USC UNIVERSIDADE DE SANTIAGO DE COMPOSTELA
FACULTADE DE MATEMÁTICAS

Traballo Fin de Grao

Análise e aplicacións do algoritmo de Dijkstra na optimización de rutas: na busca do camiño máis curto

Adrián Bacariza Villamarín

Xullo, 2025

UNIVERSIDADE DE SANTIAGO DE COMPOSTELA

GRAO DE MATEMÁTICAS

Traballo Fin de Grao

Análise e aplicacións do algoritmo de Dijkstra na optimización de rutas: na busca do camiño máis curto

Adrián Bacariza Villamarín

Xullo, 2025

UNIVERSIDADE DE SANTIAGO DE COMPOSTELA

Traballo proposto

Área de Coñecemento: Optimización
Título: Análise e aplicacións do algoritmo de Dijkstra na optimización de rutas: na busca do camiño máis curto
Breve descrición do contido
A proposta deste traballo é levar a cabo una revisión do problema do camiño máis curto e estudar os diversos algoritmos propostos na literatura para a súa resolución. Dentro da variedade de algoritmos existentes, prestarase especial atención ó algoritmo de Dijkstra. O estudo centrarase tanto na análise computacional como na base matemática dos algoritmos. Ademais, tratarase de mostrar a súa aplicación plantexando algún problema da vida real. O traballo poderase centrar nos seguintes aspectos: ▪ Introdución ao problema do camiño máis curto. ▪ Estudo dos distintos algoritmos de resolución do problema do camiño máis curto (centrándose principalmente no algoritmo de Dijkstra e variantes). ▪ Aplicacións.
Recomendacións
Outras observacións

Agradecementos

A realización deste traballo non podería ter chegado a bo porto sen moitas persoas.

En primeiro lugar, quero agradecer ao meu titor Ángel Manuel Gonzalez Rueda a súa implicación no traballo, estando sempre aí para min, aconsellándome en todo o que eu precisaba e por ter a paciencia necesaria para aguantar todos os meus sucesivos retrasos. Grazas por ser tan boa persoa, pois asegúroche que este traballo é tamén debido á túa amabilidade e forma feliz, optimista e tranquila de ver a vida, cualidades que me transmitiches en todo momento e que fixeron que elaborar este traballo fose moi sinxelo.

En segundo lugar, quero agradecer á miña familia o apoio que me brindaron durante toda a miña vida, motivándome a seguir adiante e a esforzarme. Quero mostrar especial agradecemento aos meus pais Josefa e Manuel e ao meu irmán Bruno pola paciencia que tiveron, por acompañarme nos momentos máis difíciles e alegrándose comigo nos momentos felices.

Quero agradecer tamén aos meus amigos Sonia, Mila, Paula, María, Carlota, Daniela, Lara, Irene, Uxía, Diego, Pablo, Álvaro e Xaquín polos bos momentos que pasamos xuntos, rindo e desfrutando e por aguantar as miñas parvadas; así como por motivarme nos peores momentos e evitar que tomase decisións das que logo me arrepentiría.

Por último, quero agradecer aos profesores que me deron clase durante toda a miña vida académica porque este momento non sería real sen a contribución de todos e cada un deles. En especial, de Gutier, o meu profesor de matemáticas nos dous primeiros cursos da ESO, pois grazas a el decidín cursar o dobre grao que estou a piques de rematar e esforceime ao máximo para ter a posibilidade de cursalo.

Grazas a todos e cada un de vós. Sen vós, este soño meu non tería sido posible.

Índice

Resumo

Neste traballo partimos de conceptos básicos relacionados con problemas de fluxo en redes para posteriormente indagar sobre outros máis específicos que nos axudarán no noso principal obxectivo: a explicación de distintos algoritmos para resolver o problema do camiño máis curto.

No primeiro capítulo introduciremos nocións de teoría de grafos e explicaremos conceptos importantes para os seguintes capítulos como o concepto de nó, arco ou custo. Ademais, falaremos dos problemas de fluxo en redes con custo mínimo engadindo a súa formulación matemática. Por último, introduciremos unha propiedade moi importante á hora de resolver este tipo de problemas: a unimodularidade; xunto con algunhas propiedades que facilitan a resolución dos mesmos.

No segundo capítulo centrarémonos no problema do camiño máis curto, explicando que hai distintos tipos de algoritmos para a súa resolución e focalizándonos especialmente no algoritmo de Dijkstra. Ademais, tamén agregaremos algunhas implementacións para o mesmo ou outros algoritmos que poden axilizar a obtención da solución do problema. As explicacións desta sección ilustraranse con algún exemplo para facilitar a comprensión dos respectivos algoritmos.

Por último, no último capítulo mostraremos unha aplicación real do cálculo do camiño máis curto no ámbito da red ferroviaria española. Detallaremos o programa implementado en Python, describiremos os datos empregados e presentaremos unha visualización gráfica das rutas obtidas.

Abstract

In this project, we start off from basic concepts regarding flow problems in networks and then looked at more specific ones that would help us achieve our main goal, which is to explain different algorithms for solving the shortest path problem.

In the first chapter we will introduce concepts from graph theory and explain important notions for the following chapters, such as the definition of node, arc or cost. In addition, we will introduce minimun cost flow problems joint with their mathematical formulations. Finally, we will analyze a very important property when solving this type of problem: unimodularity, along with some properties that make solving them easier.

In the second chapter we will focus on the shortest path problem, describing that there are different types of algorithms for solving it with particular emphasis on Dijkstra's algorithm. We will also provide some implentations of this and other algorithms that can help solve the problem more efficiently. Additionally, the explanations will be accompanied by examples to make the algorithms easier to understand.

Finally, the last chapter presents a practical application of shortest path computation within the Spanish railway network. It includes a detailed description of the Python implementation, the data used, and a graphical representation of the computed routes.

Introdución

Dende os seus albores, o ser humano sempre tivo moitos problemas que afrontar na súa evolución ata o día de hoxe. Un dos máis importantes e dos que abarca maiores campos (tanto dende o punto de vista loxístico, bélico coma de desenvolvemento) é o problema do camiño máis curto. Na actualidade está á orde do día, pois é utilizado non soamente polos sistemas de cálculo de rutas como Google Maps; senón tamén por sistemas informáticos para unha maior eficiencia á hora de transportar datos. É por iso que é especialmente no último século cando cobrou maior importancia, con moitos resultados teóricos desta época.

O problema non é novo, pero a teoría de grafos na que se basa xurdiu no século XVIII, co problema das pontes de Königsberg de Leonhard Euler. A súa importancia comezou a aumentar na Segunda Guerra Mundial, cando a xestión da loxística militar requiriu solucións máis óptimas para o transporte. Así comezaron a formalizarse os principais algoritmos para a solución do camiño máis curto nos anos 50 do século pasado, como o coñecido algoritmo de Dijkstra.

Coa irrupción da informática, os cálculos aceleráronse e abordáronse redes cada vez máis extensas. Aplicacións como Google Maps, que contén redes moi vastas, impulsaron a necesidade de mellorar a eficiencia dos algoritmos existentes co obxectivo de reducir o tempo de cómputo.

Na actualidade, o problema do camiño máis curto xa está presente na nosa vida diaria (por exemplo, ao facermos unha viaxe en coche), pero ademais tamén se aplica en ámbitos como a optimización de redes de comunicación e ao transporte de datos ou incluso a intelixencia artificial.

Neste traballo afondarase sobre ese problema, analizando os principais algoritmos empregados na súa resolución, explorando as súas aplicacións en diferentes áreas, e comparando diversos algoritmos entre si para determinar cales serán máis axeitados en función do contexto co que traballemos. Prestaremos especial atención ao algoritmo de Dijkstra, o cal estudaremos en profundidade, indagando na súa base matemática e a súa complexidade.

As referencias consultadas para elaborar este traballo son principalmente [1], [2] e os apuntamentos da asignatura Programación Linear e Enteira do profesor Julio González Díaz [6]. Ademais destas, cómpre salientar outras fontes de gran relevancia para o tema tratado. En [5],

Dijkstra presenta por primeira vez o algoritmo que leva o seu nome, converténdose nunha das contribucións fundamentais á teoría de grafos e algoritmos de rutas máis curtas. A implementación deste algoritmo proposta por Dial, baseada en estruturas de datos especializadas para mellorar a súa eficiencia, está detallada no artigo [4]. Pola súa banda, Bellman introduce en [3] a implementación do algoritmo xenérico de etiquetado móbil empregando unha estratexia FIFO, o que representa unha achega clave ao desenvolvemento de algoritmos de etiquetado móbil.

Capítulo 1

Problemas de optimización en redes

Para desenvolver toda a teoría do problema do camiño máis curto habemos de comezar definindo o que é un grafo (básico para a futura análise do problema) xunto cuns conceptos relacionados. Con esta teoría e algúns exemplos de por medio para facilitar a súa comprensión, exporemos logo diversos problemas que se poden resolver utilizando a teoría de grafos e mostrando a súa formalización matemática. É neste capítulo ademais onde introducimos o problema do camiño máis curto, que é no que nos centraremos nos próximos capítulos (no Capítulo 2 falaremos sobre a súa base matemática e no Capítulo 3 mostraremos a solución dun modelo real).

1.1. Fundamentos da teoría de grafos

Para desenvolver o problema do camiño máis curto é necesario introducir certos conceptos básicos da Teoría de Grafos. O primeiro e máis importante é saber que un **grafo dirixido** $G = (N, A)$ consiste nun conxunto N de *nós* (tamén denominados **vértices**) e un conxunto A de **arcos** (tamén denominados **arestas** ou **conexións**) cuxos elementos son pares ordenados de nós distintos. Cada nó é identificado e representado por unha etiqueta (nós usaremos números) e os arcos por unha dupla destas etiquetas onde a primeira indica o nó dende onde emana o arco e a segunda indica o nó onde remata o arco. Dous nós unidos por un arco diremos que son adxacentes.

Na Figura 1.1 incluímos un exemplo dun grafo dirixido. Neste exemplo temos os seguintes conxuntos de nós e arcos:

- $N = \{1, 2, 3, 4, 5, 6, 7, 8\}$
- $A = \{(1,2), (1,3), (2,4), (3,2), (3,5), (4,5), (4,6), (4,7), (4,8), (6,3), (7,2)\}$

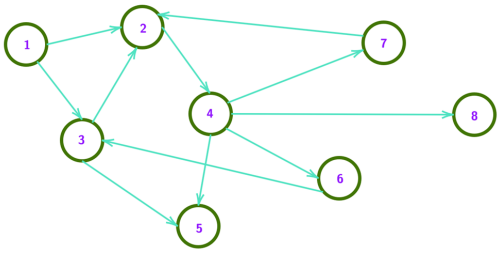

Figura 1.1: Exemplo de grafo dirixido.

Un concepto que nos servirá para futuras definicións é o de subgrafo. Un **subgrafo** dun grafo $G = (N, A)$ é un grafo $G' = (N', A')$ tal que $N' \subseteq N$ e $A' \subseteq A$. Ademais, un subgrafo $G' = (N', A')$ pode dicirse que está **inducido por** N' se A' contén cada arco de A cuxos ambos extremos pertenzan a N'. Por outra banda, un subgrafo $G' = (N', A')$ dise **de expansión** (*spanning graph*) dun grafo $G = (N, A)$ se cumpre $N' = N$ e $A' \subseteq A$ (isto significa que o subgrafo contén todos os nós presentes no grafo e pode non ter todas as conexións do grafo principal).

Continuando co exemplo da Figura 1.1, un subgrafo pode ser o formado polos nós $1, 2$ e 3 e as arcos $\{(1,3), (3,2)\}$. Se queremos que este subgrafo estea inducido por ditos nós, ao conxunto de arcos habería que engadir o arco $\{(1,2)\}$, xa que este arco ten os dous extremos no conxunto de nós que determinan o subgrafo. Un subgrafo de expansión do grafo anterior pode ser o formado polos nós $\{1, 2, 3, 4, 5, 6, 7, 8\}$ e polos arcos $\{(1,2), (3,2), (3,5), (4,5), (4,7), (4,8), (6,3), (7,2)\}$.

Centrándonos nos nós e nas arestas dun grafo, sexa agora un arco (i, j). O nó i dise que é o **nó de orixe**, e o nó j dise que é o **nó de destino** do arco. A **lista de adxacencia de arcos** $A(i)$ dun nó i é o conxunto de arcos que emanan dese nó, é dicir, $A(i) = \{(i, j) \in A \mid j \in N\}$. A **lista de adxacencia de nós** $N(i)$ é o conxunto de nós adxacentes ao nó i: $N(i) = \{j \in N \mid (i, j) \in A\}$. A **lista de adxacencia inversa de arcos** dun nó i é o conxunto de arcos que terminan en i, e a **lista de adxacencia inversa de nós** dun nó i é o conxunto de nós tales que existe un arco emanando del e rematando no nó i.

Con estas primeiras definicións, podemos agora pasar a falar sobre conceptos que abranguen máis dun nodo e/ou aresta. Así, un **percorrido** $i_j - \cdots - i_k$ nun grafo dirixido $G = (N, A)$ é un subgrafo de G consistindo nunha secuencia de nodos e arcos tales que ou ben os seus arcos $a_k = (i_k, i_{k+1})$ pertencen a A ou ben os seus inversos $a_k = (i_{k+1}, i_k)$ pertencen a A. Un percorrido dise **dirixido** se todos os seus arcos dirixidos pertencen a A. Un **camiño** é un percorrido que non pasa máis dunha vez por cada nó. Dentro dos camiños diferenciamos entre **arcos directos** (que son aqueles tales que o índice do nó de destino é maior que o índice do nó de orixe) e **arcos inversos** (cando o índice do nó de orixe é superior que o índice do nó de destino). Un **camiño dirixido** é un percorrido dirixido que non pasa máis dunha vez por cada nó.

No caso de que un percorrido $i_1 - \cdots - i_r$ estea acompañado dun arco (i_r, i_1) ou do seu inverso,

estariamos ante o caso dun **ciclo**. Este pode ser dirixido se o arco presente é o (i_r, i_1) e non o inverso. Un grafo será **acíclico** se non se atopan ciclos dentro del.

Continuando co exemplo anterior, mostramos na Figura 1.2 os diversos termos que veñen de definirse:

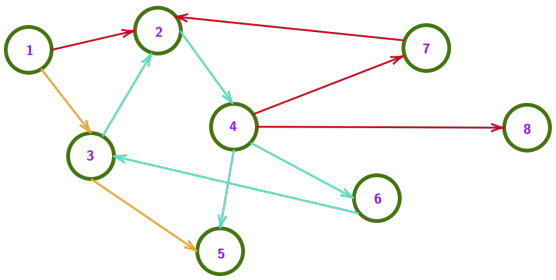

Figura 1.2: Exemplos dos termos anteriormente explicados.

- Dado o arco $(4, 6)$, o nó de orixe é o nó 4 e o nó de destino é o nó 6.
- A lista de adxacencia de arcos do nó 4, por exemplo, é o conxunto $\{(4, 5), (4, 6), (4, 7), (4, 8)\}$ e a lista de adxacencia inversa de nós do nó 2 é $\{1, 3, 7\}$.
- Un exemplo de percorrido é o sinalado en vermello $1 - 2 - 7 - 4 - 8$, pero non é dirixido porque o arco $(2, 7)$ non pertence ao conxunto de arcos (é o arco $(7, 2)$ o pertencente ao grafo dirixido). Pola contra, o sinalado en laranxa $1 - 3 - 5$ si que é un percorrido dirixido.
- Ambos percorridos (o vermello e o laranxa) son camiños, xa que pasan unicamente unha vez por cada nó; mentres que o vermello non é dirixido, o laranxa si que o é.
- Un exemplo de ciclo no grafo da imaxe é o $2 - 4 - 7 - 2$.

Con estas definicións xa podemos adentrarnos en dúas das diversas propiedades que poden presentar os grafos:

- **Conexión**: Diremos que dous nós i e j están conectados se o grafo contén polo menos un percorrido tal que comece no nó i e remate no nó j. Polo tanto, un grafo será conexo se todo par de nós están conectados.
- **Conexión forte**: Un grafo será fortemente conexo se contén polo menos un percorrido dirixido dende cada nó ata calquera outro nó do grafo.

Por último introducimos un termo que repetiremos moito durante o seguinte desenvolvemento: o corte. Un **corte** é unha partición do conxunto de nós N en dous subconxuntos S e $\overline{S}$ tales que $N = S \cup \overline{S}$. Este corte tamén dá lugar a un conxunto de arcos tales que teñen a súa orixe nun dos dous conxuntos e teñen o seu destino no outro.

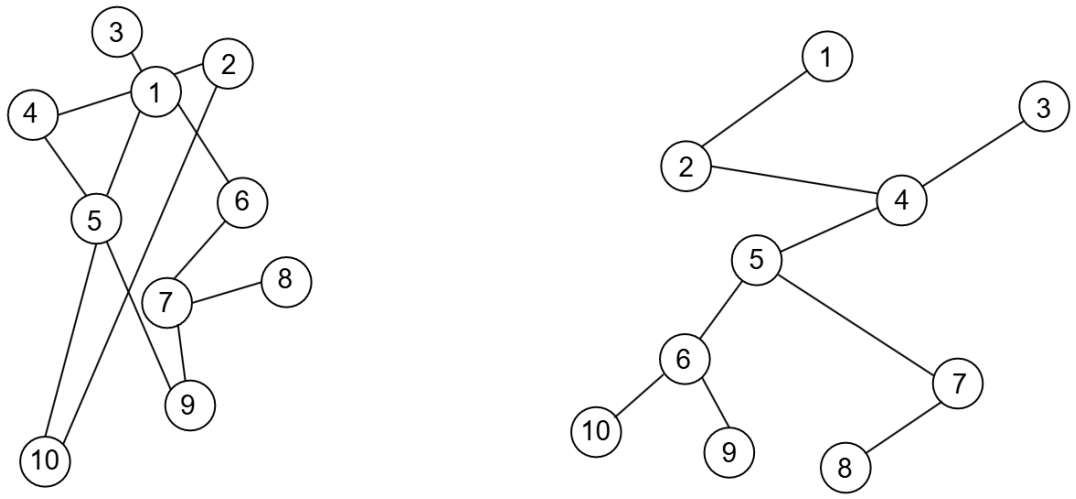

Figura 1.3: Exemplos para diferenciar unha árbore.

Outro concepto que utilizaremos é o de **árbore**, que se define como un grafo conexo que non contén ciclos. A maiores, unha *árbore de expansión* dun grafo G é unha árbore que é un subgrafo de expansión de G. Na Figura 1.3 podemos ver á esquerda un exemplo dun grafo que non é unha árbore (pois podemos trazar, por exemplo, o ciclo $1-2-10-5-1$) e á dereita un exemplo dun grafo que é unha árbore, pois non podemos atopar ciclo ningún.

