USC
UNIVERSIDADE
DE SANTIAGO
DE COMPOSTELA

FACULTADE DE MATEMÁTICAS

Traballo Fin de Grao

EL PROBLEMA DEL CAMINO MÁS CORTO: ALGORITMOS Y APLICACIONES

Alba Gude Santos

2020/2021

UNIVERSIDADE DE SANTIAGO DE COMPOSTELA

GRADO DE MATEMÁTICAS

Traballo Fin de Grao

EL PROBLEMA DEL CAMINO MÁS CORTO: ALGORITMOS Y APLICACIONES

Alba Gude Santos

05/Xullo/2021

UNIVERSIDADE DE SANTIAGO DE COMPOSTELA

Trabajo propuesto

Área de Coñecemento: Estadística e Investigación Operativa
Título: El problema del camino más corto: Algoritmos y aplicaciones
Breve descrición do contido
Neste traballo de fin de grao profundizarase no estudio do problema do camiño máis curto, tanto desde o punto de vista teórico como do punto de vista da implementación práctica de algoritmos e aplicacións.
Recomendacións
Outras observacións

Índice general

Resumen viii

Introducción xi

1. Conceptos y resultados de problemas de flujo en redes 1
- 1.1. Grafos . . . 1
- 1.2. Matrices de información de un grafo . . . 3
- 1.3. Listas . . . 4
- 1.4. Árboles . . . 4

2. Problema del camino más corto 5
- 2.1. Formulación del problema del camino más corto . . . 6
- 2.2. Símil físico del problema del camino más corto . . . 8
- 2.3. Árbol de caminos más cortos . . . 9

3. Definición, análisis y tipos de algoritmos 11
- 3.1. Análisis de complejidad de un algoritmo . . . 12
  - 3.1.1. Diferentes medidas de complejidad . . . 12
  - 3.1.2. Tamaño del problema . . . 13
  - 3.1.3. Complejidad en el peor caso . . . 13
  - 3.1.4. Tipos de notaciones . . . 14
  - 3.1.5. Algoritmos de tiempo polinomial y exponencial . . . 15
  - 3.1.6. Funciones potenciales y complejidad amortizada . . . 16
- 3.2. Desarrollo de algoritmos de tiempo polinomial . . . 16
  - 3.2.1. Enfoque de mejora geométrica . . . 17
  - 3.2.2. Enfoque de escala . . . 18
  - 3.2.3. Enfoque de programación dinámica . . . 18
  - 3.2.4. Búsqueda binaria . . . 19

3.3. Tipos de algoritmos 19
3.3.1. Algoritmos de búsqueda 19
3.3.2. Algoritmos de descomposición de flujo 22
3.3.3. Algoritmos de ajuste y corrección de etiquetas 25
3.4. Algoritmo en redes acíclicas 25

4. Algoritmos de configuración de etiquetas **29**
4.1. Algoritmo de Dijkstra 29
4.1.1. Tiempo de ejecución del algoritmo de Dijkstra 31
4.1.2. Implementación del algoritmo de Dijkstra en R 32
4.2. Algoritmo de Dial 33
4.3. Algoritmo de Radix-Heap 35

5. Algoritmos de corrección de etiquetas **41**
5.1. Condiciones de optimalidad 42
5.2. Detección de ciclos negativos 43
5.3. Algoritmo genérico de corrección de etiquetas 44
5.4. Algoritmo de corrección de etiquetas modificado 47
5.4.1. Implementación de eliminación de cola 48

6. Aplicaciones del problema del camino más corto **49**
6.1. El problema de la mochila 49

Bibliografía **53**

Resumen

En esta memoria introduciremos el problema del camino más corto y definiremos distintos tipos de algoritmos para resolverlo. En particular, trataremos de resolver el problema del camino más corto desde un nodo origen a otro nodo de la red. En este trabajo, diferenciaremos los algoritmos en dos capítulos. Primero veremos los algoritmos que resuelven el problema del camino más corto cuando las longitudes de los arcos son no negativas. Después veremos algoritmos más generales, para redes con longitudes arbitrarias que, o bien encuentran el camino más corto, o bien detectan la presencia de ciclos de longitud total negativa. Finalmente, presentaremos algunas aplicaciones prácticas del problema del camino más corto que aparecen en la vida real.

Abstract

In this report we will introduce the shortest path problem and define different types of algorithms to solve it. In particular, we will try to solve the shortest path problem from a source node to another node in the network. In this paper, we will differentiate the algorithms in two chapters. First we will look at algorithms that solve the shortest path problem when arc lengths are non-negative. Then we will look at more general algorithms, for networks with arbitrary lengths that either find the shortest path or detect the presence of cycles of negative total length. Finally, we will present some practical applications of the shortest path problem that appear in real life.

Introducción

Este trabajo se basa fundamentalmente en el libro "Network Flows: Theory, Algorithms and Applications", publicado en 1993 y cuyos autores son Ravindra K. Ahuja, Thomas L. Magnanti y James B. Orlin. Se trata de un libro dedicado completamente a los problemas de flujo en redes, y en este trabajo haremos una recopilación de los conceptos, resultados y algoritmos de ese libro relativos al problema del camino más corto. Además, para hacer el trabajo más autocontenido, nos apoyaremos en algunas definiciones y resultados básicos de los apuntes de Programación lineal y entera de Julio González Díaz, apuntes que también usaremos puntualmente en algún otro apartado de la memoria.

El problema del camino más corto es un tipo de problema de flujo en redes y consiste, como su nombre indica, en encontrar el camino de menor longitud entre dos puntos que llamaremos nodos. En particular, trataremos de calcular el camino dirigido más corto desde un nodo fuente o origen a un nodo cualquiera o a un nodo destino. Este problema aparece muy a menudo en la práctica, ya que en muchas situaciones de la vida cotidiana tenemos la necesidad de enviar un objeto (un mensaje, una carta, una llamada, un vehículo, ...) de un lugar a otro y queremos hacerlo de la forma más rápida y económica posible.

Para resolverlo describiremos diferentes tipos de algoritmos y analizaremos su eficiencia en la práctica. Un algoritmo es un conjunto de reglas que nos permite resolver un determinado problema. Dados unos datos iniciales, se realizan los sucesivos pasos del algoritmo y llegamos a una solución. Este problema puede ser resuelto, por ejemplo, por el método *Símplex*, pero el más utilizado y conocido es el *Algoritmo de Dijkstra*, del cual hablaremos en esta memoria.

Actualmente, lo más cómodo es resolver este tipo de problemas con la ayuda de un ordenador si el número de datos del problema es muy grande. Para ello, necesitamos conocer el algoritmo y saber implementarlo en un ordenador. A continuación introducimos los datos de nuestro problema y obtenemos la solución. º

Capítulo 1

Revisión de conceptos y resultados de problemas de flujo en redes

En este capítulo introduciremos las principales definiciones que necesitaremos a lo largo del trabajo.

1.1. Grafos

Un **grafo** $G = (N, A)$ es un conjunto de elementos llamados **nodos** que pueden estar conectados por un conjunto de **arcos** o aristas. Denotamos por N el conjunto de nodos y por A el conjunto de aristas. Presentamos ahora unas definiciones previas para introducir más adelante los tipos de grafos.

Definición 1.1. Un **arco dirigido** (i, j) tiene dos extremos, i y j. Llamaremos **cola** del arco (i, j) al nodo i y **cabeza** al nodo j.

Definición 1.2. Un **camino** es una secuencia de arcos dirigidos que describen la trayectoria para ir de un nodo a otro. En un camino, todos los vértices son distintos.

Definición 1.3. Una **cadena** es un camino donde los arcos son no dirigidos.

Definición 1.4. Una **cadena cerrada** es una cadena donde el nodo origen y el nodo destino son el mismo.

Definición 1.5. Un **ciclo** es una cadena cerrada en la que no coinciden más nodos que el primero y el último.

Diremos que un ciclo es de **longitud total negativa** si al sumar la longitud de los arcos que lo forman obtenemos un valor negativo. Para abreviar, nos referiremos a este tipo de ciclos como **ciclos negativos**.

Definimos a continuación dos tipos de grafos:

Definición 1.6. Un grafo $G = (N, A)$ es un **grafo dirigido** si los arcos son pares ordenados. Es decir, el arco (i, j) empieza en el nodo i y termina en el nodo j.

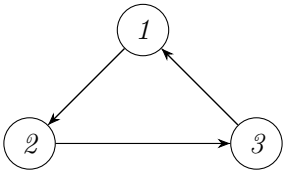
Definición 1.7. Un grafo $G = (N, A)$ es un **grafo no dirigido** si (i, j) y (j, i) representan la misma arista. Es decir, podemos ir de i a j y de j a i.

También diremos que el arco (i, j) empieza en el nodo i y termina en el nodo j, ya que es una arista saliente del nodo i y entrante del nodo j. Cuando $(i, j) \in A$, diremos que el nodo j es adyacente al nodo i.

Si existe un camino compuesto de uno o más arcos que una i con j, diremos que i es el nodo predecesor de j y lo denotaremos por $pred(j) = i$.

Definición 1.8. Un **grafo predecesor** es aquel que se forma a partir de los índices predecesores de todos los nodos que forman un camino.

Ejemplo 1.9. *Dado el siguiente grafo, vamos a obtener los conjuntos N y A en diferentes casos:*



El conjunto de nodos de este grafo dirigido es $N = \{1, 2, 3\}$ y el de aristas $A = \{(1, 2), (2, 3), (3, 1)\}$. Sin embargo, si el grafo fuera no dirigido, $N = \{1, 2, 3\}$ y $A = \{(1, 2), (1, 3), (2, 1), (2, 3), (3, 1), (3, 2)\}$.

Definición 1.10. Diremos que un grafo es **acíclico** si no contiene ningún ciclo dirigido.

Definición 1.11. Diremos que un grafo es **bipartito** si sus nodos se pueden separar en dos conjuntos disjuntos, de manera que los nodos de un mismo conjunto no están relacionados.

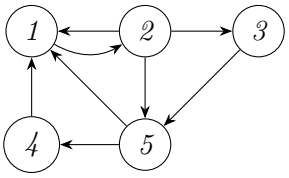
1.2. Matrices de información de un grafo

Definición 1.12. Dado un grafo $G = (N, A)$, este puede ser representado por su **matriz de adyacencia**, que es una matriz A_{nxn} (n es el número de nodos) definida por:

$$a_{ij} = \begin{cases} 1 & \text{se } (i,j) \text{ é unha arista en G} \\ 0 & \text{noutro caso.} \end{cases}$$

Definición 1.13. Dado un grafo $G = (N, A)$, para un nodo $i \in N$, definimos su **lista de adyacencia**, y será denotada por $A(i)$, como la i-ésima fila de la matriz de adyacencia de G.

Ejemplo 1.14. *Dado el siguiente grafo dirigido:*



Su matriz de adyacencia sería:

$$\begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \end{pmatrix}$$

Y la lista de adyacencia del nodo 1 será: $A(1) = (0, 1, 0, 0, 0)$

1.3. Listas

Una lista enlazada individualmente puede almacenar elementos en un orden arbitrario (en vez de un conjunto ordenado secuentalmente como en una matriz), pero requiere información adicional que nos permita acceder a los datos en el orden especificado en el conjunto ordenado. Una **celda** es un componente esencial de las listas enlazadas y podemos pensarla como una caja capaz de contener varios valores, llamados **campos**.

Definición 1.15. Entonces, una **lista enlazada individualmente** es una colección de celdas enlazadas entre sí. Cada celda tiene dos campos: un campo de datos y un campo de enlace.

Definición 1.16. Una **lista doblemente enlazada** es lo mismo que una lista enlazada individualmente, excepto que cada celda tiene dos enlaces, uno a la celda anterior y el otro a la celda siguiente.

Una ventaja de las listas doblemente enlazadas es que podemos recorrer la lista fácilmente en ambas direcciones y realizar una eliminación arbitraria de un elemento en $O(1)$ tiempo.

1.4. Árboles

Diremos que un grafo es **conexo** si para cada par de nodos existe una cadena no dirigida que los une. Si, además, existe una cadena dirigida que los une, diremos que es un grafo **fuertemente conexo**.

Definición 1.17. Un **árbol** es un grafo conexo que no contiene ciclos.

Definición 1.18. Un **árbol de camino más corto** es un árbol externo dirigido enraizado en el nodo origen, con la propiedad de que el único camino desde el origen a cualquier nodo del grafo es el camino más corto a ese nodo.

Capítulo 2

Problema del camino más corto: formulación y resultados

Los problemas del camino más corto son un caso particular de los problemas de flujo en redes y aparecen muchas veces en la práctica cuando, por ejemplo, queremos enviar un objeto entre dos lugares concretos de la forma más rápida y económica posible. Además de ser problemas fáciles de resolver de forma eficiente, son un punto de referencia para analizar problemas de red más complejos; y aparecen muchas veces como subproblemas al resolver problemas de optimización en redes. A pesar de que estos problemas son fáciles de resolver, el estudio del problema del camino más corto es un punto de partida para introducir ideas importantes de los problemas de optimización en redes, como son las estructuras de datos inteligentes o el escalado de datos para mejorar el rendimiento de un algoritmo en el peor caso.

Definición 2.1. El *problema del camino más corto* en un grafo dirigido $G = (N, A)$ con longitudes $c \in \mathbb{R}^n$ consiste en encontrar el camino más corto de un nodo s, llamado *nodo fuente*, al resto de los nodos $i \in N \setminus \{s\}$ de la red.

Hasta el momento se han estudiado diferentes tipos del problema del camino más corto:

1. Descubrir el camino más corto de un nodo a todos los demás cuando las longitudes de arco no son negativas.
2. Hallar el camino más corto de un nodo a todos los demás para redes con longitudes

de arco arbitrarias.

3. Encontrar el camino más corto de cada nodo al resto de nodos de la red.

Los dos primeros tipos se conocen como problema del camino más corto de fuente única, mientras que el tercer tipo se conoce como problema del camino más corto de todos los pares; que consiste en determinar la distancia más corta que hay entre cada par de nodos de una red.

2.1. Formulación del problema del camino más corto

El problema del camino más corto consiste en determinar para cada nodo $i \in N$, $i \neq s$, un camino dirigido de longitud mínima desde el nodo s, que es el nodo fuente de la red, al nodo i. Por otra parte, podemos pensar el problema como enviar 1 unidad de flujo de la forma más económica desde el nodo s a cada uno de los nodos en $N \setminus \{s\}$.

