compressed cycle within a supernode. Also returns a list of the correspondences between the nodes of the new graph and the nodes of the previous graph.

See Also

This function is an auxiliar function used in msArborEdmonds and getMinimumArborescence.

---

findMinCut — *Finds the minimum cut of a given graph*

Description

The `findMinCut` function can find the minimum cut of a given graph. For that, this function computes the maximum flow of the network and applies the max-flow min-cut theorem to determine the cut with minimum weight between the source and the sink nodes.

Usage

```
findMinCut(nodes, arcs, algorithm = "Ford-Fulkerson", source.node = 1,
  sink.node = nodes[length(nodes)], directed = FALSE)
```

Arguments

`nodes`	vector containing the nodes of the graph, identified by a number that goes from 1 to the order of the graph.
`arcs`	matrix with the list of arcs of the graph. Each row represents one arc. The first two columns contain the two endpoints of each arc and the third column contains their weights.
`algorithm`	denotes the algorithm used to compute the maximum flow: "Ford-Fulkerson".
`source.node`	number pointing to the source node of the graph. It's node 1 by default.
`sink.node`	number pointing to the sink node of the graph. It's the last node by default.
`directed`	logical value indicating whether the graph is directed (`TRUE`) or not (`FALSE`).

Details

The max-flow min-cut theorem proves that, in a flow network, the maximum flow between the source node and the sink node and the weight of any minimum cut between them is equal.

Value

`findMinCut` returns a list with:

- `s.cut` vector with the nodes of the `s` cut.
- `t.cut` vector with the nodes of the `t` cut.
- `max.flow` value with the maximum flow in the flow network.
- `cut.set` list of arcs of the cut set founded.

See Also

This function is an auxiliar function used in ghTreeGusfield and getMinimumCutTree.

Examples

```
# Graph
nodes <- 1:6
arcs <- matrix(c(1,2,1, 1,3,7, 2,3,1, 2,4,3, 2,5,2, 3,5,4, 4,5,1, 4,6,6,
                 5,6,2), byrow = TRUE, ncol = 3)
# Find minimum cut
findMinCut(nodes, arcs, source.node = 2, sink.node = 6)
```

`findstCut` *Determines the s-t cut of a graph*

Description

`findstCut` reviews a given graph with a cut between two nodes with the bread-first search strategy and determines the two cut set of the partition. The cut is marked in the arc list with an extra column that indicates the remaining capacity of each arc.

Usage

```
findstCut(nodes, arcs, s = 1, t = nodes[length(nodes)])
```

Arguments

- `nodes` vector containing the nodes of the graph, identified by a number that goes from 1 to the order of the graph.
- `arcs` matrix with the list of arcs of the graph. Each row represents one arc. The first two columns contain the two endpoints of each arc and the third column contains their weights.
- `s` number pointing one node of the `s` cut in a given graph. It's node 1 by default.
- `t` number pointing one node of the `t` cut in a given graph. It's the last node by default.

Value

`findstCut` returns a list with two elements:

`s.cut`	vector with the nodes of the `s` cut.
`t.cut`	vector with the nodes of the `t` cut.

See Also

This function is an auxiliar function used in ghTreeGusfield and getMinimumCutTree.

`getCheapArcs` *Substracts the minimum weight of the arcs pointing to each node*

Description

The `getCheapArcs` function substracts to each arc of a given graph the value of the minimum weight of the arcs pointing to the same node.

Usage

```
getCheapArcs(nodes, arcs)
```

Arguments

`nodes`	vector containing the nodes of the graph, identified by a number that goes from 1 to the order of the graph.
`arcs`	matrix with the list of arcs of the graph. Each row represents one arc. The first two columns contain the two endpoints of each arc and the third column contains their weights.

Value

`getCheapArcs` returns a matrix with a new list of arcs.

See Also

This function is an auxiliar function used in msArborEdmonds and getMinimumArborescence.

getComponents *Connected components of a graph*

Description

The getComponents function returns all the connected components of a graph.

Usage

```
getComponents(nodes, arcs)
```

Arguments

nodes	vector containing the nodes of the graph, identified by a number that goes from 1 to the order of the graph.
arcs	matrix with the list of arcs of the graph. Each row represents one arc. The first two columns contain the two endpoints of each arc and the third column contains their weights.

Value

getComponents returns a list with all the components and the nodes of each one ($components) and a matrix with all the arcs of the graph and its component ($components.arcs).

Examples

```
# Graph
nodes <- 1:4
arcs <- matrix(c(1,2,1, 1,6,1, 3,4,1, 4,5,1), ncol = 3, byrow = TRUE)
# Components
getComponents(nodes, arcs)
```

getMinCostArcs *Selects the minimum cost of the arcs pointing to each node*

Description

Given a directed graph, getMinCostArcs selects the minimum cost arcs entering each node and removes the others.

Usage

```
getMinCostArcs(nodes, arcs)
```

Arguments

`nodes`	vector containing the nodes of the graph, identified by a number that goes from 1 to the order of the graph.
`arcs`	matrix with the list of arcs of the graph. Each row represents one arc. The first two columns contain the two endpoints of each arc and the third column contains their weights.

Value

The `getMinCostArcs` function returns a matrix with the list of the minimum cost arcs pointing to each node of the graph.

See Also

This function is an auxiliar function used in msArborEdmonds and getMinimumArborescence.

`getMinimumArborescence` *Computes a minimum cost arborescence*

Description

Given a connected weighted directed graph, `getMinimumArborescence` computes a minimum cost arborescence. This function provides a method to find the minimum cost arborescence with Edmonds' algorithm.

Usage

```
getMinimumArborescence(nodes, arcs, source.node = 1, algorithm = "Edmonds",
  stages.data = FALSE, show.data = TRUE, show.graph = TRUE,
  check.graph = FALSE)
```

Arguments

`nodes`	vector containing the nodes of the graph, identified by a number that goes from 1 to the order of the graph.
`arcs`	matrix with the list of arcs of the graph. Each row represents one arc. The first two columns contain the two endpoints of each arc and the third column contains their weights.
`source.node`	number pointing to the source node of the graph. It's node 1 by default.
`algorithm`	denotes the algorithm used to find a minimum cost arborescence: "Edmonds".
`check.graph`	logical value indicating if it is necesary to check the graph. Is `FALSE` by default.
`show.data`	logical value indicating if the function displays the console output (`TRUE`) or not (`FALSE`). The default is `TRUE`.

show.graph	logical value indicating if the function displays a graphical representation of the graph and its minimum arborescence (TRUE) or not (FALSE). The default is TRUE.
stages.data	logical value indicating if the function returns data of each stage. The default is FALSE.

Details

Given a connected weighted directed graph, a minimum cost arborescence is an arborescence such that the sum of the weight of its arcs is minimum. In some cases, it is possible to find several minimum cost arborescences, but the proposed algorithm only finds one of them.

Edmonds' algorithm was developed by the mathematician and computer scientist Jack R. Edmonds in 1967. Although, it was previously proposed in 1965 by Yoeng-jin Chu and Tseng-hong Liu. This algorithm decreases the weights of the arcs in a graph and compacts cycles of zero weight until it can find an arborescence. This arborescence has to be a minimum cost arborescence of the graph.

Value

getMinimumArborescence returns a list with:

tree.nodes	vector containing the nodes of the minimum cost arborescence.
tree.arcs	matrix containing the list of arcs of the minimum cost arborescence.
weight	value with the sum of weights of the arcs.
stages	number of stages required.
time	time needed to find the minimum cost arborescence.

This function also represents the graph and the minimum arborescence and prints to the console the results with additional information (number of stages, computational time, etc.).

References

Chu, Y. J., and Liu, T. H., "On the Shortest Arborescence of a Directed Graph", Science Sinica, vol. 14, 1965, pp. 1396-1400.

Edmonds, J., "Optimum Branchings", Journal of Research of the National Bureau of Standards, vol. 71B, No. 4, October-December 1967, pp. 233-240.

Examples

```
# Graph
nodes <- 1:4
arcs <- matrix(c(1,2,2, 1,3,3, 1,4,4, 2,3,3, 2,4,4, 3,2,3,
                 3,4,1, 4,2,1, 4,3,2),byrow = TRUE, ncol = 3)
# Minimum cost arborescence
getMinimumArborescence(nodes, arcs)
```

getMinimumCutTree *getMinimumCutTree* ———————————————- *Computes a minimum cut tree*

Description

Given a connected weighted undirected graph, `getMinimumCutTree` computes a minimum cut tree, also called Gomory-Hu tree. This function uses the Gusfield's algorithm to find it.