1.1.1. As matrices de incidencia e de adxacencia

Sexa agora $G = (N, A)$ un grafo arbitrario. Podemos identificar os seus nós e os seus arcos, e con estes crear unha matriz que represente todo o grafo no seu conxunto indicando non só que nós están unidos por arcos, senón tamén a dirección destes arcos. Esta é a chamada **matriz de incidencia** B, de dimensións $n \times m$, e contén unha fila por cada nó e unha columna por cada arco. A columna correspondente con cada arco (i, j) ten soamente dúas entradas distintas de cero: ten un $+1$ na fila correspondente ao nó de orixe do arco e ten un -1 na fila correspondente ao nó de destino do arco. Esta matriz ten a propiedade de que soamente $2m$ entradas son non nulas. Ademais de que en cada columna hai un $+1$ e un -1, o número de $+1$ nunha fila é igual ao número de arcos emanando dese nó mentres que o número de -1 indica o número de arcos que rematan nese nó.

A modo ilustrativo, consideremos o grafo da Figura 1.1 (en adiante denominarémolo simple-

mente como grafo 1.1) e calculemos a súa matriz de incidencia asociada:

$B =$

	$(1,2)$	$(1,3)$	$(2,4)$	$(3,2)$	$(3,5)$	$(4,5)$	$(4,6)$	$(4,7)$	$(4,8)$	$(6,3)$	$(7,2)$
1	1	1	0	0	0	0	0	0	0	0	0
2	−1	0	1	−1	0	0	0	0	0	0	1
3	0	−1	0	1	1	0	0	0	0	−1	0
4	0	0	−1	0	0	1	1	1	1	0	0
5	0	0	0	0	−1	−1	0	0	0	0	0
6	0	0	0	0	0	0	−1	0	0	1	0
7	0	0	0	0	0	0	0	−1	0	0	1
8	0	0	0	0	0	0	0	0	−1	0	0

Con botar un simple vistazo a esta matriz podemos saber directamente cantos nós ten o grafo e cales son os arcos que pertencen ao grafo, así como cantas conexións ten cada nó tanto entrantes como saíntes.

Pola contra, a **matriz de adxacencia de nós** (ou simplemente **matriz de adxacencia**) $\mathcal{H}$ é unha matriz de orde $n \times n$ tal que ten unha fila e unha columna por cada nó. Nesta matriz $\mathcal{H}$, o elemento h_{ij} será igual a 1 se existe unha aresta conectando os nós i e j e 0 noutro caso.

Con isto explicado podemos pasar a construír a seguinte matriz de adxacencia referida ao grafo 1.1:

Como podemos ver nesta matriz, unha maior porcentaxe das entradas son nulas comparando coa matriz de incidencia. Isto provoca que esta representación sexa eficiente especialmente con redes moi densas (é dicir, que existan moitas conexións entre os distintos nós da rede). As filas da matriz indican os nós de orixe e as columnas indican os nós de destino, polo que para saber se o arco (i, j) pertence ao grafo simplemente hai que mirar se a entrada $h_{i,j}$ da matriz é distinta de cero.

$\mathcal{H} =$

	1	2	3	4	5	6	7	8
1	0	1	1	0	0	0	0	0
2	0	0	0	1	0	0	0	0
3	0	1	0	0	1	0	0	0
4	0	0	0	0	1	1	1	1
5	0	0	0	0	0	0	0	0
6	0	0	1	0	0	0	0	0
7	0	1	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0

Para máis información ao respecto e ver outros exemplos pode consultarse [2].

1.2. Problemas de fluxo en redes con custo mínimo

Con estas bases sentadas definimos agora o concepto de **rede** como un **grafo** con un ou máis parámetros asociados a cada arco ou nó. As súas arestas representan interaccións entre os nós que unen. Estes números poden representar distancias, custos, fiabilidade ou outros parámetros de interese. Dentro dunha rede chamamos **fluxo** f ao envío de elementos ou obxectos dun lugar a outro da rede. Exemplos de redes poden ser:

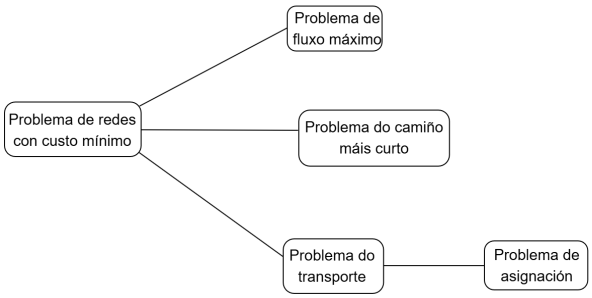

Figura 1.4: Relacións dentro da rama dos problemas de custo mínimo.

- Redes sociais: Os distintos nós representarían as distintas persoas e os arcos poderían ser as interaccións entre elas.
- Redes de transporte: Os nós poden identificar as distintas cidades que están conectadas por estradas representadas polos arcos.
- Redes eléctricas: Os nós son as centrais eléctricas e consumidores e as arestas representan as liñas de transmisión da enerxía.

Un problema de fluxo de custo mínimo é un problema no que se quere determinar cal debe ser o camiño no que o transporte de certos elementos xera un custo mínimo para cubrir certas demandas que teñen certos nós dende outros nós que teñen subministracións deses elementos. Por exemplo, para transportar unha cantidade x de mercadorías perecedoiras dende o porto de Shanghái ata o porto de Rótterdam, comandaranse a k barcos que terán a elección de atravesar a canle de Suez (aforrando tempo pero cun gasto adicional) ou rodear África a través do cabo de Boa Esperanza (aforrando o gasto de atravesar a canle de Suez pero aumentando o tempo de transporte, co risco de podremia da mercadoría). Outro exemplo pode ser destinar liñas de telefonía a través dun servidor.

Veremos neste apartado os seus casos máis relevantes (cuxa interconexión se pode ver na Figura 1.4) : o problema do camiño máis curto, o problema de fluxo máximo, o problema do transporte e o problema de asignación.

Así, podemos definir unha formulación matemática para este tipo de problemas. Sexa $G = (N, A)$ unha rede dirixida definida por un conxunto N de nós e un conxunto A de arcos dirixidos. Cada arco $(i, j) \in A$ ten asociadas tres cantidades:

- $l_{ij} \geq 0$ denota a **cota inferior** do fluxo que circula a través do arco (i, j) (a mínima cantidade que debe correr a través do arco).
- $u_{ij} \geq 0$ denota a **capacidade** ou **cota superior** do fluxo que circula a través do arco (i, j) (a cantidade máxima que pode fluír a través do arco).

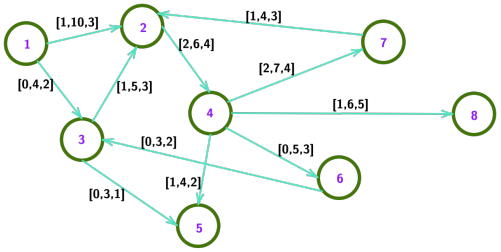

Figura 1.5: Exemplo dunha rede onde os arcos conteñen os datos $[l_{ij}, u_{ij}, c_{ij}]$.

- c_{ij} denota o **custo por unidade de fluxo** que atravesa dito arco. Se este custo é positivo entón denota o custo por unidade de fluxo que pasa a través do nó, e se é negativo indica o beneficio por unidade de fluxo que pasa a través do nó.

Agora ben, tanto estas cotas anteriores como os custos tamén poden estar referidas a nós, e se coñecen como **fluxos externos**. Así, para cada nó i podemos ter:

- l_i: se é un número positivo entón indica a cantidade mínima de fluxo que chega a dito nó, mentres que se é negativo indica a cantidade máxima de fluxo que sae dende o nó.
- u_i: se é positiva indica a cantidade máxima de fluxo que pode entrar na rede a través de dito nó, e se é negativa entón indica a mínima cantidade que ten que saír da rede a través do nó.
- c_i denota o **custo por unidade de fluxo** que pasa a través do nó i.

Sempre teremos que $l_i \leq u_i$, e para todo nó estas cantidades sempre serán non negativas ou non positivas. Ademais, se ocorre que $l_i = u_i$ entón teremos que o fluxo externo b_i (a cantidade de fluxo que entra ou sae da rede a través do nó i) é fixo.

Con todo isto establecido podemos formular o problema da seguinte maneira:
Minimizar

$$\sum_{(i,j)\in A} c_{ij} \cdot f_{ij} \tag{1.1}$$

suxeito a

$$\sum_{\{j:(i,j)\in A\}} f_{ij} - \sum_{\{j:(i,j)\in A\}} f_{ji} = b_i \qquad \forall i \in N \tag{1.2}$$

$$l_{ij} \leq f_{ij} \leq u_{ij} \qquad \forall (i,j) \in A \tag{1.3}$$

con $\sum_{i=1}^{n} b_i = 0$

Podemos tamén escribilo en modo matricial, utilizando a matriz de incidencia $\mathcal{N}$:

Minimizar

$$cx$$

suxeito a

$$\mathcal{N}x = b$$

$$l \leq x \leq u$$

As restricións (1.2) denomínanse **restricións de balance de masas**, xa que verifican que en cada nó i haxa soamente b_i unidades do elemento de transporte, pois suma os fluxos que lle chegan e réstanselle os fluxos que emanan del. Ademais, normalmente asúmese a que os valores para as capacidades dos arcos, os custos dos arcos e a demanda ou subministración de cada nó son números enteiros.

A continuación presentamos algúns problemas especiais dentro da categoría de problemas de fluxo con custo mínimo.

1.2.1. O problema do camiño máis curto

Este é o problema central do traballo. O seu obxectivo é buscar un percorrido de custo mínimo dende un nó denotado como *orixe* ou *procedencia* ata outro nó denotado como *destino*, onde cada arco da rede ten un custo asociado. Un exemplo pode ser un traballador que pola mañá ten que coller o coche dende a súa casa e conducir ata o centro de Santiago de Compostela para traballar: o obxectivo deste traballador será (polo xeral) elixir as rúas que fagan que o traxecto empregue o menor tempo posible.

A formulación do problema do camiño máis curto entre o nó fonte 1 e o nó sumideiro n é a seguinte:

Minimizar

$$\sum_{(i,j)\in A} c_{ij} \cdot f_{ij}$$

suxeito a:

$$\sum_{\{(j:(1,j)\in A\}} f_{1j} - \sum_{\{(j:(j,1)\in A\}} f_{j1} = 1 \tag{1.4}$$

$$\sum_{\{(j:(n,j)\in A\}} f_{ij} - \sum_{\{(j:(j,n)\in A\}} f_{ji} = -1 \tag{1.5}$$

$$\sum_{\{(j:(i,j)\in A\}} f_{ij} - \sum_{\{(j:(j,i)\in A\}} f_{ji} = 0 \quad \forall j \mid 1 < j < n \tag{1.6}$$

$$f_{ij} \in \{0, 1\}, \quad \forall (i, j) \in A$$

O obxectivo neste caso é minimizar o custo de ir dende o certo nó (ao que identificamos co 1) ata o nó n. Das restricións, a restrición (1.4) indica que no primeiro dos nós o fluxo saínte (o que xorde nese nó) é $+1$; a (1.5) indica que o fluxo sae polo nó n é tamén unha unidade (pero aparece negativo porque é fluxo saínte); e a (1.6) indica que o fluxo se mantén nos nós intermedios (é dicir, non hai nin ganancias nin perdas nos nós polos que pasa o camiño).

1.2.2. O problema de fluxo máximo

O obxectivo deste problema é enviar a máxima cantidade de fluxo entre dous nós (ata aquí é moi similar ao problema anterior), pero este problema non ten custos asociados cos arcos senón capacidades. Ademais, o seu obxectivo non é minimizar custos senón maximizar o fluxo que transcorre pola rede entre dous nós. Este problema preséntase, por exemplo, cando se quere estudar a capacidade dunha rede de estradas á hora de soportar tráfico.

Denotaremos por $F \in \mathbb{R}$ a variable que indica o fluxo máximo que queremos atopar, e suporemos que queremos que chegue dende o nó 1 ata o nó n. Así, a súa formulación matemática sería:

Maximizar

$$F$$

suxeito a:

$$\sum_{\{j:(1,j)\in A\}} f_{1j} - \sum_{\{j:(j,1)\in A\}} f_{j1} = F \tag{1.7}$$

$$\sum_{\{j:(i,j)\in A\}} f_{nj} - \sum_{\{j:(j,i)\in A\}} f_{jn} = -F \tag{1.8}$$

$$\sum_{\{j:(i,j)\in A\}} f_{ij} - \sum_{\{j:(j,i)\in A\}} f_{ji} = 0 \quad \forall j \mid 1 < j < n \tag{1.9}$$

$$0 \leq f_{ij} \leq u_{ij}, \quad \forall (i, j) \in A \tag{1.10}$$

Buscamos maximizar o fluxo restrinxido polas seguintes condicións: a ecuación (1.7) garante que todo o fluxo entra na rede a través do nó 1 (nó orixe); a ecuación (1.8) establece que todo o fluxo sae da rede no nó n (nó destino); a ecuación (1.9) impón que nos nós intermedios o fluxo

se conserva, é dicir, todo o fluxo que entra por un nó debe saír del; e finalmente, a ecuación (1.10) impón os límites de capacidade en cada arco da rede, é dicir, a cantidade de fluxo que pode circular entre dous nós está acotada por unha cota superior u_{ij}.

1.2.3. O problema de asignación

Este problema parte dun grafo consistente en dous conxuntos do mesmo tamaño de nós N_1 e N_2, unha colección de pares $A \subseteq N_1 \times N_2$ representando posibles asignacións e un custo c_{ij} asociado a cada elemento $(i,j) \in A$. O que este problema plantexa é emparellar co menor custo posible cada elemento de N_1 con exactamente un elemento de N_2. Este problema amósase cando por exemplo se queren asignar as habitacións aos distintos hóspedes aloxados nun hotel.

Sexa un grafo bipartito $G = (N, A)$, é dicir, existen conxuntos N_1 e N_2 tales que $N_1 \cup N_2 = N$, $N_1 \cap N_2 = \emptyset$ e $|N_1| = |N_2|$ e cada un dos arcos do conxunto A ten o seu nó de orixe no conxunto N_1 e o seu nó de destino no N_2 ou viceversa (isto é, non hai arcos que unan nós dun mesmo conxunto de nós). Así, a formulación deste problema é a seguinte:
Minimizar

$$\sum_{(i,j)\in A} c_{ij} \cdot f_{ij}$$

suxeito a:

$$\sum_{\{(j:(i,j)\in A\}} f_{ij} = 1, \quad \forall i \in N_1$$

$$\sum_{\{(i:(i,j)\in A\}} f_{ij} = 1, \quad \forall j \in N_2$$

$$0 \leq f_{ij} \leq u_{ij}, \quad \forall (i,j) \in A$$

Neste problema tamén queremos minimizar os custos, pero desta vez temos as restricións de que a cada nó de N_1 se lle asigna unicamente un nó de N_2 e viceversa (de aí vén o nome de *problema de asignación*).

1.2.4. O problema do transporte

Neste problema o conxunto de nós volve estar partido en dous N_1 e N_2 pero non teñen por que ter o mesmo número de elementos. Así, cada nó en N_1 será un nó de subministración e cada nó de N_2 será un nó de demanda. Os únicos arcos que existen unen un nó de subministracion cun nó de demanda. O seu obxectivo é satisfacer toda a demanda minimizando o custo de todos os

arcos necesarios para suplila. Como exemplo temos unha empresa que queira distribuír produtos dende diferentes almacéns e ata diversos centros de poboación.

Sexa de novo un grafo bipartito $G = (N, A)$ xunto cun vector de custos $\mathbf{c}$, un vector de subministros $\mathbf{s}$ e un vector de demandas $\mathbf{d}$ cun elemento para cada nó. Así, a formulación deste problema é a seguinte:
Minimizar

$$\sum_{(i,j)\in A} c_{ij} \cdot f_{ij}$$

suxeito a:

$$\sum_{\{(j:(i,j)\in A\}} f_{ij} \leq s_i, \quad \forall i \in N_1 \tag{1.11}$$

$$\sum_{\{(i:(i,j)\in A\}} f_{ij} = d_j, \quad \forall j \in N_2 \tag{1.12}$$

$$0 \leq f_{ij}, \quad \forall (i,j) \in A$$

Ademais, para que o problema teña solución, as demandas non poderán exceder os subministros:

$$\sum_{i\in N_1} s_i \geq \sum_{j\in N_2} d_j \tag{1.13}$$

As restricións (1.11) limitan a, como moito, s_i as unidades que poden saír dende o nó i do conxunto N_1. As restricións (1.12) indícanos que a cada nó de N_2 deben chegar as d_j unidades que demanda.

1.3. A propiedade de unimodularidade

É moi común tratar con fluxos indivisibles na teoría de grafos (vehículos, persoas...). Podemos entón engadir a restrición de que os fluxos tomen valores enteiros. Para poder estudar os problemas antes mencionados formalmente é preciso introducir o concepto de unimodularidade.

Definición 1.1. Unha matriz cadrada $\mathbf{A} \in \mathbb{Z}^{p\times p}$ é **unimodular** se o seu determinante é igual a 1 ou a -1.

Definición 1.2. Unha matriz $\mathbf{A} \in \mathbb{Z}^{p\times q}$ é **totalmente unimodular** se calquera submatriz cadrada é singular ou unimodular.

Damos a continuación algúns resultados que nos serán de utilidade:

Proposición 1.3. *Dados unha matriz totalmente unimodular $A \in \mathbb{Z}^{p\times q}$ e un vector $b \in \mathbb{Z}^p$, temos que calquera solución básica factible definida polas restricións $\mathbf{A}\cdot\mathbf{f} = \mathbf{b}$ con $\mathbf{f} \geq \mathbf{0}$ ten todas as súas compoñentes enteiras.*

Demostración. Podemos supoñer que as filas da matriz A son linearmente independentes. De non selo, podemos empregar a técnica de eliminación de Gauss para obter un sistema equivalente sen as filas redundantes.