Sea una red dirigida $G = (N, A)$, siendo N el conjunto de nodos y A el conjunto de arcos que forman la red, con un coste de arco c_{ij} [1] asociado a cada arco $(i, j) \in A$. Sea $A(i)$ la lista de adyacencia del nodo i y sea $C = \max\{c_{ij} : (i, j) \in A\}$. Para calcular el coste de un camino dirigido sumamos los costes de los arcos que lo forman. Todo esto da lugar a la siguiente formulación en forma matricial del problema del camino más corto:

$$\begin{aligned} \text{mín} \quad & c^T \cdot f \\ \text{sujeto a} \quad & A_{n\times m} \cdot f = e \\ & I_{m\times m} \cdot f \geq 0 \end{aligned}$$

donde $A_{n\times m}$ [2] es la matriz de incidencia nodo-arco asociada al grafo en el que está definido el problema y e_s=1, e_n=-1 , con $i \in N \setminus \{s\}$.

[1] Utilizaremos *coste* y *longitud* indistintivamente.

[2] m denota el número de aristas

Además, podemos formular el problema del camino más corto en términos de flujos y costes para cada arista $(i,j) \in A$ de la siguiente forma:

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

$$\text{sujeto a} \quad \sum_{\{j:(i,j)\in A\}} x_{ij} - \sum_{\{j:(i,j)\in A\}} x_{ji} = \begin{cases} -1 & \text{, si } i \text{ es el nodo inicial} \\ 1 & \text{, si } i \text{ es el nodo terminal} \\ 0 & \text{, en otro caso} \end{cases}$$

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

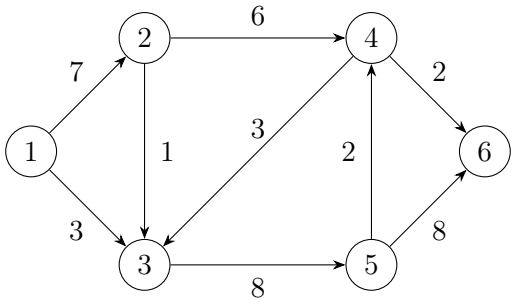
Haremos los siguientes supuestos para el estudio del problema del camino más corto:

1. Supuesto de parámetros enteros: todas las longitudes de arco son números enteros. Los algoritmos cuyo límite de complejidad depende de C, donde C es la mayor longitud de arco de la red, asumen que los datos son enteros. Por otra parte, siempre podemos multiplicar por un número suficientemente grande para transformar un número racional en entero. Además, necesitamos convertir los números en racionales para poder introducirlos en un ordenador, con lo cual este supuesto no es una restricción en la práctica.

2. Existe un camino dirigido desde el nodo s a los demás nodos de la red. Para cumplir esta condición, siempre podemos añadir un arco "ficticio" (s,j) con un coste muy alto para cada nodo j que no está conectado con s a través de un camino dirigido. En este caso, cuando un arco ficticio forme parte de alguna solución, nos estará indicando que en el grafo original no existía ningún camino entre los dos nodos unidos por dicho arco.

3. La red no contiene ciclos negativos. En caso de tener ciclos negativos en nuestra red, el problema tendría solución ilimitada porque a través de un ciclo de longitud total negativa podemos enviar una cantidad infinita de flujo (es decir, la función objetivo decrecería en cada iteración). El problema del camino más corto con ciclos negativos es difícil de resolver, ya que se trata de un problema NP-completo y es posible que no existan algoritmos polinomiales para hallar su solución. Cuando tenemos un problema del camino más corto con un ciclo negativo, el camino dirigido de menor coste puede pasar por un ciclo negativo infinitas veces, ya que en cada paso por el ciclo, se reduce el coste del camino. Para controlar esto, describiremos algoritmos que detecten la presencia de estos ciclos.

4. La red es dirigida. En caso contrario, podemos transformar la red en dirigida añadiendo un arco en cada dirección. Nótese que si los costes son negativos, se produciría un ciclo con longitud total negativa.



Este grafo es un ejemplo que cumple los cuatro supuestos (podemos tomar $s = 1$)

2.2. Símil físico del problema del camino más corto

Debido a la sencilla estructura del problema del camino más corto, se han podido desarrollar multitud de algoritmos de gran eficiencia computacional.

Vamos a considerar ahora un problema del camino más corto entre dos nodos s y t. Intentamos modelizar el problema con una cuerda de la siguiente manera: cogemos una cuerda y le hacemos nudos, cada nudo representará un nodo de nuestra red. Denotaremos por c_{ij} a la longitud de cuerda que hay entre los nodos i y j. Suponiendo que la cuerda no se puede estirar, cogemos en una mano el nudo que representa el nodo s y en la otra el que representa el nodo i. Cuando separamos las manos, vemos que hay uno o más trozos de cuerda que están tensos, que son los que representan los caminos más cortos de s a i.

Del anterior modelo, obtenemos las siguientes ideas del problema del camino más corto:

- Al estar la cuerda tensa para cualquier arco en un camino más corto, la distancia del camino más corto entre dos nodos sucesivos i y j será la longitud de cuerda que hay entre ellos, es decir, c_{ij}.

- Si conocemos la distancia más corta del nodo fuente, s, al nodo i, se cumple que la distancia de s a j es igual a la distancia de s a i más c_{ij}, cuando los nodos i y j están conectados en la red. La distancia puede ser mayor si la cuerda entre i y j no está tensa.

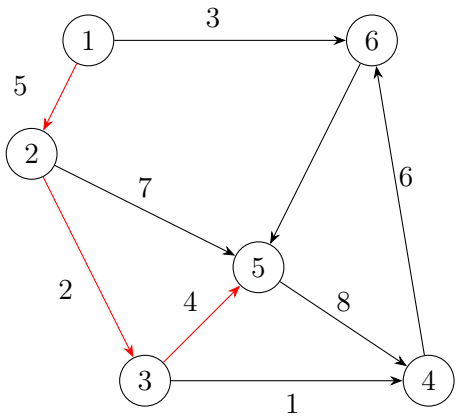
En general, todos los problemas de flujo de red expresados como problemas de minimización tienen un problema "dual" asociado, que es un *problema de maximización*. Al resolver uno de ellos, también resolvemos el otro.

2.3. Árbol de caminos más cortos

En el problema del camino más corto, nuestro objetivo es determinar un camino más corto desde el nodo fuente, s, a los $n-1$ nodos restantes de la red, siendo n el número total de nodos. En un principio, podemos pensar que un límite superior para almacenar estos caminos sería $(n-1)^2$, ya que cada camino puede contener como máximo $n-1$ arcos. Pero esto no es necesario porque, como mostramos a continuación, se puede encontrar un árbol externo dirigido desde el nodo fuente con la propiedad de que el único camino de s a cada nodo $i \in N \setminus \{s\}$ es el camino más corto a ese nodo. Por tanto, como un árbol con n nodos tiene como mucho $n-1$ arcos, este será el número que necesitaremos para almacenar todos los caminos.

Ejemplo 2.2. *En el siguiente grafo queremos encontrar el camino más corto entre los nodos 1 y 5.*

Si calculamos todos los caminos posibles para ir del nodo 1 al 5, el que tiene menor coste será el camino $1-2-3-5$ *con un coste* $c_{15} = 11$



La existencia del árbol del camino más corto se debe a la siguiente propiedad:

Proposición 2.3. *Si el camino $s = i_1 - i_2 - \cdots - i_h = k$ es un camino más corto desde s a k, entonces para cada $q = 2, 3, \ldots, h-1$, el camino $s = i_1 - i_2 - \cdots - i_q$ es el camino más corto del nodo origen, s, al nodo q.*

Para cada $i \in N \setminus \{s\}$, denotamos por $d(i)$ a la distancia del camino más cortos de s a i. La proposición 2.3 implica que P es un camino más corto de s a algún nodo j si, y sólo si, $d(j) = d(i) + c_{ij}$ para cada arco $(i, j) \in P$.

Establecemos ahora el siguiente resultado:

Proposición 2.4. *Sea P el camino más corto de s a k, y sea $s = i_1 - i_2 - \cdots - i_h = k$ la secuencia de nodos en P, entonces:*

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

donde $d(i_1) = 0$. También se cumple que $d(j) - d(i) = c_{ij}$ para cada arco $(i, j) \in P$.

Usando esta igualdad vemos que:

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

Como consecuencia, P será un camino dirigido del nodo origen al nodo k de longitud $d(k)$, y será el camino más corto al nodo k. Introducimos así la siguiente propiedad:

Proposición 2.5. *Si $d(\cdot)$ denota las distancias de caminos más cortos. Entonces, un camino dirigido P desde el nodo origen, s, al nodo j, $j \in N \setminus \{s\}$, es el camino más corto si y solo si $d(j) = d(i) + c_{ij}$ para cada arco $(i, j) \in P$.*

Esta propiedad es inmediata e implica que siempre existe un camino más corto desde s a todos los demás nodos que satisfacen que para cada arco (i, j) en el camino, $d(j) = d(i)+c_{ij}$.

Capítulo 3

Definición, análisis y tipos de algoritmos

En este capítulo hablaremos de qué es un algoritmo y los diferentes tipos que existen, que son una parte fundamental en el campo de los problemas computacionales ya que sirven para resolver un determinado tipo de problema. Un **algoritmo** es un procedimiento paso a paso que resuelve un problema. Los problemas pueden ser subproblemas unos de otros: por ejemplo, no solo define un problema el conjunto de problemas del camino más corto, sino que la clase de los problemas del camino más corto con longitudes no negativas, también lo define. Definimos una **instancia** como un caso de un problema, donde todos los parámetros del problema están determinados. Para definir una instancia del problema del camino más corto, necesitaríamos saber cuál es la topología de la red $G = (N, A)$, los nodos de origen y destino, y los costes de cada arco, los c_{ij}. Entonces, diremos que un algoritmo resuelve un cierto problema P si devuelve una solución para cada instancia de P. Como resolver un problema requiere la solución de un modelo de optimización que puede tener muchas variables, ecuaciones y desigualdades, utilizaremos un ordenador para resolverlo. Por tanto, tendremos que ser capaces de implementar los pasos de un algoritmo en un programa de ordenador. Aunque los matemáticos chinos del siglo III a.C. habían ideado algoritmos para resolver pequeñas ecuaciones, no fue hasta 1970 que comenzaron a estudiar el concepto de eficiencia de los algoritmos. Esta materia, denominada **teoría de la complejidad computacional**, proporciona diferentes maneras de medir el trabajo de los algoritmos contando las operaciones elementales que realiza. Para medir la eficiencia de un algoritmo, contaremos el tiempo que tarda debido a que el tiempo es un recurso informático dominante.

3.1. Análisis de complejidad de un algoritmo

3.1.1. Diferentes medidas de complejidad

Los pasos que suele realizar un algoritmo son los siguientes:

- Pasos de asignación: como asignar un valor a una variable.
- Pasos aritméticos: consiste en hacer una operación, es decir, una suma, resta, etc.
- Pasos lógicos: como, por ejemplo, comparar dos números.

Como cabe esperar, el número total de pasos que realiza el algoritmo es la suma de los pasos que realiza, y diferirá de una instancia a otra del problema. Esta variación de los pasos entre instancias de un problema es lo que lleva a medir el rendimiento de un algoritmo. Existen tres formar de medirlo a través de diferentes análisis:

1. Análisis empírico: su objetivo es predecir cómo se comportan los algoritmos en la práctica. Necesitamos escribir un programa de ordenador para el algoritmo y probamos el rendimiento del programa en algunos tipos de problemas. Algunos inconvenientes de este análisis son que el rendimiento del algoritmo dependerá del lenguaje de programación, el compilador, las habilidades que tenga el programador y el ordenador que vamos a utilizar. Además, necesita demasiado tiempo y suele no ser concluyente. A pesar de estas desventajas, el análisis empírico es muy importante para ver qué algoritmos utilizar en la práctica.
2. Análisis de caso promedio: este análisis intenta calcular el número de pasos esperados por el algoritmo. Para ello, seleccionaremos una distribución de probabilidad para las instancias del problema y derivamos tiempos de ejecución asintóticos para el algoritmo. El problema que presenta este tipo de análisis es que depende de la distribución de probabiliddad elegida para representar los diferentes casos o instancias del problema, ya que distintas elecciones conducen a distintas soluciones. Además, es difícil saber elegir la distribución de probabilidad apropiada. Otro inconveniente es que utiliza matemáticas complicadas, incluso para casos simples. La predicción del rendimiento de un algoritmo, basada en este análisis, está hecha para cuando necesitamos resolver muchos casos de problemas.

3. Análisis del peor caso: proporciona cotas superiores en el número de pasos de un algoritmo en cualquier instancia del problema. Lo que haremos será contar el mayor número posible de pasos. Uno de los grandes inconvenientes de este análisis es que permite determinar el rendimiento del algoritmo por ciertas instancias "patológicas".

El último análisis mencionado evita muchos de los problemas que tienen los otros dos, ya que no depende del entorno informático, nos garantiza el tiempo y los pasos que necesita un algoritmo, es fácil de realizar y proporciona pruebas concluyentes de que un algoritmo es mejor que otro en el peor caso. A pesar de ser un criterio "conservador", el análisis del peor caso es el más utilizado para medir el rendimiento algorítmico.

3.1.2. Tamaño del problema

Tomaremos el tamaño de un problema como medida de complejidad para conocer el rendimiento de un algoritmo porque el tiempo de ejecución que requiere para dar solución a un problema suele depender de su tamaño. El tamaño de un problema de red depende de cómo esté explicitado el problema. Si queremos representar en la lista de adyacencia la red, el tamaño del problema es el número de bits que necesitamos para almacenar dicha representación. Sea C el mayor coste de arco de la red y U la mayor capacidad de arco, se requieren sobre $n \log n + m \log m + m \log C + m \log U$ bits, ya que la lista de adyacencia almacena un bit para cada nodo y arco, y un elemento de datos para cada coeficiente y cada capacidad de arco. Como $m \leq n^2$, $\log m \leq \log n^2 = 2 \log n$, utilizaremos la notación *"Big O"* para hablar del tamaño de un problema. Se suele expresar el tiempo de ejecución de un algoritmo en función de los parámetros de red: n, m, C, U.

3.1.3. Complejidad en el peor caso

Es habitual que cuánto más grande sea el problema, mayor tiempo de resolución, y problemas del mismo tamaño pueden producir distintos tiempos de ejecución porque los datos son diferentes.

Definición 3.1. Una **función de complejidad** de tiempo, f, para un algoritmo es una función del tamaño del problema que especifica la mayor cantidad de tiempo que necesita el algoritmo para resolver cualquier instancia de un problema de ese tamaño.

Ejemplo 3.2. Sea $f = cnm$, con $c \geq 0$. El tiempo de ejecución necesario para resolver cualquier problema con n nodos y m arcos es, como mucho, cnm.