Usage

```
getMinimumCutTree(nodes, arcs, algorithm = "Gusfield", show.data = TRUE,
  show.graph = TRUE, check.graph = FALSE)
```

Arguments

`nodes`	vector containing the nodes of the graph, identified by a number that goes from 1 to the order of the graph.
`arcs`	matrix with the list of arcs of the graph. Each row represents one arc. The first two columns contain the two endpoints of each arc and the third column contains their weights.
`algorithm`	denotes the algorithm to use for find a minimum cut tree or Gomory-Hu tree: "Gusfield".
`check.graph`	logical value indicating if it is necesary to check the graph. Is `FALSE` by default.
`show.data`	logical value indicating if the function displays the console output (`TRUE`) or not (`FALSE`). The default is `TRUE`.
`show.graph`	logical value indicating if the function displays a graphical representation of the graph and its minimum cut tree (`TRUE`) or not (`FALSE`). The default is `TRUE`.

Details

The minimum cut tree or Gomory-Hu tree was introduced by R. E. Gomory and T. C. Hu in 1961. Given a connected weighted undirected graph, the Gomory-Hu tree is a weighted tree that contains the minimum s-t cuts for all s-t pairs of nodes in the graph. Gomory and Hu developed an algorithm to find this tree, but it involves maximum flow searchs and nodes contractions.

In 1990, Dan Gusfield proposed a new algorithm that can be used to find the Gomory-Hu tree without any nodes contraction and simplifies the implementation.

Value

`getMinimumCutTree` returns a list with:

`tree.nodes`	vector containing the nodes of the minimum cut tree.
`tree.arcs`	matrix containing the list of arcs of the minimum cut tree.
`weight`	value with the sum of weights of the arcs.

`stages`	number of stages required.
`time`	time needed to find the minimum cut tree.

This function also represents the graph and the minimum cut tree and prints in console the results whit additional information (number of stages, computational time, etc.).

References

R. E. Gomory, T. C. Hu. Multi-terminal network flows. Journal of the Society for Industrial and Applied Mathematics, vol. 9, 1961.

Dan Gusfield (1990). "Very Simple Methods for All Pairs Network Flow Analysis". SIAM J. Comput. 19 (1): 143-155.

Examples

```
# Graph
nodes <- 1:6
arcs <- matrix(c(1,2,1, 1,3,7, 2,3,1, 2,4,3, 2,5,2, 3,5,4, 4,5,1, 4,6,6,
               5,6,2), byrow = TRUE, ncol = 3)
# Minimum cut tree
getMinimumCutTree(nodes, arcs)
```

`getMinimumSpanningTree` *Computes a minimum cost spanning tree*

Description

Given a connected weighted undirected graph, `getMinimumSpanningTree` computes a minimum cost spanning tree. This function provides methods to find a minimum cost spanning tree with the three most commonly used algorithms: "Prim", "Kruskal" and "Boruvka".

Usage

```
getMinimumSpanningTree(nodes, arcs, algorithm, start.node = 1,
  show.data = TRUE, show.graph = TRUE, check.graph = FALSE)
```

Arguments

`nodes`	vector containing the nodes of the graph, identified by a number that goes from 1 to the order of the graph.
`arcs`	matrix with the list of arcs of the graph. Each row represents one arc. The first two columns contain the two endpoints of each arc and the third column contains their weights.
`algorithm`	denotes the algorithm used to find a minimum spanning tree: "Prim", "Kruskal" or "Boruvka".
`check.graph`	logical value indicating if it is necesary to check the graph. Is `FALSE` by default.

`start.node`	number which indicates the first node in Prim's algorithm. If none is specified node 1 is used by default.
`show.data`	logical value indicating if the function displays the console output (`TRUE`) or not (`FALSE`). The default is `TRUE`.
`show.graph`	logical value indicating if the function displays a graphical representation of the graph and its minimum spanning tree (`TRUE`) or not (`FALSE`). The default is `TRUE`.

Details

Given a connected weighted undirected graph, a minimum spanning tree is a spanning tree such that the sum of the weights of the arcs is minimum. There may be several minimum spanning trees of the same weight in a graph. Several algorithms were proposed to find a minimum spanning tree in a graph.

Prim's algorithm was developed in 1930 by the mathematician Vojtech Jarnik, independently proposed by the computer scientist Robert C. Prim in 1957 and rediscovered by Edsger Dijkstra in 1959. This is a greedy algorithm that can find a minimum spanning tree in a connected weighted undirected graph by adding minimum cost arcs leaving visited nodes recursively.

Kruskal's algorithm was published for first time in 1956 by mathematician Joseph Kruskal. This is a greedy algorithm that finds a minimum cost spanning tree in a connected weighted undirected graph by adding, without form cycles, the minimum weight arc of the graph in each iteration.

Boruvka's algorithm was published for first time in 1926 by mathematician Otakar Boruvka. This algorithm go through a connected weighted undirected graph, reviewing each component and adding the minimum weight arcs without repeat it until one minimum spanning tree is complete.

Value

`getMinimumSpanningTree` returns a list with:

`tree.nodes`	vector containing the nodes of the minimum cost spanning tree.
`tree.arcs`	matrix containing the list of arcs of the minimum cost spanning tree.
`weight`	value with the sum of weights of the arcs.
`stages`	number of stages required.
`stages.arcs`	stages in which each arc was added.
`time`	time needed to find the minimum cost spanning tree.

This function also represents the graph and the minimum spanning tree and prints to the console the results whit additional information (number of stages, computational time, etc.).

References

Prim, R. C. (1957), "Shortest Connection Networks And Some Generalizations", Bell System Technical Journal, 36 (1957), pp. 1389-1401.

Kruskal, Joshep B. (1956), "On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem", Proceedings of the American Mathematical Society, Vol. 7, No. 1 (Feb., 1956), pp. 48-50.

Boruvka, Otakar (1926). "O jistem problemu minimalnim (About a certain minimal problem)". Prace mor. prirodoved. spol. v Brne III (in Czech, German summary) 3: 37-58.

Examples

```
# Graph
nodes <- 1:4
arcs <- matrix(c(1,2,2, 1,3,15, 1,4,3, 2,3,1, 2,4,9, 3,4,1),
               ncol = 3, byrow = TRUE)
# Minimum cost spanning tree with several algorithms
getMinimumSpanningTree(nodes, arcs, algorithm = "Prim")
getMinimumSpanningTree(nodes, arcs, algorithm = "Kruskal")
getMinimumSpanningTree(nodes, arcs, algorithm = "Boruvka")
```

`getShortestPathTree` *Computes a shortest path tree*

Description

Given a connected weighted graph, directed or not, `getShortestPathTree` computes the shortest path tree from a given source node to the rest of the nodes the graph, forming a shortest path tree. This function provides methods to find it with two known algorithms: "Dijkstra" and "Bellman-Ford".

Usage

```
getShortestPathTree(nodes, arcs, algorithm, check.graph = FALSE,
  source.node = 1, directed = TRUE, show.data = TRUE, show.graph = TRUE,
  show.distances = TRUE)
```

Arguments

`nodes`	vector containing the nodes of the graph, identified by a number that goes from 1 to the order of the graph.
`arcs`	matrix with the list of arcs of the graph. Each row represents one arc. The first two columns contain the two endpoints of each arc and the third column contains their weights.
`algorithm`	denotes the algorithm used to find a shortest path tree: "Dijkstra" or "Bellman-Ford".
`check.graph`	logical value indicating if it is necesary to check the graph. Is `FALSE` by default.
`source.node`	number indicating the source node of the graph. It's node 1 by default.
`directed`	logical value indicating wheter the graph is directed (`TRUE`) or not (`FALSE`).
`show.data`	logical value indicating if the function displays the console output (`TRUE`) or not (`FALSE`). The default is `TRUE`.
`show.graph`	logical value indicating if the function displays a graphical representation of the graph and its shortest path tree (`TRUE`) or not (`FALSE`). The default is `TRUE`.
`show.distances`	logical value indicating if the function displays in the console output the distances from source to all the other nodes. The default is `TRUE`.

