FACULTAD DE MATEMÁTICAS

Trabajo fin de Grado

Grado en Matemáticas

Librería sobre grafos en Haskell: el problema del camino más corto

Realizado por
Pablo Manrique Merchán

Dirigido por
Francisco Jesús Martín Mateos

Departamento
Ciencias de la Computación
e Inteligencia Artificial

Sevilla, Julio de 2023

Abstract

The shortest path problem is an important problem in graph theory, consisting in finding paths between two nodes in weighted graphs such that the sum of the weights of their edges is minimized. In addition to its theoretical interest, this problem has applications in various fields, such as social networks, data analysis and route optimization.

In this project, we firstly make a theoretical study of the shortest path problem, presenting several algorithms. Then, we discuss the creation of a Haskell library for graphs, focusing on solving the shortest path problem and implementing the previously studied algorithms. Finally, we use the library to find the shortest paths in specific examples.

Resumen

El problema del camino más corto es un problema de suma importancia en teoría de grafos, que consiste en encontrar caminos entre dos nodos en grafos ponderados tales que la suma de los pesos de sus aristas sea mínima. Además de su interés teórico, este problema tiene aplicaciones en distintos campos, como redes sociales, análisis de datos y optimización de rutas.

En este trabajo, en primer lugar haremos un estudio teórico del problema del camino más corto, presentando algunos algoritmos. Después, comentaremos la creación de una librería Haskell sobre grafos, centrándonos en la resolución del problema del camino más corto e implementando los algoritmos previamente estudiados. Finalmente, usaremos la librería para encontrar caminos más cortos en ejemplos concretos.

Agradecimientos

A mi familia, por todo su apoyo. A mis amigos, por haberme acompañado todos estos años. A mis profesores, y en especial a mi tutor Francisco Jesús, por toda su ayuda.

Índice general

CAPÍTULO 1

Introducción

El trabajo de uno de los mayores matemáticos de la historia, Leonard Euler, sobre el problema de los puentes de Königsberg en 1736, es considerado como el primer resultado de la teoría de grafos. Desde entonces, esta rama de las matemáticas ha ido desarrollándose dando resultados de gran relevancia tanto desde un punto de vista puramente teórico como desde el punto de vista aplicado, siendo clave incluso en campos aparentemente tan alejados de las matemáticas como la sociometría, la antropología o la psicología social. En las últimas décadas, ha tenido mayor influencia en el campo de la informática, las telecomunicaciones y las ciencias de la computación. En programación, los grafos juegan un papel fundamental en la representación de relaciones y conexiones entre objetos, siendo estructuras de datos muy versátiles utilizadas en diversas aplicaciones, como redes sociales, sistemas de enrutamiento, análisis de datos y optimización de rutas.

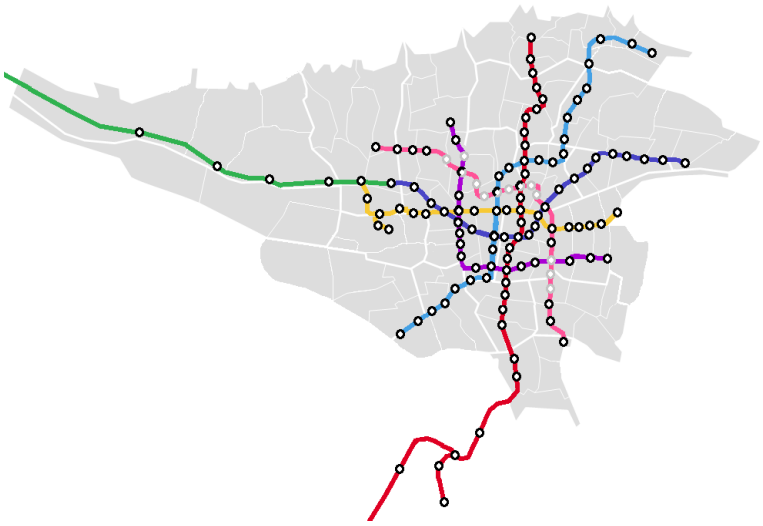


Figura 1.1: Plano de metro representado por un grafo

El lenguaje de programación Haskell es un lenguaje de programación funcional en el que desarrollaremos en este trabajo una librería que nos permita representar y trabajar

con grafos, centrándonos especialmente en un problema en particular de teoría de grafos: el problema del camino más corto. Este problema es de vital importancia en muchos escenarios, desde la planificación de rutas óptimas en aplicaciones de navegación hasta la optimización de redes de transporte.

Comenzaremos dedicando un capítulo a hacer un desarrollo teórico del problema del camino más corto, en el que definiremos el problema en términos de teoría de grafos, además de realizar un análisis de diversos algoritmos que lo resuelven, describiéndolos, demostrando su corrección y estudiando su complejidad, así como comparándolos entre sí.

Continuaremos describiendo en otro capítulo la implementación en Haskell de la librería, mostrando la construcción que hemos realizado de los grafos y las funciones que nos permiten trabajar con ellos. Además, en este capítulo implementaremos los algoritmos estudiados con anterioridad, con el objetivo de poder usarlos para resolver el problema del camino más corto.

Finalmente, concluiremos el trabajo con un último capítulo en el cual usaremos la librería creada para resolver problemas de camino más corto en grafos concretos, destacando la aplicación del algoritmo de búsqueda A* para encontrar rutas más cortas en una versión simplificada de la red de carreteras de Andalucía y, por último, la puesta en práctica de varios de los algoritmos de camino más corto para conseguir encontrar la salida de un laberinto.

CAPÍTULO 2

El problema del camino más corto

En teoría de grafos, el *problema del camino más corto* (en inglés *shortest path problem*) o *PCC*, consiste en encontrar un camino entre dos vértices (o nodos) de un grafo tal que la suma de los pesos de sus aristas constituyentes sea mínima.

Resulta difícil determinar cuándo fue planteado por primera vez el problema del camino más corto y trazar su historia. Encontrar la ruta más corta entre dos puntos, por ejemplo para buscar comida o realizar migraciones, con seguridad era un problema de suma importancia ya para sociedades muy primitivas, e incluso para animales. Sin embargo, en comparación con otros problemas de optimización combinatoria, el estudio matemático de este problema comenzó relativamente tarde, en torno a las décadas de 1940 y 1950. Esto puede deberse a que el problema puede resultar relativamente sencillo o elemental en ciertos casos, como por ejemplo encontrar el camino más corto entre dos puntos en un mapa con pocos obstáculos; sin embargo, la aparición de nuevos avances tecnológicos en el último siglo requirió de la resolución del problema en contextos más sofisticados, como en redes de carreteras y ferroviarias o en el enrutamiento de llamadas telefónicas. Además, con la aparición de los ordenadores, surgió la posibilidad de utilizar algoritmos que encuentren una solución óptima para redes de un gran tamaño. Alexander Schrijver hace en *On the history of the shortest path problem* [1] un análisis de la historia del PCC.

A finales de los años cuarenta y principios de los cincuenta se desarrollaron métodos matriciales para resolver el PCC en grafos con pesos unitarios. Estos métodos consisten en representar un grafo dirigido por una matriz, y realizar productos matriciales iterativamente para calcular la clausura transitiva del grafo. Algunos autores que desarrollaron estos métodos fueron Landahl y Runge [2] o Luce y Perry [3].

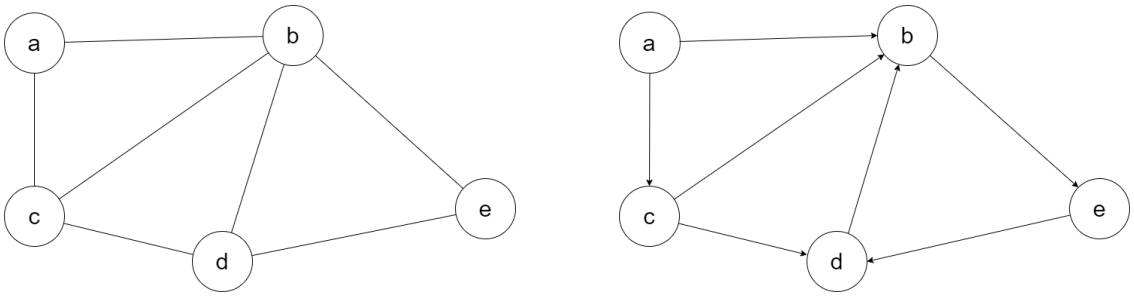
Más adelante, a finales de la década de 1950, surgieron nuevos algoritmos propuestos por autores como Bellman [4], Ford [5] o Dijkstra [6], que estudiaremos en este trabajo, así como los aportes posteriores de Warshall [7], Floyd [8] y Johnson [9]. También cabe destacar los algoritmos de tipo heurístico, como el algoritmo de búsqueda A*, desarrollado por Hart, Nilsson y Raphael [10] en 1968, que también estudiaremos en este trabajo. Este tipo de algoritmos usa información del contexto de aplicación del problema para guiar su búsqueda de la solución, pudiendo así mejorar su eficiencia.

2.1– Preliminares y definición del problema

En esta sección presentaremos los conceptos fundamentales de teoría de grafos que son necesarios para plantear y entender el problema. Comenzaremos, naturalmente, con la definición de grafo:

Definición 2.1. Un *grafo* G es una dupla (V, A), donde V es un conjunto finito cuyos elementos son llamados *vértices* o *nodos* y $A \subseteq V \times V$ es el conjunto de las *aristas* de G. Si en G no distinguimos la arista (u, v) de la arista (v, u), decimos que G es un grafo *no dirigido*. Si, por el contrario, en G distinguimos (u, v) de (v, u), entonces decimos que G es un grafo *dirigido* o un *dígrafo*.

Ejemplo 2.2. Definimos los grafos $G_1 = (V_1, A_1)$ (no dirigido) y $G_2 = (V_2, A_2)$ (dirigido), con $V_1 = V_2 := \{a, b, c, d, e\}$. Las aristas también coinciden: $A_1 = A_2 := \{(a, b), (a, c), (c, b), (c, d), (d, b), (b, e), (e, d)\}$, pero como G_2 es dirigido y G_1 no, representamos las aristas de G_1 con líneas y las de G_2 con flechas.



(a) G_1 (b) G_2

Figura 2.1: Ejemplos de grafos dirigidos y no dirigidos

Para definir el PCC, es necesario definir un tipo de grafo llamado *grafo ponderado*:

Definición 2.3. Un *grafo ponderado* G es una terna (V, A, p), donde V y A son conjuntos de nodos y aristas respectivamente, y $p : A \longrightarrow \mathbb{R}$ es una aplicación llamada *peso* tal que a cada arista del grafo asocia un valor real.

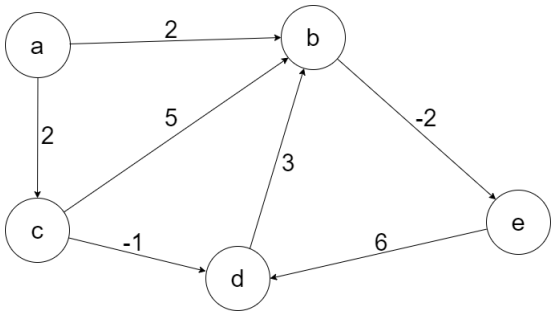


Figura 2.2: Ejemplo de grafo ponderado

Definición 2.4. Dados un grafo G (dirigido o no, ponderado o no) y dos nodos $u, v \in V$, decimos que v es *adyacente* a u si $(u, v) \in A$.

Volviendo al ejemplo 2.2, en G_2 el nodo b solo tendría a e como nodo adyacente, sin embargo en el grafo G_1 tendría como adyacentes a a, c, d y e, ya que al tratarse de un grafo no dirigido, no distinguimos la orientación de las aristas.

Definición 2.5. Un *camino* en un grafo G es una secuencia de nodos $C = (v_1, \ldots, v_n) \in V \times \ldots \times V = V^n$ tal que v_{i+1} es adyacente a v_i, $\forall i \in \{1, \ldots, n-1\}$. En este caso, decimos que C es un camino desde el nodo v_1 hasta el nodo v_n. Un *ciclo* es un camino $C = (v_1, \ldots, v_n)$ en el que $v_1 = v_n$. Un *subcamino* $C' = (v_1, \ldots, v_n)$ de un camino C en G es un camino en G tal que $C = (u_1, \ldots, u_i, v_1, \ldots, v_n, u_{i+1}, \ldots, u_N)$ con $u_j \in V$, $\forall j = 1, \ldots, N$.

En el grafo G_2, dos caminos entre a y e podrían ser por ejemplo $C_1 := (a, b, e)$ y $C_2 := (a, c, d, b, e)$. El camino $C_3 := (c, d, b)$ entre c y b es un subcamino de C_2, y $C_4 := (b, e, d, b)$ es un ciclo.

Definición 2.6. Dado un grafo ponderado G con función de pesos $p : A \longrightarrow \mathbb{R}$ y un camino $C = (v_1, \ldots, v_n)$ en G, el *peso de* C, denotado $p(C)$, se define como la suma de los pesos de las aristas que constituyen C:

$$p(C) := \sum_{i=1}^{n-1} p(v_i, v_{i+1}).$$

Definición 2.7. Suponiendo que existe algún camino entre dos nodos u y v en un grafo G, definimos el *peso* o *distancia del camino más corto* $\delta(u, v)$ entre los nodos u y v como

$$\delta(u, v) := \text{mín}\{p(C) : C = (u, \ldots, v)\}.$$

Si no existe ningún camino entre u y v, definimos $\delta(u, v) := \infty$.

Definición 2.8 (Problema del camino más corto)**.** Dado un grafo $G = (V, A, p)$ y un par de nodos $u, v \in V$, el *problema del camino más corto* consiste en encontrar un camino C en G desde u hasta v tal que $p(C) = \delta(u, v)$.

Considerando en G_2 los pesos que aparecen en la figura 2.2, obtenemos que los pesos de los caminos definidos anteriormente son: $p(C_1) = 0$, $p(C_2) = 2$, $p(C_3) = 2$ y $p(C_4) = 7$. De entre ellos, es fácil ver que los caminos C_1 y C_3 son caminos más cortos.

Notemos que aunque hemos definido el PCC sólo para grafos ponderados, también podemos hacerlo para un grafo no ponderado $G = (V, A)$ definiendo la función peso $p : A \longrightarrow \mathbb{R}$ como $p(a) := 1$ para toda arista $a \in A$. De esta forma, un camino más corto entre dos nodos dados de un grafo no ponderado es un camino entre dichos nodos con el menor número posible de aristas.

Observemos también que podemos considerar un grafo no dirigido como un caso particular de dígrafo en el que se tiene lo siguiente:

$$(u, v) \in A \iff (v, u) \in A \text{ y } p(u, v) = p(v, u). \tag{2.1}$$

Por ello, siempre que tengamos un algoritmo que resuelva el PCC para grafos dirigidos, también lo resolverá para grafos no dirigidos. En el ejemplo 2.2, las aristas de G_1 serían entonces $A_1 = A_2 \cup \{(v, u) : (u, v) \in A_2\}$, y las líneas que las representan serían en realidad flechas dobles.

Variantes

Dado un grafo $G = (V, A, p)$, existen distintas variantes del PCC, como las detalladas a continuación:

- **Single-source shortest path problem (SSSP):** consiste en encontrar un camino más corto desde un nodo *fuente* $s \in V$ hasta todos los demás nodos $v \in V$.
- **Single-destination shortest path problem (SDSP):** consiste en encontrar un camino más corto desde cualquier nodo $v \in V$ hasta un nodo *destino* $d \in V$. Podemos reducirlo a un problema SSSP invirtiendo la dirección de las aristas del grafo.
- **Single-pair shortest path problem (SPSP):** consiste en encontrar un camino más corto entre dos nodos $u, v \in V$ dados. Si resolvemos el problema SSSP con nodo fuente u, entonces también resolvemos este problema.
- **All-pairs shortest path problem (APSP):** consiste en encontrar un camino más corto entre u y v para cualquier par de nodos u y v de G. Puede resolverse ejecutando un algoritmo SSSP tomando como nodo fuente cada nodo v del grafo, aunque normalmente puede resolverse más rápido.

2.2– Introducción a los algoritmos que resuelven el PCC

Existen distintos algoritmos que resuelven el problema del camino más corto. En este trabajo nos centraremos en estudiar varios de los más importantes:

- El **algoritmo de Dijkstra** resuelve el problema SSSP para grafos con pesos no negativos.
- El **algoritmo de Bellman-Ford** resuelve el problema SSSP para grafos con pesos cualesquiera.
- El **algoritmo de búsqueda A*** es un algoritmo de tipo heurístico que resuelve el problema SPSP.
- El **algoritmo de Floyd-Warshall** resuelve el problema APSP.
- El **algoritmo de Johnson** resuelve el problema APSP, y puede ser más rápido que el de Floyd-Warshall en cierto tipo de grafos llamados *sparse* o *dispersos*.

Para el análisis de estos algoritmos, nos hemos basado principalmente en los artículos originales, citados en la introducción de este capítulo, y en la parte VI del libro *Introduction to Algorithms* de Cormen, Leiserson, Rivest y Stein [11].

2.2.1. Subestructura óptima de los caminos más cortos

Muchos algoritmos que resuelven el PCC, como por ejemplo el de Dijkstra o el de Bellman-Ford, se basan en la propiedad de que los subcaminos de caminos más cortos también son caminos más cortos. La siguiente proposición, correspondiente al lema 22.1 de [11], detalla la subestructura óptima de los caminos más cortos:

Proposición 2.1. *Dado un grafo $G = (V, A, p)$ y un camino más corto $C = (v_0, v_1, \ldots, v_n)$ entre los nodos v_0 y v_n en G, denotamos por $C_{ij} = (v_i, \ldots, v_j)$ al subcamino de C entre los nodos v_i y v_j, para cualesquiera i y j tales que $0 \leq i < j \leq n$. En estas condiciones, C_{ij} es un camino más corto entre v_i y v_j.*

Demostración. Descomponiendo el camino C en los subcaminos C_{0i}, C_{ij} y C_{jn}, tenemos que $p(C) = p(C_{0i}) + p(C_{ij}) + p(C_{jn})$. Supongamos ahora que existe otro camino C'_{ij} entre los nodos v_i y v_j tal que $p(C'_{ij}) < p(C_{ij})$. Entonces, concatenando los caminos C_{0i}, C'_{ij} y C_{jn}, obtenemos un nuevo camino C' entre v_0 y v_n con peso $p(C') = p(C_{0i}) + p(C'_{ij}) + p(C_{jn}) < p(C_{0i}) + p(C_{ij}) + p(C_{jn}) = p(C)$, pero esto contradice la hipótesis de que C es un camino más corto. □

2.2.2. Ciclos negativos

Como veremos más adelante, los ciclos negativos, o sea, los ciclos que tienen peso negativo, pueden ocasionar problemas a la hora de encontrar caminos más cortos en un grafo. Es por ello que debemos tenerlos en cuenta:

Definición 2.9. Dado un grafo ponderado $G = (V, A, p)$, un *ciclo negativo* en G es un ciclo $C = (v_1, \ldots, v_n = v_1)$ tal que $p(C) < 0$.

Ejemplo 2.10. Consideremos el grafo $G = (V, A, p)$, donde $V := \{a, b, c, d, e, f\}$, $A := \{(a, b), (b, c), (b, e), (b, d), (c, e), (d, e), (e, b), (e, f)\}$ y $p : A \longrightarrow \mathbb{R}$ es tal que $p(a, b) = 2$, $p(b, c) = 2$, $p(b, d) = 1$, $p(b, e) = 1$, $p(c, e) = 4$, $p(d, e) = -3$, $p(e, b) = 1$, $p(e, f) = 2$.