La función de complejidad muestra como el rendimiento varía en función de la medida de los datos del problema. Por lo tanto, nos referiremos a esta función como **complejidad del peor caso del algoritmo** o **límite del peor caso** porque nos da un límite superior del tiempo que necesita el algoritmo.

3.1.4. Tipos de notaciones

1. Notación *Big O*
   Como la función de complejidad depende de las constantes, podemos ver que ordenadores diferentes realizan operaciones diferentes, por lo que no podríamos comparar dos algoritmos. Para evitar estos problemas, utilizamos la notación *Big O* e ignoramos las constantes. Con esto, podemos ver el crecimiento asintótico del tiempo de ejecución, ya que la complejidad de un algoritmo determina una cota superior del tiempo de ejecución para grandes valores de n, donde n es el tamaño del problema. El término más grande en el tiempo de ejecución se representa con *Big O* debido a que, para n suficientemente grande, los términos más pequeños se vuelven insignificantes. Además, al ignorar las constantes, estamos suponiendo que todas las operaciones elementales cuestan el mismo tiempo, y podemos resumir el tiempo de ejecución de un algoritmo en función del número de operaciones elementales que realiza.

   Definición 3.3. Se dice que un algoritmo se ejecuta en $O(f(n))$ tiempo si $\exists c, n_0$ tal que el tiempo que tarda el algoritmo es como máximo $cf(n)$ para todo $n \geq n_0$.

2. Notación *Big* Ω
   Contraria a la notación *Big O*, la notación *Big* Ω especifica un límite inferior en el tiempo de ejecución.

   Definición 3.4. Se dice que un algoritmo es $\Omega(f(n))$ si $\exists c', n_0$ tal que el algoritmo toma al menos $c'f(n)$ tiempo en alguna instancia de problema, para todo $n \geq n_0$.

   Si un algoritmo se ejecuta en $O(f(n))$ tiempo, cada instancia del problema de tamaño n toma como máximo $cf(n)$ tiempo para una constante c. Por lo contrario, si un algoritmo se ejecuta en $\Omega(f(n))$ tiempo, alguna instancia de tamaño n toma al menos $c'f(n)$ tiempo para una constante c'.

3. Notación *Big* θ
   La notación *Big* θ determina un límite inferior y superior en el rendimiento de un algoritmo. Entonces,

   Definición 3.5. Diremos que un algoritmo es $\theta(f(n))$ si el algoritmo es $O(f(n))$ y $\Omega(f(n))$.

3.1.5. Algoritmos de tiempo polinomial y exponencial

Para empezar, definimos el **supuesto de similitud** como la forma de considerar que C y U están acotadas polinomialmente en n, esto quiere decir que $C = O(nk)$ y $U = O(nk)$ para cierta constante k. En esta sección vamos a analizar si un algoritmo es "bueno" o "malo".

Definición 3.6. Se dice que un algoritmo es un **algoritmo de tiempo polinomial** si su complejidad en el peor caso está acotada por una función polinomial que depende de los datos del problema.

Diremos que es un algoritmo "bueno" y algunos ejemplos son $O(nm), O(m+n\log C), \ldots$ Además, diremos que un algoritmo de tiempo polinomial es fuerte si su tiempo de ejecución está acotado por una función polinomial que sólo depende de n y m ($O(nm)$), y se dirá que es débil en otro caso, por ejemplo, si depende de C o U ($O(m + n\log C)$, nótese que el logaritmo crece más lento que n).

Definición 3.7. Un algoritmo es un **algoritmo de tiempo exponencial** si no admite una cota superior polinomial para su tiempo de ejecución en el peor caso.

Definición 3.8. Un algoritmo será un **algoritmo de tiempo pseudopolinomial** si su tiempo de ejecución está acotado polinomialmente en n, m, C, U.

Algunos ejemplos son $O(m + nC)$ y $O(mC)$. Los algoritmos de tiempo pseudopolinomial se transforman en algoritmos polinomiales si el problema satisface el supuesto de similitud. Aunque normalmente un algoritmo de tiempo polinomial es superior a cualquier algoritmo de tiempo exponencial, esto no pasa, por ejemplo, con el *"algoritmo del elipsoide"* de Khachiyan, que es polinomial, y con el *"Método Símplex"*, que es exponencial. El carácter exponencial del segundo se debe a su mal comportamiento en instancias patológicas que no suelen aparecer en aplicaciones prácticas, y por esto el rendimiento del símplex es mucho mejor que el algoritmo del elipsoide. En lo que respecta a la teoría de complejidad, lo que queremos es encontrar algoritmos de tiempo polinomial y elegir el de menor

tasa de crecimiento posible. Finalmente, decir que para muchos problemas combinatorios importantes no se han podido desarrollar algoritmos de tiempo polinomial. Sin embargo, se demostró que la mayoría de estos problemas pertenecen a la clase de los llamados problemas NP-completos, que son equivalentes en el sentido de que si existe un algoritmo de tiempo polinomial para un problema, existe un algoritmo de tiempo polinomial para cualquier otro problema NP-completo.

3.1.6. Funciones potenciales y complejidad amortizada

Debemos observar que si queremos acotar el tiempo de ejecución de un algoritmo, debemos limitar el tiempo de ejecución de cada una de sus operaciones elementales. Para hacer esto, primero, obtendríamos un límite en el número de pasos por operación, obtendríamos un límite en el número de operaciones y, después, los multiplicaríamos. En caso de que esto sea difícil de hacer, usaremos un análisis más general que nos permita obtener un límite "más estricto" en el tiempo de ejecución de la operación. Intentaríamos limitar el número de pasos en todas las ejecuciones de estas operaciones. Para ello, utilizaremos un método de función potencial. Las **técnicas de función potencial** nos permiten conocer la complejidad de un algoritmo mediante el análisis de los efectos de distintas operaciones en una función definida de forma adecuada. Este análisis se relaciona con la complejidad amortizada.

Definición 3.9. Se dice que una operación es de **complejidad amortizada** $O(f(n))$ si el tiempo para realizar una secuencia de k operaciones es $O(kf(n))$ para k suficientemente grande.

Podemos definir la complejidad amortizada como la complejidad "promedio" del peor caso. De esta forma el total será una cota superior en el número de pasos que necesita el algoritmo.

3.2. Desarrollo de algoritmos de tiempo polinomial

Para conseguir algoritmos para problemas de flujo en redes, se suelen utilizar cuatro enfoques importantes: enfoque de mejora geométrica, enfoque de escala, enfoque de programación dinámica y búsqueda binaria.

3.2.1. Enfoque de mejora geométrica

Lo que intenta expresar este enfoque es que un algoritmo se ejecuta en tiempo polinomial si, en cada iteración, se mejora el valor de la función objetivo. Sea H la diferencia entre el máximo y el mínimo de la función objetivo, para la mayoría de los problemas de red, H será función de n, m, C y U. Además, supondremos que el valor óptimo de la función objetivo es un número entero.

Teorema 3.10. *Sea z^k el valor de la función objetivo de alguna solución de un problema de minimización en la k-ésima iteración de un algoritmo y z^* el valor mínimo de la función objetivo. Supongamos también que el algoritmo garantiza que, para cada iteración k, se cumple:*

$$(z^k - z^{k+1}) \geq \alpha(z^k - z^*) \tag{3.1}$$

Entonces, el algoritmo termina en $O(\frac{logH}{\alpha})$ iteraciones.

Demostración. El valor $(z^k - z^*)$ representa la máxima mejora en el valor de la función objetivo después de la k-ésima iteración. Consideremos ahora una secuencia consecutiva de $\frac{2}{\alpha}$ iteraciones a partir de la iteración k. Si el valor de la función objetivo mejora en cada iteración del algoritmo, al menos, en $\frac{z^k - z^*}{2}$ unidades, entonces el algoritmo devolvería una solución óptima en estas $\frac{2}{\alpha}$ iteraciones. Por otra parte, supongamos que, en alguna iteración $q+1$, el algoritmo mejora el valor de la función objetivo en menos de $\frac{z^k - z^*}{2}$ unidades. Es decir,

$$z^q - z^{q+1} \leq \frac{\alpha(z^k - z^*)}{2} \tag{3.2}$$

La desigualdad 3.1 implica que :

$$\alpha(z^q - z^*) \leq z^q - z^{q+1} \tag{3.3}$$

y las desigualdades 3.2 y 3.3 implican que:

$$(z^q - z^*) \leq \frac{(z^k - z^*)}{2} \tag{3.4}$$

Por lo que vemos que el algoritmo ha reducido la mejora total posible $(z^k - z^*)$ en, al menos, 2. Por tanto, dentro de $\frac{2}{\alpha}$ iteraciones consecutivas, el algoritmo obtiene la solución óptima o reduce la mejora total posible en un factor de, al menos, 2. Como H es la máxima mejora posible y cada valor de la función objetivo es un número entero, el algoritmo debe terminar en $O(\frac{logH}{2})$ iteraciones. [2] □

Podemos resumir el enfoque de mejora geométrica con la afirmación : *"los algoritmos de red que tienen una tasa de convergencia geométrica son algoritmos de tiempo polinomial"*. Para desarrollar algoritmos de tiempo polinomial usando este enfoque, lo que hacemos es buscar técnicas de mejora local que produzcan grandes mejoras en la función objetivo en cada iteración.

3.2.2. Enfoque de escala

Con el tiempo, se han usado muchos métodos de escalado para desarollar algoritmos polinomiales para muchos problemas de optimización, combinatoria y de red. Para los problemas que cumplen el supuesto de similitud, los algoritmos de este tipo consiguen el mejor tiempo de ejecución en el peor caso para casi todos los problemas de optimización en redes. La forma más sencilla de escalado es el **escalado de bits**, que consiste en representar los datos con números binarios y resolver un problema P como una sucesión de problemas $P_1, \ldots, P_k$. El problema P_1 aproxima los datos al primer bit más significativo, el problema P_2 a los dos primeros bits más significativos, y así sucesivamente, hasta que $P = P_k$. Por otra parte, para cada $i = 2, \ldots, k$, utilizamos la solución óptima de P_{i-1} como solución inicial del problema P_i.

Propiedad 3.11. *La capacidad de un arco en P_k es el doble que en P_{k-1} más 0 ó 1.*

El enfoque de escala resuelve bien estos problemas porque:

1. P_1 suele ser fácil de resolver.

2. La solución óptima de P_{i-1} es una buena solución inicial para el problema P_i, y esto se debe a que estos dos problemas son parecidos.

3. El número de problemas de reoptimización que resolvemos es $O(\log C)$ ó $O(\log U)$. Con esto vemos que para llevar a cabo este enfoque, la reoptimización debe ser más eficiente que la optimización.

3.2.3. Enfoque de programación dinámica

Definiremos la programación dinámica como un enfoque de "llenado de tablas" donde completamos recursivamente las entradas de un cuadro de dos dimensiones.

3.2.4. Búsqueda binaria

La búsqueda binaria es una técnica para conseguir algoritmos polinomiales para muchos problemas de optimización en redes. Este método se usa para encontrar una solución que cumpla las condiciones requeridas entre un conjunto de soluciones factibles y lo que hace es eliminar, en cada paso, un porcentaje fijo del conjunto de soluciones hasta que el conjunto es tan pequeño que todas las soluciones factibles son una solución que cumple las propiedades que queremos. Veamos qué es la búsqueda binaria con el siguiente ejemplo:

Ejemplo 3.12. *Supongamos una función continua $f(x)$ que cumple que $f(0) < 0$ y $f(1) > 0$. Queremos encontrar un intervalo de tamaño ε que contenga el cero de esa función, esto es, un valor x tal que $f(x) = 0$. El primer paso es dividir el intervalo inicial a la mitad, en este caso $[0, 1]$, que sabemos que contiene un cero. Evaluamos la función en $x = 0.5$ y vemos si $f(0.5) = 0$, $f(0.5) < 0$ ó $f(0.5) > 0$. Si sucede el primer caso, $x = 0.5$ sería el cero del intervalo y terminaríamos el proceso. En el segundo caso, como $f(1) > 0$, sabemos que el intervalo $[0.5, 1]$ contiene un cero, ya que estamos suponiendo que f es continua. Análogamente haríamos con el tercer caso, en el cual el cero estaría en el intervalo $[0, 0.5]$. En los dos últimos casos, el punto nuevo con el que probaremos será, otra vez, con el punto medio, es decir con $x = 0.75$ o con $x = 0.25$, respectivamente. Repetimos el proceso hasta que el tamaño del intervalo sea menor que ε. Este método termina en $O(\log \frac{1}{\varepsilon})$.*

Normalmente, usaremos esta técnica para buscar el valor deseado de un parámetro dentro de un intervalo de posibles valores. También existe una versión generalizada que nos permite buscar valores de múltiples parámetros. El proceso que acabamos de ver en el anterior ejemplo se corresponde con el **método de dicotomía** que vimos en el grado.

3.3. Tipos de algoritmos

3.3.1. Algoritmos de búsqueda

Estos algoritmos son técnicas de grafos que intentan encontrar todos los nodos de una red que cumplan una cierta propiedad. Los algoritmos de búsqueda son útiles para:

- Encontrar todos los nodos que son accesbiles desde un determinado nodo en una red por un camino dirigido.
- Encontrar todos los nodos en una red que puedan llegar mediante un camino dirigido a un nodo i.

- Identificar los nodos de una red que están conectados y determinar si es una red bipartita o no.

- Identificar en una red un ciclo dirigido y, si la red es acíclica, reordenar los nodos $1, 2, \ldots, n$ para que $i < j$ para cada arco $(i, j) \in A$.