Details

Given a connected weighted graph, directed or not, a shortest path tree rooted at a source node is a spanning tree such that the path distance from the source to any other node is the shortest path distance between them. Differents algorithms were proposed to find a shortest path tree in a graph.

One of these algorithms is Dijkstra's algorithm. Developed by the computer scientist Edsger Dijkstra in 1956 and published in 1959, it is an algorithm that can compute a shortest path tree from a given source node to the others nodes in a connected, directed or not, graph with non-negative weights.

The Bellman-Ford algorithm gets its name for two of the developers, Richard Bellman y Lester Ford Jr., and it was published by them in 1958 and 1956 respectively. The same algorithm also was published independently in 1957 by Edward F. Moore. This algorithm can compute the shortest path from a source node to the rest of nodes in a connected, directed or not, graph with weights that can be negatives. If the graph is connected and there isn't negative cycles, the algorithm always finds a shortest path tree.

Value

`getShortestPathTree` returns a list with:

`tree.nodes`	vector containing the nodes of the shortest path tree.
`tree.arcs`	matrix containing the list of arcs of the shortest path tree.
`weight`	value with the sum of weights of the arcs.
`distances`	vector with distances from source to other nodes
`stages`	number of stages required.
`time`	time needed to find the shortest path tree.

This function also represents the graph with the shortest path tree and prints to the console the results with additional information (number of stages, computational time, etc.).

References

Dijkstra, E. W. (1959). "A note on two problems in connexion with graphs". Numerische Mathematik 1, 269-271.

Bellman, Richard (1958). "On a routing problem". Quarterly of Applied Mathematics 16, 87-90.

Ford Jr., Lester R. (1956). Network Flow Theory. Paper P-923. Santa Monica, California: RAND Corporation.

Moore, Edward F. (1959). "The shortest path through a maze". Proc. Internat. Sympos. Switching Theory 1957, Part II. Cambridge, Mass.: Harvard Univ. Press. pp. 285-292.

Examples

```
# Graph
nodes <- 1:5
arcs <- matrix(c(1,2,2, 1,3,2, 1,4,3, 2,5,5, 3,2,4, 3,5,3, 4,3,1, 4,5,0),
               ncol = 3, byrow = TRUE)
# Shortest path tree
```

```
getShortestPathTree(nodes, arcs, algorithm = "Dijkstra", directed=FALSE)
getShortestPathTree(nodes, arcs, algorithm = "Bellman-Ford", directed=FALSE)

# Graph with negative weights
nodes <- 1:5
arcs <- matrix(c(1,2,6, 1,3,7, 2,3,8, 2,4,5, 2,5,-4, 3,4,-3, 3,5,9, 4,2,-2,
                 5,1,2, 5,4,7), ncol = 3, byrow = TRUE)
# Shortest path tree
getShortestPathTree(nodes, arcs, algorithm = "Bellman-Ford", directed=TRUE)
```

`getZeroArcs` *Selects zero weight arcs of a graph*

Description

Given a directed graph, `getZeroArcs` returns the list of arcs with zero weight. Removes other arcs by assign them infinite value.

Usage

```
getZeroArcs(nodes, arcs)
```

Arguments

`nodes`	vector containing the nodes of the graph, identified by a number that goes from 1 to the order of the graph.
`arcs`	matrix with the list of arcs of the graph. Each row represents one arc. The first two columns contain the two endpoints of each arc and the third column contains their weights.

Value

The `getZeroArcs` function returns a matrix with the list of zero weight arcs of the graph.

See Also

This function is an auxiliar function used in msArborEdmonds and getMinimumArborescence.

`ghTreeGusfield` *Gomory-Hu tree with the Gusfield's algorithm*

Description

Given a connected weighted and undirected graph, the `ghTreeGusfield` function builds a Gomory-Hu tree with the Gusfield's algorithm.

Usage

```
ghTreeGusfield(nodes, arcs)
```

Arguments

`nodes`	vector containing the nodes of the graph, identified by a number that goes from 1 to the order of the graph.
`arcs`	matrix with the list of arcs of the graph. Each row represents one arc. The first two columns contain the two endpoints of each arc and the third column contains their weights.

Details

The Gomory-Hu tree was introduced by R. E. Gomory and T. C. Hu in 1961. Given a connected weighted and undirected graph, the Gomory-Hu tree is a weighted tree that contains the minimum s-t cuts for all s-t pairs of nodes in the graph. Gomory and Hu also developed an algorithm to find it that involves maximum flow searchs and nodes contractions.

In 1990, Dan Gusfield proposed a new algorithm that can be used to find a Gomory-Hu tree without nodes contractions and simplifies the implementation.

Value

`ghTreeGusfield` returns a list with:

`tree.nodes`	vector containing the nodes of the Gomory-Hu tree.
`tree.arcs`	matrix containing the list of arcs of the Gomory-Hu tree.
`stages`	number of stages required.

References

R. E. Gomory, T. C. Hu. Multi-terminal network flows. Journal of the Society for Industrial and Applied Mathematics, vol. 9, 1961.

Dan Gusfield (1990). "Very Simple Methods for All Pairs Network Flow Analysis". SIAM J. Comput. 19 (1): 143-155.

See Also

A more general function getMinimumCutTree.

`maxFlowFordFulkerson` *Maximum flow with the Ford-Fulkerson algorithm*

Description

The `maxFlowFordFulkerson` function computes the maximum flow in a given flow network with the Ford-Fulkerson algorithm.

Usage

```
maxFlowFordFulkerson(nodes, arcs, directed = FALSE, source.node = 1,
  sink.node = nodes[length(nodes)])
```

Arguments

`nodes`	vector containing the nodes of the graph, identified by a number that goes from 1 to the order of the graph.
`arcs`	matrix with the list of arcs of the graph. Each row represents one arc. The first two columns contain the two endpoints of each arc and the third column contains their weights.
`directed`	logical value indicating wheter the graph is directed (`TRUE`) or not (`FALSE`).
`source.node`	number pointing to the source node of the graph. It's node 1 by default.
`sink.node`	number pointing to the sink node of the graph. It's the last node by default.

Details

The Ford-Fulkerson algorithm was published in 1956 by L. R. Ford, Jr. and D. R. Fulkerson. This algorithm can compute the maximum flow between source and sink nodes of a flow network.

Value

`maxFlowFordFulkerson` returns a list with:

`s.cut`	vector with the nodes of the `s` cut.
`t.cut`	vector with the nodes of the `t` cut.
`max.flow`	value with the maximum flow in the flow network.

References

Ford, L. R.; Fulkerson, D. R. (1956). "Maximal flow through a network". Canadian Journal of Mathematics 8: 399.

See Also

This function is an auxiliar function used in ghTreeGusfield and getMinimumCutTree.

Examples

```
# Graph
nodes <- 1:6
arcs <- matrix(c(1,2,1, 1,3,7, 2,3,1, 2,4,3, 2,5,2, 3,5,4, 4,5,1, 4,6,6,
                5,6,2), byrow = TRUE, ncol = 3)
# Maximum flow with Ford-Fulkerson algorithm
maxFlowFordFulkerson(nodes, arcs, source.node = 2, sink.node = 6)
```

---

msArborEdmonds *Minimum cost arborescence with Edmonds' algorithm*

Description

Given a connected weighted and directed graph, msArborEdmonds uses Edmonds' algorithm to find a minimum cost arborescence.

Usage

```
msArborEdmonds(nodes, arcs, source.node = 1, stages.data = FALSE)
```

Arguments

nodes	vector containing the nodes of the graph, identified by a number that goes from 1 to the order of the graph.
arcs	matrix with the list of arcs of the graph. Each row represents one arc. The first two columns contain the two endpoints of each arc and the third column contains their weights.
source.node	source node of the graph. It's node 1 by default.
stages.data	logical value indicating if the function returns data of each stage. Is FALSE by default.

Details

Edmonds' algorithm was developed by the mathematician and computer scientist Jack R. Edmonds in 1967. Previously, it was proposed in 1965 by Yoeng-jin Chu and Tseng-hong Liu.

Value

msArborEdmonds returns a list with:

tree.nodes	vector containing the nodes of the minimum cost arborescence.
tree.arcs	matrix containing the list of arcs of the minimum cost arborescence.
stages	number of stages required.

References

Chu, Y. J., and Liu, T. H., "On the Shortest Arborescence of a Directed Graph", Science Sinica, vol. 14, 1965, pp. 1396-1400.

