FACULTAD DE INFORMÁTICA

Trabajo de investigación de MARP

Caminos cortos y largos

Alberto Maurel Serrano

Índice

1. Introducción

Este un trabajo de investigación para la asignatura de MARP. En él se tratan los algoritmos eficientes para encontrar caminos largos y cortos en un grafo. El objetivo del trabajo es que con la información contenido el lector sea capaz de entender e implementar el algoritmo, además de que tenga una intuición de por qué el algoritmo funciona y una idea de lo que está ocurriendo por debajo

Por otro lado, son numerosos los problemas que se pueden encontrar en jueces online que se resuelven con estos algoritmos. Por ello se han incluido problemas resueltos, para que el lector vea el nexo entre la formulación teórica del algoritmo y su utilización. Los problemas resueltos se pueden encontrar al final del trabajo, implementados en C++, ordenados según han ido apareciendo en el trabajo. Si el lector desea probar los algoritmos en más ejercicios, en uHunt tiene una amplia selección de ellos disponibles ordenados por temas.

Otra cosa importante es que en los códigos, para evitar que se volviesen muy extensos, se han empleado las abreviaturas típicas de programación competitiva introducidas en [8], que también se ha utilizado como bibliografía en el trabajo. Algunas de estas abreviaturas son:

```
using ii = pair<int, int>;
using vi = vector<int>;
using vvi = vector<vi>;

using ll = long long int;
using vb = vector<bool>;
using vvb = vector<vb>;
```

En cualquier caso todas ellas son muy evidentes y se espera que el lector se familiarice rápido con ellas. Espero que el texto resulte interesante al lector y que aprenda alguna cosa que no conociese.

2. Caminos cortos

En esta sección se tratan los distintos algoritmos para hallar los caminos más cortos en una matriz. Se resuelven dos problemas: SSSP (*Single Source Shortest Paths*), es decir, los caminos más cortos desde un nodo a todos los demás, y ASSP (*All Source Shortest Paths*), el camino más corto entre dos nodos cualesquiera del grafo. Hay múltiples algoritmos que resuelven estos problemas, y en casos especiales del problema se pueden utilizar algoritmos que proporcionan una mayor eficiencia. Se presentan los algoritmos más importante junto a las hipótesis bajo las que se deben utilizar:

2.1. Búsqueda en anchura

2.1.1. Introducción

El algoritmo de búsqueda en anchura es un algoritmo general para recorrer un grafo, pero veremos que nos permite hallar de forma eficiente el camino más corto en situaciones en las que hay estructura de grafo y ciertas propiedades. Concretamente se emplea en el caso de que tengamos grafos no valorados (las aristas no estén ponderadas), dirigidos o no.

Una de sus aplicaciones más evidentes es hallar el camino más corto para salir de una matriz. Este uso fue descrito por Edward F. Moore en *The shortest path through a maze* [2] en el año 1959. Este algoritmo tiene una complejidad en $O(V + E)$.

2.1.2. Algoritmo

La idea detrás de este algoritmo es que, como las aristas no están ponderadas, podemos procesar por niveles los nodos según su distancia al origen: primero el nodo origen, a distancia 0; luego los nodos adyacences al origen, que estarán a distancia 1; y así sucesivamente. Si procesamos por orden de distancia, en todo momento tendremos los nodos de este nivel que aún no hemos expandido, a distancia n, y los nodos del siguiente nivel, que son los nodos adyacentes a nodos de este nivel, luego estarán a distancia $n + 1$ del origen (al menos a través de este camino). Siguiendo este razonamiento, cuando lleguemos al nodo destino, tenemos garantizado que no habrá ningún camino mejor para llegar a él y pararemos la búsqueda. Si pudiésemos llegar por un camino más corto, como los nodos se procesan por distancia creciente al origen ya habríamos encontrado un camino al destino y habríamos parado la búsqueda.

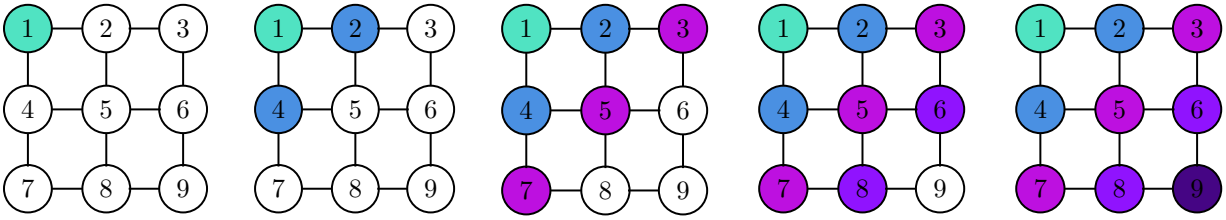


Figura 1: Ejemplo de orden de procesamiento de los nodos en una búsqueda en anchura. En azul claro el nodo origen y en cada imagen el procesamiento de un nuevo nivel.

Sin embargo, la implementación directa del párrafo anterior no es una solución eficiente. La razón es que para un mismo nodo, podríamos visitarlo múltiples veces en un mismo nivel, o incluso visitarlo en ocasiones diferentes en niveles distintos (y evidentemente si hemos logrado llegar a un nodo en una cantidad inferior de pasos, el camino actual hasta ese nodo no puede ser parte de ningún camino óptimo que vaya del nodo origen al nodo destino pasando por el nodo actual). Por tanto queremos evitar revisitar el mismo nodo muchas veces sabiendo que la primera vez siempre será la óptima (en realidad cualquier otro camino que llegase al nodo en el mismo número de pasos será igual de óptimo, pero tomamos la primera vez por simplicidad).

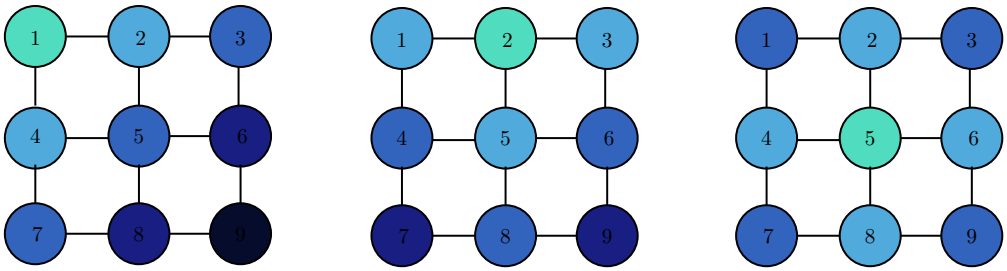


Figura 2: Ejemplo de como funciona el algoritmo de búsqueda en anchura sobre un grafo variando el nodo origen (nodo en azul claro). En azul más oscuro a medida que nos alejamos del nodo de origen.

Para evitar esto simplemente necesitamos marcar qué nodos hemos visitado ya y cuáles no. Si un nodo ya lo hemos visitado sabemos que hemos llegado a él ya en un número óptimo de pasos y que no necesitamos comprobar si el camino que estamos construyendo actualmente es óptimo o no. En caso de que no esté marcado sabemos que el camino actual es prometedor y lo añadimos a la cola para seguirlo procesando.

Uniendo todo lo anterior el pseudocódigo para el algoritmo podría ser el siguiente:

Algoritmo 1 Búsqueda en anchura

```
1:  estadoAct.nivel = 0
2:  estadoAct.nodo = nodoInicial
3:  cola.push(estadoAct)
4:  mientras !cola.empty() hacer
5:     estadoActual = cola.front()
6:     cola.pop()
7:     si estadoAct.nodo = nodoDestino entonces
8:        return estadoAct.nivel
9:     fin si
10:    para cada nodo adyacente hacer
11:       si el nodo adyacente no está marcado entonces
12:          estadoSig.nivel = estadoAct.nivel + 1
13:          estadoSig.nodo = nodo adyacente
14:       fin si
15:    fin para
16: fin mientras
17: return -1
```

Por último, en caso de que no exista un camino entre ambos nodos devolvemos un valor centinela, en este caso -1 (para indicar que no existe camino utilizamos un valor que evidentemente no se puede obtener).

2.1.3. Código

Una implementación del pseudocódigo anterior para una matriz podría ser:

```
struct tEstado {
   int x, y;
   int longitud;
};

vi fil = { -1,0,1,0 };
```

```
vi col = { 0,1,0,-1 };

int BFS(int xini, int yini, int xfin, int yfin, vvb & marcaje) {
   int C = marcaje[0].size(), F = marcaje.size();

   queue<tEstado> cola;
   if (marcaje[yini][xini]) {
      cola.push({ xini, yini, 0 });
      marcaje[yini][xini] = false;
   }

   while (!cola.empty()) {
      tEstado estadoAct = cola.front(); cola.pop();
      if (estadoAct.x == xfin && estadoAct.y == yfin) return estadoAct.longitud;
      else {
         for (int k = 0; k < fil.size(); ++k) {
            if (estadoAct.x + col[k] >= 0 && estadoAct.x + col[k] < C && estadoAct.y + fil[k] >= 0
                &&
               estadoAct.y + fil[k] < F && marcaje[estadoAct.y + fil[k]][estadoAct.x + col[k]]) {
               cola.push({ estadoAct.x + col[k] , estadoAct.y + fil[k], estadoAct.longitud + 1 });
               marcaje[estadoAct.y + fil[k]][estadoAct.x + col[k]] = false;
            }
         }
      }
   }

   return -1;
}
```

Los argumentos de la función son los puntos inicial (*xini*, *yini*) y final (*xfin*, *yfin*), y la matriz *marcaje*. Esta función BFS es la que se utilizan en el problema 385 de Acepta el reto, que se detalla más en la sección de ejemplos. En este caso era más natural pasar el marcaje con aquellas posiciones que no se podían visitar como marcadas directamente para no volverlas a visitar. Sin embargo, podría pasarse una matriz como la información como parámetro y crear en el interior de la función la matriz de marcaje.

La inicialización del algoritmo consiste en inicializar la cola con el estado inicial (en caso de que este sea accesible) y marcarlo como visitado. A partir de ahí, mientras nos quede algún estado en la cola, lo sacamos y comprobamos si es un estado final, y en ese caso devolvemos la longitud encontrada, que sabemos que es la óptima. En caso de no ser un estado final, comprobamos si tiene estados adyacentes sin marcar y en ese caso los marcamos y los incorporamos a la cola.

En este caso trabajamos con una matriz en vez de con un grafo porque como ya hemos apuntado antes, la inmensa mayoría de problemas que usan esta técnica son sobre matrices. Sin embargo, para adaptarlo a grafos simplemente necesitaríamos incluir una lista de adyacencia y buscar los nodos adyacentes en ella en vez de buscar los estados siguientes en las posiciones adyacentes de la matriz.

2.1.4. Análisis de la complejidad

En primer lugar tenemos la complejidad temporal. Gracias al marcaje solo visitaremos cada nodo sin marcar a lo sumo una única vez (V nodos). Para cada uno de estos nodos procesaremos todos los nodos que sean adyacentes a través de alguna arista (E aristas), luego a lo sumo procesaremos todas las aristas una única vez. Por tanto el coste en tiempo del algoritmo está en $O(V + E)$.

Por otro lado, vamos a llevar un marcaje de si hemos visitado o no cada uno de los vértices del grafo. Necesitaremos por tanto un vector con V posiciones, luego el coste en espacio del algoritmo está en $O(V)$.

2.1.5. Ejemplos

Búsqueda en anchura para hallar el mínimo camino de un nodo origen a un nodo destino Un ejemplo de esto es el problema 385 de Acepta el reto: *La ronda de la noche*. Nos piden que dado un patio con una entrada y una salida y una serie de muros y sensores que debemos evitar, indicar el mínimo número de movimientos que necesitamos para atravesarlo, o indicar que no es posible.

La principal dificultad de este problema es la construcción de la matriz. Tenemos que tener cuidado con que un sensor no puede cortar a un muro, pero sí a otro sensor. Para ello lo más sencillo es tener dos matrices distintas, una *marcaje* y otra *patio*, y solo modificamos el *marcaje* con los datos de *patio*. Una vez construida la matriz simplemente ejecutamos una búsqueda en anchura y tenemos la solución.

Búsqueda en anchura para hallar el mínimo camino desde varios nodos origen a varios nodos destino Vemos que el algoritmo es totalmente general en cuanto a número de nodos de origen y de destino. Tan solo tenemos que introducir varios nodos origen en la cola al inicio y finalizar la búsqueda en anchura cuando un nodo que hayamos sacado de la cola sea un nodo destino.

Un ejemplo de un problema en el que necesitamos varios nodos de inicio y destino es el problema 11624 del UVa Judge: *Fire!*. En él John trabaja en una mazmorra (quién no conoce a alguien que trabaje en una mazmorra) que se está incendiando. Hay que indicar si John logrará salir de la mazmorra o no y en caso de poder conseguirlo cuál es el mínimo número de pasos en el que puede lograrlo.

En este problema tenemos dos clases de nodos origen: John y los fuegos. Iremos simulando paso a paso el movimiento de John y de las llamas mediante una búsqueda en anchura. Si John llega a una casilla marcada, o bien es una casilla que ya había visitado, en cuyo caso no tenemos que expandir el camino, o bien es una casilla en llamas, luego no podemos expandir el camino. En caso de estar la casilla libre, la exploramos, puesto que aún es un camino igualmente prometedor. Por parte de las llamas, si llegamos a una casilla marcada o bien ya está incendiada o bien ya la ha atravesado John, y no necesitamos expandir el estado. En caso de que la casilla esté libre el fuego la hace intransitable. Una cosa importante es que dentro del mismo nivel, a diferencia de la búsqueda en anchura estándar, sí que es importante el orden en el que procesemos los nodos. Puede ser que en el mismo nivel John y el fuego llegen a la misma casilla, y en este caso John no podría transitarla. Por ello tenemos que introducir a John el último en la cola (detalle que se puede apreciar en el código). Por la forma de evaluar los estados mediante la cola siempre evaluaremos antes los estados de fuego del nivel actual que el estado de John en el nivel actual, luego respetaremos esta restricción.

Por último, como nodo destino nos vale cualquiera que esté en el borde de la matriz. Por tanto, si llegamos a un borde de la matriz transitable (que no esté marcado) y además en el estado de John habremos encontrado la solución. Si esto no ocurre en ningún momento es que John no tiene tiempo para salir.

2.1.6. ¿Y qué ocurre con la búsqueda en profundidad?

La búsqueda en profundidad procesa el grafo recorriendo primero un hijo completo antes de pasar al siguiente. De nuevo se visitará cada vértice a lo sumo una vez sin marcar y después todos sus adyacentes, luego su coste también está en $O(V + E)$. Por tanto, es razonable preguntarse por qué se emplea siempre la búsqueda en anchura en lugar de escoger entre uno u otro dependiendo del caso.

Por la propia naturaleza de la búsqueda en profundidad, cuando encontremos una solución no podemos garantizar que sea la solución óptima. Esta solución nos permitirá podar ciertos caminos posteriores. Si tenemos una solución en n pasos, todos aquellos nodos a distancia mayor o igual que n del origen no necesitarán ser expandidos. De hecho, esta definición solo está clara para un árbol, para un grafo tendríamos que definir la distancia de un nodo a otro como la mínima distancia a la que están ambos nodos (ya que al ser un grafo sin restricciones puede haber varios caminos entre ambos nodos).