Para expresar las nociones básicas de estos algoritmos, supondremos que queremos encontrar todos los nodos que son accesibles mediante caminos dirigidos desde un nodo s, llamado nodo fuente, en una red $G = (N, A)$. Un algoritmo de búsqueda comienza desde el nodo fuente e identifica un número de nodos que son accesibles desde él. En cada ejecución, el algoritmo designa todos los nodos de una red como si estuvieran en uno de los dos estados siguientes: "marcado" o "no marcado". Los nodos marcados son accesibles desde el nodo fuente, y aún no se determinó el estado de los nodos sin marcar. Nótese que si el nodo i está marcado, la red contiene el arco (i, j) y el nodo j no está marcado, podemos marcar también este nodo, ya que es accesible desde el nodo s mediante un camino dirigido hasta el nodo i más el arco (i, j). En este caso, de estar marcado i y no j, llamaremos al arco (i, j) admisible. En caso contrario, lo llamaremos inadmisible. Cuando el algoritmo marca un nuevo nodo j analizando un arco admisible (i, j), decimos que el nodo i es un predecesor del nodo j ($pred(j) = i$). El algoritmo termina cuando la red no contiene arcos admisibles. El algoritmo de búsqueda va examinando los nodos marcados en un cierto orden; el orden de entrada (i) es el orden del nodo i en el recorrido. En la descripción de este tipo de algoritmos, $LIST$ denota el conjunto de nodos marcados que aún tiene que examinar el algoritmo ya que alguno de los arcos admisibles podrían derivar de ellos. Cuanto termina el algoritmo, ha señalado todos los nodos en G que son admisibles desde s mediante un camino dirigido. Los índices predecesores definen un árbol que llamaremos **árbol de búsqueda.** Usaremos la estructura de datos "current-arc", que definimos a continuación, para identificar los arcos admisibles en una red G y para implementar algoritmos de flujos máximo y flujo de coste mínimo. Con cada nodo i mantenemos la lista de adyacencua $A(i)$ de los arcos que salen de él. El algoritmo examina los nodos dependiendo de cómo hayamos organizado los arcos en las listas de adyacencia de cada arco $A(i)$. Para cada nodo i, definimos un arco actual (i, j), que es el siguiente candidato que vamos a examinar. Al prinicipio, el arco actual del nodo i es el primer arco en $A(i)$. El algoritmo de búsqueda examina la lista $A(i)$ de forma secuencial: en una etapa, si el arco actual es inadmisible, el algoritmo determina el siguiente arco en la lista como el arco actual. Cuando el algoritmo acaba de examinar la lista de arcos, determina que el nodo no tiene un arco admisible. Suponemos ahora que hemos ordenado los arcos en $A(i)$ en el orden creciente de sus nodos principales, esto quiere decir que si (i, j) y (i, k) son dos arcos consecutivos en $A(i)$, entonces $j < k$. El

algoritmo de búsqueda tarda, en total, $O(m+n) = O(m)$ tiempo ya que, como el algoritmo marca cualquier nodo como máximo una vez, este proceso teminará después de $2n$ veces, como mucho. Además, para cada nodo i, examinamos los arcos de $A(i)$, como máximo, una vez. Con lo cual, el algoritmo de búsqueda escanea un total de $\sum_{i \in N} |A(i)| = m$ arcos, por tanto termina en $O(m)$ tiempo. Existen dos estrategias de búsqueda fundamentales: búsqueda en amplitud y búsqueda en profundidad.

Búsqueda en amplitud o BFS (Breadth First Search)

Si mantenemos el conjunto $LIST$ como una cola, siempre seleccionamos los nodos del frente de $LIST$ y los agregamos al final. Así, el algoritmo seleccionará los nodos marcados en un orden de **primero en entrar, primero en salir**. Definimos la distancia de un nodo i como el número mínimo de arcos en un camino dirigido desde el nodo s al nodo i. Lo que hace este tipo de búsqueda es marcar primero los nodos con distancia 1, luego aquellos con distancia 2, y así sucesivamente. Entonces, esta versión de búsqueda se llama **búsqueda en amplitud** y el árbol resultante es un árbol de búsqueda en amplitud.

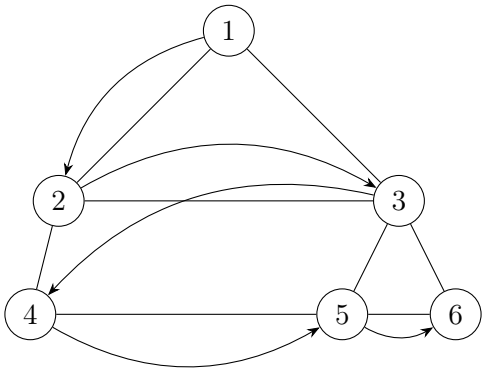


Figura 3.1: Árbol de búsqueda en amplitud

Presentamos a continuación una propiedad importante de este tipo de árbol.

Propiedad 3.13. *En el árbol de búsqueda de amplitud, el camino desde el nodo de origen a cualquier nodo i es el camino más corto, es decir, contiene el menor número de arcos entre todos los caminos que unen estos dos nodos.*

Búsqueda en profundidad o DFS (Depth First Search)

Ahora, mantenemos el conjunto $LIST$ como una pila, es decir, seleccionamos los primeros nodos de $LIST$ y los añadimos al principio, en lugar de al final, como hacíamos con el algoritmo de búsqueda en amplitud. Lo que hace en este caso el algoritmo de búsqueda es seleccionar el nodo marcado en un orden de **último en entrar, primero en salir**. Este algoritmo hace una prueba profunda intentando crear un camino lo más largo posible, y hace una copia de seguridad de un nodo para iniciar una nueva búsqueda cuando no marque ningún nuevo nodo desde el inicio del camino. Por eso llamamos a esta técnica de búsqueda, **búsqueda en profundidad** y al árbol resultante, árbol de búsqueda en profundidad.

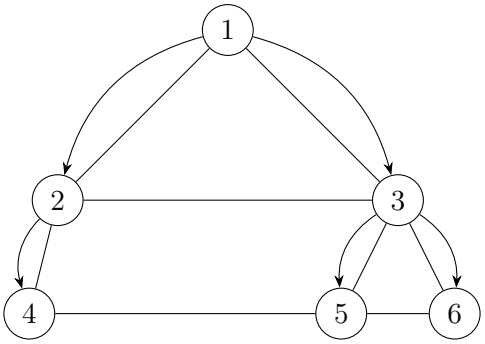


Figura 3.2: Árbol de búsqueda en profundidad

Al igual que antes, existe una propiedad importante del árbol de búsqueda en profundidad:

Proposición 3.14. *Si el nodo j es un descendiente del nodo i y $j \neq i$, entonces $orden(j) > orden(i)$. Además, todos los descendientes de cualquier nodo se ordenan consecutivamente.*

3.3.2. Algoritmos de descomposición de flujo

Cuando formulamos problemas de flujo en redes, podemos elegir entre: definir flujos en arcos o definir flujos en caminos y ciclos. En esta sección, desarrollaremos varias relaciones entre estas dos formulaciones alternativas. Cuando nos referimos a "flujo de arco", estamos hablando de un vector $x = \{x_{ij}\}$ que cumple lo siguiente:

$$\sum_{\{j:(i,j)\in A\}} x_{ij} - \sum_{\{j:(j,i)\in A\}} x_{ji} = -e(i) \quad \forall i \in N$$

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

donde $\sum_{i=1}^{n} e(i) = 0$, y donde el término $e(i)$ representa la entrada de flujo menos la salida de flujo del nodo i. Con esto, si $e(i) > 0$, entonces el flujo de entrada es mayor que el flujo que sale, y decimos que i es un **nodo con exceso**. En caso de ser el flujo de salida mayor que el flujo de entrada, $e(i) < 0$, diremos que i es un **nodo con déficit**. Finalmente, si el flujo que entra es igual al flujo que sale, entonces $e(i) = 0$, y diremos que i es un **nodo equilibrado**. La formulación de flujo de camino y ciclo comienza con una enumeración de todos los caminos dirigidos P entre cualquier par de nodos y todos los ciclos dirigidos W de la red. Las variables de decisión serán $f(P)$, que representa el flujo del camino P, y $f(W)$, el flujo del ciclo W. Denotamos por $\mathcal{P}$ al conjunto de caminos y por $\mathcal{W}$ al conjunto de ciclos. Nótese que cada conjunto de caminos y ciclos determina de forma única los flujos de arco de manera natural: el flujo x_{ij} en el arco (i,j) es igual a la suma de $f(P)$ y $f(W)$ para todos los caminos P y todos los ciclos W que contienen a ese arco. Formalizamos esto de la siguiente manera: $\delta_{ij}(P) = 1$ si el arco (i,j) pertenece al camino P, y 0 en otro caso. De la misma forma, $\delta_{ij}(W) = 1$ si el arco (i,j) está en el ciclo W y 0 en otro caso. De esta forma:

$$x_{ij} = \sum_{P\in\mathcal{P}} \delta_{ij}(P) f(P) + \sum_{W\in\mathcal{W}} \delta_{ij}(W) f(W)$$

Con el siguiente teorema, observamos que podemos descomponer cualquier flujo de arco en flujo de camino y ciclo.

Teorema 3.15 (Teorema de descomposición de flujo)**.** *Cada flujo de camino y ciclo tiene una representación única como flujos de arco no negativos. Asimismo, cada flujo de arco no negativo x se puede representar como un flujo de camino y ciclo (aunque no siempre de forma única) con las siguiente propiedades:*

1. *Todo camino dirigido con flujo positivo conecta un nodo en defecto con un nodo en exceso.*

2. *Como máximo, $n + m$ caminos y ciclos tienen un flujo distinto de cero. De estos, como máximo, m ciclos tienen un flujo distinto de 0.*

Demostración. Basándonos en observaciones previas, necesitamos establecer solamente la segunda afirmación. Damos una prueba algorítmica para mostrar cómo descomponer cualquier arco de flujo x en un camino y un ciclo de flujo. Supongamos que i_0 es un nodo en defecto. Entonces, algún arco (i_0, i_1) lleva un flujo positivo. Si i_1 es un nodo en exceso, nos detenemos; de lo contrario, algún otro arco (i_1, i_2) lleva un flujo positivo. Repetimos este proceso hasta que encontramos un nodo en exceso o volvemos a visitar un nodo previamente examinado. Uno de estos dos casos ocurrirá en n pasos. En el primer caso, obtenemos un camino dirigido P desde el nodo en defecto i_0 a algún nodo en exceso i_k, y en el último caso obtenemos un ciclo dirigido W. En cualquier caso, el camino o el ciclo consta únicamente de arcos con flujo positivo. Si obtenemos un camino dirigido, tenemos $f(P) = \text{mín}\{-e(i_0), e(i_k), min\{x_{ij} : (i,j) \in P\}\}$ y redefinimos $e(i_0) = e(i_0) + f(P)$, $e(i_k) = e(i_k) - f(P)$ y $x_{ij} = x_{ij} - f(P)$ para cada arco $(i,j) \in P$. Si obtenemos un ciclo dirigido W, ponemos $f(W) = min\{x_{ij} : (i,j) \in W\}$ y redefinimos $x_{ij} = x_{ij} - f(W)$ para cada $(i,j) \in W$. Repetimos este proceso con el problema redefinido hasta que todos los desequilibrios de nodos sean cero. Luego seleccionamos cualquier nodo con, al menos, un arco saliente con un flujo positivo como nodo inicial, y repetimos el procedimiento, que en este caso debe encontrar un ciclo dirigido. Terminamos cuando $x = 0$ para el problema redefinido. Claramente, el flujo original es la suma de los flujos en los caminos y ciclos identificados por este método. Nótese que cada vez que identificamos un camino dirigido, reducimos el exceso/déficit de algún nodo a cero o el flujo en algún arco a cero; y cada vez que identificamos un ciclo dirigido, reducimos el flujo en algún arco a cero. En conclusión, la representación de camino y ciclo del flujo x dado contiene como máximo $n + m$ caminos y ciclos dirigidos y , como máximo, m de estos son ciclos dirigidos. [2] □

Analizamos ahora cuál es el tiempo requerido por el algoritmo. Primero, construimos una lista de nodos en defecto. Mantenemos la lista como una lista doblemente enlazada (ver definición 1.16), de forma que tanto la selección de un elemento, como la adición o eliminación de un elemento, requieren $O(1)$ tiempo. El algoritmo va eliminando los nodos de la lista a medida que avanza y, cuando la lista queda vacía, lo inicializamos como el conjunto de arcos con flujo positivo. Otra operación básica del algoritmo es identificar, mediante la estructura de "current arc" (definida en la página 21), un arco con flujo positivo que sale de un nodo. Estos arcos se llamarán **arcos admisibles**. En cualquier iteración, el algoritmo necesita $O(n)$ tiempo más el tiempo que se gastó en la búsqueda de arcos para identificar aquellos que son admisibles. Debido a que los flujos de arco no aumentan, un arco que se considera inadmisible en una iteración, también será inadmisible en iteraciones posteriores. Como la estructura de datos de arco actual necesita un tiempo total de $O(m)$

en el escaneo del arco para identificar los arcos admisibles y el algoritmo realiza como máximo $n + m$ iteraciones, entonces el algoritmo de descomposición de flujo se ejecuta en un tiempo total de $O(m + (n + m)n) = O(nm)$.

El teorema 3.15 nos permite comparar dos soluciones de un problema de flujo de red y ver cómo podemos construír una solución a partir de otra a partir de simples operaciones.

3.3.3. Algoritmos de ajuste y corrección de etiquetas

Los enfoques algorítmicos para resolver problemas del camino más corto se clasifican en dos grupos:

1. Configuración de etiquetas.
2. Corrección de etiquetas.

Ambos enfoques son iterativos, asignan etiquetas de distancia a los nodos en cada paso. Estas etiquetas de distancia son estimaciones de límites superiores en las distancias de caminos más cortos. Los algoritmos que establecen etiquetas definen una etiqueta como permanente (óptima) en cada iteración. Por otra parte, los algoritmos de correción de etiquetas definen todas las etiquetas como temporales hasta el último paso, cuando todas se convierten en permanentes. Los enfoques se distinguen, por tanto, en la forma que actualizan las etiquetas de distancia de un paso a otro y en cómo "convergen" hacia las distancias de camino más corto. Los algoritmos de establecimiento de etiquetas se aplican solamente a problemas de camino más corto con redes acíclicas con longitudes de arco arbitrarias y a problemas de camino más corto con longitudes de arco no negativas. Sin embargo, los algoritmos de corrección de etiquetas son más generales y pueden aplicarse a todos los tipos de problemas, incluso a aquellos con longitudes totales negativas, por lo que ofrece más flexibilidad algorítmica. Sin embargo, los algoritmos de configuración de etiquetas son más eficientes. Podemos ver los algoritmos de etiquetas como casos particulares de algoritmos de correcicón de etiquetas.