Unha vez que supoñemos que as filas de $\mathbf{A}$ son linearmente independentes, temos que calquera solución básica factible $\mathbf{f} = (\mathbf{f}_B, \mathbf{f}_N)$ ten asociada unha submatriz regular $B_{p\times p}$ tal que $\det(B) \neq 0$, $\mathbf{f}_N = 0$ e $\mathbf{B}\cdot\mathbf{f}_B = \mathbf{b}$. Para cada compoñente i do vector $\mathbf{f}_B$ definamos $B^{\mathbf{b}-i}$ como a matriz que se obtén ao reemprazar a columna i de $\mathbf{B}$ co vector $\mathbf{b}$. Entón, empregando a regra de Cramer sabemos que cada compoñente de $\mathbf{f}_B$ será da forma

$$(\mathbf{f}_B)_i = \frac{\det(\mathbf{B}^{\mathbf{b}-i})}{\det(\mathbf{B})}$$

Como supoñemos que $\mathbf{A}$ e $\mathbf{b}$ son enteiros, $\det(\mathbf{B}^{\mathbf{b}-i})$ será un número enteiro. Ademais, como $\mathbf{A}$ é totalmente unimodular, $\det(\mathbf{B})$ será +1 ou -1, polo que $(\mathbf{f}_B)_i$ será un número enteiro. □

A proposición anterior asegúranos o seguinte resultado:

Corolario 1.4. *Dado un problema de programación linear en forma estándar ($\mathbf{A}\cdot\mathbf{f} = \mathbf{b}$, $\mathbf{f} \geq \mathbf{0}$), se a matriz de restricións é totalmente unimodular e o vector de lados dereitos é enteiro, entón toda solución factible (equivalentemente, todo punto que poida ser solución) ten todas as súas compoñentes enteiras. En particular, se o problema ten óptimos finitos, polo menos algún será enteiro.*

Proposición 1.5. *A matriz de incidencia $\mathcal{N}$ dun grafo dirixido é totalmente unimodular.*

Demostración. Temos que probar que o determinante de calquera submatriz cadrada $\mathbf{B}_{k\times k}$ de $\mathcal{N}$ é 0, +1 ou -1. Farémolo por indución en k. Tódolos elementos de $\mathcal{N}$ son 0, +1 ou -1, co que o resultado é trivial para $k = 1$. Supoñámolo certo para $k-1$ e tomemos unha submatriz $\mathbf{B}_{k\times k}$. Claramente, esta matriz $\mathbf{B}$ está nun dos tres casos seguintes:

1 Algunha columna de $\mathbf{B}$ está composta unicamente por ceros. Neste caso, directamente $\det(\mathbf{B}) = 0$.

2 Algunha columna de $\mathbf{B}$ soamente ten unha entrada distinta de cero. Supoñamos que se trata da columna j e da fila i. Denotemos por $\overline{\mathbf{B}}$ a matriz resultante de eliminar a columna j e a fila i de $\mathbf{B}$. Neste caso, $\det(\mathbf{B}) = (-1)^{i+j}\cdot B_{ij}\cdot\det(\overline{\mathbf{B}}) = \pm 1\cdot\det(\overline{\mathbf{B}})$ e, por indución, $\det(\overline{\mathbf{B}})$ é 0, +1 ou -1.

3 Toda columna de $\mathbf{B}$ ten exactamente dous elementos distintos de cero. Neste caso, un destes elementos ha de ser 1 e o outro -1. A suma de tódalas filas da matriz $\mathbf{B}$ é o vector cero, co que as filas de $\mathbf{B}$ son linearmente dependentes e temos que $\det(\mathbf{B}) = 0$.

Dado que en tódolos casos anteriores obtivemos que $\det(\mathbf{B})$ é 0, +1 ou -1 queda probado o resultado. □

Proposición 1.6. *Se todas as capacidades dun problema de fluxo con custo mínimo toman valores enteiros, entón en toda solución factible, todos os valores son enteiros. En particular, haberá polo menos un óptimo enteiro.*

Demostración. Lembremos a formulación dun problema de fluxo con custo mínimo como un problema de programación linear expresada nas ecuacións (1.1), (1.2) e (1.3). Podemos expresar estas ecuacións en notación vectorial e reformular o problema da seguinte maneira:

Minimizar

$$\mathbf{c}^T \cdot \mathbf{f}$$

suxeito a

$$\mathbf{N} \cdot \mathbf{f} = 0$$

$$\mathbf{I}_{m\times m} \cdot \mathbf{f} \leq \mathbf{u}$$

$$\mathbf{I}_{m\times m} \cdot \mathbf{f} \geq \mathbf{l}$$

sendo $\mathbf{I}$ a matriz identidade, indicando a orde no subíndice.

O que faremos agora será pasar o problema a forma estándar e ver que está nas condicións do corolario 1.4. Antes de nada, como $\mathbf{l} \geq \mathbf{0}$ as restricións $\mathbf{f} \geq \mathbf{0}$ da forma estándar son redundantes e engadilas non afecta á rexión factible (a rexión na que se atopan todos os puntos cumpren as restricións do noso problema). Cada restrición $f_i \leq u_i$ substituirémola por $f_i + s_i^u = u_i$, onde s_i^u é unha variable de holgura. Analogamente empregaremos variables de holgura $s_i^l \geq 0$ para transformar as restricións $f_i \geq l_i$ en $f_i - s_i^l = l_i$. Por tanto, o conxunto de restricións pódese expresar como $\mathbf{A} \cdot \mathbf{f} = \mathbf{b}$ onde $\mathbf{b} = (0, \dots, 0, u_1, \dots, u_m, l_1, \dots, l_m)$ e

$$\mathbf{A} = \begin{pmatrix} \mathcal{N} & \mathbf{0}_{n\times m} & \mathbf{0}_{n\times m} \\ \mathbf{I}_{m\times m} & \mathbf{I}_{m\times m} & \mathbf{0}_{m\times m} \\ \mathbf{I}_{m\times m} & \mathbf{0}_{m\times m} & -\mathbf{I}_{m\times m} \end{pmatrix} \tag{1.14}$$

onde $\mathbf{O}$ denota a matriz de dimensións indicadas no subíndice e cuxos elementos son todos nulos.

É sinxelo ver que $\mathbf{A}$ é totalmente unimodular se e soamente se $\mathcal{N}$ é totalmente unimodular. Tomemos unha submatriz cadrada $\mathbf{B}$ de $\mathbf{A}$. Podemos estar nalgún dos seguintes casos:

1 A matriz $\mathbf{B}$ ten unha ou máis columnas que proceden de columnas de $\mathbf{A}$ posteriores á columna m. Supoñamos que a columna j de $\mathbf{B}$ é unha desas columnas; dita columna terá como moito un elemento distinto de cero, B_{ij}, e $\det(\mathbf{B})$ será o produto de $(-1)^{i+j} \cdot B_{ij}$ polo determinante da submatriz $\overline{B}$ que resulta de eliminar de $\mathbf{B}$ a columna j e a fila i. Neste caso, o $\det(\mathbf{B})$ é 0, +1 ou -1 se e soamente se $\det(\overline{\mathbf{B}})$ é 0, +1 ou -1. Polo tanto, se somos capaces de probar que no resto de casos o determinante é 0, +1 ou -1, tamén o será neste caso.

2 Non estamos no caso 1 e $\mathbf{B}$ ten unha ou máis filas que proceden de filas de $\mathbf{A}$ posteriores á fila n. Supoñamos que a fila i de $\mathbf{B}$ é unha desas filas. Como non estamos no caso 1, dita fila terá como moito un elemento distinto de cero e podemos repetir o argumento do caso 1.

3 Non estamos nin no caso 1 nin no 2. Entón $\mathbf{B}$ é unha submatriz de $\mathcal{N}$ e, como $\mathcal{N}$ é totalmente unimodular, o $\det(\mathbf{B})$ é 0, +1 ou -1.

Polo tanto, podemos aplicar o corolario 1.4 á representación en forma estándar do problema de fluxo con custo mínimo de partida.

□

Podemos agora concluír esta sección introducindo unha noción fundamental para entender a utilidade práctica dos resultados anteriores.

Definición 1.7. Dado un problema de programación enteira, a súa **relaxación linear** consiste en eliminar a restrición de enteireza sobre as variables e considerar unicamente as restricións lineares, permitindo solucións reais.

Esta relaxación é moi útil desde o punto de vista computacional: os problemas de programación linear poden resolverse en tempo polinómico mediante métodos como o método do simplex ou os algoritmos de punto interior. Pola contra, os problemas de programación enteira son, en xeral, moito máis custosos de resolver.

Grazas aos resultados de unimodularidade que vimos nesta sección, sabemos que:

- A matriz de restricións dos problemas de fluxo con custo mínimo é totalmente unimodular.
- Se as capacidades son enteiras (o cal sucede en moitos casos prácticos), o vector de lados dereitos tamén o é.

En consecuencia, podemos aplicar o corolario anterior: **toda solución básica factible do problema relaxado é enteira**. Polo tanto, a relaxación linear coincide co problema enteiro, e **o problema de fluxo con custo mínimo pode resolverse eficientemente como un problema de programación linear**, sen necesidade de técnicas específicas de programación enteira. Esta propiedade é unha das razóns fundamentais polas que os problemas de fluxo son especialmente tractables dentro da optimización combinatoria.

Estas propiedades e algunhas das súas demostracións que aquí non incluímos pódense consultar en [6].

Capítulo 2

O problema do camiño máis curto e o algoritmo de Dijkstra

Neste capítulo indagaremos sobre o problema do camiño máis curto, exporemos as súas bases matemáticas e as redes sobre as que trata. Tamén estableceremos algúns algoritmos para a súa resolución e centrarémonos especialmente no algoritmo de Dijkstra. Explicaremos en que propiedades das redes se basea este algoritmo, como é o seu funcionamento e incluso mostraremos os seus puntos débiles e as melloras que introducen outros algoritmos para melloralo. Outro apartado do que falaremos é sobre a complexidade do algoritmo. Por último, incluiremos outros métodos máis rápidos para atopar a solución ao problema.

2.1. Notación e asuncións previas

Sexa unha rede dirixida $G = (N, A)$ onde cada arco $(i, j) \in A$ ten asociado unha lonxitude c_{ij} (o que definimos como custo no anterior capítulo). Esta rede ten un nó distinguido s chamado *orixe*. Sexa $A(i)$ a lista de adxacencia de arcos do nó i e sexa $C = \text{máx}\{c_{ij} \mid (i, j) \in A\}$. Definimos ahora a *lonxitude dun camiño dirixido* como a suma das lonxitudes dos arcos que percorre o camiño. O problema do camiño máis curto baséase en determinar, para cada nó $i \in N$ distinto da orixe, o camiño dirixido de menor lonxitude que une a orixe co nó i.

A formulación do problema é a seguinte:

Minimizar

$$\sum_{(i,j)\in A} c_{ij} \cdot f_{ij}$$

suxeito a

$$\sum_{(j|(i,j)\in A} f_{ij} - \sum_{(j|(j,i)\in A} f_{ji} = \begin{cases} n-1, & \text{si } i = s \\ -1, & \forall i \in N - \{s\} \end{cases}$$

$$f_{ij} \geq 0, \quad \forall (i,j) \in A$$

A restrición (2.1) anterior ten unha interpretación directa en termos de conservación de fluxo:

- Para o nó orixe s, esíxese que a cantidade total de fluxo que sae sexa igual a $n-1$, é dicir, queremos enviar un total de $n-1$ unidades de fluxo desde o nó s (unha unidade de fluxo cara a cada un dos outros $n-1$ nós do grafo).
- Para cada un dos restantes nós $i \in N \setminus \{s\}$, a ecuación obriga a que a diferenza entre o fluxo que sae e o que entra sexa exactamente -1. Isto implica que a ese nó chega unha única unidade de fluxo, e non sae ningunha (xa que só interesa chegar ata el, non continuar máis aló).

Este tipo de formulación forza a estrutura do fluxo para que se distribúa en forma de $n-1$ camiños disxuntos que parten desde o nó orixe e chegan a cada un dos demais nós unha única vez. O obxectivo (minimizar a suma dos custos ponderados polos fluxos) garante que eses camiños serán os máis curtos posibles.

No noso estudo sobre este problema asumiremos varias premisas:

1 Todas as lonxitudes dos arcos son números enteiros. Isto é necesario para algúns dos algoritmos de resolución, pero non para outros.

2 A rede contén un camiño dirixido entre o nó orixe s e todos os outros nós da rede.

3 A rede non contén ningún ciclo negativo (é dicir, non contén ningún ciclo dirixido de lonxitude total negativa). Isto é necesario, xa que no caso de que exista, o algoritmo enviará a través dese ciclo o fluxo infinitamente (lembremos que temos como obxectivo minimizar o custo).

4 A rede é dirixida.

Existen distintos tipos de problemas do camiño máis curto:

1 Encontrar os camiños máis curtos dende un nó ata todos os outros nós con lonxitudes de arcos non negativas.

2 Encontrar os camiños máis curtos dende un nó ata todos os outros nós con lonxitudes de arcos arbitrarias.

3 Encontrar os camiños máis curtos dende cada un dos nós ata cada un do resto dos nós.

4 Outras xeneralizacións do problema do camiño máis curto.

2.2. Algoritmos de etiquetado fixo e de etiquetado móbil

Normalmente os algoritmos para solucionar o problema do camiño máis curto divídense en dous grupos: algoritmos de etiquetado fixo (*label-setting*) e algoritmos de etiquetado móbil (*label-correcting*). Ambos tipos son iterativos. Asignan etiquetas de distancias provisionais a nós en cada paso, e estas etiquetas de distancia son estimacións das distancias do camiño máis curto. Os algoritmos de etiquetado fixo designan unha etiqueta como permanente (óptima) en cada iteración. Pola contra, os algoritmos de etiquetado móbil consideran todas as etiquetas como temporais ata o paso final, no que todas se volven permanentes. Os algoritmos de etiquetado fixo soamente son aplicables aos problemas de camiño máis curto definidos en redes sen ciclos con lonxitudes de arcos arbitrarias e en redes con lonxitudes de arcos non negativas. Os algoritmos de etiquetado móbil son máis xerais e poden aplicarse a toda clase de problemas do camiño máis curto, incluíndo aqueles cuxas redes teñen algún arco con lonxitude negativa. Aínda que os primeiros son más eficientes (é dicir, teñen mellores límites de complexidade no peor dos casos), os segundos, ademais de dar solución a unha maior variedade de problemas, ofrecen maior flexibilidade.

O algoritmo básico de etiquetado fixo é o *algoritmo de Dijkstra*. Primeiro exporemos unha implementación que ten unha complexidade temporal de $O(n^2)$ (sendo n o número de nós).

2.3. A árbore de camiños máis curtos

Nunha rede de $n-1$ nós, bástannos $n-1$ posicións de almacenamento para gardar os camiños máis curtos dende o nó orixe ao resto dos nós. Isto débese a que sempre podemos encontrar unha árbore dirixida que parte do nó orixe coa propiedade de que o único camiño dende a orixe a calquera outro nó é o camiño máis curto ata dito nó. Podemos entón establecer a seguinte propiedade:

Proposición 2.1. *Se o camiño $s = i_1 - i_2 - \cdots - i_h = k$ é un camiño máis curto dende o nó s ata o nó k, entón para cada índice $q \in \{2, 3, \ldots, h-1\}$ o camiño $s = i_1 - i_2 - \cdots - i_q$ é un camiño máis curto dende o nó orixe ata o nó i_q.*

A demostración deste resultado pode verse en [1].

Sexa $d(\cdot)$ unha variable que denota as distancias dos camiños máis curtos. A propiedade anterior implica que se P é un camiño máis curto dende o nó orixe ata o nó k, entón $d(j) = d(i) + c_{ij}$ para cada arco $(i,j) \in P$. O recíproco tamén é certo: se $d(j) = d(i) + c_{ij}$ para cada arco nun camiño dirixido P dende o nó orixe ata o nó k entón P debe ser un camiño máis curto. Para probar isto, sexa $s = i_1 - i_2 - \cdots - i_h = k$ unha secuencia de nós en P. Entón tense que:

$$d(k) = d(i_h) = (d(i_h) - d(i_{h-1})) + (d(i_{h-1}) - d(i_{h-2})) + \cdots + (d(i_2) - d(i_1)) \tag{2.1}$$

onde usamos que $d(i_1) = 0$. Temos que $d(j) - d(i) = c_{ij}$ para cada arco $(i,j) \in P$, polo que:

$$d(k) = c_{i_{h-1},i_h} + c_{i_{h-2},i_{h-1}} + \cdots + c_{i_1,i_2} = \sum_{(i,j)\in P} c_{ij} \tag{2.2}$$

Consecuentemente, P é un camiño dirixido do nó orixe ao nó k, e debe ser o camiño máis curto porque estamos supoñendo que $d(k)$ é a lonxitude do camiño máis curto entre ditos nós. Temos entón o seguinte resultado:

Proposición 2.2. *Sexa* $\mathbf{d}$ *un vector que representa as distancias do camiño máis curto. Entón un camiño dirixido* P *dende o nó orixe ata o nó* k *é un camiño máis curto se, e soamente se,* $d(j) = d(i) + c_{ij}$ *para cada arco* $(i,j) \in P$.

2.4. Problema do camiño máis curto en redes sen ciclos e condicións de optimalidade

Nesta sección mostraremos como resolver o problema do camiño máis curto nunha rede sen ciclos, mesmo se algúns arcos teñen lonxitude negativa, en tempo $O(m)$, sendo m o número de arcos da rede. Dado que calquera algoritmo para este problema debe examinar todos os arcos, o tempo mínimo posible será tamén $O(m)$.

Podemos empregar unha orde topolóxica sobre os nós dunha rede sen ciclos $G = (N, A)$ en tempo $O(m)$, de xeito que para todo arco $(i,j) \in A$ se cumpra $i < j$. Supoñamos que temos calculadas as distancias $d(i)$ do camiño máis curto dende o nó orixe ata os nós $i \in 2, 3, \ldots, k-1$, e consideremos o nó k. A orde topolóxica garante que todos os arcos que rematan en k parten de nós con índices menores. Pola Propiedade 2.1, o camiño máis curto ata k está formado polo camiño máis curto ata algún nó $i < k$ seguido do arco (i,k). Así, para calcular $d(k)$ abonda con tomar o mínimo de $d(i) + c_{ik}$ entre todos os arcos (i,k).

Para implementar isto, empregamos un algoritmo que propaga a información desde o nó orixe seguindo a orde topolóxica e usando a lista de adxacencia. Inicializamos $d(s) = 0$ e $d(j)$ a un

valor grande para $j \neq s$. Logo, percorremos os nós segundo a orde topolóxica, e para cada nó i, escaneamos os seus arcos saíntes $(i,j) \in A(i)$. Se $d(j) > d(i)+c_{ij}$, actualizamos $d(j) := d(i)+c_{ij}$. Unha vez examinados todos os nós, as etiquetas $d(j)$ serán óptimas.

Probamos por indución que, ao examinar un nó, a súa etiqueta de distancia é óptima. Supoñamos que os nós $1, 2, \ldots, k$ teñen etiquetas óptimas. Considérese o nó $k+1$ e sexa $s = i_1 - i_2 - \cdots - i_h - (k+1)$ un camiño máis curto ata el. Por Propiedade 2.1, o camiño ata i_h tamén é máis curto, e pola hipótese de indución $d(i_h)$ é correcto. Como $(i_h, k+1) \in A$, o algoritmo xa escaneou ese arco ao procesar i_h e actualizou $d(k+1) = d(i_h) + c_{i_h,k+1}$, que é a lonxitude do camiño máis curto. Así, cando se examina $k+1$, a súa etiqueta xa é óptima.