Por tanto aunque el coste de ambos algoritmos en el caso peor sea de $O(V + E)$, el algoritmo de búsqueda en anchura usualmente recorre menos niveles y si queremos hacer algo más eficiente la búsqueda en profundidad tendremos que implementar podas que complicarán algo el código. Luego parece razonable utilizar la búsqueda en anchura en vez de la búsqueda en profundidad para resolver esta clase de problemas.

2.1.7. Reconstruir el camino mínimo

Supongamos que queremos, además de calcular el camino mínimo, poder reconstruirlo. Una forma de hacerlo es llevar una matriz adicional al marcaje en la que para cada casilla marcamos de la casilla de la que procedemos. En caso de tenerla calculada, entonces para obtener un camino mínimo solo tendríamos que partir del nodo destino e ir mirando en la tabla qué nodo antecede al actual hasta llegar al origen (obtendríamos el camino en orden inverso). Es sencillo ver que la complejidad de este método está en $O(k)$, siendo k la longitud del camino.

Dos cosas son discutibles de este primer método. En primer lugar, existe una variación de esta técnica más avanzada que nos permite reconstruir no solo un camino mínimo sino todos los caminos mínimos, y que se discutirá en la sección del algoritmo de Dijkstra. Por otro lado, es sencillo ver que no necesitamos añadir otra tabla. Podemos simplemente sustituir el marcaje de booleanos del marcaje por un array de enteros. Los nodos no visitados (inicialmente todos salvo el origen) estarán inicializados a un valor centinela (por ejemplo, el -1). Posteriormente, marcaremos en cada nodo su antecesor en el camino, de forma que si en el marcaje el valor del nodo es un positivo significa que está marcado.

2.2. Algoritmo de Dijkstra

2.2.1. Introducción

El algoritmo de Dijkstra permite hallar el camino más corto entre dos nodos en un grafo ponderado dirigido sin aristas negativas. Fue publicado por Edsger W. Dijkstra en 1959. Es increíble como en solo 1 página describe uno de los algoritmos más conocidos y utilizados de todos los tiempos, y de una forma muy sencilla e intuitiva[1].

2.2.2. Algoritmo

El algoritmo surge de una idea muy interesante: si no hay aristas negativas, entonces si partimos desde un nodo, el nodo que aún no hayamos visitado y que esté a menor distancia del origen no puede decrementar su distancia a él. Luego simplemente iterando el proceso podemos recorrer todo el grafo hasta llegar al nodo que queremos.

Vemos además que la suposición de que las aristas son no negativas es bastante razonable para aplicaciones en la vida real. Por ejemplo en un mapa nunca nos vamos a encontrar una arista que tardemos un tiempo negativo en recorrerla, o una carretera en la que en el peaje nos tengan que pagar a nosotros.

Además, aunque la formulación natural sea minimizar la distancia a un punto, se puede pensar el algoritmo en abstracto, concibiendo los nodos simplemente como estados e ir de un estado a otro minimizando una magnitud. Entraremos en detalle sobre esto más adelante.

El pseudocódigo del algoritmo es el siguiente:

Algoritmo 2 Algoritmo de Dijkstra

```
1:  inicializar el vector de distancias a +∞
2:  introducir el estado inicial en la cola de prioridad
3:  marcar a 0 la distancia al nodo origen
4:  mientras !cola.empty() hacer
5:    estadoActual = cola.front()
6:    cola.pop()
7:    si estadoAct.nodo = nodoDestino entonces
8:      return estadoAct.distancia
9:    fin si
10:   para cada nodo adyacente hacer
11:     si distancia[nodoAdyacente] = +∞ entonces
12:       introducir el nodo en la cola
13:       marcar la nueva distancia mínima
14:     si no, si distancia[nodoActual] + arista.coste < distancia[nodoAdyacente] entonces
15:       decrementar la prioridad del nodo en la cola
16:       marcar la nueva distancia mínima
17:     fin si
18:   fin para
19: fin mientras
20: return -1
```

[1] Puede encontrarse una versión online del artículo en http://www-m3.ma.tum.de/foswiki/pub/MN0506/WebHome/dijkstra.pdf Se recomienda al lector su lectura.

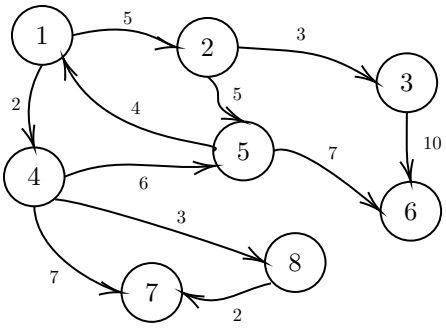
Comenzamos preparando el vector *distancias* marcando todos los nodos a $+\infty$ (a la hora de implementarlo necesitamos un valor lo suficientemente grande que pueda ser almacenado, *INT_MAX* es una buena opción en C++), para indicar que aún no hemos llegado a ellos. El nodo origen lo marcaremos con distancia 0 (no tenemos que hacer esfuerzo para llegar a él). También preparamos la cola de prioridad, que en todo momento va a mantener aquellos nodos a los que hemos llegado pero que aún no hemos procesado. Como hemos visto antes en cada momento tenemos que procesar el nodo que esté mas cerca del origen y que aún no ha sido procesado, por lo que la cola de prioridad ha de ser de mínimos.

En el bucle principal vamos a seguir hasta que o bien se acaben los nodos por procesar o bien lleguemos al nodo final. Si llegamos al nodo final, entonces sabemos que hemos llegado con coste óptimo y por tanto devolvemos directamente el coste. Si salimos del bucle porque se han acabado los nodos, sabemos que el nodo destino no está conectado por ningún camino al nodo origen, por lo que devolvemos un valor centinela, en este caso el -1.

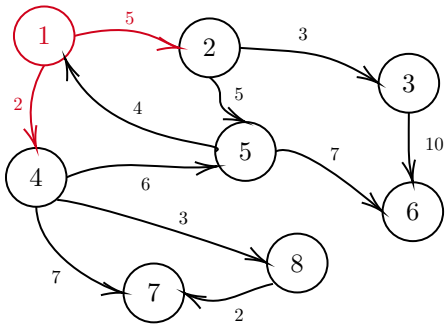
En cada iteración vamos a extraer el nodo que esté más cerca del origen y que aún no haya sido procesado. Primero comprobamos si es el nodo destino. Si no lo es, vamos a iterar por todos sus nodos adyacentes. En caso de que el nodo adyacente aún no haya sido visitado, sabemos que este es el mejor camino, luego lo marcamos en el array *distancias* e introducimos el nodo por primera vez en la cola de prioridad. Si el nodo adyacente ya había sido visitado pero el camino actual mejora el coste del camino mínimo hasta él, entonces tenemos que actualizar también el array *distancias*. Sin embargo, el nodo ya estaba dentro de la cola de prioridad, por lo que en vez de introducirlo en la cola tenemos que actualizar su distancia (que en el fondo es la prioridad).

Por último, puntualizar que la librería de C++ por ejemplo no proporciona en la cola de prioridad estándar la operación de *decrementar clave*. Por eso es habitual que en vez de decrementar la clave simplemente se inserte otra clave con el nuevo valor. Esto provoca que en el interior de la cola queden estados que no contienen la información real y que no hay que procesarlos. Por eso para cada estado hay que comprobar en el array *distancias* si es un estado que hay que procesar (si la distancia en el estado y en el array coinciden entonces el estado será correcto). Esto se verá con más claridad en la sección de código.

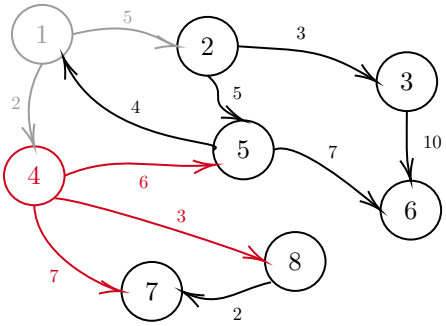
Pero antes de pasar a ver el código se expone un ejemplo de ejecución del algoritmo para clarificar su funcionamiento (Figuras 3 y 4):



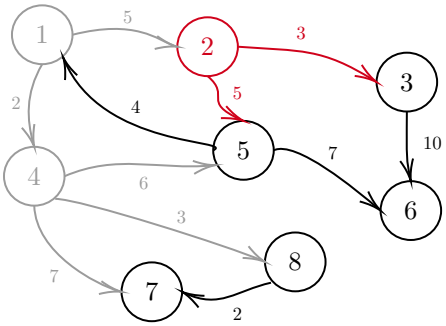
1	2	3	4	5	6	7	8
0	∞	∞	∞	∞	∞	∞	∞



1	2	3	4	5	6	7	8
0	5	∞	2	∞	∞	∞	∞



1	2	3	4	5	6	7	8
0	5	∞	2	8	∞	9	5



1	2	3	4	5	6	7	8
0	5	8	2	8	∞	9	5

Figura 3: Ejemplo de ejecución del algoritmo de Dijkstra.

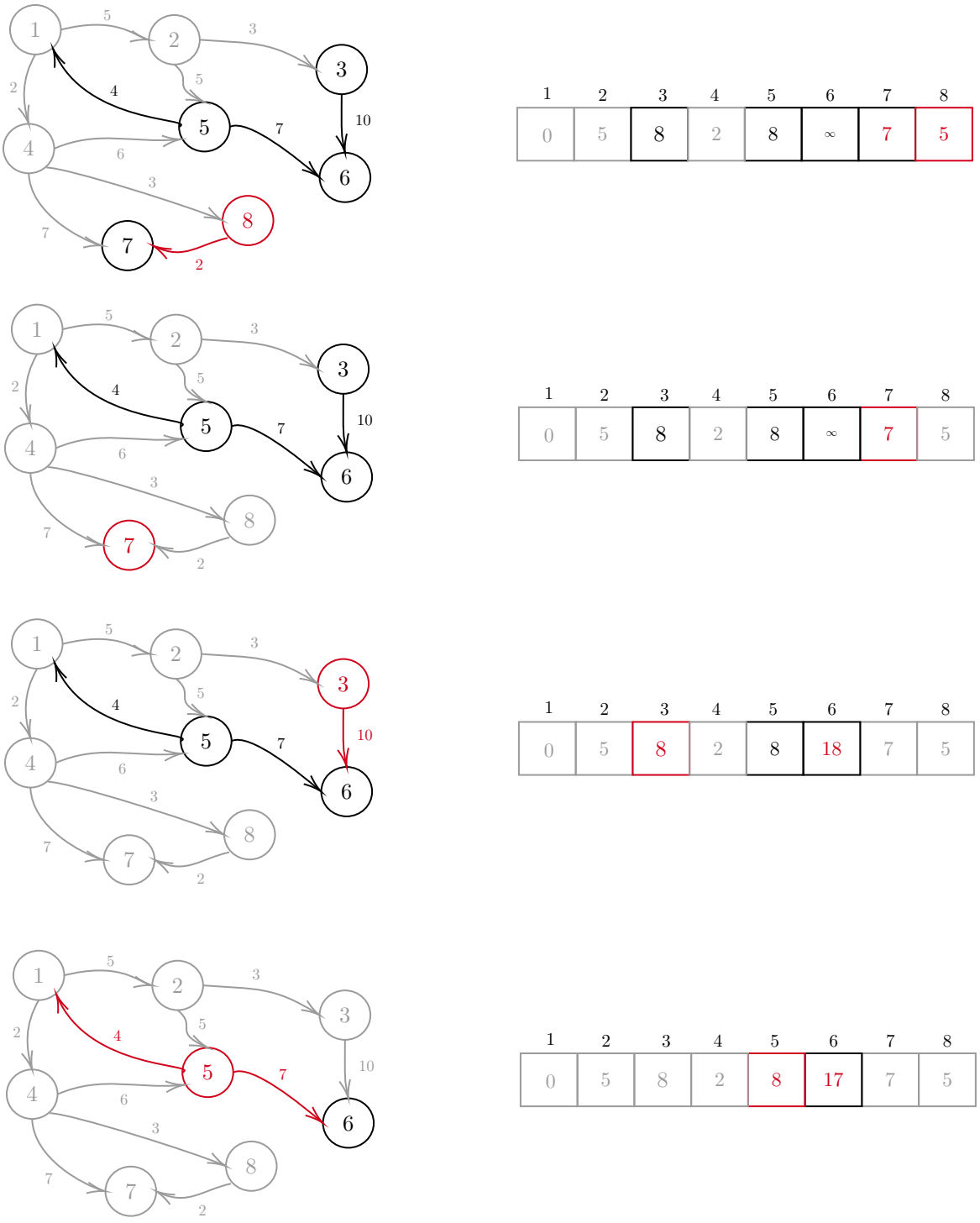


Figura 4: Ejemplo de ejecución del algoritmo de Dijkstra.

2.2.3. Código

Vamos a analizar el código del algoritmo. En esta versión el algoritmo de Dijkstra nos va a permitir calcular el tiempo mínimo desde un nodo a **todos** los demás:

```
vi dijkstra(vvii const& listaAdy, int nodoIni) {
   vi dist(listaAdy.size(), INT_MAX);

   //Queremos que salgan primero los nodos con menor coste
   //por lo que tenemos que invertir el comparador
   priority_queue<ii, vii, greater<ii>> pq;

   pq.push(mp(0, nodoIni));
   dist[nodoIni] = 0;

   while (!pq.empty()) {
      ii estadoAct = pq.top(); pq.pop();
      int costeAct = estadoAct.first;
      int nodoAct = estadoAct.second;
      if (dist[nodoAct] < costeAct) continue;
      else {
         for (int i = 0; i < listaAdy[nodoAct].size(); ++i) {
            int costeArista = listaAdy[nodoAct][i].first;
            int nodoDestino = listaAdy[nodoAct][i].second;
            if (dist[nodoDestino] > costeAct + costeArista) {
               dist[nodoDestino] = costeAct + costeArista;
               pq.push(mp(costeAct + costeArista, nodoDestino));
            }
         }
      }
   }

   return dist;
}
```

En esta versión tan solo necesitamos la lista de adyacencia y el nodo desde el que vamos a comenzar el Dijkstra. Utilizamos un vector de enteros como marcaje, e inicializamos la distancia a todos los nodos como $+\infty$ (INT_MAX).

Introducimos el nodo inicial en la cola de prioridad y marcamos que la distancia a él es 0. A partir de ahí vamos sacando nodos de la cola de prioridad por orden creciente de distancia al origen, y no paramos hasta que la cola de prioridad esté vacía. Técnicamente podríamos acabar antes en caso de que ya hayamos procesado la mejor distancia a cada nodo, ya que en ese caso tendríamos garantizado que no podemos llegar a ningún nodo con un tiempo mejor. Para implementar esto de forma eficiente simplemente necesitaríamos llevar un contador, que llamaremos *nodosProcesados* y que inicialmente está a 0. Cada vez que sacamos un nodo, si este se encuentra a mínima distancia del nodo origen (si la distancia al nodo actual coincide con la que tenemos marcada en el array *dist*) incrementamos en uno el contador. Esto no nos da problemas con los nodos repetidos porque solo incremenamos el contador cuando tenemos garantizado que estamos extrayendo un nodo a distancia óptima. En esta versión del código no se incluye esta mejora porque tiene un propósito didáctico y cuyo principal propósito es que sea fácilmente entendible.

El resto del bucle principal es exactamente igual al caso de un único nodo origen y destino, con la salvedad de que no paramos el bucle al llegar a ningún nodo particular (que en el caso de un único nodo de destino

es el nodo final). Por último, devolvemos el array con la mínima distancia a cada uno de los nodos.