3.4. Algoritmo para el problema del camino más corto en redes acíclicas

Como ya sabemos, una red es acíclica si no contiene ningún ciclo dirigido (véase la definición 1.10). En esta sección, veremos como resolver el problema del camino más corto

en tiempo $O(m)$ cuando la red es acíclica. Nótese que ningún algoritmo puede resolver el problema del camino más corto en redes acíclicas en un tiempo menor, ya que para resolverlo necesitaríamos examinar cada arco. Tengamos en cuenta que en una red acíclica $G = (N, A)$ siempre podemos establecer lo que se conoce como **orden topológico**, es decir, podemos numerar los nodos, en tiempo $O(m)$, de manera que $i < j$ para cada arco $(i, j) \in A$. Supongamos que calculamos cada $d(i)$, que es la distandia más corta del nodo s a los nodos $i = 1, 2, \ldots, k-1$. El orden topológico establece que todos los arcos dirigidos al nodo k proceden de un nodo i, donde $i \in \{1, 2, \ldots, k-1\}$. Como ya vimos, el camino más corto al nodo k es el camino más corto a uno de los nodos. Por tanto, para calcular el camino más corto al nodo k, lo que tenemos que hacer es calcular $min\{d(i) + c_{ik}\}$ para todos los arcos (i, k). Como para implementar el algoritmo necesitamos acceder a todos los arcos dirigidos a cada nodo, lo que haremos es utilizar la lista de adyacencia $A(i)$ de cada nodo $i \in N$. A continuación, decribimos los pasos del algoritmo:

Empezamos estableciendo $d(s) = 0$, donde s es el nodo fuente, y le damos un valor muy grande al resto de etiquetas de distancia. Analizamos, en orden topológico, cada nodo y, para cada nodo i, construimos la lista de adyacencia $A(i)$. Si para alguna arista $(i, j) \in A$ se cumple que $d(j) > d(i) + c_{ij}$, ponemos $d(j) = d(i) + c_{ij}$. Una vez que el algoritmo analiza todos los nodos una vez en el orden indicado, las etiquetas de distancia son óptimas.

Demostración 3.16. Demostramos por inducción que cuando el algoritmo examina un nodo, su etiqueta de distancia es óptima.
Supongamos que el algoritmo ha examinado los nodos $1, 2, \ldots, k$ y sus etiquetas de distancia son óptimas. A continuación, examinaremos el nodo $k+1$. Sea $s = i_1 - i_2 - \cdots - i_h - k + 1$ el camino más corto del nodo s al nodo k, entonces, $s = i_1 - i_2 - \cdots - i_h$ será el camino más corto de s al nodo i_h. Como tenemos los nodos ordenados en el orden topológico, $(i_h, k+1) \in A$ implica que $i_h \in 1, 2, \ldots, k$ y, por hipótesis de inducción, sabemos que la etiqueta de distancia del nodo i_h es igual a la longitud del camino $i_1 - i_2 - \cdots - i_h$, al examinar el nodo i_h, el algoritmo debe haber analizado el arco $(i_h, k+1)$ y establecer la etiqueta de distancia del nodo $k+1$ igual a la longitud del camino $i_1 - i_2 - \cdots - i_h - k + 1$. En consecuencia, cuando el algoritmo examina el nodo $k+1$, su etiqueta de distancia es óptima.

Establecemos ahora el siguiente resultado.

Teorema 3.17. *El algoritmo de alcance resuelve el problema del camino más corto en redes acíclicas en tiempo* $O(m)$

En esta sección, hemos visto como resolver un problema del camino más corto en caso de tener una red que es acíclica con el algoritmo más simple posible. Pero no podemos aplicar este algoritmo de un paso y cada arco exactamente una vez cuando la red contiene algún ciclo. Sin embargo, esta estrategia de alcance se puede utilizar para resolver cualquier problema del camino más corto con longitudes de arco no negativas, pero en este caso no tendríamos un orden establecido de los nodos y en cada paso tendríamos que investigar varios nodos para determinar de qué nodo llegar.

Capítulo 4

Algoritmos de configuración de etiquetas para el problema del camino más corto

Los **algoritmos de configuración de etiquetas** se utilizan para resolver el problema del camino más corto con longitudes de arco no negativas. Estos algoritmos asignan etiquetas de distancia temporales a los nodos y luego identifican, de forma iterativa, cuál es el camino más corto de un nodo a otro estableciendo una etiqueta de distancia permanente. El principal problema que estudiaremos del camino más corto determina el camino más corto desde un **nodo origen** a cada nodo restante del grafo. En este trabajo describiremos **el algoritmo de Dijkstra**, el **algoritmo de Dial** y el **algoritmo Radix-Heap**.

4.1. Algoritmo de Dijkstra

El **algoritmo de Dijkstra** fue desarrollado por el físico e informático holandés Edsger Wybe Dijkstra en 1959. Este algoritmo mantiene una etiqueta de distancia $d(i)$ para cada nodo $i \in N$, que es un límite superior en la longitud del camino más corto al nodo i. En cualquier paso intermedio, el algoritmo divide los nodos en dos grupos: los **nodos permanentes**, que son los que tienen una etiqueta ya permanente, y los **nodos temporales**, aquellos que aún tienen una etiqueta que puede ser actualizada. Para cualquier nodo temporal, la etiqueta de distancia es, como ya dijimos, un límite superior en la distancia de camino más corto a ese nodo.

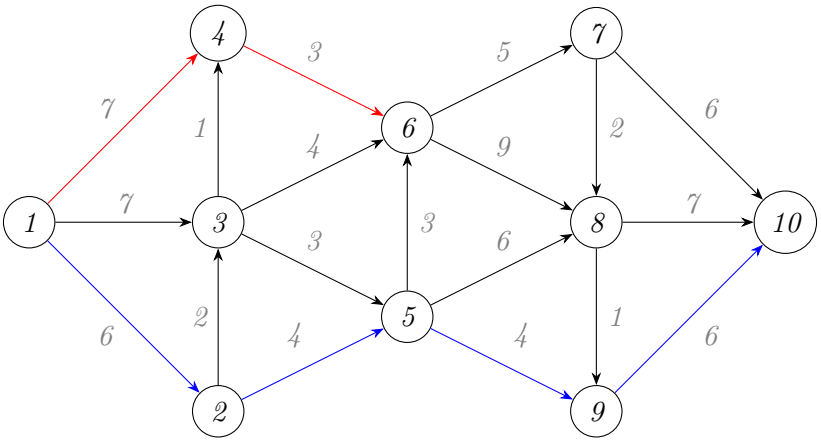
La idea básica del algoritmo es desplegarse desde el nodo s, que es el nodo origen, y etiquetar permanentemente los nodos restantes en el orden de sus distancias desde el nodo s. Describimos a continuación el algoritmo:

1. Inicialmente, le damos al nodo origen una etiqueta permanente de cero, y a cada uno de los demás nodos una etiqueta temporal igual a infinito.
2. En cada iteración, la etiqueta de un nodo i es su distancia más corta desde el nodo origen a lo largo de un camino cuyos **nodos internos** (los nodos $j \in N : j \neq s, j \neq i$) tienen etiquetas permanentes.
3. El algoritmo selecciona un nodo i con la etiqueta temporal mínima, lo hace permanente, y se extiende desde ese nodo. Es decir, escanea arcos en $A(i)$ para actualizar las etiquetas de distancia de los nodos adyacentes.
4. El algoritmo termina cuando se han designado todos los nodos como permanentes.

El algoritmo de Dijkstra mantiene un árbol externo dirigido T que parte del nodo origen y atraviesa los nodos con etiquetas de distancia finita. El algoritmo mantiene este árbol usando índices predecesores, es decir, si $(i, j) \in T$ entonces $pred(j) = i$. Además, mantiene la propiedad invariante de que cada arco (i, j) cumple que $d(j) = d(i) + c_{ij}$ con respecto a las etiquetas de distancia actuales. Por tanto, cuando estas representan distancias de camino más cortas, T es un **árbol de camino más corto** (debido a la propiedad 2.5).

Introducimos a continuación un ejemplo.

Ejemplo 4.1. *Dado el siguiente grafo, donde se muestran las distancias entre los nodos, queremos resolver el problema del camino más corto, siendo 1 el nodo origen. Para ello, utilizaremos el algoritmo de Dijkstra.*



Empezamos con el nodo origen, que es el nodo 1. Por tanto, etiquetamos ese nodo con $d(1) = (0, 1)$ [1] *de forma permanente, y el resto de la fila que corresponde al nodo 1 lo rellenamos con* $*$*. A continuación, examinamos los nodos adyacentes a 1, que son 2, 3 y 4, estableciendo* $d(2) = (6, 1)$*,* $d(3) = (7, 1)$ *y* $d(4) = (7, 1)$*. Entre estas tres etiquetas, la de menor distancia es* $d(2)$*, por tanto, el siguiente paso, será examinar los nodos que son adyacentes a 2 y actualizar sus etiquetas de distancia, es decir, obtendríamos* $d(3) = (7, 1)$*, ya que la distancia de 1 a 3 pasando por el nodo 2 es mayor (*$d(3) = (8, 2)$*). Mantenemos* $d(4) = (7, 1)$ *y, por último, obtendríamos* $d(5) = (10, 2)$*. Repetimos el procedimiento hasta examinar todos los nodos y obtendríamos la siguiente tabla:*

Nodo	Paso 1	Paso 2	Paso 3	Paso 4	Paso 5	Paso 6	Paso 7	Paso 8	Paso 9
1	(0,1)	*	*	*	*	*	*	*	*
2	(6,1)	(6,1)	*	*	*	*	*	*	*
3	(7,1)	(7,1)	(7,1)	*	*	*	*	*	*
4	(7,1)	(7,1)	(7,1)	(7,1)	*	*	*	*	*
5	-	(10,2)	(10,2)	(10,2)	(10,2)	*	*	*	*
6	-	-	(11,3)	(10,4)	(10,4)	(10,4)	*	*	*
7	-	-	-	-	-	(15,6)	(15,6)	*	*
8	-	-	-	-	(16,5)	(16,5)	(16,5)	(16,5)	*
9	-	-	-	-	(14,5)	(14,5)	(14,5)	(14,5)	*
10	-	-	-	-	-	-	(21,7)	(20,9)	(20,9)

Esta tabla representa el camino más corto desde el nodo 1 al resto de nodos del grafo. Por ejemplo, el camino más corto de 1 a 6 sería $1 - 4 - 6$ *(camino rojo) y de 1 a 10 sería* $1 - 2 - 5 - 9 - 10$ *(camino azul).*

Existen diferentes versiones del algoritmo de Dijkstra, pero en este libro solamente explicaremos la original.

4.1.1. Tiempo de ejecución del algoritmo de Dijkstra

Estudiaremos ahora la complejidad del algoritmo de Dijkstra en el peor de los casos. Para ello, podemos pensar el tiempo computacional para el algoritmo como el tiempo aisgnado a las siguientes operaciones básicas:

[1] La primera entrada de $d(i)$ es la menor distancia de S al nodo i. y la segunda el nodo predecesor

1. Selección de nodos: El algoritmo realiza esta operación n veces y cada una de esas operaciones necesita escanear cada nodo etiquetado temporalmente. Por tanto, el tiempo total de selección de nodos es $n + (n-1) + (n-2) + \cdots + 1 = O(n^2)$.

2. Actualización de las distancias: El algoritmo actualiza $|A(i)|$ veces la etiqueta de distancia del nodo i. Por tanto, el algoritmo realiza esta operación $\sum_{i \in N} |A(i)| = m$ veces. Como cada operación de actualización de distancia necesita $O(1)$ tiempo, el algoritmo necesitará $O(m)$ tiempo en total para actualizar todas las etiquetas de distancia.

Establecemos a continuación el siguiente resultado.

Teorema 4.2. *El algoritmo de Dijkstra resuelve el problema del camino más corto (con longitudes de arco no negativas) en tiempo $O(n^2)$.*

El límite establecido en el teorema 4.2 es el mejor posible para grafos completamente densos, es decir, $m = \Omega(n^2)$, pero se puede mejorar para grafos que no lo sean. Se ha intentado reducir el tiempo de selección de nodos sin aumentar el tiempo para actualizar las distancias y han surgido varias implementaciones del algoritmo que reducen el tiempo de ejecución del algoritmo o mejoran su complejidad en el peor caso. Describiremos la **implementación del Dial**, que es una buena implementación del algoritmo de Dijkstra en la práctica, y la **implementación de Radix-Heap**.

4.1.2. Implementación del algoritmo de Dijkstra en R

```
dijkstra<-function(A){
  #Preprocesado
  if (nrow(A)!=ncol(A)) stop("Matriz no cuadrada")
  nodos<-nrow(A)
  M<-sum(A,na.rm=TRUE)
  A[is.na(A)]<-M+1

  #Paso 1
  nodos<-nrow(A)
  X<-numeric()
  N<-1:nodos
  P<-rep(Inf,nodos)

  #Paso 2
  X<-c(X,1)
  XC<-setdiff(N,X)
  P[1]<-0
  P[XC]<-A[1,XC]
  E<-rep(1,nodos)

  #Paso 3
  while(length(X)!=nodos){
    i<-XC[which.min(P[XC])]
    X<-c(X,i)
```

```
25        XC<-setdiff(N,X)
26        for (j in XC){
27          if(P[j]>P[i]+A[i,j]){
28            P[j]<-P[i]+A[i,j]
29            E[j]<-i
30            }
31          }
32        }
33
34      P[P>M]<--1
35
36      camino<-c(nodos)
37      while(camino[1]!=1) camino<-c(E[camino[1]],camino)
38
39      coste<-P[nodos]
40
41      P<-rbind(1:nodos,P)
42      rownames(P)<-c("Nodo","Coste (Pi)")
43
44      E<-rbind(1:nodos,E)
45      rownames(E)<-c("Nodo","Predecesor (p)")
46
47      return(list("Coste de 1 a n"=coste,"Camino de 1 a n"=camino,"Costes
48      (Pi)"=P,"Etiquetas (p)"=E))
49      }
```

4.2. Algoritmo de Dial

La operación más costosa en el algoritmo de Dijkstra es la selección de nodo. Para mejorar el rendimiento del algoritmo, la implementación del Dial intenta reducir el tiempo de cálculo manteniendo las distancias de manera ordenada utilizando la siguiente propiedad.