Teorema 2.3. *O algoritmo de propagación resolve o problema do camiño máis curto en redes sen ciclos en tempo $O(m)$.*

Como se tratou en seccións anteriores, os algoritmos de etiquetado móbil manteñen para cada nó $j \in N$ unha etiqueta $d(j)$, que representa unha cota superior da distancia ao nó j desde o nó orixe s (onde se fixa $d(s) = 0$). Ao final da execución, estas etiquetas coincidirán coas distancias mínimas reais.

Establecemos agora condicións necesarias e suficientes para que estas etiquetas representen distancias óptimas. Se as etiquetas son efectivamente distancias mínimas, entón deben satisfacer:

$$d(j) \leq d(i) + c_{ij} \qquad \forall (i,j) \in A \tag{2.3}$$

Estas desigualdades indican que non hai ningún arco (i,j) tal que $d(j) > d(i)+c_{ij}$, xa que nese caso o camiño pasando por i melloraría a estimación de $d(j)$, contradicindo a súa optimalidade.

Estas condicións tamén son suficientes: se as etiquetas $d(j)$ satisfacen (2.3) e representan a lonxitude dalgún camiño desde s ata j, entón ese camiño debe ser óptimo. Para velo, tomemos calquera camiño $P : s = i_1 - i_2 - \cdots - i_k = j$. As desigualdades (2.3) implican:

$$\begin{aligned} d(j) = d(i_k) &\leq d(i_{k-1}) + c_{i_{k-1},i_k} \\ d(i_{k-1}) &\leq d(i_{k-2}) + c_{i_{k-2},i_{k-1}} \\ &\vdots \\ d(i_2) &\leq d(i_1) + c_{i_1,i_2} = c_{i_1,i_2} \end{aligned}$$

Sumando todas estas desigualdades obtense:

$$d(j) \leq \sum_{(i,j)\in P} c_{ij}$$

Como $d(j)$ é a lonxitude dalgún camiño e tamén unha cota inferior para todos os demais, debe coincidir coa lonxitude mínima. Polo tanto:

Teorema 2.4 (Condicións de optimalidade do camiño máis curto)**.** *Para cada nó $j \in N$, sexa $d(j)$ a lonxitude dalgún camiño dirixido dende o nó orixe ata j. Entón $d(j)$ representa a distancia do camiño máis curto se e só se:*

$$d(j) \leq d(i) + c_{ij} \qquad \forall (i,j) \in A$$

Definimos agora a lonxitude reducida do arco (i,j) con respecto ás etiquetas $d(\cdot)$ como $c_{ij}^d = c_{ij} + d(i) - d(j)$. Isto leva ás seguintes propiedades:

Teorema 2.5. *a) Para todo ciclo dirixido W cúmprese:* $\sum_{(i,j)\in W} c_{ij}^d = \sum_{(i,j)\in W} c_{ij}$.

b) Para todo camiño dirixido P dende o nó k ata o nó l: $\sum_{(i,j)\in P} c_{ij}^d = \sum_{(i,j)\in P} c_{ij} + d(k) - d(l)$.

c) Se $d(\cdot)$ representa distancias mínimas, entón $c_{ij}^d \geq 0$ para todo $(i,j) \in A$.

Se a rede contén un ciclo negativo, entón ningún conxunto de etiquetas $d(\cdot)$ pode satisfacer as condicións (2.3). De feito, sexa W un ciclo dirixido. A propiedade c) garante que $\sum_{(i,j)\in W} c_{ij}^d \geq 0$, e pola a), esa suma é igual a $\sum_{(i,j)\in W} c_{ij}$, que entón tamén é non negativa. Polo tanto, W non pode ser negativo.

2.5. O algoritmo de Dijkstra

O algoritmo de Dijkstra encontra os camiños máis curtos dende o nó orixe s ata todos e cada un dos outros nós na rede con lonxitudes de arco non negativas. O algoritmo mantén a etiqueta de distancia $d(i)$ para cada nó i, sendo esta un límite superior da lonxitude do camiño máis curto ata o nó. En cada un dos pasos intermedios o algoritmo divide os nós en dous grupos: os nós con etiquetas permanentes (ou simplemente permanentes) e os nós con etiquetas temporais (ou simplemente temporais). A etiqueta de distancia a calquera nó permanente representa a distancia máis curta dende o nó orixe a ese nó. Para cada nó temporal, a etiqueta de distancia é un límite superior da distancia do camiño máis curto ata ese nó. A idea básica do algoritmo é comezar no nó s e etiqueta permanentemente nós en orde das súas distancias dende o nó orixe s. Inicialmente dámos ao nó s unha etiqueta permanente de cero, e a cada un dos outros nós unha

etiqueta temporal igual a ∞. En cada iteración, a etiqueta dun nó i é a súa distancia dende o nó orixe ao longo do camiño cuxos nós internos están etiquetados permanentemente. O algoritmo selecciona un nó i coa mínima etiqueta temporal, vólveo permanente e despois escanea os arcos en $A(i)$ para actualizar as etiquetas de distancia dos nós adxacentes. O algoritmo remata cando todos os nós están designados como permanentes. A exactitude do algoritmo reside en que sempre podemos designar o nó coa mínima etiqueta temporal como permanente.

O algoritmo de Dijkstra mantén unha estrutura de árbore T que parte do nó orixe que se estende aos nós con etiquetas de distancia infinitas. Ao remate, cando as etiquetas de distancia representan distancias de camiños máis curtos, T é unha árbore de camiño máis curto.

No algoritmo, referímonos á operación de seleccionar unha etiqueta de distancia temporal mínima como **operación de selección nó**. Tamén nos referimos á operación de comprobación se as etiquetas actuais dos nós i e j satisfán a condición $d(j) > d(i) + c_{ij}$ e, se é así, entón establecer $d(j) = d(i) + c_{ij}$ como **operación de actualización de distancia**.

Mostramos agora o pseudocódigo do algoritmo:

Algoritmo 1: Algoritmo de Dijkstra

Input: Un grafo $G = (N, A)$ con custos non negativos c_{ij}, e un nó orixe s

Output: As etiquetas de distancia $d(i)$ que representan a distancia mínima dende s a cada nó $i \in N$

```
1  for i ∈ N do
2      d(i) ← ∞
3      marcar i como temporal
4  d(s) ← 0
5  marcar s como permanente
6  while existe algún nó temporal i con d(i) < ∞ do
7      Escoller o nó temporal i con menor d(i)
8      Marcar i como permanente
9      for j tal que (i, j) ∈ A e j é temporal do
10         if d(j) > d(i) + c_ij then
11             d(j) ← d(i) + c_ij
```

Para facilitar o entendemento do algoritmo, a continuación ilustraremos o procedemento mediante o exemplo 2.6.

Exemplo 2.6. Supoñamos que temos a rede 2.1, e queremos obter cal é o camiño máis curto entre o nó 1 (que será o noso nó orixe) e o nó 9 (que será o noso nó final).

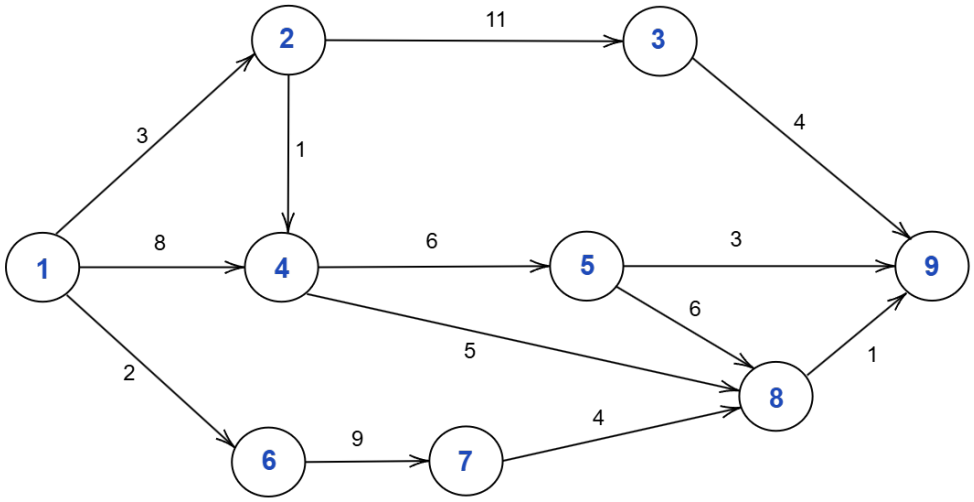

Figura 2.1: Unha rede onde os arcos conteñen os datos de custos c_{ij}.

O procedemento é o seguinte:

- Como comezamos no nó 1, o algoritmo asígnalle unha etiqueta permanente de 0. Como aínda non escaneou o resto dos nós, a estes asígnalle un etiqueta temporal igual a ∞.
- Na primeira iteración, o algoritmo escanea os arcos da lista de adxacencia de arcos do nó 1 (é dicir, escanea os arcos $\{(1,2),(1,4),(1,6)\}$) e actualiza a etiqueta temporal dos nós 2, 4 e 6, pasando de ser ∞ a ser o custo dos arco incidente en cada nó. Así, o nó 2 terá unha etiqueta igual a 3, o nó 4 terá unha etiqueta igual a 8 e o nó 6 terá unha etiqueta igual a 2.
- Na segunda iteración, o algoritmo escolle o nó con menor etiqueta e vólveo permanente. Neste caso, é o nó 6 cunha etiqueta de 2. A continuación, escanea os arcos da lista de adxacencia de arcos do nó etiquetado como permanente (no noso caso, o nó 6) e actualiza as etiquetas temporais dos nós cos que conecta o nó 6 (no noso caso, actualizaría o nó 7 e a súa etiqueta temporal pasaría a ser a suma dos arcos do camiño que van dende o nó orixe ata o nó 7: $d(6) + c_{67} = 2 + 9 = 11$).
- Na terceira iteración, o algoritmo escollerá o nó 2 (porque é o nó con menor etiqueta: 3) e vólveo permanente. Escanea a súa lista de adxacencia de arcos e actualiza o valor da etiqueta temporal do nó 3 de ∞ a 14.
- Na cuarta iteración, o algoritmo escolle o nó 4 (xa que ten etiqueta 8), vólveo permanente e escanea a súa lista de adxacencia de arcos e actualiza as etiquetas temporais do nó 5 ao valor 14 e do nó 8 ao valor 13.

- Na quinta iteración, o algoritmo escolle o nó 7 (con etiqueta 11), vólveo permanente e actualiza as etiquetas do nó 8, que toma o valor 13.
- Na sexta iteración, o algoritmo encontra que o nó con menor etiqueta é o 8 (con 13), vólveo permanente e entón actualiza a etiqueta do nó 9 a 14. Aínda que xa obtivemos un camiño ata o nó 9, como non todos os nós están etiquetados permanentemente, debemos seguir co algoritmo.
- Na séptima iteración, o algoritmo toma o nó 3 (o nó 5 ten a mesma etiqueta, pero asumimos que ante un empate escolle o nó de menor índice), vólveo permanente e actualizaría a etiqueta do nó 9. Porén, como a etiqueta temporal do nó 9 é menor que a que lle asignaríamos agora (a etiqueta que lle queremos asignar sería $3 + 11 + 4 = 18 > 14$), non actualizamos a súa etiqueta.
- Na oitava iteración, o algoritmo escolle o nó 5, convérteo en permanente e actualizaría a etiqueta dos nós 8 e 9. Como a nova etiqueta sería maior que a que xa teñen, non as actualizamos.
- Chegados a este punto, tódolos nós están etiquetados como permanentes menos o nó final. Como deste nó non emana ningún arco (ou polo menos non nos interesa, xa que este é o nó ao que queremos chegar), designámolo como permanente e terminamos o algoritmo. Obtemos así que o camiño máis curto dende o nó orixe ata o camiño 9 ten 14 unidades de distancia e este camiño é o $1 - 2 - 4 - 8 - 9$.

No Cadro 2.1, as filas denotan a iteración e as columnas denotan os nós do grafo 2.1. En verde claro están os nós que na seguinte iteración se tomarán como permanentes, en laranxa están os nós etiquetados como permanentes e en verde escuro está o valor final do camiño máis curto dende o nó orixe ata o nó 9.

	1	2	3	4	5	6	7	8	9
0	0	∞	∞	∞	∞	∞	∞	∞	∞
1	0	3	∞	8	∞	2	∞	∞	∞
2	0	3	∞	8	∞	2	11	∞	∞
3	0	3	14	8	∞	2	11	∞	∞
4	0	3	14	8	14	2	11	13	∞
5	0	3	14	8	14	2	11	13	∞
6	0	3	14	8	14	2	11	13	14
7	0	3	14	8	14	2	11	13	14
8	0	3	14	8	14	2	11	13	14

Cadro 2.1: Táboa que representa o algoritmo. As columnas representan os nós e as filas representan as iteracións do algoritmo.

2.5.1. Validez do algoritmo de Dijkstra

Para probar a validez do algoritmo de Dijkstra usaremos argumentos de indución. En cada iteración, o algoritmo parte os nós en dous sets S e $\overline{S}$ (tal que $S \cup \overline{S} = N$). As nosas hipóteses de indución son que a etiqueta de distancia de cada nó en S é óptima e que a etiqueta de distancia de cada nó en $\overline{S}$ é un camiño máis curto dende o nó orixe en base a que cada nó interno nese camiño se atopa no conxunto S. Usamos indución para a cardinalidade do conxunto S.

Para probar a primeira das hipóteses, lembremos que en cada iteración o algoritmo transfire un nó do conxunto $\overline{S}$ con etiqueta de distancia mínima ao conxunto S. Pola nosa hipótese de indución, $d(i)$ é a lonxitude dun camiño máis curto ata o nó i de entre todos os camiños que non teñen ningún nó do conxunto $\overline{S}$ como nó intermedio. Mostramos a continuación que a lonxitude de calquera camiño dende o nó orixe s ata o nó i que contén algún nó de $\overline{S}$ como nó intermedio será, canto menos, $d(i)$. Sexa calquera camiño P dende o nó orixe ata o nó i tal que contén polo menos un nó de $\overline{S}$ como nó intermedio, e dividamos ese camiño en dous segmentos P_1 (segmento que non contén ningún nó intermedio de $\overline{S}$ como nó intermedio pero si termina nun dese conxunto que denominaremos k) e P_2. Pola hipótese de indución, a lonxitude do camiño P_1 é canto menos $d(k)$, e dado que o nó i é o de etiqueta de distancia máis curta en $\overline{S}$ entón teremos $d(k) \geq d(i)$. Polo tanto, P_1 terá polo menos unha lonxitude $d(i)$. Ademais, dado que todas as lonxitudes de arcos son non negativas, a lonxitude do segmento P_2 será non negativa. En consecuencia, a lonxitude do camiño P é como mínimo $d(i)$ e esta será tamén a distancia do camiño máis curto dende o nó orixe ata o nó i.

A continuación probaremos a segunda das hipóteses. Despois de etiquetar permanentemente un novo nó i, as etiquetas de distancia dalgúns nós en $\overline{S} \setminus \{i\}$ poden diminuír, xa que o nó i pode ser agora o nó intermedio de camiños máis curtos entre o nó orixe e estes nós. Para representar o camiño máis curto actual dende o nó orixe ata calquera nó j, o algoritmo mantén tamén para cada nó un **nó predecesor** pred(j), que indica o último nó polo que pasa ese camiño antes de chegar a j. Pero recordemos que despois de etiquetar permanentemente o nó i o algoritmo examina cada arco $(i, j) \in A(i)$ e se $d(j) > d(i)+c_{ij}$ entón establece $d(j) = d(i)+c_{ij}$, e indica que o predecesor de j será i (pred(j)=i). Polo tanto, tras a operación de actualización de distancias, pola hipótese de indución o camiño dende o nó j ata o nó orixe definido polos índices dos nós predecesores satisfai a Propiedade 2.2 e a etiqueta de distancia de cada nó no conxunto $\overline{S} \setminus \{i\}$ é a lonxitude dun camiño máis curto sometido á restrición de que cada nó intermedio de dito camiño debe estar no conxunto $S \cup \{i\}$.

Pódese consultar información máis detallada en [5] e en [1].

2.5.2. Tempo de execución do algoritmo de Dijkstra

Estimamos agora a complexidade do algoritmo de Dijkstra no peor dos casos. O algoritmo emprega dúas operacións básicas:

- Selección de nós: O algoritmo emprega esta operación n veces, e de cada vez escanea cada un dos nós temporais. Polo tanto, o tempo total desta operación ao longo de todo o algoritmo será: $n + (n-1) + (n-2) + (n-3) + \cdots + 1 = O(n^2)$.
- Actualización das distancias: Para o nó i, o algoritmo emprega esta operación $|A(i)|$ veces (é dicir, tantos como arcos parten dese nó). O algoritmo empregará esta operación $\sum_{i \in N} |A(i)|$ veces en total. Como cada actualización require un tempo $O(1)$, o tempo total para esta operación será $O(m)$.

Establecemos así o seguinte resultado:

Teorema 2.7. *O algoritmo de Dijkstra emprega $O(n^2)$ tempo para resolver o problema do camiño máis curto.*

Esta cota é a óptima para redes completamente densas (é dicir, aquelas nas que todos os nós están relacionados con todos os outros). Ao longo das décadas, os investigadores tentaron reducir o tempo da selección de nós sen incrementar moito o tempo para a actualización das distancias. Estas implementacións ou ben reduciron considerablemente o tempo de execución do algoritmo ou ben melloraron a súa complexidade no peor dos casos.

2.5.3. Algoritmo de Dijkstra invertido

O obxectivo deste algoritmo é determinar o camiño máis curto dende todos os nós en $N \setminus \{t\}$ a un nó final t. Este algoritmo mantén unha distancia $d'(j)$ para cada nó j, sendo esta un límite superior da distancia do camiño máis curto dende o nó j ata o nó t. Como no algoritmo de Dijkstra, este algoritmo designa un conxunto S' de nós como permanentes e o conxunto complementario de nós $\overline{S'}$ como temporais. En cada iteración designa como permanente o nó temporal coa mínima etiqueta de distancia. Tras isto, examina cada arco (i, j) e modifica a etiqueta de distancia ao valor mín$\{d'(i), c_{ij} + d'(j)\}$. O algoritmo finaliza cando todos os nós se volven permanentes.

2.5.4. Algoritmo de Dijkstra bidireccional

Supoñamos que queremos determinar un camiño máis curto dende o nó s ata outro nó específico t. Poderíamos executar o algoritmo de Dijkstra e parar en canto designe o nó t como

permanente. Pois ben, o algoritmo de Dijkstra bidireccional permítenos resolver este problema aínda máis rápido.

Neste algoritmo, simultaneamente aplicamos o algoritmo de Dijkstra dende o nó s e o algoritmo de Dijkstra invertido dende o nó t. Cada algoritmo vai convirtindo un nó en permanente alternativamente ata que ambos denotan o mesmo nó como permanente (visualmente é como se se unisen os camiños, entón xa paran de buscar). Sexa $P(i)$ o camiño máis curto dende o nó s ata o nó $i \in S$ encontrado polo algoritmo de Dijkstra e sexa $P'(j)$ o camiño máis curto dende o nó $j \in S'$ ata o nó t atopado polo algoritmo de Dijkstra invertido. Entón o camiño máis curto dende o nó s ata o nó t é ou ben o camiño $P(k) \cup P'(k)$ ou ben o camiño $P(i) \cup \{(i,j)\} \cup P'(j)$ para algún arco (i,j) con $i \in S$ e $j \in S'$.