Edmonds, J., "Optimum Branchings", Journal of Research of the National Bureau of Standards, vol. 71B, No. 4, October-December 1967, pp. 233-240.

See Also

A more general function getMinimumSpanningTree.

---

msTreeBoruvka — *Minimum cost spanning tree with Boruvka's algorithm.*

Description

`msTreeBoruvka` computes a minimum cost spanning tree of an undirected graph with Boruvka's algorithm.

Usage

```
msTreeBoruvka(nodes, arcs)
```

Arguments

`nodes`	vector containing the nodes of the graph, identified by a number that goes from 1 to the order of the graph.
`arcs`	matrix with the list of arcs of the graph. Each row represents one arc. The first two columns contain the two endpoints of each arc and the third column contains their weights.

Details

Boruvka's algorithm was firstly published in 1926 by the mathematician Otakar Boruvka. This algorithm works in a connected, weighted and undirected graph, checking each component and adding the minimum weight arcs that connect the component to other components until one minimum spanning tree is complete.

Value

`msTreeBoruvka` returns a list with:

`tree.nodes`	vector containing the nodes of the minimum cost spanning tree.
`tree.arcs`	matrix containing the list of arcs of the minimum cost spanning tree.
`stages`	number of stages required.
`stages.arcs`	stages in which each arc was added.

References

Boruvka, Otakar (1926). "O jistem problemu minimalnim (About a certain minimal problem)". Prace mor. prirodoved. spol. v Brne III (in Czech, German summary) 3: 37-58.

See Also

A more general function getMinimumSpanningTree.

---

msTreeKruskal — *Minimum cost spanning tree with Kruskal's algorithm*

Description

`msTreeKruskal` computes a minimum cost spanning tree of an undirected graph with Kruskal's algorithm.

Usage

```
msTreeKruskal(nodes, arcs)
```

Arguments

`nodes`	vector containing the nodes of the graph, identified by a number that goes from 1 to the order of the graph.
`arcs`	matrix with the list of arcs of the graph. Each row represents one arc. The first two columns contain the two endpoints of each arc and the third column contains their weights.

Details

Kruskal's algorithm was published for first time in 1956 by mathematician Joseph Kruskal. This is a greedy algorithm that finds a minimum cost spanning tree in a connected weighted undirected graph by adding, without forming cycles, the minimum weight arc of the graph at each stage.

Value

`msTreeKruskal` returns a list with:

`tree.nodes`	vector containing the nodes of the minimum cost spanning tree.
`tree.arcs`	matrix containing the list of arcs of the minimum cost spanning tree.
`stages`	number of stages required.
`stages.arcs`	stages in which each arc was added.

References

Kruskal, Joshep B. (1956), "On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem", Proceedings of the American Mathematical Society, Vol. 7, No. 1 (Feb., 1956), pp. 48-50

See Also

A more general function getMinimumSpanningTree.

`msTreePrim` *Minimum cost spanning tree with Prim's algorithm*

Description

`msTreePrim` computes a minimum cost spanning tree of an undirected graph with Prim's algorithm.

Usage

```
msTreePrim(nodes, arcs, start.node = 1)
```

Arguments

`nodes`	vector containing the nodes of the graph, identified by a number that goes from 1 to the order of the graph.
`arcs`	matrix with the list of arcs of the graph. Each row represents one arc. The first two columns contain the two endpoints of each arc and the third column contains their weights.
`start.node`	number associated with the first node in Prim's algorithm. By default, node 1 is the first node.

Details

Prim's algorithm was developed in 1930 by the mathematician Vojtech Jarnik, later proposed by the computer scientist Robert C. Prim in 1957 and rediscovered by Edsger Dijkstra in 1959. This is a greedy algorithm that can find a minimum spanning tree in a connected, weighted and undirected graph by adding recursively minimum cost arcs leaving visited nodes.

Value

`msTreePrim` returns a list with:

`tree.nodes`	vector containing the nodes of the minimum cost spanning tree.
`tree.arcs`	matrix containing the list of arcs of the minimum cost spanning tree.
`stages`	number of stages required.
`stages.arcs`	stages in which each arc was added.

References

Prim, R. C. (1957), "Shortest Connection Networks And Some Generalizations", Bell System Technical Journal, 36 (1957), pp. 1389-1401

See Also

A more general function getMinimumSpanningTree.

`removeLoops` *Removes loops of a graph*

Description

This function reviews the arc list of a given graph and check if exists loops in it. A loop is an arc that connect a node with itself. If `removeLoops` find a loop remove it from the list of arcs.

Usage

```
removeLoops(arcs)
```

Arguments

`arcs`	matrix with the list of arcs of the graph. Each row represents one arc. The first two columns contain the two endpoints of each arc and the third column contains their weights.

Value

`removeLoops` returns a new list of arcs without any of the loops founded.

`removeMultiArcs` *Removes multi-arcs with no minimum cost*

Description

The `removeMultiArcs` function goes through the arcs list of a given graph and check if there are more than one arc between two nodes. If exist more than one, the function keeps one with minimum cost and remove the others.

Usage

```
removeMultiArcs(arcs, directed = TRUE)
```

Arguments

`arcs`	matrix with the list of arcs of the graph. Each row represents one arc. The first two columns contain the two endpoints of each arc and the third column contains their weights.
`directed`	logical value indicating whether the graph is directed (`TRUE`) or not (`FALSE`).

Value

`removeMultiArcs` returns a new list of arcs without any of the multi-arcs founded.

`repGraph` *Visual representation of a graph*

Description

The `repGraph` function uses `igraph` package to represent a graph.

Usage

```
repGraph(nodes, arcs, tree = NULL, directed = FALSE, plot.title = NULL,
  fix.seed = NULL)
```

Arguments

`nodes`	vector containing the nodes of the graph, identified by a number that goes from 1 to the order of the graph.
`arcs`	matrix with the list of arcs of the graph. Each row represents one arc. The first two columns contain the two endpoints of each arc and the third column contains their weights.
`tree`	matrix with the list of arcs of a tree, if there is one. Is `NULL` by default.
`directed`	logical value indicating whether the graph is directed (`TRUE`) or not (`FALSE`).
`plot.title`	string with main title of the graph. Is `NULL` by default.
`fix.seed`	number to set a seed for the representation.

Value

`repGraph` returns a plot with the given graph.

Examples

```
# Graph
nodes <- c(1:4)
arcs <- matrix(c(1,2,2, 1,3,15, 2,3,1, 2,4,9, 3,4,1),
               byrow = TRUE, ncol = 3)
# Plot graph
repGraph(nodes, arcs)
```

searchFlowPath *Finds a maximum flow path*

Description

`searchFlowPath` go through a given graph and obtains a maximum flow path between source and sink nodes. The function uses a deep-first search estrategy.

Usage

```
searchFlowPath(nodes, arcs, source.node = 1,
  sink.node = nodes[length(nodes)])
```

Arguments

`nodes`	vector containing the nodes of the graph, identified by a number that goes from 1 to the order of the graph.
`arcs`	matrix with the list of arcs of the graph. Each row represents one arc. The first two columns contain the two endpoints of each arc and the third column contains their weights.
`source.node`	number pointing to the source node of the graph. It's node 1 by default.
`sink.node`	number pointing to the sink node of the graph. It's the last node by default.

Value

`searchFlowPath` returns a list with two elements:

`path.nodes`	vector with nodes of the path.
`path.arcs`	matrix with the list of arcs that form the maximum flow path.

See Also

This function is an auxiliar function used in ghTreeGusfield and getMinimumCutTree.

searchWalk *Finds an open walk in a graph*

Description

This function walks a given graph, directed or not, searching for a walk from a starting node to a final node. The `searchWalk` function uses a deep-first search strategy to returns the first open walk found, regardless it has formed cycles or repeated nodes.

Usage

```
searchWalk(nodes, arcs, directed = TRUE, start.node = nodes[1],
  end.node = nodes[length(nodes)], method = NULL)
```

Arguments

`nodes`	vector containing the nodes of the graph, identified by a number that goes from 1 to the order of the graph.
`arcs`	matrix with the list of arcs of the graph. Each row represents one arc. The first two columns contain the two endpoints of each arc and the third column contains their weights.
`directed`	logical value indicating whether the graph is directed (`TRUE`) or not (`FALSE`).
`start.node`	number with one node from which a walk start.
`end.node`	number with final node of the walk.
`method`	character string specifying which method use to select the arcs that will form the open walk: `"min"` if the function chooses the minimum weight arcs, `"max"` if chooses the maximum weight arcs, or `NULL` if chooses the arcs by their order in the list of arcs.