Propiedad 4.3. *Las etiquetas de distancia que el algoritmo de Dijkstra designa como permanentes no son decrecientes.*

La propiedad anterior se debe a que el algoritmo etiqueta de manera permanente el nodo i que tiene una etiqueta temporal d(i) más pequeña y, mientras examina los arcos en $A(i)$ durante las operaciones de actualización de distancia, nunca disminuye la etiqueta de distancia de cualquier nodo etiquetado temporalmente, debido a que las longitues de arco son no negativas. Este algoritmo de Dial almacena nodos con etiquetas temporales finitas de manera ordenada. Recordemos que C es la mayor longitud de la red y, por tanto, nC es un límite superior en la etiqueta de distancia de cualquier nodo etiquetado de manera finita. El algoritmo define $nC + 1$ conjuntos, a los que llamaremos **depósitos** o **cubos**, numerados de 0 a nC. El depósito k almacena todos los nodos del grafo que tienen una etiqueta de distancia temporal igual a k. Representaremos el contenido del depósito k por *contenido*(k). En la operación de selección de nodos, escaneamos los depósitos numerados $0, 1, \ldots$ hasta que encontremos el primero no vacío. Sea k el primer depósito no vacío, cada nodo en *contenido*(k) tiene la etiqueta de distancia mínima. Iremos eliminando los

nodos de este depósito de uno en uno, diremos que están etiquetados permanentemente y escanearemos sus listas de adyacencia para actualizar las etiquetas de distancia de los nodos adyacentes. Cuando actualizamos la etiqueta de distancia de un nodo i de d_1 a d_2, movemos el nodo i de *contenido*(d_1) a *contenido*(d_2). En la siguiente operación de selección de nodos, volvemos a examinar los depósitos numerados $k+1, k+2, \ldots$ para seleccionar el siguiente que sea no vacío. Sabemos que los depósitos $0, 1, 2, \ldots, k$ estarán vacíos, debido a la propiedad 4.3, y el algoritmo no necesita examinarlos de nuevo. Almacenamos el contenido de cada *contenido*(k) como una lista doblemente enlazada (ver la definición 1.16) porque esta estructura de datos nos permite realizar cada una de las siguiente operaciones en tiempo $O(1)$:

1. Verificar si un depósito está vacío o no.
2. Eliminar un elemento de un depósito.
3. Agregar un elemento a un depósito.

Por tanto, el algoritmo necesitará un tiempo total $O(m)$ para realizar todas las actualizaciones de distancia. La operación más complicada en este procedimiento es analizar los $nC + 1$ depósitos durante la selección de nodos. En consecuencia, el tiempo de ejecución del algoritmo de Dial es $O(m + nC)$. Como $nC + 1$ es un número muy grande, la siguiente propiedad nos permite reducir el número de depósitos a $C + 1$.

Propiedad 4.4. *Si $d(i)$ es la etiqueta de distancia que el algoritmo designa como permanente al comienzo de una iteración, entonces al final de esa iteración, $d(j) \leq d(i) + C$ para cada nodo j etiquetado de forma finita.*

Esto se debe a que, por la propiedad 4.3, $d(l) \leq d(i)$ para cada nodo $l \in S$. Además, para cada $j \in S$, se tiene que $d(j) = d(l) + c_{lj}$ para algún nodo $l \in S$, debido a la propiedad de actualizaciones de distancia. Por tanto, $d(j) = d(l) + c_{lj} \leq d(i) + C$. Esto quiere decir que todas las etiquetas temporales finitas están entre $d(i)$ y $d(i)+C$. Por tanto, $C+1$ depósitos son suficientes para almacenar nodos con etiquetas de distancia temporales finitas. Por tanto, el algoritmo de Dial utiliza $C + 1$ depósitos numerados $0, 1, 2, \ldots$ que podemos ver ordenados en forma circular en la imagen 4.1.

Un nodo i que está etiquetado de manera temporal con la etiqueta de distancia $d(i)$ lo almacenamos en el cubo $d(i) \mod C + 1$. En consecuencia, el depósito k almacena nodos con etiquetas de distancia temporales $k, k + (C + 1), k + 2(C + 1), \ldots$ y así sucesivamente.

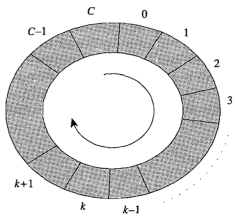


Figura 4.1: Algoritmo de Dial

Pero, en algún momento, este depósito solamente contendrá nodos con la misma etiqueta de distancia, debido a la propiedad 4.4. Esto implica que si el cubo k contiene un nodo con la etiqueta de distancia mínima, entonces el cubo $k+1, k+2, \ldots, C, 0, 1, 2, \ldots, k-1$ almacena nodos en valores crecientes de las etiquetas de distancia. Una posible desventaja del algoritmo de Dial en comparación con la implementación $O(n^2)$ original del algoritmo de Dijkstra es que necesita mucho espacio de almacenamiento cuando C es muy grande. Además, el tiempo de cálculo puede ser grande porque el algoritmo puede actualizarse hasta $n-1$ veces. El algoritmo es pseudopolinomial, ya que se ejecuta en tiempo $O(m+nC)$. Sin embargo, casi nunca alcanza este límite, por lo que el tiempo de ejecución del algoritmo de Dial es mucho mejor que el indicado por su complejidad en el peor caso.

4.3. Algoritmo de Radix-Heap

La implementación Radix-Heap del algoritmo de Dijkstra es una combinación entre la implementación original $O(n^2)$ y la implementación del Dial, que utiliza $nC+1$ depósitos. La implementación original del algoritmo de Dijkstra considera todos los nodos que están etiquetados de manera permanente juntos, es decir, los almacena todos en el mismo depósito. Por otra parte, el algoritmo de Dial utiliza muchos depósitos y almacena nodos con diferentes etiquetas de distancia en depósitos diferentes. La implementación que presentamos en esta sección adopta un punto medio y mejora estos dos métodos. Almacena etiquetas en varios depósitos, pero que no serán muchos. Por ejemplo, en vez de almacenar solo los nodos con una etiqueta temporal k en el k-ésimo depósito, podríamos almacenar etiquetas temporales de $100k$ a $100k+99$ en el depósito k. Las diferentes etiquetas temporales que se pueden almacenar en un cubo o depósito determinan su **rango** y la cardinalidad del rango se denomina **anchura**. Para el anterior ejemplo, el rango del depósito es $(100k, 100k+99)$ y su longitud es 100. El uso de estas anchuras de tamaño k permite reducir el número de depósitos necesarios por un factor de k, pero para encontrar la etiqueta de distancia más pequeña, tenemos que buscar todos los elementos en el depósito no vacío

que tenga el índice más pequeño. Además, si k es grande, solo necesitaremos un depósito, y el algoritmo resultante se reduce a la implementación original del algoritmo de Dijkstra. La **implementación con Radix-Heap** que consideramos a continuación utiliza diferentes anchuras y cambia los rangos de manera dinámica. En esta versión que presentamos se cumple que:

Proposición 4.5. *Introducimos las siguientes propiedades del algoritmo:*

1. *Las anchuras de los depósitos son* $1, 1, 2, 4, 8, 16, \ldots$, *de modo que el número de depósitos necesarios es solo* $O(log(nC))$.

2. *Modificamos los rangos de los cubos y reasignamos las etiquetas temporales de los nodos, de forma que siempre se almacena la etiqueta de distancia mínima en el depósito de anchura 1.*

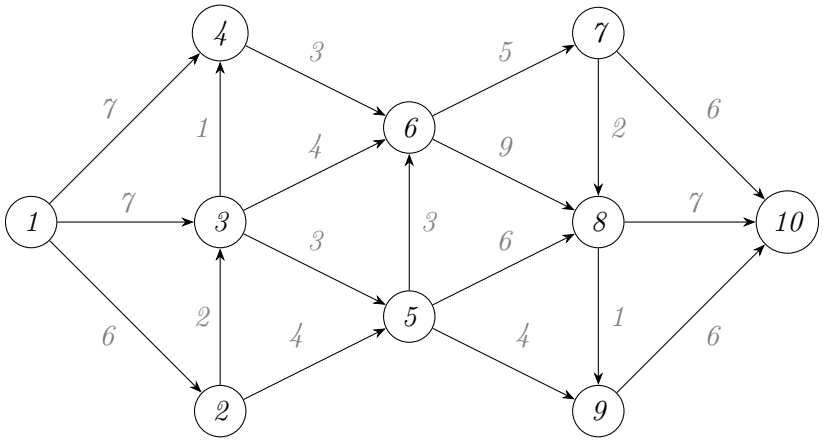
La primera propiedad de la proposición anterior nos permite mantener solamente $O(log(nC))$ depósitos y, por tanto, evita el inconveniente del *Dial* de usar demasiados cubos. La segunda propiedad evita la necesidad de buscar en todo el depósito para encontrar un nodo con la etiqueta de distancia mínima. Cuando implementamos el algoritmo Radix-Heap de esta manera, su tiempo de ejecución es $O(m+nlog(nC))$. Para un problema del camino más corto, el algoritmo consta de $\lceil 1 + log(nC) \rceil$ depósitos. Estos cubos se enumeran $0, 1, 2, \ldots, K = \lceil log(nC) \rceil$. Denotaremos el rango del cubo k por $rango(k)$ que es un intervalo cerrado de números enteros, posiblemente vacío y, como ya vimos, $contenido(k)$ denota los nodos del depósito k. Cada vez que el algoritmo modifica los rangos de los depósitos, redistribuye los nodos en estos.
Inicialmente, los depósitos tienen los siguientes rangos:

$$\begin{aligned} rango(0) &= [0] \\ rango(1) &= [1] \\ rango(2) &= [2, 3] \\ rango(3) &= [4, 7] \\ rango(4) &= [8, 15] \\ &\vdots \\ rango(k) &= [2^{k-1}, 2^k - 1] \end{aligned}$$

A medida que se ejecuta el algoritmo, estos rangos cambian, pero las anchuras de los cubos nunca aumentan más que las iniciales.
En conclusión, cada vez que el algoritmo encuentra que los nodos con la etiqueta de distancia mínima están en un depósito con una anchura mayor que 1, examina todos los nodos

de ese depósito para identificar un nodo con la etiqueta de distancia mínima. Por tanto, el algoritmo reorganiza el rango del depósito y cambia cada nodo al depósito de índice inferior. Como esta implementación contiene K depósitos, un nodo puede deplazarse como máximo K veces, con lo cual el número total de examinación de nodos es $O(nK)$.

Ejemplo 4.6. *Ilustramos ahora esta implementación para el problema del camino más corto. Consideramos el siguiente grafo, donde los números en los arcos representan la longitud entre los diferentes nodos:*



En este grafo, se tiene que : $C = 9$ *y* $K = \lceil log(900) \rceil = 10$ [2] *. Por tanto, en la etapa inicial del algoritmo tendremos:*

nodo i	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*
etiqueta d(i)	*0*	*6*	*7*	*7*	∞	∞	∞	∞	∞	∞

cubo k	*0*	*1*	*2*	*3*	*4*	*5*	*6*
rango(k)	*[0]*	*[1]*	*[2,3]*	*[4,7]*	*[8,15]*	*[16,31]*	*[32,63]*
contenido (k)	$\emptyset$	$\emptyset$	$\emptyset$	*{2,3,4}*	$\emptyset$	$\emptyset$	$\emptyset$

cubo k	*7*	*8*	*9*	*10*
rango (k)	*[64,127]*	*[128,255]*	*[256,511]*	*[512,1023]*
contenido (k)	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

[2] Utilizamos el logaritmo en base 2 (binario).

Las tablas anteriores representan las etiquetas de distancia determinadas por el algoritmo de Dijktra después de haber examinado el primer nodo, y muestra también el algoritmo Radix-Heap. Para elegir el nodo con la etiqueta de distancia más pequeña. examinamos los cubos $0, 1, 2, \ldots, K$ *para encontrar el primer cubo no vacío. El primer (y único) cubo no vacío que tenemos en la primera etapa es el cubo 3, que contiene a los nodos 2, 3 y 4. Como el rango de este cubo contiene más de un número entero, no es necesario que el primer nodo del cubo tenga la etiqueta de distancia mínima. En nuestro ejemplo, el rango del cubo 3 es* $[4, 7]$*, pero la etiqueta de distancia más pequeña en ese cubo es 6. Por tanto, redistribuimos el rango del cubo 3 a* $[6, 7]$ *entre los segmentos de indice inferior de la siguiente manera:*

$$rango(0) = [6]$$
$$rango(1) = [7]$$
$$rango(2) = \emptyset$$
$$rango(3) = \emptyset$$

El rango de los otros cubos no cambia. El rango del cubo 3 está vacío y tenemos que reasignar el contenido del cubo 3 a los cubos 0,1,2. Para eso, seleccionamos los nodos del cubo 3, escaneamos secuencialmente los cubos 2,1,0 e insertamos los nodos en el cubo adecuado. Los depósitos resultantes tienen el siguiente contenido:

$$contenido(0) = \{2\}$$
$$contenido(1) = \{3, 4\}$$
$$contenido(2) = \emptyset$$
$$contenido(3) = \emptyset$$

Esta redistribución vacía el cubo 3 y mueve el nodo con la etiqueta de distancia más pequeña al cubo 0.

Ahora, ya podemos analizar la complejidad del algoritmo. Supongamos que $j \in contenido(k)$ y que reasignamos el nodo j a un cubo de menor índice. Si $d(j) \notin rango(k)$, escaneamos los cubos de manera secuencial con índices más bajos de derecha a izquierda y agregamos el

nodo en el cubo correspondiente. En general, esta operación requiere $O(m + nK)$ tiempo, donde m denota el número de actualizaciones de distancia, y el término nK es consecuencia de que cada vez que movemos un nodo, lo movemos a un depósito con un índice más bajo: dado que tenemos $K+1$ cubos, un nodo podrá moverse, como máximo, K veces. Entonces, $O(nK)$ es un límite en el número total de movimientos de nodos. Consideramos ahora la operación de selección de nodos, que comienza escaneando los cubos de izquierda a derecha para identificar el primer cubo no vacío. Supongamos que este primer cubo no vacío es el cubo k. Esta operación necesita $O(K)$ tiempo por iteración y tiempo $O(nK)$ en total. Si $k = 0, 1$, entonces cualquier nodo en ese cubo tiene la etiqueta de distancia mínima. Por otra parte, si $k \geq 2$, redistribuimos el rango del depósito k en los depósitos $0, 1, \ldots, k-1$ y reinsertamos su contenido en esos cubos. Es decir, si el rango del depósito k es $[l, u]$ y la etiqueta de distancia más pequeña de un nodo en ese depósito es d_{min}, el rango útil del depósito será $[d_{min}, u]$. El algoritmo redistribuye el rango útil de la siguiente forma: asignamos el primer número entero al cubo 0, después el siguiente entero al cubo 1, los dos siguientes al cubo 2, y así sucesivamente. Como el cubo k tiene un ancho menos que $2k-1$ y los anchos de los primeros k cubos pueden ser $1, 1, 2, \ldots, 2k-2$ para un ancho potencial total de $2k-1$, podemos redistribuir el rango útil del cubo entre los cubos $0, 1, \ldots, k-1$ de la forma que explicamos. Esta redistribución de rangos vacía el depósito k y mueve los nodos con las etiquetas de distancia más pequeñas al depósito 0. La redistribución de rangos requiere $O(K)$ tiempo por iteración y $O(nK)$ tiempo en todas las iteraciones. En consecuencia, el tiempo que requiere el algoritmo es $O(m + nK)$. Como $K = \lceil log(nC) \rceil$, el algoritmo se ejecuta en $O(m + nlog(nC))$ tiempo.