2.5.5. Implementación de melloras

A continuación presentamos algunhas melloras que poden ser aplicadas ao algoritmo de Dijkstra co obxectivo de diminuír o seu tempo de execución no peor dos casos.

2.5.5.1. A implementación de Dial

O maior obstáculo do algoritmo de Dijkstra é a operación de selección de nós. Para mellorar este aspecto, temos que pensar en reducir o tempo de execución pero mantendo as distancias ordenadas dalgunha maneira. A implementación de Dial utiliza o seguinte feito:

Propiedade 2.8. *As etiquetas de distancia que o algoritmo de Dijkstra designa como permanentes son non decrecentes.*

Esta propiedade segue de que o algoritmo sempre etiqueta como permanente o nó i coa menor etiqueta temporal $d(i)$, e mentres se examinan os arcos en $A(i)$ durante a operación de actualización das distancias, nunca diminúe a etiqueta de distancia de ningún nó etiquetado como temporal por debaixo de $d(i)$ porque as distancias son non negativas.

Introducimos a cantidade C, que representa a maior lonxitude de arco de toda a rede, é dicir, $C = \max\{c_{ij} \mid (i,j) \in A\}$. O algoritmo de Dijkstra garda nós con etiquetas temporais finitas de certa maneira. Mantén $nC + 1$ conxuntos (chamados *caldeiros*) numerados do 0 ata o nC. O caldeiro k garda todos os nós con etiquetas temporais de distancia iguais a k. Ademais, non precisamos gardar nós con etiquetas de distancia temporais iguais a infinito en ningún caldeiro (podemos engadilos a un caldeiro en canto as súas etiquetas de distancia se actualicen a un valor finito). Representamos o contido do caldeiro k como *contido*(k).

Na operación de selección de nós examinamos os caldeiros de forma ordenada ata atopar o

primeiro non baleiro. Sexa k este caldeiro. Entón cada nó nese caldeiro ten a etiqueta de mínima distancia. Un por un, elimina eses nós do caldeiro, márcaos como permanentes e escanea as súas listas de adxacencia de arcos para actualizar as etiquetas de distancia dos nós adxacentes e movemos o nó i ao caldeiro cuxo índice é a nova distancia atopada para dito nó. Na seguinte operación de selección de nó continuamos escaneando os caldeiros numerados $k+1$, $k+2$, ... para seleccionar o seguinte caldeiro non baleiro. A Propiedade 2.8 implica que os caldeiros de índices $0, 1, 2, \ldots, k$ estarán sempre baleiros nas seguintes iteracións, e o algoritmo xa non os volve examinar.

Debido ás estruturas de datos para almacenar o contido dos caldeiros, a implementación de Dial do algoritmo de Dijkstra ten un tempo de execución pseudopolinómico $O(m + nC)$.

Unha análise máis completa pode atoparse en [1] e en [4].

2.5.5.2. A implementación de *heap data structures*

Esta implementación emprega *heap data structures*, que son unha estrutura de datos que permite facer operacións co seu contido onde a cada elemento do seu contido se lle asigna unha clave. No noso caso, o contido estaría formado por nós con etiquetas de distancia finitas e a clave de cada nó sería a súa etiqueta de distancia. Esta implementación realiza as operacións de buscar a mínima clave, eliminar o nó con clave mínima e engadir un novo nó como moito n veces, e a operación de diminuír a clave como moito m veces.

Hai diversos tipos de *data heaps*, e a continuación mostramos algúns como exemplo:

- **Implementación de d-heap**: Baséase en que a lista de adxacencia de arcos de cada nó ten, como máximo, d nós. Este tipo de estrutura de datos permite alcanzar unha complexidade de $O(m \cdot \log_d n + n \cdot d \cdot \log_d n)$. Esta cota é peor cá cota orixinal de Dijkstra para redes completamente densas (aquelas nas que todo nó está unido mediante un arco con todos os outros nós), pero é moito mellor cando $m = O\left(\frac{n^2}{\log n}\right)$.

- **Implementación de heap de Fibonacci**: Mellora o tempo de operación de buscar mínimo e inserir un novo nó, polo que a complexidade deste algoritmo sitúase en $O(m+n\cdot\log n)$. Esta implementación é actualmente o mellor algoritmo de tempo estrictamente polinómico para resolver o problema do camiño máis curto.

- **Implementación de heap de Johnson**: Esta implementación soamente se pode empregar cando todas as lonxitudes de arcos son enteiras. A súa complexidade é de $O(m \log\log C)$.

2.5.5.3. A implementación de radix heap

Esta implementación está a metade de camiño entre o algoritmo básico de Dijkstra (que podemos asumir que emprega un só caldeiro no que garda todos os nós con etiquetas temporais e ten que buscar nel o nó con menor etiqueta) e a implementación de Dial (que emprega $nC+1$ caldeiros), xa que emprega caldeiros como o segundo pero reduce moito a cantidade de caldeiros empregados (nesta implementación, denominaremos $K+1$ á cantidade de caldeiros, pois o primeiro caldeiro terá o índice 0 e o último terá o índice K).

En cada caldeiro i gárdanse nós con etiquetas temporais entre 2^{i-1} e 2^i-1 (que denominaremos como **rango de** i). Así, a cantidade de caldeiros empregados é $O(\log(nC))$. O conxunto de nós dun caldeiro denominarémolo como contido dese caldeiro. Outra propiedade importante é que a medida que avanzamos nas iteracións do algoritmo, imos redefinindo os rangos de cada caldeiro, é dicir, que se na iteración k despois da elección do nó con menor etiqueta de distancia e a actualización das etiquetas do resto dos nós, o nó con etiqueta de distancia mínima j terá $d(j)$ como etiqueta, e entón a todos os nós se lles restará esa cantidade das súas etiquetas de distancia, facendo que se despracen a caldeiros de índices inferiores.

Analicemos agora a complexidade deste algoritmo.

- Supoñamos que o nó $j \in \text{contido}(k)$ e que estamos reasignando o nó j a un caldeiro de menor índice (ben sexa porque $d(j)$ diminúe ou porque estamos redistribuíndo os nós a caldeiros de etiquetas inferiores). Se $d(j) \notin \text{rango}(k)$ escaneamos secuencialmente caldeiros con índices menores (dende o maior índice ata o menor) e engadimos o nó apropiado ao caldeiro de índice correspondente. Esta operación require $O(m+n\cdot K)$ tempo. O termo m reflicte o número de actualizacións de distancia, e o nK polos movementos a caldeiros de índices inferiores (como hai K caldeiros, o número máximo de desprazamentos para un nó será K).

- A continuación, cosideremos a operación de selección de nós. A selección de nós comeza escaneando os caldeiros de menor a maior índice para identificar o primeiro caldeiro non baleiro (asumimos que este é o caldeiro k). Esta operación require un tempo $O(K)$ por cada iteración, e $O(nK)$ tempo en total.

- Por último, sexa a operación de redistribución de nós nos caldeiros (despois de actualizar as distancias). Asignamos o primeiro enteiro ao caldeiro 0, o segundo enteiro ao caldeiro 1, os seguintes dous enteiros ao caldeiro 2, os seguintes catro enteiros ao caldeiro 3 etc. Como o caldeiro k ten menos de 2^{k-1} nós, e dado que os caldeiros de índices inferiores conteñen como moito 2^{k-1} posicións de almacenamento, podemos redistribuír o rango do caldeiro k nos caldeiros de índices $0,\ldots,k-1$ na maneira arriba descrita. Esta redistribución de

rangos e a subsecuente reinserción de nós baleira o caldeiro k e move os nós coa menor etiqueta de distancia ao caldeiro 0. Esta redistribución require tempo $O(K)$ por iteración para un total de $O(n \cdot K)$ en todas as iteracións.

Consecuentemente, o tempo que emprega esta implementación para resolver o problema do camiño máis curto é de $O(m + n \cdot K)$. Dado que $K = \lceil \log nC \rceil$, o tempo empregado é $O(m + n \cdot \log nC)$. Combinando un heap de Fibonacci con esta implementación mellorariamos o tempo a $O(m + n \cdot \sqrt{\log C})$.

O pseudocódigo desta implementación preséntase a continuación:

Algoritmo 2: Dijkstra con implementación de Radix Heap

Input: Grafo $G = (N, A)$ con custos non negativos c_{ij}, nó orixe s, $K + 1$ caldeiros
Output: Etiquetas de distancia $d(i)$ para cada nó $i \in N$

```
for i ∈ N do
    d(i) ← ∞
    inserir i no caldeiro K          // todos os nós comezan no caldeiro máis alto
d(s) ← 0
marcar s como permanente
mover s ao caldeiro 0
while existen nós con etiqueta temporal do
    // Seleccionar o primeiro caldeiro non baleiro
    for k ← 0 to K do
        if o caldeiro k non está baleiro then
            escoller o nó i no caldeiro k con menor etiqueta d(i)
            break
    marcar i como permanente
    r ← d(i)                         // nova referencia para reescalar etiquetas
    for cada nó temporal j en todos os caldeiros do
        actualizar etiqueta: d(j) ← d(j) − r
        reasignar j ao caldeiro correspondente ao intervalo de d(j)
    for cada arco (i, j) ∈ A do
        if j é temporal e d(j) > d(i) + c_ij then
            actualizar d(j) ← d(i) + c_ij
            mover j ao caldeiro correspondente ao rango de d(j)
```

Para ilustrar este algoritmo exemplificarémolo no Exemplo 2.9 empregando a rede 2.1.

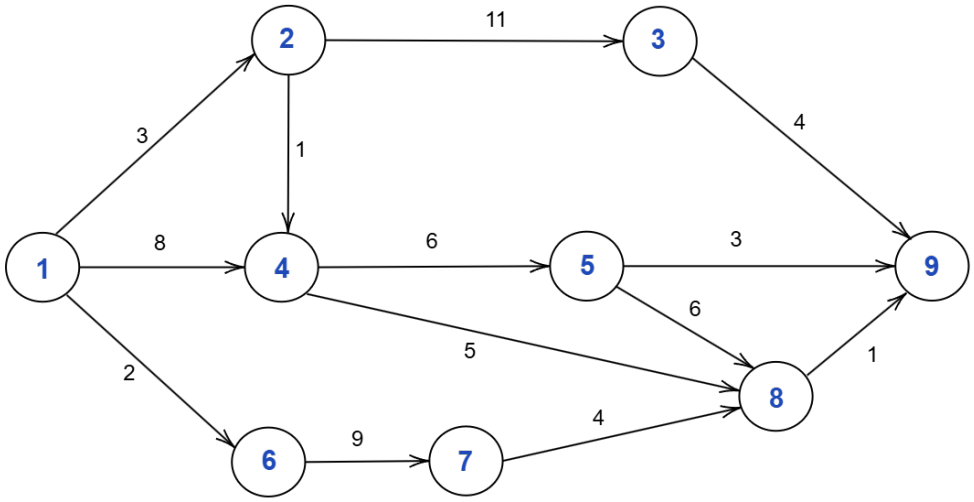

Figura 2.2: Unha rede onde os arcos conteñen os datos de custos c_{ij}.

Exemplo 2.9. Supoñamos que temos a rede 2.1 e queremos calcular o camiño máis curto entre o nó 1 e o nó 9 empregando a implementación de *radix heap* sobre o algoritmo de Dijkstra.

Os pasos a seguir por este algoritmo son:

- Comezamos creando os caldeiros con $i = 0, \ldots, k$ (neste caso escolleremos $k = 5$ porque será suficiente). Determinamos o nó 1 como permanente. Ao resto de nós asignámoslle unha etiqueta temporal de ∞.
- A continuación, na primeira iteración, vemos con que nós conecta o nó 1, que neste caso son os nós 2, 4 e 6. O nó 2 está a unha distancia de $3 = 2^1 + 1$ (e esta será a súa nova etiqueta temporal), polo que engadiremos o nó 2 ao caldeiro 2. O nó 4 está a unha distancia de $8 = 2^3$ (a súa nova etiqueta temporal), polo que o engadiremos ao caldeiro 4. O nó 6 está a unha distancia de $2 = 2^1$, polo que o engadiremos ao caldeiro 2. Neste punto, os caldeiros 0, 1, 3 e 5 están baleiros. Tras isto vemos cal é o primeiro caldeiro non baleiro (neste caso é o 2), e dentro dese caldeiro miramos cal é o no que ten a menor etiqueta de distancia e cal é esta etiqueta (no noso caso sería o nó 6 cunha etiqueta de distancia de 2). O que fai o algoritmo a continuación é marcalo como permanente e restar a súa etiqueta temporal a tódalas etiquetas temporais dos nós con etiquetas temporais finitas (é dicir, actualiza as etiquetas de tódolos nós temporais con etiqueta finita), de modo que o nó 6 pasa a ter etiqueta temporal $2^1 - 2 = 0$ e a estar no caldeiro 0; o nó 2 pasa a ter etiqueta temporal $2^1 + 1 - 2 = 1$ e a estar no caldeiro 1 e o nó 4 pasa a ter etiqueta temporal $2^3 - 2 = 2^2 + 2 = 6$ estar no caldeiro 3.

0	1	2	3	4	5	Valor substraído das etiquetas
{6}	{2}	∅	{4}	∅	∅	-2

Cadro 2.2: Contido de cada caldeiro e valor restado ao final da primeira iteración.

- A partir da segunda iteración o nó que escolleremos sempre estará no caldeiro 0 (debido a que imos restando a menor das etiquetas finitas dos nós temporais). Nesta iteración, o nó escollido é o nó 6. O algoritmo denota este nó como permanente e escanea os novos nós aos que chega a partir deste nó. Neste caso, chega ata o nó 7, polo que asigna á etiqueta temporal deste nó a distancia do arco $(6,7)$ e engade o nó 7 ao caldeiro 4 por ter unha etiqueta temporal de distancia igual a $9 = 2^3 + 1$. Agora busca cal é a menor etiqueta de distancia temporal finita e resta este valor ao resto de etiquetas temporais finitas (é dicir, actualiza as etiquetas), sendo neste caso o nó 2 con etiqueta 1. Unha vez feita esta resta, obtén que as etiquetas temporais dos nós temporais 2, 4 e 7 son, respectivamente, 0, $6-1 = 5 = 2^2+1$ e $9-1 = 8 = 2^3$, polo que estarán nos caldeiros 0, 3 e 4 respectivamente.

0	1	2	3	4	5	Valor substraído das etiquetas
{2}	∅	∅	{4}	{7}	∅	-1

Cadro 2.3: Contido de cada caldeiro e valor restado ao final da segunda iteración.

- Na terceira interación, escollemos o nó 2, denotámolo como permanente e vemos que a partir del podemos chegar ao nó 3, cunha etiqueta temporal finita de 11. Agora o nó con menor etiqueta temporal é o nó 4, que ten 5 como etiqueta temporal. Restamos entón esa cantidade a todas as etiquetas finitas de tódolos nós temporais (actualizando así as etiquetas temporais) e as novas etiquetas temporais dos nós 4, 7 e 3 serán $5-5 = 0$, $8-5 = 3 = 2^1+1$ e $11-5 = 6 = 2^2+2$ respectivamente, polo que os asignamos aos caldeiros 0, 2 e 3 respectivamente.

0	1	2	3	4	5	Valor substraído das etiquetas
{4}	∅	{7}	{3}	∅	∅	-5

Cadro 2.4: Contido de cada caldeiro e valor restado ao final da terceira iteración.

- Na cuarta interación, facemos permanente o nó 4 e a través del podemos chegar ata os nós 5 e 8, cuxas etiquetas temporais son 6 e 5 respectivamente. Entre os nós temporais 3, 7,

5 e 8, o de menor etiqueta temporal é o 7, cunha etiqueta de 3 unidades. Tras restar esta cantidade a todas as etiquetas temporais finitas, obtemos que para os nós 3, 7, 5 e 8 son, respectivamente, $6-3=3=2^1+1$, $3-3=0$, $6-3=3=2^1+1$ e $4-3=1=2^0$, polo que pertencerán aos caldeiros 2, 0, 2 e 1 respectivamente.

0	1	2	3	4	5	Valor substraído das etiquetas
$\{7\}$	$\{8\}$	$\{3,5\}$	$\emptyset$	$\emptyset$	$\emptyset$	-3

Cadro 2.5: Contido de cada caldeiro e valor restado ao final da cuarta iteración.

- Na quinta interación facemos o nó 7 permanente, e a través del podemos alcanzar o nó 8. Porén, como este nó xa está nalgún caldeiro non temos que volver engadilo. De entre os nós temporais 3, 5 e 8 o que ten menor etiqueta temporal é o nó 8 con 1. Restamos este valor a todas as etiquetas temporais e obtemos as novas etiquetas $3-1=2^1$,$3-1=2^1$ e $1-1=0$ respectivamente, e asignámolos aos caldeiros 2, 2 e 0 respectivamente.

0	1	2	3	4	5	Valor substraído das etiquetas
$\{8\}$	$\{9\}$	$\{3,5\}$	$\emptyset$	$\emptyset$	$\emptyset$	-1

Cadro 2.6: Contido de cada caldeiro e valor restado ao final da quinta iteración.

- Na sexta iteración, tornamos o nó 8 como permanente e engadimos o nó 9 (que se pode alcanzar agora que fixemos o 8 permanente), cunha etiqueta temporal de 1. Polo tanto, restamos este valor a todas as etiquetas temporais dos nós 9, 3 e 5, obtendo $1-1=0$, $2-1=1=2^0$ e $2-1=1=2^0$, e situámolos nos cadeiros 0, 1 e 1 respectivamente.

0	1	2	3	4	5	Valor substraído das etiquetas
$\{9\}$	$\{3,5\}$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	-1

Cadro 2.7: Contido de cada caldeiro e valor restado ao final da sexta iteración.

- Na séptima iteración facemos o no 9 permanente e como non se alcanzan novos nós a partir deste non engadimos nada aos caldeiros. A etiqueta de distancia mínima é 1 tanto para o nó 3 como para o nó 5 (que son os dous que quedan nos caldeiros), polo que ambos pasan a ter a etiqueta $1-1=0$ e a estar no caldeiro 0.

0	1	2	3	4	5	Valor substraído das etiquetas
$\{3,5\}$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	-1

Cadro 2.8: Contido de cada caldeiro e valor restado ao final da sétima iteración.

- Na oitava iteración denotamos o nó 3 como permanente (eliximos o 3 porque é o de menor índice de entre os pertencentes ao caldeiro 0) e non engadimos ningún novo nó a ningún caldeiro. Como a súa etiqueta de distancia é 0, non precisamos restar nada a ningunha outra etiqueta temporal.

