



Universidade de Vigo

Trabajo Fin de Máster

Implementación en R de reglas de reparto en el árbol de mínimo coste

Lucía Souto Vázquez

Máster en Técnicas Estadísticas

Curso 2025-2026

Propuesta de Trabajo Fin de Máster

Título en galego: Implementación en R de regras de reparto na árbore de custo mínimo
Título en español: Implementación en R de reglas de reparto en el árbol de mínimo coste
English title: R Implementation of Cost Allocation Rules for the Minimum Cost Spanning Tree Problem
Modalidad: Modalidad A
Autora: Lucía Souto Vázquez, Universidad de Santiago de Compostela
Directores: Gustavo Bergantiños Cid, Universidad de Vigo; Alejandro Saavedra Nieves, Universidad de Santiago de Compostela
Breve resumen del trabajo: Este trabajo revisa las principales reglas de reparto de costes en problemas del árbol de mínimo coste, tanto algorítmicas como basadas en teoría de juegos cooperativos, y presenta <code>mcstprules</code> , un paquete de R que las implementa, compara y analiza en cuanto a estabilidad y sensibilidad.
Recomendaciones: Haber cursado la materia de Redes y Planificación.
Otras observaciones: Ayudaría, aunque no es necesario, haber cursado materias de teoría de juegos.

Don Gustavo Bergantiños Cid, catedrático de la Universidad de Vigo y don Alejandro Saavedra Nieves, titular de la Universidad de Santiago de Compostela, informan que el Trabajo Fin de Máster titulado

Implementación en R de reglas de reparto en el árbol de mínimo coste

fue realizado bajo su dirección por doña Lucía Souto Vázquez para el Máster en Técnicas Estadísticas. Estimando que el trabajo está terminado, dan su conformidad para su presentación y defensa ante un tribunal. Además, don Gustavo Bergantiños Cid, don Alejandro Saavedra Nieves y doña Lucía Souto Vázquez

☒ sí ☐ no

autorizan a la publicación de la memoria en el repositorio de acceso público asociado al Máster en Técnicas Estadísticas.

En Santiago de Compostela, a 16 de julio de 2026.

El director:
Don Gustavo Bergantiños Cid

**BERGANTIÑ
O CID
GUSTAVO -
34957737Z**

Firmado digitalmente
por BERGANTIÑO CID
GUSTAVO -
34957737Z
Fecha: 2026.07.16
08:28:56 +02'00'

El director:
Don Alejandro Saavedra Nieves

**SAAVEDRA
NIEVES
ALEJANDRO -
45873390M**

Firmado digitalmente
por SAAVEDRA NIEVES
ALEJANDRO -
45873390M
Fecha: 2026.07.16
08:44:00 +02'00'

La autora:
Doña Lucía Souto Vázquez

**SOUTO
VAZQUEZ
LUCIA -
54159489D**

Firmado
digitalmente por
SOUTO VAZQUEZ
LUCIA - 54159489D
Fecha: 2026.07.16
01:03:30 +02'00'

Declaración responsable. Para dar cumplimiento a la Ley 3/2022, de 24 de febrero, de convivencia universitaria, referente al plagio en el Trabajo Fin de Máster (Artículo 11, Disposición 2978 del BOE núm. 48 de 2022), **la autora declara** que el Trabajo Fin de Máster presentado es un documento original en el que se han tenido en cuenta las siguientes consideraciones relativas al uso de material de apoyo desarrollado por otros/as autores/as:

- Todas las fuentes usadas para la elaboración de este trabajo han sido citadas convenientemente (libros, artículos, apuntes de profesorado, páginas web, programas, ...)
- Cualquier contenido copiado o traducido textualmente se ha puesto entre comillas, citando su procedencia.
- Se ha hecho constar explícitamente cuando un capítulo, sección, demostración, ... sea una adaptación casi literal de alguna fuente existente.

Y, acepta que, si se demostrara lo contrario, se le apliquen las medidas disciplinarias que correspondan.

Agradecimientos

Escribir estas líneas marca el final de una etapa muy importante de mi vida, un camino que no habría podido recorrer sin ayuda.

En primer lugar, quiero dar las gracias a mis tutores, Gustavo y Alejandro. Gracias por guiarme, por ayudarme en todo momento y por estar ahí cuando más lo necesitaba. A pesar del esfuerzo que exige un proyecto así, he disfrutado muchísimo trabajando y aprendiendo a vuestro lado.

Por último, gracias de corazón a mi familia, en especial a mis padres, a mi hermano y a mi pareja. Vosotros habéis vivido la parte más dura de este trabajo. Gracias por vuestra paciencia infinita, por aguantarme en los peores momentos y por no dejar que me rindiera nunca. Este logro también es vuestro.

Índice general

Resumen	XI
Introducción	XIII
1 Preliminares	1
1.1 Conceptos básicos de teoría de grafos	1
1.2 Conceptos básicos de teoría de juegos cooperativos	3
2 El problema del árbol de mínimo coste (MCSTP)	11
2.1 Definición formal del MCSTP	11
2.2 Algoritmos de construcción: Prim, Kruskal y Borůvka	13
2.3 La forma irreducible y la red ciclo-completa	18
2.4 Descomposición en conos	21
2.5 Juegos cooperativos asociados al MCSTP	22
3 Asignación de costes	27
3.1 Reglas definidas a través del MCSTP	27
3.1.1 Reglas basadas en Prim: Bird y Dutta-Kar	27
3.1.2 Reglas basadas en Kruskal: folk y Shapley ponderado	30
3.1.3 Regla basada en Borůvka	34
3.1.4 Reglas basadas en la descomposición en conos	36
3.2 Reglas definidas a través de juegos cooperativos	38
3.3 El método de Monte Carlo en el cálculo de asignaciones	43
4 El paquete mcstprules: implementación y aplicación	47
4.1 Arquitectura y diseño del paquete	48
4.1.1 Estructura de archivos y dependencias	49
4.1.2 Flexibilidad en la entrada de datos	50
4.1.3 Tratamiento de empates y soluciones aproximadas	54
4.1.4 Recursos adicionales	54
4.2 Reglas definidas a través del MCSTP	55
4.2.1 Funciones mcstBird y mcstDuttaKar	56

4.2.2	Funciones <code>mcstFolk</code> , <code>mcstOWShapley</code> y <code>mcstPWShapley</code>	60
4.2.3	Función <code>mcstBoruvka</code>	62
4.2.4	Función <code>mcstCone</code>	63
4.3	Reglas definidas a través de juegos cooperativos	65
4.3.1	Funciones <code>mcstGamePrivate</code> y <code>mcstKar</code>	66
4.3.2	Función <code>mcstGameIrred</code>	68
4.3.3	Función <code>mcstGameOpt</code>	69
4.3.4	Función <code>mcstGamePublic</code>	71
4.3.5	Función <code>mcstGameCC</code>	72
4.4	Herramientas de análisis	73
4.4.1	Comparativa de reglas: <code>mcstCompare</code>	73
4.4.2	Análisis del núcleo: <code>mcstCoreCheck</code> , <code>mcstCorePoints</code> y <code>mcstCorePlot</code>	74
4.4.3	Rango de estabilidad: <code>mcstStabilityRange</code>	79
4.4.4	Análisis de sensibilidad: <code>mcstSensitivity</code>	80
5	Análisis de rendimiento computacional	85
5.1	Comparativa de tiempos de ejecución: exacto vs. Monte Carlo	85
5.2	Precisión de la aproximación Monte Carlo	89
6	Conclusiones y trabajo futuro	93
A	Manual del paquete <code>mcstprules</code>	97
	Bibliografía	147

Resumen

Resumen en español

El problema del árbol de mínimo coste (MCSTP, por sus siglas en inglés) es uno de los problemas clásicos de la investigación operativa. Consiste en encontrar la forma de conectar un conjunto de agentes a una fuente común minimizando el coste total de la red. En los últimos años, se ha abordado un nuevo aspecto: cómo repartir ese coste entre los diversos agentes. Para ello, se propusieron varias reglas de reparto que se pueden dividir en dos grandes familias: las reglas basadas en los algoritmos clásicos de construcción del árbol y las reglas basadas en los juegos cooperativos. En este trabajo se lleva a cabo una revisión bibliográfica de las principales reglas de reparto existentes, que se concluye con la creación de `mcstprules`, un paquete de R que implementa, compara y analiza dichas reglas, incorporando herramientas para estudiar la estabilidad de los repartos. Finalmente, se evalúa el rendimiento computacional del paquete, comparando el cálculo exacto de ciertas reglas con una aproximación basada en el método Monte Carlo, útil para problemas de gran tamaño donde el cálculo exacto se vuelve inviable.

English abstract

The Minimum Cost Spanning Tree Problem (MCSTP) is one of the classic problems in operations research. It consists of finding the most efficient way to connect a set of agents to a common source while minimizing the total network cost. In recent years, a new aspect has been addressed: how to allocate this cost among the various agents. To this end, several cost allocation rules have been proposed, which can be divided into two major families: rules based on classic tree construction algorithms and rules based on cooperative games. This work carries out a literature review of the main existing allocation rules, concluding with the creation of `mcstprules`, an R package that implements, compares, and analyzes these rules, incorporating tools to study the stability of the allocations. Finally, the computational performance of the package is evaluated, comparing the exact calculation of certain rules with an approximation based on the Monte Carlo method, which is useful for large-scale problems where exact computation becomes unfeasible.

Introducción

Contexto y motivación

El diseño eficiente de redes para conectar múltiples puntos es un problema clásico muy extendido. Esta situación se ilustra perfectamente al diseñar, por ejemplo, una red de abastecimiento de agua para una serie de poblaciones. Aunque siempre existe la posibilidad de construir tuberías directas desde un embalse central hasta cada localidad, esta solución a menudo genera infraestructuras redundantes y costes excesivos. La alternativa más inteligente es permitir que los municipios se conecten entre sí en aquellos tramos que resulten más económicos que una conexión directa a la fuente. El objetivo radica en encontrar la red que logre llevar el suministro a todas las poblaciones asumiendo el menor gasto posible.

Este escenario es la representación clásica de uno de los problemas más estudiados de la teoría de grafos: el problema del árbol de mínimo coste (*Minimum Cost Spanning Tree Problem*, MCSTP). Intuitivamente, consiste en conectar un conjunto de nodos (las poblaciones) mediante un subconjunto de aristas (las tuberías) a una fuente común de suministro (el embalse), evitando crear conexiones innecesarias y minimizando el coste total de la red.

La necesidad de resolver este tipo de problemas tiene sus orígenes en 1926 de la mano del matemático checo Otakar Borůvka (1926). El gobierno le encargó diseñar la red eléctrica más económica para iluminar la extensa región rural de Moravia del Sur, situada en la República Checa. Para evitar gastar cable de cobre de forma innecesaria, Borůvka desarrolló el primer algoritmo matemático capaz de calcular esa red de forma óptima (Nešetřil et al. 2001).

Hoy, un siglo después de que Borůvka iluminara Moravia, el MCSTP se ha convertido en la base sobre la que se diseñan las redes de fibra óptica, las rutas de datos en internet o incluso algunos algoritmos modernos de inteligencia artificial basados en grafos, entre muchas otras cosas.

Sin embargo, encontrar la red óptima es solo la mitad del problema. Una vez construido el árbol de coste mínimo, surge de manera natural la pregunta de cómo se debe repartir el coste entre quienes se benefician de la red construida. Esta cuestión ha dado lugar a un campo de estudio propio, en el que se han propuesto numerosas reglas de reparto que podemos dividir en dos grandes familias: las reglas obtenidas directamente a partir de la estructura de la red y de los algoritmos clásicos de construcción del árbol y las reglas basadas en juegos cooperativos. Bergantiños y Vidal-Puga (2021) ofrecen una revisión exhaustiva de ambos enfoques, que será la referencia teórica principal empleada en este trabajo. Asimismo, cabe señalar que existen otros trabajos académicos que revisan parte de la literatura del MCSTP, como Fontenla Cadavid (2014), Gómez Saura (2016) y Moor et al. (2019), aunque el enfoque y los objetivos de estas investigaciones son diferentes a los de este trabajo.

A pesar de la abundante literatura teórica sobre este problema, las herramientas computacionales disponibles para aplicar estas reglas son limitadas. En el entorno de R, los paquetes `optrees` (Fontenla 2014b) y `cooptrees` (Fontenla 2014a), descritos en Fontenla Cadavid (2014), supusieron un primer

acercamiento al problema, pero sus propios autores señalaron limitaciones importantes. Esta carencia de una herramienta completa es la motivación principal de este trabajo.

Objetivos

Partiendo de la motivación anterior, este trabajo plantea dos objetivos generales. En primer lugar, realizar una revisión bibliográfica de las reglas de reparto de costes existentes en la literatura para el MCSTP, tanto de las basadas en algoritmos como de las basadas en teoría de juegos cooperativos, presentando su fundamento teórico de forma clara y didáctica. En segundo lugar, desarrollar un paquete de R que implemente dichas reglas, superando las limitaciones conocidas de las herramientas previas.

Con respecto a este segundo objetivo, se pretende ofrecer una herramienta completa que cuente con las siguientes funcionalidades:

- Una interfaz flexible que acepte los datos de entrada en distintos formatos.
- Capacidad para resolver los empates que puedan surgir en la construcción del árbol óptimo de forma equitativa, extendiendo las reglas afectadas mediante el promedio de las distintas soluciones posibles.
- Inclusión de la totalidad de los juegos cooperativos asociados al MCSTP descritos en la literatura junto con sus correspondientes reglas de reparto.
- Herramientas de análisis que permitan estudiar la estabilidad de las soluciones y la sensibilidad ante variaciones en los costes de la red.
- Capacidad para escalar a redes de gran tamaño mediante aproximaciones eficientes basadas en el método Monte Carlo.

Estructura

Esta memoria se organiza en seis capítulos y un anexo. El Capítulo 1 recoge los conceptos preliminares de teoría de grafos y teoría de juegos cooperativos necesarios para el desarrollo del resto de capítulos. El Capítulo 2 presenta formalmente el MCSTP y los algoritmos clásicos de construcción del árbol óptimo, así como otras estructuras relevantes asociadas al problema, la técnica de descomposición en conos y los distintos juegos cooperativos que pueden definirse a través de él. En el Capítulo 3 se presentan las reglas de reparto existentes en la literatura clasificadas en dos grupos bien diferenciados: las reglas definidas a través del MCSTP y las reglas definidas a través de los juegos cooperativos vistos en el capítulo anterior. Además, se emplea un mismo ejemplo que sirve de referencia para ilustrar todos los resultados teóricos expuestos.

La parte central de la memoria la constituye el Capítulo 4 que presenta `mcstprules` (Souto Vázquez 2026), el paquete de R que implementa todas las reglas presentadas en el Capítulo 3, así como otras herramientas de análisis de estabilidad y sensibilidad de los repartos. En este capítulo se describe la arquitectura interna del paquete, sus dependencias y el funcionamiento de cada una de sus funciones, ilustrando los resultados para el mismo ejemplo tratado en los capítulos previos. El Capítulo 5 evalúa el rendimiento computacional del paquete, comparando los cálculos exactos con una aproximación Monte Carlo tanto en tiempo de ejecución como en precisión. Por último, el Capítulo 6 recoge las principales conclusiones del trabajo y plantea posibles líneas de investigación futuras. Además, el Anexo A contiene el manual técnico completo del paquete, con descripciones detalladas de todas las funciones implementadas.

Capítulo 1

Preliminares

Este capítulo establece el marco formal necesario para el desarrollo del trabajo. En primer lugar, en la Sección 1.1, se introducen los conceptos fundamentales de la teoría de grafos, que proporcionan el soporte estructural para modelar problemas de conexión y redes. A continuación, en la Sección 1.2, se presentan las nociones básicas de la teoría de juegos cooperativos, una de las herramientas analíticas utilizadas para abordar el problema del reparto de costes.

1.1. Conceptos básicos de teoría de grafos

La teoría de grafos es el lenguaje natural para modelar problemas de conexión y redes. En esta sección se presentan los conceptos y la notación fundamentales que servirán de base para el desarrollo de este trabajo. Las definiciones y resultados expuestos a continuación siguen la base teórica general de Diestel (2017), adaptando la terminología y notación.

Intuitivamente, un grafo no es más que una colección de puntos junto con las conexiones que existen entre ellos, sin importar la posición geométrica de los puntos ni la longitud o forma de las líneas que los unen, sino únicamente la estructura de conectividad.

Definición 1.1.1 (Grafo). Un *grafo* G es un par (V, A) , donde V es un conjunto de vértices (o nodos) y A es un conjunto de aristas (o arcos). Cada arista $a \in A$ representa la conexión entre un par de vértices.

El número de vértices de un grafo G se denomina *orden* y se denota por $|G|$, mientras que el número de aristas se conoce como *tamaño* y se representa mediante $\|G\|$. Dependiendo de estos valores, los grafos pueden clasificarse en finitos, infinitos o numerables. En este trabajo, asumiremos siempre que los grafos son finitos.

Por otro lado, la naturaleza de las conexiones entre los vértices puede variar según si tienen o no un sentido de flujo:

Definición 1.1.2 (Grafo no dirigido). Un grafo $G = (V, A)$ es *no dirigido* si sus aristas carecen de orientación específica. Cada arista $a \in A$ vincula a dos vértices distintos¹ $i, j \in V$, y se denota mediante el par desordenado (i, j) o, alternativamente, $\{i, j\}$.

Dado que este trabajo se centra exclusivamente en el análisis de redes con relaciones bidireccionales, en lo sucesivo utilizaremos el término *grafo* para referirnos a un *grafo no dirigido*.

¹La condición de que los vértices sean distintos se debe a que, a lo largo de este trabajo, consideraremos *grafos simples*, excluyendo así la existencia de bucles (aristas que conectan un nodo consigo mismo) y de aristas múltiples entre el mismo par de nodos.

En muchas aplicaciones prácticas, las conexiones de una red poseen características cuantitativas asociadas, como distancias, capacidades o costes asociados a su construcción. Para incorporar esta información al modelo geométrico, basta con añadir al grafo una función que etiquete cada arista con un número real.

Definición 1.1.3 (Grafo ponderado). Un *grafo ponderado* es un grafo $G = (V, A)$ al que se le asocia una función de peso $w : A \rightarrow \mathbb{R}$, que asigna a cada arista $(i, j) \in A$ un valor numérico real.

La Figura 1.1 muestra un ejemplo básico de grafo ponderado sobre un conjunto de tres nodos $V = \{1, 2, 3\}$, donde cada arista lleva asociado un coste de conexión.

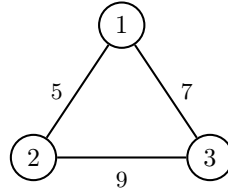


Figura 1.1: Grafo ponderado sobre $V = \{1, 2, 3\}$ con $A = \{(1, 2), (1, 3), (2, 3)\}$.

Al analizar las subestructuras de un grafo, hay dos casos que resultan de especial interés: cuando todos los elementos están conectados directamente entre sí, y cuando el estudio se limita a una parte de la red original.

Definición 1.1.4 (Grafo completo y subgrafo inducido). Un *grafo completo* es un grafo en el que cada par de vértices distintos está conectado por una arista. Por otro lado, dado un subconjunto de vértices $S \subseteq V$, el *subgrafo inducido* por S , denotado como $G[S]$, es el grafo formado por los vértices de S y todas las aristas de A que tienen ambos extremos en S .

A la hora de estudiar la topología de una red, es fundamental analizar cómo se transmite la información o el flujo entre los distintos elementos para así identificar las posibles rutas y evitar redundancias en la estructura.

Definición 1.1.5 (Camino, ciclo y árbol). Dado un grafo $G = (V, A)$:

- Un *camino* entre los nodos i y j es una secuencia de nodos distintos $(i_0, i_1, \dots, i_k)$ tal que $i_0 = i$, $i_k = j$, y la arista $(i_{r-1}, i_r) \in A$ para todo $r \in \{1, 2, \dots, k\}$.
- Un grafo es *conexo* si existe al menos un camino entre cualquier par de nodos de V .
- Un *ciclo* es un camino de al menos tres nodos donde el primer y el último nodo coinciden ($i_0 = i_k$), sin repetir ningún otro nodo ni arista en su recorrido.
- Un *árbol* es un subgrafo conexo que no contiene ciclos.

Observemos que el grafo de la Figura 1.1 contiene exactamente un ciclo definido por la secuencia $(1, 2, 3, 1)$. Por tanto, dicho grafo no cumple las condiciones para ser un árbol. Sin embargo, es posible transformarlo en uno eliminando cualquiera de sus tres aristas.

Dependiendo de la posición y del número de conexiones que posea un vértice dentro de la topología de un árbol, se puede clasificar su función e importancia dentro de la estructura:

Definición 1.1.6 (Grado de un vértice). Dado un grafo $G = (V, A)$ y un vértice $i \in V$, el *grado* de i , denotado como $d_G(i)$ o, simplemente, $d(i)$, es el número de aristas de A que tienen a i como uno de sus extremos.

Si un vértice tiene grado cero, se dice que está *aislado*. En el contexto de un árbol, un vértice i se denomina *nodo hoja* si $d(i) = 1$, es decir, si solo está conectado al resto del árbol a través de una única arista.

Retomemos el grafo de la Figura 1.1. Si eliminamos, por ejemplo, la arista $(2, 3)$, obtenemos el árbol que se detalla en la Figura 1.2, en el que $d(1) = 2$ y $d(2) = d(3) = 1$, lo que convierte a estos dos últimos en nodos hoja.

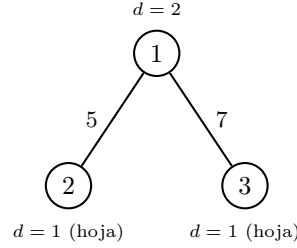


Figura 1.2: Árbol obtenido al eliminar la arista $(2, 3)$ de la Figura 1.1 con el grado de cada vértice.

Por último, cuando un grafo es conexo, resulta de gran interés hallar las subestructuras mínimas que mantengan conectados todos los elementos de la red sin añadir conexiones innecesarias.

Definición 1.1.7 (Árbol generador). Un *árbol generador* (o *spanning tree*) de un grafo conexo $G = (V, A)$ es un árbol que contiene a la totalidad de los vértices del conjunto V .

El subgrafo expuesto en la Figura 1.2 constituye un ejemplo de árbol generador para el grafo de la Figura 1.1, dado que carece de ciclos y mantiene conectados los tres vértices originales de la red.

Observación 1.1.1. La configuración de la Figura 1.2 no representa la única solución posible. Si en lugar de prescindir de la arista $(1, 2)$ en la Figura 1.1 se hubiese eliminado el arco $(1, 2)$ o el arco $(1, 3)$, se habrían obtenido árboles generadores alternativos igualmente válidos, cuya única diferencia sería su coste total ($9 + 7 = 16$ y $5 + 9 = 14$, respectivamente, frente al valor de $5 + 7 = 12$ del árbol seleccionado). En el Capítulo 2 dedicaremos una sección a la búsqueda y optimización de estas estructuras en grafos ponderados.

Como última observación sobre la terminología que se utilizará a lo largo del trabajo, usaremos, al igual que Bergantiños y Vidal-Puga (2021), los términos *arista* y *arco* como sinónimos para referirnos a una conexión no dirigida entre dos nodos, es decir, $(i, j) = (j, i)$. Aunque en otras ramas de la teoría de grafos el término *arco* se utiliza para conexiones dirigidas, en este trabajo no se hace dicha distinción, ya que las conexiones consideradas son simétricas y no dan lugar a ambigüedad.

1.2. Conceptos básicos de teoría de juegos cooperativos

La teoría de grafos nos permite describir la estructura de conectividad de una red, pero no contempla que sus elementos puedan tomar decisiones. Sin embargo, en muchos escenarios reales, los vértices representan entidades con capacidad de decisión, como ciudades, empresas o personas. Cuando estos elementos actúan de tal manera que pueden colaborar para optimizar un recurso común, necesitamos analizar cómo cooperan y de qué forma deben repartirse los costes (o beneficios) generados. Para modelar esta interacción, utilizamos una herramienta específica: la teoría de juegos cooperativos.

De forma análoga a la Sección 1.1, aquí se introducen únicamente los conceptos y la notación que necesitaremos más adelante. Seguimos, en líneas generales, la base de González-Díaz et al. (2023), complementado con la organización conceptual y la terminología empleadas en Moor et al. (2019) y Bergantiños y Vidal-Puga (2021).

Un juego cooperativo recoge, de forma muy general, qué resultado obtendría cada posible subgrupo de jugadores (coalición) si decidiera actuar de forma independiente, sin necesidad todavía de especificar cómo se reparte dicho resultado entre sus miembros. En la literatura, este tipo de escenarios donde los resultados se pueden medir y repartir libremente se conocen como juegos con utilidad transferible.

Definición 1.2.1 (Juego TU). Un *juego cooperativo con utilidad transferible* o, simplemente, *juego TU*, es un par (N, v) ², donde $N = \{1, 2, \dots, n\}$ es un conjunto finito de jugadores (o agentes) y $v : 2^N \rightarrow \mathbb{R}$ es la función característica. Esta función asigna a cada coalición $S \subseteq N$ el valor $v(S)$, que representa el resultado que ese grupo puede obtener por sí mismo actuando de forma independiente, con la convención $v(\emptyset) = 0$.

La Definición 1.2.1 es aplicable tanto a juegos de beneficios como de costes. Sin embargo, como este trabajo aborda un problema de redes concreto, nos centraremos exclusivamente en estos últimos. En consecuencia, a partir de ahora, $v(S)$ representará el coste mínimo de la coalición S para conectar a sus miembros. El siguiente paso es analizar cómo se distribuye el coste total del grupo entre todos los participantes.

Definición 1.2.2 (Asignación). Una *asignación* es un vector $x \in \mathbb{R}^N$ cuya componente x_i representa el pago correspondiente al jugador i . Para simplificar la notación, escribiremos $x(S) = \sum_{i \in S} x_i$ para cualquier $S \subseteq N$.

Por otro lado, es razonable esperar que cooperar nunca perjudique a los agentes. Si dos coaliciones disjuntas se unen, el coste de la coalición conjunta no debería superar la suma de lo que pagarían por separado. Un juego (N, v) que cumple esta condición se denomina *subaditivo*:

$$v(S) + v(T) \geq v(S \cup T), \quad \forall S, T \subseteq N \text{ con } S \cap T = \emptyset.$$

Esta propiedad refleja que formar la coalición de todos los agentes, N , siempre genera un resultado igual o mejor que actuar de forma separada. El siguiente ejemplo ilustra estas nociones con un juego de tres agentes.

Ejemplo 1.2.1 (Juego de costes). Consideremos el juego de costes (N, v) para tres agentes recogido en la Tabla 1.1. Si cada agente actuase por su cuenta, el coste total sería $v(\{1\}) + v(\{2\}) + v(\{3\}) = 6 + 4 + 7 = 17$, mientras que si cooperasen los tres juntos, el coste conjunto descendería a $v(N) = 10$. Dicha cooperación generaría, por tanto, un ahorro de 7 unidades frente a actuar de forma completamente independiente.

S	$\emptyset$	$\{1\}$	$\{2\}$	$\{3\}$	$\{1, 2\}$	$\{1, 3\}$	$\{2, 3\}$	N
$v(S)$	0	6	4	7	7	9	8	10

Tabla 1.1: Función característica de un juego de costes subaditivo con 3 agentes.

Una vez fijado el coste de cada coalición, el siguiente paso es decidir cómo repartirlo, ya que no cualquier reparto resulta razonable. Parece justo exigir, como mínimo, que se reparta exactamente el coste total y que a ningún agente le convenga más actuar por su cuenta.

Definición 1.2.3 (Imputación y núcleo). Una asignación x es una *imputación* si reparte exactamente el valor total del juego entre los jugadores (eficiencia) y, además, a ningún jugador le corresponde más de lo que le correspondería actuando en solitario (racionalidad individual). Por tanto, el conjunto de todas las imputaciones $I(N, v)$ se define matemáticamente como

$$I(N, v) = \{x \in \mathbb{R}^N : x(N) = v(N) \text{ y } x_i \leq v(\{i\}), \quad \forall i \in N\}.$$

²En lo sucesivo, nos referiremos al juego indistintamente mediante el par (N, v) o simplemente v .

El *núcleo* (o *core*), introducido por Gillies (1953), es el subconjunto de imputaciones en las que, además, a ninguna coalición le corresponde más de lo que le correspondería actuando por su cuenta (racionalidad coalicional):

$$Core(N, v) = \{x \in I(N, v) : x(S) \leq v(S), \quad \forall S \subseteq N\}.$$

En este sentido, el núcleo representa el conjunto de soluciones *estables* del juego. Al garantizar que ningún subgrupo pague más de lo que le costaría por su cuenta, todos prefieren seguir cooperando y el acuerdo no se rompe.

Para el juego del Ejemplo 1.2.1, toda imputación debe cumplir $x_1 + x_2 + x_3 = 10$ (eficiencia) y $x_1 \leq 6$, $x_2 \leq 4$, $x_3 \leq 7$ (racionalidad individual). La igualdad de eficiencia define un plano en el espacio $\mathbb{R}^3$. Al intersecar este plano con las tres cotas superiores, obtenemos la región acotada que conforma el conjunto de imputaciones.

Geométricamente, esta región forma un triángulo. Sus vértices se obtienen fijando dos de las tres restricciones de racionalidad individual en su límite de igualdad, despejando la tercera coordenada mediante la condición de eficiencia. Así, el conjunto de imputaciones es la envoltura convexa de los siguientes puntos:

$$I(N, v) = \text{conv}\{(6, 4, 0), (6, -3, 7), (-1, 4, 7)\}.$$

En general, $I(N, v)$ se corresponde con un *símples* de dimensión $n - 1$. En la práctica, esto significa que el espacio de asignaciones se puede dibujar como un segmento de recta para dos jugadores, un triángulo para tres o un tetraedro para cuatro.

Ejemplo 1.2.2 (Cálculo del núcleo — continuación del Ejemplo 1.2.1). Si aplicamos la condición de eficiencia ($x_1 + x_2 + x_3 = 10$) sobre las restricciones de las coaliciones entre dos agentes (racionalidad coalicional), obtenemos una cota inferior para cada jugador:

$$\begin{aligned} x_1 + x_2 \leq 7 &\implies x_3 \geq 3 \\ x_1 + x_3 \leq 9 &\implies x_2 \geq 1 \\ x_2 + x_3 \leq 8 &\implies x_1 \geq 2 \end{aligned}$$

Estas nuevas restricciones activas cortan el triángulo de imputaciones de manera que el núcleo toma la forma geométrica de un cuadrilátero inscrito en él. Cada vértice del núcleo se obtiene fijando dos de estas tres cotas en su valor de igualdad, quedando así la tercera coordenada determinada por la condición de eficiencia. Por ejemplo, fijando $x_2 = 1$ y $x_3 = 3$ se obtiene $x_1 = 10 - 1 - 3 = 6$, lo que nos da el vértice $(6, 1, 3)$. Procediendo de forma análoga con las demás combinaciones y comprobando que se respetan el resto de desigualdades, hallamos los demás puntos. Así, el núcleo es la envoltura convexa de los siguientes cuatro vértices:

$$Core(N, v) = \text{conv}\{(2, 1, 7), (6, 1, 3), (3, 4, 3), (2, 4, 4)\}.$$

Dado que todos los puntos válidos (tanto imputaciones como elementos del núcleo) residen sobre el plano definido por $x_1 + x_2 + x_3 = 10$, es posible proyectar este espacio tridimensional sobre un plano bidimensional sin ninguna pérdida de información. La Figura 1.3 ilustra esta proyección geométrica: el triángulo exterior punteado delimita el conjunto total de imputaciones, mientras que las rectas sólidas representan los hiperplanos de las coaliciones que acotan y definen el núcleo (área sombreada).

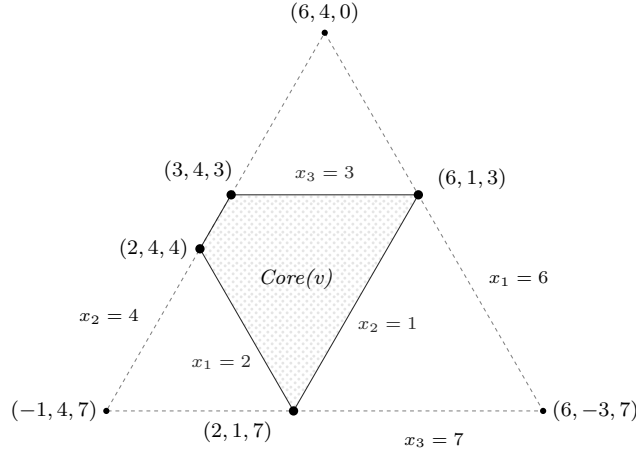


Figura 1.3: Proyección geométrica del conjunto de imputaciones (triángulo punteado) y del núcleo (área sombreada).

El núcleo nos dice qué repartos son estables, pero en general contiene infinitos puntos. Para seleccionar un único reparto, recurriremos a dos reglas: el valor de Shapley y el nucleolo. Antes, conviene introducir una propiedad de los juegos que será relevante en lo que sigue:

Definición 1.2.4 (Juego cóncavo). Un juego de costes (N, v) es *cóncavo* si, para todo $S \subseteq T \subseteq N$ y todo $i \in N \setminus T$,

$$v(S \cup \{i\}) - v(S) \geq v(T \cup \{i\}) - v(T),$$

es decir, cuanto más grande es la coalición a la que se incorpora un agente, menor es su contribución marginal en términos de coste (Bergantiños y Vidal-Puga 2021).

Para definir formalmente esta idea de contribución marginal necesitamos antes fijar un orden de llegada de los agentes al juego.

Definición 1.2.5 (Vector marginal). Dado un orden π de los agentes de N , sea $Pre(i, \pi)$ el conjunto de jugadores que preceden a i según dicho orden. El *vector marginal* asociado a $\pi \in \Pi(N)$, denotado por $m^\pi(v) \in \mathbb{R}^N$, tiene como componente i -ésima la contribución marginal del agente i al incorporarse a los jugadores que ya están presentes en π . Esto es:

$$m_i^\pi(v) = v(Pre(i, \pi) \cup \{i\}) - v(Pre(i, \pi)).$$

Precisamente, la relación entre los juegos cóncavos y los vectores marginales da lugar a un resultado fundamental en teoría de juegos, demostrado originalmente por Shapley (1971), que permite caracterizar geométricamente el núcleo.

Teorema 1.2.1. Sea (N, v) un juego de costes cóncavo. Entonces, su núcleo es no vacío y coincide exactamente con la envolvente convexa de sus vectores marginales. Formalmente:

$$Core(N, v) = \text{conv}\{m^\pi(v) : \pi \in \Pi(N)\}.$$

En consecuencia, los vértices del núcleo son precisamente los vectores marginales del juego.

Además de definir la estructura geométrica del núcleo, estos vectores marginales son clave para calcular la primera regla de reparto que veremos. El valor de Shapley promedia esta contribución marginal sobre todos los posibles órdenes de llegada, y es, con diferencia, la regla de reparto más estudiada en la literatura.

Definición 1.2.6 (Valor de Shapley). El *valor de Shapley* de un juego de costes (N, v) , denotado por $Sh(N, v)$, es el promedio de los vectores marginales sobre todos los $n!$ órdenes posibles Π_N :

$$Sh(N, v) = \frac{1}{n!} \sum_{\pi \in \Pi_N} m^\pi(v).$$

De forma equivalente, puede expresarse como una suma ponderada de contribuciones marginales sobre las subcoaliciones que no contienen a i :

$$Sh_i(N, v) = \sum_{S \subseteq N \setminus \{i\}} \frac{|S|!(n - |S| - 1)!}{n!} [v(S \cup \{i\}) - v(S)], \quad \forall i \in N.$$

El valor de Shapley (Shapley 1953) es, además, la única regla de reparto que satisface simultáneamente los axiomas de eficiencia (el coste total se distribuye por completo), simetría (jugadores con la misma contribución reciben la misma asignación), jugador nulo (jugadores cuya contribución marginal es siempre nula reciben una asignación nula) y aditividad (la solución de la suma de un número determinado de juegos equivale a la suma de las soluciones individuales).

Ejemplo 1.2.3 (Vectores marginales y valor de Shapley — continuación del Ejemplo 1.2.1). Para calcular el valor de Shapley del juego del Ejemplo 1.2.1, calculamos primero los vectores marginales $m^\pi(v)$ para los $3! = 6$ órdenes de llegada posibles. Por ejemplo, para el orden $\pi = (1, 2, 3)$, cada agente paga su contribución marginal al incorporarse:

$$\begin{aligned} m_1^\pi(v) &= v(\{1\}) - v(\emptyset) = 6 \\ m_2^\pi(v) &= v(\{1, 2\}) - v(\{1\}) = 7 - 6 = 1 \\ m_3^\pi(v) &= v(N) - v(\{1, 2\}) = 10 - 7 = 3 \end{aligned}$$

Así, $m^\pi(v) = (6, 1, 3)$ se corresponde con la primera fila de la Tabla 1.2. El resto de vectores marginales se calculan de forma completamente análoga, cambiando únicamente el orden de llegada.

Orden π	Vector marginal $m^\pi(v)$
(1, 2, 3)	(6, 1, 3)
(1, 3, 2)	(6, 1, 3)
(2, 1, 3)	(3, 4, 3)
(2, 3, 1)	(2, 4, 4)
(3, 1, 2)	(2, 1, 7)
(3, 2, 1)	(2, 1, 7)

Tabla 1.2: Vectores marginales del juego del Ejemplo 1.2.1.

Promediando estos seis vectores marginales obtenemos el valor de Shapley:

$$Sh(N, v) = (3.5, 2, 4.5).$$

Fijémonos que los vectores marginales de la Tabla 1.2 coinciden exactamente con los cuatro vértices del núcleo calculados en el Ejemplo 1.2.2. Al tratarse de un juego cóncavo (Definición 1.2.4), los vectores marginales determinan los puntos extremos de la región estable como consecuencia del Teorema 1.2.1.

La segunda regla que veremos es el nucleolo que, en lugar de promediar contribuciones marginales, busca minimizar la mayor “queja” posible entre todas las coaliciones. Para cuantificar esta insatisfacción, introducimos el siguiente concepto:

Definición 1.2.7 (Exceso). Dado un juego de costes (N, v) y una imputación $x \in I(N, v)$, el *exceso* de una coalición $S \subset N$ respecto de x se define como

$$e(S, x) = x(S) - v(S),$$

y mide el grado de queja o insatisfacción de S (cuanto mayor es el pago que se le exige $x(S)$ respecto a su coste independiente $v(S)$, mayor es la queja).

Para encontrar un reparto estable, el nucleolo no intenta complacer a todos por igual, sino que se centra en mitigar las quejas más graves. Para ello, organizamos las quejas de todas las coaliciones posibles en un vector ordenado.

Definición 1.2.8 (Nucleolo). Sea $\theta(x)$ el vector que contiene los excesos $e(S, x)$ de todas las posibles coaliciones propias $S \subset N$ ordenados de mayor a menor, es decir, $\theta_1(x) \geq \theta_2(x) \geq \dots \geq \theta_k(x)$ con $k = 2^n - 2$. El *nucleolo* $Nu(N, v)$ es la única imputación que minimiza lexicográficamente³ este vector de excesos. Esto es:

$$Nu(N, v) = \{x \in I(N, v) : \theta(x) \preceq_L \theta(y), \quad \forall y \in I(N, v)\}.$$

Esta regla garantiza una imputación eficiente y, en juegos donde el núcleo no es vacío, el nucleolo siempre se encuentra dentro de él (Schmeidler 1969).

El siguiente ejemplo calcula el nucleolo del juego que hemos utilizado a lo largo de esta sección y permite compararlo directamente con el valor de Shapley obtenido en el Ejemplo 1.2.3.

Ejemplo 1.2.4 (Cálculo del nucleolo — continuación del Ejemplo 1.2.1). Para hallar el nucleolo debemos minimizar el máximo exceso posible. En este juego de tres agentes, las coaliciones de dos agentes ($|S| = 2$) son las que soportan la mayor carga de coste y, por tanto, las que potencialmente generarán los excesos más elevados. Aplicando la condición de eficiencia $x_1 + x_2 + x_3 = 10$ (es decir, $x_i + x_j = 10 - x_k$), podemos reescribir las expresiones de sus excesos en función de la coordenada del jugador restante:

$$\begin{aligned} e(\{1, 2\}, x) &= x_1 + x_2 - v(\{1, 2\}) = (10 - x_3) - 7 = 3 - x_3 \\ e(\{1, 3\}, x) &= x_1 + x_3 - v(\{1, 3\}) = (10 - x_2) - 9 = 1 - x_2 \\ e(\{2, 3\}, x) &= x_2 + x_3 - v(\{2, 3\}) = (10 - x_1) - 8 = 2 - x_1 \end{aligned}$$

El principio de minimización lexicográfica busca reducir al máximo la queja más grave. Dado que la eficiencia obliga a que cualquier mejora en la situación de una coalición perjudique inevitablemente a otra, el óptimo se alcanza cuando ya no es posible reducir la queja máxima sin aumentar la de las demás. Este equilibrio se logra cuando todas las coaliciones principales comparten el mismo nivel de insatisfacción. Por tanto, el nucleolo se determina igualando estos excesos a un valor común t . Es decir,

$$3 - x_3 = t, \quad 1 - x_2 = t, \quad 2 - x_1 = t.$$

Despejando cada variable en función de t resulta $x_1 = 2 - t$, $x_2 = 1 - t$ y $x_3 = 3 - t$. Sustituyendo estas expresiones en la ecuación de eficiencia se obtiene:

$$(2 - t) + (1 - t) + (3 - t) = 10 \implies 6 - 3t = 10 \implies t = -\frac{4}{3}.$$

Sustituyendo el valor de t , hallamos la única imputación que conforma el nucleolo:

$$Nu(N, v) = \left(\frac{10}{3}, \frac{7}{3}, \frac{13}{3} \right) \approx (3.33, 2.33, 4.33).$$

³El orden lexicográfico compara dos vectores de excesos $\theta(x)$ y $\theta(y)$ componente a componente. Decimos que $\theta(x) \preceq_L \theta(y)$ si en la primera posición j donde difieren, se cumple que $\theta_j(x) < \theta_j(y)$. El nucleolo busca el reparto que hace que la queja máxima (θ_1) sea la menor posible. En caso de haber varios, busca minimizar la segunda queja más grande (θ_2) y así sucesivamente hasta que la solución es única.

Finalmente, analizando los excesos individuales confirmamos que sus quejas son estrictamente menores que t ($e(\{1\}, x) \approx -2.67$, $e(\{2\}, x) \approx -1.67$, $e(\{3\}, x) \approx -2.67$), por lo que el vector obtenido efectivamente minimiza el máximo exceso global.

La comparación entre ambos conceptos es muy ilustrativa. Mientras que el valor de Shapley ($Sh(v) = (3.5, 2, 4.5)$) se basa en la contribución marginal de cada agente, el nucleolo ($Nu(v) = (3.33, 2.33, 4.33)$) busca el punto central del núcleo minimizando el mayor exceso o queja posible. Como muestra la Figura 1.4, aunque ambas soluciones se encuentran en el interior del núcleo, el nucleolo es el punto más equidistante a las restricciones del juego, garantizando así la máxima estabilidad.

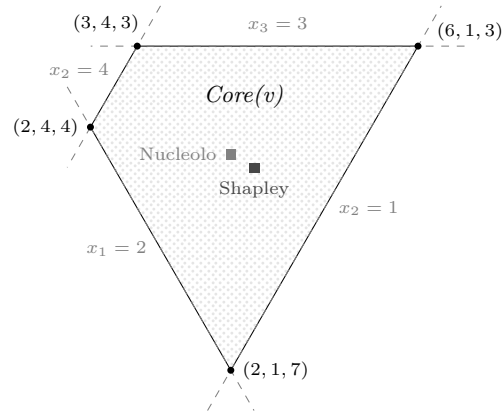


Figura 1.4: Comparativa entre el valor de Shapley y el nucleolo dentro del núcleo del juego.

Capítulo 2

El problema del árbol de mínimo coste (MCSTP)

Imaginemos una aldea en la que cada vivienda dispone de un pozo propio, pero ninguno de ellos garantiza un suministro de agua suficiente durante todo el año. El ayuntamiento decide entonces construir un único depósito común y cada vivienda puede conectarse a él mediante tuberías, ya sea de forma directa o a través de las tuberías de otras viviendas vecinas. La pregunta que se plantean los vecinos es: ¿qué tuberías conviene construir para que todas las viviendas queden conectadas al depósito gastando lo menos posible en total? (Bergantiños y Lorenzo 2004)

Esta situación, planteada originalmente como un problema real de abastecimiento de agua, es un caso particular de una familia de problemas mucho más amplia conocida como *problemas del árbol de mínimo coste* (MCSTP, por sus siglas en inglés: *Minimum Cost Spanning Tree Problems*). El problema no nació en el contexto actual, sino en el diseño de redes eléctricas. En 1926, el matemático checo Otakar Borůvka lo formuló y resolvió al estudiar cómo electrificar de forma eficiente la región de Moravia (Borůvka 1926)¹. Desde entonces, este mismo problema matemático ha reaparecido en contextos tan distintos como el diseño de redes de telecomunicaciones, el transporte o la distribución de agua.

En este capítulo nos centraremos en la parte estructural y de optimización del problema, es decir, cómo construir, de entre todas las redes posibles, aquella que conectará a todos los agentes con el proveedor de un cierto servicio al menor coste. Además, emplearemos los conceptos de teoría de grafos y juegos cooperativos introducidos en el Capítulo 1. En la Sección 2.1 presentaremos la definición formal del MCSTP y en la Sección 2.2, los algoritmos clásicos de Prim, Kruskal y Borůvka para resolverlo. A continuación, nos centraremos en el análisis de la estructura de costes mediante dos transformaciones útiles (Sección 2.3), así como en la técnica de descomposición en conos (Sección 2.4). Finalmente, en la Sección 2.5, definiremos los juegos cooperativos asociados al problema, estableciendo el marco teórico necesario para abordar, en el Capítulo 3, la segunda gran pregunta: cómo repartir el coste de la red óptima entre los agentes que se benefician de ella.

2.1. Definición formal del MCSTP

Recordemos que, en la Sección 1.1, definimos un grafo ponderado como un grafo $G = (V, A)$ junto con una función de peso que asigna un coste a cada arista. El MCSTP es, esencialmente, un grafo ponderado de este tipo, con dos particularidades: uno de sus nodos desempeña un papel especial (la

¹Dado que el texto original está en checo, se puede consultar la traducción al inglés de Nešetřil et al. (2001).

fuelle, a la que todos los demás deben conectarse) y el resto de nodos representan a los agentes del problema.

Definición 2.1.1 (Conjunto de agentes y matriz de costes). Sea $N = \{1, 2, \dots, n\}$ un conjunto finito de *agentes*, y sea 0 un nodo especial que representa la *fuelle* (el proveedor del servicio). Denotamos por $N_0 = N \cup \{0\}$ al conjunto total de nodos. El coste de conectar directamente cualquier par de nodos $i, j \in N_0$ viene dado por una *matriz de costes* $C = (c_{ij})_{i,j \in N_0}$ que satisface, para todo $i, j \in N_0$:

1. $c_{ij} \geq 0$ (no negatividad).
2. $c_{ij} = c_{ji}$ (simetría: los arcos son no dirigidos tal y como se expuso en la Sección 1.1).
3. $c_{ii} = 0$ (coste nulo de un nodo a sí mismo).

Así, un *problema de árbol de mínimo coste* (*mcstp*) queda completamente determinado por el par (N_0, C) (Bird 1976; Bergantiños y Vidal-Puga 2021).

Dado un *mcstp* (N_0, C) , podemos construir cualquier grafo (en el sentido de la Definición 1.1.3) sobre el conjunto de nodos N_0 . En este contexto lo llamaremos *red*, y denotaremos por g al conjunto de arcos que la componen. El coste de una red es, naturalmente, la suma de los costes de sus arcos:

$$c(g) = \sum_{(i,j) \in g} c_{ij}.$$

Sin embargo, no toda red nos interesa por igual. Solo aquellas que conecten a todos los agentes con la fuente resuelven el problema planteado.

Definición 2.1.2 (Red factible y árbol de mínimo coste). Una red g sobre N_0 es *factible* si todo agente $i \in N$ está conectado a la fuente 0, es decir, si existe un camino (Definición 1.1.5) entre i y 0 dentro de g . El problema de optimización del MCSTP consiste en hallar una red factible g^* de coste mínimo. Denotamos dicho coste mínimo como

$$m(N_0, C) = \min\{c(g) : g \text{ es una red factible sobre } N_0\}.$$

Ejemplo 2.1.1. Consideremos un problema de diseño de redes con un proveedor (fuente 0) y tres agentes ($N = \{1, 2, 3\}$). La matriz de costes asociada es

$$C = \begin{pmatrix} 0 & 10 & 6 & 6 \\ 10 & 0 & 5 & 3 \\ 6 & 5 & 0 & 3 \\ 6 & 3 & 3 & 0 \end{pmatrix}.$$

La Figura 2.1 muestra el grafo ponderado asociado a este problema.

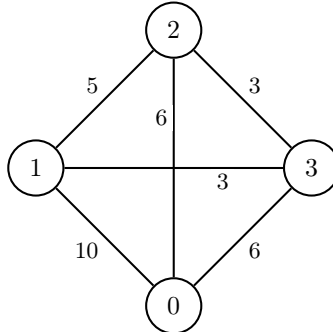


Figura 2.1: Grafo de costes asociado al problema.

Los agentes 1, 2 y 3 tienen que conectarse a la fuente 0 de la forma más barata posible. Fijémonos que las aristas más económicas de la red son $(1, 3)$ y $(2, 3)$, ambas con coste 3. Tras conectar a los tres agentes entre sí mediante estas dos aristas, basta con conectar el grupo a la fuente a través de un único nodo. En este caso, tanto el nodo 2 como el nodo 3 ofrecen la conexión más barata hacia la fuente, con coste 6. Tomando, por ejemplo, la red $g^* = \{(1, 3), (2, 3), (0, 3)\}$, obtenemos una solución factible de coste mínimo:

$$m(N_0, C) = 3 + 3 + 6 = 12.$$

Los algoritmos detallados en la Sección 2.2 formalizan y automatizan este proceso de búsqueda, que aquí hemos resuelto apoyándonos únicamente en la intuición.

Una propiedad fundamental de este tipo de problemas, es que la red de mínimo coste nunca necesita más conexiones que las estrictamente necesarias para mantener a todos los agentes conectados a la fuente. Es decir, la solución óptima siempre es un árbol generador (Definición 1.1.7). El siguiente ejemplo ilustra por qué esto es así:

Ejemplo 2.1.2. Tomemos como referencia el escenario del Ejemplo 2.1.1. Consideremos la red $g = \{(1, 3), (2, 3), (0, 2), (0, 3)\}$. Esta red conecta a los tres agentes con la fuente, pero no es una buena solución ya que su coste total es de 18.

El motivo de este sobrecoste es que la red contiene un ciclo, concretamente $(0, 2, 3, 0)$, formado por las aristas $(0, 2)$, $(2, 3)$ y $(0, 3)$. La presencia de este ciclo implica que existen dos caminos distintos entre la fuente y los nodos que lo conforman (2 y 3). Si tomamos, por ejemplo, el nodo 2, existe un camino directo, $(0, 2)$, y otro indirecto, $(0, 3, 2)$, lo que demuestra que una de las conexiones es redundante. Si eliminamos una de las dos aristas con mayor coste del ciclo, por ejemplo, $(0, 3)$ con coste 6, ningún agente se queda aislado. El agente 3, que antes usaba esa arista, ahora puede seguir llegando a la fuente de forma indirecta a través del nodo 2 utilizando el camino $(3, 2, 0)$. Al eliminar esa conexión, la nueva red $g' = g \setminus \{(0, 3)\} = \{(1, 3), (2, 3), (0, 2)\}$ sigue siendo válida y su coste total ha disminuido a $18 - 6 = 12$.