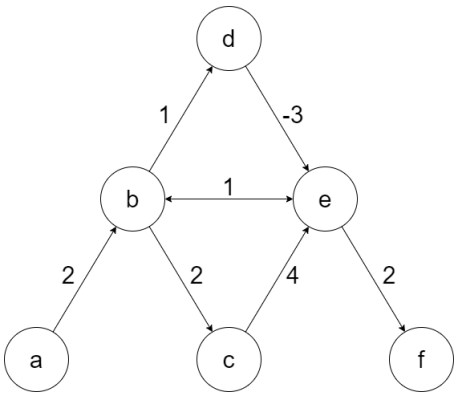


Figura 2.3: Ejemplo de grafo con un ciclo negativo

Fijemos a como nodo fuente y observemos que en G, el ciclo $C := (b, d, e, b)$ es un ciclo negativo, ya que $p(C) = 1 + (-3) + 1 = -1$. Esto quiere decir que cada vez que recorramos C, el peso del camino disminuirá en 1, luego no existe un camino de peso mínimo que

parta de a y acabe en cualquier otro nodo de G, ya que siempre podremos recorrer C una vez más y disminuir el peso del camino.

2.2.3. Representación de los caminos más cortos

El objetivo de los algoritmos que estudiaremos en este trabajo no será solamente calcular el peso de un camino más corto, sino que también nos interesa representar en sí el camino más corto en cuestión. Para ello, introducimos el concepto de predecesor: Dado un grafo $G = (V, A, p)$, el *predecesor* de $v \in V$, denotado π_v, es o bien otro nodo del grafo o bien NIL. Los algoritmos que vamos a usar asignan como predecesor de v el nodo que ocupa el lugar anterior a v en el camino más corto obtenido. De esta forma, el siguiente procedimiento, al que llamamos $IC(G, s, v)$, imprime un camino más corto desde un nodo fuente s hasta el nodo v:

- Si s y v coinciden, entonces imprimir s.
- Si π_v es NIL, entonces no existe ningún camino desde s hasta v, por lo que detenemos el procedimiento o imprimimos un mensaje de error.
- En otro caso, realizar $IC(G, s, \pi_v)$ y seguidamente imprimir v.

2.3– Algoritmo de Dijkstra

Este algoritmo fue ideado por Edsger W. Dijkstra en 1956, aunque fue publicado en 1959 [6]. Originalmente fue creado para resolver el problema del camino más corto entre dos nodos dados, pero la variante más común fija un nodo fuente y encuentra el camino más corto hasta el resto de nodos del grafo, es decir, resuelve el problema SSSP. El algoritmo de Dijkstra también puede ser empleado para resolver el problema SPSP de la siguiente manera: consideramos uno de los dos nodos fijados como nodo fuente, y detenemos el algoritmo una vez que se encuentre un camino más corto desde el nodo fuente hasta el otro nodo, considerado destino. Puede aplicarse tanto a grafos dirigidos como no dirigidos, y con pesos que pueden ser números enteros o reales no negativos.

Dijkstra encontró esta solución al problema del camino más corto en 1956 después de haber definido él mismo el problema. En ese entonces, Dijkstra trabajaba como programador principal en el *Mathematisch Centrum* de Ámsterdam, cuando la construcción de una nueva computadora automática llamada ARMAC estaba a punto de terminar. Para demostrar las capacidades de la máquina, se buscaba un problema fácilmente entendible por el público no informático para que el ordenador lo resolviera. Para ello, Dijkstra dibujó un mapa simplificado del sistema ferroviario holandés, alguien en el público preguntaba cuál era la ruta más corta entre dos ciudades, y el ordenador devolvía el trayecto más corto en menos de un minuto.

Un año más tarde, el siguiente ordenador del Instituto estaba en construcción, y los ingenieros de hardware se enfrentaron a otro problema: minimizar la cantidad de cable necesaria para conectar las clavijas de la parte trasera de la máquina. Como solución, idearon un algoritmo conocido como el algoritmo del árbol de expansión mínima de Prim (ya conocido por Jarnik y redescubierto por Prim). Dijkstra publicó este algoritmo también en 1959, dos años después de Prim [13] y 29 años después de Jarnik [14].

2.3.1. Complejidad

En cuanto a la complejidad del algoritmo, esta depende del número de nodos y aristas en el grafo y la forma en que se implementa. En el peor caso, la complejidad del algoritmo de Dijkstra es $O((n+m)\log n)$, donde n es el número de nodos en el grafo y m es el número de aristas. Esta complejidad se alcanza cuando se utiliza una estructura de datos llamada "cola de prioridad" para almacenar y recuperar los nodos más cercanos al nodo inicial en cada iteración del algoritmo. La operación de inserción y extracción en una cola de prioridad es de $O(\log n)$, lo que hace que la complejidad total sea $O((n+m)\log n)$. Sin utilizar cola de prioridad, la complejidad sería $O(n^2)$. En la práctica, el tiempo de ejecución del algoritmo puede variar ampliamente según la implementación específica y la complejidad del grafo.

2.3.2. Descripción del algoritmo

Sea $G = (V, A, p)$ un grafo con pesos no negativos y $a \in V$ el nodo inicial o fuente. La idea del algoritmo consiste en asignar una distancia inicial infinita a cada nodo del grafo (excepto a a, al que le asignamos el valor 0), e ir recorriendo los caminos que parten de a y mejorando las distancias de sus nodos, mediante un proceso denominado *relajación*. Más detalladamente, el algoritmo es el siguiente:

- Inicializamos $C = V$. En cada iteración del algoritmo, C contendrá los nodos que aún no han sido visitados.
- Asignamos a cada nodo del grafo un valor llamado distancia tentativa en una lista $D = \{d_v \mid v \in V\}$, donde inicialmente $d_a = 0$ y $d_v = \infty, \forall v \in V \setminus \{a\}$. Durante la ejecución del algoritmo, la distancia tentativa de v es la distancia del camino más corto descubierto hasta el momento entre a y v. Por otro lado, inicializamos la lista $P = \{\pi_v \mid v \in V\}$ de predecesores como $\pi_v = \text{NIL}$, $\forall v \in V$.
- Mientras $C \neq \varnothing$:
  - Escogemos el elemento $v \in C$ con menor distancia tentativa almacenada en D.
  - Eliminamos v de C (lo marcamos como visitado).
  - Recorremos todos los nodos adyacentes a v que todavía estén en C. Para cada uno de estos nodos w, actualizamos su distancia tentativa d_w y su predecesor π_w de la siguiente forma:
    - Si $d_v + p(v, w) < d_w$, cambiamos el valor de d_w por $d_v + p(v, w)$ y de π_w por v.
    - En otro caso, mantenemos el valor de d_w y de π_w.

Al final del algoritmo, D contendrá las distancias (es decir, los pesos) de los caminos más cortos desde el nodo fuente a hasta cada nodo v y P los predecesores de cada nodo v en un camino más corto desde a.

2.3.3. Prueba de corrección

A continuación probamos la corrección del algoritmo de Dijkstra, tomando para las demostraciones ideas de [11] y [18].

Teorema 2.2 (Corrección del algoritmo de Dijkstra)**.** *El algoritmo de Dijkstra, sobre un grafo $G = (V, A, p)$ con pesos no negativos y un nodo fuente $a \in V$, termina con $d_v = \delta(a, v)$ para todo $v \in V$.*

Demostración. Vamos a usar la siguiente hipótesis invariante, y la demostraremos por inducción sobre el número de iteraciones realizadas del algoritmo de Dijkstra:

Antes de cada iteración del algoritmo, para cada nodo visitado $v \notin C$, se tiene que $d_v = \delta(a, v)$, y para cada nodo no visitado $u \in C$, d_u es el peso del camino más corto desde a hasta u formado únicamente por nodos visitados, o ∞ si un tal camino no existe.

En el caso base, es decir, antes de la primera iteración, no existen nodos visitados; $d_a = 0$, que es el peso del camino que va desde a hasta a, y $d_u = \infty$ para el resto de nodos.

Supongamos ahora que la hipótesis es cierta tras $k - 1$ iteraciones (es decir, antes de la k-ésima iteración) y vamos a demostrar que es cierta tras realizar la k-ésima iteración.

Sea u el nodo visitado en la iteración k. Supongamos por reducción al absurdo que $d_u \neq \delta(a, u)$, es decir, el camino encontrado por el algoritmo no es un camino más corto. Si P es un camino más corto desde a hasta u, entonces distinguimos dos casos:

- *Caso 1: P contiene otro nodo no visitado.* Sea w el primer nodo de P no visitado, y sea x su predecesor en P, que sí ha sido visitado anteriormente, por lo que $d_x = \delta(a, x)$. De acuerdo con el algoritmo, tras procesar x se tiene que $d_w \leq d_x + p(x, w)$. Luego $d_w \leq \delta(a, x) + p(x, w)$. Como P es un camino más corto, $\delta(a, x) + p(x, w) = \delta(a, w)$, por lo que deducimos que $d_w \leq \delta(a, w)$. Como siempre se da la desigualdad contraria, necesariamente debe darse la igualdad: $d_w = \delta(a, w)$. Por ser de nuevo P un camino más corto, la distancia mínima de ir desde a hasta u no puede ser menor que la distancia desde a hasta w, por lo que

  $$d_w = \delta(a, w) \leq \delta(a, u) \leq d_u.$$

  Por otra parte, $d_u \leq d_w$, ya que el algoritmo elige el nodo de distancia tentativa menor en cada iteración, por lo que las desigualdades anteriores deben ser en realidad igualdades. De ahí, deducimos que $d_u = \delta(a, u)$, contradiciendo nuestra suposición inicial.

- *Caso 2: P no contiene otros nodos no visitados.* Sea w el penúltimo nodo (visitado) de P, es decir, w es el predecesor de u en P. Por la hipótesis de inducción, tenemos que:

  $$d_w + p(w, u) = \delta(a, w) + p(w, u) = \delta(a, u) < d_u.$$

  Pero como w es visitado, y u es adyacente a w, el algoritmo nos asegura que d_u sea como máximo $d_w + p(w, u)$, lo cual es una contradicción.

Para todos los demás nodos visitados v, por la hipótesis de inducción $d_v = \delta(a, v)$, y el algoritmo no cambia esto.

Nótese que los únicos nodos no visitados w para los que cambia d_w en la k-ésima iteración son los adyacentes a u tales que $d_u + p(u, w) < d_w$. Si $d_u + p(u, w)$ no fuera la mínima distancia por nodos visitados entre a y w, existiría un camino más corto de nodos visitados entre a y w que no contiene a u, pero en ese caso, el algoritmo ya habría actualizado el valor de d_w y por la hipótesis de inducción sería la distancia más corta por nodos visitados. Para el resto, se mantiene la hipótesis de inducción.

Tras ser visitados todos los nodos, el camino más corto desde a hasta cualquier nodo $v \in V$ está formado únicamente por nodos visitados, luego $d_v = \delta(a, v)$. El algoritmo termina ya que $C = \varnothing$. □

Teorema 2.3. *El camino C_π dado por el procedimiento $IC(G, a, v)$ tras aplicar el algoritmo de Dijkstra es un camino más corto entre $a \in V$ y $v \in V$.*

Demostración. Sea $C_\pi = (v_0, v_1, \ldots, v_k)$ donde $v_0 = a$ y $v_k = v$ dicho camino. Para $i = 1, \ldots, k$, tenemos que $d_{v_i} = \delta(a, v_i)$. Por construcción de C_π, sabemos que $\pi_{v_i} = v_{i-1}$, y se puede demostrar que cuando el algoritmo de Dijkstra termina, $d_{v_i} = d_{v_{i-1}} + p(v_{i-1}, v_i)$, luego deducimos que $p(v_{i-1}, v_i) = \delta(a, v_i) - \delta(a, v_{i-1})$. Por tanto:

$$p(C_\pi) = \sum_{i=1}^{k} p(v_{i-1}, v_i) = \sum_{i=1}^{k} (\delta(a, v_i) - \delta(a, v_{i-1})) = \delta(a, v_k) - \delta(a, v_0) = \delta(a, v)$$

Luego $p(C_\pi) = \delta(a, v)$ y por tanto C_π es un camino más corto entre a y v. □

2.4– Algoritmo de Bellman-Ford

Este algoritmo es usado para resolver el problema SSSP para dígrafos ponderados. Es más lento que el algoritmo de Dijkstra para el mismo problema, pero es más versátil, ya que puede resolver el problema del camino más corto no sólo para grafos con pesos positivos, sino también para grafos con pesos en $\mathbb{R}$. Fue inicialmente propuesto por Alfonso Shimbel en 1955, pero, sin embargo, debe su nombre a Richard Bellman [4] y a Lester Ford Jr. [5], quienes lo publicaron en 1958 y 1956, respectivamente.

El algoritmo de Bellman-Ford, a diferencia del de Dijkstra, puede no encontrar una solución para el problema SSSP debido a que al considerar grafos con pesos negativos, puede no existir un camino más corto entre un nodo fuente y el resto de nodos del grafo. Esta situación se da, como ya vimos, si se puede alcanzar un ciclo negativo desde el nodo fuente.

2.4.1. Complejidad

La complejidad del algoritmo de Bellman-Ford depende del número de nodos y aristas del grafo. Concretamente, la complejidad del algoritmo de Bellman-Ford es $O(nm)$ en el peor caso, donde n es el número de nodos en el grafo y m es el número de aristas. Esto se debe a que el algoritmo itera $n - 1$ veces sobre todas las aristas en el grafo para

encontrar el camino más corto desde el nodo de origen a todos los demás nodos. En cada iteración, el algoritmo examina cada arista en el grafo y actualiza los caminos más cortos si es necesario. En la peor de las situaciones, esto implica revisar cada una de las m aristas $n-1$ veces.

En consecuencia, el algoritmo de Bellman-Ford tiene una complejidad asintótica relativamente alta, pero puede manejar grafos con pesos negativos y detectar ciclos negativos. Es una buena opción cuando se necesita una solución más general y flexible en lugar de la eficiencia, y el grafo no es demasiado grande o denso.

2.4.2. Descripción del algoritmo

Sea $G = (V, A, p)$ un grafo ponderado con pesos en $\mathbb{R}$ y $a \in V$ el nodo fuente. El algoritmo de Bellman-Ford inicializa una estimación de las distancias desde el nodo fuente hasta el resto de los nodos (las distancias tentativas) de la misma manera que el algoritmo de Dijkstra, así como la lista de predecesores. La diferencia es que ahora se relajan todas las aristas del grafo $n-1$ veces, siendo $n = |V|$ el número de nodos de G y, seguidamente, se hace una comprobación final para asegurar que el grafo no tiene ciclos negativos alcanzables desde a. Más concretamente, el algoritmo es el siguiente:

- Asignamos a cada nodo v del grafo una distancia tentativa inicial $d_v = \infty$ si $v \neq a$ y $d_a = 0$, y las almacenamos en una lista $D = \{d_v \mid v \in V\}$. Del mismo modo inicializamos los predecesores $\pi_v = \text{NIL}\ \forall v \in V$ y los guardamos en una lista $P = \{\pi_v \mid v \in V\}$.
- Para $i \in \{1, \ldots, |V| - 1\}$:
  - Para cada arista $(u, v) \in A$:
    - Si $d_u + p(u, v) < d_v$, cambiamos el valor de d_v por $d_u + p(u, v)$ y el de π_v por u.
    - En otro caso, el valor de d_v y de π_v se mantiene.
- Para cada arista $(u, v) \in A$:
  - Si $d_u + p(u, v) < d_v$, entonces detenemos el algoritmo, ya que hemos detectado un ciclo negativo.
  - En otro caso, continuamos.

Al final del algoritmo, D contendrá las distancias de los caminos más cortos desde el nodo fuente a hasta cada nodo v y P los predecesores de cada nodo v en un camino más corto desde a.

2.4.3. Prueba de corrección

Para demostrar este primer lema, tomamos ideas de [19].

Lema 2.4. *Tras i repeticiones del primer bucle, se tiene que:*

- *Si $d_u \neq \infty$, entonces d_u es la distancia de algún camino entre el nodo fuente a y u.*
- *Si hay algún camino desde a hasta u con i nodos o menos, entonces d_u es como máximo la distancia de un camino más corto desde a hasta u con i nodos o menos.*

Demostración. Haremos la demostración por inducción sobre i:

Para $i = 0$, es decir, antes de la primera iteración del bucle, tenemos que $d_a = 0$ y $d_u = \infty$ si $u \neq a$, por lo que se cumplen trivialmente ambas afirmaciones ya que no hay caminos desde a hasta u con 0 nodos.

Para el caso inductivo, probaremos primero la primera parte: si v es un nodo cuya distancia tentativa ha sido actualizada en la i-ésima iteración (si hubiera sido actualizada anteriormente, por hipótesis de inducción ya sería la distancia de un camino desde a hasta v), entonces $d_v = d_u + p(u, v)$ para algún $u \in V$. Por hipótesis de inducción, d_u es la distancia de algún camino $(a, \ldots, u)$. Por tanto, d_v es la distancia del camino $(a, \ldots, u, v)$.

Probemos ahora la segunda parte: Sea P un camino más corto desde a hasta v con i nodos como máximo. Sea u el vértice anterior a v en P. Luego $P' = (a, \ldots, u)$ es un camino más corto con $i - 1$ nodos o menos (ya que los subcaminos de caminos más cortos son caminos más cortos). Por hipótesis de inducción, tras $i - 1$ iteraciones se tiene que $d_u \leq p(P')$, luego $d_u + p(u, v) \leq p(P)$. En la iteración i, comparamos d_v con $d_u + p(u, v)$, y asignamos $d_v = d_u + p(u, v)$ si es menor. Luego tras i iteraciones, $d_v \leq p(P)$, es decir, d_v es como máximo la distancia de un camino más corto desde a hasta v con i nodos como máximo. □

La demostración del siguiente lema es según el teorema 2.3.9. de [12].

Lema 2.5. *G no contiene ciclos negativos alcanzables desde a si y solo si no puede haber mejoras de las distancias tentativas en el segundo bucle del algoritmo.*

Demostración. Si no hay ciclos negativos alcanzables desde a en G, entonces podemos asumir que cada camino más corto P desde a visita cada nodo como máximo una vez (es decir, P no contiene ciclos ya que ninguno tiene peso negativo y, si tuviera peso nulo, escogeríamos el camino que no recorre el ciclo). Luego por el Lema 2.4, no podemos mejorar los valores de las distancias tentativas (para los nodos no alcanzables desde a, las distancias tentativas se mantienen iguales a ∞ durante todo el algoritmo).

Recíprocamente, supongamos que no podemos mejorar los valores de las distancias tentativas en el segundo bucle. Entonces sea $C = (v_0 \ldots, v_{k-1}, v_k = v_0)$ un ciclo en G. Se tiene:

$$d_{v_i} \leq d_{v_{i-1}} + p(v_{i-1}, v_i), \forall i = 1, \ldots, k.$$

Sumando en $i = 1, \ldots, k$,

$$\sum_{i=1}^{k} d_{v_i} \leq \sum_{i=1}^{k} d_{v_{i-1}} + \sum_{i=1}^{k} p(v_{i-1}, v_i).$$

Los términos $\sum_{i=1}^{k} d_{v_i}$ y $\sum_{i=1}^{k} d_{v_{i-1}}$ cancelan y nos queda $0 \leq \sum_{i=1}^{k} p(v_{i-1}, v_i) = p(C)$, por lo que C es no negativo. □

Teorema 2.6 (Corrección del algoritmo de Bellman-Ford)**.** *Dado un grafo ponderado $G = (V, A, p)$ con pesos en $\mathbb{R}$, nodo fuente $a \in V$ y sin ciclos negativos alcanzables desde a, el algoritmo de Bellman-Ford termina con $d_v = \delta(a, v), \forall v \in V$.*

Demostración. Sigue inmediatamente de los Lemas 2.4 y 2.5. □

Teorema 2.7. *El camino C_π dado por el procedimiento $IC(G, a, v)$ tras aplicar el algoritmo de Bellman-Ford es un camino más corto entre $a \in V$ y $v \in V$.*

Demostración. Análoga a la de 2.3. □

2.4.4. Comparación con el algoritmo de Dijkstra

A continuación se muestra una tabla comparativa entre el algoritmo de Dijkstra y el algoritmo de Bellman-Ford:

Características	**Dijkstra**	**Bellman-Ford**
Tipo de problema	SSSP	SSSP
Pesos	Admite pesos no negativos	Admite pesos en todo $\mathbb{R}$
Complejidad	$O((n+m)\log n)$	$O(nm)$
Detecta ciclos negativos	No	Sí

Tabla 2.1: Comparación entre el algoritmo de Dijkstra y el de Bellman-Ford

En resumen, el algoritmo de Dijkstra es más eficiente en grafos con pesos no negativos y puede proporcionar una solución más rápida en estos casos. Por otro lado, el algoritmo de Bellman-Ford puede manejar grafos con pesos negativos y detectar ciclos negativos en el grafo, lo que lo hace más adecuado para algunos tipos de problemas.