0	1	2	3	4	5	Valor substraído das etiquetas
$\{5\}$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	0

Cadro 2.9: Contido de cada caldeiro e valor restado ao final da oitava iteración.

- Na novena iteración, facemos o nó 5 como permanente, e non engadimos novos nós aos caldeiros. Como non queda ningún outro nó nos caldeiros, o algoritmo remata.

Obtemos así que a distancia do camiño máis curto dende o nó 1 ata o nó 9 é de 14 unidades e que este é o camiño $1-2-4-8-9$.

No Cadro 2.10 mostramos os caldeiros utilizados no exemplo e cal sería o rango de cada un deles.

Caldeiro	0	1	2	3	4
Rango	[0]	[1]	[2, 3]	[4, 7]	[8, 15]

Cadro 2.10: Caldeiros e os seus rangos que se empregan no Exemplo 2.9.

2.6. Algoritmos de etiquetado móbil

Nesta sección estudaremos situacións máis xerais: algoritmos que ou ben identifican un ciclo negativo cando este existe ou que resolven o problema do camiño máis curto cando a rede non contén ningún. Como xa vimos, todos os algoritmos para resolver o problema do camiño máis curto baséanse no emprego de etiquetas de distancia. En calquera punto da execución do algoritmo asociamos unha etiqueta de distancia a cada nó. Se a etiqueta é infinita, aínda

estamos por atopar un camiño unindo o nó orixe e dito nó. Se a etiqueta é finita, entón a etiqueta representa a distancia entre o nó orixe e dito nó a través dalgún camiño que os una.

Comezamos este capítulo definindo as condicións óptimas que nos permiten avaliar cando un conxunto de etiquetas de distancia son óptimas. Estas condicións ofrécennos un criterio de finalización ou un certificado de optimalidade, para saber cando unha solución factible ao noso problema é óptima, polo que non precisaríamos continuar computando. O concepto de condicións de optimalidade é un tema clave no campo da optimización, e será recurrente no noso tratamento das redes.

2.6.1. Algoritmos de etiquetado móbil xenéricos

Nesta sección asumimos que a rede non contén ningún ciclo negativo. O algoritmo xenérico de etiquetado móbil mantén un conxunto de etiquetas de distancia $d(\cdot)$ en cada etapa. A etiqueta $d(j)$ é ou ben ∞ (o que indica que aínda está por descubrir un camiño dirixido dende o nó orixe ata o nó j) ou ben é a distancia dalgún camiño dirixido dende o nó orixe ata o nó j. Para cada nó j tamén mantemos un índice de predecesor, $\text{pred}(j)$, o cal garda o nó anterior ao nó j no actual camiño dirixido de lonxitude $d(j)$. Ao remate, os índices predecesores permítennos trazar o camiño máis curto dende o nó orixe ata o nó j. O algoritmo de etiquetado móbil xenérico é un procedemento xeral para sucesivas actualizacións das etiquetas de distancia ata que satisfán as condicións (2.3).

Por definición de custos reducidos, as etiquetas de distancia $d(\cdot)$ satisfán as condicións de optimalidade se $c_{ij}^d \geq 0$ para todo arco $(i,j) \in A$. O algoritmo de etiquetado móbil xenérico selecciona un arco (i,j) que viola a súa condición de optimalidade (é dicir, cumpre $c_{ij}^d < 0$) e emprégao para actualizar a eiqueta de distancia do nó j. Esta operación diminúe a etiqueta de distancia do nó j e iguala a lonxitude de arco reducida do arco (i,j) a cero.

Como xa adiantamos, o algoritmo mantén un índice de predecesor para cada nó con etiqueta de distancia finita. Referímonos á colección de arcos $(\text{pred}(j), j)$ como o **grafo de predecesores**. Este grafo é unha árbore dirixida con raíz no nó orixe e que se estende a todos os nós con etiquetas de distancia finitas. En cada actualización de distancia usando o arco (i,j) xérase un novo grafo predecesor eliminando o arco $(\text{pred}(j), j)$ e engadindo o arco (i,j).

O algoritmo de etiquetado móbil satisfai a propiedade invariante de que para cada arco (i,j) no grafo predecesor tense que $c_{ij}^d \leq 0$. Establecemos este resultado empregando inducción no número de iteracións. É importante decatarse de que o algoritmo engade un arco (i,j) ao grafo de predecesores durante a operación de actualización de distancias, o que implica que despois desta actualización $d(j) = d(i) + c_{ij}$ ou $c_{ij} + d(i) - d(j) = c_{ij}^d = 0$. Nas seguintes iteracións $d(i)$ pode decrecer, polo que c_{ij}^d pode volverse negativo. Observamos logo que se $d(j)$ decrece durante

o algoritmo, entón para algún arco (i, j) no grafo de predecesores c_{ij}^d pode volverse positivo, o que contradiciría a propiedade invariante. Pero nótese que nese caso eliminamos inmediatamente o arco (i, j) do grafo e mantense a propiedade invariante.

O pseudocódigo deste algoritmo é o seguinte:

Algoritmo 3: Algoritmo de etiquetado móbil xenérico

Input: Grafo dirixido $G = (N, A)$ con custos c_{ij}, nó orixe $s \in N$

Output: Etiquetas de distancia $d(j)$ e predecesores pred(j) para todo $j \in N$

```
1  for j ∈ N do
2      d(j) ← ∞;
3      pred(j) ← indefinido;
4  d(s) ← 0;
5  L ← {s};                          // Inicializar a lista de nós activos
6  while L ≠ ∅ do
7      Extraer un nó i de L;
8      for (i, j) ∈ A do
9          if d(i) + c_ij < d(j) then
10             d(j) ← d(i) + c_ij;
11             pred(j) ← i;
12             if j ∉ L then
13                 Engadir j a L;
```

A continuación mostramos o Exemplo 2.10 para ilustrar este algoritmo.

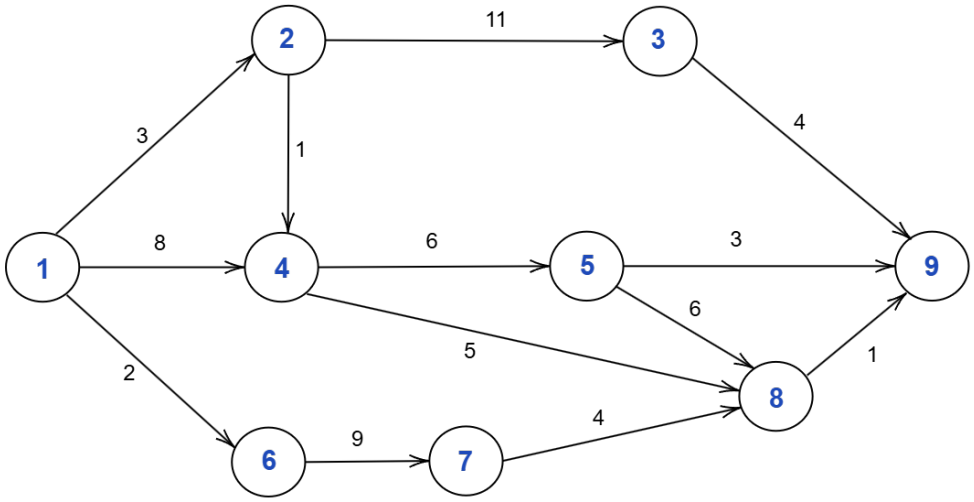

Figura 2.3: Unha rede onde os arcos conteñen os datos de custos c_{ij}.

Exemplo 2.10. Continuemos empregando a rede 2.1 do exemplo 2.6:

Como fixemos nos exemplos 2.6 e 2.9, expliquemos as iteracións do algoritmo ata atopar o camiño máis curto:

- O algoritmo comeza asignando unha etiqueta igual a ∞ a todos os nós menos ao nó orixe (o 1 neste caso), ao que denota permanente cunha etiqueta igual a cero. A continuación escanea os nós cos que conecta este nó orixe (no noso caso, o 2, o 4 e o 6) e estuda se as etiquetas temporais que teñen estes poden ser melloradas. Como é así, actualiza aqueles que pode mellorar (neste caso, melloraría todos porque todos os nós pasarían de etiqueta infinita a 3, 8 e 2 respectivamente) e garda o nó 1 como o nó predecesor dos nós cuxa etiqueta actualiza. Ademais, crearía unha lista L inicialmente baleira e engadiría estes nós á lista L.

Nó	1	2	3	4	5	6	7	8	9
Índice predecesor	-	1	-	1	-	1	-	-	-
Etiqueta de distancia	0	3	∞	8	∞	2	∞	∞	∞

Contido da lista L:	$\{2, 4, 6\}$

Figura 2.4: Información das etiquetas de distancia e índice de predecesores para cada un dos nós e da lista L tras a primeira iteración do algoritmo.

- O algoritmo elixe arbitrariamente un nó de L (sexa o 2), e estuda con que nós se conecta

(3 e 4 neste caso), e mira se pode mellorar a etiqueta temporal dalgún deles. No noso caso pode mellorar a etiqueta de ambos (a do nó 3 pasa de ∞ a 14 e a do nó 4 pasa de 8 a 4), polo que o nó 2 pasa a ser o nó predecesor dos nós 3 e 4. Ademais, engade á lista L o nó 3 (porque o 4 xa está) e quita o nó 2.

Nó	1	2	3	4	5	6	7	8	9
Índice predecesor	-	1	2	2	-	1	-	-	-
Etiqueta de distancia	0	3	14	4	∞	2	∞	∞	∞

Contido da lista L:	$\{3, 4, 6\}$

Figura 2.5: Información das etiquetas de distancia e índice de predecesores para cada un dos nós e da lista L tras a segunda iteración do algoritmo.

- Supoñamos que agora o algoritmo elixe o nó 3 dos da lista L. Este nó conecta de forma directa soamente co nó 9. Polo tanto, actualiza a etiqueta do nó 9 de ∞ a 18 e garda o nó 3 como nó predecesor do nó 9. Non engade o nó 9 á lista porque deste nó non emana ningún arco.

Nó	0	1	2	3	4	5	6	7	8	9
Índice predecesor	-	-	1	2	2	-	1	-	-	3
Etiqueta de distancia	0	∞	3	14	4	∞	2	∞	∞	18

Contido da lista L:	$\{4, 6\}$

Figura 2.6: Información das etiquetas de distancia e índice de predecesores para cada un dos nós e da lista L tras a terceira iteración do algoritmo.

- Neste punto, temos na lista soamente os nós 4 e 6. Poñamos que o algoritmo elixe o nó 6. Como este nó conecta co nó 7, o algoritmo actualiza a súa etiqueta temporal de ∞ a 11, engade o nó 7 á lista L e quita desta lista o nó 6. Ademais, garda o nó 6 como predecesor do nó 7.

Nó	0	1	2	3	4	5	6	7	8	9
Índice predecesor	-	-	1	2	2	-	1	6	-	3
Etiqueta de distancia	0	∞	3	14	4	∞	2	11	∞	18

Contido da lista L:	$\{4, 7\}$

Figura 2.7: Información das etiquetas de distancia e índice de predecesores para cada un dos nós e da lista L tras a cuarta iteración do algoritmo.

- Supoñamos que o algoritmo escolle o nó 7 da lista L. O nó 8 é incidente dun arco que ten orixe no nó 7, polo que o algoritmo mellora a súa etiqueta temporal de ∞ a 15, engade este nó á lista L, quita o nó 7 e garda o nó 7 como nó predecesor do nó 8.

Nó	0	1	2	3	4	5	6	7	8	9
Índice predecesor	-	-	1	2	2	-	1	6	7	3
Etiqueta de distancia	0	∞	3	14	4	∞	2	11	15	18

Contido da lista L:	$\{4, 8\}$

Figura 2.8: Información das etiquetas de distancia e índice de predecesores para cada un dos nós e da lista L tras a quinta iteración do algoritmo.

- Agora o nó pode escoller o nó 8. Este nó conecta co nó 9, e pode mellorar a súa etiqueta temporal de 18 a 15. De novo, como do nó 9 non emana ningún arco non o engade á lista (de suceder o contrario, si que se engadiría), e elimina o nó 8 da lista L. Tamén garda o nó 8 como nó predecesor do no 9.

Nó	1	2	3	4	5	6	7	8	9
Índice predecesor	-	1	2	2	-	1	6	7	8
Etiqueta de distancia	0	3	14	4	∞	2	11	15	15

Contido da lista L:	$\{4\}$

Figura 2.9: Información das etiquetas de distancia e índice de predecesores para cada un dos nós e da lista L tras a sexta iteración do algoritmo.

- Chegados aquí, o único nó que hai na lista L é o nó 4. Polo tanto, estudamos con que nós conecta e vemos que conecta cos nós 5 e 8. Actualiza a etiqueta do nó 5 de ∞ a 14, e actualiza tamén a etiqueta do nó 8 de 15 a 9. Polo tanto, engade tanto o nó 5 como o nó 8

á lista L, garda o nó 4 como nó predecesor de ámbolos dous nós e elimina o nó 4 da lista L.

Nó	1	2	3	4	5	6	7	8	9
Índice predecesor	-	1	2	2	4	1	6	4	8
Etiqueta de distancia	0	3	14	4	14	2	11	9	15

Contido da lista L:	$\{5, 8\}$

Figura 2.10: Información das etiquetas de distancia e índice de predecesores para cada un dos nós e da lista L tras a sétima iteración do algoritmo.

- Supoñamos que o algoritmo escolle agora o nó 5. Este nó conecta tanto co nó 9 como co nó 8. Porén, non actualiza ningunha das etiquetas destes nós porque non consegue melloralas mediante camiños que pasen a través do nó 5. Así, simplemente elimina o nó 5 da lista e non engade ningún novo. Isto tamén implica que non modifica os nós predecesores destes dous nós.

Nó	1	2	3	4	5	6	7	8	9
Índice predecesor	-	1	2	2	4	1	6	4	8
Etiqueta de distancia	0	3	14	4	14	2	11	9	15

Contido da lista L:	$\{8\}$

Figura 2.11: Información das etiquetas de distancia e índice de predecesores para cada un dos nós e da lista L tras a oitava iteración do algoritmo.

- O único nó que resta na lista L é o nó 8. Este nó conecta co nó 9, cuxa etiqueta actualiza de 15 a 14. Ademais, garda o nó 8 como predecesor do nó 9 e elimina o nó 8 da lista L.

Nó	1	2	3	4	5	6	7	8	9
Índice predecesor	-	1	2	2	4	1	6	4	8
Etiqueta de distancia	0	3	14	4	14	2	11	9	14

Contido da lista L:	$\emptyset$

Figura 2.12: Información das etiquetas de distancia e índice de predecesores para cada un dos nós e da lista L tras a novena iteración do algoritmo.

- Neste punto xa non queda ningún nó na lista L, polo que todas as etiquetas se volven permanentes. A etiqueta 14 do nó 9 indica que o camiño máis curto dende o nó 1 ata o nó 9 ten distancia 14. Ademais, para saber cal é este camiño simplemente temos que seguir inversamente o nó predecesor do camiño que parte do nó 9, que neste caso é $9-8-4-2-1$, polo que o camiño máis curto dende o nó 1 ata o nó 9 é o camiño $1-2-4-8-9$.

Na ausencia de ciclos negativos, o grafo de predecesores sempre será unha árbore.

O grafo de predecesores contén un camiño dirixido único dende o nó orixe ata todos os outros nós k e a lonxitude de ditos camiños é como moito $d(k)$. Cando o algoritmo de etiquetado móbil remata, cada arco no grafo de predecesores ten lonxitude de arco reducida nula, o que implica que a lonxitude do camiño dende a orixe ata todos os outros nós k é igual a $d(k)$. Consecuentemente, cando o algoritmo termina, o grafo de predecesores é unha árbore de camiños máis curtos.

É fácil demostrar que cando as lonxitudes son enteiras o algoritmo finaliza nun número finito de iteracións. Notemos que cada $d(j)$ está limitado superiormente por nC (porque un camiño contén como moito $n-1$ arcos e a lonxitude de cada arco é como moito C) e inferiormente por $-nC$. Polo tanto, o algoritmo actualiza calquera etiqueta $d(j)$ como moito $2nC$ veces porque cada actualización de $d(j)$ réstalle como mínimo unha unidade. Consecuentemente, o máximo número de actualizacións de etiquetas de distancia é $2n^2C$. Cada iteración actualiza a etiqueta de distancia, polo que o algoritmo executa $O(n^2C)$ iteracións.

2.6.2. O algoritmo de etiquetado móbil modificado

O algoritmo de etiquetado móbil xenérico non especifica ningún método para seleccionar un arco violando a condición de optimalidade. Un enfoque obvio é escanear a lista de arcos secuencialmente e identificar calquera arco incumprindo esta condición. O problema é que consume moito tempo, polo que a continuación describiremos un mellor enfoque.

Supoñamos que mantemos unha lista L de tódolos arcos que poden non cumprir as súas condicións de optimalidade. Se L está baleira, atopámonos obviamente ante unha solución óptima. Se non o está, examinamos a lista para seleccionar un arco, sexa (i, j) que non cumpra a condición de optimalidade. Entón eliminamos este arco da lista e empregámolo para actualizar a etiqueta de distancia do nó j. Deámonos conta de que calquera diminución na etiqueta de distancia do nó j provoca un decrecemento das lonxitudes reducidas dos arcos que emanan do nó j e algún destes arcos podería non cumprir a condición de optimalidade. Á súa vez, esta redución da distancia $d(j)$ tamén provoca que os arcos que rematan no nó j manteñan a condición de optimalidade. Polo tanto, se $d(j)$ decrece, debemos engadir os arcos de $A(j)$ á lista L. Isto implica que en vez de manter unha lista de tódolos arcos que poden incumprir as condicións de optimalidade,

debemos manter unha lista de nós coa propiedade de que se un arco (i, j) incumpre a condición de optimalidade, a lista L debe conter o nó i. Isto dá lugar a algoritmos máis rápidos.

A exactitude deste algoritmo séguese de que a lista L contén cada nó i tal que existe polo menos un arco (i, j) incumprindo a súa condición de optimalidade. Basta empregar indución nas iteracións para probar que esta propiedade mantense ao longo de todo o algoritmo. Para analizar a complexidade deste algoritmo, caben destacar algunhas observacións. É importante lembrar que sempre que o algoritmo actualiza $d(j)$, este engade o nó j á lista L. O algoritmo selecciona este nó nunha iteración posterior e escanea a súa lista de adxacencia de arcos $A(j)$. En vista de que o algoritmo pode actualizar a etiqueta de distancia $d(j)$ como moito $2nC$ veces, obtemos unha cota de $\sum_{i \in N}(2nC) \cdot \mid A(i) \mid = O(nmC)$ sobre o número total de escaneos de arcos. Polo tanto, o tempo de execución deste algoritmo é $O(nmC)$.