2.2.4. Análisis de la complejidad

En el algoritmo de Dijkstra realizamos las siguientes operaciones:

1. Insercción: tenemos que insertar una única vez cada nodo en la cola de prioridad.
2. Borrar el mínimo: cada nodo se procesa una única vez. Como el nodo que se procesa es el que tiene una clave menor tenemos que borrarlo de la cola de prioridad.
3. Decrementar clave: a lo sumo tenemos que decrementar la clave tantas veces como aristas tenga el grafo (ya que cada arista a lo sumo mejora la distancia a un nodo).

Por tanto, dependiendo de la implementación de la cola de prioridad que escojamos variará la complejidad del algoritmo. Las implementaciones usuales (por ejemplo, un montículo de Fibonacci) tienen una complejidad para la inserción en $O(1)$, para el borrado del mínimo en $O(\log V)$ y para decrementar la clave en $O(1)$. Toda la información anterior la podemos resumir en la siguiente tabla:

	Número de repeticiones	Coste
Inserción	V	O(1)
Eliminar mínimo	V	O(log V)
Decrementar clave	E	O(1)
Total		E + V log(V)

Con lo que vemos que la complejidad del algoritmo está en $O(E + V \log(V))$. Esto es evidentemente basándonos en la versión teórica del algoritmo, no para la versión implementada en el código, que no decrementa la clave sino que mantiene los nodos en ella y añade otros nuevos.

2.2.5. Grafo de estados

Usualmente cuando pensamos en un grafo pensamos por ejemplo en que los nodos son las intersecciones de una ciudad y las aristas son las calles que las conectan. Sin embargo, en abstracto un grafo es simplemente un conjunto de nodos unidos por aristas, por lo que los nodos y aristas pueden representar lo que nosotros queramos. Aquí es donde podemos ver realmente la potencia de los algoritmos sobre grafos: mientras lo que estamos representando tenga estructura de grafo podemos aplicar el algoritmo. Esto se ve especialmente bien en el algoritmo de Dijkstra, como exploraremos en la sección de ejemplos.

2.2.6. Ejemplos

Dijkstra para encontrar los caminos más cortos hasta un punto El algoritmo de Dijkstra se emplea para resolver el problema del camino más corto desde un nodo a todos los nodos restantes del grafo. Sin embargo, es sencillo ver que empleando exactamente el mismo algoritmo podemos resolver el problema opuesto: calcular el camino más corto desde cualquier nodo a uno dado.

La idea consiste en ejecutar el algoritmo de Dijkstra recorriendo las aristas en la direccion opuesta. De esta forma, las aristas que llegan a un nodo se convierten en aristas de salida del nodo, permitiéndonos realizar el recorrido inverso. Por tanto solo necesitamos sustituir el grafo G por el grafo que tiene las aristas en sentido opuesto, que comúnmente se denota por G^R.

Un ejemplo de esto es el problema 1112 del UVa Judge: *Mice and Maze*, donde se pide calcular desde cuántos nodos del grafo lograremos llegar a un nodo en menos de un tiempo T. Para ello ejecutamos un Dijkstra sobre G^R y vemos a cuantos nodos llegamos un tiempo menor o igual a T.

Dijkstra de estados En el problema 451 de Acepta el reto: *Huyendo de los zombis*, tenemos que encontrar la forma más rápida de ir a la facultad hasta nuestra casa en autobús. Sin embargo, aquí el camino más corto no es el que tiene las aristas más cortas, sino en el que tenemos que esperar menos tiempo en la parada. En este caso no podemos emplear como estado el nodo en el que nos encontramos, puesto que dependiendo del bus que queramos coger nos puede convenir más una ruta u otra. De esta observación se desprende que el estado ha de ser una tupla (*nodoActual, minutoLlegada*).

En primer lugar construimos el grafo, en el que los vértices son las paradas de bus y las aristas son los autobuses que cogemos para ir de una parada a otra. En las aristas no solo tenemos que almacenar el tiempo que tardamos en ir de una parada a otra sino también en qué momento podemos "coger" la arista, que se corresponde al momento en el que pasa el autobús. Posteriormente ejecutamos el algoritmo de Dijkstra sobre este grafo.

Es importante distinguir entre dos cosas: los **estados del marcaje** (en este problema de la forma (*nodoActual, minutoLlegada*)), y los **estados del algoritmo** (en este problema de la forma tEstadoColaPrioridad = (*nodoActual, minutoActual, tiempoEsperaAcumulado*)). Los estados del algoritmo representan una situación a la que podemos llegar durante la ejecución del algoritmo: nosotros tomando una serie de buses llegaremos a una cierta parada en un cierto minuto y con un cierto tiempo de espera. Por otro lado los estados del marcaje son clases de equivalencia de los estados del algoritmo: de todas las veces que llegamos a una parada con un tiempo, en el estado del marcaje nos quedaremos con aquella que tenga un mejor tiempo de espera, que es la que nos conviene en este problema.

Una vez que hayamos considerado todos los posibles estados óptimos del algoritmo y terminamos la ejecución del algoritmo de Dijkstra tendremos la solución del problema.

2.2.7. Reconstruir el camino mínimo

Para reconstruir el camino mínimo desde el nodo origen a cualquier otro nodo podemos utilizar la misma idea que utilizamos en la búsqueda en anchura: llevar un segundo marcaje de enteros que nos indique para cada nodo qué nodo le precede en el camino óptimo desde el origen. Esto no incrementa la complejidad del algoritmo (simplemente si la arista mejora la distancia al nodo adyacente marcamos como precedente el nodo actual) y para reconstruir el camino solo tenemos que seguir el camino en orden inverso, con una complejidad en $O(k)$, siendo k la longitud del camino.

Sin embargo, esto solo nos permite reconstuir un camino óptimo. Si queremos reconstruirlos todos quizás la mejor idea no sea guardar todos los nodos que pueden anteceder a uno dado en un camino óptimo, puesto que un mismo nodo puede aparecer en muchos caminos como antecesor. Podemos lograr un coste en espacio polinómico (de hecho, constante, ya que solo necesitaremos el array *distancias* que ya teníamos calculado) si tenemos G^R.

La idea es la siguiente: vamos a comenzar de nuevo desde el nodo final y vamos a ir retrocediendo hasta llegar al nodo origen. Ahora no tenemos un array que nos indique cual es el predecesor, sino que tenemos que reconstruir los predecesores a partir de *distancias* y G^R. Para ello miramos todos los nodos adyacentes al nodo actual en G^R. Si $distancias[adyacente] + coste[arista] = distancias[actual]$ significa que si tenemos un camino óptimo hasta el nodo adyacente, enconces un camino óptimo hasta el nodo actual consiste en concatenar un camino óptimo hasta el nodo adyacente y esta arista. Repitiendo esto recursivamente podemos construir todos los caminos óptimos. Un ejemplo de problema en el que necesitamos esto es el problema A de la ronda de práctica del SWERC 2018[2], del que no se adjunta la solución, pero en el enlace también se puede encontrar una solución del mismo.

[2]Los problemas de la ronda de práctica del SWERC 2018 se pueden encontrar en https://swerc.eu/2018/theme/problems/practice.pdf

2.3. Algoritmo de Bellman Ford

2.3.1. Introducción

El algoritmo de Bellman Ford nos permite trabajar con grafos con aristas tanto positivas como negativas. En caso de encontrar un ciclo de coste negativo lo indica y en caso contrario calcula la longitud del camino más corto desde un nodo dado a todos los demás.

Fue propuesto por Richard Bellman [3] en el año 1958 y tiene una complejidad en $O(VE)$. La idea detrás de este algoritmo es similar a la que hay detrás del algoritmo de Floyd-Warshal, que se expondrá más adelante, y que resuelve el problema de hallar el camino mínimo entre dos nodos cualesquiera del grafo. Dado que este algoritmo es algo más simple es interesante presentalo antes que el de Floyd-Warshal para facilitar su compresión.

2.3.2. Algoritmo

Este es un algoritmo de programación dinámica. Inicialmente vamos a considerar aquellos caminos que nos permiten ir directamente del nodo inicial a otro nodo empleando tan solo una arista. En una segunda iteración vamos a considerar tanto los caminos que nos permiten ir de un nodo a otro usando una o dos aristas.

Es sencillo ver que cualquier camino simple (sin ciclos) en un grafo G con V vértices contendrá a lo sumo V - 1 aristas. Por tanto el mejor camino simple para ir del nodo origen al nodo destino podrá tener a lo sumo V - 1 aristas. Si realizamos V - 1 iteraciones del proceso que introducimos en el párrafo anterior estaremos considerando todos los posibles caminos simples que nos puedan llevar del nodo origen a cualquier otro.

Tenemos ahora dos posibilidades para ir desde el nodo origen hasta otro nodo:

1. Que ningún camino para ir desde el origen hasta el destino contenga ciclos de coste negativo. Esto significa que el mejor camino será un camino simple, y por tanto al ejecutar V - 1 pasadas del algoritmo ya tendremos calculado el mejor camino.
2. Que algún camino para ir desde el origen hasta el destino contenga ciclos de coste negativo. Esto significa que podemos coger este camino y recorrer infinitas veces el ciclo, haciendo que el coste del camino sea tan pequeño como se desea.

Exploremos un poco más en detalle este segundo caso. En primer lugar en este caso el algoritmo no podrá calcular el coste del mejor camino, puesto que no lo tenemos definido (o si acaso, podríamos definirlo como $-\infty$). En el peor de los casos el ciclo de coste negativo tendrá V aristas, luego para detectarlo no nos vale con ejecutar tan solo V - 1 pasadas del algoritmo, sino que necesitamos efectuar una más.

¿Y cómo realizaremos dicha detección? Como ya hemos realizado V - 1 iteraciones tenemos garantizado que a cada uno de los nodos (del grafo en general y del ciclo en particular) ya hemos llegado con el mínimo coste con el que podíamos llegar con un camino simple (y de hecho puede que ya hayamos tomado alguna vez el ciclo de coste negativo y que el coste sea aún menor). Al ejecutar otra relajación más de las aristas tenemos garantizado que alguna de las aristas del ciclo de coste negativo reducirá el coste para llegar al nodo vecino (en caso de que ninguna arista redujese el coste, no tendríamos ciclos de coste negativo). Por tanto, si en esta última iteración se reduce el coste de llegar a un nodo, significará que hay un ciclo de coste negativo. En caso contrario, ya tendremos la distancia mínima a todos los nodos.

Uniendo todas estas ideas tenemos que el pseudocódigo del algoritmo es el siguiente:

Algoritmo 3 Bellman-Ford

```
1: para i = 0; i < G.numVértices - 1; ++i hacer
2:    para cada arista u - v de G hacer
3:       si dist[v] ≥ dist[u] + a.coste entonces
4:          dist[v] = dist[u] + a.coste
5:       fin si
6:    fin para
7: fin para
8: para cada arista u - v de G hacer
9:    si dist[v] ≥ dist[u] + a.coste entonces
10:      mostrar "Hay una arista de coste negativo"
11:   fin si
12: fin para
```

2.3.3. Código

Dado que solo tenemos que recorrer las aristas y nos da igual el orden en el que lo hagamos, podemos hacerlo de forma eficiente empleando tanto una lista de adyacencia como una lista de aristas. Aquí se expone la implementación con lista de aristas.

```
void relajar(viii const & listaAdy, vi & dist) {
   for (int i = 0; i < listaAdy.size(); ++i) {
      int nodoOrigen = listaAdy[i].second.first;
      int nodoDestino = listaAdy[i].second.second;
      int costeArista = listaAdy[i].first;

      if (dist[nodoOrigen] != INF) {
         dist[nodoDestino] = min(dist[nodoDestino], dist[nodoOrigen] + costeArista);
      }
   }
}

//True si hay un ciclo de coste negativo, false si no
bool bellmanFord(viii const& listaAdy, int nodoIni, int numVertices) {
   vi dist(numVertices, INF);
   dist[nodoIni] = 0;

   //Realizar V - 1 veces
   for (int i = 0; i < numVertices - 1; ++i) {
      //Para cada uno de los vertices tomandolo como origen vemos si mejora la distancia a algun
          nodo con el que este conectado
      relajar(listaAdy, dist);
   }

   for (int i = 0; i < listaAdy.size(); ++i) {
      int nodoOrigen = listaAdy[i].second.first;
      int nodoDestino = listaAdy[i].second.second;
      int costeArista = listaAdy[i].first;

      if (dist[nodoOrigen] != INF && dist[nodoDestino] > dist[nodoOrigen] + costeArista) {
         return true;
      }
```

```
    }
    return false;
}
```

Parece natural además que al solo necesitar las aristas sin tener que consultar todos los vértices adyacentes a uno dado las guardemos en forma de lista de aristas. La función principal (*bellmanFord*) comienza inicializando la distancia a todos los vértices a *INF* (que definimos como un valor muy grande para que sea el elemento neutro del mínimo, podría ser INT_MAX), excepto al nodo origen que será 0. Además, en cada una de las comprobaciones vamos a comprobar que la distancia a ese nodo en *dist* es distinta a *INF*. Si esto ocurriese es porque aún no hemos encontrado la forma de llegar desde el nodo origen hasta el nodo del que sale la arista que estamos procesando. Por tanto no tiene sentido hablar de camino más corto desde el nodo origen hasta el nodo destino de la arista pasando por el nodo inicial de la arista.

Posteriormente relajaremos $V - 1$ veces todas las aristas en el bucle. Por último hacemos una última pasada comprobando si alguno de las aristas de la lista de aristas mejora la distancia a algún vértice, y como ya hemos visto previamente si ocurre esto significará que hay algún ciclo de coste negativo y paramos la ejecución. En caso de que esto no ocurra sabemos que no hay ningún ciclo de coste negativo y lo indicamos. En este caso tenemos además en el vector *dist* el camino más corto a cada nodo, y que también podríamos devolver.

Por otro lado, la función *relajar* se encarga de en cada iteración comprobar si alguna de las aristas mejora la distancia de un nodo a otro y en ese caso de actualizarla.

2.3.4. Análisis de la complejidad

El análisis de la complejidad de este algoritmo es muy sencillo. Relajaremos todas las aristas del grafo $V-1$ veces en la fase inicial y una última en la comprobación de los ciclos negativos (V veces). Cada una de esas relajaciones consiste en recorrer cada una de las E aristas del grafo y para cada arista hacer una comparación y una asignación (operaciones que consideraremos de tiempo constante). Por ende la complejidad del algoritmo está en $O(VE)$.

En cuanto al coste de espacio, aparte del coste de almacenar el grafo (que es inherente al grafo y que no se debe al algoritmo, luego no lo tendremos en cuenta), necesitamos almacenar la distancia a cada uno de los vértices, luego la complejidad en espacio estará en $O(V)$.

2.3.5. Ejemplos

Bellman Ford para averiguar si hay algún ciclo negativo en un grafo Esta es la aplicación directa del algoritmo. Un problema sencillo que nos pregunta por esto es el problema 558 del UVa Judge: *Wormholes*. Tenemos que comprobar si podemos ir a través de agujeros de gusano intergalácticos de tal forma que logremos ir tan al pasado como queramos. Aquellos agujeros que nos llevan al futuro tienen coste positivo y los que nos llevan al pasado negativo. Por tanto, el problema se reduce a determinar si hay o no un ciclo de coste negativo en el grafo. Se ha resuelto además con lista de adyacencia para ver que la modificación del algoritmo es sencilla.