En resumen, si una red factible contiene un ciclo, estamos pagando por una conexión “duplicada”. Como siempre existe un camino alternativo dentro del propio ciclo, podemos eliminar la arista más cara sin desconectar a nadie. Dado que todos los costes del problema son no negativos (Definición 2.1.1), eliminar conexiones redundantes nunca aumentará el coste total. Por tanto, una red de coste mínimo nunca contendrá ciclos, lo que implica que ha de ser un árbol generador sobre N_0 .

A este árbol de coste mínimo (*mcst*, por sus siglas en inglés: *minimum cost spanning tree*), lo denotaremos en lo que sigue como t^* , siguiendo la notación de Bergantiños y Vidal-Puga (2021).

Cabe destacar que t^* no es necesariamente único. Si existen varias aristas con el mismo coste, pueden existir distintos árboles generadores, todos ellos de coste mínimo. De hecho, acabamos de ver que la red g' del Ejemplo 2.1.2 tiene exactamente el mismo coste que el árbol g^* propuesto originalmente en el Ejemplo 2.1.1. Esta falta de unicidad tendrá especial relevancia a continuación, al introducir los algoritmos de resolución y analizar cómo se gestionan los empates en los costes de las redes.

2.2. Algoritmos de construcción: Prim, Kruskal y Borůvka

Existen varios procedimientos para encontrar un árbol de coste mínimo. Los tres más estudiados en la literatura son los propuestos por Borůvka, Prim y Kruskal (Bergantiños y Vidal-Puga 2021). Estos algoritmos comparten una misma lógica: en cada paso seleccionan el arco más barato disponible. Aunque toman decisiones de forma inmediata sin analizar los pasos futuros, el árbol que se obtiene al terminar el proceso es siempre de coste mínimo. Lo que distingue a cada algoritmo es, precisamente, qué arcos considera “disponibles” en cada momento. A continuación presentamos formalmente cada uno y mostramos su ejecución paso a paso sobre la red del Ejemplo 2.1.1.

Algoritmo de Prim

El primer procedimiento que analizaremos en detalle es el algoritmo de Prim (Prim 1957). Su estrategia se basa en hacer crecer una única estructura conectada de manera progresiva a partir de la fuente. Así, partiendo inicialmente de un conjunto que solo contiene al nodo 0, en cada etapa el algoritmo evalúa todos los arcos disponibles y añade aquel que suponga el menor coste de conexión entre un nodo que ya está integrado en el árbol y otro que todavía no lo está. El proceso continúa de forma iterativa a lo largo de n etapas hasta conectar a todos los agentes del conjunto N . El árbol específico obtenido al finalizar la última etapa se denota como g^n y constituye, por definición, un árbol de coste mínimo (Bergantiños y Vidal-Puga 2021). El procedimiento se formaliza en el Algoritmo 1.

Algoritmo 1 Algoritmo de Prim

Entrada: Un $mcstp$ (N_0, C) , donde $N_0 = N \cup \{0\}$ es el conjunto de nodos y C es la matriz de costes

Salida: Un $mcst$ g^n

- 1: $S^0 \leftarrow \{0\}$ ▷ Nodos conectados (inicialmente solo la fuente)
- 2: $g^0 \leftarrow \emptyset$ ▷ Arcos del árbol inicial
- 3: $p \leftarrow 0$ ▷ Contador de etapas
- 4: **mientras** $|S^p| < |N_0|$ **hacer**
- 5: Seleccionar un arco (j, i) con $j \in S^p$ e $i \in N_0 \setminus S^p$ tal que:

$$c_{ji} = \min_{k \in S^p, l \in N_0 \setminus S^p} \{c_{kl}\}$$

▷ En caso de empate, se selecciona cualquiera de ellos

- 6: $S^{p+1} \leftarrow S^p \cup \{i\}$
 - 7: $g^{p+1} \leftarrow g^p \cup \{(j, i)\}$
 - 8: $p \leftarrow p + 1$
 - 9: **fin mientras**
 - 10: **devolver** g^n
-

Ejemplo 2.2.1 (Ejecución de Prim sobre el Ejemplo 2.1.1). Partimos de la configuración inicial $S^0 = \{0\}$ y $g^0 = \emptyset$. El algoritmo se ejecuta en tres etapas, ilustradas paso a paso en la Figura 2.2:

- Etapas 1** (a) Evaluamos los arcos que conectan $S^0 = \{0\}$ con el resto de nodos ($N \setminus S^0$). Disponemos de $(0, 1)$ con coste 10 y $(0, 2)$ y $(0, 3)$, ambos con coste 6. Al haber un empate, seleccionamos arbitrariamente el arco $(0, 2)$ y actualizamos $S^1 = \{0, 2\}$ y $g^1 = \{(0, 2)\}$.
- Etapas 2** (b) Buscamos el arco más barato desde $S^1 = \{0, 2\}$ hacia $N \setminus S^1 = \{1, 3\}$. Las opciones son $(0, 1)$ con coste 10, $(0, 3)$ con coste 6, $(1, 2)$ con coste 5 y $(2, 3)$ con coste 3. El arco de menor coste es $(2, 3)$, lo seleccionamos y actualizamos $S^2 = \{0, 2, 3\}$ y $g^2 = \{(0, 2), (2, 3)\}$.
- Etapas 3** (c) Buscamos el arco más barato desde $S^2 = \{0, 2, 3\}$ hacia el nodo restante $\{1\}$. Las opciones son $(0, 1)$ con coste 10, $(1, 2)$ con coste 5 y $(1, 3)$ con coste 3. El arco de menor coste es $(1, 3)$, por lo que actualizamos $S^3 = N_0$ y $g^3 = \{(0, 2), (2, 3), (1, 3)\}$ y el algoritmo finaliza.

El árbol resultante es $g^3 = \{(0, 2), (2, 3), (1, 3)\}$, con un coste total de $m(N_0, C) = 6 + 3 + 3 = 12$. Como vemos, al resolver el empate inicial de esta manera, el algoritmo construye exactamente la misma red

alternativa que ya identificamos en el Ejemplo 2.1.2, confirmando en la práctica que el árbol de coste mínimo no tiene por qué ser único.

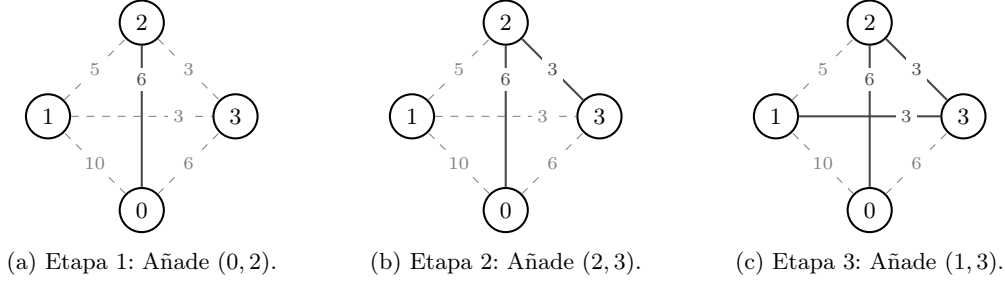


Figura 2.2: Construcción del *mcst* paso a paso mediante el algoritmo de Prim.

Algoritmo de Kruskal

El segundo procedimiento clásico que analizaremos es el algoritmo de Kruskal (Kruskal 1956). A diferencia de Prim, este método no mantiene un único árbol conectado durante el proceso de construcción, sino que ordena todos los arcos de menor a mayor coste para evaluarlos secuencialmente. Partiendo de un conjunto de arcos vacío, en cada etapa se selecciona el enlace más barato disponible y se incorpora a la solución siempre y cuando no genere ciclos con los arcos previamente incluidos. Este proceso iterativo continúa hasta que se han incorporado exactamente $|N|$ arcos, momento en el que se garantiza la formación de un árbol de coste mínimo sobre el conjunto N_0 (Bergantiños y Vidal-Puga 2021). El procedimiento se formaliza en el Algoritmo 2.

Algoritmo 2 Algoritmo de Kruskal

Entrada: Un *mcstp* (N_0, C) , donde $N_0 = N \cup \{0\}$ es el conjunto de nodos y C es la matriz de costes

Salida: Un *mcst* $g^{|N|}$

- 1: $A^0 \leftarrow \{(i, j) : i, j \in N_0, i \neq j\}$ ▷ Conjunto de todos los arcos posibles
- 2: $g^0 \leftarrow \emptyset$ ▷ Arcos del árbol inicial
- 3: $p \leftarrow 0$ ▷ Contador de etapas
- 4: **mientras** $|g^p| < |N|$ **hacer**
- 5: Seleccionar un arco $(i, j) \in A^p$ tal que:

$$c_{ij} = \min_{(k,l) \in A^p} \{c_{kl}\}$$

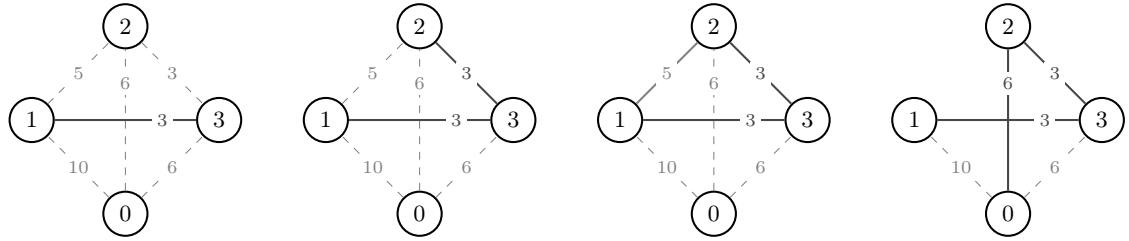
▷ En caso de empate, se selecciona cualquiera de ellos

- 6: **si** $g^p \cup \{(i, j)\}$ contiene un ciclo **entonces**
 - 7: $A^p \leftarrow A^p \setminus \{(i, j)\}$ ▷ Se descarta el arco y se repite la etapa
 - 8: **si no**
 - 9: $A^{p+1} \leftarrow A^p \setminus \{(i, j)\}$
 - 10: $g^{p+1} \leftarrow g^p \cup \{(i, j)\}$ ▷ Se añade el arco al árbol
 - 11: $p \leftarrow p + 1$
 - 12: **fin si**
 - 13: **fin mientras**
 - 14: **devolver** $g^{|N|}$
-

Ejemplo 2.2.2 (Ejecución de Kruskal sobre el Ejemplo 2.1.1). Para aplicar el algoritmo, conceptualmente ordenamos los arcos de la red de menor a mayor coste: $(1, 3)$ y $(2, 3)$, ambos con coste 3, $(1, 2)$ con coste 5, $(0, 2)$ y $(0, 3)$, ambos con coste 6 y $(0, 1)$ con coste 10. Partimos de la configuración inicial $A^0 = \{(i, j) : i, j \in N_0, i \neq j\}$ y $g^0 = \emptyset$. El algoritmo se ejecuta en tres etapas, ilustradas paso a paso en la Figura 2.3:

- Etapla 1** (a) Buscamos el arco de menor coste en A^0 . Existe un empate entre $(1, 3)$ y $(2, 3)$, ambos con coste 3. Seleccionamos arbitrariamente $(1, 3)$. Al no formar ciclos (el árbol actual está vacío), lo añadimos y actualizamos $A^1 = A^0 \setminus \{(1, 3)\}$ y $g^1 = \{(1, 3)\}$.
- Etapla 2** (b) El siguiente arco más barato disponible en A^1 es $(2, 3)$ con coste 3. Dado que no puede formar un ciclo puesto que solo hay un arco en el árbol, lo incorporamos y actualizamos $A^2 = A^1 \setminus \{(2, 3)\}$ y $g^2 = \{(1, 3), (2, 3)\}$.
- Etapla 3** (c, d) Evaluamos el siguiente arco de la lista en A^2 , que es $(1, 2)$ con coste 5. Al intentar añadirlo a g^2 , se formaría el ciclo $(1, 3, 2, 1)$. Por tanto, se descarta actualizando $A^2 = A^2 \setminus \{(1, 2)\}$ y manteniendo g^2 igual.

Buscamos el siguiente arco más barato en el conjunto A^2 actualizado. Los candidatos son $(0, 2)$ y $(0, 3)$, ambos con coste 6. Al haber un empate, escogemos $(0, 2)$ de forma arbitraria. Su inclusión no forma ciclo, por lo que actualizamos $A^3 = A^2 \setminus \{(0, 2)\}$ y $g^3 = \{(1, 3), (2, 3), (0, 2)\}$ y el algoritmo finaliza.



(a) Etapa 1: Añade $(1, 3)$. (b) Etapa 2: Añade $(2, 3)$. (c) Etapa 3: Evalúa $(1, 2)$. ¡Ciclo! (d) Etapa 3: Añade $(0, 2)$.

Figura 2.3: Construcción del *mcst* paso a paso mediante Kruskal (en rojo, la arista descartada por formar ciclo).

El árbol resultante es $g^3 = \{(1, 3), (2, 3), (0, 2)\}$, con un coste total de $m(N_0, C) = 3 + 3 + 6 = 12$. Nótese que este es exactamente el mismo árbol que se obtuvo al ejecutar el algoritmo de Prim en el Ejemplo 2.2.1, aunque esto no tiene por qué ocurrir siempre.

Algoritmo de Borůvka

El algoritmo de Borůvka (Borůvka 1926; Nešetřil et al. 2001), precursor histórico de los dos anteriores, sigue un enfoque paralelo. Empieza considerando cada nodo como una componente conexa independiente y, en cada paso, todas las componentes buscan simultáneamente su conexión externa más barata, en lugar de avanzar una sola conexión a la vez como hacían Prim y Kruskal. Para evitar ambigüedades ante empates, se fija de antemano un orden estricto π sobre el conjunto de arcos (Bergantiños y Vidal-Puga 2021). El árbol específico obtenido mediante este procedimiento se denota como t^π y constituye, por definición, un árbol de coste mínimo. El procedimiento se formaliza en el Algoritmo 3.

Algoritmo 3 Algoritmo de Borůvka

Entrada: Un $mcstp$ (N_0, C) , donde $N_0 = N \cup \{0\}$ es el conjunto de nodos y C es la matriz de costes y un orden estricto π sobre los arcos

Salida: Un $mcst$ $t^\pi = g^{\pi, s-1}$

```

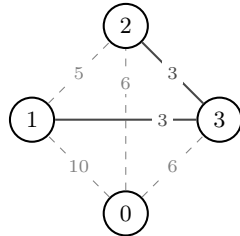
1:  $g^{\pi, 0} \leftarrow \emptyset$  ▷ Arcos del árbol inicial
2:  $s \leftarrow 1$  ▷ Contador de etapas
3:  $P(g^{\pi, 0}) = \{\{0\}, \{1\}, \dots, \{|N|\}\}$  ▷ Partición inicial en componentes conexas
4: mientras  $P(g^{\pi, s-1}) \neq \{N_0\}$  hacer
5:    $E_s \leftarrow \emptyset$  ▷ Arcos a añadir en la etapa actual
6:   para cada componente conexa  $T \in P(g^{\pi, s-1})$  tal que  $0 \notin T$  hacer
7:     Seleccionar el arco  $(i^T, j^T)$  de coste mínimo tal que  $i^T \in T$  y  $j^T \in N_0 \setminus T$ 
▷ En caso de empate, se selecciona el de menor posición en  $\pi$ 
8:      $E_s \leftarrow E_s \cup \{(i^T, j^T)\}$ 
9:   fin para
10:   $g^{\pi, s} \leftarrow g^{\pi, s-1} \cup E_s$ 
11:   $P(g^{\pi, s}) \leftarrow \text{merge}(P(g^{\pi, s-1}), E_s)$  ▷ Actualización de componentes conexas
12:   $s \leftarrow s + 1$ 
13: fin mientras
14: devolver  $g^{\pi, s-1}$ 

```

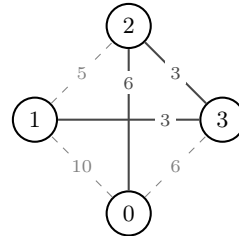
Ejemplo 2.2.3 (Ejecución de Borůvka sobre el Ejemplo 2.1.1). Fijamos un orden estricto π tal que $(1, 3) \succ (2, 3) \succ (1, 2) \succ (0, 2) \succ (0, 3) \succ (0, 1)$. Partimos de la configuración inicial $g^{\pi, 0} = \emptyset$ y $P(g^{\pi, 0}) = \{\{0\}, \{1\}, \{2\}, \{3\}\}$. El algoritmo procede como se detalla a continuación y se ilustra en la Figura 2.4:

Etapas 1 (a) Las componentes $T_2 = \{1\}$, $T_3 = \{2\}$ y $T_4 = \{3\}$ buscan simultáneamente su arco de menor coste hacia $N_0 \setminus T_k$ con $k = \{2, 3, 4\}$. La componente T_2 elige $(1, 3)$ con coste 3, la componente T_3 elige $(2, 3)$, también con coste 3 y en la componente T_4 hay un empate entre $(1, 3)$ y $(2, 3)$. Dado que $(1, 3)$ precede a $(2, 3)$ en π , seleccionamos $(1, 3)$. El conjunto de arcos seleccionados es $E_1 = \{(1, 3), (2, 3)\}$. Actualizando, $g^{\pi, 1} = E_1$ y las componentes se fusionan en $P(g^{\pi, 1}) = \{\{0\}, \{1, 2, 3\}\}$.

Etapas 2 (b) La única componente sin fuente es $T_2 = \{1, 2, 3\}$. Sus conexiones externas son $(0, 1)$ con coste 10 y $(0, 2)$ y $(0, 3)$, ambas con coste 6. Ante el empate, el orden π prioriza $(0, 2)$. Se añade $E_2 = \{(0, 2)\}$, actualizando $g^{\pi, 2} = g^{\pi, 1} \cup E_2 = \{(1, 3), (2, 3), (0, 2)\}$ y $P(g^{\pi, 2}) = \{N_0\}$, por lo que el algoritmo finaliza.



(a) Etapa 1: Fusión de componentes $\{1, 2, 3\}$.



(b) Etapa 2: Conexión a la fuente.

Figura 2.4: Construcción del $mcst$ paso a paso mediante el algoritmo de Borůvka.

El árbol resultante es $t^\pi = g^{\pi,2} = \{(1,3), (2,3), (0,2)\}$, con un coste total de $m(N_0, C) = 3+3+6 = 12$. Como vemos, este resultado coincide con los obtenidos mediante los algoritmos de Prim y Kruskal en los ejemplos anteriores. No obstante, es importante recordar nuevamente que el árbol de coste mínimo no tiene por qué ser único y que, en Borůvka, la solución obtenida está condicionada por el orden π establecido.

En resumen, los tres algoritmos resuelven el mismo problema de optimización, aunque mediante estrategias distintas. Prim hace crecer un único árbol desde la fuente, Kruskal trabaja de forma global ordenando todos los arcos de antemano, y Borůvka avanza en paralelo sobre todas las componentes a la vez. En el Capítulo 3 veremos que cada algoritmo da lugar, de forma natural, a una familia distinta de reglas de reparto del coste.

2.3. La forma irreducible y la red ciclo-completa

En muchas ocasiones la matriz de costes C de un *mcstp* suele contener conexiones directas ineficientes. Esto ocurre cuando el coste de conectar directamente a dos agentes es superior al de conectarlos mediante una ruta alternativa a través de otros nodos. Un árbol de coste mínimo evitará siempre estas aristas porque la estructura óptima optará sistemáticamente por la vía indirecta más barata. En esta sección introducimos la forma irreducible y la red ciclo-completa. Ambas transformaciones permiten refinar la matriz inicial reemplazando los valores originales por los costes mínimos reales de conexión.

La forma irreducible

La primera transformación que veremos es la forma irreducible. Esta matriz utiliza un árbol de coste mínimo para redefinir la red y establece que el coste entre dos nodos cualesquiera es igual al coste del arco más caro en el único camino que los une dentro del árbol óptimo.

Definición 2.3.1 (Forma irreducible). Dado un *mcstp* (N_0, C) y un árbol de coste mínimo t^* , la *forma irreducible* $(N_0, \bar{C})$ se obtiene asignando a cada par de nodos i y j el coste de la arista más cara en el único camino g_{ij}^* que los une dentro de t^* :

$$\bar{c}_{ij} = \max_{(k,l) \in g_{ij}^*} \{c_{kl}\}.$$

Esta matriz es única y no depende del árbol óptimo t^* seleccionado cuando existen múltiples soluciones, ya que el coste de la arista más cara entre dos nodos es idéntico en cualquier *mcst* (Bird 1976; Bergantiños y Vidal-Puga 2021).

La idea intuitiva es que el agente i siempre puede llegar hasta j empleando el árbol óptimo. Lo máximo que tendrá que pagar en ese trayecto será, precisamente, $\bar{c}_{ij}$. Por tanto, ningún agente debería tener que pagar más que ese valor por conectarse a j , incluso si la arista directa (i, j) es más cara.

Ejemplo 2.3.1 (Construcción de $\bar{C}$ sobre el Ejemplo 2.1.1). Retomemos el árbol óptimo obtenido en la Sección 2.2 dado por $t^* = \{(0,2), (1,3), (2,3)\}$ con un coste de 6 para el primer arco y de 3 para los dos restantes. Para cada par de nodos sin conexión directa en t^* determinamos su camino único dentro del árbol óptimo.

- Par {0,1}** El camino en el árbol es $(0,2,3,1)$, cuyos costes asociados son $c_{02} = 6$ y $c_{13} = c_{23} = 3$. El valor máximo de estos arcos es 6, luego $\bar{c}_{01} = 6$.
- Par {0,3}** El camino en el árbol es $(0,2,3)$, cuyos costes asociados son $c_{02} = 6$ y $c_{23} = 3$. El valor máximo de estos arcos es 6, por lo que $\bar{c}_{03} = 6$.
- Par {1,2}** El camino en el árbol es $(1,3,2)$, cuyos costes asociados son $c_{13} = 3$ y $c_{23} = 3$. El valor máximo es 3, de modo que $\bar{c}_{12} = 3$.

Los tres arcos que ya pertenecen a t^* mantienen su coste original porque el camino entre sus extremos es la propia arista, es decir, $\bar{c}_{02} = 6$ y $\bar{c}_{13} = \bar{c}_{23} = 3$. La forma irreducible completa se presenta a continuación:

$$\bar{C} = \begin{pmatrix} 0 & 6 & 6 & 6 \\ 6 & 0 & 3 & 3 \\ 6 & 3 & 0 & 3 \\ 6 & 3 & 3 & 0 \end{pmatrix}.$$

Fijémonos que las conexiones ineficientes de la matriz original ($c_{01} = 10$ y $c_{12} = 5$) han sido sustituidas por el coste real de su conexión indirecta (6 y 3, respectivamente). El grafo asociado se muestra en la Figura 2.5, resaltando con línea continua las aristas que forman el árbol óptimo t^* .

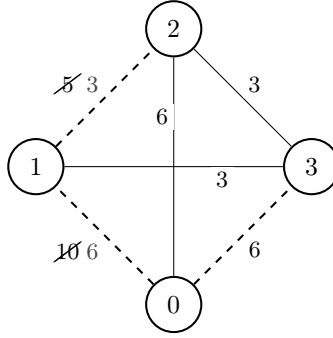


Figura 2.5: Grafo irreducible $\bar{C}$ asociado al Ejemplo 2.1.1 con el árbol óptimo t^* representado en línea continua.

La red ciclo-completa

La forma irreducible reduce el coste de una arista tan pronto como existe un camino alternativo más barato. Trudeau (2012) propone una transformación más conservadora que solo disminuye el valor de conexión cuando existen dos rutas distintas entre ambos nodos. Esto implica que los agentes forman parte de un ciclo y pueden comunicarse incluso si se bloquea uno de los trayectos. De este modo, la transformación altera menos costes y protege la estructura original de la red.

Definición 2.3.2 (Red ciclo-completa). Dado un *mcstp* (N_0, C) , sea $\mathcal{F}(i, j)$ el conjunto de ciclos (Definición 1.1.5) que contienen a los nodos i y j . La *red ciclo-completa* (N_0, C^*) introducida por Trudeau (2012) se define mediante la siguiente expresión:

$$c_{ij}^* = \min \left(c_{ij}, \min_{f \in \mathcal{F}(i, j)} \max_{e \in f} c_e \right).$$

El coste c_{ij}^* representa el menor valor entre el coste directo original y el coste de la arista más cara del mejor ciclo que conecta a ambos agentes. Si ningún ciclo ofrece una alternativa más barata que la conexión directa, el coste se mantiene sin reducciones.

Ejemplo 2.3.2 (Construcción de C^* sobre el Ejemplo 2.1.1). A diferencia del cálculo de $\bar{C}$, aquí hay que examinar directamente los ciclos del grafo original (Figura 2.1). Retomemos también el árbol óptimo $t^* = \{(0, 2), (1, 3), (2, 3)\}$. Al tratarse de un grafo completo sobre cuatro nodos, existen varias alternativas para evaluar cada par.

Par {0,1} El ciclo de cuatro nodos que no usa la arista directa es $(0, 2, 1, 3, 0)$, cuyos costes asociados son $c_{02} = c_{03} = 6$, $c_{12} = 5$ y $c_{13} = 3$. El valor máximo de estos arcos es 6. Como este valor es menor que el coste original 10, se produce una reducción, luego $c_{01}^* = \min(10, 6) = 6$.

Par {0,2} El mejor ciclo alternativo es $(0, 1, 2, 3, 0)$, cuyos costes asociados son $c_{01} = 10$, $c_{03} = 6$, $c_{12} = 5$ y $c_{23} = 3$. El valor máximo de estos arcos es 10, el cual supera al enlace directo. Por tanto, no hay reducción, de modo que $c_{02}^* = \min(6, 10) = 6$.

Par {1,2} El único ciclo de cuatro nodos que conecta a ambos extremos sin usar su arista directa es $(1, 0, 2, 3, 1)$, cuyos costes asociados son $c_{01} = 10$, $c_{02} = 6$ y $c_{13} = c_{23} = 3$. El valor máximo de estos arcos es 10. Al ser superior al valor directo de 5, el coste se mantiene, luego $c_{12}^* = \min(5, 10) = 5$.

Por su parte, los tres arcos que pertenecen a t^* ya poseen costes muy bajos y puede comprobarse de la misma manera que ningún ciclo ofrece una alternativa mejor. Por este motivo, conservan su coste original, es decir, $c_{03}^* = 6$, $c_{13}^* = 3$ y $c_{23}^* = 3$. La red ciclo-completa resultante se presenta a continuación:

$$C^* = \begin{pmatrix} 0 & 6 & 6 & 6 \\ 6 & 0 & 5 & 3 \\ 6 & 5 & 0 & 3 \\ 6 & 3 & 3 & 0 \end{pmatrix}.$$

Fijémonos que, a diferencia de lo ocurrido en la forma irreducible, únicamente la conexión ineficiente $c_{01} = 10$ ha sido sustituida por el coste del ciclo (6). La conexión $c_{12} = 5$ mantiene su valor original. El grafo asociado se muestra en la Figura 2.6, resaltando con línea continua las aristas que forman el árbol óptimo t^* .

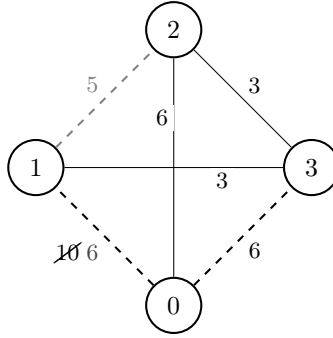


Figura 2.6: Grafo ciclo-completo C^* asociado al Ejemplo 2.1.1 con el árbol óptimo t^* representado en línea continua. En rojo, la única arista que no coincide con la forma irreducible.

Comparando ambas matrices, vemos que $\bar{C}$ y C^* coinciden en todas las entradas salvo en una: $\bar{c}_{12} = 3$ frente a $c_{12}^* = 5$. La diferencia surge porque la forma irreducible reduce el coste de $(1, 2)$ basándose únicamente en el camino $(1, 3, 2)$, donde el arco más caro cuesta 3. En cambio, la red ciclo-completa exige que esa reducción ocurra a través de un ciclo que no utilice la arista directa. Al evaluar el único ciclo de cuatro nodos que conecta 1 y 2, este incluye la arista $(0, 1)$ que cuesta 10, lo que impide abaratar el coste original de 5. En resumen, la forma irreducible acepta cualquier camino alternativo más barato, mientras que la red ciclo-completa es más exigente y solo reduce el precio si existe un ciclo completo que garantice esa mejora.

Observación 2.3.1 (Relación entre C , $\bar{C}$ y C^*). El Ejemplo 2.3.2 ilustra una propiedad general demostrada por Trudeau (2012). Para cualquier $mcstp$ y cualquier par de nodos i, j , se cumple que

$$\bar{c}_{ij} \leq c_{ij}^* \leq c_{ij}.$$

Esto significa que la forma irreducible nunca es más cara que la red ciclo-completa y esta última nunca supera el coste de la matriz original. Aunque las tres matrices pueden coincidir en casos particulares,

en términos generales se espera que la relación $\bar{C} \leq C^* \leq C$ sea estricta en al menos alguna de sus entradas, tal y como hemos comprobado en nuestro ejemplo.

2.4. Descomposición en conos

Hasta ahora hemos tratado la matriz de costes C como un objeto único. Sin embargo, resulta útil descomponerla en piezas más simples: matrices binarias donde cada conexión vale únicamente 0 o 1. Bergantiños y Vidal-Puga (2021) atribuyen este resultado a Norde et al. (2004). Esta técnica permite reconstruir cualquier matriz de costes original como una suma ponderada de estas matrices binarias, las cuales emplearemos para definir reglas de reparto en el Capítulo 3. En esta sección, no obstante, nos centramos exclusivamente en el proceso de descomposición.

Definición 2.4.1 (Matriz de costes elemental). Una matriz de costes C^q sobre N_0 es *elemental* si $c_{ij}^q \in \{0, 1\}$ para todo $i, j \in N_0$. Equivalentemente, una matriz elemental queda determinada por una red g^q sobre N_0 , de modo que $c_{ij}^q = 1$ si $(i, j) \in g^q$, y $c_{ij}^q = 0$ en caso contrario.

El resultado de Norde et al. (2004), revisado por Bergantiños y Vidal-Puga (2021), garantiza que cualquier matriz de costes, sin importar lo compleja que sea, puede escribirse como una combinación de matrices elementales.

Teorema 2.4.1 (Descomposición en matrices elementales). *Para cualquier mcstp (N_0, C) , existen una familia de matrices elementales $\{C^q\}_{q=1}^{m(C)}$ y una familia de escalares no negativos $\{x^q\}_{q=1}^{m(C)}$ tales que:*

1. $C = \sum_{q=1}^{m(C)} x^q C^q$.
2. Cada C^q es una matriz elemental, en el sentido de la Definición 2.4.1.
3. Las matrices elementales respetan el orden de la matriz original: si $c_{ij} \leq c_{kl}$, entonces $c_{ij}^q \leq c_{kl}^q$ para todo q .

El Teorema 2.4.1 se demuestra mediante un procedimiento constructivo sencillo. Sean $0 < t_1 < t_2 < \dots < t_{m(C)}$ los costes que aparecen en la matriz original C (excluyendo el 0), donde $m(C)$ es el número total de valores distintos y sea $t_0 = 0$. Podemos descomponer C siguiendo estos pasos:

1. Para cada umbral t_q con $q = 1, 2, \dots, m(C)$, definimos una matriz elemental C^q indicando con un 1 aquellas conexiones cuyo coste es al menos t_q :

$$c_{ij}^q = \begin{cases} 1 & \text{si } c_{ij} \geq t_q \\ 0 & \text{si } c_{ij} < t_q. \end{cases}$$

2. Asignamos a cada matriz elemental el peso $x^q = t_q - t_{q-1}$.

De esta forma, la matriz original se recupera como la suma ponderada $C = \sum_{q=1}^{m(C)} x^q C^q$. Esta construcción garantiza las tres condiciones del Teorema 2.4.1: cada C^q es binaria por definición y, debido a que el peso x^q representa el incremento entre niveles de coste consecutivos, la suma ponderada acumula estos saltos hasta alcanzar el valor exacto de cada entrada c_{ij} . Finalmente, la estructura respeta el orden original. Si $c_{ij} \leq c_{kl}$, cualquier matriz C^q que tenga un 1 en la posición (i, j) tendrá necesariamente tendrá otro 1 en la posición (k, l) , puesto que $c_{kl} \geq c_{ij} \geq t_q$, lo que asegura que $c_{ij}^q \leq c_{kl}^q$ para todo q .

Ejemplo 2.4.1 (Descomposición de la matriz C del Ejemplo 2.1.1). Retomemos la matriz de costes de la Figura 2.1, en la que $c_{01} = 10$, $c_{02} = c_{03} = 6$, $c_{12} = 5$ y $c_{13} = c_{23} = 3$. Como hay $m(C) = 4$ costes distintos $t_1 = 3$, $t_2 = 5$, $t_3 = 6$ y $t_4 = 10$, necesitaremos cuatro matrices elementales. Tanto la descomposición como la evolución de las aristas activas (con valor 1) se ilustra en la Figura 2.7:

- Capa 1 (a)** Establecemos el umbral $t_1 = 3$, lo que otorga a esta primera matriz un peso $x^1 = 3$. Todas las aristas superan este coste, luego C^1 activa las seis conexiones del grafo.
- Capa 2 (b)** El umbral asciende a $t_2 = 5$, asignando a esta segunda capa el peso $x^2 = 5 - 3 = 2$. Solo las aristas con coste superior o igual a 5 siguen activas en C^2 : $(0, 1)$, $(0, 2)$, $(0, 3)$ y $(1, 2)$.
- Capa 3 (c)** El umbral se actualiza a $t_3 = 6$, correspondiéndole a la matriz un peso $x^3 = 6 - 5 = 1$. La arista $(1, 2)$ deja de estar activa, por lo que ahora solo $(0, 1)$, $(0, 2)$ y $(0, 3)$ permanecen con valor 1 en C^3 .
- Capa 4 (d)** Finalmente, se alcanza el umbral $t_4 = 10$, dotando a la última capa del peso $x^4 = 10 - 6 = 4$. Únicamente la arista más cara, $(0, 1)$, sigue activa en C^4 .

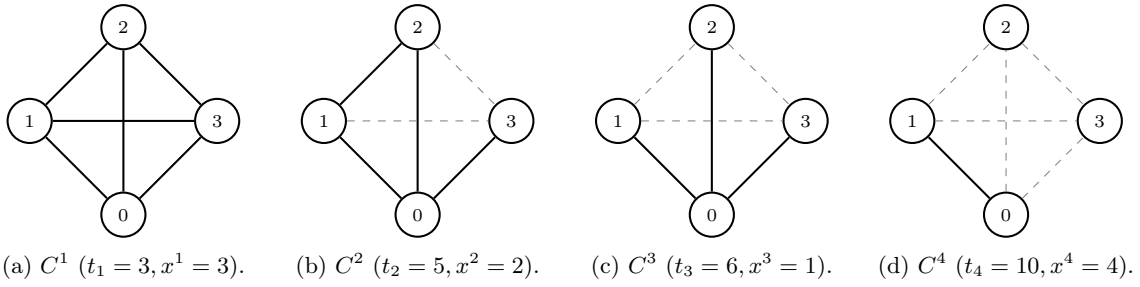


Figura 2.7: Representación de las cuatro capas de la descomposición de C . En cada capa C^q , las aristas continuas toman el valor 1 y las discontinuas, 0.

Comprobemos que la suma ponderada $\sum_{q=1}^4 x^q C^q$ reconstruye la matriz original. Por ejemplo, para los arcos $(0, 1)$ y $(1, 2)$ se tiene que

$$c_{01} = 3 \cdot 1 + 2 \cdot 1 + 1 \cdot 1 + 4 \cdot 1 = 10, \quad c_{12} = 3 \cdot 1 + 2 \cdot 1 + 1 \cdot 0 + 4 \cdot 0 = 5.$$

En efecto, estos valores coinciden con los costes originales $c_{01} = 10$ y $c_{12} = 5$. El lector puede verificar el resto de aristas siguiendo este mismo procedimiento.

2.5. Juegos cooperativos asociados al MCSTP

Recordemos la Definición 1.2.1: un juego cooperativo de costes es un par (N, v) donde $v(S)$ representa el coste que una coalición S asumiría si decidiera actuar de forma independiente. Para traducir un *mcstp* (N_0, C) a este lenguaje, basta con precisar qué significa exactamente “actuar de forma independiente” para un grupo de agentes que necesita conectarse a la fuente.

Dependiendo de qué recursos se permita utilizar a la coalición S , surgen distintas funciones características. Bergantiños y Vidal-Puga (2021) identifican cinco propuestas principales en la literatura, que detallamos a continuación. Para facilitar su comparación, ilustraremos cada una de ellas calculando el coste de la coalición $S = \{1, 2\}$ sobre la red del Ejemplo 2.1.1. Además, escribiremos la función característica simplemente como v en lugar de v_C puesto que la matriz de costes se sobreentiende a lo largo del capítulo y no da lugar a confusión.

El juego privado (o pesimista)

La primera propuesta estudiada en la literatura plantea la alternativa más restrictiva. Consiste en suponer que los miembros de una coalición solo pueden utilizar sus propios nodos para conectarse a

la fuente. En este enfoque, propuesto originalmente por Bird (1976), los nodos externos no cooperan en absoluto y se consideran estrictamente inaccesibles, por lo que S ni siquiera puede utilizarlos como puntos de paso para abaratar su conexión a la fuente. Por este motivo, a esta formulación se le conoce habitualmente como el juego pesimista.

Definición 2.5.1 (Juego privado). Dado un *mcstp* (N_0, C) , el *juego privado* (N, v^p) se define, para cada coalición $S \subseteq N$, como el coste mínimo de conectar a S con la fuente usando exclusivamente los nodos de $S_0 = S \cup \{0\}$:

$$v^p(S) = m(S_0, C).$$

Ejemplo 2.5.1 (Juego privado para $S = \{1, 2\}$). El conjunto $S_0 = \{0, 1, 2\}$ solo dispone de los arcos $(0, 1)$ con coste 10, $(0, 2)$ con coste 6 y $(1, 2)$ con coste 5. El árbol más barato usa las aristas $(0, 2)$ y $(1, 2)$. El coste total viene dado por

$$v^p(\{1, 2\}) = 6 + 5 = 11.$$

El juego irreducible

Una alternativa consiste en aplicar esa misma lógica pesimista sobre la matriz irreducible $\bar{C}$ (Definición 2.3.1) en lugar de la original. Este enfoque, propuesto también por Bird (1976), resulta en un juego donde se sigue prohibiendo el uso de nodos externos, pero los costes ya recogen de antemano las conexiones indirectas más baratas.

Definición 2.5.2 (Juego irreducible). Dado un *mcstp* (N_0, C) , el *juego irreducible* (N, v^i) se define como el juego privado asociado a la forma irreducible $\bar{C}$. Para cada coalición $S \subseteq N$, el coste viene dado por:

$$v^i(S) = m(S_0, \bar{C}).$$

Ejemplo 2.5.2 (Juego irreducible para $S = \{1, 2\}$). Sobre la matriz $\bar{C}$ disponemos de $\bar{c}_{01} = \bar{c}_{02} = 6$ y $\bar{c}_{12} = 3$. El árbol más barato para conectar S utiliza, por ejemplo, las aristas $(0, 1)$ y $(1, 2)$ (o, de forma equivalente, $(0, 2)$ y $(1, 2)$), de modo que

$$v^i(\{1, 2\}) = 6 + 3 = 9.$$

Nótese que $v^i(\{1, 2\}) = 9$ es estrictamente menor que $v^p(\{1, 2\}) = 11$. Esto sucede porque, al suavizar la matriz de costes, conectar a la misma coalición resulta más barato.

El juego optimista

En el extremo opuesto al juego privado surge este nuevo enfoque, introducido por Bergantiños y Vidal-Puga (2007). Este supone que los agentes fuera de S ya están conectados a la fuente de antemano y permiten a la coalición utilizar sus nodos como simples intermediarios, sin coste alguno por el privilegio de pasar por ellos.

Definición 2.5.3 (Juego optimista). Dado un *mcstp* (N_0, C) y una coalición $S \subseteq N$, sea $T = N \setminus S$. Definimos la matriz C^T como $c_{ij}^T = c_{ij}$ para todo $i, j \in S$, y $c_{0i}^T = \min_{j \in T_0} c_{ji}$ para todo $i \in S$, es decir, el coste de conectar i con la fuente pasa a ser el de su conexión más barata disponible a través de algún agente de T (o de la fuente). El *juego optimista* (N, v^o) se define como

$$v^o(S) = m(S_0, C^T).$$

Ejemplo 2.5.3 (Juego optimista para $S = \{1, 2\}$). En este caso, $T = N \setminus \{1, 2\} = \{3\}$. El coste entre los miembros de S no cambia, por lo que $c_{12}^T = c_{12} = 5$. Si recalculamos las conexiones a la fuente,

obtenemos $c_{01}^T = \min(c_{01}, c_{31}) = \min(10, 3) = 3$ y $c_{02}^T = \min(c_{02}, c_{32}) = \min(6, 3) = 3$. Por tanto, la matriz C^T para esta coalición quedaría del siguiente modo:

$$C^T = \begin{pmatrix} 0 & 3 & 3 \\ 3 & 0 & 5 \\ 3 & 5 & 0 \end{pmatrix}.$$

Observemos que, para ambos nodos de la coalición, el coste de conexión con la fuente se reduce al considerar la mejor alternativa a través del nodo 3, en lugar de su conexión directa con la fuente. El árbol de coste mínimo utiliza estas conexiones actualizadas:

$$v^o(\{1, 2\}) = 3 + 3 = 6.$$

Como era de esperar, este valor es el más bajo hasta el momento debido a la reducción de costes inducida por el nodo 3.

El juego público

Una postura intermedia, propuesta por Bogomolnaia y Moulin (2010), consiste en asumir que no existen derechos de propiedad sobre los nodos. Bajo este enfoque, cualquier agente, esté o no en S , puede ser utilizado como intermediario, siempre que se pague el coste de los arcos correspondientes. Esto equivale a permitir que S se beneficie de la red óptima de cualquier coalición más grande que la contenga y se quede solo con el coste total resultante.

Definición 2.5.4 (Juego público). Dado un *mcstp* (N_0, C) , el *juego público* (N, v^u) se define para cada coalición $S \subseteq N$ como

$$v^u(S) = \min_{T \supseteq S} v^p(T).$$

Ejemplo 2.5.4 (Juego público para $S = \{1, 2\}$). Las únicas coaliciones que contienen a $S = \{1, 2\}$ son la propia $\{1, 2\}$ y N . Comparamos $v^p(\{1, 2\}) = 11$ (calculado en el Ejemplo 2.5.1) con $v^p(N) = 12$. El mínimo de ambos valores es

$$v^u(\{1, 2\}) = \min(11, 12) = 11.$$

En este caso particular, a la coalición no le compensa asumir el coste del grupo completo, por lo que se queda con su propio valor privado.

El juego ciclo-completo

Por último, aplicando la misma idea pesimista del juego privado sobre la red ciclo-completa C^* (Definición 2.3.2), obtenemos el juego ciclo-completo. Este modelo, introducido por Trudeau (2012), resulta especialmente interesante en la literatura porque garantiza que el juego resultante sea siempre cóncavo.

Definición 2.5.5 (Juego ciclo-completo). Dado un *mcstp* (N_0, C) , el *juego ciclo-completo* (N, v^c) se define como el juego privado asociado a la red ciclo-completa C^* . Para cada coalición $S \subseteq N$, el coste viene dado por:

$$v^c(S) = m(S_0, C^*).$$

Ejemplo 2.5.5 (Juego ciclo-completo para $S = \{1, 2\}$). Sobre la matriz C^* disponemos de $c_{01}^* = c_{02}^* = 6$ y $c_{12}^* = 5$. El árbol más barato para conectar S utiliza, por ejemplo, las aristas $(0, 1)$ y $(1, 2)$ (o, de forma equivalente, $(0, 2)$ y $(1, 2)$), de modo que

$$v^c(\{1, 2\}) = 6 + 5 = 11.$$

Fijémonos que, en esta coalición concreta, el juego ciclo-completo coincide con el privado ($v^c(\{1, 2\}) = v^p(\{1, 2\}) = 11$). Esto se debe a que la red ciclo-completa no genera nuevas conexiones más económicas para S_0 , por lo que el árbol de coste mínimo es el mismo en ambos casos.

Repitiendo este procedimiento para el resto de coaliciones, obtenemos la Tabla 2.1, que recoge los cinco juegos completos y permite comparar cómo varía el coste de cada coalición según el enfoque considerado.

Juego \ S	{1}	{2}	{3}	{1, 2}	{1, 3}	{2, 3}	N
$v^p(S)$	10	6	6	11	9	9	12
$v^i(S)$	6	6	6	9	9	9	12
$v^o(S)$	3	3	3	6	6	6	12
$v^u(S)$	9	6	6	11	9	9	12
$v^c(S)$	6	6	6	11	9	9	12

Tabla 2.1: Función característica de los cinco juegos cooperativos asociados al Ejemplo 2.1.1.

Observación 2.5.1 (Relación entre los juegos). No es casualidad que en la Tabla 2.1 los valores de v^o sean siempre los más bajos y los de v^p los más altos. Bergantiños y Vidal-Puga (2021) prueban que, para cualquier $mcstp$ y cualquier coalición S , se cumple

$$v^o(S) \leq v^i(S) \leq v^c(S) \leq v^p(S).$$

Puede comprobarse que en nuestra tabla se verifica este orden en todas las coaliciones. El juego público v^u también se sitúa entre v^i y v^p , aunque no mantiene una relación de orden fija con v^c . Además, fijémonos que todos los juegos coinciden en la coalición N . Esto es así puesto que en todos los casos se hereda, por construcción, el coste óptimo $m(N_0, C) = 12$.

Al final del Capítulo 3 utilizaremos estos juegos como punto de partida para aplicar las principales soluciones vistas en el Capítulo 1 (núcleo, valor de Shapley y nucleolo).

Capítulo 3

Asignación de costes

En el Capítulo 2 resolvimos la parte puramente estructural del problema al construir un árbol que conecta a todos los agentes con la fuente al menor coste posible. Una vez obtenida esta red óptima el paso natural consiste en determinar cómo distribuir ese coste $m(N_0, C)$ entre los agentes que se benefician de la red. Para analizar las distintas alternativas de asignación debemos establecer primero la definición formal de regla de reparto (Bergantiños y Vidal-Puga 2021).

Una *regla de reparto* es una función f que asigna a cada *mcstp* (N_0, C) un vector $f(N_0, C) \in \mathbb{R}^N$ que satisface la condición de eficiencia:

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

donde $f_i(N_0, C)$ representa el coste asignado al agente i . Es decir, una regla reparte el coste óptimo total $m(N_0, C)$ entre los agentes de forma exacta.

En este capítulo presentamos las reglas de reparto más estudiadas en la literatura agrupadas en dos grandes familias. Las primeras se analizan en la Sección 3.1 y se derivan de los algoritmos de Prim, Kruskal y Borůvka vistos en el Capítulo 2, donde el propio proceso de construcción del árbol sugiere cómo dividir el coste de cada conexión. Las segundas se detallan en la Sección 3.2 y operan únicamente sobre las funciones características v definidas en la Sección 2.5 mediante la aplicación de los conceptos de solución introducidos en el Capítulo 1.

3.1. Reglas definidas a través del MCSTP

3.1.1. Reglas basadas en Prim: Bird y Dutta-Kar

Recordemos que la estrategia del algoritmo de Prim se basa en hacer crecer una única estructura conectada de manera progresiva a partir de la fuente. Este procedimiento motiva las dos reglas de reparto que analizamos a continuación: Bird y Dutta-Kar.

La regla de Bird

La regla de Bird es la más antigua e intuitiva y fue propuesta originalmente por Bird (1976). Establece que cada agente debe pagar exactamente el coste del arco que lo conecta por primera vez al árbol en construcción.

Definición 3.1.1 (Regla de Bird). Sea (N_0, C) un *mstp* con árbol de coste mínimo único t^* , y sea i^0 el nodo a través del cual el agente i se conecta a la componente de la fuente. La *regla de Bird* asigna

$$B_i(N_0, C) = c_{i^0 i}, \quad \forall i \in N.$$

Cuando el árbol óptimo t^* no es único, como ocurre en el Ejemplo 2.2.1, el algoritmo de Prim puede desempatar de distintas formas según el orden de selección de los arcos. Para solucionar esta falta de unicidad, la regla se extiende promediando el resultado de cada posible orden de desempate $\pi \in \Pi_N$ tal y como detallan Bergantiños y Vidal-Puga (2021):

$$B(N_0, C) = \frac{1}{n!} \sum_{\pi \in \Pi_N} B^\pi(N_0, C).$$

Ejemplo 3.1.1 (Regla de Bird sobre el Ejemplo 2.1.1). Recordemos que en la Etapa 1 del algoritmo de Prim hay un empate entre los arcos $(0, 2)$ y $(0, 3)$, ambos con coste 6 (ver Ejemplo 2.2.1). Esta situación obliga a recurrir a las permutaciones π para desempatar y calcular B^π para cada orden. Consideremos, por ejemplo, el orden $\pi = (1, 2, 3)$:

- Etapas**
- Etapas 1** El empate se resuelve a favor del agente 2 por tener prioridad en el orden de desempate, de forma que el conjunto conectado pasa a ser $\{0, 2\}$ y este paga su coste de conexión $B_2^\pi(N_0, C) = c_{02} = 6$.
 - Etapas 2** El arco más barato hacia el exterior desde el conjunto conectado es $(2, 3)$ con coste 3, lo que implica que el agente 3 se incorpora a la red y paga $B_3^\pi(N_0, C) = c_{23} = 3$.
 - Etapas 3** El agente 1 se conecta finalmente al árbol mediante el arco $(1, 3)$ con un coste de 3, por lo que este último agente asume un pago de $B_1^\pi(N_0, C) = c_{13} = 3$.

De este modo, obtenemos el vector $B^\pi(N_0, C) = (3, 6, 3)$ para dicho orden. Podemos comprobar además que se cumple la propiedad de eficiencia, ya que la suma de los pagos recibidos por todos los agentes iguala el coste total del árbol de forma que $\sum_{i \in N} B_i^\pi(N_0, C) = 3 + 6 + 3 = 12 = m(N_0, C)$.

Repitiendo este procedimiento para las $3! = 6$ permutaciones posibles, obtenemos los resultados de la Tabla 3.1.

Orden π	$B^\pi(N_0, C)$
$(1, 2, 3)$	$(3, 6, 3)$
$(1, 3, 2)$	$(3, 3, 6)$
$(2, 1, 3)$	$(3, 6, 3)$
$(2, 3, 1)$	$(3, 6, 3)$
$(3, 1, 2)$	$(3, 3, 6)$
$(3, 2, 1)$	$(3, 3, 6)$

Tabla 3.1: Asignación de Bird para cada orden de desempate π .