Value

If `searchWalk` found an open walk in the graph returns `TRUE`, a vector with the nodes of the walk and a matrix with the list of arcs of it.

`searchZeroCycle` *Zero weight cycle in a graph*

Description

Given a directed graph, `searchZeroCycle` search paths in it that forms a zero weight cycle. The function finishes when found one cycle.

Usage

```
searchZeroCycle(nodes, arcs)
```

Arguments

`nodes`	vector containing the nodes of the graph, identified by a number that goes from 1 to the order of the graph.
`arcs`	matrix with the list of arcs of the graph. Each row represents one arc. The first two columns contain the two endpoints of each arc and the third column contains their weights.

Value

`searchZeroCycle` returns a vector with the nodes and a matrix with a list of arcs of the cycle found.

See Also

This function is an auxiliar function used in msArborEdmonds and getMinimumArborescence.

`spTreeBellmanFord` *Shortest path tree with Bellman-Ford algorithm*

Description

The `spTreeBellmanFord` function computes the shortest path tree of an undirected or directed graph with the Bellman-Ford algorithm.

Usage

```
spTreeBellmanFord(nodes, arcs, source.node = 1, directed = TRUE)
```

Arguments

`nodes`	vector containing the nodes of the graph, identified by a number that goes from 1 to the order of the graph.
`arcs`	matrix with the list of arcs of the graph. Each row represents one arc. The first two columns contain the two endpoints of each arc and the third column contains their weights.
`source.node`	number pointing the source node of the graph. It's node 1 by default.
`directed`	logical value indicating whether the graph is directed (`TRUE`) or not (`FALSE`).

Details

The Bellman-Ford algorithm gets its name for two of the developers, Richard Bellman y Lester Ford Jr., that published it in 1958 and 1956 respectively. The same algorithm also was published independently in 1957 by Edward F. Moore.

The Bellman-Ford algorithm can compute the shortest path from a source node to the rest of nodes that make a connected graph, directed or not, with weights that can be negatives. If the graph is connected and there isn't negative cycles, the algorithm always finds a shortest path tree.

Value

`spTreeBellmanFord` returns a list with:

`tree.nodes`	vector containing the nodes of the shortest path tree.
`tree.arcs`	matrix containing the list of arcs of the shortest path tree.
`stages`	number of stages required.
`distances`	vector with distances from source to other nodes

References

Bellman, Richard (1958). "On a routing problem". Quarterly of Applied Mathematics 16, 87-90.

Ford Jr., Lester R. (1956). Network Flow Theory. Paper P-923. Santa Monica, California: RAND Corporation.

Moore, Edward F. (1959). "The shortest path through a maze". Proc. Internat. Sympos. Switching Theory 1957, Part II. Cambridge, Mass.: Harvard Univ. Press. pp. 285-292.

See Also

A more general function getShortestPathTree.

---

spTreeDijkstra *Shortest path tree with Dijkstra's algorithm*

Description

The `spTreeDijkstra` function computes the shortest path tree of an undirected or directed graph with Dijkstra's algorithm.

Usage

```
spTreeDijkstra(nodes, arcs, source.node = 1, directed = TRUE)
```

Arguments

`nodes`	vector containing the nodes of the graph, identified by a number that goes from 1 to the order of the graph.
`arcs`	matrix with the list of arcs of the graph. Each row represents one arc. The first two columns contain the two endpoints of each arc and the third column contains their weights.
`source.node`	number pointing the source node of the graph. It's node 1 by default.
`directed`	logical value indicating whether the graph is directed (`TRUE`) or not (`FALSE`).

Details

Dijkstra's algorithm was developed by the computer scientist Edsger Dijkstra in 1956 and published in 1959. This is an algorithm that can computes a shortest path tree from a given source node to the others nodes that make a connected graph, directed or not, with non-negative weights.

Value

`spTreeDijkstra` returns a list with:

`tree.nodes`	vector containing the nodes of the shortest path tree.
`tree.arcs`	matrix containing the list of arcs of the shortest path tree.
`stages`	number of stages required.
`distances`	vector with distances from source to other nodes

References

Dijkstra, E. W. (1959). "A note on two problems in connexion with graphs". Numerische Mathematik 1, 269-271.

See Also

A more general function getShortestPathTree.

`stepbackArbor` *Goes back between two stages of the Edmond's algorithm*

Description

The `stepbackArbor` function rebuilds an arborescence present in earlier stage of Edmonds's algorithm to find a minimum cost arborescence.

Usage

```
stepbackArbor(before, after)
```

Arguments

`before`	list with elements of the previous stage
`after`	list with elements of the next stage

Value

An updated list of elements of the earlier stage with a new arborescence

See Also

This function is an auxiliar function used in msArborEdmonds and getMinimumArborescence.

Index

ArcList2Cmat, 3

checkArbor, 3
checkGraph, 4
Cmat2ArcList, 5
compactCycle, 5

findMinCut, 6
findstCut, 7

getCheapArcs, 8
getComponents, 9
getMinCostArcs, 9
getMinimumArborescence, *2*, *4*, *6*, *8*, *10*, 10, *17*, *28*, *30*
getMinimumCutTree, *2*, *7*, *8*, 12, *18*, *19*, *26*
getMinimumSpanningTree, *2*, 13, *21–23*
getShortestPathTree, *2*, 15, *29*, *30*
getZeroArcs, 17
ghTreeGusfield, *7*, *8*, 18, *19*, *26*

maxFlowFordFulkerson, 19
msArborEdmonds, *4*, *6*, *8*, *10*, *17*, 20, *28*, *30*
msTreeBoruvka, 21
msTreeKruskal, 22
msTreePrim, 23

optrees *(*optrees-package*)*, 2
optrees-package, 2

removeLoops, 24
removeMultiArcs, 24
repGraph, 25

searchFlowPath, 26
searchWalk, 26
searchZeroCycle, 27
spTreeBellmanFord, 28
spTreeDijkstra, 29
stepbackArbor, 30

Package ‘cooptrees’

September 3, 2014

Type Package

Version 1.0

Date 2014-09-01

Title Cooperative aspects of optimal trees in weighted graphs

Author Manuel Fontenla

Maintainer Manuel Fontenla <manu.fontenla@gmail.com>

Depends R (>= 2.7.0),igraph (>= 0.7.1),optrees (>= 1.0)

Imports gtools

Description Computes several cooperative games and allocation rules associated with minimum cost spanning tree problems and minimum cost arborescence problems.

License GPL-3

R topics documented:

cooptrees-package 2
maBird 3
maCooperative 4
maERO 4
maGames 5
maIrreducible 6
maPessimistic 7
maRules 7
mstBird 8
mstCooperative 9
mstDuttaKar 10
mstEROKruskal 11
mstGames 12
mstIrreducible 13
mstKar 13

mstOptimistic 14
mstPessimistic 15
mstRules 16
shapleyValue 17

Index **18**

`cooptrees-package` *Cooperative aspects of optimal trees in weighted graphs*

Description

Computes several cooperative games and allocation rules associated with minimum cost spanning tree problems and minimum cost arborescence problems.

Details

Package:	cooptrees
Type:	Package
Version:	1.0
Date:	2014-09-01
License:	GPL-3

The most important functions are mstCooperative and maCooperative. The other functions included in the package are auxiliary ones that can be used independently.

Author(s)

Manuel Fontenla

Maintainer: Manuel Fontenla <manu.fontenla@gmail.com>

References

B. Dutta and D. Mishra, "Minimum cost arborescences", Games and Economic Behavior, vol. 74, pp. 120-143, Jan. 2012.

C. G. Bird, "On Cost Allocation for a Spanning Tree: A Game Theoretic Approach", Networks, no. 6, pp. 335-350, 1976.

B. Dutta and A. Kar, "Cost monotonicity, consistency and minimum cost spanning tree games", Games and Economic Behavior, vol. 48, pp. 223-248, Aug. 2004.

A. Kar, "Axiomatization of the Shapley Value on Minimum Cost Spanning Tree Games", Games and Economic Behavior, vol. 38, pp. 265-277, Feb. 2002.