2.6.3. Implementacións especiais do algoritmo de etiquetado móbil modificado

Unha moi boa característica do algoritmo de etiquetado móbil modificado é a súa flexibilidade: podemos seleccionar arcos que non satisfagan a condición de optimalidade sen seguir ningunha orde e aínda así asegurar a convexencia finita do algoritmo. Unha desvantaxe é que sen ningunha outra restrición na elección dos arcos o algoritmo non necesariamente corre en tempo polinómico. De feito, se tomamos unha mala elección en cada iteración, o número de pasos pode volverse exponencial en n. Polo tanto, para obter algoritmos de etiquetado móbil de cota polinómica no tempo de execución debemos organizar as computacións coidadosamente. Se aplicamos o algoritmo de etiquetado móbil modificado a un problema con distancias de arcos non negativas e sempre examinamos o nó da lista L con menor etiqueta de distancia entón atopámonos co algoritmo de Dijkstra. Con esta selección, o tempo de execución do algoritmo é $O(n^2)$.

Nesta sección estudamos dúas novas implementacións do algoritmo de etiquetado móbil modificado.

2.6.3.1. Implementación $O(nm)$

Describimos primeiro esta implementación para o algoritmo de etiquetado móbil modificado. Nesta implementación, organizamos os arcos de A nunha orde específica. Entón facemos pasadas en A. En cada pasada, escaneamos arcos en A un a un e comprobamos a condición $d(j) > d(i) + c_{ij}$. Se o arco satisfai a condición, actualizamos $d(j) = d(i) + c_{ij}$. Esta pasada remata cando non se modifica ningunha etiqueta de distancia en toda unha pasada.

Vexamos que este algoritmo executa como moito $n + 1$ pasadas da lista de arcos. Dado que

cada pasada require $O(1)$ cálculos para cada arco, conclúese que $O(nm)$ é unha cota de tempo para o algoritmo. Podemos afirmar que á fin da k-ésima pasada o algoritmo calculará as distancias dos camiños máis curtos para todos os arcos conectados co nó orixe que empregan k ou menos arcos. A demostración desta afirmación resulta fácil empregando indución no número de pasadas. A nosa afirmación é obviamente verdadeira para $k = 1$. Supoñamos agora que tamén é verdadeira para a k-ésima pasada. Entón $d(j)$ é a lonxitude do camiño máis curto ata o nó j sempre que algún camiño máis curto ata o nó j conteña k ou menos arcos, e noutro caso esta será unha cota superior da distancia do camiño máis curto.

Cosideremos un nó j que estea conectado co nó orixe mediante un camiño máis curto $s = i_0 - i_1 - i_2 - \cdots - i_k - i_{k+1} = j$ consistente en $k + 1$ arcos pero non ten camiño máis curto que teña como moito k arcos. O camiño $i_0 - i_1 - \cdots - i_k$ debe ser un camiño máis curto dende o nó orixe ata o nó i_k, e pola hipótese de indución, a etiqueta de distancia do nó i_k á fin da pasada k-ésima debe ser igual á lonxitude deste camiño. Polo tanto, cando examinamos o arco (i_k, i_{k+1}) no $(k+1)$-ésimo paso, establecemos a etiqueta de distancia do nó i_{k+1} igual á lonxitude do camiño $i_0 - i_1 - \cdots - i_k - i_{k+1}$. Con isto, queda probada a nosa hipótese de indución será tamén certa para a pasada $(k + 1)$-ésima.

Mostramos xa que o algoritmo de etiquetado móbil require $O(nm)$ tempo sempre que en cada pasada examinemos tódolos arcos (e non é necesaria ningunha orde específica). Porén, non necesitamos examinar tódolos arcos. Supoñamos que ordenamos os arcos na lista de arcos polo seu nó de orixe, de tal forma que tódolos arcos co mesmo nó de orixe aparezan consecutivamente na lista. Así, á hora de escanear os arcos, consideramos un de cada vez, sexa o nó i, escaneamos os arcos en $A(i)$ e comprobamos a condición de optimalidade. Supoñamos que durante unha pasada de toda a lista o algoritmo non actualiza a etiqueta de distancia do nó i. Entón na seguinte pasada, $d(j) \leq d(i) + c_{ij}$ para cada arco $(i, j) \in A(i)$ e o algoritmo non precisa comprobar estas condicións. Consecuentemente, podemos gardar tódolos nós cuxas etiquetas de distancia cambian durante a pasada e considerar (ou examinar) soamente ditos nós na seguinte pasada. Unha implementación deste enfoque é gardar os nós nunha lista cuxas etiquetas de distancia cambian nunha pasada e examinar esta lista en orde *first-in, first-out* (é dicir, o primeiro nó en entrar é o primeiro nó en saír), tamén chamada *FIFO*, na seguinte pasada. Establecemos entón o seguinte resultado:

Teorema 2.11. *O algoritmo de etiquetado móbil FIFO resolve o problema do camiño máis curto nun tempo $O(nm)$.*

2.6.3.2. Implementación de lista de dobre extremo

A continuación trataremos unha implementación que ten un comportamento case-polinómico no peor dos casos pero que é moi eficiente na práctica. É moi útil principalmente en redes dispersas (aquelas que teñen moi poucos arcos en comparación cunha rede totalmente densa).

Esta implementación mantén unha lista L como unha lista de dobre extremo (*dequeue*). Esta lista de dobre extremo é unha estrutura de datos que nos permite gardar unha lista tal que poidamos engadir ou eliminar elementos da lista dende calquera dos seus dous extremos. Nesta implementación, sempre se seleccionan nós nun dos extremos da lista (a fronte da lista), pero se engaden nós tanto por ese mesmo extremo como polo oposto (a cola da lista). Se un nó xa estivo na lista L anteriormente, o algoritmo engádeo na fronte; se non, engádeo na cola. A xustificación explicámola a continuación. Se un nó i xa estivo na lista L entón algúns nós, sexan $i_1, i_2, \ldots, i_k$ poden ter o nó i como predecesor. Supoñamos ademais que a lista L contén os nós $i_1, i_2, \ldots, i_k$ cando o algoritmo actualiza $d(i)$ de novo. É vantaxoso actualizar as etiquetas de distancia dos nós $i_1, i_2, \ldots, i_k$ dende o nó i tan pronto como sexa posible antes que examinar os nós $i_1, i_2, \ldots, i_k$ e despois reexaminalos cando as etiquetas de distancia finalmente diminúan por unha diminución de $d(i)$. Reducimos así a necesidade de reexaminar nós.

2.6.4. Detección de ciclos negativos

Nesta sección describiremos modificacións a algoritmos anteriormente mencionados que nos permitirán detectar a presenza dun ciclo negativo na rede no caso de que exista.

Primeiro estudaremos as modificacións necesarias no algoritmo de etiquetado móbil xenérico. Sabemos que se a rede contén un ciclo negativo, ningún conxunto de etiquetas de distancia satisfará a condición de optimalidade. Polo tanto, este algoritmo continuará reducindo as etiquetas de distancia indefinidamente e nunca rematará. Cabe lembrar que $-nC$ é unha cota inferior de calquera etiqueta de distancia sempre que a rede non conteña ningún ciclo negativo. Polo tanto, se atopamos que a etiqueta de distancia para algún nó cae por debaixo de $-nC$ entón podemos parar o algoritmo e finalizar o cálculo. Podemos obter o ciclo negativo rastrexando os índices de predecesores comezando no nó k.

Describamos a continuación outro algoritmo de detección de ciclos negativos. Este algoritmo comproba periodicamente se o grafo de predecesores contén un ciclo dirixido. Primeiro designamos o nó orixe como visitado, e o resto de nós como non visitados. Entón, un por un, examinamos cada nó k non visitado e realizamos a seguinte operación: marcamos o nó k como visitado, trazamos os índices de predecesores comezando no nó k e marcamos como visitados os nós atopados ata chegar ata o primeiro nó que previamente xa denotaramos como visitado, sexa l. Se $k = l$

entón o grafo de predecesores contén un ciclo, que debe ser un ciclo de lonxitude negativa. Este algoritmo require $O(n)$ tempo para comprobar a presenza ou ausencia dun ciclo dirixido no grafo de predecesores. En consecuencia, se aplicamos este algoritmo despois de $\alpha \cdot n$ actualizacións de distancias para unha constante α, os cálculos que realiza non se sumarán á complexidade no peor dos casos de ningún algoritmo de etiquetado móbil.

En xeral, ao tempo no que o algoritmo reetiqueta o nó j cúmprese que $d(j) = d(i) + c_{ij}$ para algún nó i que sexa predecesor do nó j. Referímonos ao arco (i, j) como **arco predecesor**. Posteriormente, $d(i)$ pode diminuír, e as etiquetas satisfarán a condición $d(j) \geq d(i) + c_{ij}$ sempre que o nó predecesor de j sexa o nó i. Supoñamos que P é un camiño de arcos predecesores dende o nó 1 ata o nó j. As desigualdades $d(k) \geq d(l) + c_{kl}$ para tódolos arcos (k, l) neste camiño implican que $d(j)$ é como mínimo a lonxitude dese camiño. Polo tanto, ningún nó j con $d(j) \leq -nC$ está conectado ao nó 1 nun camiño consistindo soamente de arcos predecesores. Concluímos que trazando cara atrás os arcos predecesores dende o nó j debemos obter un ciclo, e que ese ciclo debe ter lonxitude negativa.

O algoritmo de etiquetado móbil FIFO tamén é capaz de detectar a presenza dun ciclo negativo. Para implementar este algoritmo, gardamos o número de veces que o algoritmo examina cada nó. Se a rede non contén ningún ciclo negativo, examina calquera nó un máximo de $(n-1)$ veces. Pola contra, se algún nó é examinado máis de $(n-1)$ veces, entón a rede debe conter un ciclo negativo. Así, este algoritmo detecta a presenza de ciclos negativos ou obtén as distancias de camiños máis curtos nunha rede en $O(mn)$ tempo, que é o máis rápido dispoñible para un algoritmo en tempo polinómico forte para redes con lonxitudes de arcos non negativas.

Capítulo 3

Aplicación do algoritmo de Dijkstra á rede ferroviaria española

Neste capítulo poñeremos en práctica o algoritmo de Dijkstra cun conxunto de datos concernente á rede ferroviaria española, de forma que o usuario do programa proposto poida probar a obter a ruta máis curta entre dúas estacións calesquera.

Comezaremos explicando como se obtiveron os datos e o tratamento dos mesmos. Deseguido, entraremos no detalle do programa implementado en Python, explicando e mostrando o código desenvolvido. Para rematar, daremos exemplos do seu uso e os seus resultados.

3.1. Obtención de datos

Os datos cos que traballamos son datos crus de ADIF onde están rexistradas moitas variables (lonxitude de tramo, puntos inicial e final, o uso da rede, a provincia de orixe e a de destino...) de tódolos tramos dos que consta a rede ferroviaria española. Ademais, tamén empregamos un arquivo no que ADIF ten gardados datos das diferentes estacións, puntos kilométricos e bifurcacións que se atopan na rede (latitude e lonxitude, vila ou cidade, provincia...).

Para a obtención destes datos dispuxemos do programa QGIS (para coñecer máis, véxase [8]), onde agregamos a dirección web dos arquivos de datos de ADIF (`https://ideadif.adif.es/catalog/srv/spa/catalog.search#/metadata/28323796-c2ed-4bb2-bb85-4b6e93ea3334`). Tras isto, cargamos os datos en local e descargamos os datos en arquivos Excel.

3.2. Tratamento dos datos

Os datos en cru obtidos no apartado anterior tivemos que tratalos para obter soamente os tramos desexados, xa que un tramo que comeza e/ou remata nun punto kilométrico ou nunha bifurcación podemos fusionalo co posterior e/ou anterior ata obter un tramo máis longo englobando estes tramos máis pequenos e que comece e finalice en estacións de tren (que é o que nos interesa). Ademais, hai tramos só adicados a mercancías que tampouco nos interesan para o obxectivo deste traballo.

Por outra banda, do arquivo que contén datos de estacións, puntos kilométricos e bifurcacións, eliminamos todas as entradas de datos que non fosen estacións de pasaxeiros.

3.3. Elaboración do programa

Lembrando o pseudocódigo do algoritmo de Dijkstra (Algoritmo 1) e tendo en conta as particularidades da nosa rede, creamos o programa mostrado nos seguintes subapartados. Neles explicamos paso a paso o funcionamento do programa que permite calcular a ruta máis curta entre estacións ferroviarias da rede de ADIF, o cal foi implementado en Python (para ampliar información véxase [9]) empregando Streamlit como interface de usuario e un algoritmo baseado en Dijkstra.

3.3.1. Importación de bibliotecas e carga de datos

```
import streamlit as st
import pandas as pd
import heapq

df_merged = pd.read_csv('Tramos_ADIF.csv')
df_estaciones = pd.read_excel("listado-estaciones-completo-act.xlsx", dtype={'
    CODIGO': str})
```

Comezamos importando as bibliotecas necesarias: `streamlit` para a interface web, `pandas` para manipulacion de datos, e `heapq` para implementar a cola de prioridades do algoritmo de Dijkstra. A continuación, cárganse dous ficheiros: un CSV cos tramos da rede ferroviaria e un Excel co listado completo de estacións. O campo `CODIGO` é lido como cadea para evitar erros de formato.

3.3.2. Limpeza e filtrado de estacións e creación de dicionarios

```
df_estaciones = df_estaciones.rename(columns={...})
df_estaciones = df_estaciones[['codigo', 'nome', 'provincia', 'latitud', '
    longitud']]

codigos_validos = set(df_merged['deporigen']) | set(df_merged['depdestino'])
df_estaciones = df_estaciones[df_estaciones['codigo'].isin(codigos_validos)]

cod_a_nombre = pd.concat([...]).drop_duplicates().set_index('codigo')['nome'].
    to_dict()
nome_a_cod = {v: k for k, v in cod_a_nombre.items() if pd.notna(v)}

codigos_con_coords = set()
codigos_con_coords.update(...)
```

Nesta parte do código renoméanse as columnas do Excel para traballar cunha nomenclatura común. Tamén se seleccionan unicamente as columnas relevantes para o resto do programa, e a continuación fíltranse as estacións para conservar soamente as que aparecen nos tramos cargados, ben como orixe ou como destino. Logo créanse dous dicionarios:

- `cod_a_nombre`: asocia o código dunha estacion co seu nome.
- `nome_a_cod`: dicionario inverso para obter o código a partir do nome.

Por último, créase un conxunto co código das estacións que teñen coordenadas xeográficas (latitude e lonxitude) non nulas.

3.3.3. Interfaz de usuario con Streamlit

```
provincias_orixe = sorted(df_estaciones['provincia'].dropna().unique())
provincias_destino = sorted(df_estaciones['provincia'].dropna().unique())

# Seleccion provincia orixe
provincia_orixe = st.selectbox("Selecciona a provincia de orixe",
    provincias_orixe)

# Filtrar estacions orixe segundo provincia e que tenhan coordenadas validas
df_estaciones_orixe = df_estaciones[
    (df_estaciones['provincia'] == provincia_orixe) &
    (df_estaciones['codigo'].isin(codigos_con_coords))
]
estacions_orixe_cod = df_estaciones_orixe['codigo'].tolist()
estacions_orixe_nome = sorted([cod_a_nombre.get(c, "Sen nome") for c in
    estacions_orixe_cod])

```

```
15 # Seleccion estacion orixe
16 nome_estacion_orixe = st.selectbox("Selecciona a estacion de orixe",
       estacions_orixe_nome)
17 codigo_estacion_orixe = nome_a_cod.get(nome_estacion_orixe, None)
18
19 # Seleccion provincia destino
20 provincia_destino = st.selectbox("Selecciona a provincia de destino",
       provincias_destino)
21
22 # Filtrar estacions destino segundo provincia e que tenhan coordenadas validas
23 df_estaciones_destino = df_estaciones[
24     (df_estaciones['provincia'] == provincia_destino) &
25     (df_estaciones['codigo'].isin(codigos_con_coords))
26 ]
27 estacions_destino_cod = df_estaciones_destino['codigo'].tolist()
28 estacions_destino_nome = sorted([cod_a_nombre.get(c, "Sen nome") for c in
       estacions_destino_cod])
29
30 # Seleccion estacion destino
31 nome_estacion_destino = st.selectbox("Selecciona a estacion de destino",
       estacions_destino_nome)
32 codigo_estacion_destino = nome_a_cod.get(nome_estacion_destino, None)
33
34 # Amosar seleccion
35 st.write(f"Seleccionaches: Orixe **{nome_estacion_orixe}** ({provincia_orixe}),
       "
36          f"Destino **{nome_estacion_destino}** ({provincia_destino})")
```

Nesta parte do código amósase nunha interfaz de Streamlit. O primeiro que facemos é ordenar alfabeticamente os nomes das provincias e das estacións, e logo créanse catro desplegables:

- O primeiro despregable permite ao usuario elixir a provincia de orixe.
- Co segundo despregable, o usuario pode elixir a estación de orixe. As estacións de orixe dispoñibles están filtradas segundo a provincia de orixe escollida.
- O terceiro despregable permite ao usuario escoller a provincia de destino.
- Co cuarto despregable, podemos elixir a estación de destino da nosa ruta de entre as estacións pertencentes á provincia destino elixida previamente.

Como comprobación, tamén mandamos que o programa imprima as estacións e provincias tanto de orixe como de destino antes de facer o cálculo.

3.3.4. Creación do grafo e implementación do algoritmo de Dijkstra con cambios de dirección

```
grafo = {}
for _, row in df_merged.iterrows():
    o = row['deporigen']
    d = row['depdestino']
    w = row['shape_leng']
    grafo.setdefault(o, {})[d] = {'weight': w}

def dijkstra_con_giros(grafo, inicio, fin, codigos_con_coords):
    cola = []
    heapq.heappush(cola, (0, inicio, [inicio], 'forward'))  # estado inicial

    visitados = set()

    while cola:
        dist_actual, nodo, caminho, direccion = heapq.heappop(cola)

        clave_visita = (nodo, direccion)
        if clave_visita in visitados:
            continue
        visitados.add(clave_visita)

        if nodo == fin:
            return caminho, dist_actual

        vecinhos = grafo.get(nodo, {})

        if direccion == 'forward':
            for vecinho, datos in vecinhos.items():
                if (vecinho, direccion) not in visitados:
                    heapq.heappush(cola, (dist_actual + datos['weight'], vecinho
    , caminho + [vecinho], 'forward'))
        else:  # backward
            for pai, conexion in grafo.items():
                if nodo in conexion:
                    if (pai, direccion) not in visitados:
                        peso = conexion[nodo]['weight']
                        heapq.heappush(cola, (dist_actual + peso, pai, caminho +
     [pai], 'backward'))

        if nodo in codigos_con_coords:
            nova_direccion = 'backward' if direccion == 'forward' else 'forward'
            heapq.heappush(cola, (dist_actual, nodo, caminho, nova_direccion))

    return None, float('inf')
```

Nesta parte do código primeiro creamos un grafo dirixido onde o custo de cada arco é a respectiva lonxitude.