El reparto final es el promedio de estas seis filas:

$$B(N_0, C) = \frac{1}{6} [(3, 6, 3) \cdot 3 + (3, 3, 6) \cdot 3] = (3, 4.5, 4.5).$$

La regla de Dutta-Kar

La regla de Bird presenta un inconveniente conocido al no satisfacer la propiedad de monotonicidad de costes. En términos sencillos, esta propiedad establece que, si el coste de conectar a un agente disminuye, la cantidad que finalmente debe pagar no debería aumentar. Sin embargo, es posible construir ejemplos en los que la regla de Bird produce precisamente este comportamiento contraintuitivo. Para corregir este problema, Dutta y Kar (2004) propusieron una variante en la que el coste de cada conexión no se asigna inmediatamente al agente que se incorpora al árbol, sino que su pago se determina en la etapa siguiente del algoritmo.

Definición 3.1.2 (Regla de Dutta-Kar). Sea (N_0, C) un *mcstp* con árbol de coste mínimo único t^* . Durante la ejecución del algoritmo de Prim definimos $x^0 = 0$ y para cada etapa $p = 1, 2, \dots, n$ calculamos el coste máximo acumulado mediante la expresión

$$x^p = \max\{x^{p-1}, c_{ipj^p}\},$$

donde (i^p, j^p) es el arco incorporado en la etapa p que conecta a un nuevo agente j^p . Si el agente i se conecta en la etapa $p(i)$ de forma que $i = j^{p(i)}$, la *regla de Dutta-Kar* le asigna el pago

$$DK_i(N_0, C) = \begin{cases} \min\{x^{p(i)}, c_{ip(i)+1j^{p(i)+1}}\} & \text{si } p(i) < n \\ x^n & \text{si } p(i) = n. \end{cases}$$

Bajo esta lógica, cada agente paga el mínimo entre el coste máximo acumulado en el momento de su propia conexión y el coste de la conexión inmediatamente posterior. Por su parte, el último agente en incorporarse a la red asume directamente el máximo acumulado total x^n al no tener ninguna etapa posterior con la que comparar (Gómez Saura 2016).

Como en la regla de Bird, cuando el árbol óptimo t^* no es único, la regla de Dutta-Kar se extiende formalmente promediando las asignaciones parciales $DK^\pi(N_0, C)$ sobre todos los órdenes de desempate $\pi \in \Pi_N$ (Bergantiños y Vidal-Puga 2021):

$$DK(N_0, C) = \frac{1}{n!} \sum_{\pi \in \Pi_N} DK^\pi(N_0, C).$$

Ejemplo 3.1.2 (Regla de Dutta-Kar sobre el Ejemplo 2.1.1). Calculemos la asignación $DK^\pi(N_0, C)$ para el orden $\pi = (1, 2, 3)$ reutilizando las mismas etapas del algoritmo de Prim descritas en el Ejemplo 2.2.1. En este caso, el agente 2 se conectaba en la Etapa 1 con coste 6, el agente 3 lo hacía en la Etapa 2 con coste 3 y el agente 1 se incorporaba en la Etapa 3 con coste 3, siendo el último en conectarse. En consecuencia, el coste máximo acumulado tras cada etapa es

$$x^1 = 6, \quad x^2 = 6, \quad x^3 = 6$$

- Etapas 1** El agente 2 paga el mínimo entre $x^1 = 6$ y el coste de la siguiente conexión, correspondiente a la Etapa 2 ($c_{23} = 3$), obteniendo $DK_2^\pi(N_0, C) = \min(6, 3) = 3$.
- Etapas 2** El agente 3 paga el mínimo entre $x^2 = 6$ y el coste de la conexión de la Etapa 3 ($c_{13} = 3$), resultando en $DK_3^\pi(N_0, C) = \min(6, 3) = 3$.
- Etapas 3** El agente 1, al ser el último en conectarse, asume directamente el coste máximo acumulado, de forma que $DK_1^\pi(N_0, C) = x^3 = 6$.

Así, $DK^\pi(N_0, C) = (6, 3, 3)$ para $\pi = (1, 2, 3)$. Nuevamente, se puede comprobar la eficiencia de este reparto: $\sum_{i \in N} DK_i^\pi(N_0, C) = 6 + 3 + 3 = 12 = m(N_0, C)$.

Repitiendo el procedimiento para las seis permutaciones posibles, obtenemos la Tabla 3.2.

Orden π	$DK^\pi(N_0, C)$
(1, 2, 3)	(6, 3, 3)
(1, 3, 2)	(3, 6, 3)
(2, 1, 3)	(6, 3, 3)
(2, 3, 1)	(6, 3, 3)
(3, 1, 2)	(3, 6, 3)
(3, 2, 1)	(6, 3, 3)

Tabla 3.2: Asignación de Dutta-Kar para cada orden de desempate π .

El reparto final es el promedio de las seis filas:

$$DK(N_0, C) = \frac{1}{6} \left[(6, 3, 3) \cdot 4 + (3, 6, 3) \cdot 2 \right] = (5, 4, 3).$$

Este resultado, comparado con el obtenido con la regla de Bird del Ejemplo 3.1.1, $B(N_0, C) = (3, 4.5, 4.5)$, reparte el coste de una forma más extrema. El agente 1, que se conecta más tarde, paga ahora más (5 frente a 3), mientras que los agentes 2 y 3 pagan un poco menos.

Esta diferencia se debe a que Dutta-Kar utiliza los costes máximos acumulados de cada etapa en lugar del precio de los arcos individuales. De este modo, se garantiza que si una conexión se abarata, ningún agente sufra un aumento en su pago, lo que soluciona el fallo de la regla de Bird y asegura la propiedad de monotonicidad.

3.1.2. Reglas basadas en Kruskal: folk y Shapley ponderado

Recordemos que, a diferencia de Prim, que hace crecer un único árbol desde la fuente, el algoritmo de Kruskal ordena todos los arcos de menor a mayor coste para evaluarlos secuencialmente. Bajo esta lógica, el reparto del coste de cada arco no depende de la forma exacta que tenga la red, sino únicamente de qué dos componentes conexas se fusionan al añadirlo.

Esta diferencia tiene una consecuencia práctica muy importante cuando se producen empates entre varios arcos del mismo coste. Aunque el orden en que se procesen estos arcos pueda cambiar las fusiones intermedias en esa etapa, el resultado final de la partición es siempre el mismo. Por ello, a diferencia de las reglas de la sección anterior, los métodos que presentamos a continuación no necesitan promediarse sobre distintas permutaciones, ya que el reparto resultante es siempre único, independientemente de cómo se resuelvan los empates en el algoritmo de Kruskal. Formalmente, esta propiedad se conoce como *independencia del camino* (Bergantiños et al. 2010).

Bajo este enfoque, y apoyándonos en unas funciones especiales denominadas *funciones de obligación*, este algoritmo motiva las tres reglas de reparto que analizaremos a continuación: folk y las dos variantes, optimista y pesimista, del valor de Shapley ponderado.

El marco general: funciones de obligación

La familia de reglas que estudiaremos en esta sección se construye asignando en cada etapa del algoritmo de Kruskal una parte del coste del arco añadido a los agentes de las componentes que se fusionan. La herramienta matemática para especificar exactamente qué parte asume cada agente es la función de obligación.

Definición 3.1.3 (Función y regla de obligación). Una *función de obligación* es una aplicación o que asigna a cada coalición no vacía $S \subseteq N_0$ un vector $o(S) \in \mathbb{R}^S$ cumpliendo las tres propiedades siguientes:

1. Si $0 \notin S$, entonces $o(S)$ es un vector de reparto: $o_i(S) \geq 0$ para todo $i \in S$ y $\sum_{i \in S} o_i(S) = 1$.
2. Si $0 \in S$, entonces $o_i(S) = 0$ para todo $i \in S$.
3. Si $S \subseteq T$ e $i \in S$, entonces $o_i(S) \geq o_i(T)$ (la obligación de un agente nunca aumenta al unirse a un grupo más grande).

Dada una función de obligación o , la *regla de obligación* asociada f^o reparte el coste de cada arco (i^p, j^p) seleccionado por Kruskal en la etapa p evaluando cuánto disminuye la obligación de cada agente al pasar de la partición anterior a la nueva (Tijs et al. 2006; Bergantiños et al. 2011):

$$f_i^o(N_0, C) = \sum_{p=1}^n c_{i^p j^p} \cdot \left[o_i(S(P(g^{p-1}), i)) - o_i(S(P(g^p), i)) \right],$$

donde $S(P(g^p), i)$ denota la componente de la partición P inducida por los arcos de g^p a la que pertenece el agente i .

Intuitivamente, $o_i(S)$ mide la proporción del coste que le queda por pagar al agente i mientras pertenezca a la componente S . Cuando este grupo S se fusiona con otra componente, esa cantidad pendiente disminuye, por lo que la diferencia exacta entre la obligación antigua y la nueva es justo lo que el agente paga en esa etapa concreta.

La regla folk

La regla más estudiada y utilizada dentro de esta familia de reglas de obligación es la conocida como regla folk, introducida por Feltkamp et al. (1994) y posteriormente revisada en Bergantiños y Vidal-Puga (2021). Este método, que también recibe el nombre de regla ERO (por sus siglas en inglés: *Equal Remaining Obligation*), reparte el coste de cada arco a partes iguales entre todos los agentes que pertenecen a la componente que se beneficia de la nueva conexión.

Definición 3.1.4 (Regla folk o ERO). La *regla folk* $F(N_0, C)$ es la regla de obligación asociada a la función

$$o_i(S) = \frac{1}{|S|}, \quad \forall i \in S, \forall S \subset N.$$

Esta elección particular de o responde a los tres principios siguientes:

1. Solo paga el arco quien se beneficia directamente de él.
2. Todos los agentes de una misma componente pagan lo mismo.
3. Cuando dos componentes se fusionan, cada una asume una parte del coste que resulta proporcional al tamaño de la otra.

Ejemplo 3.1.3 (Regla folk sobre el Ejemplo 2.1.1). Retomemos la ejecución del algoritmo de Kruskal analizada en el Ejemplo 2.2.2, donde se incorporaron secuencialmente los arcos $(1, 3)$ y $(2, 3)$, ambos con coste 3 y el arco $(0, 2)$ con coste 6. A partir de lo anterior, es posible recuperar la secuencia de particiones asociada:

$$P^0 = \{\{0\}, \{1\}, \{2\}, \{3\}\} \rightarrow P^1 = \{\{0\}, \{1, 3\}, \{2\}\} \rightarrow P^2 = \{\{0\}, \{1, 2, 3\}\} \rightarrow P^3 = \{N_0\}$$

Etapas Al añadir el arco $(1, 3)$ con coste 3 se fusionan los conjuntos $\{1\}$ y $\{3\}$. La obligación de los agentes 1 y 3 se reduce desde $o_1(\{1\}) = o_3(\{3\}) = 1$ hasta $o_1(\{1, 3\}) = o_3(\{1, 3\}) = \frac{1}{2}$, mientras que el agente 2 no se ve afectado al permanecer en $\{2\}$. Por tanto, los pagos correspondientes a esta conexión son

$$3 \cdot \left(1 - \frac{1}{2}\right) = 1.5 \quad (\text{agentes 1 y 3}), \quad 0 \quad (\text{agente 2}).$$

Etapas 2 Al incorporar el arco $(2, 3)$ con coste 3 se fusiona el conjunto $\{2\}$ con la componente $\{1, 3\}$ para formar el grupo $S = \{1, 2, 3\}$. La obligación del agente 2 baja de $o_2(\{2\}) = 1$ a $o_2(\{1, 2, 3\}) = \frac{1}{3}$, mientras que las obligaciones de los agentes 1 y 3 pasan de $o_1(\{1, 3\}) = o_3(\{1, 3\}) = \frac{1}{2}$ a $o_1(\{1, 2, 3\}) = o_3(\{1, 2, 3\}) = \frac{1}{3}$. Ahora, los pagos resultantes son

$$3 \cdot \left(1 - \frac{1}{3}\right) = 2 \quad (\text{agente 2}), \quad 3 \cdot \left(\frac{1}{2} - \frac{1}{3}\right) = 0.5 \quad (\text{agentes 1 y 3}).$$

Fijémonos que la suma de las aportaciones $(2 + 0.5 + 0.5 = 3)$ cubre el coste exacto del arco. Sin embargo, el reparto no es igualitario, puesto que el agente 2 asume un pago mayor al incorporarse desde una componente individual.

Etapas 3 Con el arco $(0, 2)$ de coste 6 se fusiona el conjunto $\{1, 2, 3\}$ con la fuente. Los tres agentes pasan de su obligación previa $o_i(\{1, 2, 3\}) = \frac{1}{3}$ a una nueva obligación nula $o_i(N_0) = 0$ al contener ya a la fuente, así que cada uno paga

$$6 \cdot \left(\frac{1}{3} - 0\right) = 2.$$

Sumando los pagos acumulados en las tres etapas, obtenemos el reparto final:

$$F(N_0, C) = (1.5 + 0.5 + 2, \quad 0 + 2 + 2, \quad 1.5 + 0.5 + 2) = (4, 4, 4).$$

Reglas de Shapley ponderado

Mientras que la regla folk trata por igual a todos los agentes dentro de una misma componente, existen numerosos escenarios reales donde estos presentan características distintas (tamaño poblacional, capacidad económica o nivel de demanda). Para reflejar matemáticamente estas diferencias, sustituimos el reparto igualitario $1/|S|$ por un procedimiento que pondere el coste en función de un sistema de pesos positivos $w = (w_i)_{i \in N}$ con $w_i > 0$ para todo $i \in N$. Existen dos reglas distintas para hacer esta generalización: la variante optimista y la variante pesimista del valor de Shapley ponderado.

La primera variante es la regla de Shapley ponderado optimista, introducida por Bergantiños y Lorenzo-Freire (2008a,b). Este método adopta una perspectiva favorable para los agentes en cada etapa de Kruskal, ya que calcula sus obligaciones considerando únicamente a los miembros del grupo que se está fusionando en ese momento e ignora temporalmente al resto de agentes de la red.

Definición 3.1.5 (Regla de Shapley ponderado optimista). Dado un sistema de pesos positivos $w = (w_i)_{i \in N}$, la *regla de Shapley ponderado optimista* $f^{ow}(N_0, C)$ es la regla de obligación asociada a

$$o_i(S) = \frac{w_i}{\sum_{j \in S} w_j}, \quad \forall i \in S, \forall S \subset N.$$

Bajo este enfoque, cada agente tiene una obligación directamente proporcional a su propio peso específico dentro de la componente a la que pertenece en esa etapa. Fijémonos que si a todos los agentes se les asigna el mismo peso, la expresión anterior se reduce a $o_i(S) = 1/|S|$, recuperando así la regla folk.

Ejemplo 3.1.4 (Shapley ponderado optimista sobre el Ejemplo 2.1.1). Retomemos la ejecución del algoritmo de Kruskal analizada en el Ejemplo 2.2.2. Con el propósito de ilustrar el efecto de las asimetrías, asignemos al agente 2 el doble de peso que a los agentes 1 y 3, fijando el vector de pesos $w = (w_1, w_2, w_3) = (1, 2, 1)$. Al igual que en el ejemplo anterior, la secuencia de particiones es

$$P^0 = \{\{0\}, \{1\}, \{2\}, \{3\}\} \rightarrow P^1 = \{\{0\}, \{1, 3\}, \{2\}\} \rightarrow P^2 = \{\{0\}, \{1, 2, 3\}\} \rightarrow P^3 = \{N_0\}$$

Etapas 1 Al añadir el arco $(1, 3)$ con coste 3 se fusionan los conjuntos $\{1\}$ y $\{3\}$. Dado que ambos agentes cuentan con el mismo peso ($w_1 = w_3 = 1$), se conserva la simetría en esta primera conexión. La obligación de los agentes 1 y 3 disminuye desde $o_1(\{1\}) = o_3(\{3\}) = 1$

hasta $o_1(\{1, 3\}) = o_3(\{1, 3\}) = \frac{1}{1+1} = \frac{1}{2}$, mientras que el agente 2 permanece igual en la componente $\{2\}$. Por tanto, los pagos correspondientes a esta conexión son

$$3 \cdot (1 - \frac{1}{2}) = 1.5 \quad (\text{agentes 1 y 3}), \quad 0 \quad (\text{agente 2}).$$

Etapas 2 Al incorporar el arco $(2, 3)$ con coste 3 se fusiona el conjunto $\{2\}$ con la componente $\{1, 3\}$ formando el grupo $S = \{1, 2, 3\}$. La obligación del agente 2 baja de $o_2(\{2\}) = 1$ a $o_2(\{1, 2, 3\}) = \frac{2}{1+2+1} = \frac{1}{2}$, mientras que para los agentes 1 y 3 el valor de la función cambia de $o_1(\{1, 3\}) = o_3(\{1, 3\}) = \frac{1}{2}$ a $o_1(\{1, 2, 3\}) = o_3(\{1, 2, 3\}) = \frac{1}{1+2+1} = \frac{1}{4}$. Ahora, los pagos resultantes son

$$3 \cdot (1 - \frac{1}{2}) = 1.5 \quad (\text{agente 2}), \quad 3 \cdot (\frac{1}{2} - \frac{1}{4}) = 0.75 \quad (\text{agentes 1 y 3}).$$

Fijémonos que la suma de las aportaciones $(1.5 + 0.75 + 0.75 = 3)$ cubre el coste exacto del arco. Sin embargo, el reparto no es igualitario, puesto que el agente 2 asume una carga mayor debido a su peso relativo dentro del nuevo grupo fusionado.

Etapas 3 Con el arco $(0, 2)$ de coste 6 se fusiona el conjunto $\{1, 2, 3\}$ con la fuente. Los tres agentes pasan de su obligación previa a tener obligación nula $o_i(N_0) = 0$ al contener ya a la fuente, así que cada uno paga

$$6 \cdot (\frac{1}{4} - 0) = 1.5 \quad (\text{agentes 1 y 3}), \quad 6 \cdot (\frac{1}{2} - 0) = 3 \quad (\text{agente 2}).$$

Sumando los pagos acumulados en las tres etapas, obtenemos el reparto final:

$$f^{ow}(N_0, C) = (1.5 + 0.75 + 1.5, \quad 0 + 1.5 + 3, \quad 1.5 + 0.75 + 1.5) = (3.75, \quad 4.5, \quad 3.75).$$

Como era de esperar, el agente 2, al poseer un mayor peso en la red, asume un coste final superior al que le correspondía mediante la regla folk (4.5 frente a 4).

La segunda variante es la regla de Shapley ponderado pesimista, caracterizada formalmente por Lorenzo y Lorenzo-Freire (2009). Aunque comparte el objetivo de repartir los costes utilizando el sistema de pesos w , en lugar de un reparto proporcional directo, este método calcula la obligación de cada agente promediando sobre todos los posibles órdenes de llegada del resto de miembros a la componente.

Definición 3.1.6 (Regla de Shapley ponderado pesimista). Dado un sistema de pesos positivos $w = (w_i)_{i \in N}$, la *regla de Shapley ponderado pesimista* $f^{pw}(N_0, C)$ es la regla de obligación asociada a

$$o_i(S) = \sum_{\pi \in \Pi(S \setminus \{i\})} \prod_{j=1}^{s-1} \frac{w_{\pi^{-1}(j)}}{\sum_{k=1}^j w_{\pi^{-1}(k)} + w_i}, \quad \forall i \in S, \quad \forall S \subset N,$$

donde $s = |S|$ y $\Pi(S \setminus \{i\})$ denota el conjunto de permutaciones sobre $S \setminus \{i\}$.

A diferencia de la versión optimista, la pesimista exige evaluar todos los escenarios en los que el resto de agentes podrían haberse unido al grupo antes que el agente i .

Ejemplo 3.1.5 (Shapley ponderado pesimista sobre el Ejemplo 2.1.1). Calculemos el reparto para la misma ejecución de Kruskal evaluada en los ejemplos anteriores. Mantendremos el vector de pesos asimétrico $w = (w_1, w_2, w_3) = (1, 2, 1)$ y la secuencia de particiones ya conocida:

$$P^0 = \{\{0\}, \{1\}, \{2\}, \{3\}\} \rightarrow P^1 = \{\{0\}, \{1, 3\}, \{2\}\} \rightarrow P^2 = \{\{0\}, \{1, 2, 3\}\} \rightarrow P^3 = \{N_0\}$$

Etapas 1 Al añadir el arco $(1, 3)$ con coste 3 se fusionan los conjuntos $\{1\}$ y $\{3\}$. Dado que ambos agentes cuentan con el mismo peso ($w_1 = w_3 = 1$), la simetría se conserva en esta primera conexión y la regla pesimista ofrece el mismo resultado que la optimista. La obligación de los agentes 1 y 3 disminuye a $o_1(\{1, 3\}) = o_3(\{1, 3\}) = \frac{1}{2}$, mientras que el agente 2 permanece inalterado. Los pagos correspondientes a esta conexión son

$$3 \cdot (1 - \frac{1}{2}) = 1.5 \quad (\text{agentes 1 y 3}), \quad 0 \quad (\text{agente 2}).$$

Etapla 2 Al incorporar el arco $(2, 3)$ con coste 3 se fusiona el conjunto $\{2\}$ con la componente $\{1, 3\}$ formando el grupo $S = \{1, 2, 3\}$. En este caso, debemos promediar los órdenes de llegada. Para calcular la nueva obligación del agente 1, como $S \setminus \{1\} = \{2, 3\}$, evaluamos los dos escenarios posibles:

- Orden $(2, 3)$: el agente 2 llega primero. La probabilidad del escenario es

$$\frac{w_2}{w_2 + w_1} \cdot \frac{w_3}{w_2 + w_3 + w_1} = \frac{2}{3} \cdot \frac{1}{4} = \frac{1}{6}.$$

- Orden $(3, 2)$: el agente 3 llega primero. La probabilidad del escenario es

$$\frac{w_3}{w_3 + w_1} \cdot \frac{w_2}{w_3 + w_2 + w_1} = \frac{1}{2} \cdot \frac{2}{4} = \frac{1}{4}.$$

Sumando ambas contribuciones, obtenemos $o_1(\{1, 2, 3\}) = \frac{1}{6} + \frac{1}{4} = \frac{5}{12}$. Por simetría de pesos ($w_1 = w_3$), es evidente que $o_3(\{1, 2, 3\}) = \frac{5}{12}$. Como las obligaciones deben sumar uno, el agente 2 asume el resto, por lo que $o_2(\{1, 2, 3\}) = 1 - 2 \cdot \frac{5}{12} = \frac{1}{6}$. Ahora, con estos valores calculados, los pagos resultantes son

$$3 \cdot (1 - \frac{1}{6}) = 2.5 \quad (\text{agente 2}), \quad 3 \cdot (\frac{1}{2} - \frac{5}{12}) = 0.25 \quad (\text{agentes 1 y 3}).$$

Etapla 3 Con el arco $(0, 2)$ de coste 6 se fusiona el conjunto $\{1, 2, 3\}$ con la fuente. Los tres agentes pasan de su obligación previa a tener obligación nula $o_i(N_0) = 0$ al conectar con la fuente, así que cada uno asume la correspondiente fracción del coste:

$$6 \cdot (\frac{5}{12} - 0) = 2.5 \quad (\text{agentes 1 y 3}), \quad 6 \cdot (\frac{1}{6} - 0) = 1 \quad (\text{agente 2}).$$

Sumando los pagos acumulados en las tres etapas, obtenemos el reparto final:

$$f^{pw}(N_0, C) = (1.5 + 0.25 + 2.5, \quad 0 + 2.5 + 1, \quad 1.5 + 0.25 + 2.5) = (4.25, \quad 3.5, \quad 4.25).$$

Si reunimos los tres resultados obtenidos a lo largo de esta sección, podemos ver claramente cómo afecta la elección de la regla al reparto final:

$$F(N_0, C) = (4, 4, 4), \quad f^{ow}(N_0, C) = (3.75, 4.5, 3.75), \quad f^{pw}(N_0, C) = (4.25, 3.5, 4.25).$$

El agente 2, al tener el doble de peso que sus compañeros, paga más con la regla optimista (4.5) que con la regla folk (4). Sin embargo, con la regla pesimista ocurre exactamente lo contrario, ya que se beneficia de su mayor peso y su aportación baja a 3.5.

Esto demuestra que tener un mayor peso en la red no implica necesariamente asumir un coste más alto. Su efecto puede invertirse dependiendo de la fórmula matemática que se utilice para calcular las obligaciones.

3.1.3. Regla basada en Borůvka

Recordemos que, a diferencia de los métodos secuenciales de Prim y Kruskal, el algoritmo de Borůvka avanza de forma paralela. La regla de Borůvka, introducida originalmente por Bergantiños y Vidal-Puga (2011) y revisada en Bergantiños y Vidal-Puga (2021), divide el pago en varias iteraciones simultáneas. En lugar de pagar cada arco de manera individual e íntegra, en cada etapa, los agentes responsables aportan una misma proporción de su coste y esta se maximiza con la condición de que ningún arco supere la totalidad de su valor. Cuando un arco se paga por completo, las componentes implicadas se fusionan y la siguiente iteración se recalcula sobre esta nueva estructura.

Definición 3.1.7 (Regla de Borůvka). Sea (N_0, C) un *mcstp*, π un orden estricto sobre los arcos y t^π (o simplemente t) el árbol de coste mínimo obtenido mediante el algoritmo de Borůvka asociado a π . Como inicialmente ningún arco está pagado, sea la proporción acumulada inicial $\varrho_{ij}^0 = 0$ para todo $(i, j) \in t$. Para cada etapa de pago $s = 1, 2, \dots, \gamma$ definimos de forma recursiva las siguientes magnitudes:

- $A^{s-1} = \{(i, j) \in t : \varrho_{ij}^{s-1} < 1\}$: el conjunto de arcos de t que aún no han sido completamente pagados al inicio de la etapa s .
- $\mathcal{P}^{s-1}$: la partición de N en componentes conexas inducida por los arcos ya pagados en las etapas anteriores.
- $a_i^s \in A^{s-1}$: el arco que el agente $i \in T \in \mathcal{P}^{s-1}$ (con $0 \notin T$) contribuye a pagar en la etapa s , seleccionado como el arco más barato de t que conecta su componente T con el exterior. En caso de empate, se elige según el orden π .
- $N_{ij}^s = \{k \in N : a_k^s = (i, j)\}$: el conjunto de agentes que pagan el arco (i, j) en la etapa s .
- $p^s = \min \left\{ \frac{1 - \varrho_{ij}^{s-1}}{|N_{ij}^s|} : (i, j) \in A^{s-1}, N_{ij}^s \neq \emptyset \right\}$: la máxima proporción común que los agentes pueden pagar simultáneamente en la etapa s .
- $\varrho_{ij}^s = \varrho_{ij}^{s-1} + |N_{ij}^s| \cdot p^s$: la proporción acumulada que se ha pagado del arco (i, j) tras la etapa s .
- $f_i^s = p^s c_{a_i^s}$: el pago del agente i en la etapa s .

El proceso finaliza en un número finito de etapas γ (con $\gamma \leq |N|$) cuando todos los arcos de t cumplen $\varrho_{ij}^\gamma = 1$, lo que equivale a $\sum_{s=1}^\gamma p^s = 1$. La *regla de Borůvka* asociada al orden π se define como

$$\beta_i^\pi(N_0, C) = \sum_{s=1}^\gamma f_i^s, \quad \forall i \in N.$$

A pesar de que la construcción de las etapas intermedias depende del orden elegido, Bergantiños y Vidal-Puga (2011) demuestran que, para cualquier orden π , la regla $\beta^\pi(N_0, C)$ siempre coincide con la regla folk.

Ejemplo 3.1.6 (Regla de Borůvka sobre el Ejemplo 2.1.1). Retomemos el árbol de coste mínimo obtenido en el Ejemplo 2.2.3, dado por $t = \{(1, 3), (2, 3), (0, 2)\}$ con costes $c_{13} = 3$, $c_{23} = 3$ y $c_{02} = 6$, bajo el orden estricto π donde $(1, 3) \succ (2, 3) \succ (1, 2) \succ (0, 2) \succ (0, 3) \succ (0, 1)$. El proceso se desarrolla en las siguientes etapas:

Etapas 1 Inicialmente ningún arco está pagado, por lo que las componentes conexas de partida son los nodos individuales $\mathcal{P}^0 = \{\{1\}, \{2\}, \{3\}\}$. Cada agente busca en t su arco más barato hacia el exterior: el agente 1 selecciona $a_1^1 = (1, 3)$ y el agente 2 elige $a_2^1 = (2, 3)$. Para el agente 3 hay un empate en coste entre los arcos $(1, 3)$ y $(2, 3)$. Dado que $(1, 3)$ precede a $(2, 3)$ en el orden π , se toma $a_3^1 = (1, 3)$. Esto da lugar a los grupos de pago $N_{(1,3)}^1 = \{1, 3\}$ y $N_{(2,3)}^1 = \{2\}$. La proporción máxima común es

$$p^1 = \min \left\{ \frac{1 - 0}{|\{1, 3\}|}, \frac{1 - 0}{|\{2\}|} \right\} = \min \left\{ \frac{1}{2}, 1 \right\} = \frac{1}{2}.$$

Al terminar esta etapa, el arco $(1, 3)$ queda completamente pagado ($\varrho_{(1,3)}^1 = 1$), mientras que del arco $(2, 3)$ solo se paga la mitad ($\varrho_{(2,3)}^1 = \frac{1}{2}$). Los pagos de esta etapa son

$$f_1^1 = \frac{1}{2} \cdot 3 = 1.5, \quad f_2^1 = \frac{1}{2} \cdot 3 = 1.5, \quad f_3^1 = \frac{1}{2} \cdot 3 = 1.5.$$

Etapas 2 Como el arco $(1, 3)$ ya está pagado, los agentes 1 y 3 se fusionan en la componente $\mathcal{P}^1 = \{\{1, 3\}, \{2\}\}$. Esta componente solo dispone del arco $(2, 3)$ para conectar con el exterior dentro de t , de modo que $a_1^2 = a_3^2 = (2, 3)$. Por su parte, el agente 2 también elige $a_2^2 = (2, 3)$. Así, los tres agentes apuntan al mismo arco: $N_{(2,3)}^2 = \{1, 2, 3\}$. La proporción máxima es ahora

$$p^2 = \frac{1 - \varrho_{(2,3)}^1}{|\{1, 2, 3\}|} = \frac{1 - \frac{1}{2}}{3} = \frac{1}{6},$$

lo que completa el pago total de $(2, 3)$ ($\varrho_{(2,3)}^2 = 1$). Los pagos de esta etapa son

$$f_1^2 = f_2^2 = f_3^2 = \frac{1}{6} \cdot 3 = 0.5.$$

Etapas 3 Dado que los arcos $(1, 3)$ y $(2, 3)$ están completamente pagados, los agentes forman una única componente $\mathcal{P}^2 = \{\{1, 2, 3\}\}$. Su único arco en t hacia la fuente es $(0, 2)$, así que los tres agentes apuntan a él: $N_{(0,2)}^3 = \{1, 2, 3\}$. La proporción es

$$p^3 = \frac{1 - 0}{3} = \frac{1}{3},$$

con lo que el arco $(0, 2)$ queda pagado ($\varrho_{(0,2)}^3 = 1$) y el proceso termina. Los pagos de esta etapa son

$$f_1^3 = f_2^3 = f_3^3 = \frac{1}{3} \cdot 3 = 1.$$

Efectivamente, la suma de las proporciones verifica que $\sum_{s=1}^3 p^s = \frac{1}{2} + \frac{1}{6} + \frac{1}{3} = 1$. Por tanto, sumando los pagos obtenidos en cada una de las etapas, el reparto final es

$$\beta^\pi(N_0, C) = (1.5 + 0.5 + 2, \quad 1.5 + 0.5 + 2, \quad 1.5 + 0.5 + 2) = (4, 4, 4).$$

Este resultado demuestra numéricamente que la regla de Borůvka recupera la asignación de la regla folk.

3.1.4. Reglas basadas en la descomposición en conos

En la Sección 2.4 presentamos cómo descomponer cualquier matriz de costes C en una combinación no negativa de matrices elementales, $C = \sum_{q=1}^{m(C)} x^q C^q$, sin entrar todavía en cómo repartir el coste. Con esta herramienta ya disponible, pasamos ahora a analizar el problema del reparto. Siguiendo la propuesta original de Norde et al. (2004), revisada en Bergantiños y Vidal-Puga (2021), basta con saber distribuir el coste en un problema elemental, en el que cada arco toma el valor 0 o 1, para obtener una regla sobre cualquier problema general mediante una suma ponderada. Esta extensión se formaliza de la siguiente manera.

Definición 3.1.8 (Extensión de una regla mediante descomposición en conos). Sea R una regla definida sobre cualquier problema elemental. Esta se extiende a un *mcstp* general (N_0, C) mediante

$$R(N_0, C) = \sum_{q=1}^{m(C)} x^q R(N_0, C^q).$$

Varios autores han seguido este enfoque para definir familias de reglas. Por ejemplo, Branzai et al. (2004) demuestran que la regla folk se puede obtener exactamente por esta vía. Dado un problema elemental (N_0, C^q) , los agentes que ya están conectados a la fuente mediante caminos de coste 0 no pagan nada. Los que no lo están se agrupan en componentes según los caminos de coste 0 que comparten y reparten a partes iguales el coste 1 de conectar su componente a la fuente. Formalmente:

$$F_i(N_0, C^q) = \begin{cases} \frac{1}{|S(P(g^q), i)|} & \text{si } 0 \notin S(P(g^q), i) \\ 0 & \text{en caso contrario.} \end{cases}$$

Ejemplo 3.1.7 (La regla folk vía descomposición en conos sobre el Ejemplo 2.1.1). Retomemos las cuatro matrices elementales obtenidas en el Ejemplo 2.4.1 e ilustradas en la Figura 2.7 para aplicar a cada una la regla elemental que acabamos de definir.

- Capa 1** Con un peso $x^1 = 3$, todas las conexiones cuestan 1. Como no existe ningún camino de coste 0, cada agente forma su propia componente individual, $\{1\}$, $\{2\}$ y $\{3\}$, sin acceso gratuito a la fuente. Por tanto, cada uno paga el coste 1 íntegro, resultando en $F(N_0, C^1) = (1, 1, 1)$.
- Capa 2** Al aplicar el peso $x^2 = 2$, los arcos $(1, 3)$ y $(2, 3)$ pasan a tener coste 0. Esto conecta a los tres agentes entre sí formando una única componente conjunta $\{1, 2, 3\}$ que todavía no tiene la fuente. Al repartir el coste 1 a partes iguales, el pago es $F(N_0, C^2) = (\frac{1}{3}, \frac{1}{3}, \frac{1}{3})$.
- Capa 3** Con un peso $x^3 = 1$, se repite exactamente la misma situación que en la etapa anterior. Los arcos de coste nulo unen a los agentes en un grupo que sigue aislado de la fuente, por lo que vuelven a repartir el coste de manera equitativa y se obtiene $F(N_0, C^3) = (\frac{1}{3}, \frac{1}{3}, \frac{1}{3})$.
- Capa 4** Finalmente, con un peso $x^4 = 4$, solo la conexión $(0, 1)$ tiene coste 1 y el resto de posibles conexiones con la fuente son gratuitas. Al existir caminos de coste cero hacia el nodo central, los tres agentes pertenecen ya a la componente de la fuente y su pago es nulo, es decir, $F(N_0, C^4) = (0, 0, 0)$.

Combinando las cuatro capas siguiendo la Definición 3.1.8 obtenemos el reparto final:

$$\begin{aligned} F(N_0, C) &= 3 \cdot (1, 1, 1) + 2 \cdot (\frac{1}{3}, \frac{1}{3}, \frac{1}{3}) + 1 \cdot (\frac{1}{3}, \frac{1}{3}, \frac{1}{3}) + 4 \cdot (0, 0, 0) \\ &= (3, 3, 3) + (\frac{2}{3}, \frac{2}{3}, \frac{2}{3}) + (\frac{1}{3}, \frac{1}{3}, \frac{1}{3}) + (0, 0, 0) = (4, 4, 4). \end{aligned}$$

Bajo este enfoque, disponemos de una tercera forma de calcular el reparto de la regla folk, que se suma a los procedimientos basados en Kruskal y Borůvka presentados en los Ejemplos 3.1.3 y 3.1.6.

Además, podemos adaptar la regla elemental para obtener diferentes soluciones. En este sentido, Bergantiños y Lorenzo-Freire (2008a) demuestran que las reglas de Shapley ponderado optimista (Definición 3.1.5) pueden obtenerse mediante este procedimiento. Para ello, en lugar de un reparto igualitario, se utiliza una distribución proporcional a los pesos w_i asignados a cada agente i :

$$f_i^{ow}(N_0, C^q) = \begin{cases} \frac{w_i}{\sum_{j \in S(P(g^q), i)} w_j} & \text{si } 0 \notin S(P(g^q), i) \\ 0 & \text{en caso contrario.} \end{cases}$$

Por su parte, Bogomolnaia y Moulin (2010) definen una familia de reglas elementales que depende de un parámetro λ . Si denotamos por δ_i el número de arcos de coste no nulo para el agente i , la regla para $\lambda \in [0, +\infty)$ viene dada por:

$$R_i^\lambda(N_0, C^q) = \begin{cases} \frac{\lambda^{\delta_i}}{\sum_{j \in S(P(g^q), i)} \lambda^{\delta_j}} & \text{si } 0 \notin S(P(g^q), i) \\ 0 & \text{en caso contrario.} \end{cases}$$

Para $\lambda = +\infty$, el coste del problema se asigna exclusivamente a los agentes con el mayor número de conexiones, es decir, a aquellos en el conjunto $\arg \max_{j \in S(P(g^q), i)} \delta_j$ con $0 \notin S(P(g^q), i)$. En el caso particular de que la componente sea un árbol, estos agentes se reparten el coste unitario por igual (Bogomolnaia y Moulin 2010).

En definitiva, la descomposición en conos permite construir reglas complejas a partir de sus componentes elementales, trasladando las propiedades de los escenarios binarios a cualquier problema general.

3.2. Reglas definidas a través de juegos cooperativos

Las reglas de la sección anterior se construyen mirando directamente al grafo, identificando qué arco se añade en cada etapa y qué agente debe pagarlo. El enfoque que presentamos ahora prescinde por completo de esa información estructural y trabaja únicamente sobre las funciones características v^p , v^i , v^o , v^u y v^c que definimos en la Sección 2.5. Una vez modelado el problema como un juego cooperativo de costes (N, v) , podemos aplicarle las herramientas de solución introducidas en la Sección 1.2. Comenzaremos explorando el núcleo y posteriormente abordaremos el valor de Shapley y el nucleolo.

El núcleo (Gillies 1953)

A diferencia de otras soluciones que asignan un único punto, el núcleo de un juego puede contener infinitos repartos estables o incluso quedar vacío en escenarios con un mal comportamiento matemático. En el contexto de los MCSTP, existen múltiples formas de formular el juego asociado de manera que se garantice la existencia de soluciones estables. En esta línea, Bird (1976) demostró que el núcleo del juego privado nunca es vacío, asegurando así que siempre existe al menos un reparto frente al cual ninguna coalición tiene incentivos para abandonar la coalición total.

Aunque existen otros juegos cuyos núcleos presentan un buen comportamiento, tomaremos el juego privado como referencia. Aplicando los conceptos y la metodología descrita en el Capítulo 1, podemos determinar la estructura del núcleo para el juego privado del Ejemplo 2.1.1 (visto en la Sección 2.5).

Ejemplo 3.2.1 (Cálculo de imputaciones y del núcleo sobre el Ejemplo 2.1.1). Procediendo de forma análoga al Ejemplo 1.2.2, toda imputación debe cumplir la condición de eficiencia ($x_1 + x_2 + x_3 = 12$) y las restricciones de racionalidad individual ($x_1 \leq 10$, $x_2 \leq 6$ y $x_3 \leq 6$). Geométricamente, este conjunto se define como la envoltura convexa de sus vértices:

$$I(N, v^p) = \text{conv}\{(10, 6, -4), (10, -4, 6), (0, 6, 6)\}.$$

Para hallar el núcleo, aplicamos la condición de eficiencia sobre las restricciones de las coaliciones de dos agentes (ver Tabla 2.1), obteniendo así las cotas inferiores para cada jugador:

$$\begin{aligned} x_1 + x_2 &\leq 11 \implies x_3 \geq 1 \\ x_1 + x_3 &\leq 9 \implies x_2 \geq 3 \\ x_2 + x_3 &\leq 9 \implies x_1 \geq 3 \end{aligned}$$

Al intersecar estas nuevas restricciones activas con el triángulo de imputaciones $I(N, v^p)$, el núcleo toma la forma de un cuadrilátero inscrito en él. Calculando las intersecciones correspondientes, obtenemos que el núcleo es la envoltura convexa de los siguientes cuatro vértices:

$$\text{Core}(N, v^p) = \text{conv}\{(3, 3, 6), (8, 3, 1), (5, 6, 1), (3, 6, 3)\}.$$

Dado que todos los puntos válidos residen sobre el plano de eficiencia, es posible proyectar este espacio tridimensional sobre un plano bidimensional sin pérdida de información. La Figura 3.1 ilustra esta proyección. El triángulo exterior punteado delimita el conjunto total de imputaciones, mientras que las rectas sólidas representan los hiperplanos de las coaliciones que acotan y definen el núcleo.

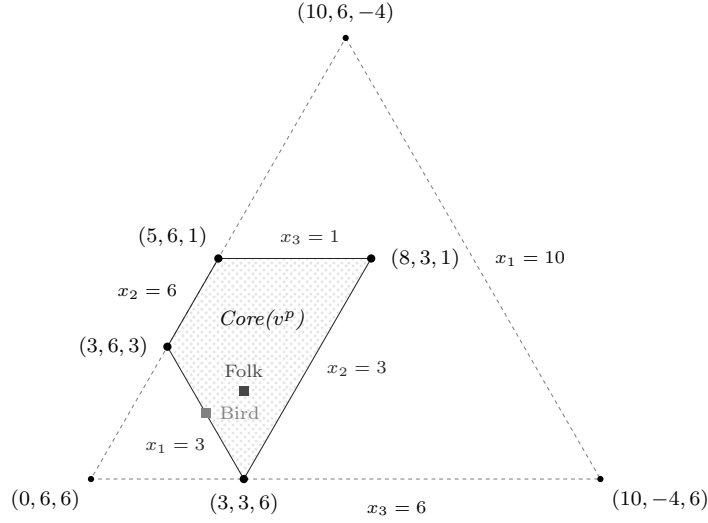


Figura 3.1: Proyección geométrica del núcleo del juego privado v^p (área sombreada) dentro del conjunto de imputaciones. Se constata visualmente que la regla de Bird y la regla folk pertenecen al núcleo.

Fijémonos que la regla de Bird $B(N_0, C) = (3, 4.5, 4.5)$ y la regla folk $F(N_0, C) = (4, 4, 4)$, calculadas en los Ejemplos 3.1.1 y 3.1.3, pertenecen ambas al núcleo (en particular, $B(N_0, C)$ a la frontera). Analíticamente vemos que se respetan todas las inecuaciones. Para la regla de Bird tenemos $x_1 + x_2 = 7.5 \leq 11$, $x_1 + x_3 = 7.5 \leq 9$ y $x_2 + x_3 = 9 \leq 9$, ocurriendo exactamente lo mismo para la regla folk.

Este hecho ilustra un resultado general, recogido en Bergantiños y Vidal-Puga (2021), que garantiza que tanto estas reglas (Feltkamp et al. 1994; Bird 1976) como la de Dutta y Kar (2004), la familia de reglas de Bogomolnaia y Moulin (2010) y las variaciones ponderadas del valor de Shapley (Tijs et al. 2006) proporcionan siempre soluciones que pertenecen al núcleo del juego privado.

El valor de Shapley (Shapley 1953)

El valor de Shapley de un juego (N, v) se puede definir como el promedio de las contribuciones marginales de cada agente sobre todos los posibles órdenes de llegada, o bien como la suma ponderada de sus contribuciones marginales a cada una de las coaliciones posibles, tal y como se formalizó en la Definición 1.2.6. Al aplicar este concepto cooperativo a las cinco funciones características asociadas a un MCSTP, obtenemos cinco reglas de reparto distintas (Bergantiños y Vidal-Puga 2021). En particular, la primera de estas reglas es tan conocida que cuenta con nombre propio en la literatura (Kar 2002).

Definición 3.2.1 (Regla de Kar). La *regla de Kar* se define como el valor de Shapley aplicado directamente sobre la función característica del juego privado:

$$Kar(N_0, C) = Sh(N, v^p).$$

Ejemplo 3.2.2 (Regla de Kar sobre el Ejemplo 2.1.1). Recuperemos los valores de v^p calculados en la Tabla 2.1: $v^p(\{1\}) = 10$, $v^p(\{2\}) = v^p(\{3\}) = 6$, $v^p(\{1, 2\}) = 11$, $v^p(\{1, 3\}) = v^p(\{2, 3\}) = 9$ y $v^p(N) = 12$. Siguiendo el mismo procedimiento que en el Ejemplo 1.2.3 de la Sección 1.2, calculamos los vectores marginales $m^\pi(v^p)$ para los seis órdenes posibles, recogidos en la Tabla 3.3.

Orden π	Vector marginal $m^\pi(v^p)$
(1, 2, 3)	(10, 1, 1)
(1, 3, 2)	(10, 3, -1)
(2, 1, 3)	(5, 6, 1)
(2, 3, 1)	(3, 6, 3)
(3, 1, 2)	(3, 3, 6)
(3, 2, 1)	(3, 3, 6)

Tabla 3.3: Vectores marginales del juego privado v^p .

A modo de ejemplo, si consideramos el orden de llegada $\pi = (1, 3, 2)$, el agente 1 entra en primer lugar asumiendo su coste individual $v^p(\{1\}) = 10$. A continuación se incorpora el agente 3, cuya aportación al grupo es la diferencia $v^p(\{1, 3\}) - v^p(\{1\}) = 9 - 10 = -1$. Finalmente, el agente 2 completa la coalición total con un coste de $v^p(N) - v^p(\{1, 3\}) = 12 - 9 = 3$.

En este caso, la contribución marginal del agente 3 resulta negativa porque, al incorporarse tras el agente 1, proporciona una conexión más barata hacia la fuente que reduce el coste total de la red.

Promediando las seis filas de la tabla, obtenemos el reparto final:

$$\begin{aligned} Kar(N_0, C) &= \frac{1}{6} [(10, 1, 1) + (10, 3, -1) + (5, 6, 1) + (3, 6, 3) + (3, 3, 6) + (3, 3, 6)] \\ &= \left(\frac{34}{6}, \frac{22}{6}, \frac{16}{6}\right) \approx (5.67, 3.67, 2.67). \end{aligned}$$

Para el resto de juegos asociados el procedimiento es idéntico, dando como resultado los siguientes vectores de reparto:

$$Sh(N, v^i) = Sh(N, v^o) = (4, 4, 4), \quad Sh(N, v^u) \approx (5.33, 3.83, 2.83), \quad Sh(N, v^c)^1 \approx (4.33, 4.33, 3.33).$$

Observación 3.2.1. El resultado $Sh(N, v^i) = (4, 4, 4)$ no ocurre por casualidad. Bergantiños y Vidal-Puga (2021) demuestran que, para cualquier MCSTP, se cumple que:

$$Sh(N, v^i) = F(N_0, C),$$

lo que implica que el valor de Shapley del juego irreducible coincide siempre con la regla folk obtenida previamente mediante los enfoques algorítmicos de Kruskal, Borůvka y la descomposición en conos de la Sección 3.1. Adicionalmente, Bergantiños y Vidal-Puga (2007) demuestran que $Sh(N, v^o) = Sh(N, v^i)$, lo que justifica que el juego optimista devuelva exactamente el mismo reparto. Este hecho resulta de gran interés puesto que vincula directamente el enfoque algorítmico con la teoría de juegos cooperativos.

Cabe destacar que el valor de Shapley del juego público no tiene garantizada, en general, su pertenencia al núcleo, ni del propio juego ni del juego privado (Trudeau y Vidal-Puga 2019). No obstante, este no es el caso de nuestro ejemplo, ya que la solución pertenece al núcleo del juego privado, tal y como se muestra en color rosa en la Figura 3.2.

El nucleolo (Schmeidler 1969)

El nucleolo busca la imputación que minimiza lexicográficamente el vector de excesos, tal y como se formalizó en la Definición 1.2.8. Primero minimiza la máxima queja posible entre todas las coaliciones

¹El valor de Shapley aplicado sobre el juego ciclo-completo recibe en la literatura el nombre específico de *cycle-complete solution* (Trudeau 2012).

(es decir, el mayor sobrecoste que soporta alguna coalición respecto a lo que le costaría actuar por su cuenta) y, en caso de empate, sigue minimizando la segunda mayor queja y así sucesivamente. Este proceso de desempate sucesivo es lo que garantiza que el nucleolo sea un único punto y no simplemente un conjunto de imputaciones que comparten la misma queja máxima. A diferencia del Ejemplo 1.2.4, donde bastaba con igualar los excesos de las tres coaliciones de dos agentes, aquí la coalición más exigente no es siempre una pareja. Resolviendo el sistema completo de excesos para todas las coaliciones se obtiene el siguiente resultado para el juego privado:

$$Nu(N, v^p) = (4.75, 4.5, 2.75).$$

En este óptimo, las coaliciones $\{2\}$ y $\{1, 3\}$ resultan ser las que tienen una mayor queja, con un exceso compartido de -1.5 : el agente 2 paga por su cuenta 1.5 unidades menos de lo que le costaría ir solo (4.5 frente a $v^p(\{2\}) = 6$) y de forma equivalente los agentes 1 y 3 conjuntamente ahorran también 1.5 unidades respecto a formar su propia coalición (7.5 frente a $v^p(\{1, 3\}) = 9$). Que ambas coaliciones compartan el mismo exceso es precisamente la condición de equilibrio que caracteriza al nucleolo, que reparte el “descontento” lo más equitativamente posible entre las coaliciones que más presionan por separarse de la gran coalición. En la Figura 3.2 puede comprobarse, además, que este punto pertenece al núcleo calculado en el Ejemplo 3.2.1.

Para el resto de los juegos, el mismo procedimiento da lugar a los siguientes resultados:

$$Nu(N, v^i) = (4, 4, 4), \quad Nu(N, v^u) = (4.75, 4.5, 2.75), \quad Nu(N, v^c) = (4.5, 4.5, 3).$$

El resultado $Nu(N, v^i) = (4, 4, 4)$ se deduce de forma inmediata. Como puede comprobarse en la Tabla 2.1, el juego irreducible es simétrico, es decir, todas las coaliciones del mismo tamaño tienen idéntico valor ($v^i(\{1\}) = v^i(\{2\}) = v^i(\{3\})$ y $v^i(\{1, 2\}) = v^i(\{1, 3\}) = v^i(\{2, 3\})$). Puesto que tanto el valor de Shapley como el nucleolo satisfacen las propiedades de eficiencia y simetría, González-Díaz y Sánchez-Rodríguez (2014) afirman que en esta clase de juegos ambos conceptos coinciden necesariamente con la solución de división igualitaria del coste total. Esto explica de forma analítica por qué $Nu(N, v^i) = Sh(N, v^i)$.

Observación 3.2.2. A diferencia de v^p o v^i , el juego optimista v^o no garantiza un conjunto de imputaciones no vacío. La racionalidad individual exige $x(N) \leq \sum_{i \in N} v^o(\{i\})$, mientras que la eficiencia requiere $x(N) = v^o(N)$. Si el coste de la coalición total supera la suma de los costes individuales, es decir, $v^o(N) > \sum_{i \in N} v^o(\{i\})$, ambas condiciones son incompatibles. En el Ejemplo 2.1.1, dado que $v^o(N) = 12$ y $\sum_{i \in N} v^o(\{i\}) = 9$, el conjunto de imputaciones resulta vacío y, por consiguiente, el núcleo y el nucleolo no están definidos. Por este motivo, se excluye $Nu(N, v^o)$ entre los resultados anteriores.