G. Bergantiños and J. J. Vidal-Puga, "A fair rule in minimum cost spanning tree problems", Journal of Economic Theory, vol. 137, pp. 326-352, Nov. 2007.

G. Bergantiños and J. J. Vidal-Puga, "The optimistic TU game in minimum cost spanning tree problems", International Journal of Game Theory, vol. 36, pp. 223-239, Feb. 2007.

V. Feltkamp, S. H. Tijs, S. Muto, "On the irreducible core and the equal remaining obligation rule of minimum cost extension problems", Mimeo, Tilburg University, 1994.

---

maBird *Bird rule for minimum cost arborescence problems*

Description

Given a graph with a minimum cost arborescence, the `maBird` function divides the cost of this arborescence among the agents. For that purpose it, uses the Bird rule, where each agent pays the cost of the last arc that connects him to the source.

Usage

```
maBird(nodes, arcs)
```

Arguments

`nodes`	vector containing the nodes of the graph, identified by a number that goes from 1 to the order of the graph.
`arcs`	matrix with the list of arcs of the graph. Each row represents one arc. The first two columns contain the two endpoints of each arc and the third column contains their weights.

Value

`maBird` returns a matrix with the agents and their costs.

See Also

The more general function maRules.

Examples

```
# Graph
nodes <- 1:4
arcs <- matrix(c(1,2,7, 1,3,6, 1,4,4, 2,3,8, 2,4,6, 3,2,6,
                 3,4,5, 4,2,5, 4,3,7), ncol = 3, byrow = TRUE)
# Bird
maBird(nodes, arcs)
```

maCooperative — *Cooperation in minimum cost arborescence problems*

Description

Given a graph with at least one minimum cost arborescence, the `maCooperative` function computes the cooperative and "Bird" and "ERO" rules.

Usage

```
maCooperative(nodes, arcs, show.data = TRUE)
```

Arguments

`nodes`	vector containing the nodes of the graph, identified by a number that goes from 1 to the order of the graph.
`arcs`	matrix with the list of arcs of the graph. Each row represents one arc. The first two columns contain the two endpoints of each arc and the third column contains their weights.
`show.data`	logical value indicating if the function displays the console output (`TRUE`) or not (`FALSE`). By default its value is `TRUE`.

Value

`maCooperative` returns and prints a list with the cooperative games and allocation rules of a minimum cost arborescence problem.

Examples

```
# Graph
nodes <- 1:4
arcs <- matrix(c(1,2,2, 1,3,3, 1,4,4, 2,3,3, 2,4,4, 3,2,3,
                 3,4,1, 4,2,1, 4,3,2), ncol = 3, byrow = TRUE)
# Cooperation in minimum cost arborescence problems
maCooperative(nodes, arcs)
```

maERO — *ERO rule for minimum cost arborescence problems*

Description

Given a graph with a minimum cost arborescence, the `maERO` function divides the cost of the arborescence among the agents according to the ERO rule. For that purpose, the irreducible form of the problem is obtained. The ERO rule is just the Shapley value of the cooperative game associated with the irreducible form.

Usage

```
maERO(nodes, arcs)
```

Arguments

`nodes`	vector containing the nodes of the graph, identified by a number that goes from 1 to the order of the graph.
`arcs`	matrix with the list of arcs of the graph. Each row represents one arc. The first two columns contain the two endpoints of each arc and the third column contains their weights.

Value

`maERO` returns a matrix with the agents and their costs.

See Also

The more general function maRules.

Examples

```
# Graphs
nodes <- 1:4
arcs <- matrix(c(1,2,7, 1,3,6, 1,4,4, 2,3,8, 2,4,6, 3,2,6,
                 3,4,5, 4,2,5, 4,3,7), ncol = 3, byrow = TRUE)
# ERO
maERO(nodes, arcs)
```

`maGames` *Cooperative game associated with minimum cost arborescences*

Description

Given a graph with at least one minimum cost arborescence the `maGames` function builds the cooperative game associated with it.

Usage

```
maGames(nodes, arcs, game = "pessimistic", show.data = TRUE)
```

Arguments

`nodes`	vector containing the nodes of the graph, identified by a number that goes from 1 to the order of the graph.
`arcs`	matrix with the list of arcs of the graph. Each row represents one arc. The first two columns contain the two endpoints of each arc and the third column contains their weights.

game	denotes the game to be obtained, known as the "pessimistic" game.
show.data	logical value indicating if the function displays the console output (TRUE) or not (FALSE). By default its value is TRUE.

Value

maGames returns a vector with the characteristic fuction of the cooperative game associated with the graph.

Examples

```
# Graph
nodes <- 1:4
arcs <- matrix(c(1,2,7, 1,3,6, 1,4,4, 2,3,8, 2,4,6, 3,2,6,
                 3,4,5, 4,2,5, 4,3,7), ncol = 3, byrow = TRUE)
# Cooperative games
maGames(nodes, arcs, game = "pessimistic")
```

---

maIrreducible — *Irreducible form for a minimum cost arborescence problem*

Description

Given a graph with at least one minimum cost arborescence the maIrreducible function obtains the irreducible form.

Usage

```
maIrreducible(nodes, arcs)
```

Arguments

nodes	vector containing the nodes of the graph, identified by a number that goes from 1 to the order of the graph.
arcs	matrix with the list of arcs of the graph. Each row represents one arc. The first two columns contain the two endpoints of each arc and the third column contains their weights.

Value

maIrreducible returns a matrix with the list of arcs of the irreducible form.

Examples

```
# Graph
nodes <- 1:4
arcs <- matrix(c(1,2,7, 1,3,6, 1,4,4, 2,3,8, 2,4,6, 3,2,6,
                 3,4,5, 4,2,5, 4,3,7), ncol = 3, byrow = TRUE)
# Irreducible form
maIrreducible(nodes, arcs)
```

maPessimistic — *Pessimistic game associated with minimum cost arborescences*

Description

Given a graph with at least one minimum cost arborescence, the `maPessimistic` function builds the pessimistic game.

Usage

```
maPessimistic(nodes, arcs)
```

Arguments

`nodes`	vector containing the nodes of the graph, identified by a number that goes from 1 to the order of the graph.
`arcs`	matrix with the list of arcs of the graph. Each row represents one arc. The first two columns contain the two endpoints of each arc and the third column contains their weights.

Value

`maPessimistic` returns a vector with the characteristic function of the pessimitic game.

Examples

```
# Graph
nodes <- 1:4
arcs <- matrix(c(1,2,7, 1,3,6, 1,4,4, 2,3,8, 2,4,6, 3,2,6,
                 3,4,5, 4,2,5, 4,3,7), ncol = 3, byrow = TRUE)
# Pessimistic game
maPessimistic(nodes, arcs)
```

maRules — *Allocation rules for minimum cost arborescence problems*

Description

Given a graph with at least one minimum cost arborescence, the `maRules` function divides the cost of the arborescence among the agents according to "Bird" and "ERO" rules.

Usage

```
maRules(nodes, arcs, rule, show.data = TRUE)
```

Arguments

`nodes`	vector containing the nodes of the graph, identified by a number that goes from 1 to the order of the graph.
`arcs`	matrix with the list of arcs of the graph. Each row represents one arc. The first two columns contain the two endpoints of each arc and the third column contains their weights.
`rule`	denotes the chosen allocation rule: "Bird" or "ERO".
`show.data`	logical value indicating if the function displays the console output (`TRUE`) or not (`FALSE`). By default its value is `TRUE`.

Value

`maRules` returns a matrix with the agents and their costs. It also prints the result in console.

Examples

```
# Graph
nodes <- 1:4
arcs <- matrix(c(1,2,7, 1,3,6, 1,4,4, 2,3,8, 2,4,6, 3,2,6,
                 3,4,5, 4,2,5, 4,3,7), ncol = 3, byrow = TRUE)
# Allocation rules
maRules(nodes, arcs, rule = "Bird")
maRules(nodes, arcs, rule = "ERO")
```

`mstBird` *Bird rule for minimum cost spanning tree problems*

Description

Given a graph with at least one minimum cost spanning tree, the `mstBird` function divides the cost of the tree obtained with Prim's algorithm among the agents. For that purpose it uses the Bird rule, where each agent pays the cost of the arc that connects him to the tree source.

