DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

# Trabajo Práctico 2

**Algoritmos sobre grafos**

23 de marzo de 2023 — Algoritmos y Estructuras de Datos III


| Integrante | LU | Correo electrónico |
|---|---|---|
| Embon Eitan | 610/20 | eembon1@gmail.com |
| Fiorino Santiago | 516/20 | fiorinosanti@gmail.com |
| Halperin Matias | 1251/21 | matias.halperin@gmail.com |
| Valentini Nicolas | 86/21 | nicolasvalentini@hotmail.com |





**Facultad de Ciencias Exactas y Naturales**
Universidad de Buenos Aires
Ciudad Universitaria - Pabellón I
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autónoma de Buenos Aires - Argentina
Tel/Fax: (54 11) 4576-3359
http://exactas.uba.ar

# 1. DFS

## 1.1. Hedge Mazes

Este problema consta de una serie de laberintos, los cuales están compuestos por habitaciones conectadas con pasillos. Cada pasillo conecta un par de habitaciones, y puede ser recorrido en ambos sentidos. Luego, dado un laberinto y una serie de consultas especificando una habitación inicial y una final, debemos determinar si tienen exactamente un camino que no repite habitaciones.

## 1.2. Modelado

Dado el conjunto de habitaciones $R = \{r_1, ..., r_n\}$ y el de de pasillos $C_i = (r_s, r_t)$, $1 \leq s \leq t \leq n$. Para resolver este problema armamos un grafo $G = (R, C)$ (sin pesos ni direcciones). Es decir, cada habitación del laberinto estará representada como un nodo en $G$, mientras que cada pasillo será una arista que conecta las habitaciones correspondientes. Ahora, el problema consiste en determinar si existe exactamente un camino simple entre cada par de nodos.

## 1.3. Solución

El primer paso del algoritmo consiste en armar el grafo $G$ siguiendo el modelado. Una vez armado el grafo, definimos $P \subseteq E(G)$ como el subconjunto de aristas de $G$ que cumplen la condición de ser puente. Luego, creamos un nuevo grafo $G' = (V(G), P)$, una copia de $G$ pero solamente con las aristas puente. Ahora, a cada nodo de $G'$ le asignamos un nivel, cumpliendo la siguiente condición: $\text{nivel}(v) = \text{nivel}(w) \iff v$ y $w$ pertenecen a la misma componente conexa en $G'$. Finalmente, podemos afirmar que dos nodos de $V(G)$ tienen un único camino simple en $G$ sii los nodos tienen el mismo nivel (Se demostrará más adelante).

### 1.3.1. Complejidad

La complejidad del preprocesamiento del grafo está compuesta por:

1. Armar un grafo de $R$ nodos (cantidad de habitaciones), y $C$ aristas (cantidad de pasillos), costando $O(R + C)$.
2. Luego, para encontrar $P$, las aristas puentes de $G$, usamos un algoritmo proporcionado por la cátedra, el cual consiste en correr *DFS* dos veces sobre $G$. Por lo tanto, la complejidad de este paso resulta $O(2(R + C)) = O(R + C)$.
3. Como $G'$ tiene la misma cantidad de nodos que $G$ y a lo sumo la misma cantidad de aristas, generar este grafo también nos costaría en el peor caso $O(R + C)$.
4. Para diferenciar las componentes conexas de $G'$ y asignarles un nivel, nuevamente corremos *DFS*, obteniendo otra vez una complejidad de $O(R + C)$.

Entonces, la complejidad total del preprocesamiento es de $O(4(R + C)) = O(R + C)$. Con este preprocesamiento ya podemos determinar el nivel de cada nodo. Al guardar los niveles en un vector, podemos acceder a estos en $O(1)$. Para responder a una consulta simplemente tenemos que fijarnos si coinciden los niveles de la habitación inicial y la final. En conclusión, podemos responder cada consulta en $O(1)$.