Tras isto, definimos o algoritmo de Dijkstra. Para calcular a ruta máis curta entre dúas estacións da rede ferroviaria, empregouse unha modificación do algoritmo clásico de Dijkstra. Esta adaptación permite cambios de dirección (é dicir, moverse en sentido contrario ao habitual dos tramos) pero só en estacións concretas: aquelas que teñen coordenadas xeográficas dispoñibles (para evitar que nunha bifurcación, por exemplo, tome un tramo inexistente). Esta restrición simula o comportamento real da rede, onde só é posible realizar cambios de sentido en estacións determinadas que contan con infraestrutura física para iso.

A función desenvolve unha busca do camiño máis curto entre un nó de inicio e un nó de destino, tendo en conta os seguintes factores:

- Só se pode avanzar nunha mesma dirección en tódolos tramos coa excepción do seguinte punto.
- Só se permite "xirar", é dicir, cambiar a dirección do desprazamento, nas estacións reais (e non nas bifurcacións, como antes comentamos).

Polo tanto, os pasos que segue esta parte do código son:

1. **Inicialización:** Emprégase unha cola de prioridade (implementada cun heap) que garda catro elementos en cada estado: a distancia acumulada, o nó actual, o camiño seguido ata ese punto, e a dirección do desprazamento (*forward* ou *backward*).
2. **Xestión de nodos visitados:** Para evitar ciclos, gárdanse os pares formados por nó e dirección nun conxunto de estados xa visitados. Isto permite que un mesmo nodo se poida visitar en distintas direccións.
3. **Condición de remate:** Cando o nó actual coincide co nó destino, remata a execución e devólvese o camiño e a distancia total percorrida.
4. **Exploración de veciños:**
   - Se se avanza en dirección *forward*, explóranse os veciños definidos polos tramos da rede, é dicir, as estacións que se poden alcanzar directamente desde a estación actual.
   - Se se avanza en dirección *backward*, búscanse todos os nós que teñan conexións entrantes cara ao nó actual, simulando un movemento en sentido inverso.
5. **Permitir cambios de dirección:** En cada nó, compróbase se este ten coordenadas xeográficas non nulas. Se é así, permítese cambiar de dirección e continuar a busca desde ese mesmo punto, sen engadir custo adicional á distancia total.

6. **Remate sen solución:** Se se esgota a cola sen chegar ao nodo destino, devólvese un valor especial indicando que non existe un camiño posible entre os dous puntos, probablemente debido a que as estacións pertencen a subredes desconectadas (como redes de Cercanías sen conexión co resto da rede principal).

3.3.5. Cálculo e visualización da ruta

```
if st.button("Calcular ruta mais curta"):
    if codigo_estacion_orixe is None or codigo_estacion_destino is None:
        st.error("As estacions seleccionadas non tenhen un codigo valido.")
    else:
        caminho, distancia = dijkstra_con_giros(grafo, codigo_estacion_orixe, codigo_estacion_destino, codigos_con_coords)
        if caminho is None:
            st.error("Non hai ruta posible entre esas estacions. Probablemente algunha desas estacions forme parte dunha rede de Cercanias desconectada do resto da rede.")
        else:
            coordenadas_caminho = []
            caminho_filtrado = []
            for idx, c in enumerate(caminho):
                fila_orixe = df_merged[df_merged['deporigen'] == c].head(1)
                fila_destino = df_merged[df_merged['depdestino'] == c].head(1)

                lat = None
                lon = None
                nome = cod_a_nombre.get(c, "Sen nome")

                if not fila_orixe.empty:
                    lat = fila_orixe.iloc[0]['latitud_origen']
                    lon = fila_orixe.iloc[0]['longitud_origen']
                elif not fila_destino.empty:
                    lat = fila_destino.iloc[0]['latitud_destino']
                    lon = fila_destino.iloc[0]['longitud_destino']

                if pd.notna(lat) and pd.notna(lon):
                    tipo = "intermedia"
                    if idx == 0:
                        tipo = "orixe"
                    elif idx == len(caminho) - 1:
                        tipo = "destino"

                    coordenadas_caminho.append({
                        "lat": lat,
                        "lon": lon,
```

```
                        "tipo": tipo
                    })
                    caminho_filtrado.append(nome)

        if not caminho_filtrado:
            st.warning("A ruta existe, pero ningunha estacion intermedia ten coordenadas disponhibles.")
        else:
            st.success("Ruta mais curta con estacions xeorreferenciadas:")
            st.write(" --> ".join(caminho_filtrado))
            st.write(f"Distancia total: {distancia / 1000:.2f} quilometros")

            import pydeck as pdk
            df_coords = pd.DataFrame(coordenadas_caminho)

            lineas_layer = pdk.Layer(
                "PathLayer",
                data=[{"path": [[row["lon"], row["lat"]] for row in coordenadas_caminho]}],
                get_path="path",
                get_width=5,
                get_color=[0, 100, 255],
                width_min_pixels=2,
            )

            estacions_layer = pdk.Layer(
                "ScatterplotLayer",
                data=df_coords,
                get_position='[lon, lat]',
                get_radius=1500,
                get_color="""
                    [tipo === 'orixe' || tipo === 'destino' ? 0 : 220,
                     tipo === 'orixe' || tipo === 'destino' ? 180 : 30,
                     tipo === 'orixe' || tipo === 'destino' ? 0 : 30,
                     255]
                """,
                pickable=True,
            )

            st.pydeck_chart(pdk.Deck(
                map_style="mapbox://styles/mapbox/light-v9",
                initial_view_state=pdk.ViewState(
                    latitude=df_coords["lat"].mean(),
                    longitude=df_coords["lon"].mean(),
                    zoom=6,
                    pitch=0,
                ),
```

```
81                      layers=[lineas_layer, estacions_layer],
82                  ))
```

Esta parte do código é a encargada de poñer en funcionamento o algoritmo de busca da ruta máis curta cando o usuario pulsa o botón `Calcular ruta mais curta`, así como de elaborar o mapa para a visualización da ruta e cada unha das estacións que atravesa. Ademais, encárgase de xestionar posibles erros e de mostrar os resultados de forma visual e textual na interface.

Ao premer o botón, execútanse varias comprobacións e operacións:

- **Comprobación de validez:** Primeiro verifícase que tanto a estación de orixe como a de destino teñen códigos válidos. Se algún deles non é válido, amósase unha mensaxe de erro indicando que non se pode proceder.
- **Execución do algoritmo:** En caso de que ambas estacións sexan válidas, chámase á función `dijkstra_con_giros` pasando como parámetros o grafo da rede ferroviaria, os códigos das estacións seleccionadas e o conxunto de estacións nas que se permiten cambios de dirección.
- **Xestión de rutas non dispoñibles:** Se o resultado da función é `None`, significa que non existe un camiño posible entre as dúas estacións. Neste caso, amósase un erro indicando que as estacións poderían pertencer a redes ferroviarias desconectadas.

Se se atopou unha ruta válida, procédese á súa análise e visualización:

- **Extracción de coordenadas:** Para cada estación no camiño devolto, búscase a súa latitude e lonxitude, consultando tanto as columnas de orixe como as de destino. Isto é necesario porque unha estación pode aparecer como punto de partida ou chegada nun tramo distinto.
- **Clasificación de estacións:** As estacións no camiño clasifícanse segundo a súa posición:
  - A primeira estación identifícase como *orixe*.
  - A última estación como *destino*.
  - As intermedias como *intermedia*.

  Esta clasificación permite distinguilas visualmente ao representalas no mapa.
- **Xestión de casos sen coordenadas:** Se ningunha das estacións ten coordenadas válidas, amósase unha advertencia indicando que a ruta existe, pero non se pode representar xeograficamente.

- **Visualización textual:** En caso contrario, móstrase o camiño como unha secuencia de nomes de estacións unidos por frechas, e tamén se amosa a distancia total en quilómetros cunha precisión de dúas cifras decimais.

Unha vez procesado o camiño e extraídas as coordenadas das estacións implicadas, utilízase a biblioteca `pydeck` para representar graficamente o traxecto sobre un mapa xeográfico.

- **Creación do `DataFrame` de coordenadas:** Primeiro, as coordenadas almacenadas nunha lista de dicionarios son convertidas nun `DataFrame` chamado `df_coords`. Este paso é necesario para poder pasarlle os datos estruturados ás capas de visualización de `pydeck`.
- **Capa de liñas (`PathLayer`):** Esta capa encárgase de debuxar a liña continua que representa o traxecto. A liña defínese como unha secuencia ordenada de puntos GPS, onde cada punto é unha parella `[lonxitude, latitude]`. A liña preséntase cun grosor e unha cor azul para destacar a ruta.
- **Capa de puntos (`ScatterplotLayer`):** Esta segunda capa representa visualmente cada estación como un punto no mapa. A súa aparencia depende do tipo de estación:
  - As estacións de orixe e destino representan os extremos da ruta, e aparecen cunha cor verde escura.
  - As estacións intermedias represéntanse cun ton alaranxado, máis suave.

  A diferenza cromática facilita distinguir rapidamente os elementos principais do traxecto.
- **Configuración da vista inicial:** A vista do mapa inicial está centrada na media das latitudes e longitudes das estacións implicadas na ruta. Así, o mapa amosa automaticamente a zona relevante, cun nivel de zoom e orientación adecuados.
- **Renderizado final:** Finalmente, utilízase a función `st.pydeck_chart()` para renderizar o mapa e amosalo na aplicación de Streamlit. Esta integración permite unha visualización dinámica, interactiva e inmediata do camiño máis curto directamente sobre o mapa base de `Mapbox`.

Esta representación gráfica é un complemento fundamental á visualización textual do traxecto, permitindo ao usuario entender mellor o percorrido, as distancias e a distribución espacial das estacións implicadas. Ademais, ao distinguir as estacións principais mediante cores, mellórase a experiencia de interpretación visual dos resultados.

3.4. Exemplos prácticos

A continuación mostramos algúns exemplos de uso do programa elaborado.

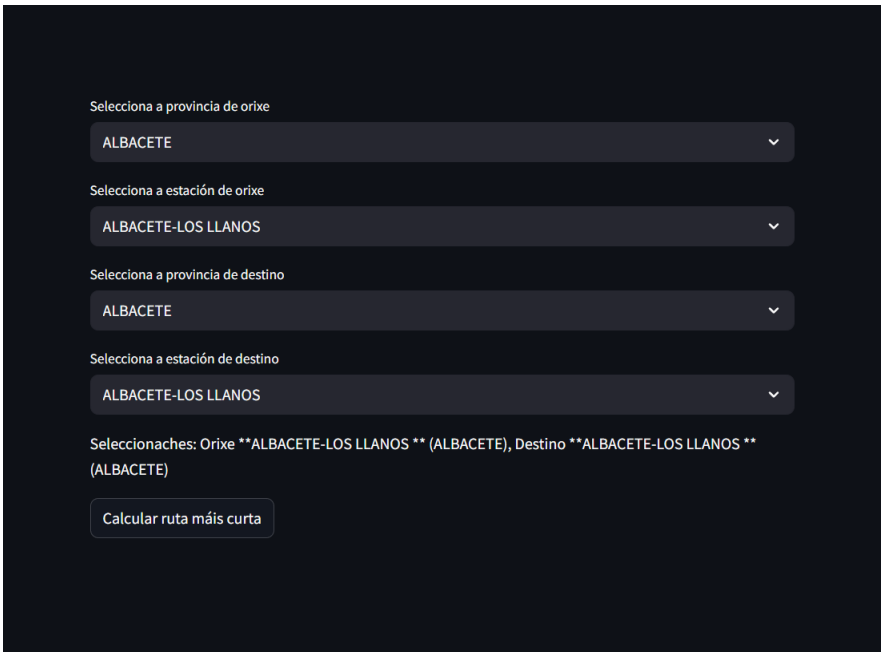

Figura 3.1: Xanela aberta cando iniciamos o programa.

Como se pode observar na Figura 3.1, dado que previamente ordenamos as provincias e as estacións alfabeticamente, ao executar o programa a provincia por defecto é Albacete tanto na orixe coma no destino e a estación por defecto tanto na orixe coma no destino é Albacete-Los Llanos. Tamén podemos ver a liña de comprobación das estacións e provincias seleccionadas e incluso o botón para calcular a ruta máis curta.

Na Figura 3.2 podemos ver exemplos de uso do programa, con distintas estacións de orixe e de destino. Podemos tamén ver como vai variando o zoom do mapa (que tamén se pode variar manualmente), e no caso da ruta entre Plasencia e Asamblea de Madrid-Entrevías da imaxe da dereita na Figura 3.2 podemos fixarnos en que os círculos das estacións inicial, intermedias e final son máis grandes debido ao maior nivel de zoom, e podemos distinguir perfectamente as cores.

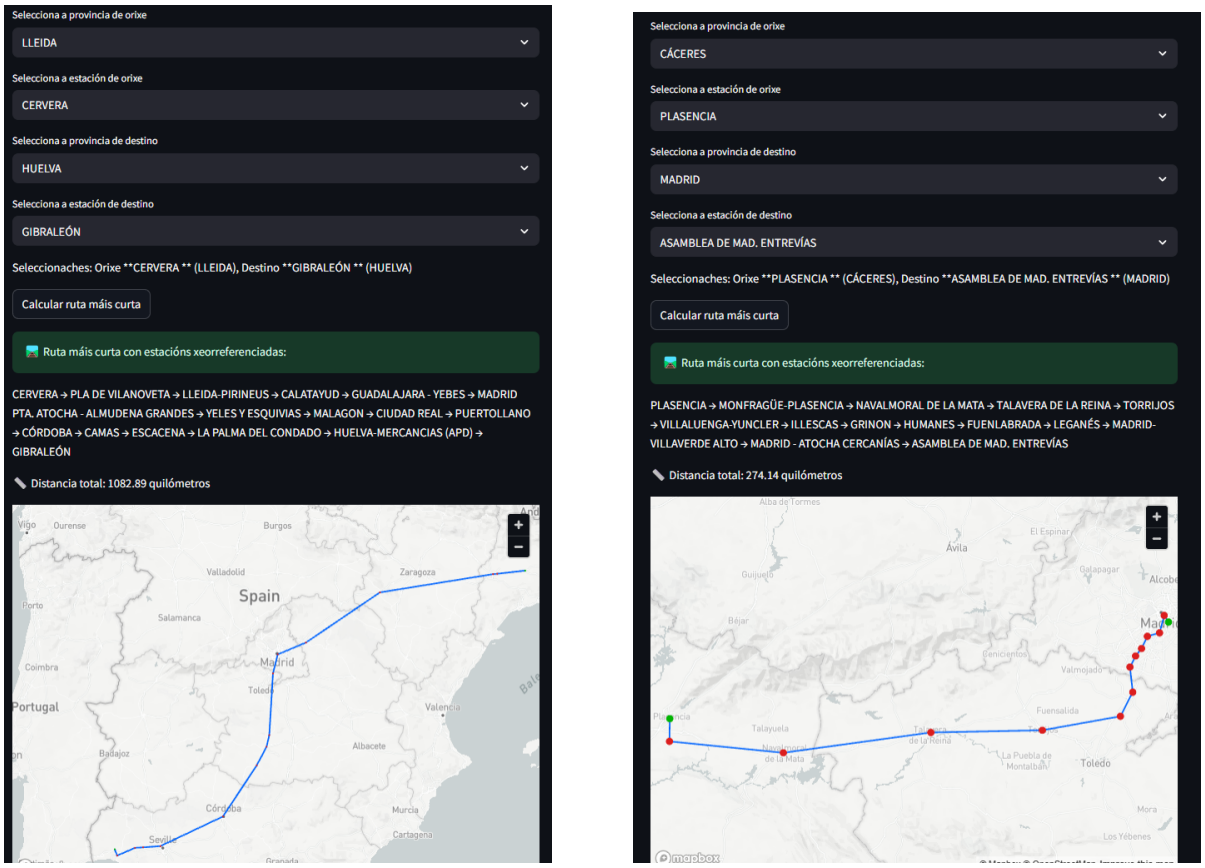

Figura 3.2: Exemplos de uso do programa.

Capítulo 4

Conclusións e futura investigación

Ao longo deste traballo realizouse unha análise profunda e sistemática do problema do camiño máis curto, abordando tanto os fundamentos teóricos que o sustentan como as distintas técnicas algorítmicas dispoñibles para a súa resolución. Prestouse especial atención ao algoritmo de Dijkstra, estudando as súas propiedades, a súa validez formal e as melloras que permiten aumentar a súa eficiencia en distintos contextos. A través deste enfoque, fíxose evidente a relevancia deste algoritmo dentro da optimización en redes e a súa ampla utilidade en problemas reais que requiren solucións rápidas e fiables.

A implementación práctica desenvolvida, centrada no cálculo de rutas ferroviarias entre estacións, demostrou a versatilidade destes algoritmos cando se integran con ferramentas modernas de visualización e programación. Esta parte aplicada resultou esencial para evidenciar o salto desde a teoría matemática ata solucións útiles en contornas reais.

Como liñas de investigación futura, sería interesante estudar algoritmos que admitan pesos negativos (como Bellman-Ford, véxase na referencia [3]), así como outros algoritmos de resolución como o algoritmo A estrela (para saber máis, consúltese [7]), e comparar o seu rendemento computacional coa do algoritmo de Dijkstra. Tamén queda aberta a investigación para un mellor e máis profundo tratamento dos datos de ADIF e a implementación doutros algoritmos no caso práctico para a comparación das rutas obtidas e o tempo empregado por cada un dos algoritmos.

En definitiva, este traballo confirma a importancia dos algoritmos de optimización en redes como ferramentas fundamentais no mundo actual, e convida a seguir afondando nas súas múltiples posibilidades.

Bibliografía

[1] R.K. Ahuja, T.L. Magnanti, and J.B. Orlin. *Network Flows: Theory, Algorithms, and Applications.* Prentice Hall, 1993.

[2] M.S. Bazaraa, J.J. Jarvis, and H.D. Sherali. *Linear Programming and Network Flows.* Wiley, 2009.

[3] R. Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 16:87–90, 1958.

[4] Robert B. Dial. Algorithm 360: shortest-path forest with topological ordering [h]. *Commun. ACM*, 12(11):632–633, November 1969.

[5] E.W. Dijkstra and E.W. Dijkstra. *A Primer of ALGOL 60 Programming: Together with Report on the Algorithmic Language ALGOL 60.* A.P.I.C. studies in data processing. Automatic Programming Information Centre, Brighton College of Technology, Eng., 1962.

[6] Julio González Díaz. *Apuntes de la asignatura Programación Lineal y Entera.* Grado en Matemáticas. Universidad de Santiago de Compostela, 2021.

[7] Peter Hart, Nils Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[8] QGIS Development Team. *QGIS Geographic Information System.* Open Source Geospatial Foundation, 2009.

[9] Guido Van Rossum and Fred L Drake Jr. *Python reference manual.* Centrum voor Wiskunde en Informatica Amsterdam, 1995.