2.5– Algoritmo de búsqueda A*

El algoritmo de búsqueda A*, o simplemente A* (leído *A estrella*, *A asterisco* o en inglés *A star*), a diferencia de los dos algoritmos vistos anteriormente, resuelve el problema SPSP para grafos con pesos positivos, es decir, fija un nodo fuente y un nodo destino y encuentra un camino más corto entre ambos. Fue creado en 1968 por Peter E. Hart, Nils J. Nilsson y Bertram Raphael [10] como parte del proyecto Shakey, un robot móvil que pudiera planear sus propias acciones.

A* puede considerarse una extensión del algoritmo de Dijkstra pero, a diferencia de este, usa una *heurística* para guiar su búsqueda, lo que mejora notablemente su velocidad y rendimiento, siendo este algoritmo la mejor solución al problema del camino más corto en muchos casos. A diferencia de un enfoque puramente matemático, que generalmente

estudia las propiedades abstractas de, en este caso, los grafos para desarrollar técnicas y algoritmos que permitan resolver el problema, el enfoque heurístico usa información adicional sobre el dominio de aplicación del grafo en cuestión para mejorar la eficiencia computacional de una solución al problema. Por ejemplo, en el problema de encontrar la ruta más corta entre dos ciudades en una red de carreteras, esta información adicional o heurística podría ser la distancia en línea recta entre la ciudad destino y el resto de nodos de la red (o mejor dicho, la distancia a lo largo del arco de círculo máximo que pasa por ambos nodos, si consideramos que la Tierra no es plana). Esta información viene dada por la llamada función heurística que, bajo ciertas condiciones, asegura que el algoritmo de búsqueda A* termine encontrando un camino más corto.

Debido a que el algoritmo A* usa una heurística específicamente dirigida al nodo destino, es un algoritmo que resuelve el problema SPSP, a diferencia del algoritmo de Dijkstra, que genera un árbol completo de caminos más cortos basados en un nodo fuente.

2.5.1. Complejidad

La complejidad del algoritmo de búsqueda A* depende de varios factores, como el factor de ramificación promedio del grafo, la calidad de la función heurística utilizada y la posición del nodo objetivo en el grafo.

En el peor de los casos, el algoritmo de búsqueda A* puede recorrer todos los nodos adyacentes de cada nodo, luego el algoritmo no será mejor que el de Dijkstra. Sin embargo, en la práctica se pueden encontrar acotaciones mucho mejores. Esto sucede cuando el grafo y la función heurística satisfacen unas determinadas propiedades. En ese caso, podemos realizar suposiciones que dan lugar a una complejidad mejor.

Por ejemplo, si la función heurística fuera "perfecta", es decir, que la estimación de la distancia desde cada nodo hasta el nodo destino fuera exactamente igual a dicha distancia en el grafo, entonces el algoritmo A* tendría la mejor complejidad posible, ya que sólo recorrería los nodos que componen el camino más corto. Sin embargo, esta es una suposición demasiado fuerte para la gran mayoría de aplicaciones, pero nos da una idea de cómo la heurística puede ayudar a mejorar la complejidad del algoritmo.

En conclusión, no podemos dar una respuesta corta a la pregunta de cuál es la complejidad del algoritmo de búsqueda A*, ya que, como veremos, éste consiste en una modificación del algoritmo de Dijkstra guiado por una función heurística que puede cambiar en cada caso.

2.5.2. Descripción del algoritmo

Sea $G = (V, A, p)$ un grafo dirigido ponderado con pesos no negativos, $a \in V$ un nodo fuente y $t \in V$ un nodo destino. Para poder aplicar el algoritmo A*, debemos conocer información extra del grafo que nos permita estimar el peso mínimo del camino entre cada nodo y el nodo destino. Esta información viene dada por la *función heurística* $h : V \longrightarrow \mathbb{R}$, donde $h(v)$ es dicha estimación de la *distancia* entre v y t. El algoritmo A* consiste en:

1. Inicializamos los valores $g(a) = 0$ y $f(a) = g(a) + h(a)$. Para el resto de nodos $v \in V \setminus \{a\}$, hacemos $f(v) = \infty$ y guardamos todos estos valores en una lista $D =$

$\{f(v) \mid v \in V\}$. Por otra parte, inicializamos $\pi_v = \text{NIL}$, $\forall v \in V$ y guardamos dichos valores en la lista $P = \{\pi_v \mid v \in V\}$ de predecesores. Finalmente, inicializamos el conjunto $S = \{a\}$, al que llamaremos *conjunto abierto* o *conjunto de nodos abiertos*, y $T = \varnothing$, llamado *conjunto de nodos cerrados*.

2. Seleccionamos $v \in S$ tal que $f(v) \leq f(u)$, $\forall u \in S$, eliminamos v de S y lo añadimos a T.

3. Si $v = t$, terminar el algoritmo. En otro caso, saltar al cuarto paso.

4. Para cada nodo w adyacente a v, si $g(w) > g(v) + p(v, w)$, actualizamos $g(w) = g(v) + p(v, w)$, $f(w) = g(w) + h(w)$ y $\pi_w = v$. Si $w \notin T$, lo añadimos a S. Si $w \in T$ y su valor de f ha sido actualizado, lo eliminamos de T y lo añadimos a S. Tras realizar todas las actualizaciones, volver al paso 2.

Obsérvese que al marcar el nodo v como abierto, es decir, añadirlo a S, el algoritmo asigna una distancia temporal a v dada por la función de evaluación

$$f(v) = g(v) + h(v).$$

Esta función de evaluación se compone de dos partes:

- $g(v)$ es una estimación de la distancia entre el nodo fuente a y el nodo v, y siempre se tiene que $g(v) \geq \delta(a, v)$.
- $h(v)$, como ya vimos, es una estimación de la distancia entre el nodo v y el nodo destino t.

La función g equivale a la estimación de la distancia entre el nodo fuente y el nodo procesado que hacíamos en el algoritmo de Dijkstra d_v, que llamábamos distancia tentativa. De hecho, si $h(v) = 0, \forall v \in V$, el algoritmo A* se reduce básicamente al algoritmo de Dijkstra. Por otra parte, intuitivamente podemos pensar en la función heurística h como una especie de penalización sobre los nodos del grafo, donde los nodos más cercanos al destino t son menos penalizados que los que se encuentran más lejos de t. Esta componente h de la función de evaluación ayuda a dirigir la búsqueda hacia el nodo destino, algo que no sucede en el algoritmo de Dijkstra, donde se expande la búsqueda uniformemente en todas las direcciones. Tanto el algoritmo de Dijkstra como A* identifican y procesan los nodos en orden de distancia, aunque el de Dijkstra lo hace en orden ascendente de distancia desde el nodo fuente, mientras que A* también tiene en cuenta una estimación de la distancia hasta el nodo destino gracias a la función heurística.

Hasta el momento no hemos dado una definición concreta de la función heurística h ni hemos dicho nada sobre su implementación. En el ejemplo que dimos en la introducción del algoritmo de búsqueda A*, consideramos un grafo cuyos nodos representan ciertas ciudades en un mapa y las aristas las carreteras entre ciudades vecinas, y nos planteamos el problema de encontrar la ruta más corta entre dos ciudades dadas. En este caso, podríamos utilizar como valor de la función heurística la distancia en línea "recta" entre cada ciudad y la ciudad destino. Obsérvese que esta distancia es la menor distancia posible de cualquier

carretera que conecte una ciudad v con la ciudad objetivo t, por lo que es una cota inferior de $\delta(v,t)$. Esta condición, como veremos más adelante, nos servirá para asegurar la *admisibilidad* del algoritmo, es decir, la garantía de que el algoritmo encuentre un camino más corto entre el nodo fuente y el destino en cualquier grafo con pesos no negativos.

2.5.3. Prueba de corrección

En toda esta sección, tomamos las ideas de las demostraciones del artículo original [10]. Sea $v \in V$ y consideremos la función de evaluación usada en A*, $f(v) = g(v) + h(v)$, donde $g(v)$ es el peso del camino entre el nodo fuente $a \in V$ y v con el menor peso encontrado por A*, y $h(v)$ es una estimación del peso del camino más corto entre v y el nodo destino t. Primero probaremos un lema:

Lema 2.8. *Para cualquier nodo* no cerrado *v (es decir, $v \notin T$) y para cualquier camino más corto C entre a y v, existe un nodo* abierto *v' (es decir, $v' \in S$) en el camino C tal que $g(v') = \delta(a, v')$.*

Demostración. Sea $C = (a = v_0, v_1, \ldots, v = v_k)$. Si a es abierto, es decir, A* no ha completado ninguna iteración, entonces se tiene el resultado ya que $g(a) = \delta(a,a) = 0$.

Supongamos ahora que a es cerrado. Sea A el conjunto de nodos $v_i \in C$ tales que $g(v_i) = \delta(a, v_i)$ y v_i es cerrado. Claramente $A \neq \varnothing$ ya que $a \in A$. Sea $\bar{v}$ el elemento de A con mayor índice. Nótese que $\bar{v} \neq v$ ya que v no es cerrado. Sea v' el sucesor de $\bar{v}$ en C. Entonces tenemos que $g(v') \leq g(\bar{v}) + p(\bar{v}, v')$ por definición de g. Como $\bar{v} \in A$, $g(\bar{v}) = \delta(a, \bar{v})$ y como C es un camino más corto, $\delta(a, v') = \delta(a, \bar{v}) + p(\bar{v}, v')$. Por tanto, $g(v') \leq \delta(a, \bar{v}) + p(\bar{v}, v') = \delta(a, v')$, pero como sabemos que $g(v') \geq \delta(a, v')$, entonces necesariamente tenemos que $g(v') = \delta(a, v')$. Además, por construcción de A y de acuerdo con el algoritmo, v' debe ser abierto. □

Corolario 2.9. *Supongamos $h(v) \leq \delta(v,t)$, $\forall v \in V$, y supongamos que A* no ha terminado. Entonces, para cualquier camino más corto C desde a hasta t, existe un nodo $v' \in C$ abierto tal que $f(v') \leq \delta(a,t)$.*

Demostración. Por el lema anterior, con $v = t$ no cerrado ya que el algoritmo no ha terminado, $\exists v' \in C$ abierto tal que $g(v') = \delta(a, v')$. Por definición de f,

$$f(v') = g(v') + h(v') = \delta(a, v') + h(v') \leq \delta(a, v') + \delta(v', t).$$

Pero como $v' \in C$ y C es un camino más corto, tenemos que $\delta(a, v') + \delta(v', t) = \delta(a, t)$, de donde sigue el resultado. □

Teorema 2.10. *Sean $G = (V, A, p)$ un grafo con pesos positivos y $h : V \longrightarrow \mathbb{R}$ una función heurística no negativa sobre los nodos de G. Si $h(v) \leq \delta(v,t), \forall v \in V$, entonces el algoritmo de búsqueda A* es* admisible*, es decir, termina encontrando un camino más corto entre a y t.*

Demostración. Supongamos que, por el contrario, A* no termina encontrando un camino más corto entre a y t. Entonces, hay tres casos posibles: que el algoritmo termine en un

nodo distinto a t, que el algoritmo no termine, o que el algoritmo acabe en t pero sin alcanzar el peso mínimo.

- *Caso 1*: A* termina en un nodo distinto a t. Este caso contradice la condición de parada del algoritmo (paso 3), luego lo descartamos.

- *Caso 2*: A* no termina. Sea $\varepsilon := \min_{(u,v)\in A} p(u,v) > 0$ por hipótesis. Entonces, para cada nodo v a más de $M := \frac{\delta(a,t)}{\varepsilon}$ pasos desde a, tenemos que

$$f(v) \geq g(v) \geq \delta(a,v) > M\varepsilon = \delta(a,t).$$

  Claramente, ningún nodo v a más de M pasos de a es expandido, ya que por el Corolario 2.9, existiría un nodo v' en un camino más corto desde a hasta t tal que $f(v') \leq \delta(a,t) < f(v)$, por lo que el algoritmo escogería v' en el paso 2 en lugar de v.

  Luego A* no termina debido a la reapertura continua de nodos que están a M o menos pasos de a. Sea V_M el conjunto de nodos accesibles desde a en a lo más M pasos. Cualquier nodo $v \in V_M$ puede reabrirse como máximo un número finito de veces $k_M(v)$, ya que hay un número finito de caminos desde a hasta v pasando por nodos de V_M. Sea $k_M := \max_{v\in V_M} k_M(v)$ (el máximo número de veces que un nodo de V_M puede ser reabierto). Luego, tras como máximo $|V_M|k_M$ expansiones, todos los nodos de V_M deben quedar cerrados para siempre. Como no hay nodos fuera de V_M que puedan ser expandidos, A* debe terminar. Como esto contradice la suposición inicial, descartamos este caso.

- *Caso 3*: A* termina en t pero no alcanza la distancia mínima entre a y t. Supongamos por tanto que $f(t) = g(t) > \delta(a,t)$. Pero, por el Corolario 2.9., justo antes de la parada del algoritmo existiría un nodo abierto v' en un camino más corto C con $f(v') \leq \delta(a,t) < f(t)$. Luego, en este paso del algoritmo, v' debería haber sido elegido en lugar de t, por lo que el algoritmo no habría terminado.

Como no puede darse ninguno de los casos, concluimos que A* es admisible. □

Además de esta condición de admisibilidad, en [10] se prueba que si la función heurística h cumple otra condición adicional, entonces el algoritmo A* es óptimo, que a grandes rasgos significa que cualquier otro algoritmo de búsqueda expande más nodos que A*; y por otra parte A* no vuelve a etiquetar como abierto ningún nodo que ya sea cerrado, es decir, que cuando el algoritmo expande un nodo, entonces ya ha sido encontrado un camino más corto hasta ese nodo. Esta condición es llamada *condición de consistencia*, y se da si la función h satisface la siguiente "desigualdad triangular":

$$h(u) \leq \delta(u,v) + h(v), \forall u, v \in V.$$

2.5.4. Sobre la función heurística h

En realidad, podemos considerar el algoritmo A* como una *familia* de algoritmos, donde cada posible elección de la función heurística h define un algoritmo distinto. Dependiendo de la aplicación concreta del grafo que se esté usando, podría convenir hacer una elección de h u otra.

Como vimos anteriormente, la elección $h \equiv 0$ correspondería al caso en el que no se conoce ninguna información (o al menos no se usa) sobre el dominio del problema, a parte del grafo en sí y sus pesos. En el ejemplo que pusimos de una red de carreteras, al tener información sobre la localización de las ciudades, podemos considerar $h(v) = \sqrt{x^2 + y^2}$, donde x e y representan las diferencias entre las coordenadas de la ciudad v y la ciudad destino (en un mapa plano, ya que consideramos ciudades lo suficientemente cercanas como para despreciar la curvatura de la Tierra). Con esta h, el algoritmo también encontraría un camino más corto entre las ciudades, y además expandiría muchos menos nodos, lo que en principio supondría un coste computacional menor. Sin embargo, podría ocurrir que el coste computacional de calcular $\sqrt{x^2 + y^2}$ para cada nodo sea mayor que el que se ahorra por visitar menos nodos. Es por ello que, dependiendo del caso, puedan usarse otras elecciones de h, como por ejemplo $h(v) = \frac{x+y}{2}$. Como $\frac{x+y}{2} < \sqrt{x^2 + y^2}$, el algoritmo sigue siendo admisible y, aunque expanda más nodos, puede acabar ahorrando coste computacional total.

2.6– Algoritmo de Floyd-Warshall

El algoritmo de Floyd-Warshall, también conocido como algoritmo de Floyd, de Roy-Warshall o de Roy-Floyd, es un algoritmo que resuelve el problema del camino más corto entre todas las posibles parejas de nodos de un grafo, es decir, resuelve el problema APSP. Fue publicado por Robert W. Floyd [8] en 1962 basándose en un teorema sobre matrices booleanas demostrado por Sthephen Warshall [7], también en 1962. El algoritmo puede aplicarse a grafos dirigidos con pesos cualesquiera, siempre que no haya ciclos negativos.

Este algoritmo, además de para resolver el problema del camino más corto, puede usarse para calcular la clausura transitiva de un grafo, que es otro grafo con los mismos nodos que el original y aristas entre aquellos nodos para los cuales existe un camino entre ellos en el grafo original, es decir, es el menor grafo conteniendo las mismas aristas que el grafo original en el que se cumple la propiedad transitiva: Si $(u, v) \in A$ y $(v, w) \in A$, entonces $(u, w) \in A$.

Para el estudio de este algoritmo, nos basamos en la sección 23.2 de [11].

2.6.1. Caracterización de la estructura de los caminos más cortos

Para definir el algoritmo de Floyd-Warshall, caracterizaremos la estructura de los caminos más cortos de una forma distinta a como lo habíamos hecho anteriormente. Comenzaremos dando la definición de *nodo intermedio*:

Definición 2.11. Sea $G = (V, A, p)$ un grafo y $C = (v_1, \ldots, v_l)$ un camino en G. Decimos que $v \in V$ es un *nodo intermedio* de C si v está en C y es distinto de v_1 y v_l, es decir, es

alguno de los nodos del subcamino $C' = (v_2, \ldots, v_{l-1})$.

Sea $n := |V|$. A partir de ahora, asumiremos que $V = \{1, 2, \ldots, n\}$ y denotaremos los nodos de V por las letras i, j, k, etc. en lugar de por v, u, w, etc. Definimos el conjunto $V_k := \{1, 2, \ldots, k\} \subseteq V$ para $1 \leq k \leq n$. Además, asumiremos que G no contiene ciclos negativos.

Para cada par de nodos $i, j \in V$, y para k fijo, consideremos todos los caminos desde i hasta j con todos sus nodos intermedios en V_k y, en caso de existir, sea P un camino más corto de entre todos ellos. Como G no contiene ciclos negativos, podemos asumir que P pasa por cada nodo como máximo una vez. Distinguimos dos situaciones:

1. *k no es un nodo intermedio de P.* En este caso, todos los nodos intermedios de P pertenecen al conjunto $\{1, \ldots, k-1\} = V_{k-1}$. Por tanto, un camino más corto desde i hasta j con todos sus nodos intermedios en V_k también es un camino más corto desde i hasta j con todos sus nodos intermedios en V_{k-1}.

2. *k es un nodo intermedio de P.* En este caso, descomponemos P en $P_1 = (1, \ldots, k)$ y $P_2 = (k, \ldots, j)$. Al ser P un camino más corto entre i y j con nodos intermedios en V_k, P_1 es un camino más corto entre i y k con todos sus nodos intermedios en V_k. De hecho, como k no es un nodo intermedio de P_1, entonces los nodos intermedios de P_1 están en V_{k-1}. Análogamente, P_2 es un camino más corto entre k y j con nodos intermedios en V_{k-1}.

Esta es, esencialmente, la idea que hay detrás del algoritmo de Floyd-Warshall, que nos permite dar una solución recursiva al problema APSP.