Comparativa de reglas

La Tabla 3.4 recopila las reglas mencionadas a lo largo del capítulo para el ejemplo considerado para su ilustración. Estas soluciones se representan geométricamente junto al núcleo del juego privado en la Figura 3.2, seleccionando un único representante para aquellas que devuelven el mismo valor. Esto nos permite extraer las siguientes conclusiones:

En primer lugar, destaca la gran coincidencia en torno a la regla folk, cuyo reparto $(4, 4, 4)$ se obtiene de forma idéntica mediante cinco enfoques independientes: dos algorítmicos (Kruskal y Borůvka), uno algebraico (descomposición en conos) y dos cooperativos (el valor de Shapley y el nucleolo del juego irreducible), coincidiendo además con el valor de Shapley del juego optimista.

En segundo lugar, aunque teóricamente la pertenencia al núcleo del juego privado v^p no está garantizada de forma general para cualquier regla de reparto (Bergantiños y Vidal-Puga 2021), se puede comprobar

en la Figura 3.2 que todas las soluciones calculadas caen efectivamente dentro de él. En tercer lugar, destaca la sensibilidad de los pesos en las reglas de Shapley ponderado. Al introducir asimetrías, las versiones optimista y pesimista se alejan de la regla folk en direcciones opuestas para el agente más pesado, lo que demuestra que ambos enfoques miden la importancia de forma distinta aunque coincidan cuando los pesos son iguales.

Familia	Regla	Agente 1	Agente 2	Agente 3
<i>Reglas algorítmicas</i>	Bird	3.00	4.50	4.50
	Dutta-Kar	5.00	4.00	3.00
	Folk Borůvka	4.00	4.00	4.00
	Shapley pond. optimista	3.75	4.50	3.75
	Shapley pond. pesimista	4.25	3.50	4.25
<i>Juegos cooperativos</i>	Kar ($Sh(v^p)$)	5.67	3.67	2.67
	$Sh(v^i) = Sh(v^o)$	4.00	4.00	4.00
	$Sh(v^u)$	5.33	3.83	2.83
	$Sh(v^c)$	4.33	4.33	3.33
	$Nu(v^p) = Nu(v^u)$	4.75	4.50	2.75
	$Nu(v^i)$	4.00	4.00	4.00
	$Nu(v^o)$	—	—	—
	$Nu(v^c)$	4.50	4.50	3.00

Tabla 3.4: Comparativa de las reglas de reparto sobre el Ejemplo 2.1.1.

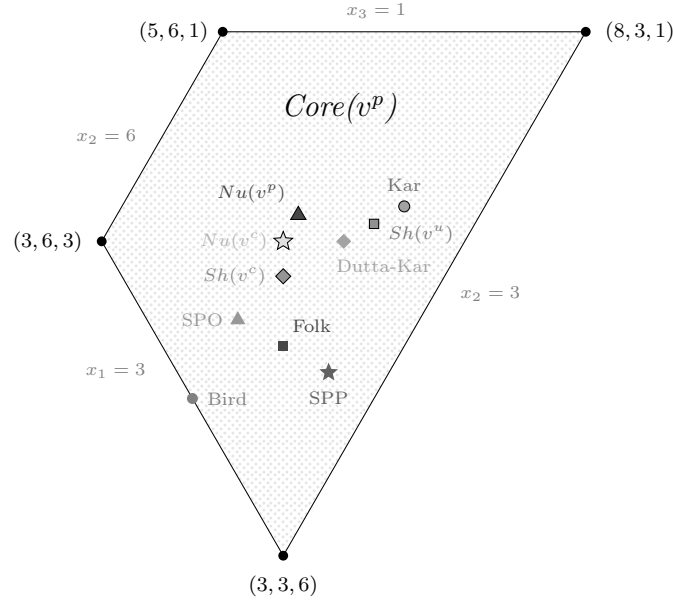


Figura 3.2: Detalle ampliado del núcleo del juego privado v^p de la Figura 3.1 mostrando las distintas reglas de reparto.

Finalmente, podemos destacar la diferencia entre el juego irreducible v^i y el optimista v^o . Aunque ambos comparten el mismo valor de Shapley, v^i garantiza un núcleo y un nucleolo bien definidos, mientras que en v^o el conjunto de imputaciones resulta vacío. Lo interesante de esto es que, aunque el escenario optimista sea inestable por sí solo, el reparto que propone su valor de Shapley acaba siendo una solución perfectamente válida y estable bajo las reglas del juego privado (ver Figura 3.2).

3.3. El método de Monte Carlo en el cálculo de asignaciones

Algunas de las reglas presentadas en las secciones anteriores pueden requerir, en determinados casos, un cálculo computacionalmente inviable. Cuando un problema presenta conexiones con costes idénticos, el árbol de mínimo coste puede no ser único. En estas situaciones, las reglas basadas en el algoritmo de Prim, como la de Bird y Dutta-Kar, dependen del orden en el que los agentes se van conectando a la red, tal y como se explicó en la Sección 3.1. Para resolver este problema de forma justa, dichas reglas se extienden promediando la asignación obtenida para cada uno de los $n!$ posibles órdenes de entrada de los agentes $\pi \in \Pi_N$ que se usan para deshacer los empates en el algoritmo de Prim:

$$f(N_0, C) = \frac{1}{n!} \sum_{\pi \in \Pi_N} f^\pi(N_0, C), \quad (3.1)$$

donde $f = B$ representa la regla extendida de Bird (Definición 3.1.1) y $f = DK$ la de Dutta-Kar (Definición 3.1.2). El número de términos de esta suma crece de forma factorial con el tamaño de la red, lo que convierte el cálculo exacto en completamente inviable a medida que aumenta el número de agentes.

No todas las reglas requieren este tratamiento especial. Reglas como la folk o la basada en el algoritmo de Borůvka construyen la red fusionando componentes conectadas en lugar de hacer crecer un único árbol desde el origen. Como se demostró en este capítulo, el resultado de estas reglas no depende del orden en que se resuelvan los empates durante la construcción, por lo que devuelven siempre una asignación única sin necesidad de promediar nada.

El valor de Shapley, por su parte, presenta una formulación basada en recorrer todas las coaliciones $S \subseteq N$ necesarias para construir la función característica completa, $2^n - 1$:

$$Sh_i(N, v) = \sum_{S \subseteq N \setminus \{i\}} \frac{|S|! (n - |S| - 1)!}{n!} [v(S \cup \{i\}) - v(S)], \quad \forall i \in N, \quad (3.2)$$

cuyo número de términos crece de forma exponencial. Tal y como se formalizó en la Definición 1.2.6, este valor admite una segunda formulación matemáticamente equivalente, aunque computacionalmente muy distinta, basada en un promedio sobre las permutaciones de entrada $\pi \in \Pi_N$, de manera similar a las reglas de Bird y Dutta-Kar:

$$Sh(N, v) = \frac{1}{n!} \sum_{\pi \in \Pi_N} m^\pi(v), \quad (3.3)$$

donde $m^\pi(v)$ representa el vector marginal asociado a $\pi \in \Pi_N$ dado por la Definición 1.2.5 y cuyo número de términos crece también de forma factorial, más rápido que el crecimiento exponencial de la Ecuación (3.2). Por tanto, la formulación basada en coaliciones es la empleada habitualmente para el cálculo exacto.

En esta sección presentamos, desde un punto de vista teórico, el método de Monte Carlo que permite aproximar ambos cálculos cuando el método exacto deja de ser viable, distinguiendo explícitamente los dos problemas. La idea general es que para las reglas basadas en el algoritmo de Prim se muestrean directamente las permutaciones y se promedia el reparto obtenido de cada una, mientras que para el

valor de Shapley se reemplaza la fórmula de la Ecuación (3.2) por la fórmula basada en permutaciones de la Ecuación (3.3) que admite un estimador por muestreo. De este modo, solo se evalúa v para las coaliciones que aparecen en las permutaciones muestreadas, siguiendo la técnica propuesta por Castro et al. (2009) para el cálculo aproximado del valor de Shapley.

El método de Monte Carlo

Para comenzar, recordemos el resultado general en el que se apoya cualquier estimador de Monte Carlo. Todos los resultados expuestos en este apartado se derivan del Capítulo 2 de Owen (2013).

Sea Y una variable aleatoria con $\mu = \mathbb{E}(Y)$ y sean $Y_1, \dots, Y_m$ variables iid con la misma distribución que Y . El estimador de Monte Carlo de μ es la media muestral

$$\hat{\mu}_m = \frac{1}{m} \sum_{k=1}^m Y_k.$$

Proposición 3.3.1 (Insensatez y convergencia). *El estimador $\hat{\mu}_m$ es insesgado, $\mathbb{E}(\hat{\mu}_m) = \mu$ y por la Ley Fuerte de los Grandes Números converge casi seguramente a μ cuando $m \rightarrow \infty$. Si además $\sigma^2 := \text{Var}(Y)$ es finita, la varianza del estimador es $\text{Var}(\hat{\mu}_m) = \sigma^2/m$, de modo que su error cuadrático medio decrece a la velocidad*

$$\text{RMSE}(\hat{\mu}_m) = \sqrt{\mathbb{E}[(\hat{\mu}_m - \mu)^2]} = \sqrt{\text{Var}(\hat{\mu}_m)} = \frac{\sigma}{\sqrt{m}} = \mathcal{O}(m^{-1/2}).$$

En virtud de la Proposición 3.3.1, el error de un estimador de Monte Carlo depende únicamente del número de simulaciones m y de la varianza σ^2 de la cantidad que se promedia, pero en ningún caso del tamaño del problema. Es por eso que podemos recurrir a este método en ambos problemas manteniendo las propiedades teóricas del estimador.

El estimador Monte Carlo de la regla extendida

Sea $f^\pi(N_0, C) \in \mathbb{R}^n$ la asignación de Bird o Dutta-Kar obtenida al resolver los empates de Prim según el orden $\pi \in \Pi_N$ y sea $f(N_0, C)$, dada por la Ecuación (3.1), la regla extendida exacta asociada. El método Monte Carlo sustituye esta media exacta, computacionalmente inviable para n grande, por el estimador que resulta de aplicar la Proposición 3.3.1 con $Y_k = f^{\pi_k}(N_0, C)$, $\pi_k \sim \text{Unif}(\Pi_N)$:

$$\hat{f}_{nsim}(N_0, C) = \frac{1}{nsim} \sum_{k=1}^{nsim} f^{\pi_k}(N_0, C), \quad \pi_1, \dots, \pi_{nsim} \stackrel{iid}{\sim} \text{Unif}(\Pi_N). \quad (3.4)$$

En términos computacionales, calcularlo consiste en elegir una permutación π_k al azar, resolver el algoritmo de Prim con esa permutación para deshacer los empates y almacenar el resultado para promediarlo, repitiendo el proceso $nsim$ veces. Camerini (1978) señala que resolver el algoritmo de Prim sobre un grafo completo tiene un coste de $\mathcal{O}(n^2)$. Dado que esto se repite $nsim$ veces, el coste computacional sería $\mathcal{O}(nsim \cdot n^2)$, frente a $\mathcal{O}(n! \cdot n^2)$ que exige recorrer las $n!$ permutaciones del método exacto.

Por la Proposición 3.3.1, $\hat{f}_{nsim}(N_0, C)$ es insesgado y converge casi seguramente a $f(N_0, C)$, con un error que decrece a velocidad $\mathcal{O}(nsim^{-1/2})$, siendo $\sigma_i^2 = \text{Var}_\pi(f_i^\pi(N_0, C))$ la varianza para cada agente i (finita, ya que Π_N es un conjunto finito). Esta tasa se contrastará empíricamente en la Sección 5.2.

El estimador Monte Carlo del valor de Shapley

Sea $Sh(N, v)$ la fórmula exacta del valor de Shapley basada en permutaciones dada por la Ecuación (3.3). Dicha ecuación muestra que $Sh(N, v) = \mathbb{E}_\pi[m^\pi(v)]$, con $\pi \sim \text{Unif}(\Pi_N)$. El estimador de Monte

Carlo del valor de Shapley, propuesto por Castro et al. (2009), aplica de nuevo la Proposición 3.3.1, esta vez con $Y_k = m^{\pi_k}(v)$, $\pi_k \sim \text{Unif}(\Pi_N)$:

$$\widehat{Sh}_{nsim}(N, v) = \frac{1}{nsim} \sum_{k=1}^{nsim} m^{\pi_k}(v), \quad \pi_1, \dots, \pi_{nsim} \stackrel{iid}{\sim} \text{Unif}(\Pi_N). \quad (3.5)$$

A diferencia de Bird o Dutta-Kar, aquí cada permutación π_k muestreada requiere evaluar la función característica en las n coaliciones que se van formando a medida que los agentes entran uno a uno según ese orden. De este modo, el algoritmo realiza $\mathcal{O}(nsim \cdot n)$ evaluaciones de la función característica, frente a las $2^n - 1$ evaluaciones exigidas por la Ecuación (3.2), evitando calcular coaliciones que nunca llegan a utilizarse.

Por la misma Proposición 3.3.1, $\widehat{Sh}_{nsim}(N, v)$ hereda las propiedades de insesgadez, convergencia casi segura y velocidad $\mathcal{O}(nsim^{-1/2})$. En particular, Castro et al. (2009) proporcionan una fórmula explícita para su varianza.

Capítulo 4

El paquete `mcstprules`: implementación y aplicación

La teoría explicada a lo largo de los capítulos anteriores se materializa ahora en una herramienta informática accesible, fácil de utilizar e implementada en software libre. Este capítulo detalla el diseño, la programación y el uso práctico de `mcstprules` (Souto Vázquez 2026), un nuevo paquete desarrollado en el entorno estadístico R (R Core Team 2026). El objetivo principal de este software es integrar dentro de un mismo ecosistema los diferentes métodos existentes para la resolución del problema de reparto de costes en árboles de mínimo coste, conectando directamente con la teoría vista en los Capítulos 2 y 3.

Hasta el momento la comunidad de R contaba con algunas opciones previas para abordar este tipo de problemas gracias a paquetes como `optrees` (Fontenla 2014b) y `cooptrees` (Fontenla 2014a), descritos en Fontenla Cadavid (2014). Aunque supusieron un buen punto de partida, sus propios autores señalaron ciertas limitaciones. En particular, no se contemplaba la resolución equitativa de los empates cuando la red admitía múltiples árboles óptimos y el coste computacional del valor de Shapley resultaba inviable para redes de más de diez agentes. Además de carecer de flexibilidad en la entrada de datos, las representaciones gráficas estaban limitadas a un máximo de tres agentes. Por otro lado, no se incluía la totalidad de los juegos asociados al MCSTP descritos en la revisión de Bergantiños y Vidal-Puga (2021), dejando fuera enfoques como el juego irreducible, el optimista, el público o el ciclo-completo, así como el análisis de la estabilidad del núcleo.

El paquete `mcstprules` nace desde cero precisamente para superar estas limitaciones y reunir en una única herramienta todas las funcionalidades necesarias. Tomando como base teórica la revisión de Bergantiños y Vidal-Puga (2021), el software implementa soluciones exactas y añade aproximaciones eficientes mediante el método de Monte Carlo. Esto permite, mediante muestreos aleatorios, por un lado, promediar las distintas soluciones que surgen al resolver los empates en el algoritmo de Prim y, por otro, estimar el valor de Shapley en redes con un gran número de agentes. Además, se incorporan las cinco funciones características del MCSTP junto con sus reglas de reparto (valor de Shapley y nucleolo).

Además de calcular los repartos, el paquete incluye herramientas para analizar la estabilidad. Sus funciones permiten comprobar si una asignación pertenece al núcleo del juego cooperativo, calcular sus vértices y crear gráficos interactivos en dos y tres dimensiones. También permite realizar análisis de sensibilidad ante variaciones en los costes y determinar los rangos de estabilidad de las conexiones que componen la red, un problema clásico en el estudio de árboles de mínimo coste (Tarjan 1982). Finalmente, para facilitar su uso, acepta múltiples formatos de entrada de datos: matrices de adyacencia,

listas de aristas, vectores numéricos, grafos e incluso archivos externos.

El código fuente es público y está alojado en un repositorio de GitHub: <https://github.com/lucasouto13/mcstprules>. La Figura 4.1 muestra el aspecto del repositorio en el momento de la entrega de este trabajo. El paquete se puede instalar directamente al entorno de trabajo local mediante las siguientes instrucciones:

```
# install.packages("remotes")
remotes::install_github("lucasouto13/mcstprules")
library(mcstprules)
```

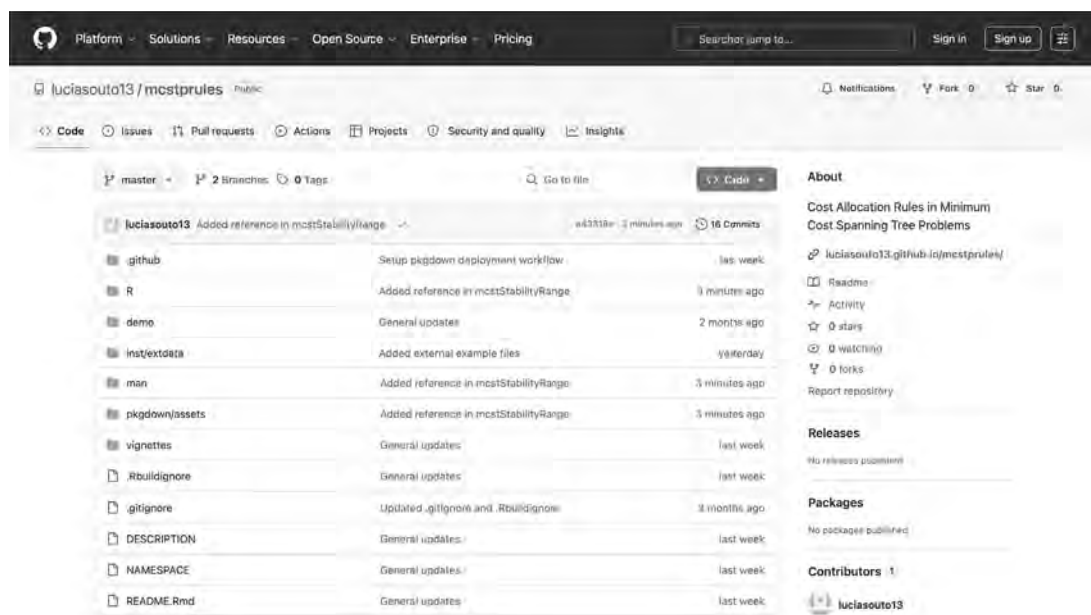


Figura 4.1: Página del repositorio de mcstprules en GitHub.

A lo largo de las siguientes secciones ilustraremos el funcionamiento del paquete mediante el Ejemplo 2.1.1, ya empleado en los capítulos anteriores. En R, se define introduciendo su matriz de costes, asumiendo que la primera fila y columna representan al nodo fuente. Comenzaremos analizando la arquitectura interna del paquete, detallando cómo se organizan sus archivos, dependencias y estructura. Por último, cabe destacar que en el Anexo A se incluye el manual técnico completo con la descripción de todas las funciones y argumentos de mcstprules.

4.1. Arquitectura y diseño del paquete

El diseño interno de mcstprules está orientado a facilitar su uso, ampliación y mantenimiento. Al tratarse de una herramienta con un número considerable de funciones, era importante que el usuario final trabajara con comandos claros y sencillos, dejando que el código interno se encargara de toda la complejidad de los cálculos. Por eso el paquete se ha organizado de forma modular, separando claramente los algoritmos clásicos, los juegos cooperativos y las herramientas de análisis y visualización.

La Tabla 4.1 recoge todas las funciones públicas del paquete, agrupadas en las tres familias que se explican a lo largo de este capítulo.

Familia	Función	Descripción
<i>Reglas algorítmicas</i>	<code>mcstBird</code>	Regla de Bird (vía Prim)
	<code>mcstDuttaKar</code>	Regla de Dutta-Kar (vía Prim)
	<code>mcstFolk</code>	Regla folk (vía Kruskal)
	<code>mcstOWShapley</code>	Shapley ponderado optimista (vía Kruskal)
	<code>mcstPWShapley</code>	Shapley ponderado pesimista (vía Kruskal)
	<code>mcstBoruvka</code>	Regla de Borůvka
	<code>mcstCone</code>	Descomposición en conos
<i>Juegos cooperativos</i>	<code>mcstGamePrivate</code>	Juego privado v^p con Shapley o nucleolo
	<code>mcstKar</code>	Regla de Kar (acceso directo a $Sh(v^p)$)
	<code>mcstGameIrred</code>	Juego irreducible v^i con Shapley o nucleolo
	<code>mcstGameOpt</code>	Juego optimista v^o con Shapley o nucleolo
	<code>mcstGamePublic</code>	Juego público v^u con Shapley o nucleolo
	<code>mcstGameCC</code>	Juego ciclo-completo v^c con Shapley o nucleolo
<i>Herramientas de análisis</i>	<code>mcstCompare</code>	Comparativa de reglas
	<code>mcstCoreCheck</code>	Verificación de pertenencia al núcleo
	<code>mcstCorePoints</code>	Existencia y vértices del núcleo
	<code>mcstCorePlot</code>	Visualización 2D o 3D del núcleo y los repartos
	<code>mcstStabilityRange</code>	Rango de estabilidad de los arcos
	<code>mcstSensitivity</code>	Análisis de sensibilidad de los repartos
	<code>mcstRules</code>	Guía/índice explicativo del paquete

Tabla 4.1: Funciones públicas del paquete `mcstprules`.

4.1.1. Estructura de archivos y dependencias

El paquete `mcstprules` sigue la estructura estándar de cualquier paquete de R, con la lógica organizada en varios archivos dentro del directorio R/ según su funcionalidad (reglas algorítmicas, juegos cooperativos, herramientas de análisis y funciones internas), además de los archivos habituales de documentación y metadatos. Este paquete también incluye otros recursos, como una página web y varias demostraciones de uso, que se describen con más detalle en la Sección 4.1.4.

Para su funcionamiento, `mcstprules` se apoya en varios paquetes ya existentes en R, en lugar de programar de nuevo procesos que ya están optimizados. La representación y manipulación de las redes se hace con `igraph` (Csárdi y Nepusz 2006; Antonov et al. 2023; Csárdi et al. 2026). El estudio del núcleo combina `lpSolve` (Berkelaar 2024), que resuelve el sistema de inecuaciones necesario para saber si el núcleo está vacío, con `rcdd` (Geyer et al. 2026), que calcula sus vértices de forma exacta. El cálculo del valor de Shapley y del nucleolo se hace con el paquete `CoopGame` (Staudacher y Anwander 2019). Para las soluciones exactas basadas en permutaciones se usa `gtools` (Warnes et al. 2023), y `readxl` (Wickham y Bryan 2023) permite leer directamente archivos de Excel como entrada de datos. Por último, las representaciones interactivas en tres dimensiones se generan con `plotly` (Sievert 2020), apoyado en `geometry` (Habel et al. 2025) para construir las superficies del núcleo, mientras que `fields` (Douglas Nychka et al. 2021) se emplea en el análisis de sensibilidad para construir los mapas de calor que se generan cuando dos arcos varían su coste de forma independiente.

4.1.2. Flexibilidad en la entrada de datos

A diferencia de los paquetes anteriores, que exigían formatos de entrada muy rígidos, todas las funciones de `mcstprules` aceptan múltiples formas de introducir los datos a través del argumento `C`. El usuario puede elegir entre cinco formatos distintos y, aunque todos representan la misma información matemática, cada uno se organiza de forma diferente. Esto permite una gran flexibilidad a la hora de trabajar, dependiendo de cómo se tengan guardados los datos originales o de los conocimientos de programación de cada usuario.

A lo largo de esta sección usaremos como referencia el grafo de costes de la Figura 2.1. Recordemos que el nodo 0 representa la fuente, los nodos 1, 2 y 3 representan los agentes y el peso de cada conexión indica el coste entre ellos. A continuación mostramos las distintas formas equivalentes de introducir este mismo problema en R.

Matriz de adyacencia

La forma más habitual de representar los costes es mediante una matriz de adyacencia $C = (c_{ij})_{i,j \in N_0}$, donde c_{ij} indica el coste entre los nodos i y j (el nodo 0 ocupa siempre la primera fila y columna). Como las conexiones son no dirigidas, la matriz resultante es simétrica, y los elementos de la diagonal principal valen cero, ya que representan el coste de conectar un nodo consigo mismo (Definición 2.1.1). Para nuestro ejemplo, y como ya vimos en el Capítulo 2, la matriz de costes asociada al problema es:

$$C = \begin{pmatrix} 0 & 10 & 6 & 6 \\ 10 & 0 & 5 & 3 \\ 6 & 5 & 0 & 3 \\ 6 & 3 & 3 & 0 \end{pmatrix}.$$

En R, esta matriz puede definirse como una matriz clásica o como un *data frame*, tal y como se muestra en la Figura 4.2. El paquete interpreta correctamente cualquiera de las dos opciones. El usuario puede crear directamente un *data frame* o bien definir primero una matriz y convertirla después empleando el comando `as.data.frame(C)`, o viceversa, con `as.matrix(C)`.

```
# Formato matricial clásico
C <- matrix(c(0, 10, 6, 6,
             10, 0, 5, 3,
             6, 5, 0, 3,
             6, 3, 3, 0),
            byrow = TRUE, ncol = 4)
```

```
# Formato data frame
C <- data.frame(c( 0, 10, 6, 6),
               c(10, 0, 5, 3),
               c( 6, 5, 0, 3),
               c( 6, 3, 3, 0))
```

Figura 4.2: Opciones base de R para crear la matriz de costes.

Lista de aristas

Otra forma muy práctica de representar la misma información es mediante una lista de aristas. En este formato, cada fila corresponde a una conexión directa entre dos nodos junto con su coste asociado. Para nuestro ejemplo, la estructura correspondiente se muestra en la Tabla 4.2.

from	to	cost
0	1	10
0	2	6
0	3	6
1	2	5
1	3	3
2	3	3

Tabla 4.2: Lista de aristas correspondiente al grafo de la Figura 2.1.

Esta alternativa es especialmente útil cuando la red es dispersa o cuando los datos provienen de una base de datos externa, ya que evita tener que guardar todas las conexiones nulas de una matriz completa. En R, esta representación puede introducirse nuevamente mediante una matriz o un *data frame*, ambos formatos con las columnas **from** (origen), **to** (destino) y una tercera columna con el coste, como se muestra en la Figura 4.3.

```
# Formato matricial clásico
C <- matrix(c(0, 1, 10,
             0, 2, 6,
             0, 3, 6,
             1, 2, 5,
             1, 3, 3,
             2, 3, 3),
           byrow = TRUE, ncol = 3,
           dimnames = list(
             NULL, c("from", "to",
                     "cost")
           ))
```

```
# Formato data frame
C <- data.frame(
  from = c(0, 0, 0, 1, 1, 2),
  to   = c(1, 2, 3, 2, 3, 3),
  cost = c(10, 6, 6, 5, 3, 3)
)
```

Figura 4.3: Opciones base de R para crear la lista de aristas.

Vector correspondiente al triángulo inferior

Cuando la matriz de costes es simétrica, guardar a la vez la parte superior y la inferior implica duplicar la información. Por eso, el paquete también permite especificar únicamente un vector con los valores del triángulo inferior de la matriz, sin incluir la diagonal. Este vector se obtiene leyendo los elementos por debajo de la diagonal, columna a columna, como se muestra en el siguiente esquema:

$$\begin{pmatrix} 0 & 10 & 6 & 6 \\ 10 & 0 & 5 & 3 \\ 6 & 5 & 0 & 3 \\ 6 & 3 & 3 & 0 \end{pmatrix} \xrightarrow{\text{Lectura por columnas}} \begin{pmatrix} 0 & 10 & 6 & 6 \\ 10 & 1 & 0 & 5 & 3 \\ 6 & 2 & 5 & 4 & 0 & 3 \\ 6 & 3 & 3 & 5 & 3 & 6 & 0 \end{pmatrix}$$

Tomando la matriz de costes anterior y recorriendo el triángulo inferior por columnas se obtiene el vector:

$$C = (10, 6, 6, 5, 3, 3).$$

En R, este formato es muy sencillo de introducir, lo que reduce mucho el espacio de memoria necesario sin perder ninguna información:

```
# Vector correspondiente al triángulo inferior de la matriz de adyacencia
C <- c(10, 6, 6, 5, 3, 3)
```

Objeto igraph

Para quienes ya tienen experiencia analizando redes, el paquete también acepta directamente objetos de la librería **igraph**. En este caso, la estructura de costes se guarda en un objeto donde las aristas representan las conexiones entre nodos y los pesos representan los costes asociados.

El grafo de nuestro ejemplo puede crearse con cualquiera de las tres opciones presentadas en la Figura 4.4. Una vez creado, el objeto puede emplearse directamente en cualquier función del paquete que acepte una matriz de costes como argumento.

```
# Cargar librería
library(igraph)
```

```
# A partir de data frame
C <- graph_from_data_frame(
  data.frame(
    from = c(0, 0, 0, 1, 1, 2),
    to   = c(1, 2, 3, 2, 3, 3),
    weight = c(10, 6, 6, 5, 3, 3)
  ), directed = FALSE
)
```

```
# A partir de lista de aristas
C <- graph_from_edgelist(
  matrix(as.character(c(
    0,1, 0,2, 0,3,
    1,2, 1,3, 2,3
  )), ncol = 2, byrow = TRUE),
  directed = FALSE)
E(C)$weight <- c(10, 6, 6, 5, 3, 3)
```

```
# A partir de matriz de adyacencia
C <- graph_from_adjacency_matrix(
  matrix(c(0, 10, 6, 6,
           10, 0, 5, 3,
           6, 5, 0, 3,
           6, 3, 3, 0),
        nrow = 4, byrow = TRUE),
  mode = "undirected", weighted = TRUE)
```

Figura 4.4: Opciones en R para crear el grafo mediante el paquete **igraph**.

Lectura desde fichero

Por último, los costes también pueden proporcionarse mediante un archivo externo. Esta opción es la más cómoda para usuarios sin conocimientos de programación, ya que solo hace falta indicar la ruta del archivo guardado en el equipo.

La función admite archivos de Excel y archivos de texto separados por comas, con extensiones **.xls**, **.xlsx** y **.csv**. Como se observa en la Figura 4.5, estos archivos pueden contener tanto una matriz cuadrada sin encabezados como una lista de aristas con las columnas **from**, **to** y el coste asociado.



Figure 4.5(a) shows two Excel spreadsheets. The left spreadsheet, titled 'matrix', displays an adjacency matrix with nodes 1 through 6. The right spreadsheet, titled 'edges', displays a list of edges with columns 'from', 'to', and 'cost'.

	A	B	C	D	E	F
1	0	10	6	6		
2	10	0	5	3		
3	6	5	0	3		
4	6	3	3	0		
5						
6						
7						
8						

	A	B	C	D	E	F
1	from	to	cost			
2	0	1	10			
3	0	2	6			
4	0	3	6			
5	1	2	5			
6	1	3	3			
7	2	3	3			
8						

(a) Estructuras en formato Excel (.xlsx/.xls): matriz de adyacencia (izquierda) y lista de aristas (derecha).

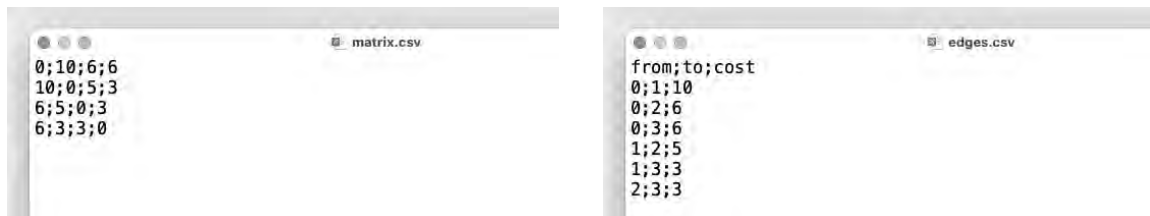


Figure 4.5(b) shows two CSV files. The left file, titled 'matrix.csv', contains the adjacency matrix data. The right file, titled 'edges.csv', contains the edge list data.

	A	B	C	D	E	F
1	0	10	6	6		
2	10	0	5	3		
3	6	5	0	3		
4	6	3	3	0		
5						
6						
7						
8						

	A	B	C	D	E	F
1	from	to	cost			
2	0	1	10			
3	0	2	6			
4	0	3	6			
5	1	2	5			
6	1	3	3			
7	2	3	3			
8						

(b) Estructuras en formato CSV (.csv): matriz de adyacencia (izquierda) y lista de aristas (derecha).

Figura 4.5: Formatos de archivos admitidos por la función `mcstprules` para la creación del grafo.

La Figura 4.6 muestra cómo se implementaría la lectura en R en el caso de que el usuario utilice un archivo propio guardado en su equipo (suponiendo, a modo de ejemplo, que el fichero se llama “edges.xlsx” o “edges.csv”):

```
# Formato Excel
library(readxl) # Cargar librería
C <- "ruta/a/edges.xlsx"
```

```
# Formato CSV
C <- "ruta/a/edges.csv"
```

Figura 4.6: Código en R para la lectura automática de archivos externos.

No obstante, para facilitar las pruebas, los cuatro ficheros de ejemplo mencionados en la Figura 4.5 se han incluido internamente en el paquete y también se encuentran alojados en el repositorio de GitHub. Esto ofrece al usuario dos alternativas adicionales para probar las funciones. Por un lado, puede descargar los archivos desde GitHub directamente a su directorio de trabajo local. Por otro, si ya ha instalado el paquete, puede obtener la ruta directa de estos ejemplos de prueba empleando la función `system.file`. Para ello, basta con ejecutar el siguiente comando, pudiendo sustituir “edges.csv” por el nombre de cualquiera de los otros tres archivos:

```
# Cargar datos de prueba incluidos en el paquete
C <- system.file("extdata", "edges.csv", package = "mcstprules")
```

Por último, cabe destacar que, en aquellos formatos que lo requieran, el usuario puede usar el valor especial `Inf` para representar conexiones inexistentes, es decir, nodos que no están conectados

directamente entre sí, lo que permite trabajar con redes “incompletas”.

4.1.3. Tratamiento de empates y soluciones aproximadas

Como ya comentamos en el Capítulo 3, cuando el árbol de mínimo coste no es único, las reglas basadas en el algoritmo de Prim dependen del orden en el que se van conectando los agentes. Las funciones del paquete `mcstBird` y `mcstDuttaKar` detectan automáticamente esta situación y recurren, de forma intrínseca y sin que el usuario deba intervenir, al cálculo de la regla extendida expuesto en la Ecuación (3.1).

Como se explicó en la Sección 3.3, el cálculo exacto de esta media se vuelve rápidamente inviable a medida que aumenta el tamaño de la red debido a su crecimiento factorial. Por este motivo, tanto `mcstBird` como `mcstDuttaKar` incorporan un argumento llamado `method` que permite al usuario elegir cómo calcular esta media:

- `method = “exact”` (opción por defecto): calcula las $n!$ permutaciones del conjunto de agentes y promedia el resultado obtenido en cada una, devolviendo el valor exacto de la regla extendida siguiendo la Ecuación (3.1). A lo largo de este capítulo emplearemos esta opción junto con nuestro ejemplo, ya que permite ilustrar con claridad el funcionamiento de cada regla.
- `method = “montecarlo”`: en lugar de recorrer todas las permutaciones, estima la media mediante el estimador presentado en la Ecuación (3.4), a partir de un número determinado de permutaciones elegidas al azar (`nsim`, fijado en 10 000 por defecto). Esta aproximación ofrece resultados muy precisos en una fracción del tiempo que exige el método exacto, por lo que resulta la opción recomendada cuando el número de agentes es grande.

Como se explicó en la Sección 3.3, reglas como la folk o la basada en el algoritmo de Borůvka devuelven siempre una asignación única, por lo que sus funciones correspondientes no incluyen el argumento `method`.

El mismo argumento `method` aparece también en las seis funciones que calculan juegos cooperativos, aunque en este caso resuelve el problema del crecimiento exponencial en la definición del valor de Shapley explicado en la Sección 3.3. Para `method = “exact”`, el cálculo se realiza a través de la Ecuación (3.2), mediante la función `shapleyValue` del paquete `CoopGame` (Staudacher y Anwander 2019), que obliga a evaluar v para las $2^n - 1$ coaliciones no vacías. La opción `method = “montecarlo”` evita este problema aproximando el valor de Shapley mediante el muestreo de `nsim` permutaciones aleatorias, siguiendo el resultado expuesto en la Ecuación (3.5), sin necesidad de construir la función característica completa.

Dado que esta elección entre el método exacto y la aproximación de Monte Carlo tiene un impacto muy importante en el tiempo de cómputo, tanto en la extensión de las reglas de Bird y Dutta-Kar como en la aproximación del valor de Shapley, el Capítulo 5 se dedica por completo a comprobar empíricamente los límites del método exacto y la velocidad y fiabilidad de estas aproximaciones.

4.1.4. Recursos adicionales

Además del código y su documentación, el paquete incluye varios recursos que ayudan a empezar a usarlo sin dificultad.

En primer lugar, la carpeta `demo/` incluye tres demostraciones ejecutables desde la consola de R. La demo `demo(basic_rules)` plantea un problema de ejemplo y muestra el cálculo y la comparación de las principales reglas, tanto algorítmicas como basadas en juegos cooperativos, incluyendo su representación gráfica y el tratamiento de los empates. La demo `demo(analysis_tools)` presenta las herramientas de análisis mediante el cálculo del juego privado, la comprobación de pertenencia al núcleo, la obtención de sus vértices y un análisis de sensibilidad. Por último, `demo(scalability)` genera una red de veinte agentes y muestra cómo el método de Monte Carlo permite aproximar la regla de

Bird y el reparto de Kar con un coste computacional muy inferior al método exacto, adelantando así el análisis de rendimiento del Capítulo 5.

Como punto de partida, la función `mcstRules` imprime en consola una guía con todas las reglas algorítmicas, los juegos cooperativos y las herramientas disponibles. Se puede consultar en cualquier momento con `?mcstRules`.

Por último, toda la documentación técnica de cada función está publicada en la página web del proyecto (<https://luciasouto13.github.io/mcstpules/>), generada automáticamente con `pkgdown`¹ (Wickham et al. 2025) a partir del código fuente. Esta generación se automatiza mediante un flujo de trabajo de GitHub Actions, cuya configuración se encuentra en la carpeta `.github/`, que reconstruye y publica la página web cada vez que se actualiza el repositorio. En dicha página se puede consultar, sin salir del navegador, la descripción teórica de cada método, sus argumentos y ejemplos de uso. La Figura 4.7 muestra su página de inicio.

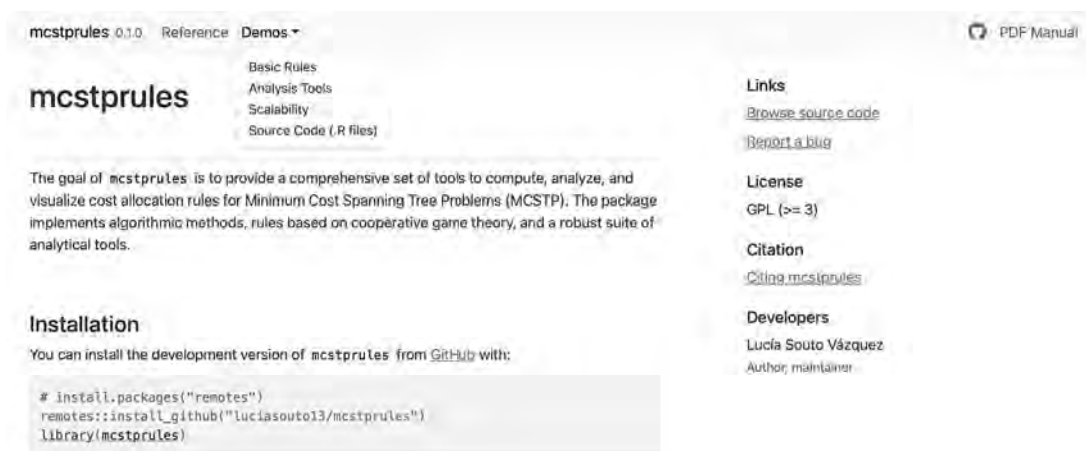


Figura 4.7: Interfaz de la página web oficial del paquete con la documentación completa.

Finalmente, cabe destacar que el paquete ya cuenta con la carpeta `vignettes/`, pensada para incorporar viñetas que sirvan como tutoriales autocontenidos. Por el momento, sin embargo, su elaboración queda pendiente para futuras versiones del paquete.

4.2. Reglas definidas a través del MCSTP

Las funciones de esta sección implementan las reglas de reparto obtenidas a partir de los algoritmos de construcción de árboles vistos en el Capítulo 2. Todas ellas aceptan la matriz de costes mediante el argumento `C` en cualquiera de los formatos descritos en la Sección 4.1.2.

El objetivo de este capítulo no es volver a justificar los resultados de las reglas explicadas, sino mostrar cómo se utilizan en la práctica y de qué información dispone el usuario para analizar cada problema. Para ello, emplearemos el problema presentado en el Ejemplo 2.1.1. Cabe destacar que trabajaremos bajo el método exacto (`method = "exact"`) para el cálculo de los repartos extendidos de las reglas de Bird y Dutta-Kar. La comparación con el método Monte Carlo se abordará en el Capítulo 5.

La descripción completa de todos los argumentos, componentes del objeto devuelto, etc. puede consultarse tanto desde la ayuda de R (por ejemplo, mediante `?mcstBird`) como en la documentación del

¹`pkgdown` es un paquete de R que construye automáticamente una página web con la documentación de un paquete a partir de sus archivos de ayuda y sus metadatos.

paquete incluida en el Anexo A.

4.2.1. Funciones `mcstBird` y `mcstDuttaKar`

Recordemos que estas dos funciones implementan reglas de reparto basadas en la construcción de un árbol de coste mínimo mediante el algoritmo de Prim. En particular, la función `mcstBird` calcula la regla propuesta por Bird (1976) y presenta la siguiente estructura de uso:

Uso

```
mcstBird(C, draw = FALSE, which.plot = c("main", "details"), titles = TRUE,
        method = c("exact", "montecarlo"), nsim = 10000)
```

Al activar `draw = TRUE`, la función representa gráficamente el árbol obtenido, indicando entre corchetes la etapa en la que se incorpora cada arco en el algoritmo de Prim y mostrando dentro de cada nodo el coste finalmente asignado a cada agente. En problemas con menos de cuatro agentes, cuando existen varios árboles óptimos, el argumento `which.plot` ofrece dos representaciones, tal y como veremos a continuación con nuestro ejemplo. El argumento `titles` es un argumento lógico que, por defecto, añade automáticamente títulos informativos, especificando la regla de asignación, algoritmo, método y coste total de la red. Se trata de una opción puramente estética, pensada para poder desactivarse (`titles = FALSE`) si las figuras van a ser integradas en otros proyectos o si el usuario prefiere una visualización más limpia, con menos información en pantalla. Estos argumentos actúan de la misma forma para todas las funciones que veremos a lo largo de la Sección 4.2.

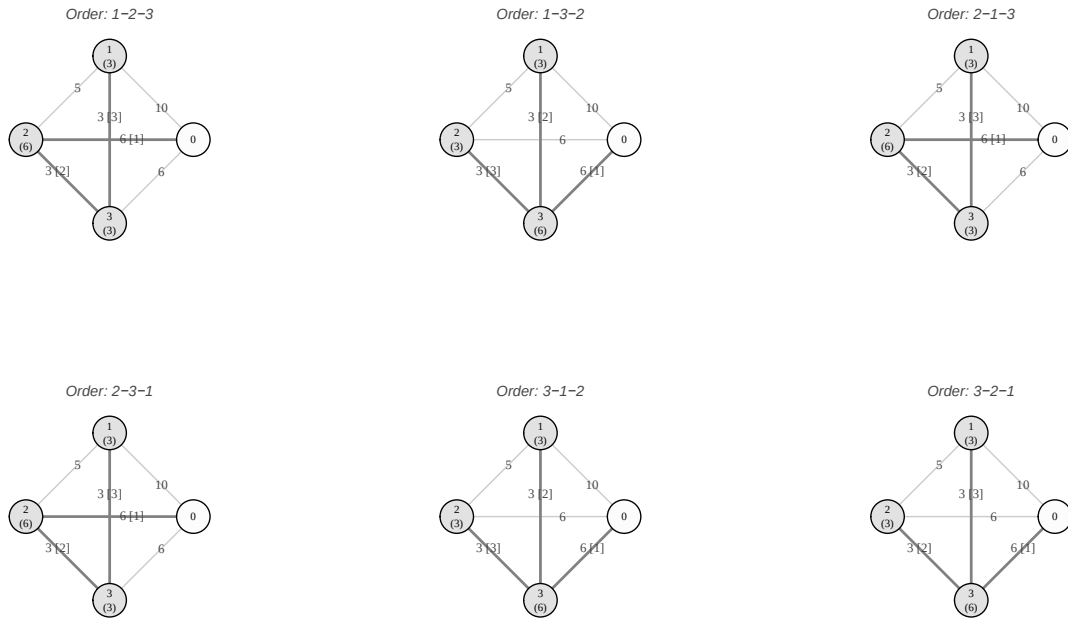
Como ya se vio en el Capítulo 3, nuestro problema presenta un empate en la primera etapa del algoritmo de Prim, ya que los agentes 2 y 3 pueden conectarse inicialmente a la fuente con el mismo coste. En consecuencia, el árbol óptimo no es único y la función lo detecta automáticamente, sin que el usuario deba intervenir en ningún momento. Dado que el problema abordado es un problema con tres agentes, `mcstBird` ofrece dos representaciones complementarias: un gráfico que muestra los árboles generados y el reparto para todas las permutaciones y un gráfico principal que resume el reparto extendido obtenido tras promediarlas y muestra el árbol óptimo correspondiente a la ordenación natural. En R, podemos escribir la siguiente instrucción:

```
# Regla de Bird
bird <- mcstBird(C, draw = TRUE, titles = FALSE)
```

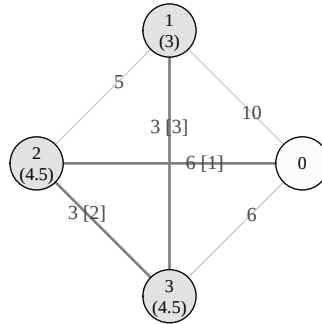
La Figura 4.8a muestra, permutación a permutación, los diferentes árboles óptimos obtenidos al realizar el desempate en la primera etapa de Prim siguiendo las distintas ordenaciones. Fijémonos que de las $3! = 6$ permutaciones posibles solo surgen dos árboles distintos, que difieren únicamente en el primer arco elegido desde la fuente, (0, 2) o (0, 3), dando lugar a dos repartos diferentes: (3, 6, 3) y (3, 3, 6). Comparando estos gráficos con los resultados obtenidos de forma analítica en el Ejemplo 3.1.1, podemos ver cómo la función considera correctamente todas las alternativas de desempate siguiendo al pie de la letra la definición propuesta en el Capítulo 3.

Por su parte, la Figura 4.8b presenta directamente, dentro de cada nodo del grafo, el resultado final del reparto que le corresponde a cada agente una vez promediados los repartos que se muestran en la Figura 4.8a.

Por defecto, para redes con menos de cuatro agentes, la función devuelve ambos gráficos, cuya salida se puede controlar mediante el argumento `which.plot`. Para redes de mayor tamaño, puesto que el número de permutaciones crece de forma factorial, no resulta viable mostrar todas las posibilidades y únicamente se devuelve la figura correspondiente al reparto final.



(a) Árboles óptimos obtenidos para todas las permutaciones con sus respectivos repartos.



(b) Árbol óptimo correspondiente al orden natural con el reparto extendido.

Figura 4.8: Representaciones generadas por `mcstBird` cuando existen varios árboles óptimos.

Además de la representación gráfica, al ejecutar la función en la consola de R, se imprime automáticamente, si es el caso, un mensaje que advierte de la falta de unicidad del árbol óptimo y muestra, en un *data frame*, el reparto correspondiente a la ordenación natural y el reparto extendido, facilitando su comparación directa:

```
bird # B(N0,C)

## Non-unique MCST detected

##      1  2  3
## B(id) 3 6.0 3.0
## E[B(pi)] 3 4.5 4.5
```

Además de lo que se imprime por defecto, el objeto devuelto por `mcstBird` permite acceder a mucha más información sobre el problema. Esta se resume brevemente en la Tabla 4.3. La descripción detallada de cada campo puede consultarse en la documentación de la función.

Campo	Tipo	Descripción
<code>\$bird</code>	vector numérico	Reparto de Bird para el orden natural
<code>\$e_bird</code>	vector numérico	Reparto de Bird extendido
<code>\$arcs</code>	data frame / lista	Arcos del árbol óptimo; lista de árboles si hay empates
<code>\$total</code>	escalar	Coste total del árbol
<code>\$percentage</code>	vector numérico	Porcentaje del coste asignado a cada agente
<code>\$ranking</code>	vector carácter	Orden de los agentes según su pago
<code>\$perms</code>	lista	Información sobre las permutaciones si hay empates
<code>\$is_unique</code>	lógico	Indica si el árbol óptimo es único
<code>\$method</code>	carácter	Método empleado para el cálculo

Tabla 4.3: Campos devueltos por el objeto resultado de la función `mcstBird`.

Esta información resulta especialmente útil en ejemplos con un mayor número de agentes, en los que obtener estos campos de un vistazo no es evidente. Por ejemplo, `$ranking` permite identificar rápidamente a qué agentes les corresponde un mayor pago o cuál es el que menos paga. Además, en caso de pagar lo mismo, la asignación de esos agentes se marca con un asterisco:

```
bird$ranking # Orden de los agentes según su pago
```

```
## [1] 2* 3* 1
```

En este caso, los agentes 2 y 3 aparecen marcados porque asumen exactamente el mismo coste de 4.5 unidades. Por su parte, el campo `$is_unique` nos dice si el árbol óptimo es único para el problema considerado. En estas funciones, que calculan explícitamente el promedio cuando existen empates, esta información puede resultar menos relevante, puesto que el hecho de recurrir a las permutaciones indica implícitamente que el árbol óptimo no es único. Sin embargo, sí puede resultar interesante en las reglas presentadas en las secciones posteriores, donde el reparto es independiente del camino elegido para construir el árbol óptimo.

```
bird$is_unique # Unicidad del árbol óptimo
```

```
## [1] FALSE
```

La función `mcstDuttaKar`, que calcula la regla propuesta por Dutta y Kar (2004), comparte la misma interfaz que `mcstBird`, ya que recibe los mismos argumentos y puede utilizarse de forma análoga. Además, la función devuelve un objeto con estructura idéntica, sustituyendo únicamente los campos `$bird` y `$e_bird` por `$dk` y `$e_dk`. Como ya vimos en el Capítulo 3, la única diferencia entre esta función y la anterior está en la lógica interna de reparto, por lo que el árbol construido es el mismo que en el caso de Bird. Así, las figuras generadas con `draw = TRUE` solo se diferencian en el coste asignado que se muestra dentro de cada nodo y, por tanto, no las repetiremos aquí. Nos centraremos en consultar otra información adicional en base a los campos vistos en la Tabla 4.3.

```
# Regla de Dutta-Kar
dk <- mcstDuttaKar(C)
dk$e_dk # DK(N0,C)
```

```
## 1 2 3
## 5 4 3
```

Cabe destacar el correcto funcionamiento tanto de esta función como de la anterior, cuyos resultados coinciden exactamente con los obtenidos en la Sección 4.2. Además, estas funciones permiten explorar qué arcos han sido seleccionados en cada etapa del algoritmo de Prim y, en caso de empates, devuelven una lista con los distintos árboles óptimos:

```
dk$arcs # Lista con los árboles óptimos obtenidos
```

```
## [[1]]
##   i j stage
## 1 0 2     1
## 2 2 3     2
## 3 3 1     3
##
## [[2]]
##   i j stage
## 1 0 3     1
## 2 3 1     2
## 3 3 2     3
```

El campo `$arcs` identifica explícitamente los dos árboles óptimos generados por el empate inicial, detallando en cada uno el arco de conexión y la etapa en la que se incorporó.