Usage

```
mstBird(nodes, arcs)
```

Arguments

`nodes`	vector containing the nodes of the graph, identified by a number that goes from 1 to the order of the graph.
`arcs`	matrix with the list of arcs of the graph. Each row represents one arc. The first two columns contain the two endpoints of each arc and the third column contains their weights.

Value

`mstBird` returns a matrix with the agents and their costs.

See Also

The more general function mstRules.

Examples

```
# Graphs
nodes <- 1:4
arcs <- matrix(c(1,2,6, 1,3,10, 1,4,6, 2,3,4, 2,4,6, 3,4,4),
               byrow = TRUE, ncol = 3)
# Bird
mstBird(nodes, arcs)
```

`mstCooperative` *Cooperation in minimum cost spanning tree problems*

Description

Given a graph with at least one minimum cost spanning tree, the `mstCooperative` function computes the pessimistic and optimistic games; and the most known allocation rules: "Bird", "Dutta-Kar","Kar" and "ERO".

Usage

```
mstCooperative(nodes, arcs, show.data = TRUE)
```

Arguments

`nodes`	vector containing the nodes of the graph, identified by a number that goes from 1 to the order of the graph.
`arcs`	matrix with the list of arcs of the graph. Each row represents one arc. The first two columns contain the two endpoints of each arc and the third column contains their weights.
`show.data`	logical value indicating if the function displays the console output (`TRUE`) or not (`FALSE`). By default its value is `TRUE`.

Value

`mstCooperative` returns and print a list with the cooperative games and the allocation rules of a minimum cost spanning tree problem.

Examples

```
# Graph
nodes <- 1:4
arcs <- matrix(c(1,2,6, 1,3,10, 1,4,6, 2,3,4, 2,4,6, 3,4,4),
               byrow = TRUE, ncol = 3)
# Cooperation in minimum cost spanning tree problems
mstCooperative(nodes, arcs)
```

---

mstDuttaKar — *Dutta-Kar rule for minimum cost spanning tree problems*

Description

Given a graph with at least one minimum cost spanning tree, the `mstDuttaKar` function divides the cost of the tree obtained with Prim's algorithm among the agents according to the Dutta-Kar rule. This rule specifies that each agent chooses to pay the minimum cost between the last arc that connects him to the source and the cost that rejects his successor. The order is set by Prim's algorithm.

Usage

```
mstDuttaKar(nodes, arcs)
```

Arguments

`nodes`	vector containing the nodes of the graph, identified by a number that goes from 1 to the order of the graph.
`arcs`	matrix with the list of arcs of the graph. Each row represents one arc. The first two columns contain the two endpoints of each arc and the third column contains their weights.

Value

`mstDuttaKar` returns a matrix with the agents and their costs.

See Also

The more general function mstRules.

Examples

```
# Graph
nodes <- 1:4
arcs <- matrix(c(1,2,6, 1,3,10, 1,4,6, 2,3,4, 2,4,6, 3,4,4),
               byrow = TRUE, ncol = 3)
# Dutta-Kar
mstDuttaKar(nodes, arcs)
```

mstEROKruskal — *ERO rule for minimum cost spanning tree problems with Kruskal's algorithm*

Description

Given a graph with at least one minimum cost spanning tree, the `mstEROKruskal` function divides the cost of the tree among the agents according to the ERO rule.

Usage

```
mstEROKruskal(nodes, arcs)
```

Arguments

`nodes`	vector containing the nodes of the graph, identified by a number from 1 until the order of the graph.
`arcs`	matrix with the list of arcs of the graph. Each row represents one arc. The first two columns contain the two endpoints of each arc and the third column contains their weights.

Value

`mstEROKruskal` returns a matrix with the agents and their costs.

See Also

The more general function mstRules.

Examples

```
# Graph
nodes <- 1:4
arcs <- matrix(c(1,2,6, 1,3,10, 1,4,6, 2,3,4, 2,4,6, 3,4,4),
               byrow = TRUE, ncol = 3)
# ERO with Kruskal
mstEROKruskal(nodes, arcs)
```

mstGames *Cooperative games from minimum cost spanning tree problems*

Description

Given a graph with at least one minimum cost spanning tree, `mstGames` builds both cooperative games: the pessimistic and the optimistic game.

The pessimistic game associated with a minimum cost spanning tree problem is a cooperative game in which every coalition of agents obtains the minimum cost assuming that the agents outside the coalition are not connected.

The optimistic game associated with with a minimum cost spanning tree problem is a cooperative game in which every coalition of agents obtains the minimum cost assuming that that the agents outside the coalition are connected. Thus, the agents in the coalition can benefit from their connections to the source

Usage

```
mstGames(nodes, arcs, game, show.data = TRUE)
```

Arguments

`nodes`	vector containing the nodes of the graph, identified by a number that goes from 1 to the order of the graph.
`arcs`	matrix with the list of arcs of the graph. Each row represents one arc. The first two columns contain the two endpoints of each arc and the third column contains their weights.
`game`	denotes the game that we want to obtain: "pessimistic" or "optimistic".
`show.data`	logical value indicating if the function displays the console output (`TRUE`) or not (`FALSE`). By default its value is `TRUE`.

Value

`mstGames` returns a vector with the characteristic fuction of the selected game associated with the graph and prints the result in console.

Examples

```
# Graph
nodes <- 1:4
arcs <- matrix(c(1,2,6, 1,3,10, 1,4,6, 2,3,4, 2,4,6, 3,4,4),
               byrow = TRUE, ncol = 3)
# Cooperative games
mstGames(nodes, arcs, game = "pessimistic")
mstGames(nodes, arcs, game = "optimistic")
```

`mstIrreducible` *Irreducible form for a minimum cost spanning tree problem*

Description

Given a graph with at least one minimum cost spanning tree, the `mstIrreducible` function obtains the irreducible form.

Usage

```
mstIrreducible(nodes, arcs)
```

Arguments

`nodes`	vector containing the nodes of the graph, identified by a number that goes from 1 to the order of the graph.
`arcs`	matrix with the list of arcs of the graph. Each row represents one arc. The first two columns contain the two endpoints of each arc and the third column contains their weights.

Value

`mstIrreducible` returns a matrix with the list of arcs of the irreducible form.

Examples

```
# Graph
nodes <- 1:4
arcs <- matrix(c(1,2,6, 1,3,10, 1,4,6, 2,3,4, 2,4,6, 3,4,4),
               byrow = TRUE, ncol = 3)
# Irreducible form
mstIrreducible(nodes, arcs)
```

`mstKar` *Kar rule for minimum cost spanning tree problems*

Description

Given a graph with at least one minimum cost spanning tree, the `mstKar` function divides the cost of the tree among the agents according to the Kar rule. That rule is obtained with the Shapley value of the pessimistic game.

Usage

```
mstKar(nodes, arcs)
```

Arguments

nodes
: vector containing the nodes of the graph, identified by a number that goes from 1 to the order of the graph.

arcs
: matrix with the list of arcs of the graph. Each row represents one arc. The first two columns contain the two endpoints of each arc and the third column contains their weights.

Value

`mstKar` returns a matrix with the agents and their costs.

See Also

The more general function mstRules.

Examples

```
# Graph
nodes <- 1:4
arcs <- matrix(c(1,2,6, 1,3,10, 1,4,6, 2,3,4, 2,4,6, 3,4,4),
               byrow = TRUE, ncol = 3)
# Kar
mstKar(nodes, arcs)
```

`mstOptimistic` *Optimistic game of a minimum cost spanning tree problem*

Description

Given a graph with at least one minimum cost spanning tree, the `mstOptimistic` function builds the optimistic game associated with it.

Usage

```
mstOptimistic(nodes, arcs)
```

Arguments

nodes
: vector containing the nodes of the graph, identified by a number that goes from 1 to the order of the graph.

arcs
: matrix with the list of arcs of the graph. Each row represents one arc. The first two columns contain the two endpoints of each arc and the third column contains their weights.

Value

`mstOptimistic` returns a vector with the characteristic function of the optimistic game.

See Also

The more general function mstGames.

Examples

```
# Graph
nodes <- 1:4
arcs <- matrix(c(1,2,6, 1,3,10, 1,4,6, 2,3,4, 2,4,6, 3,4,4),
               byrow = TRUE, ncol = 3)
# Optimistic game
mstOptimistic(nodes, arcs)
```

`mstPessimistic` *Pessimistic game from a minimum cost spanning tree problem*

Description

Given a graph with at least one minimum cost spanning tree, the `mstPessimistic` function builds the pessimistic game.