Teorema 4.7. *La **implementación Radix-Heap** del algoritmo de Dijkstra resuelve el problema del camino más corto en $O(m + nlog(nC))$ tiempo.*

Este algoritmo requiere $1 + \lceil log(nC) \rceil$ depósitos. Como en el algoritmo de Dial, la propiedad 4.4 nos permite reducir el número de cubos a $1 + \lceil (logC) \rceil$. Esta implementación del algoritmo se ejecuta en tiempo $O(m + nlogC)$.

Capítulo 5

Algoritmos de corrección de etiquetas para el problema del camino más corto

En el capítulo anterior hemos visto diferentes algoritmos para resolver problemas del camino más corto cuando las longitudes de arco son no negativas. Pero este problema es más difícil de resolver cuando las redes tienen costes arbitrarios. La teoría de la complejidad computacional clasifica el problema del camino más corto para las redes con ciclos negativos como un problema $NP - completo$, por lo que resolverlo equivale a resolver muchos problemas en los campos de la combinatoria y la programación entera. En consecuencia, difícilmente podremos diseñar algoritmos de tiempo polinomial para la configuración de estos problemas, pero podemos describir algoritmos polinomiales que detecten un ciclo negativo cuando exista. Básicamente, todos los algoritmos del camino más corto se basan en las etiquetas de distancia. El algoritmo más básico que consideraremos en este capítulo, que es el *algoritmo genérico de corrección de etiquetas*, que reduce la etiqueta de distancia de un nodo en cada iteración al considerar solamente información local, es decir, la longitud del arco y las etiquetas de distancia actuales de sus nodos adyacentes. Bajo el supuesto de costes enteros, las etiquetas de distancia serán enteras y, por tanto, el algoritmo genérico siempre será finito. Pero también queremos trabajar con algoritmos que no solo sean finitos, sino que requieran una cantidad de cálculos que crezcan polinomialmente en el tamaño del problema. Comenzaremos este capítulo hablando de las condiciones de optimalidad, que nos permiten decidir cuándo un conjunto de etiquetas de distancia es óptimo. Estas condiciones nos permiten saber cuando una solución factible de nuestro problema es

óptima y, cuando una posible solución no satisface estas condiciones, nos sugieren cómo podríamos modificar la solución actual para que se acerce más a la óptima. Los algoritmos de corrección de etiquetas establecen $d(s) = 0$, donde s es el nodo origen, y mantienen una etiqueta de distancia $d(i)$, para cada nodo $i \in N$, que es una aproximación de la distancia del camino más corto de "prueba" del nodo s al nodo i y que, finalmente, es la etiqueta del camino más corto.

5.1. Condiciones de optimalidad

Si las etiquetas de distancia son distancias de los caminos más cortos, deben satisfacer la siguiente condición:

$$d(j) \leq d(i) + c_{ij} \qquad \forall (i,j) \in A \tag{5.1}$$

Esto quiere decir que la longitud del camino más corto al nodo j no es mayor que la longitud del camino más corto de s a i más la longitud del arco (i,j), para cada $(i,j) \in A$. Por el contrario, si $d(j) > d(i) + c_{ij}$ $\forall (i,j) \in A$, podemos mejorar la longitud del camino más corto al nodo j pasando por el nodo i, pero esto contradice la optimalidad de la etiqueta de distancia $d(j)$. Además, si cada $d(j)$ representa la longitud de algún camino dirigido de s a j y esta solución satisface (5.1), entonces debe ser óptima. Consideremos cualquier solución $d(j)$ que verifique (5.1). Sea $s = i_1 - i_2 - \cdots - i_k = j$, cualquier camino dirigido P de s a j, se tiene que:

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

La última igualdad se debe a que $d(i_1) = d(s) = 0$. Sumando estas desigualdades, obtenemos que:

$$d(j) = d(i_k) \leq c_{i_{k-1}i_k} + c_{i_{k-2}i_{k-1}} + c_{i_{k-3}i_{k-2}} + \cdots + c_{i_1 i_2} = \sum_{(i,j)\in P} c_{ij}$$

Entonces $d(j)$ es un límite inferior en la longitud de cualquier camino dirigido desde s hasta j. Como $d(j)$ es la longitud de algún camino dirigido desde s a j, también es un límite superior en la longitud del camino más corto. En conclusión, $d(j)$ es la longitud del camino más corto y establecemos el siguiente resultado:

Teorema 5.1 (Condiciones de optimalidad del camino más corto)**.** *Para cada nodo $j \in N$, sea $d(j)$ la longitud de algún camino dirigido desde el nodo origen al nodo j. Entonces, las etiquetas $d(j)$ representan distancias de caminos más cortos si y solo si satisfacen las siguientes condiciones de optimización de camino más corto:*

$$d(j) \leq d(i) + c_{ij} \qquad \forall (i,j) \in A \tag{5.2}$$

Definimos ahora la longitud de arco reducida c_{ij}^d de un arco (i,j) con respecto a las etiquetas de distancia $d(\cdot)$ como $c_{ij}^d = c_{ij} + d(i) - d(j)$. Introducimos a continuación unas propiedades que serán útiles más tarde.

Proposición 5.2.

1. *Para cualquier ciclo dirigido W, se tiene que: $\sum_{(i,j)\in W} c_{ij}^d = \sum_{(i,j)\in W} c_{ij}$*
2. *Para cualquier camino dirigido P desde el nodo k al nodo l, se verifica:*

$$\sum_{(i,j)\in P} c_{ij}^d = \sum_{(i,j)\in P} c_{ij} + d(k) - d(l)$$

3. *Si $d(\cdot)$ representa las distancias de caminos más cortos, entonces $c_{ij}^d \geq 0$ para cada arco $(i,j) \in A$.*

La tercera propiedad se obtiene directamente del teorema 5.1. Nótese que si la red tiene un ciclo negativo, ningún conjunto de etiquetas satisface 5.2. Supongamos que G tiene un ciclo dirigido W. Entonces, por la propiedad 5.2, $\sum_{(i,j)\in W} c_{ij}^d \neq 0$ y, por tanto, W no puede ser un ciclo negativo.

5.2. Detección de ciclos negativos

Hasta ahora, hemos supuesto que no hay costes negativos y hemos descrito algoritmos que resuelven este tipo de problemas del camino más corto. En esta sección vamos a permitir costes negativos y veremos qué modificaciones son necesarias para que los algoritmos detecten la presencia de ciclos negativos. Como ya hemos dicho, si el grafo contiene un ciclo negativo, ningún conjunto de etiquetas de distancia cumple las condiciones de optimalidad y el algoritmo de corrección de etiquetas reducirá las etiquetas de distancia indefinidamente y no terminará.. Sin embargo, acabaríamos los cálculos si encontramos que la etiqueta de

distancia de algún nodo es menor que $-nC$, ya que esta cantidad es un límite inferior en las etiquetas de distancia. Describimos a continuación el procedimiento de un algoritmo que detecta ciclos negativos:

1. Designa el nodo origen como marcado y los demás como no marcados.
2. Examina cada nodo no marcado k, lo designa como marcado, traza los índices predecesores comenzando en el nodo k, y marca los nodos encontrados hasta llegar al primer nodo marcado i.
3. Si $k = i$, el grafo predecesor contiene un ciclo negativo.

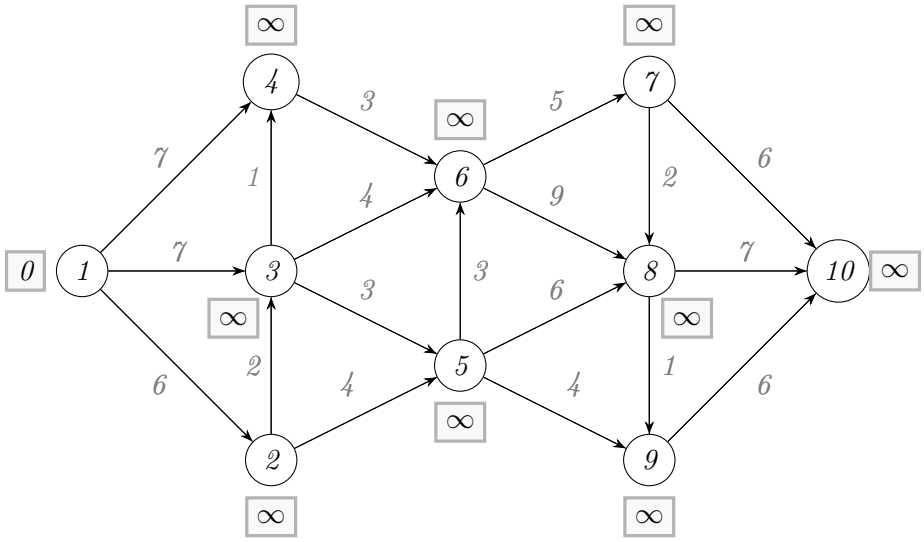
Este algoritmo necesita $O(n)$ tiempo para verificar la presencia de un ciclo negativo en el grafo predecesor. En consecuencia, si aplicamos este algoritmo después de cada actualización de distancia, los cálculos que realiza no incrementarán la complejidad computacional de un algoritmo de corrección de etiquetas. En general, en el momento en que el algoritmo vuelve a etiquetar el nodo j, $d(j) = d(i) + c_{ij}$ para algún nodo i que es el predecesor de j. Nos referimos al arco (i, j) como arco predecesor. Posteriormente, $d(i)$ podría disminuir, y las etiquetas cumplirán la condición $d(j) \leq d(i) + c_{ij}$ siempre que $pred(j) = i$. Supongamos que P es un camino de arcos predecesores desde el nodo 1 al nodo j. Las desigualdades $d(k) \leq d(l) + c_{kl}$ para todos los arcos (k, l) en este camino implican que $d(j)$ es, al menos la longitud de este camino. En consecuencia, ningún nodo j con $d(j) \leq -nC$ está conectado al nodo 1 en un camino que consta solo de arcos predecesores. Concluimos que rastrear los arcos predecesores desde el nodo j debe llevarnos a un ciclo negativo.

5.3. Algoritmo genérico de corrección de etiquetas

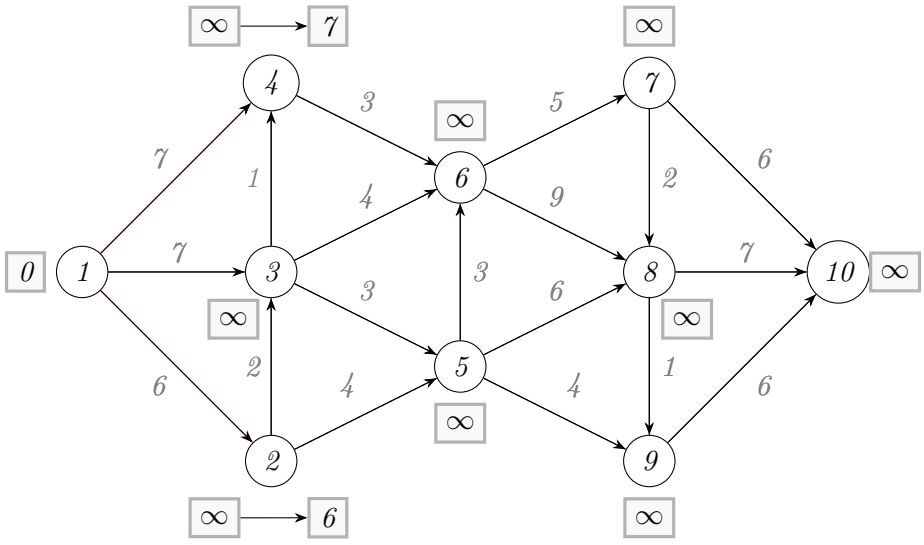
El algoritmo genérico de corrección de etiquetas es un procedimiento para actualizar de manera sucesiva las etiquetas de distancia hasta que verifiquen las condiciones de optimalidad, explicadas en la sección anterior. Este algoritmo mantiene un conjunto de etiquetas de distancia $d(\cdot)$ en cada etapa. Si $d(i) = \infty$, todavía tenemos que encontrar un camino dirigido desde el nodo origen hasta el nodo i. En otro caso, $d(i)$ representa la longitud de algún camino dirigido del nodo s al nodo i. Además, para cada nodo $i \in N$ mantenemos un índice predecesor, $pred(i)$, que almacena el nodo anterior al nodo i en el camino dirigido actual de longitud $d(i)$. Al terminar el algoritmo, estos índices predecesores nos permiten conocer el camino más corto desde el nodo origen hasta el nodo i. Por definición de costes reducidos, las etiquetas de distancia satisfacen las condiciones de optimalidad si $c_{ij}^d \geq 0$

para todo $(i, j) \in A$. El algoritmo genérico de corrección de etiquetas selecciona un arco (i, j) que no cumple las condiciones de optimalidad, es decir $c_{ij}^d < 0$, y lo utiliza para actualizar la etiqueta de distancia del nodo j. Esto hace que disminuya esta etiqueta de distancia y que la longitud de arco reducida de (i, j) sea cero.

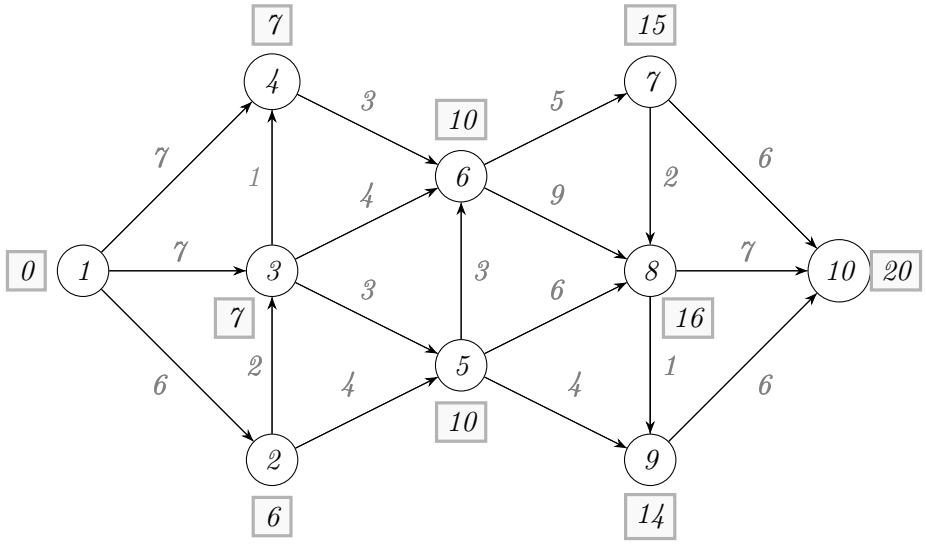
Ejemplo 5.3. *Usaremos ahora el ejemplo que utilizamos en el capítulo 4, donde el grafo asociado es el siguiente:*



Las etiquetas rectangulares que aparecen cerca de cada nodo son las $d(i)$. Si el algoritmo selecciona los arcos $(1, 2)$ y $(1, 4)$, por ejemplo, actualizaríamos las etiquetas de distancia de la siguiente manera:



Repitiendo el proceso con todos los nodos obtendríamos las siguientes etiquetas de distancia, que serían óptimas:



El algoritmo mantiene un índice predecesor para cada nodo etiquetado de manera finita. Nos referiremos al conjunto de arcos $(pred(i), i)$ *para cada nodo* $i \in N$ *etiquetado de manera finita como el grafo predecesor, que es un árbol externo dirigido* T *con raíz en el nodo origen que abarca todos los nodos con etiquetas de distancia finita.*

Proposición 5.4. *El algoritmo de corrección de etiquetas satisface la* ***propiedad invariante****, es decir, para cada arco* (i, j) *en el gráfico predecesor, se tiene que* $c_{ij}^d \geq 0$.