En caso de empates, también es posible obtener información detallada acerca de las permutaciones a través de la componente `$perms`, que contiene una lista con diversos cálculos asociados a los distintos repartos. Por ejemplo, `$perms$summary` resume el resultado de todas las permutaciones evaluadas mostrando únicamente los repartos que resultan distintos y la multiplicidad con la que se producen.

```
dk$perms$summary # Resumen de los repartos obtenidos
```

```
##   1 2 3 times
## 1 6 3 3     4
## 2 3 6 3     2
```

En este caso, existen únicamente dos repartos distintos: cuatro permutaciones conducen a (6,3,3) y dos a (3,6,3). Además, para problemas con menos de cuatro agentes, `$perms$table` muestra en un *data frame* el resultado derivado de cada permutación. Sin embargo, en problemas más grandes, la salida anterior resulta mucho más informativa, ya que una tabla con todas las permutaciones dejaría de ser manejable.

Como comprobación adicional, podemos ver que los repartos obtenidos verifican la propiedad de eficiencia a través del campo `$total`. Nuevamente, esta componente será útil en problemas de mayor tamaño, donde la comprobación de la suma no sea tan directa.

```
dk$total # Coste total
```

```
## [1] 12
```

4.2.2. Funciones `mcstFolk`, `mcstOWShapley` y `mcstPWShapley`

Estas tres funciones implementan reglas de reparto basadas en el algoritmo de Kruskal, que construye el árbol de coste mínimo fusionando progresivamente componentes conexas a través de los arcos más baratos disponibles en cada momento. La función `mcstFolk` calcula la regla folk propuesta por Feltkamp et al. (1994):

Uso

```
mcstFolk(C, draw = FALSE, titles = TRUE)
```

A simple vista, podemos notar ciertas diferencias en los argumentos que admite esta función con respecto a las explicadas en el apartado previo. Tal y como mencionamos en la Sección 3.1, el reparto folk no depende del orden en el que Kruskal maneje los empates entre arcos, ya que cualquier ordenación conduce siempre al mismo resultado. Esto simplifica mucho su uso, puesto que no hay que preocuparse por la posible existencia de árboles óptimos alternativos. Es por eso que tanto esta función como las que veremos a continuación no incluyen el argumento `method`.

```
# Regla folk
folk <- mcstFolk(C, draw = TRUE, titles = FALSE)
folk$folk # F(N0,C)
```

```
## 1 2 3
## 4 4 4
```

El reparto obtenido coincide con el que ya calculamos en el Ejemplo 3.1.3. El objeto devuelto por `mcstFolk` sigue la misma filosofía que los anteriores y sus componentes se resumen en la Tabla 4.4.

Campo	Tipo	Descripción
<code>\$folk</code>	vector numérico	Reparto folk para cada agente
<code>\$arcs</code>	data frame	Arcos del árbol óptimo
<code>\$total</code>	escalar	Coste total del árbol
<code>\$percentage</code>	vector numérico	Porcentaje del coste asignado a cada agente
<code>\$ranking</code>	vector carácter	Orden de los agentes según su pago
<code>\$is_unique</code>	lógico	Indica si el árbol óptimo es único

Tabla 4.4: Campos devueltos por el objeto resultado de la función `mcstFolk`.

Fijémonos que, a diferencia de `mcstBird` y `mcstDuttaKar`, aquí no existen los campos `$e_folk` ni `$perms`, ya que al no verse afectada por los empates, la regla folk no necesita una versión extendida ni un registro de permutaciones. Por otro lado, el campo `$arcs` sigue estando disponible para consultar el árbol concreto sobre el que se ha calculado el reparto:

```
folk$arcs # Arcos del árbol óptimo
```

```
## i j stage
## 1 1 3 1
## 2 2 3 2
## 3 0 2 3
```

A pesar de que el árbol óptimo coincide con uno de los obtenidos anteriormente, la etapas de inclusión de cada arco ya no son las mismas dado que la estrategia de los algoritmos es diferente. La Figura 4.9 recoge la visualización obtenida con `draw = TRUE`, mostrando el árbol de coste mínimo obtenido mediante Kruskal junto con el reparto asignado a cada agente.

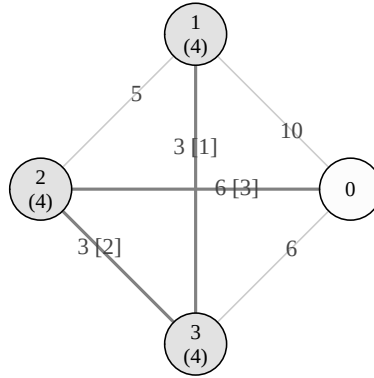


Figura 4.9: Representación del árbol óptimo de Kruskal generado por `mcstFolk` con el reparto asignado a cada agente entre paréntesis.

Por su parte, `mcstOWShapley` y `mcstPWSshapley` generalizan la regla folk introduciendo un sistema de pesos no negativos $w = (w_i)_{i \in N}$, que permite modelar asimetrías entre los agentes, ya sea por tamaño, poder de negociación o cualquier otro criterio relevante para el problema. La primera implementa la versión optimista propuesta por Bergantiños y Lorenzo-Freire (2008a,b), mientras que la segunda recoge la versión pesimista de Lorenzo y Lorenzo-Freire (2009):

Uso

```
mcst*WShapley(C, weights, draw = FALSE, titles = TRUE)
```

El `*` se sustituye por una “O” para calcular la versión optimista de Shapley o por una “P” para la pesimista, ambas letras en mayúsculas. Fijémonos que la única diferencia con la regla folk es que estas admiten el argumento `weights`, que acepta cualquier vector de números positivos de longitud igual al número de agentes. Por ejemplo, para otorgar el doble de peso al agente 2 que a los agentes 1 y 3 escribimos lo siguiente:

```
w <- c(1, 2, 1) # Agente 2 con doble peso
```

Ambas funciones devuelven un objeto con la misma estructura que `mcstFolk` (Tabla 4.4), sustituyendo el campo `$folk` por `$ows` o `$pws` según corresponda e incorporando el propio vector `$weights` empleado en el cálculo, útil para dejar constancia de qué ponderación se ha utilizado en cada análisis.

```
# Reglas Shapley ponderado
mcstOWShapley(C, weights = w) # Optimista
```

```
## Weights: 1 2 1
##      1      2      3
## 3.75 4.50 3.75
```

```
mcstPWShapley(C, weights = w) # Pesimista

## Weights: 1 2 1
##      1      2      3
## 4.25 3.50 4.25
```

Estos valores coinciden exactamente con los obtenidos en los Ejemplos 3.1.4 y 3.1.5. Gracias a estas funciones es inmediato explorar distintos pesos, sin más que modificar el vector `weights`, y ver cómo afectan al reparto final. En particular, se puede comprobar fácilmente que ambas reglas ponderadas convergen a la regla folk cuando todos los agentes reciben el mismo peso. Podemos probar a ejecutar, por ejemplo, `mcstOWShapley(C, weights = rep(1,3))` y comparar el resultado con `folk$folk`.

```
mcstOWShapley(C, weights = rep(1, 3)) # Regla folk

## Weights: 1 1 1
## 1 2 3
## 4 4 4
```

4.2.3. Función `mcstBoruvka`

Esta función implementa la regla de Borůvka propuesta por Bergantiños y Vidal-Puga (2011), en la que todas las componentes conexas buscan simultáneamente su conexión más barata, tal y como se formalizó en la Definición 3.1.7.

Uso

```
mcstBoruvka(C, draw = FALSE, titles = TRUE)
```

Al igual que `mcstFolk`, esta función no incluye el argumento `method`, pues el reparto de Borůvka tampoco depende de la forma en la que se manejan los empates durante el algoritmo. El objeto devuelto por esta función se presenta en la Tabla 4.5 y es completamente análogo al visto para la regla folk:

Campo	Tipo	Descripción
<code>\$boruvka</code>	vector numérico	Reparto de Borůvka para cada agente
<code>\$arcs</code>	data frame	Arcos del árbol óptimo
<code>\$total</code>	escalar	Coste total del árbol
<code>\$percentage</code>	vector numérico	Porcentaje del coste asignado a cada agente
<code>\$ranking</code>	vector carácter	Orden de los agentes según su pago
<code>\$is_unique</code>	lógico	Indica si el árbol óptimo es único

Tabla 4.5: Campos devueltos por el objeto resultado de la función `mcstBoruvka`.

En esta función las gráficas resultantes de activar `draw = TRUE` también son análogas a las ya vistas, por lo que las omitiremos aquí. El cálculo de la regla se hace mediante el siguiente comando:

```
# Regla de Boruvka
bor <- mcstBoruvka(C)
bor   #  $\beta^{\pi}(N\emptyset, C)$ 
```

```
## 1 2 3
## 4 4 4
```

El reparto obtenido es exactamente el mismo que devuelve `mcstFolk`. Tal y como se mencionó en el Capítulo 3, ambos coinciden independientemente de la ordenación escogida para Borůvka (Bergantiños y Vidal-Puga 2011). El objeto devuelto por `mcstBoruvka` comparte además la misma estructura que el de `mcstFolk` (Tabla 4.4), sustituyendo únicamente el campo `$folk` por `$boruvka`.

Por otro lado, el campo `$arcs` permite consultar el árbol resultante y la etapa en la que se incorporó cada agente, de forma análoga al resto de reglas:

```
bor$arcs   # Arcos del árbol óptimo
```

```
##   i j stage
## 1 1 3     1
## 2 2 3     2
## 3 0 2     3
```

4.2.4. Función `mcstCone`

La función `mcstCone` implementa la técnica de descomposición en conos propuesta por Norde et al. (2004), que permite extender cualquier regla definida para problemas elementales a un problema general (Definición 3.1.8).

Uso

```
mcstCone(C, rule = c("folk", "owshapley", "bogomolnaia"), weights = NULL, lambda = 1)
```

El argumento `rule` selecciona la regla elemental aplicada en cada capa de la descomposición: “`folk`” (Branzei et al. 2004), “`owshapley`” (Bergantiños y Lorenzo-Freire 2008a), que requiere especificar el argumento `weights` o “`bogomolnaia`”, que recoge la familia de Bogomolnaia y Moulin (2010) y está controlada mediante el argumento `lambda`.

Fijémonos que aquí no se incluye el argumento `draw`, ya que el objetivo de la función no es la visualización, sino entender lo que ocurre en cada capa de la descomposición. Para la visualización de los resultados se puede recurrir a las funciones directas que existen para ello: `mcstFolk` y `mcstOWShapley`. La Tabla 4.6 muestra los campos que devuelve el objeto resultado de emplear esta función:

Campo	Tipo	Descripción
\$allocation	vector numérico	Reparto correspondiente a la regla elegida
\$rule	cadena de texto	Regla empleada para el cálculo del reparto
\$total	escalar	Coste total del árbol
\$percentage	vector numérico	Porcentaje del coste asignado a cada agente
\$ranking	vector carácter	Orden de los agentes según su pago
\$decomposition	data frame	Detalle de la descomposición en conos
\$weights	vector numérico	Vector de pesos empleado (solo para Shapley ponderado optimista)
\$lambda	escalar	Valor de λ empleado (solo para la familia de Bogomolnaia y Moulin)

Tabla 4.6: Campos devueltos por el objeto resultado de la función `mcstCone`.

Por tanto, otra forma de obtener el reparto dado por la regla `folk` es el siguiente:

```
# Regla folk vía descomposición en conos
cone_folk <- mcstCone(C, rule = "folk")
cone_folk$allocation # F(N0,C)
```

```
## 1 2 3
## 4 4 4
```

El resultado coincide de nuevo con el reparto `folk` obtenido tanto por `mcstFolk` como por `mcstBoruvka`, confirmando una tercera forma de obtener dicho resultado. Realmente, la función `mcstCone` no tiene como objetivo aportar un resultado nuevo, sino poder examinar cómo se llega a él a través de las distintas capas de la descomposición. Por ello, la componente `$decomposition` devuelve un *data frame* con la descomposición explícita, incluyendo tanto los pesos x^q en la columna `weight` como el resultado del reparto bajo la regla elemental. Esto permite reconstruir la contribución de cada capa al reparto final, tal y como se detalló en el Ejemplo 3.1.7.

```
cone_folk$decomposition # Descomposición en matrices elementales
```

```
##      threshold weight      1      2      3
## Layer 1         3      3 1.0000 1.0000 1.0000
## Layer 2         5      2 0.3333 0.3333 0.3333
## Layer 3         6      1 0.3333 0.3333 0.3333
## Layer 4        10      4 0.0000 0.0000 0.0000
```

Las otras dos reglas siguen exactamente el mismo patrón. Para obtener el reparto asociado a la versión optimista de Shapley ponderado se debe especificar además el vector de pesos. Retomando el definido anteriormente, podemos escribir lo siguiente:

```
# Regla OW-Shapley vía descomposición en conos
cone_ows <- mcstCone(C, rule = "owshapley", weights = w)
cone_ows$allocation # f^ow(N0,C)
```

```
##      1      2      3
## 3.75 4.50 3.75
```

Este resultado, como cabía esperar, coincide con el que ya obtuvimos directamente mediante la función `mcstOWShapley`, lo que confirma que ambas implementan la misma regla desde dos perspectivas distintas. Nuevamente, el objeto `cone_ow` conserva el vector de pesos utilizado en `$weights`:

```
cone_ows$weights  # Vector de pesos

## 1 2 3
## 1 2 1
```

Por último, la familia de Bogomolnaia y Moulin introduce un parámetro adicional λ que regula cómo se reparte el coste dentro de cada componente. Para calcular el reparto asociado a esta familia usamos el siguiente código:

```
# Familia Bogomolnaia-Moulin con lambda = 2
cone_bog <- mcstCone(C, rule = "bogomolnaia", lambda = 2)
cone_bog$allocation  # R^lambda(N\emptyset, C)

##          1          2          3
## 4.133333 4.133333 3.733333
```

```
cone_bog$lambda  # Lambda

## [1] 2
```

En este caso, el objeto devuelto incorpora también el valor `$lambda` empleado, de modo que quede constancia del valor concreto para el que se ha obtenido el reparto. Esta función puede resultar útil para comparar cómo varían los repartos en función del λ asignado.

4.3. Reglas definidas a través de juegos cooperativos

Centrémonos ahora en el segundo bloque de funciones implementadas en este paquete, que engloban el marco de la teoría de juegos cooperativos. Cada función construye internamente la función característica del juego correspondiente y le aplica uno de los conceptos de solución vistos en los capítulos previos. Para redes grandes, podremos sustituir el cálculo exacto por una aproximación mediante `method = "montecarlo"`, tal y como se explicó en la Subsección 4.1.3. Sin embargo, dado que emplearemos el mismo ejemplo que llevamos empleando a lo largo de toda la memoria, nos ceñiremos a calcular el resultado exacto y profundizaremos más en las aproximaciones en el Capítulo 5.

La interfaz es uniforme para los cinco juegos programados:

```
Uso

mcstGame*(C, sol = c("shapley", "nucleolus"), draw = FALSE,
           which.plot = c("main", "details"), titles = TRUE,
           method = c("exact", "montecarlo"), nsim = 10000)
```

El `*` se sustituye en función del juego que nos interese calcular: “`Private`” para el juego privado, “`Irred`” para el irreducible, “`Opt`” para el optimista, “`Public`” para el público o “`CC`” para el ciclo-completo.

El argumento `sol` permite seleccionar el concepto de solución aplicado al juego: “`shapley`” para el valor de Shapley y “`nucleolus`” para el nucleolo. Este último se calcula únicamente de forma exacta, de manera que el argumento `method` solo afecta al cálculo del valor de Shapley. Como ya adelantamos,

los detalles sobre el rendimiento computacional de ambos métodos se abordarán en el Capítulo 5. Los argumentos `draw`, `which.plot` y `titles` actúan de la misma manera que lo hacían para las funciones anteriores, con la particularidad de que ahora el gráfico detallado no muestra una serie de árboles óptimos en base a permutaciones, sino el árbol óptimo obtenido al restringirnos a cada coalición.

Todas las funciones de esta familia devuelven un objeto con una serie de campos en común recogidos en la Tabla 4.7. Sin embargo, algunas de ellas incorporan campos específicos que se detallarán en los apartados referentes a las mismas.

Campo	Tipo	Contenido
<code>\$coalitions</code>	vector carácter	Todas las coaliciones $S \subseteq N$ evaluadas
<code>\$v_*</code>	vector numérico	Función característica $v^*(S)$ para cada coalición
<code>\$allocation</code>	vector numérico	Reparto: valor de Shapley o nucleolo
<code>\$total</code>	escalar	Coste total
<code>\$percentage</code>	vector numérico	Porcentaje del coste total asignado a cada agente
<code>\$ranking</code>	vector carácter	Orden de los agentes según su pago
<code>\$sol</code>	carácter	Concepto de solución empleado
<code>\$method</code>	carácter	Método empleado para el cálculo

Tabla 4.7: Campos comunes devueltos por el objeto resultado de las funciones `mcstGame*`.

Cabe destacar que hay dos campos, `$contributions` y `$nperms`, que no se mencionan en la tabla puesto que solo devuelven información cuando usamos `method = "montecarlo"`. Además, en la componente `$v_*`, el `*` se sustituye por la letra minúscula que identifica a cada juego, por ejemplo, “p” para el juego privado o “u” para el juego público.

4.3.1. Funciones `mcstGamePrivate` y `mcstKar`

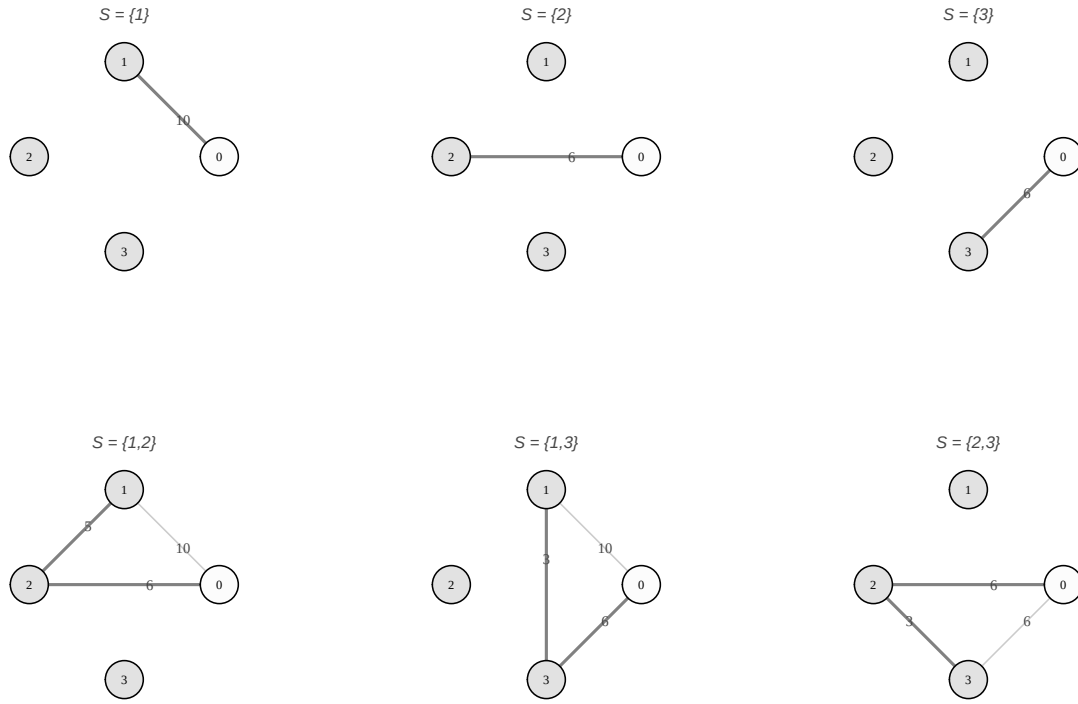
Recordemos que el juego privado es el más restrictivo de los cinco, tal y como se deriva de la Definición 2.5.1. La función `mcstGamePrivate` da acceso completo a este juego, permitiendo elegir entre el valor de Shapley y el nucleolo como concepto de solución. El código para ejecutar dicho juego para el valor de Shapley junto con su representación gráfica se recoge en las siguientes líneas:

```
# Juego privado
priv <- mcstGamePrivate(C, sol = "shapley", draw = TRUE, titles = FALSE)
priv$v_p # Función característica
```

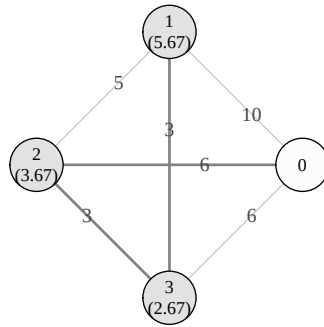
```
## [1] 10 6 6 11 9 9 12
```

Se puede comprobar que el resultado obtenido reproduce exactamente la primera fila de la Tabla 2.1, calculada a mano en la Sección 2.5. Esto resulta muy útil para obtener de forma inmediata el coste de cualquier coalición, muy conveniente para redes con un mayor número de agentes. Además, los gráficos devueltos con `draw = TRUE` apoyan estos resultados y permiten entender mejor su origen.

La Figura 4.10 muestra los dos gráficos de los que hablamos empleando arbitrariamente el algoritmo de Prim para seleccionar el árbol óptimo, puesto que nos interesa el coste de la red, no su estructura. Así, la Figura 4.10a muestra los árboles óptimos obtenidos al restringirnos a cada coalición específica, mientras que la Figura 4.10b muestra el árbol óptimo para la coalición total junto con el reparto final: el valor de Shapley. Cabe destacar que la visualización del primer gráfico se restringe a casos en los que el número de agentes es inferior a cuatro, de forma análoga a lo que ocurría con las funciones `mcstBird` y `mcstDuttaKar`.



(a) Árboles óptimos obtenidos para cada una de las coaliciones.



(b) Árbol óptimo para la coalición total con el reparto asignado a cada agente entre paréntesis.

Figura 4.10: Representaciones de los árboles óptimos (construidos mediante Prim) generados por `mcstGamePrivate`.

Esta representación se genera exactamente igual en el resto de funciones basadas en juegos cooperativos que veremos a continuación, por lo que no repetiremos la figura en cada una de ellas.

Cuando el concepto de solución elegido es precisamente el valor de Shapley, el reparto resultante recibe el nombre de regla de Kar, como ya vimos en la Definición 3.2.1. Para este caso concreto, el paquete ofrece `mcstKar`, un atajo con la misma interfaz que `mcstGamePrivate` que evita tener que especificar `sol = "shapley"` cada vez.

Uso

```
mcstKar(C, draw = FALSE, which.plot = c("main", "details"), titles = TRUE,
       method = c("exact", "montecarlo"), nsim = 10000)
```

De este modo, para obtener el reparto asociado a la regla de Kar basta con escribir las siguientes instrucciones:

```
# Regla de Kar
kar <- mcstKar(C)
kar$allocation # Kar(N0,C) = Sh(N,v^p)
```

```
##      1      2      3
## 5.666667 3.666667 2.666667
```

Como era de esperar, esta cifra coincide tanto con la que se puede extraer del gráfico de la Figura 4.10b como con el resultado calculado a mano en la Sección 2.5.

A pesar de existir este atajo, la función `mcstGamePrivate` sigue siendo útil cuando lo que nos interesa es calcular el nucleolo del juego:

```
# Juego privado: nucleolo
priv_nu <- mcstGamePrivate(C, sol = "nucleolus")
priv_nu$allocation # Nu(N,v^p)
```

```
##      1      2      3
## 4.75 4.50 2.75
```

4.3.2. Función `mcstGameIrred`

El juego irreducible aplica el mismo enfoque pesimista que el juego privado, pero evaluado sobre la matriz irreducible $\bar{C}$, tal y como se presentó en la Definición 2.5.2. Recordemos que esta matriz recoge, para cada par de nodos, el coste del arco más caro en el único camino que los conecta dentro del árbol óptimo. Se calcula internamente a partir del argumento `C` sin que el usuario deba construirla por su cuenta. La función `mcstGameIrred` almacena dicha matriz en el campo `$C_irred`, específico para esta función:

```
# Juego irreducible
irred <- mcstGameIrred(C, draw = TRUE, which.plot = "main", titles = FALSE)
irred$C_irred # Matriz irreducible
```

```
##      0 1 2 3
## 0 0 6 6 6
## 1 6 0 3 3
## 2 6 3 0 3
## 3 6 3 3 0
```

Como ya se mostró en el Capítulo 2, todos los costes de conexión directa a la fuente quedan igualados a 6, que es el coste del arco más caro del árbol óptimo. Esto explica por qué ahora las coaliciones individuales pasan a valer todas lo mismo en la función característica. El campo `$v_i` permite comprobar este efecto:

```
irred$v_i # Función característica
```

```
## [1] 6 6 6 9 9 9 12
```

La Figura 4.11 muestra uno de los gráficos obtenidos con `draw = TRUE`. Fijémonos que la red ha sido adaptada a los valores de la matriz irreducible y, análogamente a lo ya visto, el reparto asignado a cada agente se muestra entre paréntesis.

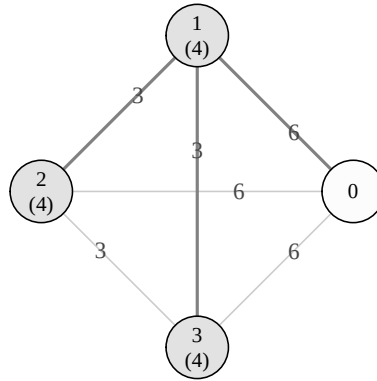


Figura 4.11: Representación del árbol óptimo de Prim generado por `mcstGameIrred` con el reparto de Shapley asignado a cada agente entre paréntesis.

Como el resto de funciones, la función `mcstGameIrred` permite acceder directamente al resultado del reparto. En este caso, todas las funciones comparten un mismo nombre para este campo:

```
irred$allocation # Sh(N,v^i)
```

```
## 1 2 3
```

```
## 4 4 4
```

Como cabía esperar por lo adelantado en el Capítulo 3, este resultado coincide exactamente con el obtenido mediante `folk$folk`. Como vemos, múltiples caminos conducen hacia este mismo reparto e incluso se puede comprobar que la regla de Bird aplicada sobre la matriz irreducible proporciona otra alternativa más para la obtención del resultado (Bergantiños y Vidal-Puga 2021):

```
mcstBird(irred$C_irred)$e_bird # Regla folk
```

```
## 1 2 3
```

```
## 4 4 4
```

4.3.3. Función `mcstGameOpt`

Recordemos que el juego optimista invierte el supuesto del juego privado, permitiendo que sus agentes se conecten gratuitamente a través de los nodos de los agentes externos, siguiendo la Definición 2.5.3.

A diferencia de los dos juegos anteriores, aquí la matriz de costes utilizada para evaluar cada coalición S no es fija, sino que cambia según qué agentes quedan fuera de S . La función `mcstGameOpt` construye

internamente una matriz optimista distinta para cada coalición y las almacena en el campo `$C_opt`. Por ejemplo, para extraer la matriz asociada a la coalición $S = \{1, 2\}$, escribimos lo siguiente:

```
# Juego optimista
opt <- mcstGameOpt(C)
opt$C_opt$S_12 # Matriz optimista para {1,2}

##    0 1 2
## 0 0 3 3
## 1 3 0 5
## 2 3 5 0
```

Este campo es exclusivo de `mcstGameOpt`, ya que es el único juego que necesita modificar la matriz de costes coalición a coalición y, además, resulta útil para entender de dónde proviene cada valor de la función característica. En este caso, la coalición $\{1, 2\}$ puede conectarse gratuitamente a través de la red de conexiones del agente 3. Así, tanto el agente 1 como el 2 disponen de una conexión a la fuente con coste 3, lo que da lugar a un total de 6 unidades, tal y como se muestra en cuarta posición en el vector de la función característica:

```
opt$vo # Función característica

## [1] 3 3 3 6 6 6 12
```

Como se puede ver en `$vo`, los valores de las coaliciones individuales y de tamaño dos son los más bajos vistos hasta ahora, lo cual tiene sentido ya que estamos suponiendo el escenario más favorable posible para cualquier coalición. El reparto asociado a este juego se obtiene igual que antes:

```
opt$allocation # Sh(N, v^o)

## 1 2 3
## 4 4 4
```

A pesar de partir de una función característica muy distinta a la del juego irreducible, el valor de Shapley obtenido es exactamente el mismo, lo que demuestra el correcto funcionamiento de esta función y de las anteriores. Para finalizar, podríamos calcular el reparto asociado al nucleolo. Sin embargo, por lo visto en la Sección 3.2, cabe esperar que la función devuelva un resultado nulo en concordancia con lo que allí se concluyó. El conjunto de imputaciones resulta ser vacío para este juego, por lo que el nucleolo no está definido.

```
# Juego optimista: nucleolo
mcstGameOpt(C, sol = "nucleolus") # Nu(N, v^o)

## [1] "Solution concept is only provided for games with a nonempty imputation set."

##      S v^o(S)
## 1 {1}      3
## 2 {2}      3
## 3 {3}      3
## 4 {1,2}    6
## 5 {1,3}    6
## 6 {2,3}    6
```

```
## 7      N      12
##
## Solution: Nucleolus
## 1  2  3
## NA NA NA
```

4.3.4. Función `mcstGamePublic`

El juego público elimina cualquier derecho de propiedad sobre los agentes. Formalmente, como se vio en la Definición 2.5.4, el coste de una coalición S se define como el mínimo coste privado entre todos los superconjuntos $T \supseteq S$, de modo que calcular v^u requiere resolver primero el juego privado sobre todas las coaliciones.

Dado que v^u se apoya en v^p , la función `mcstGamePublic` devuelve ambas funciones características, permitiendo comparar directamente sus resultados. Para calcular la función característica asociada al juego público hacemos lo siguiente:

```
# Juego público
pub <- mcstGamePublic(C)
pub$v_u # Función característica
```

```
## [1] 9 6 6 11 9 9 12
```

Además de `$v_u`, el objeto incluye un campo exclusivo para esta función, `$best_T`, que identifica para cada coalición S cuál es el superconjunto T que alcanza ese mínimo coste:

```
pub$best_T # Superconjuntos de coste mínimo
```

```
##      S      T
## 1  {1} {1,3}
## 2  {2} {2}
## 3  {3} {3}
## 4 {1,2} {1,2}
## 5 {1,3} {1,3}
## 6 {2,3} {2,3}
## 7      N      N
```

En este caso, solo la coalición $\{1\}$ mejora su coste privado al ampliarse a $\{1,3\}$, mientras que para el resto de coaliciones el superconjunto óptimo coincide con la propia coalición. Esto se refleja perfectamente en los valores de ambas funciones características que únicamente difieren en el coste de su primera componente. De nuevo, podemos obtener el reparto correspondiente al valor de Shapley mediante el campo `$allocation`:

```
pub$allocation # Sh(N, v^u)
```

```
##      1      2      3
## 5.333333 3.833333 2.833333
```

Aquí el valor de Shapley no coincide con la regla folk. Además, al agente 1, que es el que tiene un coste individual más alto de los tres, $v(\{1\}) = 9$, se le asigna un reparto superior.

4.3.5. Función mcstGameCC

El juego ciclo-completo aplica nuevamente el mismo enfoque pesimista que el juego privado, pero evaluado sobre la matriz ciclo-completa C^* , ya explicada en la Sección 2.1. La función `mcstGameCC` calcula internamente esta matriz y la almacena en el campo `$C_cc`:

```
# Juego ciclo-completo
cc <- mcstGameCC(C, draw = TRUE, which.plot = "main", titles = FALSE)
cc$C_cc # Matriz ciclo-completa

##  0 1 2 3
## 0 0 6 6 6
## 1 6 0 5 3
## 2 6 5 0 3
## 3 6 3 3 0
```

Tal y como ya habíamos deducido en un capítulo previo, la única entrada en la que difieren la matriz irreducible y la ciclo-completa es la que representa el coste entre los agentes 1 y 2. En base a esta matriz, podemos obtener la función característica para este juego:

```
cc$v_c # Función característica

## [1] 6 6 6 11 9 9 12
```

Como cabía esperar de la comparación anterior, v^c difiere de v^i únicamente en la coalición $\{1, 2\}$ ($v^c(\{1, 2\}) = 11$ frente a $v^i(\{1, 2\}) = 9$), lo que refleja el efecto de considerar ciclos entre los agentes al evaluar una coalición determinada.

La Figura 4.12 muestra uno de los gráficos obtenidos con `draw = TRUE`. Fijémonos que ahora la red ha sido adaptada a los valores de la matriz ciclo-completa:

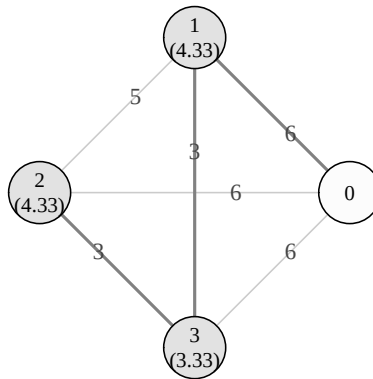


Figura 4.12: Representación del árbol óptimo de Prim generado por `mcstGameCC` con el reparto de Shapley asignado a cada agente entre paréntesis.

La función `mcstGameCC` también permite acceder directamente al resultado del reparto mediante `$allocation`:

```
cc$allocation # Sh(N,v^c): solución ciclo-completa

##      1      2      3
## 4.333333 4.333333 3.333333
```

Por último, se podría calcular el nucleolo para este juego y comparar el resultado con el resto de asignaciones. Para ello, escribimos las siguientes líneas de código:

```
# Juego ciclo-completo: nucleolo
cc_nu <- mcstGameCC(C, sol = "nucleolus")
cc_nu$allocation # Nu(N,v^c)

##      1      2      3
## 4.5 4.5 3.0
```

Fijémonos que el nucleolo redistribuye ligeramente el coste con respecto al valor de Shapley. Para este último, las coaliciones $\{1, 3\}$ y $\{2, 3\}$ son las que tienen un margen de ahorro más bajo. Mientras que $v(\{1, 3\}) = v(\{2, 3\}) = 9$, su valor de Shapley sería de aproximadamente $4.33 + 3.33 = 7.66$ unidades para ambos. Como el nucleolo trata de buscar la asignación que produzca una menor insatisfacción entre los agentes, este actúa reduciendo el coste al agente común a ambas coaliciones, el 3.

4.4. Herramientas de análisis

El tercer bloque del paquete ofrece herramientas para comparar y analizar la estabilidad y sensibilidad de los repartos obtenidos mediante las funciones de los dos bloques anteriores.

4.4.1. Comparativa de reglas: mcstCompare

Hasta ahora hemos ido calculando cada regla por separado llamando a una función distinta y accediendo al campo correspondiente del reparto. La función `mcstCompare` evita tener que repetir este proceso cuando lo que nos interesa es comparar varias reglas entre sí para un mismo problema:

Uso

```
mcstCompare(C)
```

Esta función únicamente admite el argumento `C` y se restringe a la comparación de las reglas más populares: Bird, Dutta-Kar, folk y Kar. La salida devuelve un *data frame* que recoge los repartos asociados a estas reglas:

```
# Comparativa de reglas
comp <- mcstCompare(C)
comp

##      _____
## MCST Allocation Rules Comparison
##      _____
## agent bird dutta_kar folk kar
##      1  3.0          5   4 5.7
##      2  4.5          4   4 3.7
##      3  4.5          3   4 2.7
```

El formato de salida permite al usuario manipular el objeto con las herramientas habituales de R, por ejemplo, para exportar los resultados a un fichero, filtrar el *data frame* por regla o por agente o incluso reutilizar los valores como entrada para otras funciones.

Además de la tabla, `mcstCompare` incluye un argumento `plot` que representa gráficamente estas mismas diferencias mediante un diagrama de barras agrupado por agente:

```
plot(comp, titles = FALSE) # Gráficamente
```

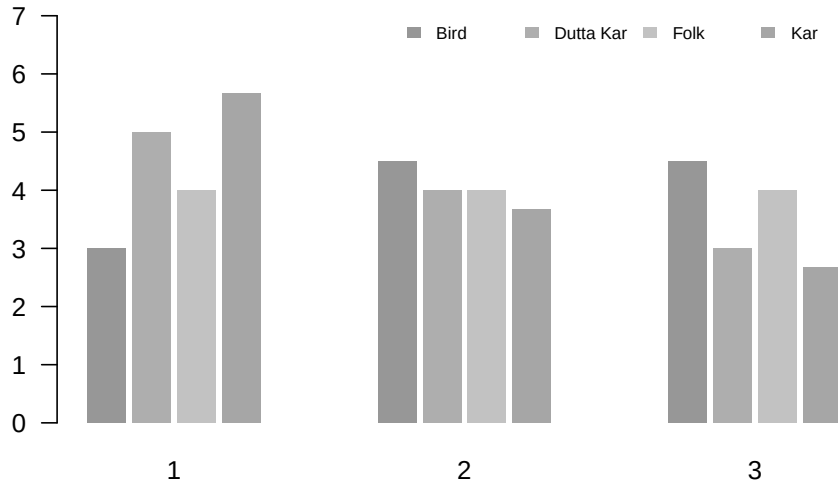


Figura 4.13: Comparación gráfica de los repartos de Bird, Dutta-Kar, folk y Kar.

Esta visualización resulta útil cuando lo que nos interesa no es tanto el valor exacto de cada reparto, sino el patrón que sigue cada una de las reglas. La Figura 4.13 muestra cómo la regla folk distribuye el coste de manera equitativa entre todos los agentes, mientras que el resto de reglas lo hacen de forma asimétrica. Por ejemplo, la regla de Bird beneficia al agente 1, al contrario de lo que ocurre con las reglas de Kar y Dutta Kar, que penalizan en mayor medida a este agente y reducen de manera significativa el pago del agente 3.

4.4.2. Análisis del núcleo: `mcstCoreCheck`, `mcstCorePoints` y `mcstCorePlot`

Hasta ahora, hemos visto funciones que nos devuelven el resultado concreto de un reparto. Sin embargo, ninguna de ellas nos da información acerca de su estabilidad. Por ello, el paquete ofrece tres funciones complementarias para analizar diferentes aspectos del núcleo: `mcstCoreCheck` para comprobar si un reparto es estable, `mcstCorePoints` para comprobar su existencia y `mcstCorePlot` para visualizarlo geométricamente.

Las tres funciones están programadas para recibir el juego de la misma forma. Puede pasarse directamente el objeto devuelto por cualquiera de las funciones `mcstGame*` a través del argumento `game`, o bien introducir directamente la función característica del juego que nos interese a través del argumento `v`. Esta segunda forma resulta útil para estudiar juegos que no se derivan de ningún problema concreto o cuando ya se dispone de los valores $v(S)$ calculados por otro medio.

Estabilidad de un reparto: mcstCoreCheck

La función `mcstCoreCheck` evalúa si un reparto concreto es estable. Tal y como se formalizó en la Definición 1.2.3, esto viene garantizado por dos condiciones: eficiencia y racionalidad coalicional. Esta función presenta la siguiente estructura de uso:

Uso

```
mcstCoreCheck(game = NULL, allocation = NULL, v = NULL, tol = 1e-07)
```

Cabe destacar que si el usuario emplea el argumento `game`, la función comprobará el reparto asociado a dicho juego, por lo que el usuario no debe introducir nada en el argumento `allocation`. Siempre que se quiera verificar la estabilidad de un reparto diferente al proporcionado por el juego, se deben utilizar conjuntamente los argumentos `v` y `allocation`. Por ejemplo, para comprobar si el reparto de Bird calculado anteriormente es estable para el juego privado, escribimos lo siguiente:

```
# ¿El reparto de Bird está en Core(v^p)?
check_bird <- mcstCoreCheck(allocation = bird$e_bird, v = priv$v_p)
check_bird
```

```
## -----
## Core Stability Analysis
## -----
## Players (n) : 3
## Efficiency   : Passed (Sum x: 12.00 | v(N): 12.00)
## Rationality  : Passed
##
## Result: the allocation IS IN THE CORE (Stable)
```

Además de esta salida, el campo `$in_core` resume el resultado global. Incluso, es posible comprobar si se han verificado o no ambas condiciones de estabilidad consultando los campos `$is_efficient` y `$is_rational`:

```
check_bird$is_efficient # ¿Se reparte exactamente el coste total?
```

```
## [1] TRUE
```

```
check_bird$is_rational # ¿Ninguna coalición paga de más?
```

```
## [1] TRUE
```

Esto puede resultar útil para los casos en los que el reparto no es estable, ya que permite identificar qué es lo que ha fallado exactamente. Además, si alguna coalición tuviera incentivos para separarse, el campo `$is_rational` devolvería `FALSE` y un nuevo campo `$blocking_coals` identificaría explícitamente qué coalición o coaliciones impiden la estabilidad del reparto, mostrando el coste de más que estarían asumiendo:

```
# Ejemplo hipotético con un reparto inestable
check_inestable <- mcstCoreCheck(allocation = c(2, 2, 8), v = priv$v_p)
check_inestable$blocking_coals
```

```
##      S x(S) v(S) excess
## 1  {3}   8   6     2
## 2 {1,3}  10   9     1
## 3 {2,3}  10   9     1
```

En este ejemplo, creado específicamente para ilustrar cómo la función maneja los repartos inestables, los excesos positivos (2, 1, 1) indican que las respectivas coaliciones están insatisfechas, pagando más de lo que les costaría actuar por su cuenta y bloqueando la estabilidad del reparto.

Existencia del núcleo de un juego: `mcstCorePoints`

Mientras que `mcstCoreCheck` evalúa la estabilidad de un único reparto dado, `mcstCorePoints` va un paso más allá: determina si el núcleo del juego está vacío y, si no lo está, devuelve el resultado de una posible asignación estable y calcula explícitamente todos sus vértices geométricos mediante programación lineal.

Uso

```
mcstCorePoints(game = NULL, v = NULL)
```

Fijémonos que esta función ya no requiere del argumento `allocation`, puesto que ahora ya no se comprueba la estabilidad de ningún reparto concreto, sino que se verifica la existencia de al menos un punto que satisfaga las inecuaciones asociadas al núcleo, de forma que es la propia función la que devuelve una asignación estable si existe. Para ver cómo funciona, podemos verificar la existencia del núcleo del juego privado calculado en la Sección 3.2 empleando el campo `$is_empty`, a través del siguiente código:

```
# ¿El núcleo del juego privado es vacío?
priv_cp <- mcstCorePoints(game = priv)
priv_cp$is_empty
```

```
## [1] FALSE
```

Como era de esperar, el núcleo del juego privado no es vacío. En consecuencia, la función calcula sus vértices y devuelve una asignación estable. Esto se puede consultar en los campos `$vertices` y `$core_point`, respectivamente:

```
priv_cp$vertices # Enumeración de los vértices que conforman el núcleo
```

```
##      [,1] [,2] [,3]
## [1,]   3   6   3
## [2,]   3   3   6
## [3,]   5   6   1
## [4,]   8   3   1
```

```
priv_cp$core_point # Reparto estable dentro del núcleo
```

```
## 1 2 3
## 8 3 1
```

Los cuatro vértices reproducen exactamente el resultado del Ejemplo 3.2.1. Disponer de todos ellos, es útil para tener una idea del rango completo de repartos estables posibles, en lugar de tener que probar reparto a reparto con `mcstCoreCheck`.

Cuando el núcleo está vacío, la función devuelve directamente esta información sin intentar calcular los vértices. Para ilustrarlo, retomemos el resultado correspondiente al juego optimista visto en la Observación 3.2.2, que nos decía que el conjunto de imputaciones de este juego es vacío.

```
# ¿El núcleo del juego optimista es vacío?
mcstCorePoints(game = opt)

## -----
## Core Emptiness Analysis
## -----
## Players (n) : 3
## Total Cost : 12
##
## Result: the core IS EMPTY
## No allocation satisfies efficiency and coalitional rationality simultaneously
```

Visualización del núcleo: `mcstCorePlot`

Las dos funciones anteriores devuelven información numérica, pero entender la forma y tamaño del núcleo y dónde se sitúan los distintos repartos, resulta mucho más intuitivo gráficamente. Para ello, `mcstCorePlot` genera una representación geométrica que se adapta automáticamente al número de agentes: un segmento de recta para 2 agentes, un triángulo en coordenadas baricéntricas para 3 y un poliedro tridimensional interactivo para 4. La estructura de uso de esta función es la siguiente:

Uso

```
mcstCorePlot(game = NULL, v = NULL, allocations = list(), titles = TRUE)
```

Cabe destacar que si el usuario emplea únicamente el argumento `game`, la función devuelve por defecto tanto el núcleo del juego como la solución interna asociada. Por el contrario, si se emplea el argumento `v` (o se desea sobrescribir el reparto por defecto proporcionado por `game`), es necesario especificar el reparto o repartos que se quieren mostrar mediante el argumento `allocations`. En estos casos, la función representará exclusivamente los repartos proporcionados explícitamente por el usuario. Por ejemplo, podemos representar el núcleo del juego privado junto con las asignaciones de Bird, folk y Kar. El resultado deberá ser algo similar al obtenido en la Figura 3.1. La instrucción sería la siguiente:

```
# Representación gráfica de Core(v^p) junto con los repartos de Bird, folk y Kar
mcstCorePlot(v = priv$v_p, allocations = list(Bird = bird$e_bird,
                                              Folk = folk$folk,
                                              Kar = kar$allocation), titles = FALSE)
```

En la Figura 4.14, el triángulo completo representa el conjunto de imputaciones del juego privado, mientras que la región sombreada de verde en su interior es su núcleo. Como ya se comprobó, los tres repartos caen dentro de esa región, por lo que todas son soluciones estables.

Una de las características más novedosas de esta función es que permite representar el núcleo para problemas con cuatro agentes, es decir, en tres dimensiones. Con el fin de ilustrar esta capacidad, se

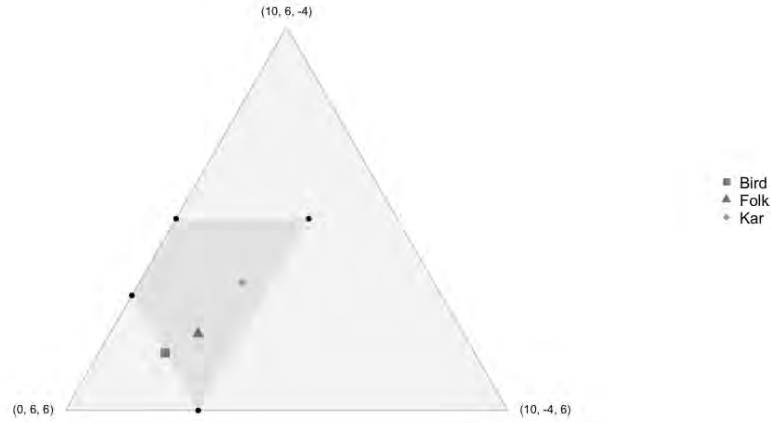
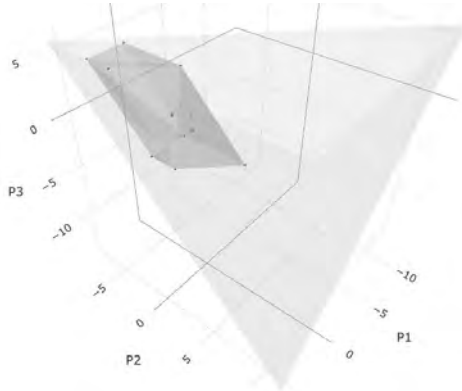


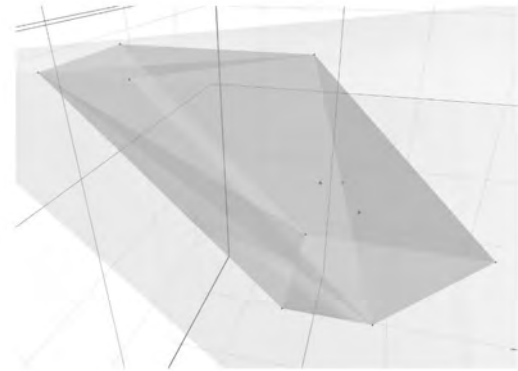
Figura 4.14: Representación del núcleo del juego privado v^p (región verde) y las asignaciones de Bird, folk y Kar sobre el conjunto de imputaciones para el Ejemplo 2.1.1 mediante `mcstCorePlot`.

presenta a continuación un escenario diseñado específicamente para ello. La Figura 4.15 recoge dicha representación desde dos perspectivas diferentes:

```
# Ejemplo hipotético para un problema con 4 agentes
C4 <- c(6, 12, 6, 7, 3, 10, 3, 1, 1, 7)
mcstCorePlot(game = mcstGamePrivate(C4),
             allocations = list(Bird = mcstBird(C4)$e_bird,
                               Folk = mcstFolk(C4)$folk,
                               Kar = mcstKar(C4)$allocation), titles = FALSE)
```



(a) Vista general.



(b) Vista detallada.

Figura 4.15: Representación del núcleo del juego privado (región verde) y las asignaciones de Bird, folk y Kar sobre el conjunto de imputaciones del ejemplo con cuatro agentes mediante `mcstCorePlot`.

Dado que se trata de un gráfico en tres dimensiones, el código anterior devuelve una representación interactiva, que permite al usuario manipular la figura para rotar los ejes, modificar la perspectiva o ampliar una zona específica. Análogamente a la Figura 4.14, ahora el conjunto de imputaciones es

un tetraedro y el núcleo un poliedro contenido en su interior. Al igual que antes, para este ejemplo particular, los tres repartos caen dentro él, por lo que también son soluciones estables.

Cabe destacar que la función `mcstCorePoints` admite un argumento `plot` que reutiliza internamente esta misma representación, mostrando el núcleo y el punto factible calculado mediante programación lineal.

Todas estas funciones pueden resultar de gran utilidad para justificar la elección de una regla sobre otra en un problema real en base a algo más que un simple resultado numérico.

4.4.3. Rango de estabilidad: `mcstStabilityRange`

En un problema real la matriz de costes no siempre es estática. Por ejemplo, esos valores pueden fluctuar ante cambios en precios de materiales o servicios. La función `mcstStabilityRange` permite calcular, siguiendo el problema clásico de análisis de sensibilidad de árboles de mínimo coste de Tarjan (1982), cuánto puede variar el coste de cada arco antes de que cambie el árbol óptimo, aunque el coste total y el reparto lógicamente se ajusten a los nuevos precios.

Uso

```
mcstStabilityRange(C)
```

Esta función únicamente admite como entrada el argumento `C`, por lo que es muy sencilla de utilizar. Es importante destacar que está diseñada para evaluar la estabilidad de un único árbol en concreto, por lo que si la matriz de costes presenta empates, el algoritmo interno (basado en Prim) seleccionará arbitrariamente uno de ellos. Como consecuencia, si el usuario estuviera trabajando con otro árbol óptimo alternativo, los resultados de la salida no tendrían sentido para él.