Bellman Ford para averiguar qué elementos pertenecen a algún ciclo negativo en un grafo En este caso, no nos vale simplemente con relajar $V - 1$ veces los nodos y mirar una vez más si la distancia a algún nodo se reduce. Puede darse el caso de que el ciclo negativo tenga V aristas, y por ende en el peor caso tenemos que relajar V veces todas las aristas para asegurarnos de que en caso de que hemos detectado todos los elementos del ciclo de coste negativo.

Un ejemplo de esto es el problema 10449 del UVa Judge: *Traffic*, en el que dadas una serie de carreteras con coste tanto positivo como negativo, tenemos que averiguar cual es el mínimo coste con el que podemos llegar a otros nodos, y en caso de ser menor a 3 indicarlo con un *?* en vez de con la distancia. Por tanto, aquellos nodos que se encuentren en un ciclo negativo tendremos que mostrarlos con *?*. Para resolverlo ejecutamos un Bellman Ford sobre el grafo con la modificación expuesta en el párrafo anterior, precalculando todas las posibles consultas a la vez y luego mostrando los nodos que nos pidan.

2.4. Algoritmo de Floyd - Warshall

2.4.1. Introducción

El algoritmo de Floyd-Warshall nos permite encontrar la distancia mínima entre cualesquiera dos aristas del grafo, frente al resto de algoritmos que hemos estudiado que solo nos permiten encontrar la distancia de un nodo a todos los demás.

Básicamente hay tres artículos que describieron distintas variantes de este algoritmo. Los dos primeros, de Ron Bernard [4] y Robert W. Floyd [6] describen el algoritmo que nos ocupa. Por otro lado, el artículo de Stephen Warshall [9] permitía hallar el cierre transitivo de una relación binaria, algo que es muy similar conceptualmente a este algoritmo.

2.4.2. Algoritmo

El algoritmo de Floyd-Warshall es un algoritmo de programación dinámica. Comencemos pensando en el problema de ir de un nodo u a otro nodo v no usando ningún nodo intermedio. Entonces, para ir de un nodo al otro solamente podremos hacerlo usando una arista que los conecte directamente. En caso de haber varias, evidentemente nos tenemos que quedar solo con la más corta.

Supongamos ahora que queremos ir desde el nodo u hasta el nodo v pero ahora se nos permite utilizar otro nodo como nodo intermedio, al que llamaremos w. Entonces tenemos dos posibilidades. O bien quedarnos con el camino anterior (que iba de u a v directamente), o bien podemos tomar un camino que vaya de u a w y luego otro que vaya de w a v. Puede ser que no exista alguno de los dos caminos, al igual que puede ser que antes no existiese un camino directo. Pero en caso de existir uno o varios, nos quedamos con el más corto.

Ahora, en vez de usar un nodo arbitrario como nodo intermedio vamos a ir usando como nodos intermedios los distintos nodos del grafo por orden. Una vez hemos permitido usar todos los nodos del grafo como nodos intermedios habremos encontrado los caminos mínimos entre cualesquiera dos nodos del grafo. Si llamamos i al nodo inicial, j al nodo final y k al nodo hasta el cual podemos utilizar como intermedio nos queda la recurrencia:

minCamino(i,j,k) = min(minCamino(i, j, k-1), minCamino(i, k, k - 1) + minCamino(k, j, k - 1))

Y como casos base:

minCamino(i,i,0) = 0
minCamino(i,j,0) = costeArista(i,j) si existe una arista que una ambos nodos
minCamino(i,j,0) = ∞ si no existe una arista que una ambos nodos

Ahora que tenemos un algoritmo que parece funcionar vamos a ver si podemos mejorarlo. No parece que podamos reducir su complejidad temporal: tenemos que, para cada pareja de vértices (V^2), utilizar cada nodo del grafo (V) como nodo intermedio. Sin embargo, sí que podemos reducir la complejidad espacial. Si nos fijamos en la recurrencia de $minCamino(i, j, k)$ vemos que necesitamos tres términos: $minCamino(i, j, k-1)$, $minCamino(i, k, k-1)$, $minCamino(k, j, k-1)$, todos ellos con el parámetro k dependiente del nivel $k-1$. Por tanto, no necesitamos mantener 1 matriz de tres dimensiones, sino 2 matrices de dos dimensiones: una con el nivel $k-1$, de la cual sacamos la información que necesitamos para calcular el nivel actual, y otra con el nivel k que es la información que estamos completando actualmente.

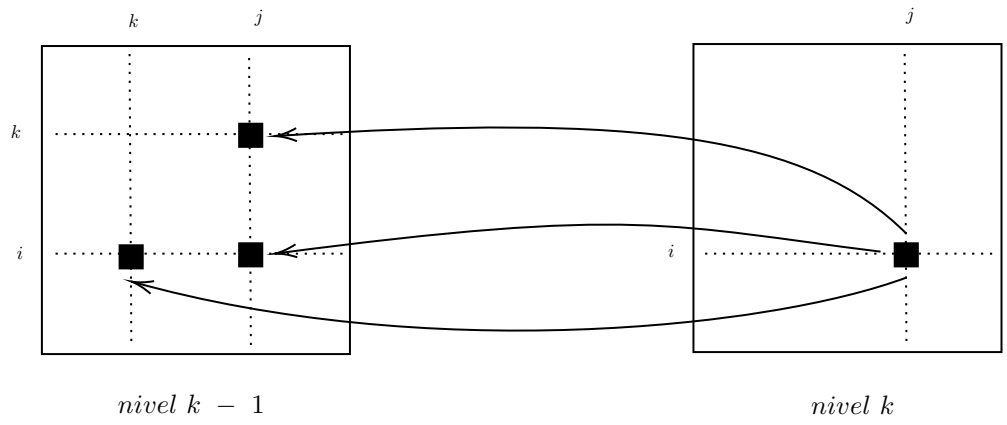


Figura 5:

Observemos ahora la fórmula de la recurrencia para el elemento en la posición (i, k) nivel k:

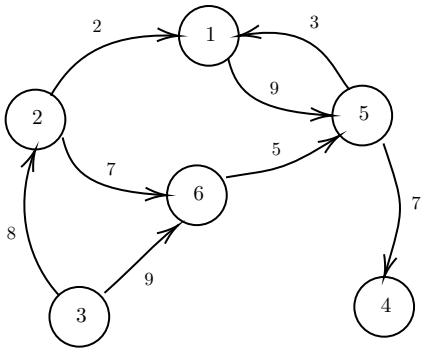
$$\begin{aligned} minCamino(i,k,k) &= min(minCamino(i,k,k-1), minCamino(i,k,k-1) + minCamino(k,k,k-1)) = \\ &= min(minCamino(i,k,k-1), minCamino(i,k,k-1)) = minCamino(i,k,k-1) \end{aligned} \tag{1}$$

Recordemos que uno de los casos base es que $minCamino(i, i, 0) = 0$, luego en todas las iteraciones el mínimo entre 0 y un número positivo será 0 y por tanto $minCamino(i, i, s) = 0 \ \forall \ s$. De igual manera para el elemento en la posición (k, j) tenemos:

$$\begin{aligned} minCamino(k,j,k) &= min(minCamino(k,j,k-1), minCamino(k,k,k-1) + minCamino(k,j,k-1)) = \\ &= min(minCamino(k,j,k-1), minCamino(k,j,k-1)) = minCamino(k,j,k-1) \end{aligned} \tag{2}$$

Luego tenemos que los elementos que necesitamos para calcular el nivel k (los elementos que se encuentran en la fila y en la columna k) no se van a modificar respecto al nivel $k-1$. Esta observación es muy importante, puesto que nos permite usar una única matriz en vez de 2, sobrescribiendo los resultados del nivel k sobre los del nivel $k-1$ teniendo la garantía de que no vamos a modificar ningún dato que posteriormente fuésemos a necesitar para el cálculo de otro camino.

Para ilustrar el funcionamiento del algoritmo tenemos el siguiente ejemplo:



	1	2	3	4	5	6
1	0	∞	∞	∞	9	∞
2	2	0	∞	∞	∞	7
3	∞	8	0	∞	∞	9
4	∞	∞	∞	0	∞	∞
5	3	∞	∞	7	0	∞
6	∞	∞	∞	∞	5	0

$k = 0$

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

$k = 1$

	1	2	3	4	5	6
1	0	∞	∞	∞	9	∞
2	2	0	∞	∞	11	7
3	10	8	0	∞	19	9
4	∞	∞	∞	0	∞	∞
5	3	∞	∞	7	0	∞
6	∞	∞	∞	∞	5	0

$k = 2$

	1	2	3	4	5	6
1	0	∞	∞	∞	9	∞
2	2	0	∞	∞	11	7
3	10	8	0	∞	19	9
4	∞	∞	∞	0	∞	∞
5	3	∞	∞	7	0	∞
6	∞	∞	∞	∞	5	0

$k = 3$

	1	2	3	4	5	6
1	0	∞	∞	∞	9	∞
2	2	0	∞	∞	11	7
3	10	8	0	∞	19	9
4	∞	∞	∞	0	∞	∞
5	3	∞	∞	7	0	∞
6	∞	∞	∞	∞	5	0

$k = 4$

	1	2	3	4	5	6
1	0	∞	∞	16	9	∞
2	2	0	∞	18	11	7
3	10	8	0	26	19	9
4	∞	∞	∞	0	∞	∞
5	3	∞	∞	7	0	∞
6	8	∞	∞	12	5	0

$k = 5$

	1	2	3	4	5	6
1	0	∞	∞	16	9	∞
2	2	0	∞	18	11	7
3	10	8	0	21	14	9
4	∞	∞	∞	0	∞	∞
5	3	∞	∞	7	0	∞
6	8	∞	∞	12	5	0

$k = 6$

Figura 6: Ejemplo del algoritmo de Floyd para hallar los caminos mínimos entre dos nodos cualesquiera de un grafo

Hablemos ahora un poco de el algoritmo desarrollado por Warshall para hallar el cierre transitivo de una relación binaria, y entender así por qué ambos algoritmos están tan íntimamente relacionados. Una relación binaria es una relación R de elementos entre dos conjuntos A y B. Supongamos $A = B$, se define el cierre transitivo de la relación binaria como la menor relación binaria R' que contiene a R y además es transitiva. Esto significa que si en R existen las relaciones $a \to b$ y $b \to c$, entonces en R' además de estas dos se tiene que encontrar $a \to c$.

Vamos a representar la relación binaria como una matriz de booleanos. En caso de que exista una relación entre dos términos marcaremos la casilla a true y en caso contrario a false. Si queremos encontrar el cierre transitivo del conjunto necesitamos encontrar aquellos términos que, empleando un número arbitrario

de términos intermedios, estén relacionados. Vemos que una forma de resolver el problema es mediante la siguiente recurrencia:

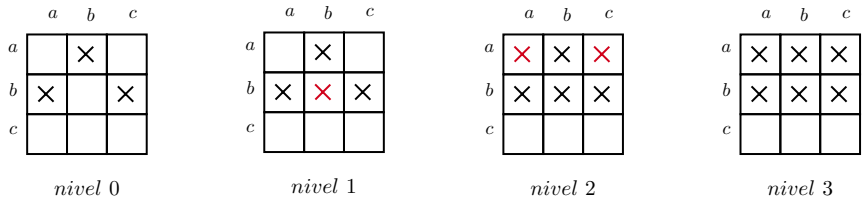
$$existeRelación(i,j,k) = existeRelación(i,j,k-1) \parallel (existeRelación(i,k,k-1) \;\&\&\; existeRelación(k,j,k-1))$$

Donde $\parallel$ y && son el OR y el AND booleano respectivamente. Y como casos base:

$existeRelación(i,j,0) = true$	si ambas variables están relacionadas en R
$existeRelación(i,j,0) = false$	si ambas variables no están relacionadas en R

Evidentemente la recursión es la misma sustituyendo enteros por booleanos y de ahí que ambos algoritmos sean básicamente el mismo. Un pequeño ejemplo para clarificar el algoritmo:

$$R = \{(a,b),(b,\ c),(b,a)\}$$



$$R' = \{(a,b),(b,\ c),(b,a),(a,a),(b,b),(a,c)\}$$

Figura 7: Ejemplo del algoritmo de Warshall para hallar el cierre transitivo de un grafo

2.4.3. Código

Una implementación del algoritmo podría ser:

```
void Floyd-Warshall(vvi & m) {
   for (int k = 0; k < m.size(); ++k) {
      for (int i = 0; i < m.size(); ++i) {
         for (int j = 0; j < m.size(); ++j) {
            m[i][j] = min(m[i][j], m[i][k] + m[k][j]);
         }
      }
   }
}
```

Vemos que la implementación, una vez hemos hecho todo el razonamiento teórico es totalmente trivial. Simplemente tenemos que iterar en el bucle exterior para ir permitiendo un mayor número de nodos intermedios, y en los dos bucles interiores sobre los nodos entre los que queremos hallar el camino mínimo.

2.4.4. Análisis de la complejidad

Es trivial ver que la complejidad de este algoritmo está en $O(V^3)$, puesto que tenemos tres bucles anidados y en cada uno de ellos hacemos V iteraciones, el bucle interior con una operación constante.

La parte interesante del análisis de complejidad es ver la complejidad en espacio. En apariencia, al ser un algoritmo de programación dinámica en el que tenemos 3 estados, debería de ser una tabla con tres dimensiones, lo que nos dejaría un coste en espacio en $O(V^3)$. Sin embargo, como hemos visto podemos reducir una dimensión la tabla, reduciendo el coste en espacio a $O(V^2)$.

Por otro lado, tenemos que el algoritmo de Floyd-Warshall nos permite hallar la distancia entre todos los nodos del grafo con un coste en $O(V^3)$. Si ejecutamos el algoritmo de Dijkstra V veces (una desde cada nodo del grafo), tendría un coste en $O(V(E + V\ log(V))) = O(VE + V^2\ log(V))$. Evidentemente, el caso peor es que el grafo sea completo, en cuyo caso tendrá $V(V-1)/2 \approx V^2$ aristas (obviamente si dos nodos están unidos por más de una arista nos quedaremos con la más corta). Por tanto, en el peor caso, la complejidad de ejecutar V Dijkstras ($O(V^3 + V^2\ log(V)) = O(V^3)$) y de ejecutar un Floyd es la misma, y si el grafo es disperso (el número de aristas es pequeño), seguirá siendo mejor ejecutar múltiples veces el algoritmo de Dijkstra. La principal ventaja que nos aporta el algoritmo de Floyd es su sencillez: tan solo necesitamos 4 líneas de código para codificarlo. Además, aunque asintóticamente la complejidad sea la misma, el algoritmo de Dijkstra tiene mayores constantes implícitas.

2.4.5. Ejemplos

Un ejemplo estándar de utilización del algoritmo es el problema 821 del UVa Judge: *Page Hopping*. El objetivo es, dada una serie de páginas web, hallar el mínimo camino entre dos cualesquiera y computar la media. Solamente tenemos que aplicar el algoritmo de Floyd y después hallar la media entre todas las distancias.

2.5. Caminos cortos en un grafo acíclico

2.5.1. Introducción

El problema de hallar el camino más corto en un grafo acíclico dirigido es exactamente el mismo que en un grafo general, pero se puede aprovechar la estructura del grafo para reducir la complejidad frente al caso general.

2.5.2. Algoritmo

La idea es exactamente la misma que se desarrollará más adelante para los caminos largos en grafos acíclicos dirigidos más adelante. Por eso en esta sección el desarrollo de la teoría será bastante breve. Se recomienda al lector mirar primero la sección 2.2 y luego volver aquí para ver las ligeras diferencias entre ambos algoritmos.