## 1.4. Demostración

Sean $G$ un grafo, $P$ las aristas puentes de $G$, y $G' = (V(G), P)$. Para demostrar que el algoritmo propuesto es correcto, basta con demostrar el siguiente teorema:
*Teorema:* Dos nodos $v, w \in V(G)$ tienen un único camino simple en $G \iff v$ y $w$ pertenecen a la misma componente conexa en $G'$ ($\text{nivel}(v) = \text{nivel}(w)$)

$\Rightarrow$)

Supongamos que $v$ y $w$ tienen un único camino simple en $G$, el cual llamaremos $C$. Sacando una arista de $C$ estaremos desconectando los nodos $v$ y $w$, porque $C$ es el único camino que los une. Esto quiere decir que todas las aristas de $C$ son puente, ya que sacando cualquiera de ellas la cantidad de componentes conexas de $G$ aumenta. Si todas las aristas de $C$ son puente, todas las aristas pertenecen a $G'$, por lo tanto $v$ y $w$ estarán conectados por $C$ en $G'$, entonces pertenecerán a la misma componente conexa.

$\Leftarrow$)

Supongamos que $v$ y $w$ pertenecen a la misma componente conexa en $G'$. Esto implica que existe un camino simple entre $v$ y $w$ en $G'$. Como $G'$ contiene únicamente las aristas puente de $G$, entonces existe un camino simple entre $v$ y $w$ en $G$ formado únicamente por aristas puente, el cual llamaremos $C_1$. Ahora, supongamos que existe otro camino simple entre $v$ y $w$ en $G$, el cual llamaremos $C_2$. Siguiendo esas suposiciones, estaríamos en uno de los siguientes casos:

1. Los caminos son disjuntos: En este caso, existe un ciclo que consiste en ir desde $v$ a $w$ por uno de estos caminos, y volver por el otro. Todas las aristas de $C_1$ y de $C_2$ pertenecen a ese ciclo.

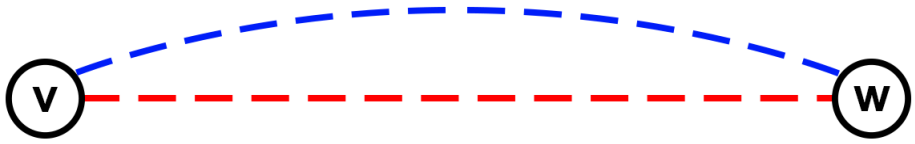


Figura 1: Caminos disjuntos

2. Empezando por el nodo $v$, los caminos coinciden hasta un nodo $x$: En este caso, existe un ciclo que consiste en ir desde $x$ a $w$ por uno de los caminos, y volver por el otro. Este ciclo contiene aristas de ambos caminos. El caso en el que empezando por el nodo $w$, los caminos coinciden hasta un nodo $x$ es análogo, el ciclo sería entre $v$ y $x$.

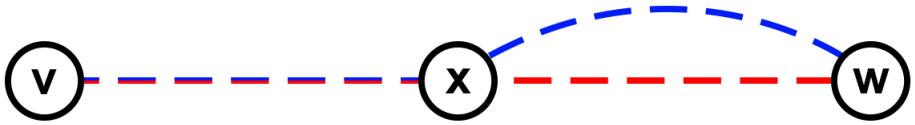


Figura 2: Caminos que coinciden hasta un nodo

3. Los caminos difieren entre un nodo $x$ y un nodo $y$: En este caso, existe un ciclo que consiste en ir desde $x$ a $y$ por un camino, y volver por el otro. Este ciclo contiene aristas de ambos caminos. Los caminos podrían tener más de una de estas discrepancias, en ese caso habría un ciclo por cada una de ellas. Notar que si permitimos que $x = v$, y que $y = w$, este caso cubriría los dos anteriores.


Figura 3: Caminos que difieren entre dos nodos

En los tres casos existen aristas de $C_1$ y de $C_2$ que pertenecen a un ciclo, lo cual es absurdo, ya que todas las aristas de $C_1$ son puente, y si una arista es puente, no puede pertenecer a un ciclo (Demostrado en el ejercicio 2 de la práctica 3). El absurdo surgió de suponer que había otro camino, por lo tanto, el camino es único.