Para ilustrar el uso de la función sin dar lugar a ambigüedades, modificaremos ligeramente la matriz `C` empleada a lo largo de toda la memoria, asignando $c_{03} = 7$ para deshacer el empate:

```
# Rango de estabilidad de los arcos del problema
C_mod <- c(10, 6, 7, 5, 3, 3)
mcstStabilityRange(C_mod)
```

```
## from to in_mst cost      range LB UB
##    0  1 FALSE   10 [-4, Inf]  6 Inf
##    0  2  TRUE    6  [-6,  1]  0  7
##    0  3 FALSE    7  [-1, Inf]  6 Inf
##    1  2 FALSE    5  [-2, Inf]  3 Inf
##    1  3  TRUE    3  [-3,  2]  0  5
##    2  3  TRUE    3  [-3,  2]  0  5
```

En la salida, podemos diferenciar entre los arcos que pertenecen al árbol óptimo y los que no a través de la columna `in_mst`. Si un arco ya forma parte de la red es porque ha sido la elección más barata según el algoritmo, por lo que abaratarlo aún más no provocaría ningún cambio en el árbol. Matemáticamente, esto implicaría poder fijar su límite inferior en $-\infty$. Sin embargo, dado que estamos tratando con problemas de costes (no negativos), su valor se restringiría a 0. El valor crítico estaría, por tanto, en su límite superior (UB), que es el coste de la alternativa más barata que podría sustituirlos. Por ejemplo, si el coste del arco (0,2) superase el valor de 7, el arco (0,3) pasaría a ser el elegido para conectarse a la fuente.

Por su parte, los arcos que no pertenecen al árbol óptimo presentan un límite superior ∞ . Esto es así porque encarecer un arco que ya fue descartado por su alto coste tampoco alteraría la estructura del

árbol. En este caso, el valor crítico de estos arcos estaría en su límite inferior (LB), que indica el valor exacto al que debería bajar su precio para volverse competitivos. Por ejemplo, el arco (0,1) tendría que abaratar su coste un mínimo de 4 unidades para considerarlo como una posible alternativa.

Es importante señalar que si el coste de algún arco alcanzara uno de los extremos finitos de su intervalo aparecerían empates. Sin embargo, el árbol considerado seguiría siendo óptimo, aunque ya no necesariamente el único.

La información devuelta por esta función puede resultar útil como paso previo al análisis de sensibilidad que veremos a continuación, ya que permite identificar de antemano los arcos más propensos a modificar la estructura del árbol.

4.4.4. Análisis de sensibilidad: `mcstSensitivity`

La última función de este bloque y con la que se concluye el conjunto completo de funciones del paquete es `mcstSensitivity`. Esta realiza un análisis de sensibilidad para los arcos del problema, es decir, muestra cómo evoluciona exactamente el reparto de cada agente a medida que el coste de uno o varios arcos varía y presenta la siguiente estructura de uso:

Uso

```
mcstSensitivity(C, rule = c("bird", "duttakar", "folk", "kar"), arcs,
               delta = list(c(-5, 5), c(-3, 3)), step = 1, independent = FALSE)
```

La función admite el análisis para las cuatro reglas más destacadas: Bird, Dutta-Kar, folk y Kar. El argumento `arcs` especifica qué arco o arcos se perturban dentro de un cierto rango dado por `delta`. Además, se puede especificar el tamaño del paso empleado para recorrer `delta` mediante el argumento `step`. Por último, si quisiéramos variar el rango de dos arcos simultáneamente de forma independiente activaríamos `independent = TRUE`. A continuación, analizaremos las distintas funcionalidades que ofrece esta función según el caso que queramos estudiar.

Caso 1: variación de un único arco

El uso más simple consiste en perturbar el coste de un solo arco a lo largo de un intervalo y observar la evolución del reparto bajo la regla elegida. Por ejemplo, podríamos estudiar cómo afecta al reparto de Dutta-Kar variar el arco (2,3) para $\Delta \in [-3,3]$:

```
# ¿Cómo varía la regla de Dutta-Kar al perturbar el arco (2,3)?
sens <- mcstSensitivity(C, rule = "duttakar", arcs = c(2, 3), delta = c(-3, 3))
sens
```

```
##   delta arc_cost   1   2   3
## 1    -3         0 6.0 1.5 1.5
## 2    -2         1 6.0 2.0 2.0
## 3    -1         2 6.0 2.5 2.5
## 4     0         3 5.0 4.0 3.0
## 5     1         4 5.0 5.0 3.0
## 6     2         5 4.5 5.5 4.0
## 7     3         6 4.0 5.5 4.5
```

La columna `arc_cost` muestra el coste resultante del arco tras aplicar cada variación `delta` y las columnas siguientes recogen el reparto que le correspondería a cada agente bajo ese coste. En este ejemplo, se puede comprobar que la suma de las aportaciones de los tres agentes crece de manera

constante a medida que el arco se encarece, en concordancia con la propiedad de monotonicidad de costes de Dutta-Kar.

Para visualizar la evolución de los repartos basta con aplicar el argumento `plot` sobre el objeto resultante:

```
plot(sens, titles = FALSE) # Gráficamente
```

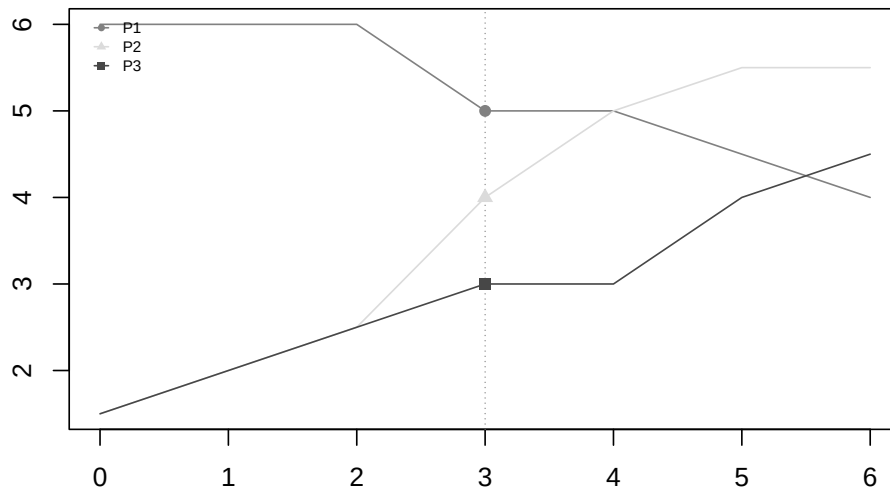


Figura 4.16: Evolución del reparto de Dutta-Kar para cada agente al variar el coste del arco (2,3) en $\Delta \in [-3, 3]$. La línea vertical discontinua marca el coste original del arco ($c_{23} = 3$).

La Figura 4.16 muestra que la evolución del pago asignado a cada agente es continua y líneas a trozos, sin saltos bruscos a lo largo del intervalo. Además, se ve cómo los pagos de los agentes involucrados en el arco tienden a subir a medida que aumenta su coste, obligando a reducir el pago del agente 1 para preservar la eficiencia.

Caso 2: variación conjunta de varios arcos

Hay casos en los que interesa estudiar el efecto conjunto de perturbar varios arcos al mismo tiempo. Para ello, basta con pasar una lista de arcos en `arcs` y mantener el argumento `independent` con su valor por defecto, pues consideraremos que los arcos especificados varían simultáneamente sobre un mismo intervalo marcado por `delta`:

```
# ¿Cómo varía la regla de Kar al perturbar conjuntamente los arcos (0,3) y (1,3)?
sens_sim <- mcstSensitivity(C, rule = "kar", arcs = list(c(0, 3), c(1, 3)),
                          delta = c(-2, 2))
sens_sim
```

```
##   delta      1      2      3
## 1    -2 4.666667 3.666667 -0.3333333
## 2    -1 5.166667 3.666667  1.1666667
## 3     0 5.666667 3.666667  2.6666667
## 4     1 6.166667 3.166667  3.6666667
## 5     2 6.666667 2.666667  4.6666667
```

En este caso la tabla ya no incluye una columna `arc_cost`, puesto que se están variando dos costes distintos a la vez y no tendría sentido resumirlos en un único valor. Nuevamente, podemos visualizar los resultados aplicando `plot` sobre el objeto anterior:

```
plot(sens_sim, titles = FALSE) # Gráficamente
```

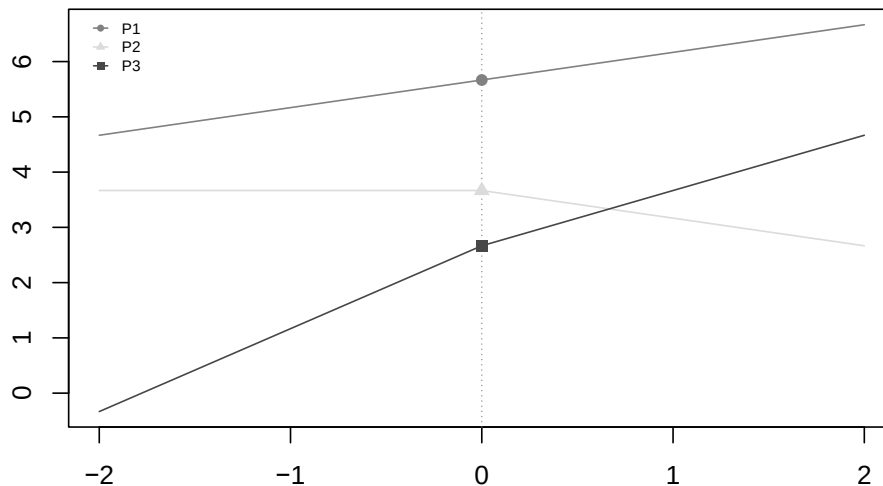


Figura 4.17: Evolución del reparto de Kar para cada agente al variar conjuntamente el coste de los arcos (0, 3) y (1, 3) en $\Delta \in [-2, 2]$. La línea vertical discontinua marca el estado original sin perturbación ($\Delta = 0$).

A diferencia del caso anterior, el eje horizontal de la Figura 4.17 ahora representa únicamente la perturbación `delta` aplicada a ambos arcos. La regla de Kar también tiene un comportamiento continuo y lineal a trozos, incluso cuando se perturban varios arcos simultáneamente. Fijémonos que el pago del agente 3, común a los dos arcos perturbados, se ve muy afectado en ambas direcciones por la perturbación, llegando incluso a alcanzar un “pago negativo” (equivalente a una compensación del resto de agentes) cuando $\Delta = -2$.

Caso 3: variación independiente de dos arcos

Por último, para explorar el efecto cruzado de dos arcos que varían de forma independiente, se activa el argumento `independent = TRUE`. Cabe destacar que con este argumento solo se permite perturbar exactamente dos arcos, ya que si no sería inviable representar el resultado en dos dimensiones. En este caso, `delta` pasa a ser una lista con un rango de variación distinto para cada arco. Veamos un ejemplo:

```
# ¿Cómo varía la regla folk al perturbar de forma independiente los arcos (1,2)
# y (2,3)?
sens_indep <- mcstSensitivity(C, rule = "folk", arcs = list(c(1, 2), c(2, 3)),
                             delta = list(c(-4, 4), c(-2, 2)), independent = TRUE)
head(sens_indep)
```

```
##   delta1 delta2      1      2      3
## 1    -4     -2 2.666667 2.666667 2.666667
## 2    -4     -1 2.833333 2.833333 3.333333
```

```
## 3    -4      0 3.000000 3.000000 4.000000
## 4    -4      1 3.000000 3.000000 4.000000
## 5    -4      2 3.000000 3.000000 4.000000
## 6    -3     -2 3.333333 2.833333 2.833333
```

Fijémonos que, nuevamente, el *data frame* resultante no contiene la columna `arc_cost`. En su lugar, la función devuelve en las dos primeras columnas (`delta1` y `delta2`) todas las posibles combinaciones de los intervalos especificados en `delta`. En este caso, aunque la interpretación no cambia con respecto a los otros escenarios, resulta muy útil visualizar un resumen de los resultados en forma de mapa de calor para ver cómo interactúan ambas perturbaciones. Aplicando el argumento `plot` sobre el objeto anterior se tiene lo siguiente:

```
plot(sens_indep, titles = FALSE) # Gráficamente
```

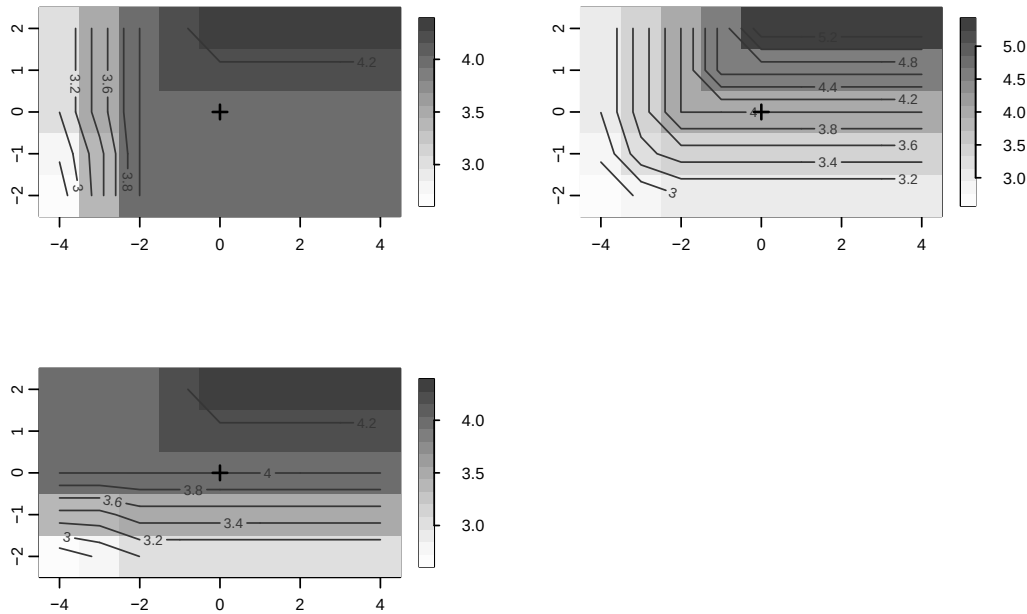


Figura 4.18: Mapas de calor asociados al reparto folk para cada agente al variar el coste del arco (1,2) en $\Delta_1 \in [-4, 4]$ y (2,3) en $\Delta_2 \in [-2, 2]$. La cruz marca el estado original ($\Delta_1 = \Delta_2 = 0$).

Cada panel de la Figura 4.18 corresponde a un agente y las curvas de nivel representan el coste asociado a una combinación concreta de variaciones en ambos arcos. Por ejemplo, en el primer panel, correspondiente al agente 1, destaca una amplia región del mismo color, lo que implica que para ciertas combinaciones de perturbaciones el pago de este se estabiliza. Por el contrario, en la mitad derecha del panel del agente 3, las curvas de nivel son prácticamente horizontales, lo que sugiere que su pago en esa región depende esencialmente del coste del arco (2,3) y deja de depender de las variaciones en el arco (1,2).

Capítulo 5

Análisis de rendimiento computacional

En la Sección 3.3 presentamos el método de Monte Carlo, explicando que tanto la extensión de las reglas de Bird y Dutta-Kar como el cálculo del valor de Shapley en los juegos cooperativos recurren a una aproximación de Monte Carlo cuando el cálculo exacto deja de ser viable. Posteriormente, en la Sección 4.1.3 introdujimos el argumento `method` que permite implementar esta vía en el paquete de R. A continuación, retomamos esta idea desde un punto de vista práctico, usando los conceptos teóricos vistos como referencia para interpretar los resultados empíricos de este capítulo.

Cabe señalar que, como vimos en la Sección 3.3, aunque ambos estimadores comparten la misma velocidad de convergencia asintótica $\mathcal{O}(nsim^{-1/2})$, el coste por iteración es distinto en cada caso, ya que una resolución de Prim cuesta $\mathcal{O}(n^2)$ frente a las n evaluaciones de v que exige el valor de Shapley. Todos estos órdenes describen la complejidad algorítmica teórica de cada procedimiento y no el tiempo de ejecución o error real.

5.1. Comparativa de tiempos de ejecución: exacto vs. Monte Carlo

En esta sección calculamos empíricamente cuánto tarda el método exacto frente a Monte Carlo para cada uno de los problemas planteados en la Sección 3.3. En particular, ilustraremos los resultados para las reglas de Bird y Kar, respectivamente. Además, también veremos a partir de qué tamaño de red es conveniente cambiar de método, qué ocurre para el método Monte Carlo cuando empleamos redes que el método exacto no puede abordar y, finalmente, una comparativa con el paquete `cooptrees` de Fontenla (2014a) mencionado en la introducción del Capítulo 4. Todos los tiempos han sido medidos en la misma máquina con `nsim = 10 000` fijo en todas las llamadas a `method = "montecarlo"` y usando matrices generadas forzando empates, de manera que la regla de Bird tenga que recurrir a la regla extendida.

Regla de Bird: coste factorial frente a Monte Carlo

La Tabla 5.1 recoge los tiempos obtenidos con `mcstBird` para $n = 6, 7, \dots, 10$ bajo ambos métodos.

n	Permutaciones ($n!$)	Exacto (s)	Monte Carlo (s)
6	720	1.36	12.35
7	5 040	6.52	12.69
8	40 320	60.97	13.37
9	362 880	505.31	14.09
10	3 628 800	5 593.33	16.22

Tabla 5.1: Tiempos ejecución de `mcstBird` según el método para $n = 6, 7, \dots, 10$.

La Figura 5.1 refleja gráficamente estos resultados. Dado que el método exacto crece de forma factorial, usaremos la escala logarítmica en el eje vertical para facilitar su correcta visualización y comparación, sin que los valores más bajos queden comprimidos.

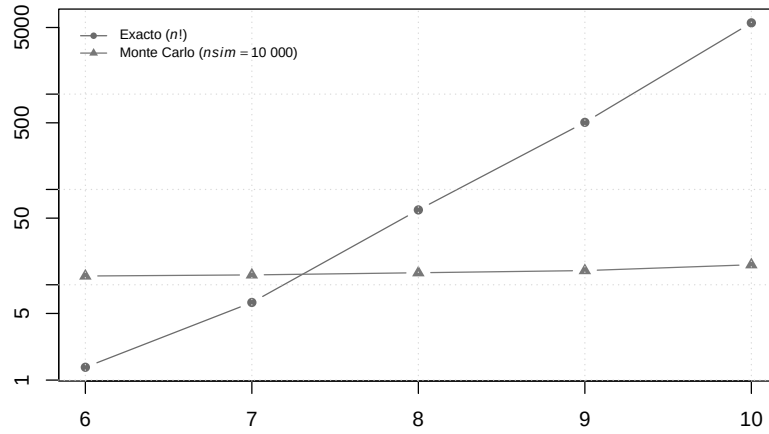


Figura 5.1: Tiempo de ejecución (escala logarítmica) del método exacto frente a Monte Carlo para la regla de Bird y $n = 6, 7, \dots, 10$.

La curva del método exacto en la Figura 5.1 es prácticamente una recta en escala logarítmica y crece de forma explosiva con cada agente que añadimos al problema. Por su parte, la curva de Monte Carlo se mantiene prácticamente plana con independencia de n . Fijémonos que el cruce entre ambas curvas se produce entre $n = 7$ y $n = 8$. Con 7 agentes el método exacto (6.52 s) todavía es más rápido que Monte Carlo (12.69 s), pero con 8 la situación se invierte (60.97 s frente a 13.37 s).

Centrémonos ahora en el resultado obtenido para $n = 10$. El método exacto tardó un total de 5 593.33 segundos (más de una hora y media) frente a los 16.22 segundos que tardó Monte Carlo, lo que implica que este último fue aproximadamente 345 veces más rápido. Además, para el método exacto, el tiempo requerido para pasar de $n = 9$ a $n = 10$ fue 11.07 veces superior, muy cercano al factor que se predice teóricamente en base a lo visto en la Sección 3.3 sobre el coste computacional para la regla extendida: $(10! \cdot 10^2)/(9! \cdot 9^2) = 12.35$.

Cabe destacar que para los casos $n = 6$ y $n = 7$ el número de permutaciones posibles es menor que el número de simulaciones fijado, por lo que carece de sentido muestrear con reemplazamiento cuando se puede recorrer simplemente el conjunto completo de permutaciones. En estos casos, la propia función lanza un aviso (`Warning: n! <= nsim. Consider using method = 'exact' for maximum efficiency`) recomendando el uso del método exacto. Este hecho se confirma en vista de los resultados

de la Tabla 5.1 y la Figura 5.1, en las que se ve que para ambos casos el método exacto resulta más rápido que la propia aproximación Monte Carlo.

Regla de Kar: coste exponencial frente a Monte Carlo

Para este caso, hemos ampliado el rango de n hasta 20, ya que el crecimiento en $2^n - 1$ coaliciones es exponencial, lo que nos permite validar tamaños de red superiores a los que el método exacto de Bird era capaz de abordar. La Tabla 5.2 recoge los tiempos obtenidos con `mcstKar` para $n = 12, 14, \dots, 20$.

n	Coaliciones ($2^n - 1$)	Exacto (s)	Monte Carlo (s)
12	4 095	4.56	23.36
14	16 383	19.37	29.29
16	65 535	88.68	41.32
18	262 143	409.27	43.71
20	1 048 575	1 890.44	54.47

Tabla 5.2: Tiempos de ejecución de `mcstKar` según el método para $n = 12, 14, \dots, 20$.

La Figura 5.2 permite visualizar los resultados anteriores. De nuevo, usaremos la escala logarítmica para el eje vertical.

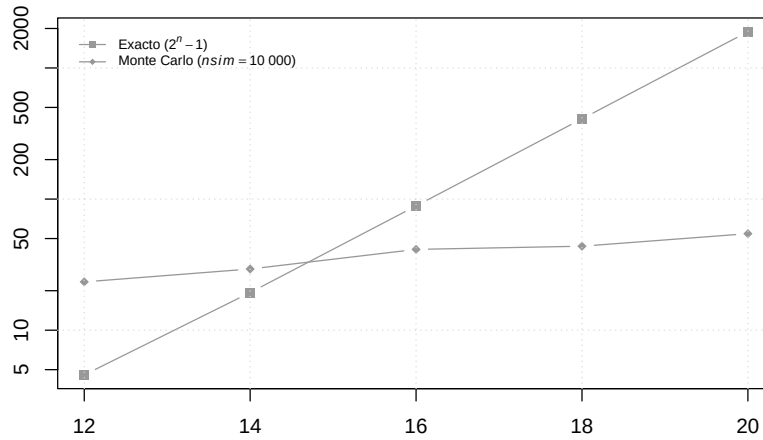


Figura 5.2: Tiempo de ejecución (escala logarítmica) del método exacto frente a Monte Carlo para la regla de Kar y $n = 12, 14, \dots, 20$.

En la Figura 5.2 se aprecia que el cruce entre ambos métodos se produce entre $n = 14$ y $n = 16$. Con 14 agentes el método exacto (19.37 s) sigue siendo más rápido que Monte Carlo (29.29 s), pero con 16 el exacto ya necesita 88.68 s frente a los 41.32 s de la aproximación.

De nuevo, para $n = 12$ y $n = 14$ (donde $2^n - 1$ es menor que $nsim \cdot n$), la función devuelve un aviso recomendando el método exacto, algo que se confirma en base a los resultados empíricos. Sin embargo, fijémonos que para $n = 16$ el número de evaluaciones del método exacto sería $2^{16} - 1 = 65\,535$, mientras que Monte Carlo realizaría $nsim \cdot n = 160\,000$ evaluaciones (más del doble) y aun así Monte Carlo resulta más rápido en la práctica (41.32 s frente a 88.68 s). Esto confirma lo señalado sobre que el

tiempo real no depende únicamente del número de evaluaciones, sino también del coste de cada una, que no tiene por qué ser el mismo en ambos casos. Por su parte, para $n = 20$ el método exacto tarda 1 890.44 segundos (más de media hora) frente a los 54.47 segundos de Monte Carlo, aproximadamente 34.7 veces más rápido.

Además, si comparamos el punto de corte para la regla de Kar, situado entre $n = 14$ y $n = 16$, con el de la regla de Bird, entre $n = 7$ y $n = 8$, se confirma lo que comentamos en un inicio sobre que el método exacto para el cálculo del valor de Shapley puede abordar con mayor facilidad tamaños de red muy superiores debido a que $2^n - 1$ crece mucho más despacio que $n!$.

Escalabilidad de Monte Carlo en redes grandes

Como hemos podido comprobar, el método exacto deja de ser una alternativa razonable a partir de cierto tamaño en ambas familias de funciones. En este apartado se analiza el comportamiento de Monte Carlo para redes de hasta 50 agentes. La Tabla 5.3 y la Figura 5.3 recogen los resultados.

n	Bird MC (s)	Kar MC (s)
10	15.60	16.50
20	28.11	57.31
30	41.55	130.53
40	57.86	240.09
50	76.95	373.40

Tabla 5.3: Tiempos de ejecución de Monte Carlo ($nsim = 10\,000$) para las reglas de Bird y Kar en redes de hasta $n = 50$ agentes.

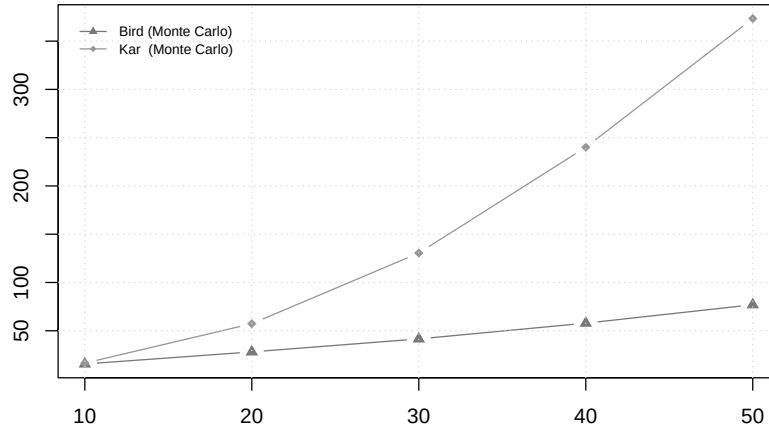


Figura 5.3: Escalabilidad de Monte Carlo ($nsim = 10\,000$) para las reglas de Bird y Kar en redes de hasta $n = 50$ agentes.

Ambas curvas crecen de forma mucho más suave que sus versiones exactas. En el caso de Bird se puede apreciar una tendencia lineal, mientras que para Kar la curva es más pronunciada pero su comportamiento está lejos de ser exponencial. Esto es así porque para la regla de Kar cada permutación muestreada requiere evaluar n subproblemas distintos, frente a una única resolución del algoritmo de

Prim para la regla de Bird. Aun así, cabe destacar que en ambos casos se consigue resolver una red de 50 agentes en menos de siete minutos, lo cual es un resultado muy satisfactorio.

Comparación con el paquete `cooptrees`

Por último, hemos querido comparar los tiempos de ejecución de la función `mcstBird` con la función equivalente del paquete `cooptrees` (Fontenla 2014a), distinguiendo dos escenarios: uno sin empates, donde ambos paquetes calculan exactamente el mismo árbol único y otro con empates, donde solo `mcstprules` (Souto Vázquez 2026) calcula la regla extendida. La Tabla 5.4 muestra los resultados de esta comparación:

Escenario	<code>mcstprules</code> (s)	<code>cooptrees</code> (s)
Sin empates ($n = 8$, árbol único)	0.002	0.007
Con empates ($n = 6$)	1.363	0.001
Con empates ($n = 7$)	6.521	0.001
Con empates ($n = 8$)	60.972	0.001

Tabla 5.4: Comparativa de tiempos entre `mcstprules` y `cooptrees`.

El resultado del escenario sin empates es una sorpresa favorable, pues `mcstprules` resulta más rápido que `cooptrees` (0.002 s frente a 0.007 s). Como vemos, el hecho de ofrecer más funcionalidades, como el cálculo de `arcs`, `percentage` o `perms` entre otros, no implica necesariamente un coste en el rendimiento.

En el escenario con empates, el tiempo de `cooptrees` se mantiene constante en torno a 0.001 s independientemente de n , porque su función se limita a resolver Prim una única vez deshaciendo cualquier empate de forma arbitraria, sin considerar el resto de árboles óptimos alternativos. Por su parte, `mcstprules` invierte más tiempo, ya que trata de detectar todos los empates y promediar sobre las $n!$ permutaciones, siguiendo la definición exacta de la regla extendida.

Por tanto, este aumento en el tiempo de ejecución por parte de `mcstprules` no implica que `cooptrees` sea más eficiente, sino que está resolviendo un problema distinto más simple.

5.2. Precisión de la aproximación Monte Carlo

Una vez comprobado que el método Monte Carlo resulta mucho más rápido que el método exacto a partir de un cierto tamaño de red, nos interesa analizar si los resultados que ofrece son verdaderamente fiables con respecto al valor exacto y cómo se reduce el error de aproximación a medida que aumentamos el número de permutaciones muestreadas. En esta sección, calcularemos empíricamente esos aspectos y los contrastaremos con la velocidad de convergencia $\mathcal{O}(nsim^{-1/2})$ establecida teóricamente en la Sección 3.3.

En particular, análogamente a la Sección 5.1, ilustraremos los resultados para las reglas de Bird (extendida) y Kar con $n = 10$ y $n = 20$, respectivamente. Usaremos como referencia el resultado exacto y calcularemos el error absoluto máximo de los repartos obtenidos para nueve niveles de `nsim` entre 100 y 50 000, promediando en cada uno 30 réplicas independientes del problema.

Cabe destacar que los resultados se representarán en escala log-log con el fin de comparar la tasa de convergencia teórica con la empírica. La explicación es que dado que $RMSE(nsim) = \mathcal{O}(nsim^{-1/2})$ por la Proposición 3.3.1, se puede escribir de forma equivalente que

$$RMSE(nsim) \approx c \cdot nsim^{-1/2}, \text{ para alguna constante } c > 0.$$

Aplicando logaritmos a ambos lados se obtiene una ecuación de la forma $Y \approx a + bX$:

$$\log(\text{RMSE}(nsim)) \approx \log(c) - \frac{1}{2} \log(nsim),$$

es decir, una relación lineal entre $Y = \log(\text{RMSE})$ y $X = \log(nsim)$ con pendiente $b = -1/2$. Para estimar esa pendiente empíricamente basta con ajustar una regresión lineal simple entre X e Y empleando la función `lm` de R, que devuelve directamente la estimación $\hat{b}$. Si la teoría es correcta, $\hat{b}$ debería ser próximo a -0.5 con independencia del valor de la constante c que no afecta a la pendiente de la relación.

Convergencia del error para la regla de Bird

La Tabla 5.5 recoge los resultados completos obtenidos para la regla de Bird en un problema con $n = 10$ agentes.

<code>nsim</code>	Error máx. medio	Error medio global	Desv. típica	Tiempo medio (s)
100	4.550	1.440	1.482	0.20
250	2.679	0.930	0.794	0.68
500	1.848	0.629	0.664	1.24
1 000	1.139	0.419	0.415	2.27
2 500	0.851	0.325	0.258	5.72
5 000	0.601	0.218	0.207	11.78
10 000	0.395	0.150	0.109	15.66
25 000	0.247	0.092	0.085	39.02
50 000	0.183	0.064	0.046	78.07

Tabla 5.5: Error de Monte Carlo frente al valor exacto para la regla de Bird con $n = 10$ y 30 réplicas por cada nivel de `nsim`.

En vista de los resultados de la tabla, es posible analizar el equilibrio entre precisión y tiempo de ejecución. Fijémonos que para pasar de `nsim` = 10 000 (el valor por defecto del paquete) a `nsim` = 50 000 se multiplica el tiempo de cómputo por 5 (de 15.66 s a 78.07 s), pero solo se reduce el error máximo medio de 0.395 a 0.183, que es menos de la mitad. A pesar de esto, el tiempo de ejecución para `nsim` = 50 000 no supera el minuto y medio y resulta en una aproximación más fiable, por lo que, quizá en este caso, sí merezca la pena usar un número grande de simulaciones. Aun y todo, dependiendo de la situación y del objetivo del usuario, la opción por defecto `nsim` = 10 000 podría resultar una elección razonable si lo que nos interesa es mantener una buena relación entre tiempo y precisión.

La Figura 5.4 muestra la gráfica asociada a los resultados de la tabla anterior:

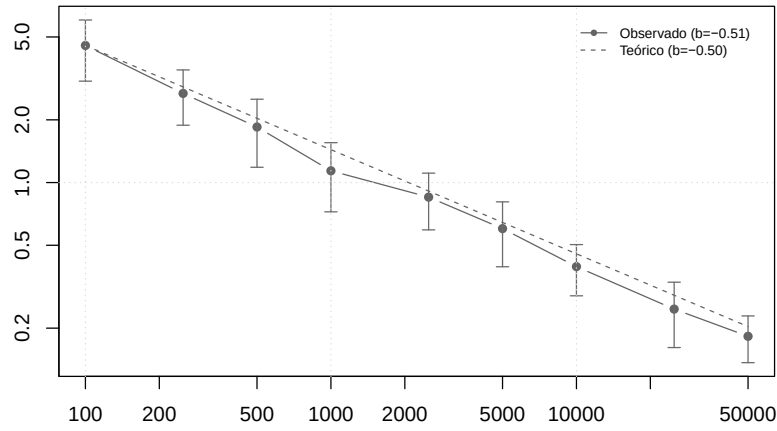


Figura 5.4: Convergencia de Monte Carlo para la regla de Bird ($n = 10$): error absoluto máximo medio frente a nsim (escala log-log), con las barras de desviación típica y la recta teórica de referencia con pendiente $b = -1/2$.

En el gráfico se puede apreciar que el error decrece de forma monótona y suave y que la desviación típica también disminuye a medida que aumentamos el número de simulaciones. Ajustando la regresión log-log descrita anteriormente sobre estos puntos, se obtiene una pendiente empírica de $\hat{b} = -0.513$, prácticamente igual al -0.5 que predice la teoría.

Las barras de la Figura 5.4 representan la desviación típica del error máximo observado entre las 30 réplicas independientes ejecutadas para cada nivel de nsim . Cabe destacar que en el gráfico parecen intervalos asimétricos debido a que la representación está en escala logarítmica, pero en realidad no lo son.

Convergencia del error para la regla de Kar

La Tabla 5.6 repite el mismo análisis para la regla de Kar en un problema con $n = 20$ agentes.

nsim	Error máx. medio	Error medio global	Desv. típica	Tiempo medio (s)
100	3.983	1.372	1.222	0.60
250	2.291	0.821	0.671	1.68
500	1.739	0.628	0.441	2.91
1000	1.133	0.422	0.238	5.81
2500	0.802	0.276	0.228	14.79
5000	0.503	0.186	0.104	29.13
10000	0.336	0.128	0.069	58.76
25000	0.226	0.081	0.051	144.88
50000	0.161	0.059	0.049	292.84

Tabla 5.6: Error de Monte Carlo frente al valor exacto para la regla de Kar con $n = 20$ (30 réplicas por cada nivel de nsim).

Comparando los tiempos de la Tabla 5.6 con los de la Tabla 5.5 para el mismo nsim se puede apreciar

que cada simulación de Kar cuesta más que una de Bird (292.84 s frente a 78.07 s para $\text{nsim} = 50\,000$), confirmando también aquí lo ya observado en la Sección 5.1.

En la Figura 5.5 se muestran los resultados gráficamente:

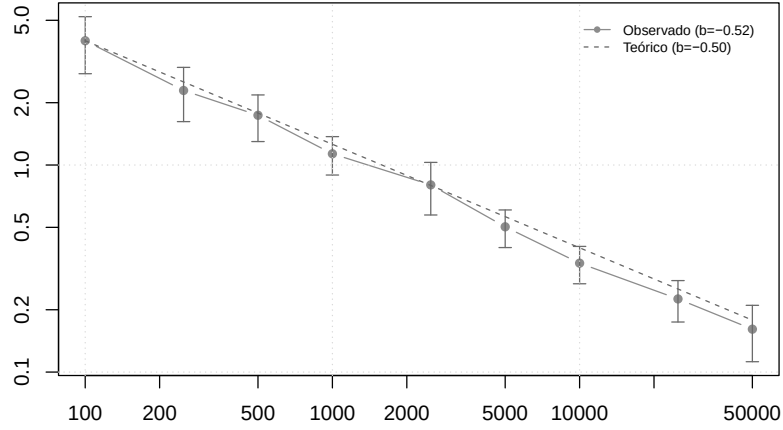


Figura 5.5: Convergencia de Monte Carlo para la regla de Kar ($n = 20$): error absoluto máximo medio frente a nsim (escala log-log), con las barras de desviación típica y la recta teórica de referencia con pendiente $b = -1/2$.

La pendiente empírica en este caso es $\hat{b} = -0.516$, de nuevo prácticamente igual al valor teórico $b = -0.5$. Esto confirma que el cambio de objeto promediado para el método Monte Carlo no afecta a la validez de las propiedades expuestas en la Proposición 3.3.1.

Al igual que ocurría con Bird, los intervalos representados en la Figura 5.5 son simétricos, pero se ven alterados por la escala de la representación.

Capítulo 6

Conclusiones y trabajo futuro

En este trabajo se ha hecho una revisión bibliográfica de los conceptos y reglas de reparto existentes para problemas de árboles de mínimo coste y se ha construido el paquete de R `mcstprules`, que reúne en una sola herramienta las dos grandes formas de repartir el coste que hemos tratado a lo largo de esta memoria: la vía algorítmica mediante Prim, Kruskal, Borůvka o la descomposición en conos y la vía basada en juegos cooperativos mediante el cálculo de la función característica y la aplicación de conceptos de solución.

La creación de este nuevo paquete en R nos ha permitido verificar en la práctica propiedades y resultados vistos en la teoría, tales como la equivalencia entre algunas reglas que fueron implementadas de forma independiente, y comprobar que efectivamente se trata de una herramienta robusta que es capaz de detectar correctamente las limitaciones de ciertos conceptos teóricos y devolver un resultado coherente. Además, resultan especialmente interesantes los resultados del Capítulo 5, que muestran cómo el método Monte Carlo propuesto permite solventar en gran medida los problemas de cómputo que surgían para problemas con un gran número de agentes al intentar calcular el resultado exacto, garantizando la validez de la aproximación.

Con todo esto, cabe destacar la enorme cantidad de pruebas por las que ha pasado este paquete hasta considerarse algo sólido. No solo debe comprobarse que las funciones producen el resultado final esperado, sino que internamente siguen la lógica correcta. Además, algunas pruebas permitieron detectar cuellos de botella computacionales, que obligaron a modificar ciertas funciones para hacer viable su uso. En conjunto, el paquete cumple el objetivo de ser una herramienta completa, ampliable y con un coste computacional razonable para estudiar este tipo de problemas, pero eso no significa que no admita mejoras que puedan valorarse para una próxima versión.

Para finalizar la memoria, se presentan algunas posibles líneas de trabajo futuro derivadas de los resultados obtenidos. La primera consistiría en optimizar en R el cálculo de la función `mcstBird`, que calcula la regla de Bird. Recordemos que esta función resuelve los empates evaluando de manera explícita las $n!$ permutaciones $\pi \in \Pi_N$. Esto supone un cuello de botella computacional que crece a medida que lo hace el número de agentes n . Para solventarlo, se crea la alternativa de Monte Carlo que muestrea dichas permutaciones y devuelve una aproximación válida y precisa de la solución.

Sin embargo, si analizamos con detenimiento la mayoría de problemas de este tipo, nos daremos cuenta que en el cálculo exacto de la regla extendida muchas de las permutaciones convergen exactamente al mismo árbol de coste mínimo y, por tanto, al mismo reparto. Para entender mejor a qué nos estamos refiriendo, consideremos el problema empleado a lo largo de la memoria dado por el Ejemplo 2.1.1. Tal y como ilustramos en el Ejemplo 1, en la etapa 1 del algoritmo de Prim se produce un empate entre dos de los arcos que conectan con la fuente, $c_{02} = c_{03} = 6$. Como consecuencia, la regla de Bird recurría

a su versión extendida para manejar la presencia de dichos empates. Una primera vía de mejora, no explorada en este trabajo, consistiría en paralelizar en R el cálculo de estas permutaciones, dado que cada una se resuelve de forma independiente del resto. No obstante, la Tabla 3.1 mostraba que había únicamente dos repartos diferentes para las $3! = 6$ permutaciones totales.

Fijémonos que, en lugar de evaluar exhaustivamente todas las permutaciones, se podría construir un árbol de decisión donde cada bifurcación o corte viniera dado por la forma en la que se deshace un determinado empate en la red, lo que permitiría reconstruir el reparto extendido a través de los repartos calculados en sus nodos terminales y del peso de cada rama, tal y como se muestra en la Figura 6.1.

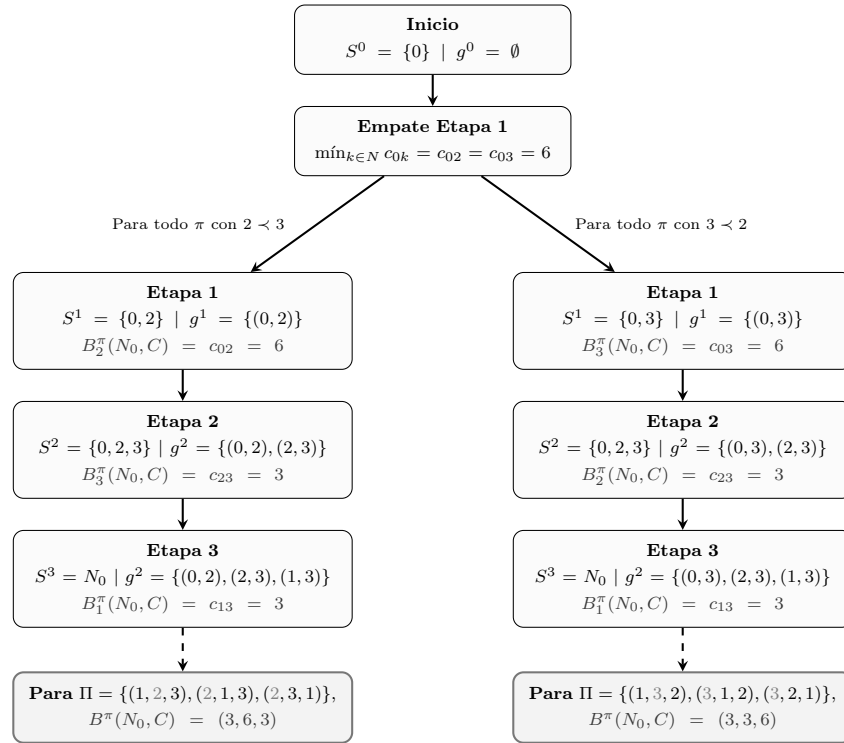


Figura 6.1: Árbol de decisión con dos ramas terminales obtenido para la regla de Bird al deshacer el empate entre los agentes 2 y 3 en la Etapa 1 del algoritmo de Prim. En azul, los pagos $B_i^\pi(N_0, C)$.

La ventaja de este enfoque es que los pesos de las ramas se pueden determinar de manera analítica mediante combinatoria, sin necesidad de ejecutar las permutaciones. Por ejemplo, la condición para llegar al reparto $(3, 6, 3)$ de la primera rama del árbol en la Figura 6.1 es que π cumpla que $2 < 3$, es decir, que ese reparto se obtiene para todas las ordenaciones en las que el agente 2 va antes que el 3. Matemáticamente, eso es exactamente la mitad de las permutaciones totales, es decir, $3!/2 = 3$, que es justo la cantidad de veces que se repite el reparto $(3, 6, 3)$ en la Tabla 3.1.

Existe la posibilidad de que para un cierto árbol haya ramas distintas que converjan al mismo reparto, pero aunque esto fuera así el ahorro computacional seguiría siendo notable comparado con $n!$.

Es importante destacar que esta idea de ramificación no se podría trasladar directamente a la regla de Dutta-Kar. Para esta regla, las mismas seis permutaciones se distribuyen de manera asimétrica (2 para un reparto y 4 para otro), tal y como mostramos en la Tabla 3.2, por lo que el razonamiento expuesto en la Figura 6.1 no tendría sentido aquí.

Cabe destacar que esta idea que acabamos de exponer es únicamente una propuesta inicial que no

debería intepretarse como algo establecido, sino como algo que requiere una futura investigación para valorar su viabilidad. Por otro lado, se podrían plantear mejoras analíticas y de visualización. Actualmente, la función `mcstCorePlot` restringe la representación geométrica del núcleo a problemas con un máximo de cuatro agentes debido a que no somos capaces de visualizar estructuras en más de tres dimensiones. Por tanto, una práctica interesante podría ser explorar técnicas de reducción de la dimensionalidad para extender la representación a dimensiones superiores.

Finalmente, una última propuesta sugeriría la revisión permanente del código fuente para su optimización general de cara a futuras versiones del paquete, ya sea para mejorar la gestión de la memoria o para reescribir alguna rutina algorítmica mejorable desde el punto de vista computacional.

Apéndice A

Manual del paquete `mcstprules`

En este apéndice se incluye el manual de referencia completo correspondiente al paquete `mcstprules`. En él se documenta en detalle cada una de las funciones implementadas, especificando sus argumentos y valores devueltos, y se presenta un ejemplo ilustrativo de su uso práctico.

Package ‘mcstprules’

July 15, 2026

Type Package

Title Cost Allocation Rules in Minimum Cost Spanning Tree Problems

Version 0.1.0

Description Computes, analyzes, and visualizes cost allocation rules for Minimum Cost Spanning Tree Problems (MCSTP). The package implements both algorithmic methods and rules based on cooperative game theory, alongside a comprehensive suite of analytical tools to evaluate core stability, geometric properties, and cost sensitivity.

License GPL (>= 3)

Encoding UTF-8

LazyData true

RoxygenNote 7.3.3

Imports gtools,
igraph,
readxl,
CoopGame,
fields,
geometry,
lpSolve,
plotly,
rcdd

URL <https://github.com/luciasouto13/mcstprules>, <https://luciasouto13.github.io/mcstprules/>

BugReports <https://github.com/luciasouto13/mcstprules/issues>

Suggests knitr,
rmarkdown

VignetteBuilder knitr

Contents

mcstBird	2
mcstBoruvka	5
mcstCompare	7
mcstCone	8
mcstCoreCheck	10

mcstCorePlot	12
mcstCorePoints	14
mcstDuttaKar	15
mcstFolk	18
mcstGameCC	20
mcstGameIrred	24
mcstGameOpt	27
mcstGamePrivate	30
mcstGamePublic	33
mcstKar	36
mcstOWShapley	38
mcstPWShapley	40
mcstRules	42
mcstSensitivity	44
mcstStabilityRange	46

Index	48
--------------	-----------

mcstBird	<i>Bird's Rule for MCSTP</i>
----------	------------------------------

Description

This function computes Bird's cost allocation (Bird, 1976) for a minimum cost spanning tree problem (N_0, C) . The Bird rule, denoted as $B(N_0, C)$, assigns to each agent the cost of the arc that connects them to the source during the construction of a minimum cost spanning tree using Prim's algorithm.

Usage

```
mcstBird(
  C,
  draw = FALSE,
  which.plot = c("main", "details"),
  titles = TRUE,
  method = c("exact", "montecarlo"),
  nsim = 10000
)
```

Arguments

C	cost matrix between nodes. Accepts multiple formats (see "Supported formats for C" below).
draw	logical; if TRUE, plots the network highlighting an optimal tree in red, indicating the stage at which each arc is added (in brackets) and the cost allocated to each agent (in parentheses below the node). For $n \leq 3$ with ties and method = "exact", it also displays all possible trees.
which.plot	character string indicating which plots to display (only for $n \leq 3$ with ties and method = "exact"). If "details", displays the detailed breakdown of all trees and allocations according to the agents' entry orders; if "main", the average allocation is plotted. Default is c("main", "details").

titles	logical; if TRUE (default), adds a main title specifying the allocation rule and a subtitle with the algorithm, method (exact or Monte Carlo), and the total network cost.
method	character string specifying the calculation method when ties occur. If "exact" (default), computes all $n!$ permutations; if "montecarlo", estimates the average allocation through random sampling using n_{sim} permutations (recommended for large networks).
nsim	integer; the number of permutations to sample when using method = "montecarlo". Default is 10,000.

Details

For a given tree g^n constructed following Prim's algorithm, the allocation for each agent $i \in N$ is given by:

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

where i^0 is the node to which agent i is first connected in the process of building the network.

When the minimum cost spanning tree is not unique, the rule is extended by averaging the allocations over all possible permutations $\pi \in \Pi_N$ (method = "exact"):

$$B(N_0, C) = \frac{1}{n!} \sum_{\pi \in \Pi_N} B^\pi(N_0, C),$$

where $B^\pi(N_0, C)$ is the allocation obtained by applying Prim's algorithm to (N_0, C) and solving indifferences by selecting the first agent given by the permutation π , as proposed by Dutta and Kar (2004).

Alternatively, method = "montecarlo" estimates the average allocation through random sampling using n_{sim} permutations. This is highly useful when the number of agents is large, as it significantly reduces computation time while providing a very accurate approximation:

$$B(N_0, C) \approx \frac{1}{n_{\text{sim}}} \sum_{k=1}^{n_{\text{sim}}} B^{\pi_k}(N_0, C),$$

where π_k represents each of the randomly sampled permutations. For networks with few agents, the exact method is preferred to avoid potential overestimations.

Value

A list containing:

- bird: the Bird's allocation vector $B(N_0, C)$ for the natural ordering.
- e_bird: the extended Bird's allocation (average over permutations if ties exist; equal to bird otherwise).
- arcs: if the MCST is unique, a data frame of edges (i, j) in the tree and the stage at which they were added; if ties exist, a list of such data frames, one per distinct optimal tree.
- total: the total cost of the MCST, $m(N_0, C)$.
- percentage: the share of the total cost allocated to each agent, based on e_bird.
- ranking: a ranking of agents by cost in e_bird (from highest to lowest; ties marked with *).

- **perms**: if ties exist, a list with \$table (allocations for at most the first 6 permutations computed, regardless of n), \$summary (distinct allocations with their multiplicity, sorted in decreasing order) and \$nperms (total number of permutations computed); NULL if the MCST is unique.
- **is_unique**: logical; if TRUE, the MCST is unique.
- **method**: the method applied for computing the extended rule ("exact" or "montecarlo").

Supported formats for C

Note: In all cases, Inf is accepted for disconnected node pairs. The first row/column is assumed to be the source node (0).

The function accepts the following formats for the cost input:

Numeric vector The lower triangle of a symmetric cost matrix, excluding the diagonal. Length must be $n(n+1)/2$, where n is the number of agents (excluding the source).

Square matrix / Data frame A standard adjacency matrix where $C[i, j]$ represents the cost between node i and node j .

Edge list (matrix or data frame) An alternative to the full matrix. It must contain columns named "from" and "to", plus a third column for the weights/costs.

'igraph' object A weighted undirected graph. See the igraph package for details on creating these objects.

File path A string pointing to a .csv, .xlsx, or .xls file. The file can contain either a square matrix (without headers) or an edge list (with "from" and "to" headers).

References

Bergantiños G, Vidal-Puga J (2021) A review of cooperative rules and their associated algorithms for minimum-cost spanning tree problems. *SERIEs*, 12:73-100

Bird C (1976) On cost allocation for a spanning tree: a game theoretic approach. *Networks* 6(4):335–350.