El algoritmo de nuevo consiste en hallar el orden topológico del grafo. Después, iremos actualizando los nodos de acuerdo al orden topológico. Como el grafo es acíclico, una vez que hayamos procesado un nodo (evidentemente siguiendo el orden topológico), no podemos tomar una arista que vuelva a él y disminuya el coste, por lo que al procesar un nodo tendremos garantizado que hemos llegado a él con la distancia óptima. Iterando el proceso calculamos la mínima distancia de un nodo a todos los demás.

El algoritmo quedaría así:

Algoritmo 4 Camino más corto en un DAG

```
1:  Hallar el orden topológico de G
2:  Inicializar la distancia a todos los nodos a ∞
3:  Inicializar la distancia al nodo inicial a 0
4:  para cada nodo por orden topológico hacer
5:    si hay un camino desde el origen hasta el nodo entonces
6:      para cada nodo adyacente hacer
7:        minDistancia[nodoAdyacente] =
8:        = min(minDistancia[nodoAdyacente], minDistancia[nodoActual] + longitudArista)
9:      fin para
10:   fin si
11: fin para
```

Vemos que tan solo hay 2 diferencias frente al caso de hallar el camino más largo. En primer lugar las líneas 7 y 8 calculan el mínimo entre el camino que llega al nodo adyacente que habíamos calculado previamente y el camino que llega al nodo actual y que utiliza la arista que estamos utilizando para llegar al nodo adyacente (antes calculábamos el máximo). El otro cambio es en la línea 2: dado que utilizamos el mínimo en vez de el máximo tenemos que sustituir $-\infty$, que es el elemento neutro para el máximo, por ∞, que es el elemento neutro para el mínimo.

Y ahora un ejemplo de ejecución:

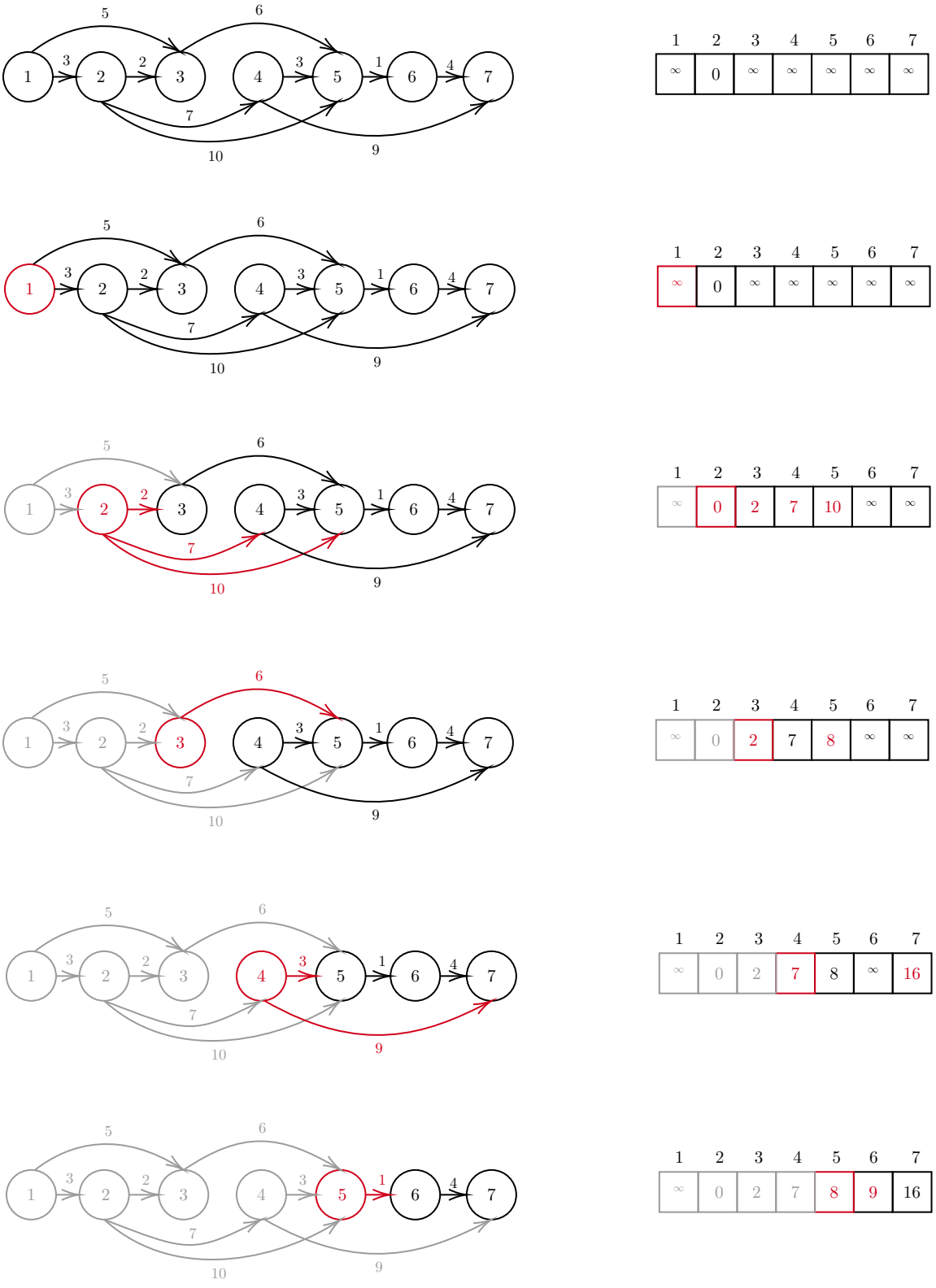


Figura 8: Ejemplo de obtención de caminos mínimos en un grafo acíclico.

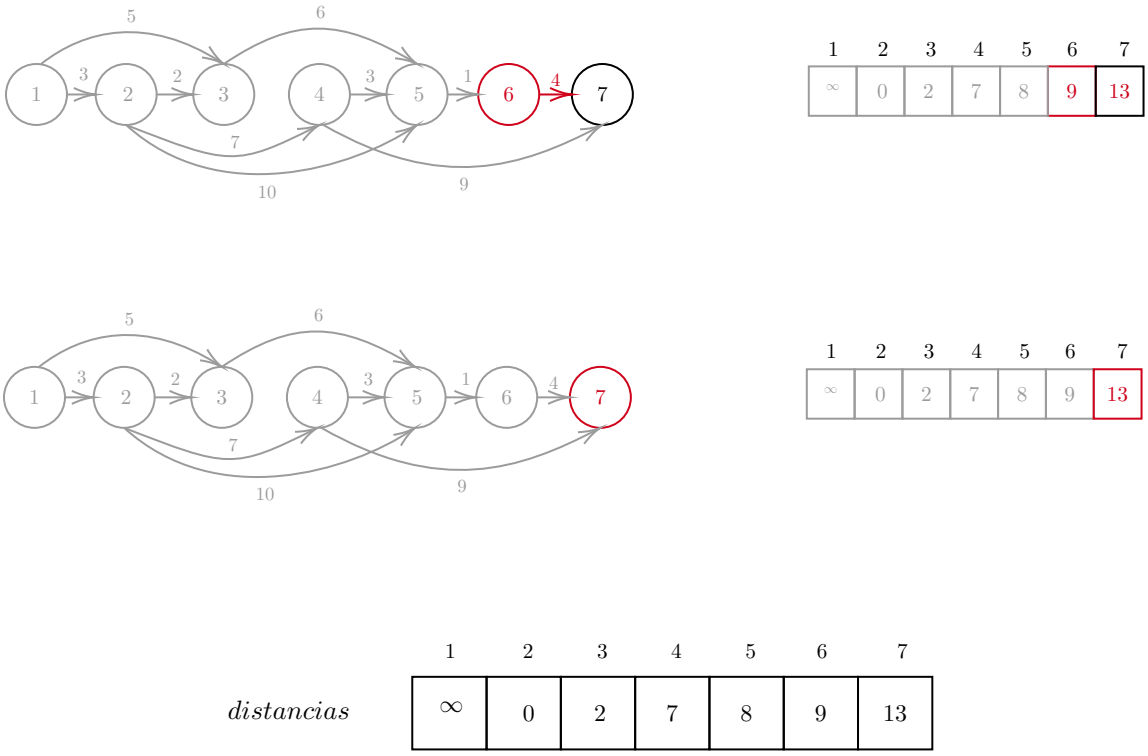


Figura 9: Ejemplo de obtención de caminos mínimos en un grafo acíclico.

2.5.3. Código

Un código que implemente el algoritmo podría ser:

```
vi caminosMinimosDAG(vector<vector<arista>> const& listaAdy) {
   vi distancias(listaAdy.size(), INT_MAX);

   vi ordenTopologico = kahn(listaAdy);
   distancias[0] = 0;

   for (int i = 0; i < ordenTopologico.size(); ++i) {
      int nodoAct = ordenTopologico[i];
      if (distancias[nodoAct].first != INT_MAX) {
         for (int j = 0; j < listaAdy[nodoAct].size(); ++j) {
            if (distancias[nodoAct] + listaAdy[nodoAct][j].coste <
                distancias[listaAdy[nodoAct][j].fin]) {
               distancias[listaAdy[nodoAct][j].fin] = distancias[nodoAct] +
                   listaAdy[nodoAct][j].coste;
            }
         }
      }
   }
   return distancias;
}
```

En primer lugar inicializamos el array *distancias* a ∞. Posteriormente hallamos el orden topológico del grafo acíclico dirigido (en este caso se hace con el algoritmo de Kahn, pero podría hacerse con cualquier otro). A continuación tenemos que procesar los nodos por orden topológico. Para cada nodo comprobamos en primer lugar que exista un camino desde el nodo origen. Después tenemos que, para cada uno de sus nodos adyacentes, actualizar la distancia si hemos descubierto un camino mejor. Esto ocurrirá si la menor distancia al nodo adyacente que hemos encontrado hasta ahora es mayor que la distancia hasta el nodo actual más la longitud de la arista que estamos considerando.

2.5.4. Análisis de la complejidad

El análisis es el mismo que se realiza en la sección 2.2.4. Tenemos que hallar primero el orden topológico del grafo (coste temporal en $O(V + E)$) y posteriormente recorrer los vértices y las aristas, cada uno de ellos una única vez, realizando para cada uno de ellos operaciones con coste constante, luego el coste de nuevo es $O(V + E)$. La complejidad temporal total está por tanto en $O(V + E)$.

En cuanto a la complejidad espacial, necesitamos un array para guardar el orden topológico del grafo y otro para guardar las distancias al nodo origen, ambos de tamaño V. Por tanto la complejidad espacial está en $O(V)$.

2.5.5. Ejemplos

Probablemente el problema más complicado de todos los que se resuelven en este trabajo es el problema E del SWERC 2017, que es también el problema 13286 del UVa Judge: *Ingredients*. Una prestigiosa chef está esperando la visita de unos catadores. Tiene un presupuesto límite para el menú que les va a preparar y quiere ofrecerles el menú que tenga el mayor *"prestigio"* posible. Cada plato tiene un coste y un prestigio. No se pueden repetir platos. Además, no se dan los platos con prestigio y coste directamente, sino que se dan las recetas. Hay platos *básicos* que no tienen coste ni prestigio. El resto de platos son platos *compuestos*, que se forman al añadir a un plato otro ingrediente, añadiendole además coste y prestigio. Se garantiza que ninguno de los platos se forma al añadirle un ingrediente a si mismo. Puede haber además distintas formas de conseguir el mismo plato, nos quedaremos en ese caso con el de mayor prestigio.

El problema hay que dividirlo en dos partes. La primera, que podríamos considerar la *"construcción de la carta"* tenemos que ver que platos podemos formar con las indicaciones que se nos dan, cuál es el prestigio y coste de cada uno. Esto equivale a hallar el camino óptimo (de menor coste o en caso de empate de mayor prestigio) desde un plato inicial a cada uno dado. Esta es la parte que se resuelve mediante caminos mínimos en un grafo acíclico dirigido. La segunda parte consiste en ver cuál es la elección óptima de platos para maximizar el prestigio, que se resuelve con un knapsack 0-1. Se adjunta también la implementación con el algoritmo de Dijkstra, que también se puede utilizar para su resolución.

3. Caminos largos

El análisis de los caminos largos tiene también un gran número de aplicaciones, entre las que tenemos por ejemplo encontrar el camino crítico en un proceso de producción. Sin embargo cualquier lector con conocimientos intermedios en algoritmia se dará cuenta de que la teoría en torno a ellos es mucho menos extensa que en torno a como hallar el camino más corto. Por ejemplo, uno de los libros de referencia en la algoritmia: *Introduction to Algorithms* [5], dedica tan solo dos párrafos y un ejercico a un caso particular del problema de encontrar el camino más largo (Pág. 657), frente a los dos capítulos (Cáps. 24 y 25) que dedica al problema de hallar el camino más corto en diferentes situaciones.

La razón de esto es que para un grafo general, el problema de hallar el camino más largo es un problema NP-completo, de lo cual se dará una prueba más adelante. El caso particular interesante son los gráfos acíclicos dirigidos (DAG) en los cuales se puede hallar una solución al problema en tiempo lineal.

3.1. Grafo general

3.1.1. Introducción

En primer lugar tenemos que definir exactamente el problema al que nos estamos enfrentando. Dado un grafo G, no tendría mucho sentido considerar el camino más largo en G que no sea un camino no simple, puesto que podríamos pasar por cada vértice un número arbitrario de veces y por ende la solución al camino más largo sería siempre ∞. Vamos a buscar por tanto el camino simple más largo en G.

3.1.2. NP-Completitud

A continuación probamos que la versión de decisión del problema es NP-Completo, que podríamos formular como: *Dado un grafo G y un número k, ¿existe en G un camino simple de al menos longitud k?*. Para probar la NP-Completitud tenemos que probar que el problema está en NP y que existe una reducción polinómica del problema a otro NP-Completo.

Veamos en primer lugar que el problema está en NP. Para ello tenemos que, dada una posible solución al problema, verificar en tiempo polinómico si efectivamente es solución al problema. Esto podríamos hacerlo mediante la siguiente función:

```
bool verificar(vvi const& matrizAdy, vi camino, int k) {
   int sumaCoste = 0;
   for (int i = 0; i < camino.size() - 1; ++i) {
      if (matrizAdy[camino[i]][camino[i + 1]] != -1) {
         sumaCoste += matrizAdy[camino[i]][camino[i + 1]]
      }
      else return false;
   }
   //Si la longitud del camino es mayor que la distancia que nos pedian
   if (sumaCoste >= k) return true;
   //Si el camino no tiene la longitud deseada
   else return false;
}
```

Suponemos que tenemos almacenado el grafo como una matriz de adyacencia, y que aquellas aristas que no existan están marcadas a -1. Entonces miramos para cada pareja de nodos adyacentes en el camino si existe una arista que los conecte. En caso de que existan todas las aristas intermedias, vemos si la suma de sus longitudes es mayor o igual que k. En caso afirmativo el camino es una solución válida al problema y en caso contrario no.