2.6.2. Solución recursiva

Si $V = \{1, \ldots, n\}$ son los nodos de un grafo G, podemos representar las aristas y sus pesos por una matriz $n \times n$ $W = (w_{ij})_{ij}$ con $i, j \in V$. Los valores de w_{ij} vienen dados por:

$$w_{ij} = \begin{cases} 0 & \text{si } i = j \\ p(i,j) & \text{si } i \neq j, (i,j) \in A \\ \infty & \text{si } i \neq j, (i,j) \notin A. \end{cases} \tag{2.2}$$

Basándonos en las observaciones hechas en la sección 2.6.1, daremos una definición recursiva de los estimadores de los pesos de los caminos más cortos que vamos a usar en este algoritmo. Sea d_{ij}^k el peso de un camino más corto desde el nodo i hasta el nodo j con nodos intermedios en V_k. Definimos d_{ij}^k como:

$$d_{ij}^k = \begin{cases} w_{ij} & \text{si } k = 0 \\ \text{mín}(d_{ij}^{k-1}, d_{ik}^{k-1} + d_{kj}^{k-1}) & \text{si } k \geq 1. \end{cases} \tag{2.3}$$

Teniendo en cuenta el razonamiento de la sección anterior, no es difícil demostrar el siguiente teorema:

Teorema 2.11. *Sea $G = (V, A, p)$ un grafo, y w_{ij} y d_{ij}^k definidos como arriba. Para $i, j \in V$ y para $k = 0, \ldots, n = |V|$, sea $\delta_k(i, j)$ el peso del camino más corto desde i hasta j con todos sus nodos intermedios en $V_k := \{1, \ldots, k\}$. Entonces se tiene que $d_{ij}^k = \delta_k(i, j)$.*

Demostración. Probaremos el teorema por inducción sobre k:

- Para $k = 0$, un camino más corto desde i hasta j tiene como máximo una arista, luego $d_{ij}^0 = w_{ij} = \delta_0(i, j)$ es evidente.
- Supongamos que $d_{ij}^{k-1} = \delta_{k-1}(i, j), \forall i, j \in V$. Sea P un camino más corto entre los nodos i y j con nodos intermedios en V_k, es decir $p(P) = \delta_k(i, j)$. Pueden ocurrir dos situaciones, las dos que vimos en la sección 2.6.1. Si estamos en la primera, entonces $\delta_k(i, j) = \delta_{k-1}(i, j) = d_{ij}^{k-1}$. Si se da la segunda, $\delta_k(i, j) = \delta_{k-1}(i, k) + \delta_{k-1}(k, j) = d_{ik}^{k-1} + d_{kj}^{k-1}$. Por tanto, $p(P) = \delta_k(i, j) = \text{mín}(d_{ij}^{k-1}, d_{ik}^{k-1} + d_{kj}^{k-1}) = d_{ij}^k$.

□

Observemos que para cualquier camino en G, sus nodos intermedios están en $V_n = V$. Por tanto, la matriz $D^{(n)} = (d_{ij}^n)_{ij}$ da la respuesta final: $d_{ij}^n = \delta(i, j), \forall i, j \in V$.

2.6.3. Construcción de los caminos más cortos

De forma similar a como hicimos en la sección 2.2.3, para representar los caminos más cortos obtenidos por el algoritmo, nos basaremos en el concepto de predecesor. Definimos π_{ij}^k como el predecesor del nodo j en un camino más corto entre los nodos i y j con todos sus nodos intermedios en V_k. Al igual que hemos hecho anteriormente para d_{ij}^k, podemos dar una definición recursiva de π_{ij}^k.

Si $k = 0$, es decir, consideramos caminos de, como máximo, una arista, definimos π_{ij}^0 como:

$$\pi_{ij}^0 = \begin{cases} \text{NIL} & \text{si } i = j \text{ o } w_{ij} = \infty \\ i & \text{si } i \neq j \text{ y } w_{ij} < \infty. \end{cases} \tag{2.4}$$

Para $k \geq 1$, si tomamos el camino $(i, \ldots, k, \ldots, j)$ con $k \neq j$, entonces el predecesor de j que elegimos es el mismo que el predecesor de j que habíamos elegido en un camino más corto desde k hasta j con todos sus nodos intermedios en V_{k-1}. En otro caso, elegimos el predecesor de j en el camino más corto de i a j con nodos intermedios en V_{k-1}. Es decir, para $k \geq 1$:

$$\pi_{ij}^k = \begin{cases} \pi_{ij}^{k-1} & \text{si } d_{ij}^{k-1} \leq d_{ik}^{k-1} + d_{jk}^{k-1} \\ \pi_{kj}^{k-1} & \text{si } d_{ij}^{k-1} > d_{ik}^{k-1} + d_{jk}^{k-1}. \end{cases} \tag{2.5}$$

Se tiene el siguiente resultado:

Teorema 2.12. *Para todo $k \in \{1, \ldots, n\}$ y para todos $i, j \in V$, π_{ij}^k es el predecesor de j en un camino más corto desde i hasta j con todos sus nodos intermedios en V_k.*

Demostración. Procedemos por inducción. Para $k = 0$, la tesis es trivial. Para $k \geq 1$, supongamos la tesis cierta para $k-1$ y sea P un camino más corto entre i y j con sus nodos intermedios en V_k. Distinguimos dos casos:

- Si $d_{ij}^{k-1} \leq d_{ik}^{k-1} + d_{kj}^{k-1}$, podemos suponer que P no contiene al nodo k, ya que de acuerdo con el Teorema 2.11, el primer miembro de la desigualdad corresponde al peso de un camino más corto entre i y j con nodos intermedios en V_{k-1} (y por tanto en V_k); y el segundo miembro corresponde al peso de un camino más corto entre i y j que sí contenga a k. En este caso, por hipótesis de inducción, tenemos que el predecesor de j en P es π_{ij}^{k-1} que, de acuerdo con 2.5, coincide con π_{ij}^{k}.
- Si $d_{ij}^{k-1} > d_{ik}^{k-1} + d_{kj}^{k-1}$, podemos asumir que k pertenece a P, ya que en caso contrario, la desigualdad implica que existiría un camino $(i, \ldots, k, \ldots, j)$ con nodos intermedios en V_k con peso menor que el de P, contradiciendo que P es un camino más corto entre i y j con nodos intermedios en V_k. Luego si P contiene a k, el predecesor de j en P es el predecesor de j en el subcamino $(k, \ldots, j)$ de P, que es un camino más corto con nodos intermedios en V_{k-1}. Por hipótesis de inducción, este predecesor es π_{kj}^{k-1}. Como en este caso, por 2.5 se tiene $\pi_{kj}^{k-1} = \pi_{ij}^{k}$, se prueba el resultado.

□

Tras n iteraciones, la matriz $\Pi^{(n)} = (\pi_{ij}^{n})_{ij}$ contendrá en la posición (i, j) el predecesor del nodo j en un camino más corto que comienza en el nodo i. Si tomamos $\Pi = (\pi_{ij}) := \Pi^{(n)}$, para cada par $i, j \in V$, el siguiente procedimiento $IC'(\Pi, i, j)$ imprime un camino más corto entre los nodos i y j:

- Si $i = j$, entonces imprimir i.
- Si $\pi_{ij} = \text{NIL}$, entonces no existe ningún camino desde i hasta j, por lo que detenemos el procedimiento o imprimimos un mensaje de error.
- En otro caso, realizar $IC'(\Pi, i, \pi_{ij})$ y seguidamente imprimir j.

2.6.4. Descripción del algoritmo

Sea $G = (V, A, p)$ un grafo sin ciclos negativos con $|V| = n$. El algoritmo de Floyd-Warshall es el siguiente:

- Inicializamos dos matrices de dimensiones $n \times n$ de la siguiente forma: $D^{(0)} = (d_{ij}^{0})$ con d_{ij}^{0} como en 2.3 y $\Pi^{(0)} = (\pi_{ij}^{0})$ con π_{ij}^{0} como en 2.4.
- Para $k \in \{1, \ldots, n\}$, calculamos las matrices $D^{(k)} = (d_{ij}^{k})$, que es la matriz de distancias más cortas encontradas entre cada par de nodos; y $\Pi^{(k)} = (\pi_{ij}^{k})$, que es la matriz de predecesores en caminos más cortos entre i y j encontrados.
  - Para $i, j \in \{1, \ldots, n\}$, calculamos d_{ij}^{k} y π_{ij}^{k} de acuerdo con 2.3 y 2.5 respectivamente.

- Al final del algoritmo, devolvemos las matrices $D^{(n)}$ y $\Pi^{(n)}$.

Obsérvese que los Teoremas 2.11 y 2.12 prueban la corrección del algoritmo. Tras ejecutarlo, podemos usar el procedimiento $IC'(\Pi, i, j)$ para imprimir los caminos más cortos encontrados por el algoritmo.

2.6.5. Complejidad

Sabiendo cómo funciona el algoritmo de Floyd-Warshall una vez descrito en la sección anterior, es fácil ver que la complejidad del algoritmo es de $O(n^3)$, con n el número de nodos del grafo. Esto se debe a que realizamos tres bucles anidados, cada uno de los cuales iterando n veces. El algoritmo actualiza el camino más corto entre todos los pares de nodos n veces, y en cada actualización, considera todos los posibles nodos intermedios. Esto resulta en un total de n^3 operaciones.

En resumen, el algoritmo de Floyd-Warshall es una forma eficiente de encontrar el camino más corto entre todos los pares de vértices en un grafo, pero su complejidad de tiempo puede ser prohibitiva para grafos grandes.

2.7– Algoritmo de Johnson

El algoritmo de Johnson debe su nombre a Donald B. Johnson [9], quien lo publicó en 1977. Resuelve el PCC entre cada par de nodos de un grafo dirigido con pesos cualesquiera (pero sin ciclos negativos). Así, es un algoritmo que resuelve el problema APSP. Para ello, usa el algoritmo de Bellman-Ford para eliminar todos los pesos negativos del grafo, y seguidamente utiliza el algoritmo de Dijkstra en el grafo transformado. Al ser la complejidad del algoritmo de Dijkstra menor que la del de Bellman-Ford, esta estrategia nos permite encontrar una solución al problema APSP de forma más eficiente que calcular el algoritmo de Bellman-Ford tomando como nodo fuente cada nodo del grafo.

Como ya estudiamos anteriormente, sabemos que el algoritmo de Dijkstra resuelve el problema del camino más corto para grafos con pesos no negativos. Si $G = (V, A, p)$ es un grafo con pesos cualesquiera, buscamos encontrar una nueva función de pesos $\hat{p} : A \longrightarrow \mathbb{R}$ tal que nos permita aplicar el algoritmo de Dijkstra y resolver el PCC correctamente. Esta función $\hat{p}$ debe cumplir dos condiciones:

1. $\forall u, v \in V$, $C = (u, \ldots, v)$ es un camino más corto entre u y v con la función de pesos p si y solo si lo es con la función $\hat{p}$.
2. $\forall (u, v) \in A$, $\hat{p}(u, v) \geq 0$.

En el estudio de este algoritmo, nos basamos en la sección 23.3 de [11].

2.7.1. Reponderación

Así como δ denota los pesos de los caminos más cortos usando la función de pesos p, la nueva función de ponderación $\hat{p}$ induce unos nuevos pesos mínimos $\hat{\delta}$. El siguiente lema,

correspondiente al lema 23.1 de [11], muestra cómo podemos reponderar las aristas de un grafo para que se cumpla la primera condición vista anteriormente.

Lema 2.13 (La reponderación no cambia los caminos más cortos)**.** *Sean $G = (V, A, p)$ un grafo dirigido con peso $p : A \longrightarrow \mathbb{R}$ y una aplicación $h : V \longrightarrow \mathbb{R}$ cualquiera que asocia valores reales a los nodos de G. Para $(u, v) \in A$, definimos*

$$\hat{p}(u, v) := p(u, v) + h(u) - h(v). \tag{2.6}$$

Si $C = (v_0, v_1, \ldots, v_k)$ es un camino en G, entonces C es un camino más corto para p si y solo si lo es para $\hat{p}$. Es decir, $p(C) = \delta(v_0, v_k) \iff \hat{p}(C) = \hat{\delta}(v_0, v_k)$.

Además, G tiene un ciclo negativo con p si y solo si $\hat{G} := (V, A, \hat{p})$ lo tiene con $\hat{p}$.

Demostración. Comenzaremos probando que $\hat{p}(C) = p(C) + h(v_0) - h(v_k)$:

$$\begin{aligned} \hat{p}(C) &= \sum_{i=1}^{k} \hat{p}(v_{i-1}, v_i) = \sum_{i=1}^{k} (p(v_{i-1}, v_i) + h(v_{i-1}) - h(v_i)) = \\ &= \sum_{i=1}^{k} p(v_{i-1}, v_i) + h(v_0) - h(v_k) = p(C) + h(v_0) - h(v_k). \end{aligned} \tag{2.7}$$

Luego, para cualquier camino C desde v_0 hasta v_k se tiene que $\hat{p}(C) = p(C) + h(v_0) - h(v_k)$. Como $h(v_0)$ y $h(v_k)$ no dependen de C, si un camino es más corto que otro usando p, también lo es usando $\hat{p}$. Por tanto, $p(C) = \delta(v_0, v_k) \iff \hat{p}(C) = \hat{\delta}(v_0, v_k)$.

Sea ahora un ciclo $C = (v_0, v_1, \ldots, v_k)$ con $v_0 = v_k$. Por 2.7, $\hat{p}(C) = p(C) + h(v_0) - h(v_k) = p(C)$. Luego $p(C) \leq 0 \iff \hat{p}(C) \leq 0$. □

Ahora queremos asegurarnos de que se cumple la segunda condición, es decir, que todos los nuevos pesos sean no negativos. Dado un grafo $G = (V, A, p)$ dirigido y con pesos cualesquiera, consideramos un nuevo grafo $G' = (V', A', p)$ donde $V' := V \cup \{s\}$, donde $s \notin V$ es un nuevo nodo; $A' := A \cup \{(s, v) \mid v \in V\}$ y p es extendida a A' de tal forma que $p(s, v) = 0, \forall v \in V$. Obsérvese que como s no tiene aristas entrantes, solo forma parte de caminos que salen de él. Además, G no contiene ciclos negativos si y solo si G' tampoco.

Asumiendo que ni G ni G' tienen ciclos negativos, definimos h como $h(v) := \delta(s, v), \forall v \in V$. Observar que como existen todas las aristas $(s, v) \in A'$, entonces $h(v) < \infty$. Como siempre se tiene la desigualdad triangular, es decir, $\delta(s, v) \leq \delta(s, u) + p(u, v)$, entonces tenemos que $h(v) \leq h(u) + p(u, v), \forall (u, v) \in A'$. Por tanto, para $\hat{p}$ definida como en 2.6, tenemos que

$$\hat{p}(u, v) = p(u, v) + h(u) - h(v) \geq 0. \tag{2.8}$$

2.7.2. Descripción del algoritmo

El algoritmo de Johnson utiliza los algoritmos de Bellman-Ford y de Dijkstra para calcular los caminos más cortos entre cada par de nodos de un grafo. Dado un grafo $G = (V, A, p)$ dirigido y con pesos cualesquiera, el algoritmo o bien encuentra un ciclo negativo o bien devuelve una matriz $D = (d_{ij})_{ij}$ de dimensiones $|V| \times |V|$ con las distancias

más cortas entre cada par de nodos, así como una matriz $\Pi = (\pi_{ij})_{ij}$ de predecesores. En concreto, el algoritmo es el siguiente:

- En primer lugar, calcular el grafo G', con $V' = V \cup \{s\}$, $A' = A \cup \{(s, v) \mid v \in V\}$ y $p(s, v) = 0, \forall v \in V$.
- Ejecutar el algoritmo de Bellman-Ford sobre G' con s como nodo fuente.
  - Si detecta un ciclo negativo, detener el algoritmo.
  - En caso contrario:
    - Para cada $v \in V$, definir $h(v) = \delta(s, v)$, valor calculado por el algoritmo de Bellman-Ford.
    - Para cada $(u, v) \in A$, definir $\hat{p}(u, v) = p(u, v) + h(u) - h(v)$.
    - Definir dos matrices $D = (d_{uv})$ y $\Pi = (\pi_{uv})$ de dimensiones $|V| \times |V|$.
    - Para cada $u \in V$, ejecutar el algoritmo de Dijkstra sobre $\hat{G} = (V, A, \hat{p})$ con u como nodo fuente para calcular $\hat{\delta}(u, v)$ y el predecesor de v en un camino más corto desde u para todos los nodos $v \in V$.
      - Para cada $v \in V$, definir $d_{uv} = \hat{\delta}(u, v) + h(v) - h(u)$ y $\pi_{uv} = \pi_v$, con π_v el predecesor de v en un camino más corto desde u hasta v encontrado por el algoritmo de Dijkstra.
- Finalmente, devolver D y Π.

2.7.3. Prueba de corrección

Teorema 2.14. *Dado un grafo $G = (V, A, p)$ sin ciclos negativos, el algoritmo de Johnson encuentra las distancias más cortas d_{uv} entre cada par de nodos u y v de G y los predecesores π_{uv} en los caminos más cortos.*

Demostración. Por 2.8, sabemos que los nuevos pesos son no negativos, luego podemos aplicar el algoritmo de Dijkstra y, por el Lema 2.13, sabemos que al reponderar las aristas del grafo, los caminos más cortos no se ven afectados. Además, por 2.7 sabemos que $\hat{\delta}(u, v) = \delta(u, v) + h(u) - h(v), \forall u, v \in V$. Finalmente, los Teoremas 2.6, 2.2 y 2.3 nos aseguran la corrección del algoritmo. □

2.7.4. Complejidad y comparación con Floyd-Warshall

Como hemos visto, para encontrar el camino más corto entre todos los pares de vértices en un grafo, el algoritmo de Johnson utiliza el algoritmo de Bellman-Ford y el algoritmo de Dijkstra. Ya vimos que el algoritmo de Bellman-Ford se utiliza para encontrar la distancia mínima desde un vértice de origen a todos los demás nodos del grafo. Este paso se realiza agregando un nuevo nodo con aristas de peso cero a todos los vértices del grafo y aplicando el algoritmo de Bellman-Ford a partir de este nuevo vértice. El algoritmo de Bellman-Ford tiene una complejidad de tiempo de $O(nm)$, donde n es el número de vértices y m es el número de aristas en el grafo.

Luego, el algoritmo de Johnson realiza una reasignación de pesos de las aristas del grafo original utilizando las distancias mínimas encontradas por el algoritmo de Bellman-Ford. Esta reasignación asegura que todos los pesos de las aristas en el grafo sean no negativos. Después de la reasignación, el algoritmo de Dijkstra se utiliza para encontrar el camino más corto entre todos los pares de vértices en el grafo ponderado con pesos no negativos. El algoritmo de Dijkstra tiene una complejidad de tiempo de $O((n+m)\log n)$ en el peor caso. Por tanto, la complejidad total del algoritmo de Johnson queda $O(n(n+m)\log n + nm)$.

En términos de complejidad de tiempo, el algoritmo de Floyd-Warshall, que es el otro algoritmo de tipo APSP que hemos estudiado, tiene un tiempo de ejecución más lento que el algoritmo de Johnson para grafos dispersos, es decir, aquellos cuyo número de aristas es relativamente bajo. La complejidad de tiempo del algoritmo de Johnson es de $O(n(n+m)\log n + nm)$, mientras que la complejidad de tiempo del algoritmo de Floyd-Warshall es de $O(n^3)$. Sin embargo, para grafos densos, o sea, grafos con relativamente un gran número de aristas, el algoritmo de Floyd-Warshall puede ser más eficiente. Por tanto, debemos tener en cuenta que dado un problema del camino más corto en concreto, la elección del algoritmo para resolverlo dependerá de las características específicas del grafo que se está analizando.

CAPÍTULO 3

Diseño de la librería

En este capítulo describiremos la librería sobre grafos que hemos creado para este trabajo. El código se desarrolla en Haskell, un lenguaje de programación funcional puro con evaluación perezosa. Además, Haskell cuenta con una fuerte tipificación estática, o sea, el lenguaje controla que todas las operaciones se realicen únicamente para los datos cuyo tipo sea efectivamente aquel para el que la operación esté definida. Esto ayuda a detectar errores de tipo antes de que el programa se ejecute, mejorando la seguridad y confiabilidad. También cabe destacar que Haskell cuenta con un sistema de inferencia de tipos que puede deducir automáticamente los tipos de las expresiones. Sin embargo, nosotros especificaremos el tipo de las funciones que aparecen en la librería para evitar problemas.

Como veremos a continuación, hemos considerado dos maneras diferentes de definir grafos en nuestra librería: en una primera versión definimos lo que hemos llamado grafos *algebraicos* mediante una definición recursiva, tomando la librería `Algebra.Graph` [16] de Haskell como inspiración; mientras que en una segunda versión definimos los grafos a partir de una lista de nodos y otra de aristas, lo que ayuda a mejorar la eficiencia. Como el objetivo final de este trabajo es la resolución del problema del camino más corto, la construcción de grafos que haremos únicamente contemplará grafos ponderados. Eligiendo cualquiera de las dos versiones, para cada una de las cuales hemos creado un módulo Haskell, posteriormente implementaremos en un tercer módulo los algoritmos que resuelven el problema del camino más corto estudiados en el capítulo anterior.