Dutta B, Kar A (2004) Cost monotonicity, consistency and minimum cost spanning tree games. *Games Econom Behav* 48(2):223–248.

See Also

mcstDuttaKar for other rule based on Prim's algorithm.

mcstGameIrred for a cooperative game whose Shapley value coincides with Bird's solution when applied to the irreducible matrix $\bar{C}$.

mcstRules for an overview of the available rules and analysis tools in the package.

Examples

```
# Simple vector input
mcstBird(c(12, 15, 20, 4, 6, 8), draw = TRUE)

# Input with infinite costs (disconnected nodes)
C_inf <- c(5, 9, Inf, 15, 6, Inf, 7, Inf, Inf, Inf,
          Inf, 8, 7, Inf, Inf, 5, Inf, Inf, 8, 9, 11)
mcstBird(C_inf)

# Matrix input with ties
```

```
C_mat <- matrix(c(0, 12, 15, 12,
                  12, 0, 4, 6,
                  15, 4, 0, 8,
                  12, 6, 8, 0), byrow = TRUE, ncol = 4)
mcstBird(C_mat, draw = TRUE, which.plot = "main", titles = FALSE)
```

mcstBoruvka	<i>Boruvka's Rule for MCSTP</i>
-------------	---------------------------------

Description

This function computes Boruvka's cost allocation (Bergantiños and Vidal- Puga, 2011) for a minimum cost spanning tree problem (N_0, C) . Boruvka's rule, denoted as $\beta^\pi(N_0, C)$, is defined through Boruvka's algorithm.

Usage

```
mcstBoruvka(C, draw = FALSE, titles = TRUE)
```

Arguments

C	cost matrix between nodes. Accepts multiple formats (see "Supported formats for C" below).
draw	logical; if TRUE, plots the network highlighting an optimal tree in red, indicating the stage at which each arc is added (in brackets) and the cost allocated to each agent (in parentheses below the node).
titles	logical; if TRUE (default), adds a main title specifying the allocation rule and a subtitle with the algorithm used and the total network cost.

Details

The Boruvka allocation divides the cost of the optimal tree through a step-by-step process over γ stages. At each stage s , agents pay the maximum possible proportion p^s of the cheapest arc associated with their connected component.

Let A^s be the set of non-completely paid arcs, q_{ij}^s the proportion of the cost of arc (i, j) already paid, and N_{ij}^s the set of agents paying for arc (i, j) in stage s . The proportion p^s is given by:

$$p^s = \min \left\{ \frac{1 - q_{ij}^{s-1}}{|N_{ij}^s|} : (i, j) \in A^{s-1}, N_{ij}^s \neq \emptyset \right\}.$$

Let a_i^s be the arc partially paid by agent i in stage s . The cost paid by agent i in this stage is:

$$f_i^s = p^s c_{a_i^s}.$$

The process finishes in γ stages when the tree is completely paid, i.e., $\sum_{s=1}^{\gamma} p^s = 1$. The Boruvka allocation for each agent $i \in N$ is given by:

$$\beta_i^\pi(N_0, C) = \sum_{s=1}^{\gamma} f_i^s.$$

Value

A list containing:

- **boruvka**: the Boruvka's allocation vector $\beta^\pi(N_0, C)$.
- **arcs**: a data frame of edges (i, j) in the final tree and the stage at which they were added.
- **total**: the total cost of the MCST, $m(N_0, C)$.
- **percentage**: the share of the total cost allocated to each agent.
- **ranking**: a ranking of agents by cost (from highest to lowest; ties marked with *).
- **is_unique**: logical; if TRUE, the MCST is unique.

Supported formats for C

Note: In all cases, Inf is accepted for disconnected node pairs. The first row/column is assumed to be the source node (0).

The function accepts the following formats for the cost input:

Numeric vector The lower triangle of a symmetric cost matrix, excluding the diagonal. Length must be $n(n+1)/2$, where n is the number of agents (excluding the source).

Square matrix / Data frame A standard adjacency matrix where $C[i, j]$ represents the cost between node i and node j .

Edge list (matrix or data frame) An alternative to the full matrix. It must contain columns named "from" and "to", plus a third column for the weights/costs.

'igraph' object A weighted undirected graph. See the *igraph* package for details on creating these objects.

File path A string pointing to a .csv, .xlsx, or .xls file. The file can contain either a square matrix (without headers) or an edge list (with "from" and "to" headers).

Note

Boruvka's allocation is uniquely determined even when the minimum cost spanning tree is not unique. Since the rule depends on the evolution of the connected components rather than the specific arcs selected, any tie-breaking selection in Boruvka's algorithm yields the same cost distribution.

Furthermore, as demonstrated by Bergantiños and Vidal-Puga (2011), for any order π used in the algorithm, Boruvka's allocation is equivalent to the `mcstFolk`.

References

Bergantiños G, Vidal-Puga J (2011) The folk solution and Boruvka's algorithm in minimum cost spanning tree problems. *Discrete Appl Math* 159(12):1279–1283

Bergantiños G, Vidal-Puga J (2021) A review of cooperative rules and their associated algorithms for minimum-cost spanning tree problems. *SERIEs*, 12:73-100

See Also

`mcstFolk` for an equivalent rule based on Kruskal's algorithm.

`mcstRules` for an overview of the available rules and analysis tools in the package.

Examples

```
# Simple vector input
mcstBoruvka(c(12, 15, 20, 4, 6, 8), draw = TRUE)

# Input with infinite costs (disconnected nodes)
C_inf <- c(5, 9, Inf, 15, 6, Inf, 7, Inf, Inf, Inf,
          Inf, 8, 7, Inf, Inf, 5, Inf, Inf, 8, 9, 11)
mcstBoruvka(C_inf)

# Matrix input
C_mat <- matrix(c(0, 12, 15, 12,
                  12, 0, 4, 6,
                  15, 4, 0, 8,
                  12, 6, 8, 0), byrow = TRUE, ncol = 4)
mcstBoruvka(C_mat, draw = TRUE, titles = FALSE)
```

mcstCompare

Compare Allocation Rules for MCST

Description

This function computes and compares the cost allocations for a minimum cost spanning tree (MCST) problem using four standard rules: Bird, Dutta-Kar, folk, and Kar.

Usage

```
mcstCompare(C)

## S3 method for class 'mcstp_compare'
plot(x, titles = TRUE, ...)
```

Arguments

C	a cost matrix between nodes. Accepts multiple formats (see "Supported formats for C" in <code>mcstBird</code>).
x	the output from <code>mcstCompare</code> .
titles	logical; if TRUE (default), adds informative main titles and axis labels to the plot.
...	additional graphical parameters passed to <code>barplot</code> .

Value

A data frame containing the allocations for each agent under the Bird, Dutta-Kar, folk, and Kar rules.

See Also

`mcstBird`, `mcstDuttaKar`, `mcstFolk`, `mcstKar` for the rule definitions.
`mcstRules` for an overview of the available rules and analysis tools in the package.

Examples

```
# Matrix input
C_mat <- matrix(c(0, 10, 15, 20,
                  10, 0, 25, 12,
                  15, 25, 0, 8,
                  20, 12, 8, 0), byrow = TRUE, ncol = 4)

# Compare all rules
comparison <- mcstCompare(C_mat)
comparison

# Plot the comparison
plot(comparison)
```

mcstCone

Cone-wise Decomposition Rules for MCSTP

Description

This function computes cost allocations for minimum cost spanning tree problems (N_0, C) using the cone-wise decomposition introduced by Norde et al. (2004). This technique allows extending rules defined for elementary MCSTPs to any MCSTP.

Usage

```
mcstCone(
  C,
  rule = c("folk", "owshapley", "bogomolnaia"),
  weights = NULL,
  lambda = 1
)
```

Arguments

<code>C</code>	cost matrix between nodes. Accepts multiple formats (see "Supported formats for C" below).
<code>rule</code>	a character string indicating the rule R to be applied to the elementary MCSTPs. One of "folk" (default), "owshapley", or "bogomolnaia".
<code>weights</code>	a numeric vector of strictly positive weights for each agent in N . The length of the vector must match the number of agents; only if <code>rule = "owshapley"</code> .
<code>lambda</code>	a non-negative numeric parameter (1 by default); only if <code>rule = "bogomolnaia"</code> .

Details

Any MCSTP can be written as a nonnegative combination of elementary MCST $C = \sum_{q=1}^{m(C)} x^q C^q$, where the costs of the arcs in C^q are 0 or 1. A rule R is extended as:

$$R(N_0, C) = \sum_{q=1}^{m(C)} x^q R(N_0, C^q).$$

Let g^q be the network at stage $q \in \{1, \dots, m(C)\}$, and $S_i^q := S(P(g^q), i)$ the connected component containing agent i in the graph induced by zero-cost arcs. For each elementary problem (N_0, C^q) , the agents in a component S not containing the source (0) divide the cost of connecting to the source according to the chosen rule:

- Folk rule ("folk"):

$$F(N_0, C^q) = \begin{cases} \frac{1}{|S_i^q|} & \text{if } 0 \notin S_i^q \\ 0 & \text{otherwise.} \end{cases}$$

- Optimistic weighted Shapley rule ("owshapley"):

$$f^{ow}(N_0, C^q) = \begin{cases} \frac{w_i}{\sum_{j \in S_i^q} w_j} & \text{if } 0 \notin S_i^q \\ 0 & \text{otherwise.} \end{cases}$$

- Bogomolnaia and Moulin family ("bogomolnaia"):

For $\lambda \in [0, +\infty)$,

$$R^\lambda(N_0, C^q) = \begin{cases} \frac{\lambda^{\delta_i}}{\sum_{j \in S_i^q} \lambda^{\delta_j}} & \text{if } 0 \notin S_i^q \\ 0 & \text{otherwise,} \end{cases}$$

where δ_i denotes the number of non-null arcs in C^q containing agent i .

For $\lambda = +\infty$,

$$R^\lambda(N_0, C^q) = \begin{cases} \arg \max_{j \in S_i^q} \delta_j & \text{if } 0 \notin S_i^q \\ 0 & \text{otherwise,} \end{cases}$$

i.e., agents attaining the maximum value of δ_j share the cost equally.

Value

A list containing:

- allocation: the cost allocation vector $R(N_0, C)$ for the chosen rule R .
- rule: the rule R applied for the allocation.
- total: the total cost of the MCST, $m(N_0, C)$.
- percentage: the share of the total cost allocated to each agent.
- ranking: a ranking of agents by cost (from highest to lowest; ties marked with *).
- decomposition: a data frame containing the details of the cone-wise decomposition, including the cost threshold, the weight of each layer and the elementary allocation for each agent.
- weights: the weight vector used for the allocation; only if rule = "owshapley".
- lambda: the lambda value used for the allocation; only if rule = "bogomolnaia".

Supported formats for C

Note: In all cases, Inf is accepted for disconnected node pairs. The first row/column is assumed to be the source node (0).

The function accepts the following formats for the cost input:

Numeric vector The lower triangle of a symmetric cost matrix, excluding the diagonal. Length must be $n(n+1)/2$, where n is the number of agents (excluding the source).

Square matrix / Data frame A standard adjacency matrix where $C[i, j]$ represents the cost between node i and node j .

Edge list (matrix or data frame) An alternative to the full matrix. It must contain columns named "from" and "to", plus a third column for the weights/costs.

'igraph' object A weighted undirected graph. See the `igraph` package for details on creating these objects.

File path A string pointing to a .csv, .xlsx, or .xls file. The file can contain either a square matrix (without headers) or an edge list (with "from" and "to" headers).

Note

This function focuses on the analytical and numerical decomposition of the cost allocation. For graphical representations, use the direct rule functions (e.g., `mcstFolk`).

The allocation for rule = "folk" is also obtained when all weights are equal in rule = "owshapley", or when $\lambda = 1$ in rule = "bogomolnaia".

References

Bergantiños G, Vidal-Puga J (2021) A review of cooperative rules and their associated algorithms for minimum-cost spanning tree problems. *SERIEs*, 12:73-100

Norde H, Moretti S, Tijs S (2004) Minimum cost spanning tree games and population monotonic allocation schemes. *Eur J Oper Res* 154(1):84-97

See Also

`mcstFolk`, `mcstOWShapley` for direct implementations of these rules.

`mcstRules` for an overview of the available rules and analysis tools in the package.

Examples

```
C <- c(12, 15, 20, 4, 6, 8)

# Folk rule
mcstCone(C, rule = "folk")

# Optimistic weighted Shapley rule
mcstCone(C, rule = "owshapley", weights = c(2, 1, 1))

# Bogomolnaia and Moulin family
mcstCone(C, rule = "bogomolnaia", lambda = 2)
mcstCone(C, rule = "bogomolnaia", lambda = Inf)
```

mcstCoreCheck

Core Stability Check for Cost Allocations

Description

This function evaluates whether a given cost allocation belongs to the *core* of a cooperative game. It checks for both efficiency and coalitional rationality, identifying any blocking coalitions if the allocation is unstable.

Usage

```
mcstCoreCheck(game = NULL, allocation = NULL, v = NULL, tol = 1e-07)
```

```
## S3 method for class 'mcstp_core'
plot(x, col = "firebrick", titles = TRUE, ...)
```

Arguments

game	list; an object containing a cooperative game computed with method = "exact" (e.g., from mcstGamePrivate). Default is NULL.
allocation	numeric vector; a proposed cost allocation for each agent. Required if game is NULL.
v	numeric vector; the characteristic function values $v(S)$ for all $2^n - 1$ coalitions, ordered by cardinality. Required if game is NULL.
tol	numeric; tolerance for floating-point comparisons to avoid precision issues. Default is $1e-7$.
x	the output from mcstCoreCheck.
col	character string specifying the color for the excess bars. Default is "firebrick".
titles	logical; if TRUE (default), adds a main title and a y-axis label to the plot.
...	additional graphical parameters passed to barplot.

Details

An allocation $x \in \mathbb{R}^N$ belongs to the *core* of a cooperative game (N, v) if it satisfies two fundamental conditions:

1. Efficiency: the total cost is completely allocated among the agents, i.e.,

$$\sum_{i \in N} x_i = v(N).$$

2. Coalitional rationality: no coalition pays more than its stand-alone cost, i.e.,

$$\sum_{i \in S} x_i \leq v(S) \quad \forall S \subset N.$$

If the coalitional rationality condition is violated for any coalition S , S is considered a *blocking coalition*, as its members would have an incentive to leave the grand coalition and form their own sub-network. The excess $x(S) - v(S)$ quantifies the magnitude of this violation.

The function can be executed in two ways: by passing a complete game object (which must have been computed with method = "exact"), or by manually providing the proposed allocation vector and the characteristic function v .

The plot method visualizes the excess of the blocking coalitions (if any) using a bar chart.

Value

A list containing:

- in_core: logical; if TRUE, the allocation is stable (belongs to the *core*).
- is_efficient: logical; if TRUE, the sum of the allocation equals the total cost.

- `is_rational`: logical; if TRUE, no blocking coalitions exist.
- `blocking_coals`: a data frame detailing the blocking coalitions S , their allocated cost $x(S)$, their value $v(S)$, and the excess. Empty if the allocation is rational.
- `n`: the number of agents.
- `sum_x`: the sum of the allocated costs.
- `total`: the total cost of the game $v(N)$.

References

Bergantiños G, Vidal-Puga J (2021) A review of cooperative rules and their associated algorithms for minimum-cost spanning tree problems. *SERIEs*, 12:73-100

See Also

`mcstGamePrivate`, `mcstGameIrred` for computing cooperative games that can be passed to this function.

`mcstRules` for an overview of the available rules and analysis tools in the package.

Examples

```
# Example 1: Using manual vectors
vvals <- c(10, 12, 15, 20, 22, 25, 30) # 3 agents: v(1), v(2), v(3), v(12), ...
# Stable check
alloc1 <- c(8, 10, 12)
check1 <- mcstCoreCheck(allocation = alloc1, v = vvals); check1
# Unstable check
alloc2 <- c(15, 10, 5)
check2 <- mcstCoreCheck(allocation = alloc2, v = vvals); check2
plot(check2)

# Example 2: Using a game object
C <- matrix(c(0, 12, 15, 12, 0, 4, 15, 4, 0), byrow = TRUE, ncol = 3)
# Check if the Shapley value of the private game is in the core
mcstCoreCheck(game = mcstGamePrivate(C))
```

`mcstCorePlot`

Plot the Core Region and Cost Allocations

Description

This function generates a geometric visualization of the *core* region and the imputation set for cooperative games with 2, 3, or 4 players. It also allows plotting multiple cost allocation rules simultaneously to visually check their stability.

Usage

```
mcstCorePlot(game = NULL, v = NULL, allocations = list(), titles = TRUE)
```

Arguments

game	list; an object containing a cooperative game (e.g., from <code>mcstGamePrivate</code>). If provided and <code>allocations</code> is empty, its allocation and solution concept are used by default. Default is <code>NULL</code> .
v	numeric vector; the characteristic function values $v(S)$ for all $2^n - 1$ coalitions, ordered by cardinality. Required if <code>game</code> is <code>NULL</code> .
allocations	list; a named list of numeric vectors representing different cost allocations to be plotted as points. Default is <code>list()</code> .
titles	logical; if <code>TRUE</code> (default), adds a main title and subtitle/legend information to the plot.

Details

The geometric representation of the *core* and the imputation set depends on the number of players n :

1. $n = 2$ players: the imputation set is represented as a 1D line segment where $x_1 + x_2 = v(N)$. The *core* is highlighted as a sub-segment within it, delimited by individual rationality.
2. $n = 3$ players: allocations are projected into 2D barycentric coordinates (simplex). The imputation set forms a large triangle, and the *core* is plotted inside as a bounded convex polygon.
3. $n = 4$ players: the function switches to an interactive 3D plot using the `plotly` package. The imputation set is drawn as a transparent 3D tetrahedron (mesh), and the *core* is represented as a solid 3D polytope inside it.

If an allocation lies inside the green *core* region, it means the allocation is coalitionally stable (no group of players has incentives to defect).

The function automatically extracts the default allocation if a `game` object is passed. Alternatively, a custom named `list` of allocations can be provided to compare different rules (e.g., Bird, Shapley value, folk) on the same graph.

Value

For $n = 2$ or $n = 3$, the function draws a base R plot. For $n = 4$, it returns an interactive `plotly` object containing the 3D visualization.

See Also

`mcstCoreCheck` for checking stability numbers and finding blocking coalitions.

`mcstRules` for an overview of the available rules and analysis tools in the package.

Examples

```
# Example 1
vvals1 <- c(10, 12, 15, 20, 22, 25, 30) # 3 players: v(1), v(2), v(3), v(12), ...
allocs1 <- list(RuleA = c(8, 10, 12), RuleB = c(9, 10, 11), RuleC = c(7, 11, 12))
mcstCorePlot(v = vvals1, allocations = allocs1)

# Example 2 (interactive 3D plot)
vvals2 <- c(10, 10, 10, 10, 18, 18, 18, 18, 18,
           18, 25, 25, 25, 25, 32) # 4 players: v(1), v(2), v(3), v(4), v(12), ...
allocs2 <- list(RuleA = c(8, 8, 8, 8), RuleB = c(10, 6, 11, 5))
```

```
mcstCorePlot(v = vvals2, allocations = allocs2)
```

mcstCorePoints

Check Core Emptiness

Description

This function determines whether the *core* of a cooperative game is empty by solving a Linear Programming (LP) problem. If the *core* is non-empty, it returns a feasible stable allocation and computes all the geometric vertices of the *core* polytope.

Usage

```
mcstCorePoints(game = NULL, v = NULL)

## S3 method for class 'mcstp_core_point'
plot(x, titles = TRUE, ...)
```

Arguments

game	list; an object containing a cooperative game computed with method = "exact" (e.g., from mcstGamePrivate). Default is NULL.
v	numeric vector; the characteristic function values $v(S)$ for all $2^n - 1$ coalitions, ordered by cardinality. Required if game is NULL.
x	the output from mcstCorePoints.
titles	logical; if TRUE (default), adds a main title and subtitle/legend information to the plot.
...	additional graphical parameters passed to mcstCorePlot.

Details

The problem of finding a cost allocation $x \in \mathbb{R}^N$ in the *core* is formulated as a Linear Programming system. The allocation must satisfy two fundamental conditions:

1. Efficiency: the total cost is completely allocated among the agents, i.e.,

$$\sum_{i \in N} x_i = v(N).$$

2. Coalitional rationality: no coalition pays more than its stand-alone cost, i.e.,

$$\sum_{i \in S} x_i \leq v(S) \quad \forall S \subset N.$$

The function uses the lpSolve package to evaluate this system of equations and inequalities. If a solution exists, the *core* is non-empty and a feasible allocation is returned.

Furthermore, for games with up to 6 players ($n \leq 6$), the function automatically utilizes the rcdd package to perform vertex enumeration. It converts the system of linear inequalities into the exact geometric vertices of the *core*. This is particularly useful for analyzing the bounds of the *core* region.

The plot method provides a visualization of the *core* and the computed feasible point by internally calling mcstCorePlot. It is supported for games with 2, 3, or 4 players.

Value

A list containing:

- `is_empty`: logical; if TRUE, the *core* is empty (no stable allocation satisfies all conditions).
- `core_point`: numeric vector; a feasible allocation inside the *core*. NULL if the *core* is empty.
- `vertices`: a matrix where each row represents a geometric vertex of the *core* polytope. NULL if the *core* is empty or $n > 6$.
- `n`: the number of agents.
- `total`: the total cost of the game $v(N)$.
- `values`: the characteristic function values $v(S)$.

See Also

`mcstCoreCheck` for checking the stability of a specific allocation.

`mcstCorePlot` for general *core* region visualizations.

`mcstRules` for an overview of the available rules and analysis tools in the package.

Examples

```
# Example 1: Non-empty core
vvals1 <- c(10, 12, 15, 20, 22, 25, 30) # 3 players
cp1 <- mcstCorePoints(v = vvals1); cp1
cp1$vertices
plot(cp1)

# Example 2: Empty core
vvals2 <- c(10, 10, 10, 15, 15, 15, 30)
cp2 <- mcstCorePoints(v = vvals2); cp2

# Example 3: Using a game object
C <- matrix(c(0, 12, 15, 12, 0, 4, 15, 4, 0), byrow = TRUE, ncol = 3)
mcstCorePoints(game = mcstGamePrivate(C))
```

Description

This function computes Dutta-Kar's cost allocation (Dutta and Kar, 2004) for a minimum cost spanning tree problem (N_0, C) . The Dutta-Kar rule, denoted as $DK(N_0, C)$, is defined through Prim's algorithm with a pivotal switch in the allocation cost at each step to satisfy cost monotonicity.

Usage

```
mcstDuttaKar(
  C,
  draw = FALSE,
  which.plot = c("main", "details"),
  titles = TRUE,
  method = c("exact", "montecarlo"),
  nsim = 10000
)
```

Arguments

C	cost matrix between nodes. Accepts multiple formats (see "Supported formats for C" below).
draw	logical; if TRUE, plots the network highlighting an optimal tree in red, indicating the stage at which each arc is added (in brackets) and the cost allocated to each agent (in parentheses below the node). For $n \leq 3$ with ties and method = "exact", it also displays all possible trees.
which.plot	character string indicating which plots to display (only for $n \leq 3$ with ties and method = "exact"). If "details", displays the detailed breakdown of all trees and allocations according to the agents' entry orders; if "main", the average allocation is plotted. Default is c("main", "details").
titles	logical; if TRUE (default), adds a main title specifying the allocation rule and a subtitle with the algorithm, method (exact or Monte Carlo), and the total network cost.
method	character string specifying the calculation method when ties occur. If "exact" (default), computes all $n!$ permutations; if "montecarlo", estimates the average allocation through random sampling using nsim permutations (recommended for large networks).
nsim	integer; the number of permutations to sample when using method = "montecarlo". Default is 10,000.

Details

Following Prim's algorithm, let $x^0 = 0$. At each stage $p \in \{1, \dots, n\}$, the algorithm selects the minimum cost arc (i^p, j^p) that connects a new agent j^p to the already connected set of nodes. We update the accumulated cost as:

$$x^p = \max\{x^{p-1}, c_{i^p j^p}\}.$$

For each agent $i \in N$, there exists a stage $p(i)$ such that $i = j^{p(i)}$. Dutta-Kar's allocation for agent $i \in N$ is defined as:

$$DK_i(N_0, C) = \min\{x^{p(i)-1}, c_{i^{p(i)} j^{p(i)}}\}.$$

When the minimum cost spanning tree is not unique, the rule is extended by averaging the allocations over all possible permutations $\pi \in \Pi_N$ (method = "exact"):

$$DK(N_0, C) = \frac{1}{n!} \sum_{\pi \in \Pi_N} DK^\pi(N_0, C),$$

where $DK^\pi(N_0, C)$ is the allocation obtained by applying Prim's algorithm to (N_0, C) and solving indifferences by selecting the first agent given by the permutation π .

Alternatively, method = "montecarlo" estimates the average allocation through random sampling using n_{sim} permutations. This is highly useful when the number of agents is large, as it significantly reduces computation time while providing a very accurate approximation:

$$DK(N_0, C) \approx \frac{1}{n_{\text{sim}}} \sum_{k=1}^{n_{\text{sim}}} DK^{\pi_k}(N_0, C),$$

where π_k represents each of the randomly sampled permutations. For networks with few agents, the exact method is preferred to avoid potential overestimations.

Value

A list containing:

- dk: the Dutta-Kar's allocation vector $DK(N_0, C)$ for the natural ordering.
- e_dk: the extended Dutta-Kar's allocation (average over permutations if ties exist; equal to dk otherwise).
- arcs: if the MCST is unique, a data frame of edges (i, j) in the tree and the stage at which they were added; if ties exist, a list of such data frames, one per distinct optimal tree.
- total: the total cost of the MCST, $m(N_0, C)$.
- percentage: the share of the total cost allocated to each agent, based on e_dk.
- ranking: a ranking of agents by cost in e_dk (from highest to lowest; ties marked with *).
- perms: if ties exist, a list with \$table (allocations for at most the first 6 permutations computed, regardless of n), \$summary (distinct allocations with their multiplicity, sorted in decreasing order) and \$nperms (total number of permutations computed); NULL if the MCST is unique.
- is_unique: logical; if TRUE, the MCST is unique.
- method: the method applied for computing the extended rule ("exact" or "montecarlo").

Supported formats for C

Note: In all cases, Inf is accepted for disconnected node pairs. The first row/column is assumed to be the source node (0).

The function accepts the following formats for the cost input:

Numeric vector The lower triangle of a symmetric cost matrix, excluding the diagonal. Length must be $n(n+1)/2$, where n is the number of agents (excluding the source).

Square matrix / Data frame A standard adjacency matrix where $C[i, j]$ represents the cost between node i and node j .

Edge list (matrix or data frame) An alternative to the full matrix. It must contain columns named "from" and "to", plus a third column for the weights/costs.

'igraph' object A weighted undirected graph. See the igraph package for details on creating these objects.

File path A string pointing to a .csv, .xlsx, or .xls file. The file can contain either a square matrix (without headers) or an edge list (with "from" and "to" headers).

References

- Bergantiños G, Vidal-Puga J (2021) A review of cooperative rules and their associated algorithms for minimum-cost spanning tree problems. *SERIEs*, 12:73-100
- Dutta B, Kar A (2004) Cost monotonicity, consistency and minimum cost spanning tree games. *Games Econom Behav* 48(2):223–248

See Also

mcstBird for other rule based on Prim’s algorithm.

mcstRules for an overview of the available rules and analysis tools in the package.

Examples

```
# Simple vector input
mcstDuttaKar(c(12, 15, 20, 4, 6, 8), draw = TRUE)

# Input with infinite costs (disconnected nodes)
C_inf <- c(5, 9, Inf, 15, 6, Inf, 7, Inf, Inf, Inf,
          Inf, 8, 7, Inf, Inf, 5, Inf, Inf, 8, 9, 11)
mcstDuttaKar(C_inf)

# Matrix input with ties
C_mat <- matrix(c(0, 12, 15, 12,
                  12, 0, 4, 6,
                  15, 4, 0, 8,
                  12, 6, 8, 0), byrow = TRUE, ncol = 4)
mcstDuttaKar(C_mat, draw = TRUE, which.plot = "main", titles = FALSE)
```

mcstFolk

Folk Rule for MCSTP

Description

This function computes folk cost allocation (Feltkamp et al., 1994) for a minimum cost spanning tree problem (N_0, C) . The folk rule, denoted as $F(N_0, C)$, also known as the ERO (Equal Remaining Obligation) rule, is defined through Kruskal’s algorithm using a function with specific properties, called the obligation function $o_i(S)$.

Usage

```
mcstFolk(C, draw = FALSE, titles = TRUE)
```

Arguments

C	cost matrix between nodes. Accepts multiple formats (see "Supported formats for C" below).
draw	logical; if TRUE, plots the network highlighting an optimal tree in red, indicating the stage at which each arc is added (in brackets) and the cost allocated to each agent (in parentheses below the node).
titles	logical; if TRUE (default), adds a main title specifying the allocation rule and a subtitle with the algorithm used and the total network cost.

Details

Following Kruskal's algorithm, let g^p be the network at stage $p \in \{1, \dots, n\}$, and $S_i^p := S(P(g^p), i)$ the connected component containing agent i . The folk rule is based on the obligation function:

$$o_i(S) = \frac{1}{|S|}.$$

At each stage p , let c_{ipj^p} be the cost of the arc added to the network. The folk allocation for each agent $i \in N$ is given by:

$$F_i(N_0, C) = \sum_{p=1}^n c_{ipj^p} \left(o_i(S_i^{p-1}) - o_i(S_i^p) \right).$$

Value

A list containing:

- **folk**: the folk allocation vector $F(N_0, C)$.
- **arcs**: a data frame of edges (i, j) in the final tree and the stage at which they were added.
- **total**: the total cost of the MCST, $m(N_0, C)$.
- **percentage**: the share of the total cost allocated to each agent.
- **ranking**: a ranking of agents by cost (from highest to lowest; ties marked with *).
- **is_unique**: logical; if TRUE, the MCST is unique.

Supported formats for C

Note: In all cases, Inf is accepted for disconnected node pairs. The first row/column is assumed to be the source node (0).

The function accepts the following formats for the cost input:

Numeric vector The lower triangle of a symmetric cost matrix, excluding the diagonal. Length must be $n(n+1)/2$, where n is the number of agents (excluding the source).

Square matrix / Data frame A standard adjacency matrix where $C[i, j]$ represents the cost between node i and node j .

Edge list (matrix or data frame) An alternative to the full matrix. It must contain columns named "from" and "to", plus a third column for the weights/costs.

'igraph' object A weighted undirected graph. See the `igraph` package for details on creating these objects.

File path A string pointing to a .csv, .xlsx, or .xls file. The file can contain either a square matrix (without headers) or an edge list (with "from" and "to" headers).

Note

The folk allocation is uniquely determined even when the minimum cost spanning tree is not unique. Since the rule depends on the evolution of the connected components rather than the specific arcs selected, any tie-breaking selection in Kruskal's algorithm yields the same cost distribution.

References

Bergantiños G, Vidal-Puga J (2021) A review of cooperative rules and their associated algorithms for minimum-cost spanning tree problems. *SERIEs*, 12:73-100

Feltkamp V, Tijs S, Muto S (1994) On the irreducible core and the equal remaining obligations rule of minimum cost spanning extension problems. Technical Report 106, CentER DP 1994, Tilburg University, The Netherlands

See Also

mcstOWShapley, mcstPWSShapley for other rules based on Kruskal's algorithm.

mcstBoruvka for an equivalent rule based on Boruvka's algorithm.

mcstCone with rule = "folk" for the equivalent implementation based on cone-wise decomposition.

mcstGameIrred, mcstGameOpt for cooperative games whose Shapley value coincides with the folk solution.

mcstRules for an overview of the available rules and analysis tools in the package.

Examples

```
# Simple vector input
mcstFolk(c(12, 15, 20, 4, 6, 8), draw = TRUE)

# Input with infinite costs (disconnected nodes)
C_inf <- c(5, 9, Inf, 15, 6, Inf, 7, Inf, Inf, Inf,
          Inf, 8, 7, Inf, Inf, 5, Inf, Inf, 8, 9, 11)
mcstFolk(C_inf)

# Matrix input
C_mat <- matrix(c(0, 12, 15, 12,
                  12, 0, 4, 6,
                  15, 4, 0, 8,
                  12, 6, 8, 0), byrow = TRUE, ncol = 4)
mcstFolk(C_mat, draw = TRUE, titles = FALSE)
```

mcstGameCC

Cost Allocations based on the Cycle-complete Game for MCSTP

Description

This function computes the cost allocation for the cycle-complete game (Trudeau, 2012) associated with a minimum cost spanning tree problem (N_0, C) , using a specified solution concept. The cycle-complete game, denoted as (N, v_C^c) (or simply (N, v^c)), is defined as the private game associated with the cycle-complete network (N_0, C^*) .

Usage

```
mcstGameCC(
  C,
  sol = c("shapley", "nucleolus"),
  draw = FALSE,
  which.plot = c("main", "details"),
  titles = TRUE,
  method = c("exact", "montecarlo"),
  nsim = 10000
)
```

Arguments

C	cost matrix between nodes. Accepts multiple formats (see "Supported formats for C" below).
sol	character; the solution concept to apply. Options are "shapley" (default), or "nucleolus" (only for method = "exact").
draw	logical; if TRUE, plots the network highlighting an optimal tree in red (constructed using Prim's algorithm), indicating the cost allocated to each agent in brackets below the node. For $n \leq 3$ and method = "exact", it also displays the optimal network for each subcoalition $S \subset N$.
which.plot	character string indicating which plots to display. If "details" (only for $n \leq 3$ and method = "exact"), displays the optimal network for each subcoalition $S \subset N$; if "main", the allocation for N is plotted. Default is c("main", "details").
titles	logical; if TRUE (default), adds a main title specifying the cooperative game and a subtitle with the solution concept, method (exact or Monte Carlo) and the total network cost.
method	character string specifying the calculation method. If "exact" (default), computes the exact allocation (Shapley value or nucleolus) and the full characteristic function; if "montecarlo", estimates an approximation of the Shapley value through random sampling using nsim permutations. It uses lazy evaluation to avoid computing all possible coalitions (recommended for large networks).
nsim	integer; the number of permutations to sample when using method = "montecarlo". Default is 10,000.

Details

The cycle-complete game (N, v_C^c) is a cooperative game with transferable utility (TU game) defined for each coalition $S \subseteq N$ as:

$$v_C^c(S) = m(S_0, C^*),$$

where $m(S_0, C^*)$ is the cost of the minimum spanning tree of the subproblem induced by the agents in S and the source, evaluated over the cycle-complete costs C^* . The cycle-complete cost is defined as the minimum between the direct cost c_{ij} and the smallest value of the maximum arc cost over all cycles containing i and j , i.e., $c_{ij}^* = \min(c_{ij}, \min_{f \in \mathcal{C}(i,j)} \max_{e \in f} c_e)$, where $\mathcal{C}(i, j)$ denotes the set of all cycles that include both i and j .

For computational efficiency, C^* is obtained via the irreducible matrices $\bar{C}$ of the subgraphs as:

$$c_{ij}^* = \max_{k \in N \setminus \{i,j\}} \bar{c}_{ij}^{N \setminus \{k\}} \text{ for } i, j \in N,$$

and

$$c_{0i}^* = \max_{k \in N \setminus \{i\}} \bar{c}_{0i}^{N \setminus \{k\}} \text{ for } i \in N,$$

where $\bar{C}^{N \setminus \{k\}}$ is the irreducible cost matrix of the network excluding agent k . If no agents can be removed, the irreducible cost of the full network is used.

Under method = "exact", the function builds the full characteristic function (all $2^n - 1$ values). Depending on the chosen solution concept, it computes the allocation via `shapleyValue` (called *cycle-complete solution*, which relies on the coalition-based formula) or `nucleolus`.

For larger networks, method = "montecarlo" provides an approximation of the Shapley value through random sampling using n_{sim} permutations. This approach implements lazy evaluation based on the permutation formula of the Shapley value: instead of computing the MST cost for all $2^n - 1$ possible subsets, it only evaluates the specific subcoalitions generated by the random permutations.

This is highly useful when the number of agents is large, as it significantly reduces memory usage and computation time while providing a very accurate approximation of the *cycle-complete solution*. For networks with few players, the exact method is preferred to avoid potential overestimations.

For more details on these formulas, see `mcstRules`.

Value

A list containing:

- `coalitions`: a character vector representing all possible coalitions $S \subseteq N$, ordered by cardinality. If method = "montecarlo", returns a string indicating that lazy evaluation was used.
- `C_cc`: the cycle-complete cost matrix C^* .
- `v_c`: a numeric vector containing the characteristic function values $v^c(S)$ for all coalitions. If method = "montecarlo", returns a string indicating that lazy evaluation was used.
- `contributions`: a data frame with the marginal contributions for the sampled permutations. If method = "exact", returns a string indicating that this table is not used.
- `allocation`: a numeric vector containing the computed cost allocation for each agent.
- `total`: the total cost of the MCST, $v^c(N)$.
- `nperms`: the number of permutations n_{sim} used in the calculation. If method = "exact", returns a string indicating that it was not computed.
- `percentage`: the share of the total cost allocated to each agent.
- `ranking`: ranking of agents by cost (from highest to lowest; ties marked with *).
- `method`: the method used for the calculation ("exact" or "montecarlo").
- `sol`: the solution concept used ("shapley" or "nucleolus").

Supported formats for C

Note: In all cases, Inf is accepted for disconnected node pairs. The first row/column is assumed to be the source node (0).

The function accepts the following formats for the cost input:

Numeric vector The lower triangle of a symmetric cost matrix, excluding the diagonal. Length must be $n(n+1)/2$, where n is the number of agents (excluding the source).

Square matrix / Data frame A standard adjacency matrix where $C[i, j]$ represents the cost between node i and node j .

Edge list (matrix or data frame) An alternative to the full matrix. It must contain columns named "from" and "to", plus a third column for the weights/costs.

'igraph' object A weighted undirected graph. See the `igraph` package for details on creating these objects.

File path A string pointing to a .csv, .xlsx, or .xls file. The file can contain either a square matrix (without headers) or an edge list (with "from" and "to" headers).

Note

The cycle-complete game is concave, which implies that its Shapley value (the *cycle-complete solution*) belongs to its core. Moreover, $\text{core}(N, v^c) \subseteq \text{core}(N, v^p)$.

References

Bergantiños G, Vidal-Puga J (2021) A review of cooperative rules and their associated algorithms for minimum-cost spanning tree problems. *SERIEs*, 12:73-100

Trudeau C (2012) A new stable and more responsible cost sharing solution for mcst problems. *Games Econom Behav* 75(1):402–412

See Also

`mcstGamePrivate` for the pessimistic game based on the original costs.

`mcstGameIrred` for the pessimistic game based on the irreducible cost matrix $\bar{C}$.

`mcstRules` for an overview of the available rules and analysis tools in the package.

Examples

```
# Simple vector input
mcstGameCC(c(12, 15, 20, 4, 6, 8), sol = "shapley", draw = TRUE)

# Input with infinite costs (disconnected nodes)
C_inf <- c(5, 9, Inf, 15, 6, Inf, 7, Inf, Inf, Inf,
          Inf, 8, 7, Inf, Inf, 5, Inf, Inf, 8, 9, 11)
mcstGameCC(C_inf, sol = "shapley")

# Matrix input
C_mat <- matrix(c(0, 12, 15, 12,
                 12, 0, 4, 6,
                 15, 4, 0, 8,
                 12, 6, 8, 0), byrow = TRUE, ncol = 4)
mcstGameCC(C_mat, draw = TRUE, which.plot = "main", titles = FALSE)
```

Description

This function computes the cost allocation for the irreducible game (Bird, 1976) associated with a minimum cost spanning tree problem (N_0, C) , using a specified solution concept. The irreducible game, denoted as (N, v_C^i) (or simply (N, v^i)), is defined as the private game associated with the irreducible cost matrix C .

Usage

```
mcstGameIrred(
  C,
  sol = c("shapley", "nucleolus"),
  draw = FALSE,
  which.plot = c("main", "details"),
  titles = TRUE,
  method = c("exact", "montecarlo"),
  nsim = 10000
)
```

Arguments

<code>C</code>	cost matrix between nodes. Accepts multiple formats (see "Supported formats for C" below).
<code>sol</code>	character; the solution concept to apply. Options are "shapley" (default), or "nucleolus" (only for method = "exact").
<code>draw</code>	logical; if TRUE, plots the network highlighting an optimal tree in red (constructed using Prim's algorithm), indicating the cost allocated to each agent in brackets below the node. For $n \leq 3$ and method = "exact", it also displays the optimal network for each subcoalition $S \subset N$.
<code>which.plot</code>	character string indicating which plots to display. If "details" (only for $n \leq 3$ and method = "exact"), displays the optimal network for each subcoalition $S \subset N$; if "main", the allocation for N is plotted. Default is c("main", "details").
<code>titles</code>	logical; if TRUE (default), adds a main title specifying the cooperative game and a subtitle with the solution concept, method (exact or Monte Carlo) and the total network cost.
<code>method</code>	character string specifying the calculation method. If "exact" (default), computes the exact allocation (Shapley value or nucleolus) and the full characteristic function; if "montecarlo", estimates an approximation of the Shapley value through random sampling using nsim permutations. It uses lazy evaluation to avoid computing all possible coalitions (recommended for large networks).
<code>nsim</code>	integer; the number of permutations to sample when using method = "montecarlo". Default is 10,000.

Details

The irreducible game (N, v_C^i) is a cooperative game with transferable utility (TU game) defined for each coalition $S \subseteq N$ as:

$$v_C^i(S) = m(S_0, \bar{C}),$$

where $m(S_0, \bar{C})$ is the cost of the minimum spanning tree of the subproblem induced by the agents in S and the source, evaluated over the irreducible costs $\bar{C}$. The irreducible cost between any two nodes corresponds to the maximum cost arc on the unique path connecting them in the optimal tree.

Under method = "exact", the function builds the full characteristic function (all $2^n - 1$ values). Depending on the chosen solution concept, it computes the allocation via `shapleyValue` (which relies on the coalition-based formula) or `nucleolus`.

For larger networks, method = "montecarlo" provides an approximation of the Shapley value through random sampling using n_{sim} permutations. This approach implements lazy evaluation based on the permutation formula of the Shapley value: instead of computing the MST cost for all $2^n - 1$ possible subsets, it only evaluates the specific subcoalitions generated by the random permutations.

This is highly useful when the number of agents is large, as it significantly reduces memory usage and computation time while providing a very accurate approximation of the exact allocation. For networks with few players, the exact method is preferred to avoid potential overestimations.

For more details on these formulas, see `mcstRules`.

Value

A list containing:

- `coalitions`: a character vector representing all possible coalitions $S \subseteq N$, ordered by cardinality. If method = "montecarlo", returns a string indicating that lazy evaluation was used.
- `C_irred`: the irreducible cost matrix $\bar{C}$ associated with the MCST of the problem.
- `v_i`: a numeric vector containing the characteristic function values $v^i(S)$ for all coalitions. If method = "montecarlo", returns a string indicating that lazy evaluation was used.
- `contributions`: a data frame with the marginal contributions for the sampled permutations. If method = "exact", returns a string indicating that this table is not used.
- `allocation`: a numeric vector containing the computed cost allocation for each agent.
- `total`: the total cost of the MCST, $v^i(N)$.
- `nperms`: the number of permutations n_{sim} used in the calculation. If method = "exact", returns a string indicating that it was not computed.
- `percentage`: the share of the total cost allocated to each agent.
- `ranking`: ranking of agents by cost (from highest to lowest; ties marked with *).
- `method`: the method used for the calculation ("exact" or "montecarlo").
- `sol`: the solution concept used ("shapley" or "nucleolus").

Supported formats for C

Note: In all cases, Inf is accepted for disconnected node pairs. The first row/column is assumed to be the source node (0).

The function accepts the following formats for the cost input:

Numeric vector The lower triangle of a symmetric cost matrix, excluding the diagonal. Length must be $n(n+1)/2$, where n is the number of agents (excluding the source).

Square matrix / Data frame A standard adjacency matrix where $C[i, j]$ represents the cost between node i and node j .

Edge list (matrix or data frame) An alternative to the full matrix. It must contain columns named "from" and "to", plus a third column for the weights/costs.

'igraph' object A weighted undirected graph. See the `igraph` package for details on creating these objects.

File path A string pointing to a .csv, .xlsx, or .xls file. The file can contain either a square matrix (without headers) or an edge list (with "from" and "to" headers).

Note

The irreducible game is a strong reference for stability. Its core (the irreducible core) is always non-empty and constitutes a subset of the core of v^p . Classical rules such as Bird and folk are known to belong to the core of v^i .

It has been proven that the Shapley value of the irreducible game coincides with both the `mcstFolk` of the original problem and `mcstBird` applied to the irreducible costs (Bergantiños and Vidal-Puga, 2007a).

Unlike the private game, calculating solution concepts for the irreducible game is computationally efficient. Computing both the Shapley value and the nucleolus of v^i can be done in $O(|N|^2)$ time.

References

- Bergantiños G, Vidal-Puga J (2007a) A fair rule in minimum cost spanning tree problems. *J Econom Theory* 137(1):326–352
- Bergantiños G, Vidal-Puga J (2021) A review of cooperative rules and their associated algorithms for minimum-cost spanning tree problems. *SERIEs*, 12:73-100
- Bird C (1976) On cost allocation for a spanning tree: a game theoretic approach. *Networks* 6(4):335–350

See Also

`mcstBird` of $(N_0, \bar{C})$, `mcstFolk` of (N_0, C) for classical rules belonging to the core of v^i that coincide with $Sh(N, v^i)$.

`mcstGamePrivate`, `mcstGamePublic` for the pessimistic and optimistic games based on the original costs.

`mcstGameCC` for the pessimistic game based on the cycle-complete cost matrix C^* .

`mcstGameOpt` for an equivalent game under the Shapley value.

`mcstRules` for an overview of the available rules and analysis tools in the package.

Examples

```
# Simple vector input
mcstGameIrred(c(12, 15, 20, 4, 6, 8), sol = "shapley", draw = TRUE)

# Input with infinite costs (disconnected nodes)
C_inf <- c(5, 9, Inf, 15, 6, Inf, 7, Inf, Inf, Inf,
          Inf, 8, 7, Inf, Inf, 5, Inf, Inf, 8, 9, 11)
mcstGameIrred(C_inf, sol = "shapley")
```

```
# Matrix input
C_mat <- matrix(c(0, 12, 15, 12,
                  12, 0, 4, 6,
                  15, 4, 0, 8,
                  12, 6, 8, 0), byrow = TRUE, ncol = 4)
mcstGameIrred(C_mat, draw = TRUE, which.plot = "main", titles = FALSE)
```

mcstGameOpt

Cost Allocations based on the Optimistic Game for MCSTP

Description

This function computes the cost allocation for the optimistic game (Bergantiños and Vidal-Puga, 2007b) associated with a minimum cost spanning tree problem (N_0, C) , using a specified solution concept. The optimistic game, denoted as (N, v_C^o) (or simply (N, v^o)), evaluates the cost of each coalition $S \subseteq N$ assuming that agents in $N \setminus S$ are already connected. Thus, agents in S can connect to the source through agents in $N \setminus S$ for free.

Usage

```
mcstGameOpt(
  C,
  sol = c("shapley", "nucleolus"),
  draw = FALSE,
  which.plot = c("main", "details"),
  titles = TRUE,
  method = c("exact", "montecarlo"),
  nsim = 10000
)
```

Arguments

C	cost matrix between nodes. Accepts multiple formats (see "Supported formats for C" below).
sol	character; the solution concept to apply. Options are "shapley" (default), or "nucleolus" (only for method = "exact").
draw	logical; if TRUE, plots the network highlighting an optimal tree in red (constructed using Prim's algorithm), indicating the cost allocated to each agent in brackets below the node. For $n \leq 3$ and method = "exact", it also displays the optimal network for each subcoalition $S \subset N$.
which.plot	character string indicating which plots to display. If "details" (only for $n \leq 3$ and method = "exact"), displays the optimal network for each subcoalition $S \subset N$; if "main", the allocation for N is plotted. Default is c("main", "details").
titles	logical; if TRUE (default), adds a main title specifying the cooperative game and a subtitle with the solution concept, method (exact or Monte Carlo) and the total network cost.

method	character string specifying the calculation method. If "exact" (default), computes the exact allocation (Shapley value or nucleolus) and the full characteristic function; if "montecarlo", estimates an approximation of the Shapley value through random sampling using <code>nsim</code> permutations. It uses lazy evaluation to avoid computing all possible coalitions (recommended for large networks).
nsim	integer; the number of permutations to sample when using method = "montecarlo". Default is 10,000.

Details

Let $T := N \setminus S$. The optimistic game (N, v_C^o) is a cooperative game with transferable utility (TU game) defined for each coalition $S \subseteq N$ as:

$$v_C^o(S) = m(S_0, C^T),$$

where $m(S_0, C^T)$ is the cost of the minimum spanning tree of the subproblem induced by the agents in S and the source, using the modified cost matrix C^T . In this matrix, the cost between agents in S remains the same, i.e., $c_{ij}^T = c_{ij}$ for all $i, j \in S$, but the cost to connect any agent $i \in S$ to the source is updated as:

$$c_{0i}^T = \min_{j \in T \cup \{0\}} c_{ji}.$$

Under method = "exact", the function builds the full characteristic function (all $2^n - 1$ values). Depending on the chosen solution concept, it computes the allocation via `shapleyValue` (which relies on the coalition-based formula) or `nucleolus`.

For larger networks, method = "montecarlo" provides an approximation of the Shapley value through random sampling using n_{sim} permutations. This approach implements lazy evaluation based on the permutation formula of the Shapley value: instead of computing the MST cost for all $2^n - 1$ possible subsets, it only evaluates the specific subcoalitions generated by the random permutations.

This is highly useful when the number of agents is large, as it significantly reduces memory usage and computation time while providing a very accurate approximation of the exact allocation. For networks with few players, the exact method is preferred to avoid potential overestimations.

For more details on these formulas, see `mcstRules`.

Value

A list containing:

- `coalitions`: a character vector representing all possible coalitions $S \subseteq N$, ordered by cardinality. If method = "montecarlo", returns a string indicating that lazy evaluation was used.
- `C_opt`: a list of the optimistic cost matrices C^T for each coalition $S \subseteq N$. If method = "montecarlo", returns a string indicating that lazy evaluation was used.
- `v_o`: a numeric vector containing the characteristic function values $v^o(S)$ for all coalitions. If method = "montecarlo", returns a string indicating that lazy evaluation was used.
- `contributions`: a data frame with the marginal contributions for the sampled permutations. If method = "exact", returns a string indicating that this table is not used.
- `allocation`: a numeric vector containing the computed cost allocation for each agent.
- `total`: the total cost of the MCST, $v^o(N)$.

- `nperms`: the number of permutations `nsim` used in the calculation. If `method = "exact"`, returns a string indicating that it was not computed.
- `percentage`: the share of the total cost allocated to each agent.
- `ranking`: ranking of agents by cost (from highest to lowest; ties marked with *).
- `method`: the method used for the calculation ("exact" or "montecarlo").
- `sol`: the solution concept used ("shapley" or "nucleolus").

Supported formats for C

Note: In all cases, Inf is accepted for disconnected node pairs. The first row/column is assumed to be the source node (0).

The function accepts the following formats for the cost input:

Numeric vector The lower triangle of a symmetric cost matrix, excluding the diagonal. Length must be $n(n+1)/2$, where n is the number of agents (excluding the source).

Square matrix / Data frame A standard adjacency matrix where $C[i, j]$ represents the cost between node i and node j .