Usage

```
mstPessimistic(nodes, arcs)
```

Arguments

`nodes`	vector containing the nodes of the graph, identified by a number that goes from 1 to the order of the graph.
`arcs`	matrix with the list of arcs of the graph. Each row represents one arc. The first two columns contain the two endpoints of each arc and the third column contains their weights.

Value

`mstPessimistic` returns a vector with the characteristic function of the pessimistic game.

See Also

The more general function mstGames.

Examples

```
# Graph
nodes <- 1:4
arcs <- matrix(c(1,2,6, 1,3,10, 1,4,6, 2,3,4, 2,4,6, 3,4,4),
               byrow = TRUE, ncol = 3)
# Pessimistic game
mstPessimistic(nodes, arcs)
```

`mstRules` *Allocation rules for minimum cost spanning tree problem*

Description

Given a graph with at least one minimum cost spanning tree, the `mstRules` function divides the cost of the tree among the agents according to the most known rules: "Bird", "Dutta-Kar", "Kar", "ERO".

Usage

```
mstRules(nodes, arcs, rule, algorithm = "Kruskal", show.data = TRUE)
```

Arguments

`nodes`	vector containing the nodes of the graph, identified by a number that goes from 1 to the order of the graph.
`arcs`	matrix with the list of arcs of the graph. Each row represents one arc. The first two columns contain the two endpoints of each arc and the third column contains their weights.
`rule`	denotes the chosen allocation rule: "Bird", "Dutta-Kar", "Kar" or "ERO".
`algorithm`	denotes the algorithm used with the ERO rule: "Kruskal".
`show.data`	logical value indicating if the function displays the console output`TRUE` or no `FALSE`. The deafult is `TRUE`.

Value

`mstRules` returns a matrix with the agents and the cost that each one of them has to pay. It also prints the result in console.

Examples

```
# Graph
nodes <- 1:4
arcs <- matrix(c(1,2,6, 1,3,10, 1,4,6, 2,3,4, 2,4,6, 3,4,4),
               byrow = TRUE, ncol = 3)
# Allocation Rules
mstRules(nodes, arcs, rule = "Bird")
mstRules(nodes, arcs, rule = "Dutta-Kar")
mstRules(nodes, arcs, rule = "Kar")
mstRules(nodes, arcs, rule = "ERO", algorithm = "Kruskal")
```

shapleyValue *Shapley value of a cooperative game*

Description

Given a cooperative game, the `shapleyValue` function computes its Shapley value.

Usage

```
shapleyValue(n, S = NULL, v)
```

Arguments

`n`	number of players in the cooperative game.
`S`	vector with all the possible coalitions. If none has been specified the function generates it automatically.
`v`	vector with the characteristic function of the cooperative game.

Details

The Shapley value is a solution concept in cooperative game theory proposed by Lloyd Shapley in 1953. It is obtained as the average of the marginal contributions of the players associated with all the posible orders of the players.

Value

The `shapleyValue` functions returns a matrix with all the marginal contributions of the players (`contributions`) and a vector with the Shapley value (`value`).

References

Lloyd S. Shapley. "A Value for n-person Games". In Contributions to the Theory of Games, volume II, by H.W. Kuhn and A.W. Tucker, editors. Annals of Mathematical Studies v. 28, pp. 307-317. Princeton University Press, 1953.

Examples

```
# Cooperative game
n <- 3  # players
v <- c(4, 4, 4, 8, 8, 8, 14)  # characteristic function
# Shapley value
shapleyValue(n, v = v)
```

Index

cooptrees (cooptrees-package), 2
cooptrees-package, 2

maBird, 3
maCooperative, *2*, 4
maERO, 4
maGames, 5
maIrreducible, 6
maPessimistic, 7
maRules, *3*, *5*, 7
mstBird, 8
mstCooperative, *2*, 9
mstDuttaKar, 10
mstEROKruskal, 11
mstGames, 12, *15*
mstIrreducible, 13
mstKar, 13
mstOptimistic, 14
mstPessimistic, 15
mstRules, *9–11*, *14*, 16

shapleyValue, 17

Bibliografía

[1] “The R Project for Statistical Computing.” http://www.r-project.org/.

[2] Google, “Google’s R Style Guide.” http://google-styleguide.googlecode.com/svn/trunk/Rguide.xml.

[3] Henrik Bengtsson, “R Coding Conventions (RCC).” http://www.maths.lth.se/help/R/RCC/, 2009.

[4] “igraph - Network analysis software.” http://igraph.org/.

[5] “Shiny.” http://shiny.rstudio.com/.

[6] “D3.js - Data-Driven Documents.” http://d3js.org/.

[7] O. Boruvka, “Boruvka, Otakar: Scholarly works,” *Prace Moravske prirodovedecke spolecnosti*, no. 3, pp. 37–58, 1926.

[8] R. C. Prim, “Shortest Connection Networks And Some Generalizations,” *Bell System Technical Journal*, no. 36, pp. 1389–1401, 1957.

[9] J. B. Kruskal, “On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem Author,” *Proceedings of the American Mathematical Society*, vol. 7, no. 1, pp. 48–50, 1956.

[10] J. Edmonds, “Optimum Branchings,” vol. 71, no. 4, pp. 233–240, 1967.

[11] E. W. Dijkstra, “A Note on Two Problems in Connexion with Graphs,” *Numerische Mathematik*, no. 1, pp. 269–271, 1959.

[12] L. R. Ford, “Network Flow Theory,” 1956.

[13] R. Bellman, “On a routing problem,” *Quarterly of Applied Mathematics*, no. 16, pp. 87–90, 1958.

[14] R. E. Gomory and T. C. Hu, “Multi-Terminal Network Flows,” *Journal of the Society for Industrial and Applied Mathematics*, vol. 9, no. 4, pp. 551–570, 1961.

[15] D. Gusfield, “Very simple methods for all pairs network flow analysis,” *Society for Industrial and Applied Mathematics*, vol. 19, no. 1, pp. 143–155, 1990.

[16] L. R. Ford and D. R. Fulkerson, "Maximal flow through a network," *Canadian Journal of Mathematics*, no. 8, pp. 399–404, 1956.

[17] "TUGlab." http://webs.uvigo.es/mmiras/TUGlab/.

[18] C. G. Bird, "On Cost Allocation for a Spanning Tree: A Game Theoretic Approach," *Networks*, no. 6, pp. 335–350, 1976.

[19] B. Dutta and A. Kar, "Cost monotonicity, consistency and minimum cost spanning tree games," *Games and Economic Behavior*, vol. 48, pp. 223–248, Aug. 2004.

[20] A. Kar, "Axiomatization of the Shapley Value on Minimum Cost Spanning Tree Games," *Games and Economic Behavior*, vol. 38, pp. 265–277, Feb. 2002.

[21] L. S. Shapley, "A Value for n-person Games," *Contributions to the Theory of Games*, vol. II, no. 28, pp. 307–317, 1953.

[22] V. Feltkamp, S. H. Tijs, and S. Muto, "On the irreducible core and the equal remaining obligation rule of minimum cost extension problems," *Mimeo. Tilburg University.*, 1994.

[23] G. Bergantiños and J. J. Vidal-Puga, "A fair rule in minimum cost spanning tree problems," *Journal of Economic Theory*, vol. 137, pp. 326–352, Nov. 2007.

[24] G. Bergantiños and J. J. Vidal-Puga, "The optimistic TU game in minimum cost spanning tree problems," *International Journal of Game Theory*, vol. 36, pp. 223–239, Feb. 2007.

[25] B. Dutta and D. Mishra, "Minimum cost arborescences," *Games and Economic Behavior*, vol. 74, pp. 120–143, Jan. 2012.

[26] "shiny-incubator." https://github.com/rstudio/shiny-incubator.

[27] K. Savin, "SelectableRows." https://github.com/ksavin/SelectableRows.

[28] C. Beeley, *Web Application Development with R Using Shiny.* 2013.

[29] D. Jungnickel, *Graphs, Networks and Algorithms.* Spiringer, fourth edi ed., 2013.

[30] J. L. Gross and J. Yellen, *Graph Theory and Its Applications.* Chapman and Hall/CRC, second edi ed., 2006.