3.1– Módulo de grafos algebraicos (primera versión)

Como hemos comentado anteriormente, en esta primera versión definimos los grafos de forma recursiva. Para ello, nos basamos en la idea de "unión de grafos":

Definición 3.1. Dados dos grafos ponderados $G_1 = (V_1, A_1, p_1)$ y $G_2 = (V_2, A_2, p_2)$ tales que $p_1|_{A_1 \cap A_2} = p_2|_{A_1 \cap A_2}$, definimos el *grafo unión* de G_1 y G_2, $G_1 \cup G_2$, como $(V_1 \cup V_2, A_1 \cup A_2, p)$, donde $p : A_1 \cup A_2 \longrightarrow \mathbb{R}$ está definida de la siguiente manera:

$$p(a) = \begin{cases} p_1(a) & \text{si } a \in A_1 \\ p_2(a) & \text{si } a \in A_2. \end{cases}$$

Nótese que esta definición difiere ligeramente de lo que usualmente se entiende como *unión disjunta* o simplemente *unión* de grafos, donde se requiere que los conjuntos de nodos (y por tanto de aristas) sean disjuntos. En nuestro caso, nos interesa que, de alguna forma, obtengamos un grafo resultante de *superponer* los anteriores para ir construyendo grafos cada vez más grandes, no necesariamente con varias componentes conexas distintas. La idea, como veremos explícitamente más adelante, consiste en considerar los nodos y las aristas de un grafo como subgrafos más pequeños, para posteriormente superponerlos mediante esta operación de unión y así obtener el grafo original.

A continuación, procederemos a comentar la implementación en Haskell de este módulo sobre grafos algebraicos. Esta primera versión puede encontrarse en el archivo `GrafoAlgebraico.hs`.

3.1.1. Declaración del módulo

Lo primero que haremos será declarar el módulo Haskell que estamos creando, especificando las funciones y tipos de datos que queremos exportar posteriormente para usar en otros programas:

Código 3.1: Declaración del primer módulo

```
1  module GrafoAlgebraico
2    (Orientacion (..),
3     Grafo,
4     vacio,
5     grafo,
6     orientacion,
7     adyacentes,
8     nodos,
9     aristas,
10    aristasConPeso,
11    aristaEn,
12    nNodos,
13    nAristas,
14    peso,
15    pesoCamino,
16    completo
17   ) where
```

Código 3.1: Declaración del primer módulo

3.1.2. Librerías

En este módulo hemos usado las librerías predefinidas de Haskell `Data.List` y `Data.Set`, para tratar con listas y con conjuntos.

Código 3.2: Librerías usadas en la primera versión

```
import Data.List as L
import Data.Set as S
```

Código 3.2: Librerías usadas en la primera versión

3.1.3. Tipos de datos

Los tipos de datos que hemos definido son `Orientacion`, que nos permite distinguir entre grafos dirigidos (`D`) y no dirigidos (`ND`); y `Grafo v p`, que representa grafos con nodos de tipo `v` y pesos de tipo `p`. Distinguimos cuatro tipos de grafos: el grafo vacío (`Vacio`), grafos formados por un nodo (`Nodo`), grafos formados por una arista ponderada (`Arista`) y, por último, grafos que sean la unión de dos grafos del tipo que se está definiendo (`Une`). Cabe comentar que en realidad `Arista` relaciona dos grafos completos con un peso `p`, algo que es más complejo que simplemente un grafo formado por una arista, pero que en la práctica siempre consideraremos dos grafos que sean nodos.

Código 3.3: Tipos de datos de la primera versión

```
data Orientacion = D
                 | ND
                 deriving (Eq, Show)

data Grafo v p = Vacio
               | Nodo v
               | Arista p (Grafo v p) (Grafo v p)
               | Une (Grafo v p) (Grafo v p)
```

Código 3.3: Tipos de datos de la primera versión

3.1.4. Funciones de construcción

A continuación detallaremos diferentes funciones a las que hemos llamado *funciones de construcción*, ya que son las que nos permiten construir grafos.

La función `vacio` crea el grafo vacío:

Código 3.4: Función `vacio`

```
vacio :: Num p => Grafo v p
vacio = Vacio
```

Código 3.4: Función `vacio`

La función `nodo` construye un grafo formado únicamente por un nodo `v`:

Código 3.5: Función `nodo`

```
nodo :: v -> Grafo v p
nodo v = Nodo v
```

Código 3.5: Función `nodo`

La función `arista` crea el grafo formado únicamente por una arista desde un nodo `u` hasta otro nodo `v`, con un peso `p`:

Código 3.6: Función `arista`

```
arista :: v -> v -> p -> Grafo v p
arista u v p = Arista p (nodo u) (nodo v)
```

Código 3.6: Función `arista`

La función `une` crea el grafo unión de dos grafos `g1` y `g2`:

Código 3.7: Función `une`

```
une :: Grafo v p -> Grafo v p -> Grafo v p
une g1 g2 = Une g1 g2
```

Código 3.7: Función `une`

A partir de estas cuatro funciones básicas, podemos definir la función `grafo`, que será la que usaremos para crear un grafo dadas una orientación `o`, cualquier lista de nodos `vs` y cualquier lista de aristas con pesos `as`.

Código 3.8: Función `grafo`

```
grafo :: (Eq v, Num p) => Orientacion -> [v] -> [(v,v,p)] -> Grafo v p
grafo _ [] _ = vacio
grafo _ [v] _ = nodo v
grafo D vs as =
  let asg = [arista u v p | (u,v,p) <- as]
      as1 = [(u,v) | (u,v,p) <- as]
      vsg = [nodo v | v <- vs, and (L.map (not . enPar v) as1)]
  in cgauxFL (asg ++ vsg)
grafo ND vs as =
  let asg = [arista u v p | (u,v,p) <- as, u /= v] ++ [arista v u p | (u,v,p) <-
      as]
      as2 = [(u,v) | (u,v,p) <- as] ++ [(v,u) | (u,v,p) <- as]
      vsg = [nodo v | v <- vs, and (L.map (not . enPar v) as2)]
  in cgauxFL (asg ++ vsg)


enPar :: Eq v => v -> (v,v) -> Bool
enPar v a = v == fst a || v == snd a


cgauxFL gs =
  L.foldl (\ ac g -> une g ac) vacio gs
```

Código 3.8: Función `grafo`

Observemos que para crear grafos dirigidos, basta considerar las aristas que aparecen en la lista `as`, mientras que para construir grafos no dirigidos consideramos las aristas de `as` y también las aristas que van en sentido contrario.

La función `enPar` es una función auxiliar que determina si un nodo es uno de los elementos de un par de nodos (es decir, una arista no ponderada). Por otro lado, tenemos

otra función auxiliar llamada `cgauxFL`, que construye por plegado un grafo resultante de la unión de una lista de grafos.

3.1.5. Funciones de descripción

En esta sección trataremos sobre diversas funciones que hemos llamado *funciones de descripción*, ya que actúan sobre grafos para obtener alguna información de ellos, como por ejemplo determinar cuáles son sus nodos, obtener el peso de una arista, etc.

La primera función que definimos es `nodos`, que devuelve la lista de nodos del grafo `g`:

Código 3.9: Función `nodos`

```
1 nodos :: (Eq v) => Grafo v p -> [v]
2 nodos Vacio = []
3 nodos (Nodo v) = [v]
4 nodos (Arista p u v) = L.union (nodos u) (nodos v)
5 nodos (Une g1 g2) = L.union (nodos g1) (nodos g2)
```

Código 3.9: Función `nodos`

De igual forma, `aristas g` devuelve la lista de aristas del grafo:

Código 3.10: Función `aristas`

```
1 aristas :: Eq v => Grafo v p -> [(v,v)]
2 aristas Vacio = []
3 aristas (Nodo v) = []
4 aristas (Arista p (Nodo u) (Nodo v)) = [(u,v)]
5 aristas (Une g1 g2) = L.union (aristas g1) (aristas g2)
```

Código 3.10: Función `aristas`

Además, si queremos obtener la lista de aristas de un grafo y sus pesos, tenemos la función `aristasConPeso`:

Código 3.11: Función `aristasConPeso`

```
1 aristasConPeso :: (Eq v, Eq p) => Grafo v p -> [((v,v),p)]
2 aristasConPeso Vacio = []
3 aristasConPeso (Nodo v) = []
4 aristasConPeso (Arista p (Nodo u) (Nodo v)) = [((u,v),p)]
5 aristasConPeso (Une g1 g2) = L.union (aristasConPeso g1) (aristasConPeso g2)
```

Código 3.11: Función `aristasConPeso`

La función `aristaEn` devuelve el valor `True` si `a` es una de las aristas un grafo `g`, y `False` en caso contrario.

Código 3.12: Función `aristaEn`

```
1 aristaEn :: (Eq v) => (v,v) -> Grafo v p -> Bool
2 aristaEn a g = elem a (aristas g)
```

Código 3.12: Función `aristaEn`

Las funciones `nNodos` y `nAristas` calculan el número de nodos y de aristas de un grafo, respectivamente.

Código 3.13: Funciones `nNodos` y `nAristas`

```
1 nNodos :: Eq v => Grafo v p -> Int
2 nNodos g = length $ nodos g
3
4
5 nAristas :: (Eq v, Eq p) => Grafo v p -> Int
6 nAristas g = length $ aristas g
```

Código 3.13: Funciones `nNodos` y `nAristas`

La siguiente función `orientacion` devuelve la orientación del grafo `g`, es decir, devuelve el valor `D` si es dirigido y `ND` si no lo es.

Código 3.14: Función `orientacion`

```
1 orientacion :: (Eq v, Eq p) => Grafo v p -> Orientacion
2 orientacion g =
3   if and [elem ((v,u),p) (aristasConPeso g) | ((u,v),p) <- aristasConPeso g]
4   then ND
5   else D
```

Código 3.14: Función `orientacion`

Obsérvese que en la definición de la función `orientacion`, la línea 3 se corresponde con la caracterización de grafo no dirigido que dimos en 2.1. Así, cualquier grafo $G = (V, A, p)$ en el que si $(u, v) \in A$ entonces $(v, u) \in A$ y $p(u, v) = p(v, u)$ es considerado no dirigido, aunque haya sido definido como dirigido usando la función `grafo`. Por otra parte, el grafo vacío y los grafos sin aristas son trivialmente no dirigidos, y un grafo formado únicamente por una arista siempre es dirigido.

Siguiendo con las definiciones de funciones, la función `adyacentes` devuelve la lista de nodos adyacentes al nodo `v` en el grafo `g`:

Código 3.15: Función `adyacentes`

```
1 adyacentes :: (Eq v, Eq p) => Grafo v p -> v -> [v]
2 adyacentes g v =
3   let as = aristas g
4   in L.map snd (L.filter (\ (a,b) -> a == v) as)
```

Código 3.15: Función `adyacentes`

Finalmente, la función `peso` devuelve el peso de la arista `a` en el grafo `g`. Además, si la arista no pertenece al grafo, devuelve un mensaje de error. Por otra parte, la función `pesoCamino` calcula el peso de un camino en un grafo.

Código 3.16: Funciones `peso` y `pesoCamino`

```
1 peso :: (Eq v, Eq p) => Grafo v p -> (v,v) -> p
2 peso g a =
3   if elem a (aristas g)
```

```
4   then snd (head (L.filter (\(x,y) -> a == x) (aristasConPeso g)))
5   else error "La arista no pertenece al grafo."
6
7
8  pesoCamino :: (Eq v, Eq p, Num p) => Grafo v p -> [v] -> p
9  pesoCamino g [] = 0
10 pesoCamino g [v] = 0
11 pesoCamino g (v1:v2:vs) = peso g (v1,v2) + pesoCamino g (v2:vs)
```

Código 3.16: Funciones `peso` y `pesoCamino`

3.1.6. Otras funciones

También hemos incluido en el módulo la función `completo`, que construye el grafo completo con un determinado número de nodos, es decir, un grafo donde cada par de nodos está conectado por una arista.

Código 3.17: Función `completo`

```
1  completo :: Num p => Int -> Grafo Int p
2  completo n =
3    grafo ND [1..n] [(i,j,1) | i <- [1..n], j <- [i..n]]
```

Código 3.17: Función `completo`

3.1.7. Igualdad de grafos

Las siguientes líneas de código sirven para distinguir si dos grafos son iguales o no, esto es, si sus conjuntos de nodos y de aristas son iguales.

Código 3.18: Igualdad de grafos

```
1  instance (Ord v, Eq v, Ord p, Eq p) => Eq (Grafo v p) where
2    x == y = ((fromList (nodos x) == fromList (nodos y)) &&
3              (fromList (aristasConPeso x) == fromList (aristasConPeso y)))
```

Código 3.18: Igualdad de grafos

3.1.8. Escritura de los grafos

El siguiente procedimiento sirve para representar en pantalla los grafos de una forma simple, mostrando la orientación del grafo, así como sus conjuntos de nodos y aristas.

Código 3.19: Procedimiento de escritura de los grafos

```
1  instance (Eq v, Eq p, Num p, Show v, Show p) => Show (Grafo v p) where
2    show g = let aps = [(a,peso g a) | a <- aristas g]
3                 o = orientacion g
4             in "Grafo["++(show o)++"], N:{"++(showNodos (nodos g))++
5                "}, A:{"++(showAristas aps)++"}"
```

```
6
7  showNodos [] = ""
8  showNodos [v] =
9    (show v)
10 showNodos (v:vs) =
11   (show v)++","++(showNodos vs)
12
13 showAristas [] = ""
14 showAristas [(a,p)] =
15   (show a)++"["++(show p)++"]"
16 showAristas ((a,p):aps) =
17   (show a)++"["++(show p)++"],"++(showAristas aps)
```

Código 3.19: Procedimiento de escritura de los grafos

Así, si por ejemplo quisiéramos definir un grafo dirigido y con pesos nulos con conjunto de nodos $\{1, 2, 3, 4\}$ y conjunto de aristas $\{(1, 2), (2, 3), (2, 4), (3, 1), (3, 4)\}$, en pantalla veríamos:

Código 3.20: Salida: escritura de los grafos

```
1     > grafo D [1..4] [(1,2,0),(2,3,0),(2,4,0),(3,1,0),(3,4,0)]
2  Grafo[D], N:{3,4,1,2}, A:{(3,4)[0],(3,1)[0],(2,4)[0],(2,3)[0],(1,2)[0]}
```

Código 3.20: Salida: escritura de los grafos

3.1.9. Eficiencia

La razón principal de haber añadido la función `completo` es para experimentar con esta construcción de grafos y ver cómo de eficiente es. Con el comando `:set +s` podemos ver en pantalla la cantidad de memoria usada y el tiempo de ejecución de cada comando. Por ejemplo, creando un grafo completo de 10 nodos obtenemos lo siguiente (omitimos el output):

Código 3.21: Salida: Tiempo de ejecución y memoria usada de `completo 10`, 1ª versión

```
1     > completo 10
2  --
3  (0.46 secs, 95,564,112 bytes)
```

Código 3.21: Salida: Tiempo de ejecución y memoria usada de `completo 10`, 1ª versión

Ya con un grafo completo de sólo 10 nodos, tardamos casi medio segundo en cargarlo. Si aumentamos ligeramente el número de nodos, tenemos que:

Código 3.22: Salida: Tiempo de ejecución y memoria usada de `completo 15`, 1ª versión

```
1     > completo 15
2  --
3  (10.78 secs, 985,788,552 bytes)
```

Código 3.22: Salida: Tiempo de ejecución y memoria usada de `completo 15`, 1ª versión

Por lo que vemos que construir grafos grandes se hace bastante pesado para la máquina con esta implementación. Es por eso que hemos decidido definir los grafos en Haskell de una segunda forma, para ver si podemos ganar algo de eficiencia.

3.2– Módulo de grafos con listas (segunda versión)

En esta segunda versión, definimos los grafos a partir de la lista de sus nodos y de la lista de sus aristas, con sus correspondientes pesos. Las librerías predefinidas de Haskell que usamos son las mismas que antes (`Data.List` y `Data.Set`), así como el procedimiento de escritura de los grafos y de igualdad de grafos. Por ello, en esta sección solamente hablaremos de los tipos de datos definidos, de las funciones de construcción y de las funciones de descripción. El archivo en el que se encuentra esta versión es `GrafoLista.hs`.

3.2.1. Tipos de datos

Además de los tipos de datos `Orientacion` y `Grafo`, añadiremos el tipo `Arista` para representar aristas ponderadas. Así, representamos los grafos con el nuevo tipo de dato `Grafo`, a partir de una lista de nodos y de una lista de aristas.

Código 3.23: Tipos de datos en la segunda versión

```haskell
type Arista v p = ((v,v),p)

data Orientacion = D | ND
                 deriving (Show, Eq)

data Grafo v p = G [v] [Arista v p]
```

Código 3.23: Tipos de datos en la segunda versión

3.2.2. Funciones de construcción

Las funciones de construcción que consideraremos serán la función `vacio`, que crea un grafo sin nodos ni aristas; y la función `grafo`, que toma como argumentos una orientación (`D` o `ND`), una lista de nodos y una lista de tripletas asociadas a las aristas del grafo, cuyas dos primeras entradas son los nodos de la arista y la tercera corresponde al peso de esta. Esta función construye un `Grafo`, teniendo en cuenta si el grafo es dirigido o no. Las funciones `aristasBienDefinidas` y `aristaSinPeso` son funciones auxiliares.

Código 3.24: Funciones de construcción de la segunda versión

```haskell
vacio :: Grafo v p
vacio = G [] []

grafo :: Eq v => Orientacion -> [v] -> [(v,v,p)] -> Grafo v p
grafo D vs as = if aristasBienDefinidas vs as
                then G vs [((u,v),p) | (u,v,p) <- as]
                else error "Las aristas no están bien definidas."
grafo ND vs as = if aristasBienDefinidas vs as
```

```
                then G vs ([((u,v),p) | (u,v,p) <- as, u /= v]
                     ++ [((v,u),p) | (u,v,p) <- as])
                else error "Las aristas no están bien definidas."

aristasBienDefinidas :: Eq v => [v] -> [(v,v,p)] -> Bool
aristasBienDefinidas vs as =
  and [ a `elem` vs && b `elem` vs | (a,b) <- L.map aristaSinPeso as]

aristaSinPeso (a,b,c) = (a,b)
```

Código 3.24: Funciones de construcción de la segunda versión

3.2.3. Funciones de descripción

A continuación se muestran las funciones de descripción que, como vemos, son las mismas que en la versión anterior, solo que algunas de ellas debemos modificarlas para adaptarlas a esta nueva construcción de los grafos.