Ahora tenemos que encontrar una reducción polinómica de un problema NP-Completo a él. En este caso vamos a hacerlo al problema de decisión del camino Hamiltoniano [3]. Está demostrado en la página 271 de [7] que el problema de búsqueda del camino Hamiltoniano (en el libro llamado Rudrata path problem) se puede reducir polinómicamente al problema de búsqueda del ciclo Hamiltoniano (análogo para la versión de decisión), y este se puede reducir varias veces hasta llegar al problema SAT. Ahora vamos a reducir el problema de decisión del camino Hamiltoniano ($PDCH$) al problema de hallar el camino más largo en un grafo general (CML), esto es:

$$PDCH \leq^p CML$$

Para ello tenemos en primer lugar que transformar una instancia de $PDCH$ en una de CML en tiempo polinómico. Una instancia de $PDCH$ viene dada por un grafo G, y una de CML por un grafo G' y un valor k. Vamos a transformarla haciendo que G' sea un grafo con las mismas aristas que G pero el coste de estas aristas es 1, y como valor de k vamos a escoger $V-1$. Una función de coste polinómico que realice esto podría ser:

```
pair<Grafo, int> transformar(Grafo G) {
   Grafo Gprima;
   for (Vertice v : G) {
      for (Arista a : G.nodosAdyacentes(v)) {
         Gprima.insertarArista(v, a.destino, a.coste);
      }
   }
   return make_pair(Gprima, G.numVertices() - 1);
}
```

Por último tenemos que probar que $PDCH$ devuelve cierto $\iff$ CML devuelve cierto

$\Rightarrow$) Supongamos que PDCH nos devuelve que existe un camino Hamiltoniano. Sabemos que dicho camino tiene longitud $V-1$ (pasa por todos los vértices del grafo, V), luego existiría un camino de longitud al menos $V-1$ en el grafo y por tanto CML devolvería cierto.

$\Leftarrow$) Supongamos que CML devuelve cierto. Entonces existe en el grafo un camino simple de longitud al menos $V-1$. No obstante, como todas las aristas tienen longitud 1, esto es equivalente a decir que el camino tiene al menos $V-1$ aristas. Tenemos que cualquier camino simple ha de tener a lo sumo $V-1$ aristas (al ser simple no se pueden repetir vértices, y consideramos que no es un ciclo, luego no pueden coincidir tampoco el primer vértice con el último). Por tanto el camino ha de tener exactamente $V-1$ aristas

Por ser simple pasa por V vértices distintos. Como G' tiene exactamente V vértices, entonces tenemos que el camino que hemos encontrado es además un camino Hamiltoniano. Este mismo camino en G nos devuelve cierto al problema PDCH.

3.2. Grafo acíclico dirigido

3.2.1. Introducción

No tenía mucho interés resolver el problema de caminos largos para un caso general desde un punto de vista algoritmico: no disponemos de un algoritmo eficiente para hacerlo por ser un problema NP-completo. Tampoco podemos aplicar programación dinámica al problema puesto que no verifica el principio de optimalidad, por lo que tenemos que conformarnos con soluciones de fuerza bruta o ramificación y poda. El caso interesante está cuando el grafo es acíclico, que como veremos sí que tiene una solución óptima. Es

[3] Sacado de [1]

sorprendente como un problema que en apariencia parece dual al de los caminos cortos en el caso general no tiene nada que ver con él, pero que en el caso de grafos acíclicos es exactamente igual.

3.2.2. Algoritmo

El algoritmo consta de dos fases. En primer lugar vamos a hallar un orden topológico del grafo. El orden topológico de un grafo consiste en ordenar los vértices de tal forma que ningún vértice posterior tenga un arista saliente hacia un vértice anterior. En un grafo general el orden topológico no tiene sentido, puesto que si existe un ciclo no podremos hallar una ordenación de los vértices con estas características.

Es sabido además que el orden topológico de un grafo no es único. Para este algoritmo en particular no nos importará el orden topológico obtenido, por lo que podemos utilizar cualquier método para hallarlo (si sí que nos importase el orden topológico obtenido, entonces no podríamos utilizar por ejemplo una búsqueda en profundidad para hallarlo).

Inicialmente asigmanos a todos los nodos una distancia arbitrariamente pequeña, excepto al nodo origen al que le asignamos una distancia de 0. Posteriormente procesamos los nodos por orden topológico. Para cada arista saliente de este nodo, la distancia al nodo destino será el máximo entre la distancia que ya hubiésemos calculado y la distancia necesaria para llegar al nodo que estamos procesando más la distancia de la arista que estamos considerando. Aquí se ve además por qué asignamos inicialmente una distancia arbitrariamente pequeña a los nodos (porque es el elemento neutro para el máximo).

Tras procesar todos los nodos, tendremos en el array con las distancias el camino más largo a cada uno de los nodos. El pseudocódigo del algoritmo será por tanto:

Algoritmo 5 Camino más largo en un DAG

```
1: Hallar el orden topológico de G
2: Inicializar la distancia a todos los nodos a -∞
3: Inicializar la distancia al nodo inicial a 0
4: para cada nodo por orden topológico hacer
5:    si hay un camino desde el origen hasta el nodo entonces
6:       para cada nodo adyacente hacer
7:          maxDistancia[nodoAdyacente] =
8:          = max(maxDistancia[nodoAdyacente], maxDistancia[nodoActual] + longitudArista)
9:       fin para
10:   fin si
11: fin para
```

Y un ejemplo de ejecución para ver el funcionamiento del algoritmo:

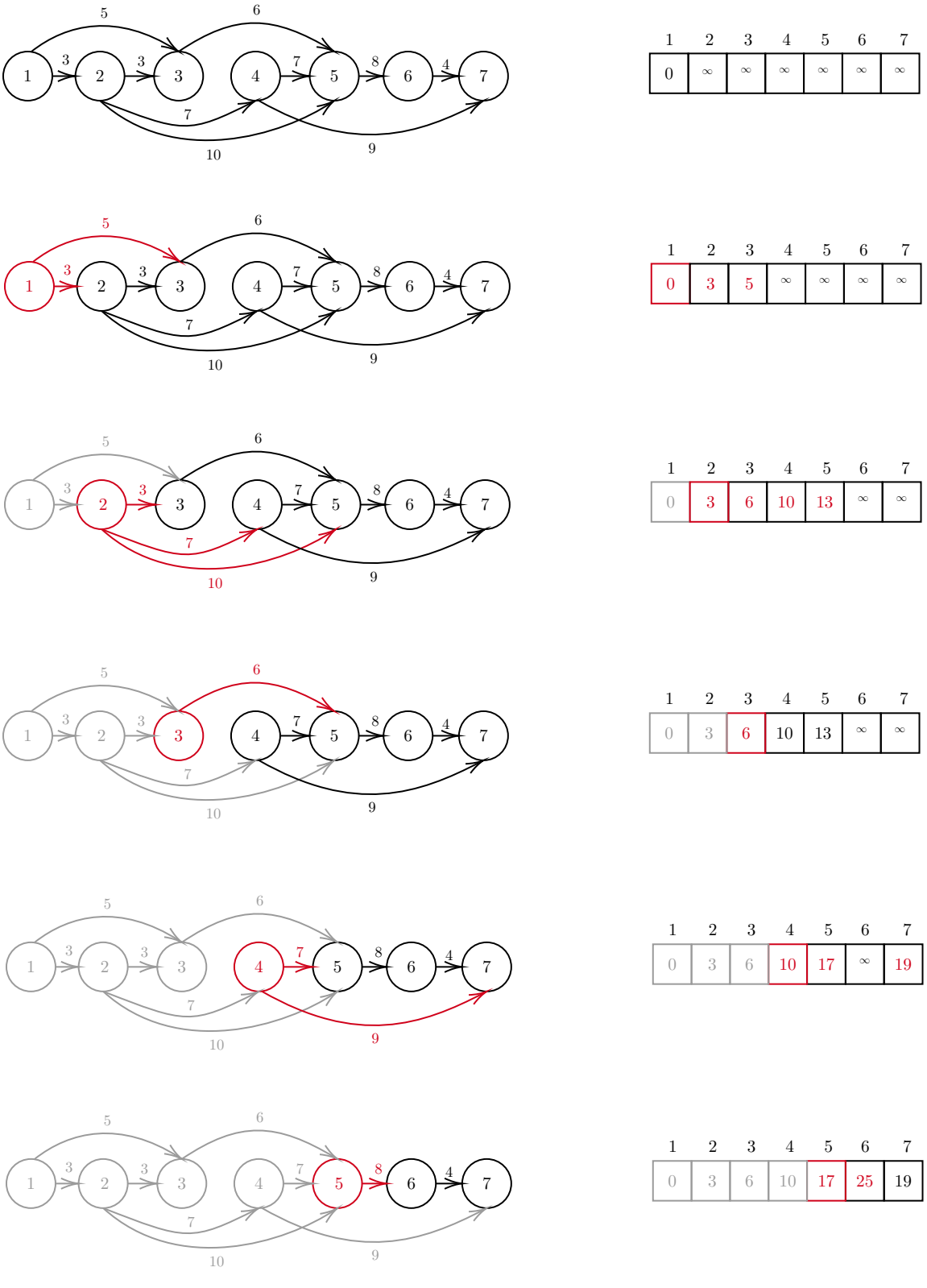

1 2 3 4 5 6 7
0 ∞ ∞ ∞ ∞ ∞ ∞
1 2 3 4 5 6 7
0 3 5 ∞ ∞ ∞ ∞
1 2 3 4 5 6 7
0 3 6 10 13 ∞ ∞
1 2 3 4 5 6 7
0 3 6 10 13 ∞ ∞
1 2 3 4 5 6 7
0 3 6 10 17 ∞ 19
1 2 3 4 5 6 7
0 3 6 10 17 25 19

Figura 10:

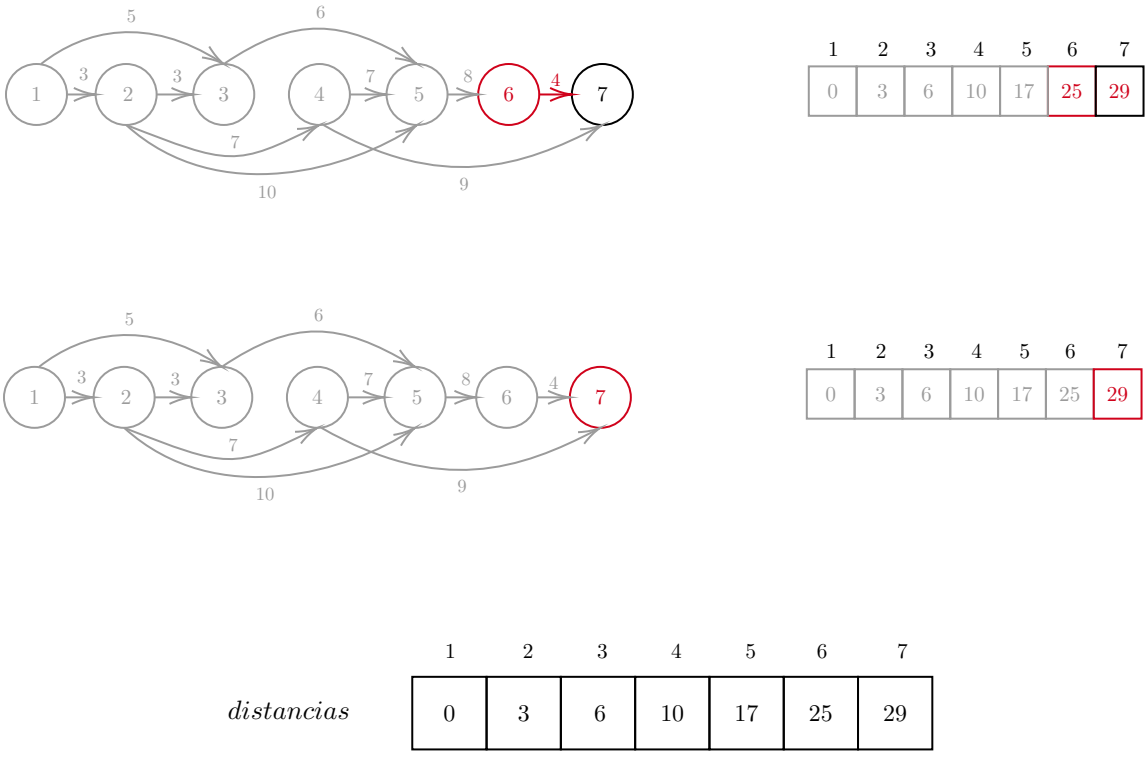


Figura 11:

3.2.3. Código

Una implementación del pseudocódigo anterior podría ser:

```
vi caminoMasLargo(vvii const& listaAdy, int nodoIni) {
   //En primer lugar calculamos un orden topologico del grafo
   vi ordenTopo = kahn(listaAdy);

   //Y ahora computamos las maximas distancias
   vi distancias(listaAdy.size(), INT_MIN);
   distancias[nodoIni] = 0;

   for (int i = 0; i < ordenTopo.size(); ++i) {
      int nodoAct = ordenTopo[i];
      if (distancias[nodoAct] != INT_MIN) {
         for (int j = 0; j < listaAdy[nodoAct].size(); ++j) {
            int nodoDestino = listaAdy[nodoAct][j].second;
            int costeArista = listaAdy[nodoAct][j].first;
            distancias[nodoDestino] = max(distancias[nodoDestino], distancias[nodoAct] +
                costeArista);
         }
      }
   }
   return distancias;
}
```

En primer lugar se calcula un orden topológico del grafo representado por *listaAdy* y se guarda en el vector *ordenTopo*. Se omite aquí la implementación de la función *kahn* que se encarga de calcular el orden topológico del grafo. Si el lector lo desea puede encontrar su implementación en los ejemplos de aplicación del algoritmo.

Posteriormente, para cada nodo del grafo por orden topológico comprobamos que haya un camino desde el origen hasta él (esto ocurre si en *distancias* la distancia al nodo es distinta de *INT_MIN*). Si hay un camino hasta el nodo, entonces actualizamos la distancia a los nodos adyacentes. Tras procesar todos los nodos tenemos en *distancias* la mínima distancia a todos los vértices, que es lo que devolvemos.

3.2.4. Análisis de la complejidad

Hallar el orden topológico de un DAG tiene una complejidad temporal en $O(V + E)$. Posteriormente, recorremos cada vértice una vez y procesamos cada una de las aristas que tienen como origen ese nodo realizando operaciones de tiempo constante (suma, comparación y asignación). Como resultado de esto la segunda parte del algoritmo tiene también una complejidad temporal en $O(V + E)$. Por tanto la complejidad temporal total del algoritmo está en $O(V + E)$.

En cuanto a la complejidad en espacio, necesitamos dos arrays de tamaño V, uno para almacenar el orden topológico del grafo y otro para almacenar la distancia a cada uno de los elementos. Por tanto la complejidad espacial total estará en $O(V)$.

3.2.5. Ejemplos

Un ejemplo de problema en el que se requiere hallar el camino más largo en un grafo acíclico es el problema 10000 del UVa Judge: *Longest Paths*. Simplemente tenemos que aplicar el algoritmo presentado anteriormente y obtenemos la respuesta al problema.