Demostración. Realizaremos la demostración mediante un proceso de inducción sobre el número de iteraciones. Nótese que el algoritmo agrega un arco (i, j) al gráfico predecesor en una actualización de distancia, por tanto, después de la actualización $d(j) = d(i) + c_{ij}$ ó $c_{ij}^d + d(i) - d(j) = c_{ij}^d = 0$. En las siguientes iteraciones, puede disminuir $d(i)$, con lo cual c_{ij}^d podría ser negativo. Además, $d(j)$ disminuye a lo largo del algoritmo y, para algún arco (i, j) en el grafo predecesor, c_{ij}^d puede convertire en positivo, contradiciendo la propiedad invariante. Pero en este caso, borramos el arco (i, j) del gráfico y mantenemos así dicha propiedad. [2] □

Proposición 5.5. *El grafo predecesor contiene un único camino dirigido desde el nodo origen a cada nodo* k *y la longitud de ese camino es, como mucho,* $d(k)$.

Demostración. Sea P el camino dirigido desde el nodo origen al nodo k, como cada arco en el grafo predecesor tiene una longitud de arco reducida no positiva, se tiene: $\sum_{(i,j)\in P} c_{ij}^d \geq 0$. Debido a la propiedad 2 de la proposición 5.2, $\sum_{(i,j)\in P} c_{ij}^d \geq 0 =$
$= \sum_{(i,j)\in P} c_{ij} \geq 0$ .Alternativamente, $\sum_{(i,j)\in P} c_{ij}^d \geq d(k)$. Cuando el algoritmo termina, cada arco en el grafo predecesor tiene una longitud de arco reducida igual a 0, lo que implica que la longitud del camino más corto desde el nodo origen al nodo k es $d(k)$. En consecuencia, el grafo predecesor es un árbol de camino más corto cuando el algoritmo termina. [2]

□

En ausencia de ciclo negativos, el grafo predecesor es, como dijimos, un árbol. Sin embargo, cuando existe algún ciclo negativo, el algoritmo lo detecta y el grafo predecesor no es un árbol.

5.4. Algoritmo de corrección de etiquetas modificado

El algoritmo de corrección de etiquetas que hemos descrito no explica ningún método para seleccionar un arco que no cumpla las condiciones de optimalidad. Un enfoque sencillo es examinar la lista de arcos secuencialmente e identificar los arcos que no cumplan dichas condiciones. Pero esto necesita un tiempo $O(m)$ por iteración, por lo que describiremos un enfoque mejorado que reduce este tiempo a $O(\frac{m}{n})$ por iteración. Explicamos la idea principal de este algoritmo: Suponemos que tenemos una lista de todos los arcos que no cumplen las condiciones de optimalidad. Si la lista está vacía, estamos ante una solución óptima. En caso contrario, seleccionamos un arco (i, j) de la lista. Eliminamos este arco de la lista y, si no cumple las condiciones de optimalidad, lo utilizamos para actualizar la etiqueta de distancia del nodo j. Nótese que una disminución en esta etiqueta de distancia disminuye las longitudes reducidas de todos los arcos que salen del nodo j. Por tanto, en este caso, deberíamos introducir $A(j)$ en la lista. Obsérvese que cuando introducimos arcos en la lista, agregamos todos aquellos que salen de un solo nodo, cuya etiqueta de distancia disminuye. Esto quiere decir que podemos mantener una lista de nodos con la propiedad de que si un arco (i, j) incumple las condiciones de optimalidad, entonces la lista debe contener al nodo i. Mantener una lista de nodos en lugar de una lista de arcos requiere menos trabajo y da lugar a algoritmos más rápidos en la práctica. La exactitud del algoritmo de corrección de etiquetas modificado se debe a que la lista contiene todos los nodos i que entran en un arco (i, j) que incumple las condiciones de optimalidad.

Si hacemos inducción en el número de iteraciones, vemos que esta propiedad es válida en todo el algoritmo. Nótese que siempre que el algoritmo actualice $d(i)$, agrega el nodo i a la lista. El algoritmo selecciona este nodo en una iteración posterior y escanea $A(i)$. Como el algoritmo puede actualizar la etiqueta de distancia $d(i)$, como máximo, $2nC$ veces, obtenemos un límite de $\sum_{i \in N} 2nC|A(i)| = O(nmC)$ en el número total de examinaciones de arco. Con lo cual, esta versión del algoritmo genérico de corrección de etiquetas se ejecuta en tiempo $O(nmC)$.

5.4.1. Implementación de eliminación de cola

El algoritmo de corrección de etiquetas modificado que presentamos a continuación, denominado **Implementación de eliminación de cola**, es un algoritmo pseudopolinomial en el peor caso, pero ha demostrado ser uno de los algoritmos más rápidos en la práctica para resolver problemas del camino más corto en redes poco densas. [1] Una eliminación de cola es una estructura de datos que permite almacenar una lista para que podamos añadir o eliminar elementos de la lista tanto al principio como al final de ella. Una eliminación de cola se puede implementar fácilmente mediante una matriz o una lista enlazada (ver la definición 1.15). Esta implementación selecciona los nodos del principio de la eliminación de cola, pero agrega nodos en la parte final. Si el nodo ha estado en la lista anteriormente, el algoritmo lo agrega al principio; de lo contrario, lo añade al final. Esta regla tiene la siguiente explicación: si un nodo i ha aparecido antes en la lista, algunos nodos, por ejemplo $i_1, i_2, \ldots, i_k$, podrían tener al nodo i como su predecesor. Supongamos además que la lista contiene los nodos $i_1, i_2, \ldots, i_k$ cuando el algoritmo actualiza $d(i)$ de nuevo. Entonces es preferible actualizar las etiquetas de distancia de los nodos $i_1, i_2, \ldots, i_k$ desde el nodo i lo antes posible en lugar de examinar primero los nodos $i_1, i_2, \ldots, i_k$ y luego reexaminarlos, ya que sus etiquetas de distancia disminuirán debido a la disminución en $d(i)$. Los estudios empíricos reflejan que esta implementación examina menos nodos que la mayoría de algoritmos de corrección de etiquetas.

[1] Una red se dice que es densa si $m \approx n^2$. Por el contrario, en una red poco densa se tiene que $m << n^2$.

Capítulo 6

Algunas aplicaciones del problema del camino más corto

El problema del camino más corto surge de muchos problemas prácticos, tanto modelos independientes como subproblemas de otros problemas más complejos. Pueden surgir en las industrias de las telecomunicaciones, por ejemplo, cuando queremos enviar un mensaje de un lugar a otro de la forma más rápida y económica posible. La planificación del tráfico urbano es otro ejemplo, ya que los modelos que se utilizan para calcular los patrones de flujo de tráfico son problemas de optimización no lineales complejos; sin embargo, se basan en que los usuarios se transportan, con respecto a la congestión del tráfico, por los caminos más cortos desde sus orígenes hasta sus destinos. Por tanto, la mayor parte de los algoritmos que se utilizan para encontrar patrones de tráfico resuelven muchos problemas de camino más corto como subproblemas (uno para cada par origen-destino). En este trabajo consideraremos algunas aplicaciones del problema del camino más corto a la vida real.

6.1. El problema de la mochila

La primera aplicación que presentaremos en este capítulo será **El problema de la mochila** [1] , descrito a continuación. Un excursionista tiene que decidir entre los n objetos que tiene cuales debe incluir en su mochila para realizar su próximo viaje. Cada objeto tendrá un peso p_k y una utilidad u_k, y el problema consiste en maximinar la utilidad del viaje sujeto a la limitación de peso, que será P. Podemos expresar este problema como un problema de programación entera de la siguiente forma:

$$\begin{aligned} \text{máx} \quad & \sum_{k=1}^{n} u_k x_k \\ \text{sujeto a} \quad & \sum_{k=1}^{n} p_k x_k \leq P \\ & x_k \in \{0,1\} \qquad k \in \{1,\ldots,n\} \end{aligned}$$

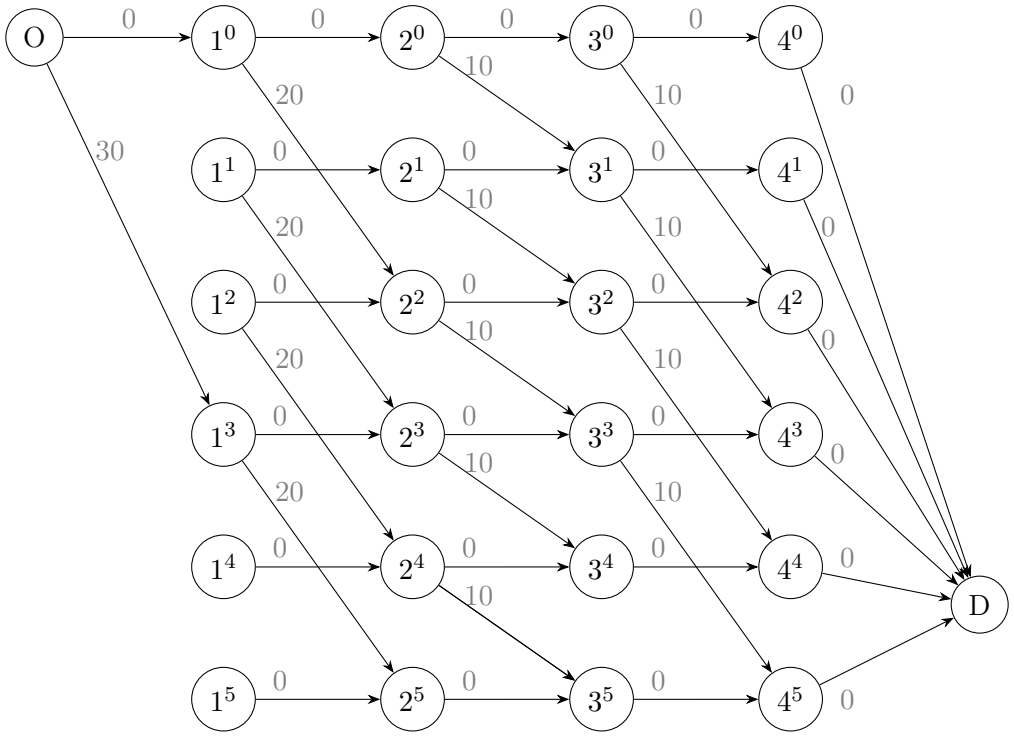
Donde x_k es 0 si no metemos el objeto en la mochila y 1 en otro caso. El problema de la mochila es un problema NP-completo, pero existen algoritmos aproximados completamente polinomiales y algoritmos "pseudo-polinomiales" para resolverlo. Como vamos a ver a continuación, prácticamente cualquier problema de programación dinámica se puede reformular como un problema de camino más corto.

Lo que haremos será plantear el problema como un problema del camino más largo (para pasarlo a un problema del camino más corto, cambiamos el signo a la longitud de todos los arcos) [1]. Formulamos el problema de la siguiente forma: tenemos un nodo origen y un nodo destino y, para cada objeto i, tenemos una columna con $P+1$ nodos $i^0, i^1, \ldots, i^P$. El nodo i^j representa que los objetos $1,\ldots,i$ consumen j unidades de la capacidad de la mochila. De cada nodo salen dos arcos hacia la siguiente columna que representa la entrada o no entrada del objeto de esa columna en la mochila, y solamente saldrá uno si ese objeto no cabe ya en la mochila. El arco de no entrada tendrá utilidad 0 y el de entrada tiene la utilidad asociada a dicho objeto. Pongamos, por ejemplo, un caso en el que el peso máximo es $P = 5$ y la siguiente tabla representa las utilidades y los pesos de cada uno de los 4 objetos que podemos introducir en la mochila:

k	1	2	3	4
u_k	30	20	10	15
p_k	3	2	1	2

Entonces, por lo que dijimos en el anterior párrafo, el grafo tendría 4 "columnas" y 6 "filas":

[1] Tenemos que tener cuidado de no obtener un ciclo de longitud total negativa.



Por ejemplo, el camino $O - 1^0 - 2^2 - 3^3 - 4^5 - D$ indica que los elementos 2, 3 y 4 entrarían en la mochila.

Además del problema de la mochila, existen muchas otras aplicaciones del problema del camino más corto a la vida real.

Una aplicación curiosa del camino más corto es, por ejemplo, la teoría de los **Seis grados de separación**, que es la idea de intentar probar que cualquier persona está conectada con cualquier otra mediante una cadena de, como mucho, seis personas.

Otra aplicación interesante es la de los dispositivos gps, los cuales nos indican el camino más corto desde el punto en el que nos encontramos y cualquier destino al que queramos llegar. En la realidad, varios departamentos económicos necesitan realizar consultas de redes de transporte a gran escala, los departamentos de logística deben cruzar ciudades y provincias para el transporte, los departamentos de turismo deben ir a atracciones turísticas lejos de las áreas urbanas, etc.

También cabe destacar que, otro ejemplo, son las empresas (tiendas, grandes almacenes, mensajeros, etc.) que necesitan repartir sus pedidos a diferentes lugares del mundo. Lo que hacen es diseñar el camino más corto para poder entregar todos los pedidos lo más rápido y económico posible. Para eso, tendrán en cuenta los lugares que deben visitar, la disponibilidad de los clientes para entregar los pedidos, el tiempo que tardará en llegar a cada sitio, etc.

Bibliografía

[1] González-Díaz, J. (2021): Apuntes de Programación Lineal y Entera, Universidad de Santiago de Compostela, Grado en Matemáticas.

[2] K. Ahuja R. , L. Magnanti T. , B. Orlin J. (1993), *Network Flows. Theory, Algorithms, and Applications.*