Código 3.25: Funciones de descripción de la segunda versión

```
nodos :: (Eq v) => Grafo v p -> [v]
nodos (G vs _) = vs

aristas :: Eq v => Grafo v p -> [(v,v)]
aristas (G _ as) = L.map aristaSinPeso2 as

aristaSinPeso2 ((a,b),c) = (a,b)

aristasConPeso :: Eq v => Grafo v p -> [Arista v p]
aristasConPeso (G _ as) = as

aristaEn :: (Eq v) => (v,v) -> Grafo v p -> Bool
aristaEn a g = elem a (aristas g)

nNodos :: Eq v => Grafo v p -> Int
nNodos g = length $ nodos g

nAristas :: (Eq v, Eq p) => Grafo v p -> Int
nAristas g = length $ aristas g

orientacion :: (Eq v, Eq p) => Grafo v p -> Orientacion
orientacion g =
  let as = aristasConPeso g
  in if and [elem ((v,u),p) as | ((u,v),p) <- as]
     then ND
     else D

adyacentes :: Eq v => Grafo v p -> v -> [v]
adyacentes g v =
  let vs = nodos g
      as = aristas g
  in L.map snd [(a,b) | (a,b) <- as, a == v]
```

```haskell

peso :: (Eq v, Eq p, Num p) => Grafo v p -> (v,v) -> p
peso g a =
  let as = aristasConPeso g
      as2 = aristas g
  in if a `elem` as2
     then head $ L.map snd [(c,p) | (c,p) <- as, c == a]
     else error "La arista no pertece al grafo."

pesoCamino :: (Eq v, Eq p, Num p) => Grafo v p -> [v] -> p
pesoCamino g [] = 0
pesoCamino g [v] = 0
pesoCamino g (v1:v2:vs) =
  let as = aristas g
  in if elem (v1,v2) as
     then peso g (v1,v2) + pesoCamino g (v2:vs)
     else error "El camino no existe."
```

Código 3.25: Funciones de descripción de la segunda versión

3.2.4. Eficiencia

En esta sección vamos a repetir el experimento que realizamos en la primera versión construyendo grafos completos. Usamos de nuevo el comando `:set +s`, y seguidamente observamos lo siguiente:

Código 3.26: Salida: Tiempo de ejecución y memoria usada de `completo 10`, 2ª versión

```
   > completo 10
--
(0.02 secs, 2,190,656 bytes)
```

Código 3.26: Salida: Tiempo de ejecución y memoria usada de `completo 10`, 2ª versión

Comparando con 3.1.9, vemos que tanto el tiempo de ejecución como la cantidad de memoria usada son significativamente mejores. Si repetimos para 15 nodos:

Código 3.27: Salida: Tiempo de ejecución y memoria usada de `completo 15`, 2ª versión

```
   > completo 15
--
(0.03 secs, 7,962,344 bytes)
```

Código 3.27: Salida: Tiempo de ejecución y memoria usada de `completo 15`, 2ª versión

Con la versión anterior, para construir el grafo completo de 15 nodos tardamos 10.78 segundos, unas 360 veces más que con esta. Tenemos que aumentar algo más el número de nodos para que el tiempo y cantidad de memoria usada también aumenten:

Código 3.28: Saldia: Tiempo de ejecución y memoria usada de `completo 50`, 2ª versión

```
   > completo 50
--
```

```
(1.70 secs, 703,336,816 bytes)
```

Código 3.28: Saldia: Tiempo de ejecución y memoria usada de `completo 50`, 2ª versión

En la figura 3.1 se muestra una gráfica comparativa de los tiempos de ejecución de `completo n` usando los módulos `GrafoAlgebraico` o `GrafoLista`.

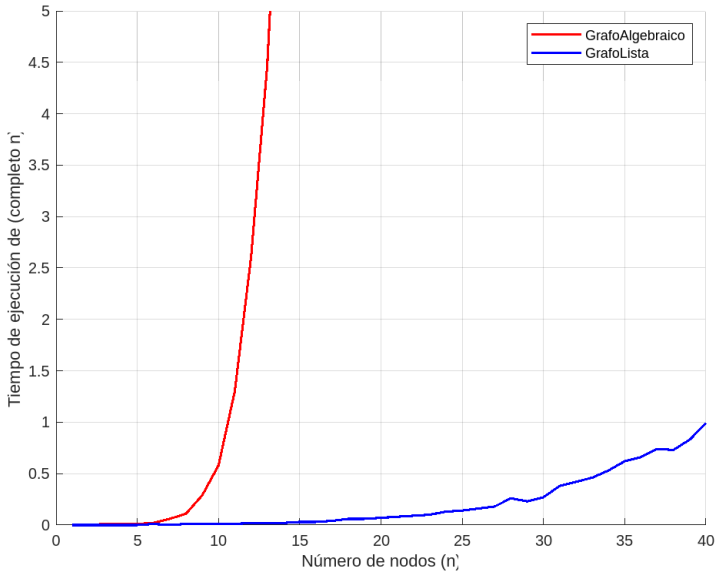


Figura 3.1: Comparación de eficiencia entre `GrafoAlgebraico` y `GrafoLista`

3.3– Implementación de los algoritmos que resuelven el PCC

En esta sección, finalizaremos el desarrollo de la librería implementando en Haskell los algoritmos que resuelven el problema del camino más corto estudiados en el segundo capítulo. Lo haremos en el fichero `AlgoritmosGrafos.hs`.

3.3.1. Declaración del módulo

En primer lugar, declaramos el módulo con las funciones y tipos de datos que queramos que incluya:

Código 3.29: Declaración del módulo de algoritmos

```
module AlgoritmosGrafos
  (DT,
   dist,
   inf,
   sumaD,
   dijkstra,
   imprCaminoD,
   caminoConPesoD,
   bellmanford,
   imprCaminoBF,
```

```
11    caminoConPesoBF,
12    astar,
13    floydwarshall,
14    imprCaminoFW,
15    caminoConPesoFW,
16    johnson,
17    imprCaminoJ,
18    caminoConPesoJ) where
```

Código 3.29: Declaración del módulo de algoritmos

3.3.2. Librerías

Las librerías predefinidas de Haskell que vamos a usar son `Data.List` y `Data.Set` como anteriormente, y además usaremos `Data.Matrix` para los algoritmos de Floyd-Warshall y de Johnson.

Código 3.30: Librerías predefinidas usadas en los algoritmos

```
1 import qualified Data.List as L
2 import qualified Data.Set as S
3 import qualified Data.Matrix as M
```

Código 3.30: Librerías predefinidas usadas en los algoritmos

Además, deberemos usar una de las dos versiones del módulo de grafos descritas anteriormente. Podemos escribir los comandos para importarlas en líneas de código diferentes y comentar la que no queramos usar:

Código 3.31: Módulo sobre grafos

```
1 --import GrafoAlgebraico
2 import GrafoLista
```

Código 3.31: Módulo sobre grafos

3.3.3. Tipos de datos

Hemos definido un tipo de dato llamado `DT` para representar las distancias tentativas. Esto se debe a que en los algoritmos necesitamos usar un valor *infinito* que sea mayor que cualquier número.

Código 3.32: Dato distancia tentativa

```
1 data DT a = Dist a | Inf
2          deriving (Eq, Show)
3
4 instance (Ord a) => Ord (DT a) where
5   Inf <= Inf = True
6   Inf <= Dist x = False
7   Dist x <= Inf = True
```

```
  Dist x <= Dist y = x <= y
  Inf < Inf = False
  Inf < Dist x = False
  Dist x < Inf = True
  Dist x < Dist y = x < y
```

Código 3.32: Dato distancia tentativa

Además, tenemos las funciones `dist` e `inf`, que usaremos para trabajar con las distancias tentativas:

Código 3.33: Funciones `dist` e `inf`

```
dist :: a -> DT a
dist x = Dist x

inf :: DT a
inf = Inf
```

Código 3.33: Funciones `dist` e `inf`

Asimismo, definimos una función `sumaD` que nos permite sumar distancias tentativas, de forma que sumar infinito dé como resultado infinito.

Código 3.34: Suma de distancias

```
sumaD :: (Num a) => DT a -> DT a -> DT a
sumaD (Dist x) (Dist y) = Dist (x+y)
sumaD _ _ = Inf
```

Código 3.34: Suma de distancias

De forma similar, para tratar con los predecesores usaremos el tipo de dato `Maybe`, donde el valor NIL será representado en Haskell por `Nothing`.

Las siguientes funciones serán útiles para programar los algoritmos:

Código 3.35: Funciones de obtención

```
obtenjust (Just v) = v

obtendist (Dist a) = a
```

Código 3.35: Funciones de obtención

3.3.4. Funciones de inicialización

La función `distInicial` sirve como inicialización de las distancias tentativas, donde si un nodo es el nodo fuente `a` entonces se le asigna una distancia nula, y si es distinto del nodo fuente su distancia es el valor `Inf`. Esta función será usada en los algoritmos de Dijkstra, Bellman-Ford (y por tanto en el de Johnson) y en el algoritmo A*.

Código 3.36: Función `distInicial`

```
distInicial :: (Eq v, Num x) => v -> v -> DT x
distInicial a v =
  if v == a
  then Dist 0
  else Inf
```

Código 3.36: Función `distInicial`

Sin embargo, ya vimos que el algoritmo de Floyd-Warshall tiene un enfoque esencialmente distinto a los demás, por lo que necesitamos otra función de inicialización de las distancias estimadas, basándonos en la definición de 2.2:

Código 3.37: Función `distInicialFW`

```
distInicialFW :: (Num p, Eq p) => Grafo Int p -> M.Matrix (DT p)
distInicialFW g =
  let n = length (nodos g)
      as = aristas g
  in M.matrix n n (\(i,j) -> if i == j
                             then Dist 0
                             else if elem (i,j) as
                                  then Dist (peso g (i,j))
                                  else Inf )
```

Código 3.37: Función `distInicialFW`

Análogamente, definimos otra función de inicialización de los predecesores para el algoritmo de Floyd-Warshall de acuerdo con 2.4:

Código 3.38: Función `predInicialFW`

```
predInicialFW :: (Num p, Eq p) => Grafo Int p -> M.Matrix (Maybe Int)
predInicialFW g =
  let n = length (nodos g)
      as = aristas g
      w = distInicialFW g
  in M.matrix n n (\(i,j) -> if i == j || (w M.! (i,j)) == Inf
                             then Nothing
                             else Just i )
```

Código 3.38: Función `predInicialFW`

3.3.5. Algoritmo de Dijkstra

En primer lugar, consideraremos algunas funciones auxiliares que nos serán de utilidad posteriormente:

Código 3.39: Funciones auxiliares para el algoritmo de Dijkstra

```
minimo2 :: (Eq a, Ord b) => [(a,b)] -> (a,b)
minimo2 xs =
  let x = minimum $ map snd xs
```

```
4   in head $ filter (\(a,b) -> x == b) xs
5
6
7  extrae2 :: Eq a => a -> [(a,b)] -> b
8  extrae2 w xs =
9    snd (head (L.filter (\ (a,b) -> a == w) xs))
```

Código 3.39: Funciones auxiliares para el algoritmo de Dijkstra

La función `minimo2` toma una lista de pares y extrae aquel cuya segunda coordenada sea mínima. En la práctica, las primeras coordenadas corresponderán a nodos y la segunda a sus distancias estimadas. Esta función también será usada en el algoritmo A*.

Por otra parte, la función `extrae2` toma un dato `w` (que en la práctica será un nodo) y una lista de pares `xs`, y devuelve la segunda coordenada del par que tiene a `w` como primera coordenada. Nótese que se está asumiendo que tal par existe y es único. Esta función también se usará en el resto de algoritmos, excepto en el de Floyd-Warshall.

A continuación mostramos la función principal del algoritmo, `dijkstra`, que recibe como argumentos un grafo `g` y un nodo fuente `a`, y devuelve como resultado dos listas: la primera indica la distancia más corta encontrada desde el nodo fuente hasta cada nodo del grafo, mientras que la segunda indica el predecesor de cada nodo en un camino más corto desde el nodo fuente. Para ello, en primer lugar se inicializan los valores de las distancias tentativas y los predecesores, y posteriormente se usa la función auxiliar `algDijkstra`. Esta es la responsable de llevar a cabo propiamente las iteraciones del algoritmo actualizando las distancias tentativas y los predecesores hasta que la lista de nodos no visitados `cs`, correspondiente a la lista C del segundo capítulo, quede vacía.

Código 3.40: Algoritmo de Dijkstra

```
1  dijkstra :: (Eq v, Num p, Ord p) => Grafo v p -> v -> ([(v,DT p)],[(v,Maybe v)])
2  dijkstra g a =
3    let ds = [(v,distInicial a v) | v <- nodos g] -- Inicialización dist.
         tentativas
4        ps = [(v,Nothing) | v <- nodos g] -- Inicialización predecesores
5        cs = nodos g
6    in algDijkstra g ds ps cs
7
8  algDijkstra g ds ps [] = (ds,ps)
9  algDijkstra g ds ps cs =
10   let v = fst (minimo2 [(u,extrae2 u ds) | u <- cs])
11       css = cs L.\\ [v]
12       ady = adyacentes g v
13       ws = L.intersect ady css
14       dists = [ if elem w ws
15                 then if sumaD (extrae2 v ds) (Dist (peso g (v,w))) < extrae2 w ds
16                      then (w,sumaD (extrae2 v ds) (Dist (peso g (v,w))))
17                      else (w,extrae2 w ds)
18                 else (w,extrae2 w ds)
19                 | w <- nodos g ]
20       preds = [ if elem w ws
21                 then if sumaD (extrae2 v ds) (Dist (peso g (v,w))) < extrae2 w ds
22                      then (w,Just v)
```

```
                    else (w,extrae2 w ps)
               else (w,extrae2 w ps)
               | w <- nodos g ]
  in algDijkstra g dists preds css
```

Código 3.40: Algoritmo de Dijkstra

Por último, la función `imprCaminoD` representa el camino más corto entre dos nodos de un grafo encontrado por el algoritmo de Dijkstra. Esta función se corresponde con el procedimiento $IC(G, a, v)$ de 2.2.3. La función `caminoConPesoD` además muestra el peso del camino.

Código 3.41: Impresión de caminos más cortos, algoritmo de Dijkstra

```
imprCaminoD :: (Eq v, Num p, Ord p) => Grafo v p -> v -> v -> [v]
imprCaminoD g a t =
  let ps = snd (dijkstra g a)
      p = snd (head (L.filter (\ (x,y) -> t == x) ps))
  in if t == a
     then [a]
     else if p == Nothing
          then error "No existe el camino"
          else (imprCaminoD g a (obtenjust p)) ++ [t]


caminoConPesoD :: (Eq v, Num p, Ord p) => Grafo v p -> v -> v -> (p,[v])
caminoConPesoD g a t =
  let ds = fst $ dijkstra g a
      dt = obtendist $ extrae2 t ds
      cs = imprCaminoD g a t
  in (dt,cs)
```

Código 3.41: Impresión de caminos más cortos, algoritmo de Dijkstra

3.3.6. Algoritmo de Bellman-Ford

De forma análoga al algoritmo de Dijkstra, definimos una función principal llamada `bellmanford` que dados un grafo y un nodo fuente, devuelva las distancias y predecesores del resto de nodos:

Código 3.42: Algoritmo de Bellman-Ford

```
bellmanford :: (Eq v, Num p, Ord p) => Grafo v p -> v ->
               ([(v,DT p)],[(v,Maybe v)])
bellmanford g a =
  let vs = nodos g
      ds = [(v, distInicial a v) | v <- vs]
      ps = [(v,Nothing) | v <- vs]
      n = length vs
  in algBellmanFord g ds ps (n-1)
```

Código 3.42: Algoritmo de Bellman-Ford

Esta función principal inicializa los valores de las distancias tentativas y los predecesores, y usa la función auxiliar `algBellmanFord` para realizar las iteraciones del algoritmo. Esta, a su vez, llama a la función `bfaux` para ir actualizando las distancias y predecesores de cada nodo, por medio de una tercera función `actualizaAC`. Notar que `algBellmanFord` está definida recursivamente sobre el número de nodos del grafo, y en cada paso de la recursión la función `bfaux` realiza otra recursión sobre la lista de aristas del grafo. Una vez `algBellmanFord` termina sus iteraciones, se llama a la función `ciclosNegBF`, que corresponde al segundo bucle del algoritmo, y que comprueba que el grafo no contiene ciclos negativos.

Código 3.43: Funciones auxiliares, algoritmo de Bellman-Ford

```
algBellmanFord g ds ps 0 = ciclosNegBF g ds ps
algBellmanFord g ds ps k =  -- Corresponde al primer bucle del algoritmo
  let as = aristas g
      bf = bfaux g as [(v,(extrae2 v ds,extrae2 v ps)) | v <- nodos g]
      dss = [(v,d) | (v,(d,p)) <- bf]
      pss = [(v,p) | (v,(d,p)) <- bf]
  in algBellmanFord g dss pss (k-1)

ciclosNegBF g ds ps =   -- Corresponde al segundo bucle del algoritmo
  let as = aristas g
      xs = [ sumaD (extrae2 u ds) (Dist (peso g (u,v))) < extrae2 v ds | (u,v) <-
           as]
  in if or xs
     then error "Se ha detectado un ciclo negativo."
     else (ds,ps)

bfaux g [] ac = ac
bfaux g ((u,v):as) ac =
  bfaux g as (actualizaAC g (u,v) ac )

actualizaAC g (u,v) ac =
  let tu = head (L.filter (\ (a,(b,c)) -> a == u) ac)
      du = fst (snd tu)
      pu = snd (snd tu)
  in [ if ver == v
       then if sumaD du (Dist (peso g (u,v))) < d
            then (v,(sumaD du (Dist (peso g (u,v))),Just u))
            else (v,(d,p))
       else (ver,(d,p)) | (ver,(d,p)) <- ac]
```

Código 3.43: Funciones auxiliares, algoritmo de Bellman-Ford

También tenemos la función `imprCaminoBF`, que muestra un camino más corto entre dos nodos de un grafo usando el algoritmo de Bellman-Ford:

Código 3.44: Función `imprCaminoBF`

```
imprCaminoBF :: (Eq v, Num p, Ord p) => Grafo v p -> v -> v -> [v]
imprCaminoBF g a t =
  let ps = snd (bellmanford g a)
      p = snd (head (L.filter (\ (x,y) -> t == x) ps))
  in if t == a
```

```
6       then [a]
7       else if p == Nothing
8            then error "No existe el camino"
9            else (imprCaminoD g a (obtenjust p)) ++ [t]
```

Código 3.44: Función `imprCaminoBF`

Y la función `caminoConPesoBF`:

Código 3.45: Función `caminoConPesoBF`

```
1 caminoConPesoBF :: (Eq v, Num p, Ord p) => Grafo v p -> v -> v -> (p,[v])
2 caminoConPesoBF g a t =
3   let ds = fst $ bellmanford g a
4       dt = obtendist $ extrae2 t ds
5       cs = imprCaminoBF g a t
6   in (dt,cs)
```

Código 3.45: Función `caminoConPesoBF`

3.3.7. Algoritmo de búsqueda A*

La función principal `astar` inicializa los valores de las distancias tentativas y de los predecesores de cada nodo del grafo. Además, inicializa de acuerdo con 2.5.2 la lista de nodos abiertos `ss` y la lista de nodos cerrados `ts`. Observemos que esta función recibe como argumentos un grafo `g`, un nodo fuente `a`, un nodo objetivo `t` y una lista `hs` que corresponde a la gráfica de la función heurística, es decir, el conjunto de pares $(v, h(v))$ tales que $v \in V$; y devuelve como resultado directamente la distancia más corta entre `a` y `t`, así como un camino más corto entre ellos.

Código 3.46: Algoritmo A*

```
1 astar :: (Eq v, Num p, Ord p) => Grafo v p -> v -> v -> [(v,DT p)] -> (DT p,[v])
2 astar g a t hs =
3   let vs = nodos g
4       gs = [(v,distInicial a v) | v <- vs]
5       ds = [(v,sumaD (extrae2 v gs) (extrae2 v hs)) | v <- vs]
6       ps = [(v,Nothing) | v <- vs]
7       ss = [a]
8       ts = []
9   in algAstar g a t ds gs ps hs ss ts
```

Código 3.46: Algoritmo A*

Como función auxiliar, se utiliza la función `algAstar`, que realiza las iteraciones del algoritmo de acuerdo con 2.5.2.

Código 3.47: Función auxiliar `algAstar`

```
1 algAstar g a t ds gs ps hs ss ts =
2   let v = fst (minimo2 [(u,extrae2 u ds) | u <- ss])
3       sss = ss L.\\ [v]
4       tss = L.union ts [v]
```

```
5       ady = adyacentes g v
6       gss = [ if elem w ady
7               then if sumaD (extrae2 v gs) (Dist (peso g (v,w))) < extrae2 w gs
8                    then (w,sumaD (extrae2 v gs) (Dist (peso g (v,w))))
9                    else (w,extrae2 w gs)
10              else (w,extrae2 w gs) | w <- nodos g]
11      dss = [ (w,sumaD (extrae2 w gss) (extrae2 w hs)) | w <- nodos g]
12      pss = [ if elem w ady
13              then if sumaD (extrae2 v gs) (Dist (peso g (v,w))) < extrae2 w gs
14                   then (w,Just v)
15                   else (w,extrae2 w ps)
16              else (w,extrae2 w ps) | w <- nodos g]
17      tsss = tss L.\\ [w | w <- ady,
18                       sumaD (extrae2 v gs) (Dist (peso g (v,w))) < extrae2 w gs]
19      ssss = L.union sss [ w | w <- ady, (notElem w tss) ||
20            ((elem w tss && sumaD (extrae2 v gs) (Dist (peso g (v,w))) < extrae2 w
                  gs)) ]
21    in if v == t
22       then (extrae2 t gss,imprCaminoAstar a t pss)
23       else algAstar g a t dss gss pss hs ssss tsss
```

Código 3.47: Función auxiliar `algAstar`

La función `imprCaminoAstar` hace que la función `astar` devuelva un camino más corto entre dos nodos, a partir de una lista de predecesores.