4. Apéndices

4.1. Problemas resueltos

En esta sección se recopilan los distintos códigos empleados a lo largo del trabajo:

4.1.1. 385 Acepta el reto: La ronda de la noche

```
#include <iostream>
#include <vector>
#include <queue>

using namespace std;

using vi = vector<int>;
using vc = vector<char>;
using vvc = vector<vc>;
using vb = vector<bool>;
using vvb = vector <vb>;

struct tEstado {
   int x, y;
   int longitud;
};

vi fil = { -1,0,1,0 };
vi col = { 0,1,0,-1 };

int BFS(int xini, int yini, int xfin, int yfin, vvb & marcaje) {
   int C = marcaje[0].size(), F = marcaje.size();

   queue<tEstado> cola;
   if (marcaje[yini][xini]) {
      cola.push({xini, yini, 0});
      marcaje[yini][xini] = false;
   }

   while (!cola.empty()) {
      tEstado estadoAct = cola.front(); cola.pop();
      if (estadoAct.x == xfin && estadoAct.y == yfin) return estadoAct.longitud;
      else {
         for (int k = 0; k < fil.size(); ++k) {
            if (estadoAct.x + col[k] >= 0 && estadoAct.x + col[k] < C && estadoAct.y + fil[k] >= 0
                &&
               estadoAct.y + fil[k] < F && marcaje[estadoAct.y + fil[k]][estadoAct.x + col[k]]) {
               cola.push({ estadoAct.x + col[k] , estadoAct.y + fil[k], estadoAct.longitud + 1 });
               marcaje[estadoAct.y + fil[k]][estadoAct.x + col[k]] = false;
            }
         }
      }
   }

   return -1;
}
```

```
int main() {
   std::ios_base::sync_with_stdio(false);
   char buffer[1000000];
   cout.rdbuf()->pubsetbuf(buffer, 1000000);
   int numCasos;
   cin >> numCasos;
   for (int z = 0; z < numCasos; ++z) {
      int c, f;
      cin >> c >> f;

      vvc patio(f, vc(c));
      //Leemos la entrada
      for (int i = 0; i < f; ++i) {
         for (int j = 0; j < c; ++j) {
            cin >> patio[i][j];
         }
      }

      //Construimos el marcaje
      int xini, yini, xfin, yfin;
      vvb marcaje(f, vb(c, true));
      for (int i = 0; i < f; ++i) {
         for (int j = 0; j < c; ++j) {
            if (patio[i][j] == 'E') { //Entrada
               xini = j; yini = i;
            }
            else if (patio[i][j] == 'P') { //Salida
               xfin = j; yfin = i;
            }
            //else if (patio[i][j] == '.') {} Si es una casilla libre no tenemos que marcar nada
            else if (patio[i][j] == '#') { //Muro
               marcaje[i][j] = false;
            }
            else { //Sensor
               for (int k = 0; k < fil.size(); ++k) {
                  for (int h = 0; h <= patio[i][j] - '0'; ++h) {
                     if (i + h * fil[k] >= 0 && i + h * fil[k] < f && j + h * col[k] >= 0 &&
                        j + h * col[k] < c && patio[i + h * fil[k]][j + h * col[k]] != '#') {
                        marcaje[i + h * fil[k]][j + h * col[k]] = false;
                     }
                     else break;
                  }
               }
            }
         }
      }

      //Ejecutamos el BFS
      int solucion = BFS(xini, yini, xfin, yfin, marcaje);

      if (solucion == -1) cout << "NO\n";
      else cout << solucion << '\n';
   }

   return 0;
}
```

4.1.2. 11624 UVa Judge: Fire

```
#include <iostream>
#include <vector>
#include <queue>

using namespace std;

using vb = vector<bool>;
using vvb = vector<vb>;

vector<int> fila = { -1,0,1,0 };
vector<int> columna = { 0,1,0,-1 };

struct tEst {
   int fila,
      columna,
      pasos,
      objeto; //0 - john, 1 - fuego
};

int busquedaEnAnchura(int f, int c, vvb & m, queue<tEst> & q) {
   //BFS
   int sol = -1;
   while (!q.empty()) {
      tEst estAct = q.front(); q.pop();
      //Si John ha llegado a una casilla de salida terminamos
      if (estAct.objeto == 0 && (estAct.fila == 0 || estAct.fila == f - 1 || estAct.columna == 0 ||
          estAct.columna == c - 1)) {
         sol = estAct.pasos;
         break;
      }
      for (int k = 0; k < 4; ++k) {
         if (estAct.fila + fila[k] >= 0 && estAct.fila + fila[k] < f && estAct.columna + columna[k]
             >= 0 && estAct.columna + columna[k] < c) {
            //Si el nodo no ha sido aun procesado
            if (m[estAct.fila + fila[k]][estAct.columna + columna[k]]) {
               //Introducimos el estado en la cola
               q.push({ estAct.fila + fila[k] , estAct.columna + columna[k] , estAct.pasos + 1,
                   estAct.objeto });
               //Marcamos que la casilla no est disponible
               m[estAct.fila + fila[k]][estAct.columna + columna[k]] = false;
            }
         }
      }
   }
   return sol;
}

int main() {
   int nc;
   cin >> nc;
   for (int z = 0; z < nc; ++z) {
      int f, c;
      cin >> f >> c;
      vvb m(f, vb(c, true));
```

```
      int xini, yini;
      queue<tEst> q;

      //Construimos la matriz directamente sobre el marcaje
      for (int i = 0; i < f; ++i) {
         for (int j = 0; j < c; ++j) {
            char aux;
            cin >> aux;
            if (aux == '#') m[i][j] = false;
            if (aux == 'J') {
               xini = j;
               yini = i;
            }
            if (aux == 'F') q.push({ i, j, 0,1 });
         }
      }

      q.push({ yini, xini, 0,0 });

      int sol = busquedaEnAnchura(f, c, m, q);

      //Si la solucion es -1 no hay forma de salir
      if (sol == -1) cout << "IMPOSSIBLE\n";
      //Si no tenemos la soluci en sol
      else cout << sol + 1 << '\n';
   }
   return 0;
}
```

4.1.3. 1112 UVa Judge: Mice and Maze

```
#include <iostream>
#include <vector>
#include <queue>
#include <functional>
#include <climits>

using namespace std;

using vi = vector<int>;

using ii = pair<int, int>;
using vii = vector<ii>;
using vvii = vector<vii>;
#define mp(a,b) make_pair(a,b)

vi dijkstra(vvii const& listaAdy, int nodoIni) {
   vi dist(listaAdy.size(), INT_MAX);

   priority_queue<ii, vii, greater<ii>> pq; //Queremos que salgan primero los nodos con menor
       coste,
                              //por lo que tenemos que invertir el comparador

   pq.push(mp(0, nodoIni));
   dist[nodoIni] = 0;

   while (!pq.empty()) {
      ii estadoAct = pq.top(); pq.pop();
      int costeAct = estadoAct.first;
      int nodoAct = estadoAct.second;
      if (dist[nodoAct] < costeAct) continue;
      else {
         for (int i = 0; i < listaAdy[nodoAct].size(); ++i) {
            int costeArista = listaAdy[nodoAct][i].first;
            int nodoDestino = listaAdy[nodoAct][i].second;
            if (dist[nodoDestino] > costeAct + costeArista) {
               dist[nodoDestino] = costeAct + costeArista;
               pq.push(mp(costeAct + costeArista, nodoDestino));
            }
         }
      }
   }

   return dist;
}

void resolver(bool primerCaso) {
   int numVertices, casillaSalida, tiempoMaximo, numAristas;
   cin >> numVertices >> casillaSalida >> tiempoMaximo >> numAristas;
   --casillaSalida; //Las casillas en el problema son [1,N], y las queremos [0, N-1]

   vvii listaAdy(numVertices);

   for (int i = 0; i < numAristas; ++i) {
      int ini, fin, coste;
```

```
      cin >> ini >> fin >> coste;
      --ini; --fin;
      listaAdy[fin].push_back(mp(coste, ini)); //Metemos las aristas al rev para obtener G^R
   }

   vi distancias = dijkstra(listaAdy, casillaSalida);

   int numCasillasSalen = 0;
   for (int i = 0; i < numVertices; ++i) {
      if (distancias[i] <= tiempoMaximo) ++numCasillasSalen;
   }

   if (!primerCaso) cout << '\n';
   cout << numCasillasSalen << '\n';
}

int main() {
   int numCasos;
   cin >> numCasos;
   for (int nc = 0; nc < numCasos; ++nc) resolver((nc == 0) ? true : false);
   return 0;
}
```

4.1.4. 451 Acepta el reto: Huyendo de los zombis

```
#include <iostream>
#include <vector>
#include <queue>
#include <string>
#include <map>
#include <sstream>
#include <climits>
using namespace std;

typedef pair<int, int> ii;
typedef pair<int, ii> iii;
typedef vector<int> vi;
typedef vector<ii> vii;
typedef vector<iii> viii;

struct tEstadoListaAdy {
   int verticeDestino,
      tiempoTrayecto,
      minutoEnElQuePasaElBus;
};

struct tEstadoColaPrioridad {
   int paradaActual,
      minutoActual,
      tiempoEsperaAcumulado;
};

struct comp {
   bool operator() (tEstadoColaPrioridad const& p1, tEstadoColaPrioridad const& p2) {
      return p1.tiempoEsperaAcumulado > p2.tiempoEsperaAcumulado;
   }
};

void dijkstra(vector<vector<tEstadoListaAdy>> & adjList, vector<vector<int>> & sol) {
   vector<vector<int>> tiempoEsp(60, vector<int>(adjList.size(), INT_MAX));
   //Meter los primeros estados en el mejor minuto para cada la
   priority_queue<tEstadoColaPrioridad, vector<tEstadoColaPrioridad>, comp> pq;

   for (int j = 0; j < 60; j++) {
      tiempoEsp[j][0] = 0;
   }

   for (int j = 0; j < adjList[0].size(); j++) {
      tEstadoListaAdy v = adjList[0][j];
      tiempoEsp[(v.minutoEnElQuePasaElBus + v.tiempoTrayecto) % 60][v.verticeDestino] = 0;
      pq.push({ v.verticeDestino, (v.minutoEnElQuePasaElBus + v.tiempoTrayecto) % 60 , 0 });
   }


   while (!pq.empty()) {
      tEstadoColaPrioridad front = pq.top(); pq.pop();
      //int d = front.first, u = front.second;
      if (front.tiempoEsperaAcumulado > tiempoEsp[front.minutoActual][front.paradaActual]) continue;
      if (front.paradaActual == adjList.size() - 1) break;
```

```
      for (int j = 0; j < adjList[front.paradaActual].size(); j++) {
         tEstadoListaAdy v = adjList[front.paradaActual][j];
         //Sumamos +
         int tiempoDeEspera = (v.minutoEnElQuePasaElBus + 60 - front.minutoActual) % 60;
         if (tiempoEsp[front.minutoActual][front.paradaActual] + tiempoDeEspera <
             tiempoEsp[(v.minutoEnElQuePasaElBus + v.tiempoTrayecto) % 60][v.verticeDestino]) {
            tiempoEsp[(v.minutoEnElQuePasaElBus + v.tiempoTrayecto) % 60][v.verticeDestino] =
                tiempoEsp[front.minutoActual][front.paradaActual] + tiempoDeEspera;
            pq.push({ v.verticeDestino, (v.minutoEnElQuePasaElBus + v.tiempoTrayecto) % 60
                ,tiempoEsp[(v.minutoEnElQuePasaElBus + v.tiempoTrayecto) % 60][v.verticeDestino] });
         }
      }
   }
   sol = tiempoEsp;
}


bool resuelveCaso() {
   int nParadas, nLineasAutobus;

   stringstream ss;
   string lineaAux;
   cin >> nParadas;
   if (!cin) return false;
   vector<vector<tEstadoListaAdy>> listaAdy(nParadas);
   cin >> nLineasAutobus;
   std::getline(cin, lineaAux);
   for (int i = 0; i < nLineasAutobus; ++i) {
      std::getline(cin, lineaAux);
      stringstream ss(lineaAux);
      int paradaIni, parada1, parada2;
      int tiempo, tiempoAcum = 0;
      ss >> paradaIni;
      --paradaIni;
      parada1 = paradaIni;
      while (!ss.eof()) {
         ss >> tiempo >> parada2;
         --parada2;
         listaAdy[parada1].push_back({ parada2, tiempo, tiempoAcum });
         tiempoAcum += tiempo;
         //Reseteamos
         parada1 = parada2;
      }
      listaAdy[parada1].push_back({ paradaIni, 60 - tiempoAcum, tiempoAcum });
   }
   vector<vector<int>> sol;
   dijkstra(listaAdy, sol);
   //Recorremos los 60 mins para ver cual es el mejor de la ultima fila
   int mejorTiempo = INT_MAX;
   for (int i = 0; i < 60; ++i) {
      if (mejorTiempo > sol[i][listaAdy.size() - 1]) mejorTiempo = sol[i][listaAdy.size() - 1];
   }

   if (mejorTiempo == INT_MAX) cout << "Hoy no vuelvo\n";
   else cout << mejorTiempo << '\n';
   return true;
```

```
}

int main() {
   while (resuelveCaso()) {
   }
   return 0;
}
```

4.1.5. 558 UVa Judge: Wormholes

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

using vi = vector<int>;
using vb = vector<bool>;

using ii = pair<int, int>;
using vii = vector<ii>;
using vvii = vector<vii>;
const int INF = 1000 * 1000 * 1000;
#define mp(a,b) make_pair(a,b)

void relajar(vvii const & listaAdy, vi & dist) {
   for (int j = 0; j < listaAdy.size(); ++j) {
      for (int k = 0; k < listaAdy[j].size(); ++k) {
         int nodoOrigen = j;
         int nodoDestino = listaAdy[j][k].second;
         int costeArista = listaAdy[j][k].first;
         if (dist[nodoOrigen] != INF) {
            dist[nodoDestino] = min(dist[nodoDestino], dist[nodoOrigen] + costeArista);
         }
      }
   }
}

//True si hay un ciclo de coste negativo, false si no
bool bellmanFord(vvii const& listaAdy, int nodoIni, int numVertices) {
   vi dist(numVertices, INF);

   dist[nodoIni] = 0;

   //Realizar V - 1 veces
   for (int i = 0; i < numVertices - 1; ++i) {
      //Para cada uno de los vices  tomolo  como origen vemos si mejora la distancia a algn nodo
          con el que est conectado
      relajar(listaAdy, dist);
   }

   //Y lo hacemos otras V - 1 veces para detectar todos los nodos en ciclos de coste negativo
   bool ciclosCosteNegativo = false;
   for (int i = 0; i < numVertices; ++i) {
      for (int j = 0; j < listaAdy.size(); ++j) {
         for (int k = 0; k < listaAdy[j].size(); ++k) {
            int nodoOrigen = j;
            int nodoDestino = listaAdy[j][k].second;
            int costeArista = listaAdy[j][k].first;
            if (dist[nodoOrigen] != INF && dist[nodoDestino] > dist[nodoOrigen] + costeArista) {
               return true;
            }
         }
      }
   }
```

```
   return false;
}

void resuelveCaso() {
   int numVertices, numAristas;
   cin >> numVertices >> numAristas;

   vvii listaAdy(numVertices);
   //Construimos el grafo
   for (int i = 0; i < numAristas; ++i) {
      int ini, fin, coste;
      cin >> ini >> fin >> coste;
      listaAdy[ini].push_back(mp(coste, fin));
   }

   bool solucion = bellmanFord(listaAdy, 0, numVertices);

   if (solucion) cout << "possible\n";
   else cout << "not possible\n";
}

int main() {
   int numCasos;
   cin >> numCasos;
   for (int i = 0; i < numCasos; ++i) resuelveCaso();
   return 0;
}
```

4.1.6. 10449 UVa Judge: Traffic