Edge list (matrix or data frame) An alternative to the full matrix. It must contain columns named "from" and "to", plus a third column for the weights/costs.

'igraph' object A weighted undirected graph. See the `igraph` package for details on creating these objects.

File path A string pointing to a .csv, .xlsx, or .xls file. The file can contain either a square matrix (without headers) or an edge list (with "from" and "to" headers).

Note

The optimistic game associated with any MCST problem coincides with the optimistic game associated with its irreducible form, i.e.,

$$v_C^o = v_{\bar{C}}^o.$$

If $(N_0, \bar{C})$ is irreducible, the private and optimistic games, v^p and v^o , are dual: $v^p(S) + v^o(N \setminus S) = m(N_0, C)$ for all $S \subset N$.

Furthermore, $Sh(N, v_C^o) = Sh(N, v_{\bar{C}}^o)$, so applying the Shapley value to v^o provides an alternative way of obtaining the `mcstFolk` (Bergantiños and Vidal-Puga, 2007b).

References

- Bergantiños G, Vidal-Puga J (2007b) The optimistic TU game in minimum cost spanning tree problems. *Int J Game Theory* 36(2):223–239
- Bergantiños G, Vidal-Puga J (2021) A review of cooperative rules and their associated algorithms for minimum-cost spanning tree problems. *SERIEs*, 12:73-100

See Also

- `mcstGamePrivate` for a game based on a pessimistic approach.
- `mcstGameIrred` for an equivalent game under the Shapley value.
- `mcstRules` for an overview of the available rules and analysis tools in the package.

Examples

```
# Simple vector input
mcstGameOpt(c(12, 15, 20, 4, 6, 8), sol = "shapley", draw = TRUE)

# Input with infinite costs (disconnected nodes)
C_inf <- c(5, 9, Inf, 15, 6, Inf, 7, Inf, Inf, Inf,
          Inf, 8, 7, Inf, Inf, 5, Inf, Inf, 8, 9, 11)
mcstGameOpt(C_inf, sol = "shapley")

# Matrix input
C_mat <- matrix(c(0, 12, 15, 12,
                  12, 0, 4, 6,
                  15, 4, 0, 8,
                  12, 6, 8, 0), byrow = TRUE, ncol = 4)
mcstGameOpt(C_mat, draw = TRUE, which.plot = "main", titles = FALSE)
```

mcstGamePrivate

Cost Allocations based on the Private Game for MCSTP

Description

This function computes the cost allocation for the private or pessimistic game (Bird, 1976) associated with a minimum cost spanning tree problem (N_0, C) , using a specified solution concept. The private game, denoted as (N, v_C^p) (or simply (N, v^p)), evaluates the cost of each coalition $S \subseteq N$ under the assumption that agents outside the coalition are not available to provide their nodes as intermediaries.

Usage

```
mcstGamePrivate(
  C,
  sol = c("shapley", "nucleolus"),
  draw = FALSE,
  which.plot = c("main", "details"),
  titles = TRUE,
  method = c("exact", "montecarlo"),
  nsim = 10000
)
```

Arguments

C	cost matrix between nodes. Accepts multiple formats (see "Supported formats for C" below).
sol	character; the solution concept to apply. Options are "shapley" (default), or "nucleolus" (only for method = "exact").
draw	logical; if TRUE, plots the network highlighting an optimal tree in red (constructed using Prim's algorithm), indicating the cost allocated to each agent in brackets below the node. For $n \leq 3$ and method = "exact", it also displays the optimal network for each subcoalition $S \subset N$.

which.plot	character string indicating which plots to display. If "details" (only for $n \leq 3$ and method = "exact"), displays the optimal network for each subcoalition $S \subset N$; if "main", the allocation for N is plotted. Default is c("main", "details").
titles	logical; if TRUE (default), adds a main title specifying the cooperative game and a subtitle with the solution concept, method (exact or Monte Carlo) and the total network cost.
method	character string specifying the calculation method. If "exact" (default), computes the exact allocation (Shapley value or nucleolus) and the full characteristic function; if "montecarlo", estimates an approximation of the Shapley value through random sampling using nsim permutations. It uses lazy evaluation to avoid computing all possible coalitions (recommended for large networks).
nsim	integer; the number of permutations to sample when using method = "montecarlo". Default is 10,000.

Details

The private game (N, v_C^p) is a cooperative game with transferable utility (TU game) defined for each coalition $S \subseteq N$ as:

$$v_C^p(S) = m(S_0, C),$$

where $m(S_0, C)$ is the cost of the minimum spanning tree of the subproblem induced by the agents in S and the source. This approach is termed "private" or "pessimistic" because the nodes in $N \setminus S$ belong to these agents, and their participation is needed in order to use their nodes.

When the Shapley value (sol = "shapley") is applied to this game, the resulting allocation is known as Kar's rule (Kar, 2002). For a function that particularizes this specific case, see mcstKar.

Under method = "exact", the function builds the full characteristic function (all $2^n - 1$ values). Depending on the chosen solution concept, it computes the allocation via shapleyValue (which relies on the coalition-based formula) or nucleolus.

For larger networks, method = "montecarlo" provides an approximation of the Shapley value through random sampling using n_{sim} permutations. This approach implements lazy evaluation based on the permutation formula of the Shapley value: instead of computing the MST cost for all $2^n - 1$ possible subsets, it only evaluates the specific subcoalitions generated by the random permutations.

This is highly useful when the number of agents is large, as it significantly reduces memory usage and computation time while providing a very accurate approximation of Kar's rule. For networks with few players, the exact method is preferred to avoid potential overestimations.

For more details on these formulas, see mcstRules.

Value

A list containing:

- **coalitions**: a character vector representing all possible coalitions $S \subseteq N$, ordered by cardinality. If method = "montecarlo", returns a string indicating that lazy evaluation was used.
- **v_p**: a numeric vector containing the characteristic function values $v^p(S)$ for all coalitions. If method = "montecarlo", returns a string indicating that lazy evaluation was used.
- **contributions**: a data frame with the marginal contributions for the sampled permutations. If method = "exact", returns a string indicating that this table is not used.

- **allocation**: a numeric vector containing the computed cost allocation for each agent.
- **total**: the total cost of the MCST, $v^p(N)$.
- **nperms**: the number of permutations $nsim$ used in the calculation. If `method = "exact"`, returns a string indicating that it was not computed.
- **percentage**: the share of the total cost allocated to each agent.
- **ranking**: ranking of agents by cost (from highest to lowest; ties marked with *).
- **method**: the method used for the calculation ("exact" or "montecarlo").
- **sol**: the solution concept used ("shapley" or "nucleolus").

Supported formats for C

Note: In all cases, Inf is accepted for disconnected node pairs. The first row/column is assumed to be the source node (0).

The function accepts the following formats for the cost input:

Numeric vector The lower triangle of a symmetric cost matrix, excluding the diagonal. Length must be $n(n+1)/2$, where n is the number of agents (excluding the source).

Square matrix / Data frame A standard adjacency matrix where $C[i, j]$ represents the cost between node i and node j .

Edge list (matrix or data frame) An alternative to the full matrix. It must contain columns named "from" and "to", plus a third column for the weights/costs.

'igraph' object A weighted undirected graph. See the `igraph` package for details on creating these objects.

File path A string pointing to a .csv, .xlsx, or .xls file. The file can contain either a square matrix (without headers) or an edge list (with "from" and "to" headers).

Note

The private game is a strong reference for stability. Classical rules such as Bird, Dutta-Kar, or the folk rule are known to belong to the core of v^p for any MCST problem. As a result, these allocations are coalitionally stable under the pessimistic cost perspective.

Although the Shapley value and the nucleolus are well-established solution concepts, it has been proven that computing both for this game is NP-hard.

References

- Bergantiños G, Vidal-Puga J (2021) A review of cooperative rules and their associated algorithms for minimum-cost spanning tree problems. *SERIEs*, 12:73-100
- Bird C (1976) On cost allocation for a spanning tree: a game theoretic approach. *Networks* 6(4):335–350
- Kar A (2002) Axiomatization of the Shapley value on minimum cost spanning tree games. *Games Econom Behav* 38(2):265–277

See Also

`mcstBird`, `mcstDuttaKar`, `mcstFolk` for classical rules belonging to the core of v^p .
`mcstKar` for the specific cost allocation rule based on the Shapley value of this game.
`mcstGameIrred`, `mcstGameCC` for the pessimistic games based on the irreducible cost matrix $\bar{C}$ and the cycle-complete cost matrix C^* , respectively.

mcstGameOpt, mcstGamePublic for optimistic alternatives to the private approach.
 mcstRules for an overview of the available rules and analysis tools in the package.

Examples

```
# Simple vector input
mcstGamePrivate(c(12, 15, 20, 4, 6, 8), sol = "shapley", draw = TRUE)

# Input with infinite costs (disconnected nodes)
C_inf <- c(5, 9, Inf, 15, 6, Inf, 7, Inf, Inf, Inf,
          Inf, 8, 7, Inf, Inf, 5, Inf, Inf, 8, 9, 11)
mcstGamePrivate(C_inf, sol = "nucleolus")

# Matrix input
C_mat <- matrix(c(0, 12, 15, 12,
                 12, 0, 4, 6,
                 15, 4, 0, 8,
                 12, 6, 8, 0), byrow = TRUE, ncol = 4)
mcstGamePrivate(C_mat, draw = TRUE, which.plot = "main", titles = FALSE)
```

 mcstGamePublic

Cost Allocations based on the Public Game for MCSTP

Description

This function computes the cost allocation for the public game (Bogomolnaia and Moulin, 2010) associated with a minimum cost spanning tree problem (N_0, C) , using a specified solution concept. The public game, denoted as (N, v_C^u) (or simply (N, v^u)), evaluates the cost of each coalition $S \subseteq N$ assuming that there are no property rights on nodes. Therefore, agents in S can use the nodes of agents in $N \setminus S$ to connect to the source, paying the corresponding arc costs.

Usage

```
mcstGamePublic(
  C,
  sol = c("shapley", "nucleolus"),
  draw = FALSE,
  which.plot = c("main", "details"),
  titles = TRUE,
  method = c("exact", "montecarlo"),
  nsim = 10000
)
```

Arguments

C	cost matrix between nodes. Accepts multiple formats (see "Supported formats for C" below).
sol	character; the solution concept to apply. Options are "shapley" (default), or "nucleolus" (only for method = "exact").

draw	logical; if TRUE, plots the network highlighting an optimal tree in red (constructed using Prim's algorithm), indicating the cost allocated to each agent in brackets below the node. For $n \leq 3$ and method = "exact", it also displays the optimal network for each subcoalition $S \subset N$.
which.plot	character string indicating which plots to display. If "details" (only for $n \leq 3$ and method = "exact"), displays the optimal network for each subcoalition $S \subset N$; if "main", the allocation for N is plotted. Default is c("main", "details").
titles	logical; if TRUE (default), adds a main title specifying the cooperative game and a subtitle with the solution concept, method (exact or Monte Carlo) and the total network cost.
method	character string specifying the calculation method. If "exact" (default), computes the exact allocation (Shapley value or nucleolus) and the full characteristic function; if "montecarlo", estimates an approximation of the Shapley value through random sampling using n_{sim} permutations. It uses lazy evaluation to avoid computing all possible coalitions (recommended for large networks).
nsim	integer; the number of permutations to sample when using method = "montecarlo". Default is 10,000.

Details

The public game (N, v_C^u) is a cooperative game with transferable utility (TU game) where the cost for each coalition $S \subseteq N$ is defined as the minimum private game cost among all its supersets $T \supseteq S$, i.e.:

$$v_C^u(S) = \min_{S \subseteq T} v_C^p(T),$$

where $v_C^p(\cdot)$ is the characteristic function of the private game. This means computing $v^u(S)$ requires calculating the private game costs for all possible supersets of S .

Under method = "exact", the function builds the full characteristic function (all $2^n - 1$ values). Depending on the chosen solution concept, it computes the allocation via `shapleyValue` (which relies on the coalition-based formula) or `nucleolus`.

For larger networks, method = "montecarlo" provides an approximation of the Shapley value through random sampling using n_{sim} permutations. This approach implements lazy evaluation based on the permutation formula of the Shapley value: instead of computing the MST cost for all $2^n - 1$ possible subsets, it only evaluates the specific subcoalitions generated by the random permutations.

This is highly useful when the number of agents is large, as it significantly reduces memory usage and computation time while providing a very accurate approximation of the exact allocation. For networks with few players, the exact method is preferred to avoid potential overestimations.

For more details on these formulas, see `mcstRules`.

Value

A list containing:

- `coalitions`: a character vector representing all possible coalitions $S \subseteq N$, ordered by cardinality. If method = "montecarlo", returns a string indicating that lazy evaluation was used.
- `v_u`: a numeric vector containing the characteristic function values $v^u(S)$ for all coalitions. If method = "montecarlo", returns a string indicating that lazy evaluation was used.

- `v_p`: a numeric vector containing the characteristic function values $v^p(S)$ for all coalitions. If `method = "montecarlo"`, returns a string indicating that lazy evaluation was used.
- `best_T`: a data frame mapping each coalition S to the optimal superset T that minimizes its cost. If `method = "montecarlo"`, returns a string indicating that lazy evaluation was used.
- `contributions`: a data frame with the marginal contributions for the sampled permutations. If `method = "exact"`, returns a string indicating that this table is not used.
- `allocation`: a numeric vector containing the computed cost allocation for each agent.
- `total`: the total cost of the MCST, $v^u(N)$.
- `nperms`: the number of permutations `nsim` used in the calculation. If `method = "exact"`, returns a string indicating that it was not computed.
- `percentage`: the share of the total cost allocated to each agent.
- `ranking`: ranking of agents by cost (from highest to lowest; ties marked with *).
- `method`: the method used for the calculation ("exact" or "montecarlo").
- `sol`: the solution concept used ("shapley" or "nucleolus").

Supported formats for C

Note: In all cases, Inf is accepted for disconnected node pairs. The first row/column is assumed to be the source node (0).

The function accepts the following formats for the cost input:

Numeric vector The lower triangle of a symmetric cost matrix, excluding the diagonal. Length must be $n(n+1)/2$, where n is the number of agents (excluding the source).

Square matrix / Data frame A standard adjacency matrix where `C[i, j]` represents the cost between node i and node j .

Edge list (matrix or data frame) An alternative to the full matrix. It must contain columns named "from" and "to", plus a third column for the weights/costs.

'igraph' object A weighted undirected graph. See the `igraph` package for details on creating these objects.

File path A string pointing to a .csv, .xlsx, or .xls file. The file can contain either a square matrix (without headers) or an edge list (with "from" and "to" headers).

Note

The core of v^u coincides with the core of v^p restricted to nonnegative cost shares. Consequently, classical rules such as Bird, Dutta-Kar, or the folk rule belong to the core of the public game for any MCST problem.

However, unlike in the private game, the Shapley value of the public game does not necessarily belong to the core of v^u (Trudeau and Vidal-Puga, 2019).

References

- Bergantiños G, Vidal-Puga J (2021) A review of cooperative rules and their associated algorithms for minimum-cost spanning tree problems. *SERIEs*, 12:73-100
- Bogomolnaia A, Moulin H (2010) Sharing a minimal cost spanning tree: beyond the folk solution. *Games and Econom Behav* 69(2):238–248
- Trudeau C, Vidal-Puga J (2019) The Shapley value in minimum cost spanning tree problems. In: Algaba E, Fragnelli V, Sánchez-Soriano J (eds) *Chapters in Game Theory: The Shapley value*, chapter 24. CRC Press, Taylor & Francis Group

See Also

mcstBird, mcstDuttaKar, mcstFolk for classical rules belonging to the core of v^u .
 mcstGamePrivate for the pessimistic alternative to the public game.
 mcstGameIrred for a game that can be defined from either the private or the public approach.
 mcstRules for an overview of the available rules and analysis tools in the package.

Examples

```
# Simple vector input
mcstGamePublic(c(12, 15, 20, 4, 6, 8), sol = "shapley", draw = TRUE)

# Input with infinite costs (disconnected nodes)
C_inf <- c(5, 9, Inf, 15, 6, Inf, 7, Inf, Inf, Inf,
          Inf, 8, 7, Inf, Inf, 5, Inf, Inf, 8, 9, 11)
mcstGamePublic(C_inf, sol = "shapley")

# Matrix input
C_mat <- matrix(c(0, 12, 15, 12,
                  12, 0, 4, 6,
                  15, 4, 0, 8,
                  12, 6, 8, 0), byrow = TRUE, ncol = 4)
mcstGamePublic(C_mat, draw = TRUE, which.plot = "main", titles = FALSE)
```

mcstKar

Kar's Rule for MCSTP

Description

This function computes Kar's rule for a minimum cost spanning tree problem (N_0, C) . It is defined as the Shapley value of the associated private game (Kar, 2002).

Usage

```
mcstKar(
  C,
  draw = FALSE,
  which.plot = c("main", "details"),
  titles = TRUE,
  method = c("exact", "montecarlo"),
  nsim = 10000
)
```

Arguments

C	cost matrix between nodes. Accepts multiple formats (see the "Supported formats for C" section in mcstGamePrivate).
draw	logical; if TRUE, plots the network highlighting an optimal tree in red (constructed using Prim's algorithm), indicating the cost allocated to each agent in brackets below the node. For $n \leq 3$ and method = "exact", it also displays the optimal network for each subcoalition $S \subset N$.

<code>which.plot</code>	character string indicating which plots to display. If "details" (only for $n \leq 3$ and <code>method = "exact"</code>), displays the optimal network for each subcoalition $S \subset N$; if "main", the allocation for N is plotted. Default is <code>c("main", "details")</code> .
<code>titles</code>	logical; if TRUE (default), adds a main title specifying the cooperative game and a subtitle with the solution concept, method (exact or Monte Carlo) and the total network cost.
<code>method</code>	character string specifying the calculation method. If "exact" (default), computes the exact Shapley value allocation and the full characteristic function; if "montecarlo", estimates an approximation of the Shapley value through random sampling using <code>nsim</code> permutations. It uses lazy evaluation to avoid computing all possible coalitions (recommended for large networks).
<code>nsim</code>	integer; the number of permutations to sample when using <code>method = "montecarlo"</code> . Default is 10,000.

Value

A list containing the computed allocation and calculation details. For a comprehensive breakdown of the elements returned in this list, see the "Value" section in `mcstGamePrivate`.

Note

For an in-depth explanation of the private game's properties, its characteristic function formulation, and discussions on computational complexity, please refer to the documentation for `mcstGamePrivate`.

References

- Bergantiños G, Vidal-Puga J (2021) A review of cooperative rules and their associated algorithms for minimum-cost spanning tree problems. *SERIEs*, 12:73-100
- Kar A (2002) Axiomatization of the Shapley value on minimum cost spanning tree games. *Games Econom Behav* 38(2):265–277

See Also

`mcstGamePrivate` for the general pessimistic game.

`mcstRules` for an overview of the available rules and analysis tools in the package.

Examples

```
# Simple vector input
mcstKar(c(12, 15, 20, 4, 6, 8), draw = TRUE)

# Input with infinite costs (disconnected nodes)
C_inf <- c(5, 9, Inf, 15, 6, Inf, 7, Inf, Inf, Inf,
          Inf, 8, 7, Inf, Inf, 5, Inf, Inf, 8, 9, 11)
mcstKar(C_inf)

# Matrix input
C_mat <- matrix(c(0, 12, 15, 12,
                  12, 0, 4, 6,
                  15, 4, 0, 8,
                  12, 6, 8, 0), byrow = TRUE, ncol = 4)
mcstKar(C_mat, draw = TRUE, which.plot = "main", titles = FALSE)
```

mcstOWShapley

*Optimistic Weighted Shapley's Rule for MCSTP***Description**

This function computes the optimistic weighted Shapley cost allocation (Bergantiños and Lorenzo-Freire, 2008a,b) for a minimum cost spanning tree problem (N_0, C) . The optimistic weighted Shapley rule, denoted as $f^{\theta^{ow}}(N_0, C)$, is defined through Kruskal's algorithm using a function with specific properties, called the obligation function $o_i(S)$.

Usage

```
mcstOWShapley(C, weights, draw = FALSE, titles = TRUE)
```

Arguments

C	cost matrix between nodes. Accepts multiple formats (see "Supported formats for C" below).
weights	a numeric vector of strictly positive weights for each agent in N . The length of the vector must match the number of agents.
draw	logical; if TRUE, plots the network highlighting an optimal tree in red, indicating the stage at which each arc is added (in brackets) and the cost allocated to each agent (in parentheses below the node).
titles	logical; if TRUE (default), adds a main title specifying the allocation rule and a subtitle with the algorithm used and the total network cost.

Details

Following Kruskal's algorithm, let g^p be the network at stage $p \in \{1, \dots, n\}$, and $S_i^p := S(P(g^p), i)$ the connected component containing agent i . The optimistic weighted Shapley rule is based on the obligation function:

$$o_i(S) = \frac{w_i}{\sum_{j \in S} w_j},$$

where $w_i > 0$ is the weight associated with agent i .

At each stage p , let c_{ipjp} be the cost of the arc added to the network. The optimistic weighted Shapley allocation for each agent $i \in N$ is given by:

$$f_i^{\theta^{ow}}(N_0, C) = \sum_{p=1}^n c_{ipjp} \left(o_i(S_i^{p-1}) - o_i(S_i^p) \right).$$

Value

A list containing:

- **ows**: the optimistic weighted Shapley allocation vector $f^{\theta^{ow}}(N_0, C)$.
- **weights**: the weight vector used for the allocation.
- **arcs**: a data frame of edges (i, j) in the final tree and the stage at which they were added.

- **total**: the total cost of the MCST, $m(N_0, C)$.
- **percentage**: the share of the total cost allocated to each agent.
- **ranking**: a ranking of agents by cost (from highest to lowest; ties marked with *).
- **is_unique**: logical; if TRUE, the MCST is unique.

Supported formats for C

Note: In all cases, Inf is accepted for disconnected node pairs. The first row/column is assumed to be the source node (0).

The function accepts the following formats for the cost input:

Numeric vector The lower triangle of a symmetric cost matrix, excluding the diagonal. Length must be $n(n+1)/2$, where n is the number of agents (excluding the source).

Square matrix / Data frame A standard adjacency matrix where $C[i, j]$ represents the cost between node i and node j .

Edge list (matrix or data frame) An alternative to the full matrix. It must contain columns named "from" and "to", plus a third column for the weights/costs.

'igraph' object A weighted undirected graph. See the *igraph* package for details on creating these objects.

File path A string pointing to a .csv, .xlsx, or .xls file. The file can contain either a square matrix (without headers) or an edge list (with "from" and "to" headers).

Note

The optimistic weighted Shapley allocation is uniquely determined even when the minimum cost spanning tree is not unique. Since the rule depends on the evolution of the connected components rather than the specific arcs selected, any tie-breaking selection in Kruskal's algorithm yields the same cost distribution.

Furthermore, when all agents share the same weight, the optimistic weighted Shapley allocation is mathematically equivalent to the *mcstFolk*.

References

- Bergantiños G, Lorenzo-Freire S (2008a) A characterization of optimistic weighted Shapley rules in minimum cost spanning tree problems. *Econ Theor* 35(3):523–538
- Bergantiños G, Lorenzo-Freire S (2008b) "Optimistic" weighted Shapley rules in minimum cost spanning tree problems. *European J Operatl Res* 185(1):289–298
- Bergantiños G, Vidal-Puga J (2021) A review of cooperative rules and their associated algorithms for minimum-cost spanning tree problems. *SERIEs*, 12:73-100

See Also

mcstFolk, *mcstPWShapley* for other rules based on Kruskal's algorithm.

mcstCone with `rule = "owshapley"` for the equivalent implementation based on cone-wise decomposition.

mcstRules for an overview of the available rules and analysis tools in the package.

Examples

```
# Simple vector input with custom weights
mcstOWShapley(c(12, 15, 20, 4, 6, 8), weights = c(2, 1, 1))

# Matrix input with equal weights (equivalent to folk rule)
C_mat <- matrix(c(0, 12, 15, 12,
                  12, 0, 4, 6,
                  15, 4, 0, 8,
                  12, 6, 8, 0), byrow = TRUE, ncol = 4)
mcstOWShapley(C_mat, weights = rep(1,3), draw = TRUE, titles = FALSE)
```

mcstPWShapley

Pessimistic Weighted Shapley's Rule for MCSTP

Description

This function computes the pessimistic weighted Shapley cost allocation (Lorenzo and Lorenzo-Freire, 2009) for a minimum cost spanning tree problem (N_0, C) . The pessimistic weighted Shapley rule, denoted as $f^{\ell^{pw}}(N_0, C)$, is defined through Kruskal's algorithm using a function with specific properties, called the obligation function $o_i(S)$.

Usage

```
mcstPWShapley(C, weights, draw = FALSE, titles = TRUE)
```

Arguments

C	cost matrix between nodes. Accepts multiple formats (see "Supported formats for C" below).
weights	a numeric vector of strictly positive weights for each agent in N . The length of the vector must match the number of agents.
draw	logical; if TRUE, plots the network highlighting an optimal tree in red, indicating the stage at which each arc is added (in brackets) and the cost allocated to each agent (in parentheses below the node).
titles	logical; if TRUE (default), adds a main title specifying the allocation rule and a subtitle with the algorithm used and the total network cost.

Details

Following Kruskal's algorithm, let g^p be the network at stage $p \in \{1, \dots, n\}$, and $S_i^p := S(P(g^p), i)$ the connected component containing agent i . The pessimistic weighted Shapley rule is based on the obligation function:

$$o_i(S) = \sum_{\pi \in \Pi(S \setminus \{i\})} \prod_{j=1}^{|S|-1} \frac{w_{\pi^{-1}(j)}}{\sum_{k=1}^j w_{\pi^{-1}(k)} + w_i},$$

where $\Pi(\cdot)$ denotes the set of all permutations of a given set, and $w_i > 0$ is the weight associated with agent i .

At each stage p , let c_{ij^p} be the cost of the arc added to the network. The pessimistic weighted Shapley allocation for each agent $i \in N$ is given by:

$$f_i^{p^w}(N_0, C) = \sum_{p=1}^n c_{ij^p} \left(o_i(S_i^{p-1}) - o_i(S_i^p) \right).$$

Value

A list containing:

- pws: the pessimistic weighted Shapley allocation vector $f^{p^w}(N_0, C)$.
- weights: the weight vector used for the allocation.
- arcs: a data frame of edges (i, j) in the final tree and the stage at which they were added.
- total: the total cost of the MCST, $m(N_0, C)$.
- percentage: the share of the total cost allocated to each agent.
- ranking: a ranking of agents by cost (from highest to lowest; ties marked with *).
- is_unique: logical; if TRUE, the MCST is unique.

Supported formats for C

Note: In all cases, Inf is accepted for disconnected node pairs. The first row/column is assumed to be the source node (0).

The function accepts the following formats for the cost input:

Numeric vector The lower triangle of a symmetric cost matrix, excluding the diagonal. Length must be $n(n+1)/2$, where n is the number of agents (excluding the source).

Square matrix / Data frame A standard adjacency matrix where $C[i, j]$ represents the cost between node i and node j .

Edge list (matrix or data frame) An alternative to the full matrix. It must contain columns named "from" and "to", plus a third column for the weights/costs.

'igraph' object A weighted undirected graph. See the `igraph` package for details on creating these objects.

File path A string pointing to a .csv, .xlsx, or .xls file. The file can contain either a square matrix (without headers) or an edge list (with "from" and "to" headers).

Note

The pessimistic weighted Shapley allocation is uniquely determined even when the minimum cost spanning tree is not unique. Since the rule depends on the evolution of the connected components rather than the specific arcs selected, any tie-breaking selection in Kruskal's algorithm yields the same cost distribution.

Furthermore, when all agents share the same weight, the pessimistic weighted Shapley allocation is mathematically equivalent to the `mcstFolk`.

References

- Bergantiños G, Vidal-Puga J (2021) A review of cooperative rules and their associated algorithms for minimum-cost spanning tree problems. *SERIEs*, 12:73-100
- Lorenzo L, Lorenzo-Freire S (2009) A characterization of Kruskal sharing rules for minimum cost spanning tree problems. *Int J Game Theory* 38(1):107–126

See Also

mcstFolk, mcstOWShapley for other rules based on Kruskal's algorithm.

mcstRules for an overview of the available rules and analysis tools in the package.

Examples

```
# Simple vector input with custom weights
mcstPWShapley(c(12, 15, 20, 4, 6, 8), weights = c(2, 1, 1))

# Matrix input with equal weights (equivalent to folk rule)
C_mat <- matrix(c(0, 12, 15, 12,
                 12, 0, 4, 6,
                 15, 4, 0, 8,
                 12, 6, 8, 0), byrow = TRUE, ncol = 4)
mcstPWShapley(C_mat, weights = rep(1,3), draw = TRUE, titles = FALSE)
```

mcstRules

Overview of Rules and Analysis Tools for Minimum Cost Spanning Tree Problems

Description

This page provides an overview of the cost allocation rules implemented in this package for minimum cost spanning tree problems (MCSTP). The rules are organized into two main approaches: those defined directly through algorithms and decomposition techniques, and those defined through the analysis of associated cooperative games. It also includes a suite of analysis tools.

Rules defined through the problem

These rules are obtained directly from the network's structure and classical MST algorithms.

Based on Prim's algorithm:

These rules follow the growth of a single tree starting from the source.

- mcstBird: agents sequentially connect to the growing tree, paying the cost of the arc through which they connect to the tree.
- mcstDuttaKar: a modification of Bird's rule that introduces a pivotal switch in the allocation cost at each step to ensure cost monotonicity.

Based on Kruskal's algorithm:

Based on obligation functions, these rules follow Kruskal's logic of merging connected components using the cheapest available arcs.

- mcstFolk: the agents divide the cost of connecting their components equally (ERO rule).
- mcstOWShapley: optimistic weighted Shapley rule. A generalization of the folk rule that distributes costs proportionally based on asymmetric positive weights.
- mcstPWShapley: pessimistic weighted Shapley rule. An alternative weighted extension based on a permutation-driven obligation function.

Based on Boruvka's algorithm:

- `mcstBoruvka`: the agents pay the maximum possible proportion of the cheapest arc selected by their component. The numerical outcome is mathematically equivalent to the `mcstFolk`.

Based on a cone-wise decomposition:

- `mcstCone`: a technique that decomposes any general MCSTP into a nonnegative combination of elementary problems (where costs are 0 or 1). It allows extending rules like "folk", "owshapley", or the Bogomolnaia and Moulin family ("bogomolnaia") to general networks.

Rules defined through cooperative games

These rules are defined by first associating a cooperative game (in characteristic function form v) to the MCSTP and then applying a solution concept (like the Shapley value or the Nucleolus).

Cooperative games:

- `mcstGamePrivate`: defines the cost of a coalition as the MST cost of the subproblem formed by its members and the source (pessimistic approach).
- `mcstKar`: defined as the Shapley value of the associated private game.
- `mcstGameIrred`: based on the irreducible matrix $\bar{C}$, where costs reflect the cheapest way to connect each pair of nodes through the network.
- `mcstGameOpt`: allows a coalition to connect through nodes outside the coalition at no cost.
- `mcstGamePublic`: assigns to each coalition the minimum cost achievable when connections through other agents are publicly available.
- `mcstGameCC`: based on the cycle-complete matrix C^* , which induces a concave cooperative game.

Solution concepts:

- Shapley value:
 - Based on permutations: the average of marginal contributions over all $n!$ possible orderings (Π_N):

$$Sh_i(N, v) = \frac{1}{n!} \sum_{\pi \in \Pi_N} [v(Pre(i, \pi) \cup \{i\}) - v(Pre(i, \pi))],$$

where $Pre(i, \pi)$ is the set of players that precede agent i in the ordering π .

- Based on coalitions: a weighted sum of marginal contributions over all possible subcoalitions S that do not contain i :

$$Sh_i(N, v) = \sum_{S \subseteq N \setminus \{i\}} \frac{|S|!(n - |S| - 1)!}{n!} [v(S \cup \{i\}) - v(S)].$$

The Shapley value is the unique solution that satisfies the properties of efficiency, symmetry, additivity, and the null player property.

- Nucleolus: the unique cost allocation x that lexicographically minimizes the maximum excesses (complaints) of all coalitions. In a cost game, the excess e of a coalition S measures its dissatisfaction:

$$e(S, x) = \sum_{j \in S} x_j - v(S).$$

The nucleolus ensures that the most dissatisfied coalition is as happy as possible. It is always efficient and lies within the Core if the Core is non-empty.

Analysis tools

The package includes a comprehensive suite of tools to analyze the stability, sensitivity, and geometric properties of the cost allocations and the network:

- `mcstCoreCheck`: evaluates whether a given cost allocation belongs to the core of a cooperative game, identifying any blocking coalitions if unstable.
- `mcstCorePoints`: determines if the core of a game is empty via Linear Programming, computes a feasible stable allocation, and finds the exact geometric vertices of the core polytope.
- `mcstCorePlot`: generates geometric visualizations of the core region and the imputation set for 2, 3, or 4 players, allowing for a graphical stability check.
- `mcstStabilityRange`: calculates the stability range (lower and upper bounds) for the cost of individual arcs without altering the optimal tree topology.
- `mcstSensitivity`: performs sensitivity analysis by varying the costs of specific arcs (individually, jointly, or independently) and observing the dynamic impact on the agents' allocations.
- `mcstCompare`: computes and compares the cost allocations of four standard algorithmic rules (Bird, Dutta-Kar, folk, and Kar) simultaneously.

Note

When the optimal tree is not unique, rules based on Prim's algorithm compute the symmetric average allocation over all possible permutations. In contrast, rules based on Kruskal's and Boruvka's algorithms, as well as the cone-wise decomposition, are unaffected by tie-breaking choices.

For game-theoretic rules, the functions in this package can compute either the Shapley value or the Nucleolus of the associated cooperative game.

References

Bergantiños, G., & Vidal-Puga, J. (2021). A review of cooperative rules and their associated algorithms for minimum-cost spanning tree problems. *SERIEs*, 12:73-100.

mcstSensitivity

Sensitivity Analysis for Cost Allocations

Description

This function performs a sensitivity analysis on cost allocation rules for a minimum cost spanning tree (MCST) problem by varying the costs of specific arcs and observing the impact on the agents' cost allocation.

Usage

```
mcstSensitivity(
  C,
  rule = c("bird", "duttakar", "folk", "kar"),
  arcs,
  delta = list(c(-5, 5), c(-3, 3)),
  step = 1,
  independent = FALSE
)

## S3 method for class 'mcstp_sens'
plot(x, titles = TRUE, ...)
```

Arguments

<code>C</code>	cost matrix between nodes. Accepts multiple formats (see "Supported formats for C" in <code>mcstBird</code>).
<code>rule</code>	character string specifying the allocation rule to evaluate. Must be one of "bird", "duttakar", "folk", or "kar".
<code>arcs</code>	a vector of length 2 specifying a single arc (i, j) , or a list of vectors specifying multiple arcs to analyze.
<code>delta</code>	a numeric vector specifying the lower and upper limits of cost variation $[\Delta LB, \Delta UB]$, or a list of two vectors if analyzing two independent arcs. Default is <code>list(c(-5, 5), c(-3, 3))</code> .
<code>step</code>	numeric value specifying the step size for the sequence of variations. Default is 1.
<code>independent</code>	logical; if TRUE, evaluates the cross-sensitivity of exactly two arcs independently, generating an evaluation grid. Default is FALSE.
<code>x</code>	the output from <code>mcstSensitivity</code> .
<code>titles</code>	logical; if TRUE (default), adds informative main titles and axis labels to the plots.
<code>...</code>	additional graphical parameters passed to the plot function.

Details

The sensitivity analysis can be conducted under three different scenarios depending on the structure of the arcs parameter and the independent flag:

1. Single arc variation: when a single arc is provided, its cost varies along the designated delta interval. The output plot displays evolution lines for each agent's cost allocation relative to the absolute cost of the arc.
2. Joint arc variation: when multiple arcs are provided and `independent = FALSE`, the same delta variation value is applied simultaneously to all selected arcs.
3. Independent variation of two arcs: when exactly two arcs are provided and `independent = TRUE`, a grid of cross-variations is constructed using `delta1` and `delta2`. The output plot renders a selection of heatmaps (one per agent) mapping the allocations across the variation space.

Value

A list containing:

- `delta` (or `delta1` and `delta2`): the variation step values applied.
- `arc_cost`: the resulting absolute cost of the arc (computed only in the single arc case).
- Columns named after each agent showing the dynamic cost allocations obtained under the selected rule.

See Also

`mcstBird`, `mcstDuttaKar`, `mcstFolk`, `mcstKar` for the rule definitions.

`mcstStabilityRange` for calculating static boundaries.

`mcstRules` for an overview of the available rules and analysis tools in the package.

Examples

```
# Matrix input
C_mat <- matrix(c(0, 10, 15, 20,
                  10, 0, 25, 12,
                  15, 25, 0, 8,
                  20, 12, 8, 0), byrow = TRUE, ncol = 4)

# Example 1: Single arc sensitivity analysis
sens_single <- mcstSensitivity(C_mat, rule = "bird", arcs = c("0", "1"), delta = c(-5, 15))
plot(sens_single)

# 2. Joint sensitivity analysis for multiple arcs
arcs_list <- list(c("0", "1"), c("2", "3"))
sens_joint <- mcstSensitivity(C_mat, rule = "folk", arcs = arcs_list, delta = c(-3, 3))
plot(sens_joint)
```

mcstStabilityRange	<i>Stability Range for MCST Arcs</i>
--------------------	--------------------------------------

Description

This function calculates the stability range for the arcs of a minimum cost spanning tree (MCST). It determines how much the cost of an individual arc can increase or decrease without changing the topology of the optimal tree.

Usage

```
mcstStabilityRange(C)
```

Arguments

C	cost matrix between nodes. Accepts multiple formats (see "Supported formats for C" in mcstBird).
---	--

Details

The stability range of an arc (i, j) indicates the interval $[LB, UB]$ within which its original cost c_{ij} can vary while preserving the current minimum cost spanning tree as optimal.

At the finite boundary points of the stability interval, the minimum cost spanning tree becomes non-unique due to cost ties. In these scenarios, component or path-based rules (e.g., Folk, Kar) remain stable, whereas permutation-based rules such as Bird's rule or Dutta-Kar's rule average allocations over all alternative optimal trees, which can induce non-proportional jumps in the final cost allocation vector.

Value

A data frame containing:

- from: the starting node of the arc.
- to: the ending node of the arc.
- in_mst: logical; if TRUE, the arc belongs to the optimal tree.

- cost: the original cost of the arc c_{ij} .
- range: a formatted string showing the allowed cost variation $[\Delta LB, \Delta UB]$.
- LB: the absolute lower bound for the arc's cost.
- UB: the absolute upper bound for the arc's cost.

References

Tarjan RE (1982) Sensitivity analysis of minimum spanning trees and shortest path trees. Information Processing Letters, 14(1):30–33

See Also

mcstBird, mcstDuttaKar, mcstFolk, mcstKar for allocation rules.

mcstRules for an overview of the available rules and analysis tools in the package.

Examples

```
# Matrix input
C_mat <- matrix(c(0, 10, 15, 20,
                 10, 0, 25, 12,
                 15, 25, 0, 8,
                 20, 12, 8, 0), byrow = TRUE, ncol = 4)

# Calculate stability range
mcstStabilityRange(C_mat)
```

Index

* Algorithmic Rules

mcstBird, 2
 mcstBoruvka, 5
 mcstCone, 8
 mcstDuttaKar, 15
 mcstFolk, 18
 mcstOWShapley, 38
 mcstPWSapley, 40
 mcstRules, 42

* Analysis Tools

mcstRules, 42

* Comparison

mcstCompare, 7

* Cooperative Games

mcstCoreCheck, 10
 mcstCorePlot, 12
 mcstCorePoints, 14
 mcstGameCC, 20
 mcstGameIrred, 24
 mcstGameOpt, 27
 mcstGamePrivate, 30
 mcstGamePublic, 33
 mcstKar, 36
 mcstRules, 42

* Core Stability

mcstCoreCheck, 10
 mcstCorePlot, 12
 mcstCorePoints, 14

* Linear Programming

mcstCorePoints, 14

* MCSTP

mcstBird, 2
 mcstBoruvka, 5
 mcstCompare, 7
 mcstCone, 8
 mcstDuttaKar, 15
 mcstFolk, 18
 mcstGameCC, 20
 mcstGameIrred, 24
 mcstGameOpt, 27
 mcstGamePrivate, 30
 mcstGamePublic, 33
 mcstKar, 36

mcstOWShapley, 38
 mcstPWSapley, 40
 mcstRules, 42
 mcstSensitivity, 44
 mcstStabilityRange, 46

* Sensitivity Analysis

mcstSensitivity, 44
 mcstStabilityRange, 46

* Visualization

mcstCorePlot, 12

barplot, 7, 11

igraph, 4, 6, 10, 17, 19, 23, 26, 29, 32, 35, 39, 41

mcstBird, 2, 7, 18, 26, 32, 36, 42, 45–47
 mcstBoruvka, 5, 20, 43
 mcstCompare, 7, 44
 mcstCone, 8, 20, 39, 43
 mcstCoreCheck, 10, 13, 15, 44
 mcstCorePlot, 12, 14, 15, 44
 mcstCorePoints, 14, 44
 mcstDuttaKar, 4, 7, 15, 32, 36, 42, 45, 47
 mcstFolk, 6, 7, 10, 18, 26, 29, 32, 36, 39, 41–43, 45, 47
 mcstGameCC, 20, 26, 32, 43
 mcstGameIrred, 4, 12, 20, 23, 24, 29, 32, 36, 43
 mcstGameOpt, 20, 26, 27, 33, 43
 mcstGamePrivate, 11–14, 23, 26, 29, 30, 36, 37, 43
 mcstGamePublic, 26, 33, 33, 43
 mcstKar, 7, 31, 32, 36, 43, 45, 47
 mcstOWShapley, 10, 20, 38, 42
 mcstPWSapley, 20, 39, 40, 42
 mcstRules, 4, 6, 7, 10, 12, 13, 15, 18, 20, 22, 23, 25, 26, 28, 29, 31, 33, 34, 36, 37, 39, 42, 42, 45, 47
 mcstSensitivity, 44, 44
 mcstStabilityRange, 44, 45, 46
 nucleolus, 22, 25, 28, 31, 34
 plot.mcstp_compare (mcstCompare), 7

`plot.mcstp_core (mcstCoreCheck)`, 10
`plot.mcstp_core_point (mcstCorePoints)`,
14
`plot.mcstp_sens (mcstSensitivity)`, 44
`shapleyValue`, 22, 25, 28, 31, 34

Bibliografía

- Antonov M, Csárdi G, Horvát S, Müller K, Nepusz T, Noom D, Salmon M, Traag V, Welles BF, Zanini F (2023) igraph enables fast and robust network analysis across programming languages. arXiv preprint arXiv:2311.10260.
- Bergantiños G, Lorenzo L (2004) A non-cooperative approach to the cost spanning tree problem. *Mathematical Methods of Operations Research* 59(3):393–403.
- Bergantiños G, Lorenzo L, Lorenzo-Freire S (2010) The family of cost monotonic and cost additive rules in minimum cost spanning tree problems. *Social Choice and Welfare* 34(4):695–710.
- Bergantiños G, Lorenzo L, Lorenzo-Freire S (2011) A generalization of obligation rules for minimum cost spanning tree problems. *European Journal of Operational Research* 211(1):122–129.
- Bergantiños G, Lorenzo-Freire S (2008a) “Optimistic” weighted Shapley rules in minimum cost spanning tree problems. *European Journal of Operational Research* 185(1):289–298.
- Bergantiños G, Lorenzo-Freire S (2008b) A characterization of optimistic weighted Shapley rules in minimum cost spanning tree problems. *Economic Theory* 35(3):523–538.
- Bergantiños G, Vidal-Puga J (2007) The optimistic TU game in minimum cost spanning tree problems. *International Journal of Game Theory* 36(2):223–239.
- Bergantiños G, Vidal-Puga J (2011) The folk solution and Borůvka’s algorithm in minimum cost spanning tree problems. *Discrete Applied Mathematics* 159(12):1279–1283.
- Bergantiños G, Vidal-Puga J (2021) A review of cooperative rules and their associated algorithms for minimum-cost spanning tree problems. *SERIEs* 12(1):73–100.
- Berkelaar M (2024) lpSolve: Interface to ‘Lp_solve’ v. 5.5 to Solve Linear/Integer Programs. R package version 5.6.23. <https://CRAN.R-project.org/package=lpSolve>.
- Bird CG (1976) On cost allocation for a spanning tree: a game theoretic approach. *Networks* 6(4):335–350.
- Bogomolnaia A, Moulin H (2010) Sharing a minimal cost spanning tree: Beyond the Folk solution. *Games and Economic Behavior* 69(2):238–248.
- Borůvka O (1926) O jistém problému minimálním. *Práce moravské přírodovědecké společnosti v Brně* 3:37–58.
- Branzei R, Moretti S, Norde H, Tijs S (2004) The P-value for cost sharing in minimum cost spanning tree situations. *Theory and Decision* 56(1):47–61.
- Camerini PM (1978) The min-max spanning tree problem and some extensions. *Information Processing Letters* 7(1):10–14.
- Castro J, Gómez D, Tejada J (2009) Polynomial calculation of the Shapley value based on sampling. *Computers & operations research* 36(5):1726–1730.

- Csárdi G, Nepusz T (2006) The igraph software package for complex network research. *InterJournal Complex Systems*:1695. <https://igraph.org>.
- Csárdi G, Nepusz T, Traag V, Horvát S, Zanini F, Noom D, Müller K, Schoch D, Salmon M (2026) igraph: Network Analysis and Visualization in R. R package version 2.2.2. <https://CRAN.R-project.org/package=igraph>.
- Diestel R (2017) Graph Theory. 5th. Vol. 173. Graduate Texts in Mathematics. Springer, Berlín.
- Douglas Nychka, Reinhard Furrer, John Paige, Stephan Sain (2021) fields: Tools for spatial data. R package version 17.3. Boulder, CO, USA: University Corporation for Atmospheric Research. <https://github.com/dnychka/fieldsRPackage>.
- Dutta B, Kar A (2004) Cost monotonicity, consistency and minimum cost spanning tree games. *Games and Economic Behavior* 48(2):223–248.
- Feltkamp V, Tijs SH, Muto S (1994) On the irreducible core and the equal remaining obligations rule of minimum cost spanning extension problems.
- Fontenla M (2014a) cooptrees: Cooperative aspects of optimal trees in weighted graphs. R package version 1.0. <https://CRAN.R-project.org/package=cooptrees>.
- Fontenla M (2014b) optrees: Optimal Trees in Weighted Graphs. R package version 1.0. <https://CRAN.R-project.org/package=optrees>.
- Fontenla Cadavid M (2014) Optimal Trees: Programación en R de problemas de búsqueda de árboles óptimos. Trabajo Fin de Máster, Universidade de Santiago de Compostela. http://eio.usc.es/pub/mte/descargas/ProyectosFinMaster/Proyecto_1075.pdf.
- Geyer CJ, Meeden GD, Fukuda K (2026) rcdd: Computational Geometry. R package version 1.6-1. <https://CRAN.R-project.org/package=rcdd>.
- Gillies DB (1953) Some theorems on n-person games. Princeton University.
- Gómez Saura D (2016) Reparto de costes en problemas de árbol generador de coste mínimo. Trabajo Fin de Máster, Universidad de Murcia.
- González-Díaz J, García-Jurado I, Fiestras-Janeiro MG (2023) An introductory course on mathematical game theory and applications. 2nd. Vol. 238. American Mathematical Society, United States of America.
- González-Díaz J, Sánchez-Rodríguez E (2014) Understanding the coincidence of allocation rules: symmetry and orthogonality in TU-games. *International Journal of Game Theory* 43(4):821–843.
- Habel K, Grasman R, Gramacy RB, Mozharovskiy P, Sterratt DC (2025) geometry: Mesh Generation and Surface Tessellation. R package version 0.5.2. <https://CRAN.R-project.org/package=geometry>.
- Kar A (2002) Axiomatization of the Shapley value on minimum cost spanning tree games. *Games and Economic Behavior* 38(2):265–277.
- Kruskal JB (1956) On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical society* 7(1):48–50.
- Lorenzo L, Lorenzo-Freire S (2009) A characterization of Kruskal sharing rules for minimum cost spanning tree problems. *International Journal of Game Theory* 38(1):107–126.
- Moor D de, Borm P, Bosma W (2019) Optimization and Allocation in Minimum Cost Spanning Tree Problems with Time. Master's thesis, Radboud University.
- Nešetřil J, Milková E, Nešetřilová H (2001) Otakar Borůvka on minimum spanning tree problem Translation of both the 1926 papers, comments, history. *Discrete mathematics* 233(1-3):3–36.

- Norde H, Moretti S, Tijs S (2004) Minimum cost spanning tree games and population monotonic allocation schemes. *European Journal of Operational Research* 154(1):84–97.
- Owen AB (2013) Monte Carlo theory, methods and examples. <https://artowen.su.domains/mc/>.
- Prim RC (1957) Shortest connection networks and some generalizations. *The Bell System Technical Journal* 36(6):1389–1401.
- R Core Team (2026) R: A Language and Environment for Statistical Computing. R Foundation for Statistical Computing. Vienna, Austria. <https://www.R-project.org/>.
- Schmeidler D (1969) The nucleolus of a characteristic function game. *SIAM Journal on applied mathematics* 17(6):1163–1170.
- Shapley LS (1953) A value for n-person games. Ed. por Kuhn HW, Tucker AW:307–317.
- Shapley LS (1971) Cores of convex games. *International journal of game theory* 1(1):11–26.
- Sievert C (2020) Interactive Web-Based Data Visualization with R, plotly, and shiny. Chapman y Hall/CRC. ISBN: 9781138331457. <https://plotly-r.com>.
- Souto Vázquez L (2026) mcstprules: Cost Allocation Rules in Minimum Cost Spanning Tree Problems. R package version 0.1.0, <https://luciasouto13.github.io/mcstprules>. <https://github.com/luciasouto13/mcstprules>.
- Staudacher J, Anwander J (2019) Using the R package CoopGame for the analysis, solution and visualization of cooperative games with transferable utility. R Vignette. <https://cran.r-project.org/package=CoopGame>.
- Tarjan RE (1982) Sensitivity analysis of minimum spanning trees and shortest path trees. *Information Processing Letters* 14(1):30–33.
- Tijs S, Branzei R, Moretti S, Norde H (2006) Obligation rules for minimum cost spanning tree situations and their monotonicity properties. *European journal of operational research* 175(1):121–134.
- Trudeau C (2012) A new stable and more responsive cost sharing solution for minimum cost spanning tree problems. *Games and Economic Behavior* 75(1):402–412.
- Trudeau C, Vidal-Puga J (2019) The Shapley value in minimum cost spanning tree problems. En: *Handbook of the Shapley value*. Chapman y Hall/CRC. pp 537–559.
- Warnes GR, Bolker B, Lumley T, Magnusson A, Venables B, Ryodan G, Moeller S (2023) gtools: Various R Programming Tools. R package version 3.9.5. <https://CRAN.R-project.org/package=gtools>.
- Wickham H, Bryan J (2023) readxl: Read Excel Files. R package version 1.4.3. <https://CRAN.R-project.org/package=readxl>.
- Wickham H, Hesselberth J, Salmon M, Roy O, Brüggemann S (2025) pkgdown: Make Static HTML Documentation for a Package. R package version 2.2.0. <https://CRAN.R-project.org/package=pkgdown>.