Código 3.48: Función auxiliar `imprCaminoAstar`

```
1  imprCaminoAstar a t ps =
2    let jp = snd (head (L.filter (\ (x,y) -> t == x) ps))
3    in if t == a
4       then [a]
5       else if jp == Nothing
6            then error "No existe el camino"
7            else (imprCaminoAstar a (obtenjust jp) ps) ++ [t]
```

Código 3.48: Función auxiliar `imprCaminoAstar`

3.3.8. Algoritmo de Floyd-Warshall

Para el algoritmo de Floyd-Warshall, consideraremos grafos cuyos nodos son números enteros. La primera función que definimos es `distInicialFW`, que crea una matriz correspondiente a la matriz W definida en 2.2, para inicializar los valores de las distancias tentativas.

Código 3.49: Función `distInicialFW`

```
1  distInicialFW :: (Num p, Eq p) => Grafo Int p -> M.Matrix (DT p)
2  distInicialFW g =
3    let n = length (nodos g)
4        as = aristas g
5    in M.matrix n n (\(i,j) -> if i == j
6                               then Dist 0
```

```
7                             else if elem (i,j) as
8                                  then Dist (peso g (i,j))
9                                  else Inf )
```

Código 3.49: Función `distInicialFW`

De forma análoga, la función `predInicialFW` inicializa los predecesores, creando una matriz cuya entrada (i, j) se corresponde con π^0_{ij}, definido en 2.4.

Código 3.50: Función `predInicialFW`

```
1 predInicialFW :: (Num p, Eq p) => Grafo Int p -> M.Matrix (Maybe Int)
2 predInicialFW g =
3   let n = length (nodos g)
4       as = aristas g
5       w = distInicialFW g
6   in M.matrix n n (\(i,j) -> if i == j || (w M.! (i,j)) == Inf
7                              then Nothing
8                              else Just i )
```

Código 3.50: Función `predInicialFW`

Para el cálculo recursivo de las distancias y de los predecesores, se usa la función `iteracionFW`, que actualiza dichos valores de acuerdo con lo descrito en 2.3 y 2.5.

Código 3.51: Función `iteracionFW`

```
1  iteracionFW :: (Num p, Eq p, Ord p) => Grafo Int p -> Int ->
2                 (M.Matrix (DT p),M.Matrix (Maybe Int))
3  iteracionFW g 0 = (distInicialFW g, predInicialFW g)
4  iteracionFW g k =
5    let n = length (nodos g)
6        d = fst (iteracionFW g (k-1))
7        p = snd (iteracionFW g (k-1))
8    in (M.matrix n n (\(i,j) -> min (d M.! (i,j)) (sumaD (d M.! (i,k)) (d M.! (k,j)
         ))),
9        M.matrix n n (\(i,j) -> if (d M.! (i,j)) <= sumaD (d M.! (i,k)) (d M.! (k,j
             ))
10                                then p M.! (i,j)
11                                else p M.! (k,j)) )
```

Código 3.51: Función `iteracionFW`

Usando las funciones anteriores, definimos la función principal del algoritmo, llamada `floydwarshall` que, dado un grafo, devuelve dos matrices: una cuya entrada (i, j) corresponde a la distancia de un camino más corto entre el nodo i y el nodo j encontrado por el algoritmo de Floyd-Warshall, correspondiente a la matriz $D^{(n)}$ de la sección 2.6.2; y otra cuya entrada (i, j) es el predecesor del nodo j en un camino más corto desde i encontrado por el algoritmo, correspondiente a la matriz $\Pi^{(n)}$ de 2.6.3.

Código 3.52: Algoritmo de Floyd-Warshall

```
1 floydwarshall :: (Num p, Eq p, Ord p) => Grafo Int p ->
2                  (M.Matrix (DT p),M.Matrix (Maybe Int))
```

```
floydwarshall g =
  let n = length (nodos g)
  in iteracionFW g n
```

Código 3.52: Algoritmo de Floyd-Warshall

De forma análoga a los algoritmos anteriores, definimos la función `imprCaminoFW` que devuelve un camino más corto calculado por el algoritmo de Floyd-Warshall entre dos nodos de un grafo:

Código 3.53: Función `imprCaminoFW`

```
imprCaminoFW :: (Num p, Ord p) => Grafo Int p -> Int -> Int -> [Int]
imprCaminoFW g i j =
  let dist = fst (floydwarshall g)
      pred = snd (floydwarshall g)
      p = pred M.! (i,j)
  in if i == j
     then [i]
     else if p == Nothing
          then error "No existe el camino"
          else (imprCaminoFW g i (obtenjust p)) ++ [j]
```

Código 3.53: Función `imprCaminoFW`

Además, la función `caminoConPesoFW` también devuelve el peso:

Código 3.54: Función `caminoConPesoFW`

```
caminoConPesoFW :: (Num p, Ord p) => Grafo Int p -> Int -> Int -> (p,[Int])
caminoConPesoFW g i j =
  let d = fst $ floydwarshall g
      dj = obtendist $ (d M.! (i,j))
      cs = imprCaminoFW g i j
  in (dj,cs)
```

Código 3.54: Función `caminoConPesoFW`

3.3.9. Algoritmo de Johnson

Por último, a continuación comentaremos la implementación en Haskell del algoritmo de Johnson. Comenzamos definiendo la función `g'`, que crea un grafo resultado de añadir un nodo fuente al grafo original, tal y como describimos en la sección 2.7 del segundo capítulo.

Código 3.55: Grafo auxiliar del algoritmo de Johnson

```
g' :: (Eq v, Eq p, Num p) => Grafo v p -> v -> Grafo v p
g' g s =
  let vsg = nodos g
      asg = aristasConPeso g
      assg = [(u,v,p) | ((u,v),p) <- asg]
      vs = vsg ++ [s]
```

```
      as = assg ++ [(s,v,0) | v <- vsg]
  in grafo D vs as
```

Código 3.55: Grafo auxiliar del algoritmo de Johnson

La función principal es `johnson`, que primero usa la función `bellmanford` para hallar las imágenes de la función h de 2.7.1 y reponderar el grafo original para después poder aplicar el algoritmo de Dijkstra con la función `dijkstra`. Finalmente, se deshace la reponderación y se devuelven las matrices de distancias y predecesores.

Código 3.56: Algoritmo de Johnson

```
johnson :: (Num p, Ord p) => Grafo Int p ->
          (M.Matrix (DT p), M.Matrix (Maybe Int))
johnson g =
  let vs = nodos g
      s = 0
      n = length vs
      as = aristas g
      gg = g' g s
      ds = fst (bellmanford gg s)
      hs = [(v,obtendist (extrae2 v ds)) | v <- vs]
      pgorros = [(u,v,(peso gg (u,v)) + (extrae2 u hs) - (extrae2 v hs))
                | (u,v) <- as]
      gp = grafo D vs pgorros
      dijkstras = [(u,dijkstra gp u) | u <- vs]
      dists = [(u,fst (extrae2 u dijkstras)) | u <- vs]
      preds = [(u,snd (extrae2 u dijkstras)) | u <- vs]
      d = M.matrix n n (\(u,v) -> sumaD (sumaD (extrae2 v (extrae2 u dists))
                      (Dist (extrae2 v hs))) (Dist (-extrae2 u hs)))
      p = M.matrix n n (\(u,v) -> extrae2 v (extrae2 u preds))
  in (d,p)
```

Código 3.56: Algoritmo de Johnson

Por último, tenemos las funciones `imprCaminoJ` y `caminoConPesoJ`, análogas a las correspondientes del resto de algoritmos.

Código 3.57: Impresión de los caminos más cortos, algoritmo de Johnson

```
imprCaminoJ :: (Num p, Ord p) => Grafo Int p -> Int -> Int -> [Int]
imprCaminoJ g i j =
  let dist = fst (johnson g)
      pred = snd (johnson g)
      p = pred M.! (i,j)
  in if i == j
     then [i]
     else if p == Nothing
          then error "No existe el camino"
          else (imprCaminoJ g i (obtenjust p)) ++ [j]

caminoConPesoJ :: (Num p, Ord p) => Grafo Int p -> Int -> Int -> (p,[Int])
caminoConPesoJ g i j =
  let d = fst $ johnson g
```

```
15      dj = obtendist $ (d M.! (i,j))
16      cs = imprCaminoJ g i j
17  in (dj,cs)
```

Código 3.57: Impresión de los caminos más cortos, algoritmo de Johnson

CAPÍTULO 4

Aplicaciones

Una vez creada la librería, en este capítulo daremos algunos ejemplos de aplicación, donde pondremos en práctica los distintos algoritmos para encontrar caminos más cortos en grafos concretos. El objetivo no es tanto resolver problemas de camino más corto complejos, sino más bien probar los algoritmos y verificar que funcionan como se espera.

Para ello, crearemos un nuevo fichero Haskell donde definiremos los grafos y funciones necesarias. En nuestro caso, lo hemos llamado `Ejemplos.hs`.

4.1– Cargar nuestra librería

Lo primero que haremos será cargar los módulos creados en el capítulo anterior. Deberemos escoger entre `GrafoAlgebraico` y `GrafoLista`:

Código 4.1: Librerías usadas en los ejemplos

```
1 --import GrafoAlgebraico
2 import GrafoLista
3 import AlgoritmosGrafos
```

Código 4.1: Librerías usadas en los ejemplos

4.2– Algunos ejemplos de grafos

A continuación definimos algunos grafos de ejemplo donde probaremos los algoritmos estudiados con anterioridad.

Código 4.2: Ejemplos de grafos

```
1 g1 :: Grafo Int Int
2 g1 = grafo D [1..7] [(1,3,1),(1,4,2),(2,1,3),(2,7,4),(3,2,4),(3,4,1),(3,5,5),
3                      (4,2,3),(4,5,4),(4,7,1),(5,6,2),(6,4,5),(7,6,3)]
4
```

```
g2 :: Grafo Int Int
g2 = grafo D [1..7] [(1,3,1),(1,4,-2),(2,1,3),(2,7,4),(3,2,-2),(3,4,1),(3,5,5),
                     (4,2,3),(4,5,-3),(4,7,1),(5,6,2),(6,4,5),(7,6,-2)]

g3 :: Grafo Int Int
g3 = grafo D [1..7] [(1,3,1),(1,4,-2),(2,1,3),(2,7,4),(3,2,-5),(3,4,1),(3,5,5),
                     (4,2,3),(4,5,-3),(4,7,1),(5,6,2),(6,4,5),(7,6,-2)]
```

Código 4.2: Ejemplos de grafos

Comencemos con el primer grafo, `g1`. Podemos representarlo de la siguiente forma:

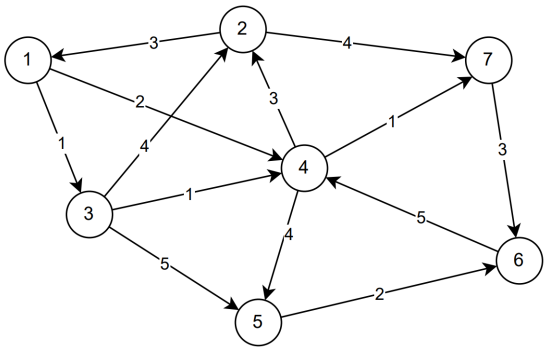


Figura 4.1: Grafo `g1`

Observemos que este grafo tiene todos sus pesos positivos, luego podemos aplicar el algoritmo de Dijkstra para calcular caminos más cortos. Por ejemplo, para calcular un camino más corto usando dicho algoritmo entre los nodos 1 y 6 y su peso, usamos el siguiente comando:

Código 4.3: Salida: Aplicación de Dijkstra a `g1`

```
   > caminoConPesoD g1 1 6
(6,[1,4,7,6])
```

Código 4.3: Salida: Aplicación de Dijkstra a `g1`

Vemos que el camino más corto encontrado es $(1, 4, 7, 6)$, con peso 6. Podemos verificar fácilmente en la figura 4.1 que este camino es efectivamente un camino más corto.

Usando el resto de algoritmos (excepto el algoritmo A*, para el que necesitamos una heurística), obtenemos el mismo resultado:

Código 4.4: Salida: Aplicación del resto de algoritmos a `g1`

```
   > caminoConPesoBF g1 1 6
(6,[1,4,7,6])
   > caminoConPesoFW g1 1 6
(6,[1,4,7,6])
   > caminoConPesoJ g1 1 6
(6,[1,4,7,6])
```

Código 4.4: Salida: Aplicación del resto de algoritmos a `g1`

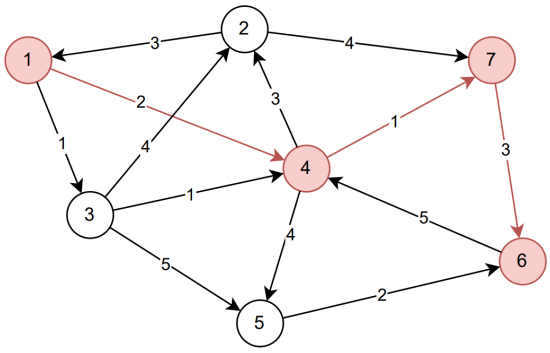


Figura 4.2: Camino más corto en el grafo `g1`

Pasando al segundo ejemplo, el grafo `g2`, observamos que este grafo tiene los mismos nodos que el anterior y también las mismas aristas, solo que tiene pesos diferentes. Al tener pesos negativos, esta vez no podemos usar el algoritmo de Dijkstra, pero sí el de Bellman-Ford, el de Floyd-Warshall y el de Johnson:

Código 4.5: Salida: Aplicación de los algoritmos a `g2`

```
   > caminoConPesoBF g2 1 6
(-3,[1,4,5,6])
   > caminoConPesoFW g2 1 6
(-3,[1,4,5,6])
   > caminoConPesoJ g2 1 6
(-3,[1,4,5,6])
```

Código 4.5: Salida: Aplicación de los algoritmos a `g2`

Al cambiar los pesos de las aristas, también los caminos más cortos han cambiado. Por ello, ahora obtenemos el camino $(1, 4, 5, 6)$, con peso -3.

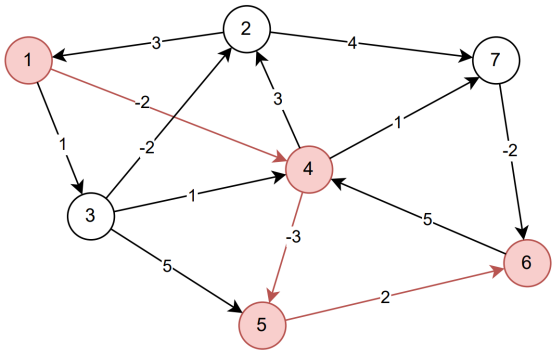


Figura 4.3: Camino más corto en el grafo `g2`

Como todos los nodos de `g2` son alcanzables desde el nodo 1, sabemos que el grafo no tiene ciclos negativos, ya que en otro caso los algoritmos lo habrían detectado.

En último lugar, consideremos el grafo `g3`, que también tiene pesos negativos. ¿Tendrá ciclos negativos? La respuesta es que sí: basta ejecutar el algoritmo de Bellman-Ford.

Código 4.6: Salida: Ciclo negativo en g3

```
   > caminoConPesoBF g3 1 6
(*** Exception: Se ha detectado un ciclo negativo.
CallStack (from HasCallStack):
  error, called at .\AlgoritmosGrafos.hs:237:11 in main:AlgoritmosGrafos
```

Código 4.6: Salida: Ciclo negativo en g3

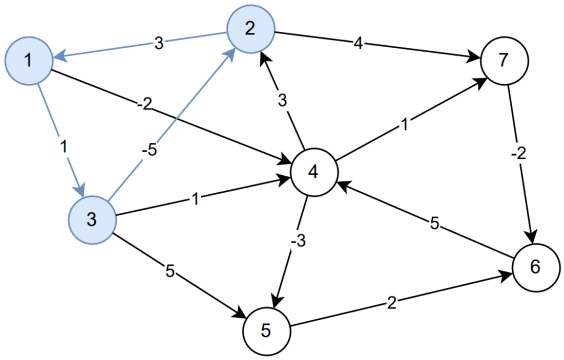


Figura 4.4: Ciclo negativo en el grafo g3

Al tratarse de un grafo relativamente simple, no tardamos mucho en encontrar un ciclo negativo: $(1, 3, 2, 1)$. Podemos comprobarlo rápidamente con la función `pesoCamino`.

Código 4.7: Salida: Peso del ciclo negativo de g3

```
   > pesoCamino g3 [1,3,2,1]
-1
```

Código 4.7: Salida: Peso del ciclo negativo de g3

4.3– Aplicación de A*: Red de carreteras de Andalucía

En esta sección, usaremos el algoritmo de búsqueda A* para encontrar rutas más cortas por carretera entre las capitales de provincia andaluzas. Para ello, deberemos definir un grafo que represente la red de carreteras (o una versión simplificada de ella), así como disponer de información extra que podamos usar en la función heurística. En este caso, esta información será la distancia en línea recta entre las diferentes ciudades.

El grafo que definiremos tendrá por nodos las capitales de provincia, y para cada par de ellos, habrá una arista entre ellos si las provincias que representan son vecinas y existe una carretera que conecte directamente sus capitales. Dichas aristas deberán ser no dirigidas, ya que es posible realizar viajes en carretera en ambos sentidos, y además estarán ponderadas con la longitud en kilómetros de las carreteras que representan.

En Haskell, definimos este grafo, al que vamos a llamar `andalucia`, de la siguiente forma:

Código 4.8: Definición del grafo `andalucia`

```
andalucia :: Grafo [Char] Double
andalucia = grafo ND
          ["Huelva","Cadiz","Malaga","Granada","Almeria","Jaen","Cordoba","
             Sevilla"]
          [("Huelva","Sevilla",92.7),("Sevilla","Cadiz",120),("Sevilla","Malaga"
             ,200),
           ("Sevilla","Cordoba",141),("Cordoba","Malaga",163),("Cordoba","Jaen"
              ,108),
           ("Cordoba","Granada",165),("Malaga","Cadiz",233),("Malaga","Granada"
              ,129),
           ("Granada","Jaen",91),("Granada","Almeria",167)]
```

Código 4.8: Definición del grafo `andalucia`

Figura 4.5: Red de carreteras de Andalucía, imagen de Faelomx/ CC BY 4.0 [20]

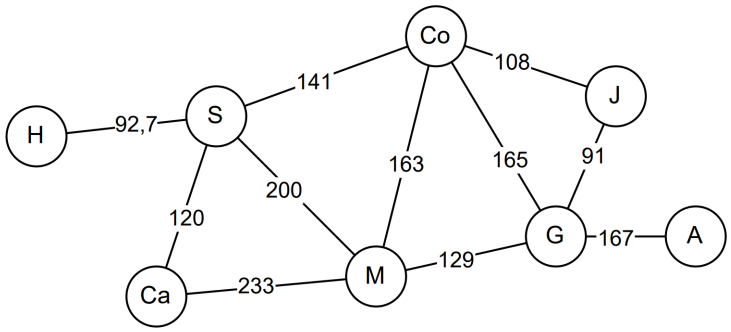


Figura 4.6: Grafo de representación de la red de carreteras

Mediante la función `heuristica`, almacenamos la distancia en línea recta entre cada par de ciudades. En este caso, al no ser demasiadas ciudades, hemos proporcionado los

datos directamente en lugar de, por ejemplo, definir una función que calcule las distancias entre ellas a partir de sus coordenadas.