```
#include <iostream>
#include <vector>
#include <climits>
#include <algorithm>
#include <cmath>
using namespace std;

using vi = vector<int>;
using vb = vector<bool>;

using ii = pair<int, int>;
using vii = vector<ii>;
using vvii = vector<vii>;
const int INF = 1000 * 1000 * 1000;
#define mp(a,b) make_pair(a,b)

void relajar(vvii const & listaAdy, vi & dist) {
   for (int j = 0; j < listaAdy.size(); ++j) {
      for (int k = 0; k < listaAdy[j].size(); ++k) {
         int nodoOrigen = j;
         int nodoDestino = listaAdy[j][k].second;
         int costeArista = listaAdy[j][k].first;
         if (dist[nodoOrigen] != INF) {
            dist[nodoDestino] = min(dist[nodoDestino], dist[nodoOrigen] + costeArista);
         }
      }
   }
}

void encontrarCiclosNeg(vvii const & listaAdy, vi & dist, vb & cicloNegativo) {
   for (int j = 0; j < listaAdy.size(); ++j) {
      for (int k = 0; k < listaAdy[j].size(); ++k) {
         int nodoOrigen = j;
         int nodoDestino = listaAdy[j][k].second;
         int costeArista = listaAdy[j][k].first;
         if (dist[nodoOrigen] != INF && dist[nodoDestino] > dist[nodoOrigen] + costeArista) {
            dist[nodoDestino] = dist[nodoOrigen] + costeArista;
            cicloNegativo[nodoDestino] = true;
         }
      }
   }
}

pair<vi, vb> bellmanFord(vvii const& listaAdy, int nodoIni, int numVertices) {
   vi dist(numVertices, INF);
   vb cicloNegativo(numVertices, false);

   dist[nodoIni] = 0;

   //Realizar V - 1 veces
   for (int i = 0; i < numVertices - 1; ++i) {
      //Para cada uno de los vices  tomolo  como origen vemos si mejora la distancia a algn nodo
          con el que est conectado
      relajar(listaAdy, dist);
```

```
    }

    //Y lo hacemos otras V - 1 veces para detectar todos los nodos en ciclos de coste negativo
    bool ciclosCosteNegativo = false;
    for (int i = 0; i < numVertices; ++i) {
        encontrarCiclosNeg(listaAdy, dist, cicloNegativo);
    }
    return mp(dist, cicloNegativo);
}

bool resuelveCaso(int & nc) {
    int numVertices;
    cin >> numVertices;
    if (!cin) return false;

    vi busyness(numVertices);
    for (int i = 0; i < numVertices; ++i) cin >> busyness[i];

    int numAristas;
    cin >> numAristas;

    vvii listaAdy(numVertices);
    //Construimos el grafo
    for (int i = 0; i < numAristas; ++i) {
        int ini, fin;
        cin >> ini >> fin;
        --ini; --fin;
        listaAdy[ini].push_back(mp(pow(busyness[fin] - busyness[ini], 3), fin));
    }

    cout << "Set #" << nc << '\n';
    int numConsultas;
    cin >> numConsultas;
    //Procesamos cada una de las consultas
    if (numVertices > 0 && numConsultas > 0) {
        auto solucion = bellmanFord(listaAdy, 0, numVertices);
        for (int i = 0; i < numConsultas; ++i) {
            int destino;
            cin >> destino;
            //Tenemos que sacar '?' si no se puede llegar al nodo o se puede llegar con coste menor a
                3, lo que es equivalente a que se pueda llegar al nodo con coste menor que 3 o que el
                nodo est en un ciclo de coste negativo (y por ende se pueda llegar con un coste tan
                peque como se quiera)
            if (solucion.first[destino - 1] < 3 || solucion.second[destino - 1] ||
                solucion.first[destino - 1] == INF) cout << "?\n";
            else cout << solucion.first[destino - 1] << '\n';
        }
    }
    ++nc;
    return true;
}

int main() {
    int casoAct = 1;
    while (resuelveCaso(casoAct)) {}
    return 0;
```

```
}
```

4.1.7. 821 UVa Judge: Page Hopping

```
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>
#include <iomanip>

using namespace std;
const int INF = 1000 * 1000 * 1000;

using vi = vector<int>;
using vvi = vector<vi>;

void fw(vvi & m) {
   for (int k = 0; k < 101; ++k) {
      for (int i = 0; i < 101; ++i) {
         for (int j = 0; j < 101; ++j) {
            m[i][j] = min(m[i][j], m[i][k] + m[k][j]);
         }
      }
   }
}

bool resuelveCaso(int & c) {
   int a, b;
   cin >> a >> b;
   if (a == 0 && b == 0) return false;

   vvi m(101, vi(101, INF));
   for (int i = 0; i < m.size(); ++i) {
      m[i][i] = 0;
   }
   while (a != 0 || b != 0) {
      m[a][b] = 1;
      cin >> a >> b;
   }

   fw(m);

   int nlinks = 0;
   double mdist = 0;
   for (int i = 0; i < 101; ++i) {
      for (int j = 0; j < 101; ++j) {
         if (m[i][j] != INF && m[i][j] != 0) {
            ++nlinks;
            mdist += m[i][j];
         }
      }
   }

   cout << "Case " << c << ": average length between pages = " << fixed << setprecision(3) <<
       mdist / nlinks << " clicks\n";
   ++c;
   return true;
}
```

```
int main() {
   int c = 1;
   while (resuelveCaso(c)) {}
   return 0;
}
```

4.1.8. 13286 UVa Judge: Ingredients

```
#include <iostream>
#include <vector>
#include <map>
#include <string>
#include <queue>

using namespace std;

using vi = vector<int>;
using ii = pair<int, int>;
using vii = vector<ii>;


#define EMPTY -5
#define INF 1000 * 1000* 1000
#define NINF (-1000) * 1000* 1000
#define mp(a,b) make_pair(a,b)

struct arista {
   int fin,
      coste,
      prestigio;
};

struct estado {
   int platoAct,
      costeAct,
      prestigioAct;
};

struct comp {
   bool operator()(estado const& e1, estado const& e2) {
      return e1.costeAct > e2.costeAct;
   }
};

//distancias, first - coste,, second - prestigio
/*
void dijkstra(vector<vector<arista>> const& listaAdy, vii & distancias) {
   priority_queue<estado, vector<estado>, comp> pq;
   pq.push({0,0,0});
   while (!pq.empty()) {
      estado front = pq.top(); pq.pop();
         if (front.costeAct < distancias[front.platoAct].first ||
            (front.costeAct == distancias[front.platoAct].first && front.prestigioAct >
                distancias[front.platoAct].second)) {
            distancias[front.platoAct].first = front.costeAct;
            distancias[front.platoAct].second = front.prestigioAct;
            for (int i = 0; i < listaAdy[front.platoAct].size(); ++i) {
               pq.push({listaAdy[front.platoAct][i].fin,
               front.costeAct + listaAdy[front.platoAct][i].coste,
               front.prestigioAct + listaAdy[front.platoAct][i] .prestigio});
         }
      }
```

```
   }
}
*/

vi kahn(vector<vector<arista>> const& listaAdy) {
   vi indegree(listaAdy.size(), 0);
   vi topo;

   for (int i = 0; i < listaAdy.size(); ++i) {
      for (int j = 0; j < listaAdy[i].size(); ++j) {
         ++indegree[listaAdy[i][j].fin];
      }
   }

   queue<int> nodos;
   for (int i = 0; i < listaAdy.size(); ++i) {
      if (indegree[i] == 0) nodos.push(i);
   }

   while (!nodos.empty()) {
      int nodoAct = nodos.front(); nodos.pop();
      topo.push_back(nodoAct);
      for (int i = 0; i < listaAdy[nodoAct].size(); ++i) {
         --indegree[listaAdy[nodoAct][i].fin];
         if (indegree[listaAdy[nodoAct][i].fin] == 0) nodos.push(listaAdy[nodoAct][i].fin);
      }
   }

   return topo;
}

void caminosMinimosDAG(vector<vector<arista>> const& listaAdy, vii & distancias) {
   vi ordenTopologico = kahn(listaAdy);
   distancias[0] = mp(0, 0);

   for (int i = 0; i < ordenTopologico.size(); ++i) {
      int nodoAct = ordenTopologico[i];
      if (distancias[nodoAct].first != INF) {
         for (int j = 0; j < listaAdy[nodoAct].size(); ++j) {
            if (distancias[nodoAct].first + listaAdy[nodoAct][j].coste <
                distancias[listaAdy[nodoAct][j].fin].first ||
               (distancias[nodoAct].first + listaAdy[nodoAct][j].coste ==
                   distancias[listaAdy[nodoAct][j].fin].first &&
                distancias[nodoAct].second + listaAdy[nodoAct][j].prestigio >
                    distancias[listaAdy[nodoAct][j].fin].second)) {
               distancias[listaAdy[nodoAct][j].fin].first = distancias[nodoAct].first +
                   listaAdy[nodoAct][j].coste;
               distancias[listaAdy[nodoAct][j].fin].second = distancias[nodoAct].second +
                   listaAdy[nodoAct][j].prestigio;
            }
         }
      }
   }
}

bool resuelveCaso() {
```

```
int b, n;
cin >> b;
if (!cin) return false;
cin >> n;

map<string, int> biy;
vector<vector<arista>> listaAdy;

biy.insert(mp("AAAAAAAAAAAAAAAAAAAA", 0));
vector<bool> basicos(1, false);
int cont = 1;
vector<arista> auxA; listaAdy.push_back(auxA);

for (int i = 0; i < n; ++i) {
   string s1, s2, s3;

   int c, p;
   cin >> s1 >> s2 >> s3 >> c >> p;

   if (biy.find(s1) == biy.end()) {
      biy.insert(mp(s1, cont));

      ++cont;
      listaAdy.push_back(auxA);
      basicos.push_back(true);
   }

   if (biy.find(s2) == biy.end()) {
      biy.insert(mp(s2, cont));
      ++cont;

      listaAdy.push_back(auxA);
      basicos.push_back(true);
   }

   listaAdy[biy[s2]].push_back({ biy[s1],c,p });
   basicos[biy[s1]] = false;
}

for (int i = 0; i < basicos.size(); ++i) {
   if (basicos[i]) {
      listaAdy[0].push_back({ i, 0, 0 });
   }
}

vii costeyprest(cont, mp(INF, NINF)); //first - coste,, second - prestigio
caminosMinimosDAG(listaAdy, costeyprest);


//Una vez que tenemos almacenado el coste y prestigio de cada plato ejecutamos un knapsack
vi marcaje(b + 1, EMPTY);
marcaje[0] = 0;

for (int i = 0; i < costeyprest.size(); ++i) {
   for (int j = b; j >= 0; --j) {
```

```
        if (marcaje[j] != EMPTY && j + costeyprest[i].first <= b && marcaje[j +
            costeyprest[i].first] < marcaje[j] + costeyprest[i].second) {
           marcaje[j + costeyprest[i].first] = marcaje[j] + costeyprest[i].second;
        }
     }
  }

  //Y cogemos el mmo prestigio
  int maxprest = 0, mincoste = 0;
  for (int i = b; i >= 0; --i) {
     if (marcaje[i] >= maxprest) {
        maxprest = marcaje[i];
        mincoste = i;
     }
  }

  cout << maxprest << '\n' << mincoste << '\n';
  return true;
}

int main() {
  std::ios_base::sync_with_stdio(false);
  char buffer[1000000];
  cout.rdbuf()->pubsetbuf(buffer, 1000000);
  while (resuelveCaso()) {}
  return 0;
}
```

4.1.9. 10000 UVa Judge: Longest Paths

```
#include <iostream>
#include <vector>
#include <queue>
#include <climits>
#include <algorithm>
using namespace std;

using vi = vector<int>;

using ii = pair<int, int>;
using vii = vector<ii>;
using vvii = vector<vii>;

#define mp(a,b) make_pair(a,b)

//Halla el orden  topolo  del grafo
vi kahn(vvii const& listaAdy) {
   //En primer lugar calculamos el grado de entrada de cada vice
   vi indegree(listaAdy.size(),0);
   vi topo;

   for (int i = 0; i < listaAdy.size(); ++i) {
      for (int j = 0; j < listaAdy[i].size(); ++j) {
         ++indegree[listaAdy[i][j].second];
      }
   }

   queue<int> nodosCandidatos;
   //Aquellos que tengan indegree 0 ser los primeros en el orden topolo:
   for (int i = 0; i < indegree.size(); ++i) {
      if (indegree[i] == 0) {
         nodosCandidatos.push(i);
      }
   }

   while (!nodosCandidatos.empty()) {
      int nodoAct = nodosCandidatos.front();
      nodosCandidatos.pop();
      for (int j = 0; j < listaAdy[nodoAct].size(); ++j) {
         --indegree[listaAdy[nodoAct][j].second];
         if (indegree[listaAdy[nodoAct][j].second] == 0) {
            nodosCandidatos.push(listaAdy[nodoAct][j].second);
         }
      }
      topo.push_back(nodoAct);
   }
   return topo;
}

vi caminoMasLargo(vvii const& listaAdy, int nodoIni) {
   //En primer lugar calculamos un orden topolo del grafo
   vi ordenTopo = kahn(listaAdy);

   //Y ahora computamos las mmas distancias
```

```
   vi distancias(listaAdy.size(), INT_MIN);
   distancias[nodoIni] = 0;

   for (int i = 0; i < ordenTopo.size(); ++i) {
      int nodoAct = ordenTopo[i];
      if (distancias[nodoAct] != INT_MIN) {
         for (int j = 0; j < listaAdy[nodoAct].size(); ++j) {
            int nodoDestino = listaAdy[nodoAct][j].second;
            int costeArista = listaAdy[nodoAct][j].first;
            distancias[nodoDestino] = max(distancias[nodoDestino], distancias[nodoAct] +
                costeArista);
         }
      }
   }
   return distancias;
}

int main() {
   int numVertices;
   cin >> numVertices;
   int nc = 1;
   while (numVertices != 0) {
      int nodoIni;
      cin >> nodoIni;
      --nodoIni;

      vvii listaAdy(numVertices);
      int ini, fin;
      cin >> ini >> fin;
      while (ini != 0 || fin != 0) {
         --ini; --fin;
         listaAdy[ini].push_back(mp(1, fin));

         cin >> ini >> fin;
      }

      vi distanciaNodos = caminoMasLargo(listaAdy, nodoIni);

      int maxDist = -1;
      int nodoMaxDist = -1;
      for (int i = 0; i < distanciaNodos.size(); ++i) {
         if (distanciaNodos[i] != INT_MIN && distanciaNodos[i] > maxDist) {
            maxDist = distanciaNodos[i];
            nodoMaxDist = i;
         }
      }

      cout << "Case " << nc << ": The longest path from " << nodoIni + 1 << " has length " <<
          maxDist << ", finishing at " << nodoMaxDist + 1<< ".\n\n";

      cin >> numVertices;
      ++nc;
   }
   return 0;
}
```

Referencias

[1] *Longest path problem.* https://en.wikipedia.org/wiki/Longest$_p ath_p roblem$[*Accedido* : 07/04/2019].

[2] *The shortest path through a maze.* Harvard University Press, 1959.

[3] Richard Bellman. On a routing problem. *Quarterly of Applied Mathematics*, (16(1)), 1958.

[4] Ron Bernard. Transitivité et connexité. *C. R. Acad. Sci. Paris*, (249), 1959.

[5] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*, pages 651–654. The MIT Press, 2009.

[6] Robert W. Floyd. Algorithm 97: Shortest path. *Communications of the ACM*, (5 (6)), 1962.

[7] C. H. Papadimitriou S. Dasgupta and U. V. Vazirani. *Algorithms.* 2006.

[8] Felix Halim Steven Halim. *Competitive Programming 3.* 2013.

[9] Stephen Warshall. A theorem on boolean matrices. *Journal of the ACM*, (9 (1)), 1962.