Código 4.9: Función `heuristica`

```
heuristica :: [Char] -> [Char] -> DT Double
heuristica "Huelva" "Sevilla" = dist 87
heuristica "Huelva" "Cadiz" = dist 100
heuristica "Huelva" "Cordoba" = dist 203
heuristica "Huelva" "Malaga" = dist 232
heuristica "Huelva" "Granada" = dist 297
heuristica "Huelva" "Jaen" = dist 285
heuristica "Huelva" "Almeria" = dist 401
heuristica "Sevilla" "Cadiz" = dist 100
heuristica "Sevilla" "Cordoba" = dist 120
heuristica "Sevilla" "Malaga" = dist 157
heuristica "Sevilla" "Granada" = dist 212
heuristica "Sevilla" "Jaen" = dist 199
heuristica "Sevilla" "Almeria" = dist 319
heuristica "Cadiz" "Cordoba" = dist 201
heuristica "Cadiz" "Malaga" = dist 167
heuristica "Cadiz" "Granada" = dist 249
heuristica "Cadiz" "Jaen" = dist 261
heuristica "Cadiz" "Almeria" = dist 341
heuristica "Cordoba" "Malaga" = dist 130
heuristica "Cordoba" "Granada" = dist 130
heuristica "Cordoba" "Jaen" = dist 88
heuristica "Cordoba" "Almeria" = dist 235
heuristica "Malaga" "Granada" = dist 89
heuristica "Malaga" "Jaen" = dist 130
heuristica "Malaga" "Almeria" = dist 175
heuristica "Granada" "Jaen" = dist 70
heuristica "Granada" "Almeria" = dist 107
heuristica "Jaen" "Almeria" = dist 157
heuristica x y =
  if x == y
  then dist 0
  else heuristica y x
```

Código 4.9: Función `heuristica`

Las siguientes funciones permiten usar el algoritmo A* para encontrar las rutas más cortas. Para obtener el resultado final, deberemos usar la función `astarAndalucia`.

Código 4.10: Aplicación de A*

```
vs = nodos andalucia

hsAnd x = [ (v,heuristica x v) | v <- vs ]

astarAndalucia a t = astar andalucia a t (hsAnd t)
```

Código 4.10: Aplicación de A*

Por ejemplo, podemos usar `astarAndalucia` para obtener una ruta óptima entre Huelva y Almería, o entre Jaén y Cádiz:

Código 4.11: Salida: Rutas más cortas en la red de carreteras

```
1    > astarAndalucia "Huelva" "Almeria"
2 (Dist 565.7,["Huelva","Sevilla","Cordoba","Granada","Almeria"])
3    > astarAndalucia "Jaen" "Cadiz"
4 (Dist 369.0,["Jaen","Cordoba","Sevilla","Cadiz"])
```

Código 4.11: Salida: Rutas más cortas en la red de carreteras

Además, obtenemos que la primera ruta tiene 565,7 kilómetros de distancia, mientras que la segunda mide 369 kilómetros. Si quisiéramos encontrar la ruta más corta entre cualquier otro par de ciudades, simplemente debemos ejecutar `astarAndalucia` seguido de dichas ciudades.

4.4– Cómo escapar de un laberinto

Los algoritmos que hemos estudiado en este trabajo también pueden ser usados, por ejemplo, para encontrar la salida de un laberinto. Es bien conocido que existen otros métodos, como el de la mano derecha, que consiste en ir recorriendo el laberinto sin separar la mano derecha de la primera pared que encontremos y, tarde o temprano, terminaremos encontrando la salida. Sin embargo, métodos como este pueden hacernos recorrer caminos muy largos, pasando varias veces por el mismo lugar. Los algoritmos que resuelven el problema del camino más corto, como su propio nombre indica, no sólo encuentran la salida, sino además el camino más corto hasta ella.

Por ejemplo, tenemos el laberinto de la figura 4.7 y queremos encontrar la solución, es decir, un camino que vaya desde la abertura inicial hasta la final.

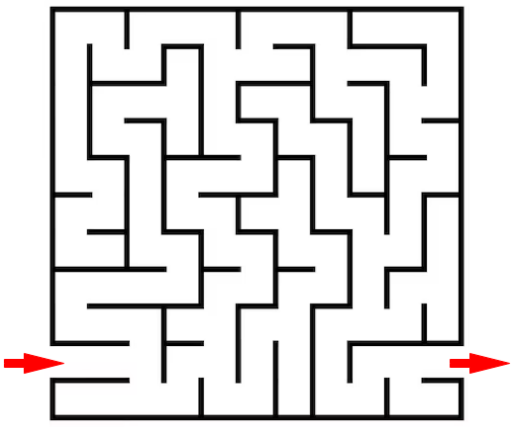


Figura 4.7: Ejemplo de laberinto

Para poder atacar el problema usando nuestros algoritmos, lo primero que tenemos que hacer es definir un grafo al que aplicarlos. Para ello, podemos colocar una cuadrícula sobre el laberinto, de 11×11 casillas, quedando cada una de ellas determinada por unas

coordenadas (x, y), con $1 \leq x, y \leq 11$. Así, los nodos del grafo serán cada una de estas 121 casillas. Entre cada par de ellos habrá una arista con peso unitario si las casillas son contiguas y no existe una pared que las separe. De esta forma, un camino en el grafo se corresponderá con un camino válido en el laberinto.

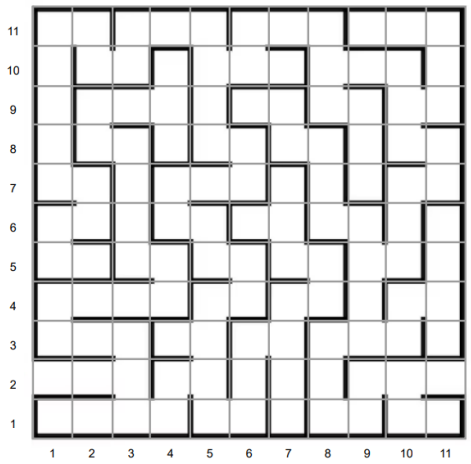


Figura 4.8: Numeración de las casillas

En Haskell, definimos la lista de nodos del grafo de la siguiente manera:

Código 4.12: Nodos del grafo, ejemplo de laberinto

```
1 nodosL = [(i,j) | i <- [1..11], j <- [1..11]]
```

Código 4.12: Nodos del grafo, ejemplo de laberinto

Con un poco de paciencia, definimos la lista de aristas con ayuda de la figura 4.8:

Código 4.13: Aristas del grafo, ejemplo de laberinto

```
1  aristasL = [((1,2),(2,2)),((2,2),(3,2)),((3,2),(3,3)),((3,3),(2,3)),
2              ((2,3),(1,3)),((1,3),(1,4)),((1,4),(2,4)),((2,4),(3,4)),
3              ((3,4),(4,4)),((4,4),(4,5)),((4,5),(3,5)),((3,5),(3,6)),
4              ((3,6),(3,7)),((3,7),(3,8)),((3,8),(2,8)),((2,8),(2,9)),
5              ((2,9),(3,9)),((3,9),(4,9)),((4,9),(4,10)),((4,9),(4,8)),
6              ((3,2),(3,1)),((3,1),(2,1)),((2,1),(1,1)),((3,1),(4,1)),
7              ((4,1),(4,2)),((4,2),(5,2)),((5,2),(5,1)),((5,1),(6,1)),
8              ((6,1),(6,2)),((6,2),(6,3)),((6,3),(7,3)),((7,3),(7,4)),
9              ((7,4),(8,4)),((8,4),(8,5)),((8,5),(7,5)),((7,5),(7,6)),
10             ((7,6),(6,6)),((7,6),(7,7)),((7,3),(7,2)),((7,2),(7,1)),
11             ((5,2),(5,3)),((5,3),(4,3)),((5,3),(5,4)),((5,4),(6,4)),
12             ((6,4),(6,5)),((6,5),(5,5)),((5,5),(5,6)),((5,6),(4,6)),
13             ((4,6),(4,7)),((4,7),(5,7)),((5,7),(6,7)),((6,7),(6,8)),
14             ((6,8),(5,8)),((5,8),(5,9)),((5,9),(5,10)),((5,10),(5,11)),
15             ((5,11),(4,11)),((4,11),(3,11)),((3,11),(3,10)),((3,10),(2,10)),
16             ((2,10),(2,11)),((2,11),(1,11)),((1,11),(1,10)),((1,10),(1,9)),
17             ((1,9),(1,8)),((1,8),(1,7)),((1,7),(2,7)),((2,7),(2,6)),
18             ((2,6),(1,6)),((1,6),(1,5)),((1,5),(2,5)),((5,10),(6,10)),
19             ((6,10),(7,10)),((6,10),(6,11)),((6,11),(7,11)),((7,11),(8,11)),
```

```
            ((8,11),(8,10)),((8,10),(8,9)),((8,9),(9,9)),((9,9),(9,8)),
            ((9,8),(9,7)),((8,10),(9,10)),((9,10),(10,10)),((10,10),(10,9)),
            ((10,9),(11,9)),((11,9),(11,10)),((11,10),(11,11)),
            ((11,11),(10,11)),((10,11),(9,11)),((10,9),(10,8)),((10,8),(11,8)),
            ((11,8),(11,7)),((11,7),(10,7)),((10,7),(10,6)),((10,6),(10,5)),
            ((10,5),(9,5)),((9,5),(9,6)),((9,6),(8,6)),((8,6),(8,7)),
            ((8,7),(8,8)),((8,8),(7,8)),((7,8),(7,9)),((7,9),(6,9)),
            ((9,5),(9,4)),((9,4),(9,3)),((9,3),(10,3)),((10,3),(10,4)),
            ((10,4),(11,4)),((11,4),(11,3)),((11,4),(11,5)),((11,5),(11,6)),
            ((9,3),(8,3)),((8,3),(8,2)),((8,2),(8,1)),((8,1),(9,1)),
            ((9,1),(9,2)),((9,2),(10,2)),((10,2),(10,1)),((10,1),(11,1)),
            ((10,2),(11,2))]
```

Código 4.13: Aristas del grafo, ejemplo de laberinto

La función `apu` asigna un peso unitario a cada arista:

Código 4.14: Asignación de pesos unitarios

```
apu :: [(a,a)] -> [(a,a,Double)]
apu [] = []
apu ((x,y):xs) = (x,y,1):(apu xs)
```

Código 4.14: Asignación de pesos unitarios

Por tanto, el grafo correspondiente al laberinto será el siguiente:

Código 4.15: Grafo del laberinto

```
grafoL :: Grafo (Integer,Integer) Double
grafoL = grafo ND nodosL (apu aristasL)
```

Código 4.15: Grafo del laberinto

Según la figura 4.8, los nodos de inicio y de final son $(1, 2)$ y $(11, 2)$ respectivamente:

Código 4.16: Nodos inicial y final

```
inicio = (1,2)

final = (11,2)
```

Código 4.16: Nodos inicial y final

Ahora ya podemos aplicar alguno de los algoritmos para encontrar un camino más corto entre la entrada y la salida del laberinto. El algoritmo de Dijkstra parece una buena opción para resolver el problema, ya que es más rápido que el de Bellman-Ford, y los algoritmos de Floyd-Warshall y Jonhson son más útiles para problemas de tipo APSP.

Código 4.17: Salida: Aplicación de Dijkstra para salir del laberinto

```
    > :set +s
    > imprCaminoD grafoL inicio final
[(1,2),(2,2),(3,2),(3,1),(4,1),(4,2),(5,2),(5,3),(5,4),(6,4),(6,5),(5,5),(5,6)
    ,(4,6),(4,7),(5,7),(6,7),(6,8),(5,8),(5,9),(5,10),(6,10),(6,11),(7,11),(8,11)
    ,(8,10),(9,10),(10,10),(10,9),(10,8),(11,8),(11,7),(10,7),(10,6),(10,5),(9,5)
    ,(9,4),(9,3),(8,3),(8,2),(8,1),(9,1),(9,2),(10,2),(11,2)]
```

```
4 (36.09 secs, 3,572,211,360 bytes)
```

Código 4.17: Salida: Aplicación de Dijkstra para salir del laberinto

El algoritmo de Dijkstra encuentra un camino, representado por la figura 4.9, pero hemos tenido que esperar 36 segundos para obtener la solución. Veamos si podemos usar el algoritmo A* para obtener una solución en un tiempo menor.

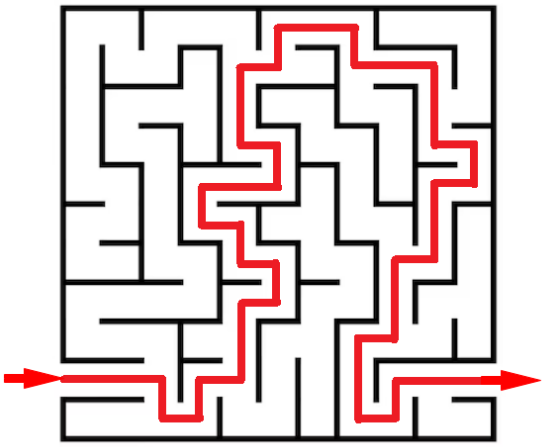


Figura 4.9: Laberinto resuelto

La función heurística que usaremos será la distancia euclídea hasta la casilla final. Para ello definimos las funciones `hL` y `hLs`:

Código 4.18: Heurística del laberinto

```
1 hL :: (Integer,Integer) -> (Integer,Integer) -> DT Double
2 hL (a,b) (x,y) =
3   dist $ sqrt (fromInteger ((x-a)^2 + (y-b)^2))
4
5 hLs t = [(v,hL v t) | v <- nodosL]
```

Código 4.18: Heurística del laberinto

La función `astarL` nos permite usar el algoritmo A* para encontrar la solución:

Código 4.19: Aplicación de A* para encontrar la salida del laberinto

```
1 astarL a t = astar grafoL a t (hLs t)
```

Código 4.19: Aplicación de A* para encontrar la salida del laberinto

Finalmente, ejecutamos la siguiente orden:

Código 4.20: Salida: Aplicación de A* para salir del laberinto

```
1    > astarL inicio final
2 (Dist 44.0,[(1,2),(2,2),(3,2),(3,1),(4,1),(4,2),(5,2),(5,3),(5,4),(6,4),(6,5)
    ,(5,5),(5,6),(4,6),(4,7),(5,7),(6,7),(6,8),(5,8),(5,9),(5,10),(6,10),(6,11)
    ,(7,11),(8,11),(8,10),(9,10),(10,10),(10,9),(10,8),(11,8),(11,7),(10,7)
    ,(10,6),(10,5),(9,5),(9,4),(9,3),(8,3),(8,2),(8,1),(9,1),(9,2),(10,2),(11,2)
    ])
3 (0.68 secs, 82,887,880 bytes)
```

Código 4.20: Salida: Aplicación de A* para salir del laberinto

Como vemos, obtenemos la misma solución que antes, pero en menos de un segundo, lo que demuestra la rapidez del algoritmo A*.

Observemos que no estamos limitados a únicamente poder encontrar un camino desde la entrada del laberinto hasta la salida, sino que podemos encontrar caminos más cortos entre cualquier par de casillas del laberinto. Luego, no importa que "nos hayamos perdido" en el laberinto y no estemos en la casilla de entrada, que siempre podremos encontrar un camino hasta la salida.

CAPÍTULO 5

Conclusiones

El estudio teórico sobre el problema del camino más corto desarrollado en este trabajo nos ha permitido conocer algoritmos que emplean diversas estrategias para atacarlo, así como comprender con la suficiente profundidad su funcionamiento. Por otra parte, la librería implementada cumple con el objetivo principal del trabajo: ser capaces de construir grafos y trabajar con ellos, además de aplicar los algoritmos estudiados para resolver el problema del camino más corto en ejemplos sencillos. En este proceso, hemos debido adaptar la programación de los algoritmos a un lenguaje de programación funcional como lo es Haskell, a pesar de que el enfoque empleado más habitualmente sea el imperativo.

Aunque realmente la optimización de la librería no era el objetivo principal del trabajo, como vimos en el capítulo sobre el diseño de esta, finalmente desarrollamos una segunda versión del módulo de construcción y descripción de grafos para mejorar la eficiencia de la primera. Aún así, consideramos interesante la idea de construcción de los grafos basada en el concepto de unión de esta primera versión, por lo que podría quedar para una posible ampliación futura su optimización. A este trabajo futuro podría añadirse la optimización del módulo de algoritmos para poder ser aplicados en problemas más complejos, ya que como observamos en el capítulo de aplicaciones, pueden quedarse cortos para grafos grandes. Otras vías de ampliación que podrían explorarse podrían ser el estudio de nuevos algoritmos para resolver el problema, o incluso procedimientos que construyan grafos a partir de bases de datos para no tener que lidiar con la incomodidad de proporcionar manualmente todos los nodos y aristas cuando los grafos sean muy grandes.

En conclusión, a pesar de que siempre existirá margen de mejora, consideramos que los objetivos que se habían marcado inicialmente han sido cumplidos con la realización de este trabajo, además de pensar que el aprendizaje y la experiencia adquiridas serán de gran utilidad en futuros proyectos.

Bibliografía

[1] A. Schrijver, 2012. *On the history of the shortest path problem.* Documenta Mathematica. Extra Volume ISMP, 155–167.

[2] H.D. Landahl y R. Runge, 1946. *Outline of a matrix algebra for neural nets.* Bulletin of Mathematical Biophysics, 8: 75-81.

[3] R.D. Luce y A.D. Perry, 1949. *A method of matrix analysis of group structure.* Psychometrika, 14: 95–116.

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

[5] L.R. Ford, 1956. *Network Flow Theory.* Santa Monica, CA: RAND Corporation.

[6] E.W. Dijkstra, 1959. *A note on two problems in connexion with graphs.* Numerische Mathematik, 1: 269-271.

[7] S. Warshall, 1962. *A theorem on boolean matrices.* Journal of the ACM, 9(1): 11–12.

[8] R.W. Floyd, 1962. *Algorithm 97: Shortest Path.* Communications of the ACM, 5(6): 345.

[9] D.B. Johnson, 1977. *Efficient Algorithms for Shortest Paths in Sparse Networks.* Journal of the ACM, 24(1): 1–13.

[10] P.E. Hart, N.J. Nilsson y B. Raphael, 1968. *A formal basis for the heuristic determination of minimum cost paths.* IEEE Transactions of Systems Science and Cybernetics, 4: 100-107.

[11] T.H. Cormen, C.E. Leiserson, R.L. Rivest y C. Stein, 2009. *Introduction to Algorithms: Third Edition.* MIT Press.

[12] J. Bang-Jensen y G. Gutin, 2000. *Digraphs: Theory, Algorithms and Applications, First Edition.* Springer-Verlag.

[13] R.C. Prim, 1957. *Shortest connection networks and some generalizations.* Bell System Technical Journal, 36(6): 1389–1401.

[14] V. Jarník, 1929. *O jistém problému minimálním (Sobre un problema minimal).* Práce Moravské Přírodovědecké Společnosti, 6, 1930, pp. 57–63 (en checo).

[15] J.A. Alonso Jiménez y F.J. Martín Mateos. Librerías `GrafoConMatrizDeAdyacencia.hs` y `GrafoConVectorDeAdyacencia.hs`. Asignatura de Informática 2020/2021, Grado en Matemáticas, Universidad de Sevilla. Consultado en `https://www.cs.us.es/cursos/i1m3/?contenido=ejercicios.php`.

[16] Haskell, *Algebra.Graph*. Consultado en `https://hackage.haskell.org/package/algebraic-graphs-0.7/docs/Algebra-Graph.html`.

[17] Wikipedia, *Shortest Path Problem*. Consultado en `https://en.wikipedia.org/wiki/Shortest_path_problem`.

[18] Wikipedia, *Dijkstra's algorithm*. Consultado en `https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm`.

[19] Wikipedia, *Bellman-Ford algorithm*. Consultado en `https://en.wikipedia.org/wiki/Bellman%E2%80%93Ford_algorithm`.

[20] Wikipedia, *Anexo: Red de Carreteras de Andalucía*. Consultado en `https://es.wikipedia.org/wiki/Anexo:Red_de_Carreteras_de_Andaluc%C3%ADa`.

Índice de figuras

Índice de código