# 2. Árbol generador mínimo

## 2.1. Tour Belt

Dado un conjunto de islas, se quiere seleccionar un subconjunto de ellas para conformar un *Tour Belt*. Para algunos pares de islas $(v, w)$ se calculó el *efecto de sinergia* $SE(v, w) = SE(w, v)$, que denota el valor turístico de tener ambas islas en el tour. Sea $G = (V, E)$ un grafo pesado, en el cual $V$ es el conjunto de islas a considerar, y donde cada *efecto de sinergia* $SE(v, w)$ calculado está representado como una arista $(v, w)$ (no dirigida) con peso $SE(v, w)$. Dado un subconjunto $B \subseteq V$ de islas, definimos como *ejes internos a la componente* $B$ a los ejes $(v, w)$ tales que $v, w \in B$, y definimos como *ejes del borde* a los $(v, w)$ tales que $v \in B$ y $w \notin B$ o viceversa.

Decimos que un subconjunto de islas $B$ es *candidato* a ser un *Tour Belt*, si $|B| \geq 2$, la componente es conexa, y todos los ejes internos de $B$ tienen peso mayor a los ejes del borde. El problema consiste en encontrar la sumatoria de los tamaños de los subconjuntos candidatos.

## 2.2. Modelado

Para resolver este problema, creamos un grafo pesado $G$ siguiendo el enunciado anterior. Cada isla estará representada como un nodo, y los *efectos de sinergia* indicarán los pesos de las aristas.

Sobre el grafo $G$, se correrá el algoritmo de *Kruskal* para buscar el árbol generador máximo. Si tomamos de base que cada nodo va a ser su propia componente en un inicio y durante la ejecución de *Kruskal* se van a ir uniendo, entonces a nosotros nos va a interesar estudiar las conexiones entre componentes. Para cada componente, hay 3 valores en particular que vamos a querer tener en cuenta: su arista interna de menor peso, y las aristas de peso mínimo y máximo que unen dicha componente con cada una de las otras componentes. De esta manera, si hubiésemos analizado eso para las aristas incidentes a cada uno de los nodos de una componente, tenemos entonces la arista mínima interna de la componente y la máxima del borde. Al compararlas, podemos ver si es efectivamente una componente candidata. La estructura principal que permite darle forma a esto es *Union Find* (UF a partir de ahora). Definamos a $n$ como la cantidad de islas, nodos en nuestro grafo. Para guardar los 3 valores mencionados anteriormente, vamos a usar un arreglo de $n$ posiciones en el que guardar la arista mínima de cada componente. Para guardar las conexiones mínimas y máximas entre componentes, usaremos una matriz de $n \times n$. Ambas estructuras no serán indexadas según el nodo, sino mediante la componente en la que la estructura UF nos indique que se encuentra, por lo que siempre vamos a estar accediendo a la información actualizada. De otra forma, deberíamos replicar la información para todos los nodos de la componente.

## 2.3. Solución

Nuestra solución consiste en, por cada arista que Kruskal selecciona, previo al join, guardar la información que nos interesa (explicada en Modelado) de ambos nodos que conecta. De esta manera, por cada candidata, ya vamos a haber recorrido todas sus adyacencias y por ende guardado la información de todas las aristas que involucra. Para esto nos interesa separar en 3 casos: aristas que unen dos nodos sueltos, aristas que unen un nodo suelto con una componente y aristas que unen dos componentes. Para el primero, la mínima de la componente va a ser la arista seleccionada (ya que es la única por ahora), y hay que recorrer las adyacencias de ambos nodos para guardar esas conexiones. En el segundo caso (nodo a componente), hay que recorrer las adyacencias del nodo suelto y evaluar cada arista. Si conecta con un nodo de la componente, hay que evaluar si es menor que la mínima de la componente. Si es externa, hay que evaluar si es una conexión mínima o una máxima con esa componente a la que conecta. Por ultimo, para el caso de unión de dos componentes, hay que unificar la información de ambas y evaluar la arista seleccionada como una posible mínima interna. Tras realizar el join, queremos evaluar si esa nueva componente es candidata, por lo que debemos registrar dos valores: la máxima de las aristas que conectan con otras componentes y la mínima arista interna. Si la primera es menor que la segunda, entonces todas las aristas internas son mayores que las externas y por ende esa componente es candidata. Finalmente, sumamos la cantidad de nodos a la solución y continuamos con la ejecución de *Kruskal*.

### 2.3.1. Complejidad

Kruskal, de base, tiene una complejidad $O(mlog(n) + n\alpha^{-1}(n))$. Previo a su ejecución vamos a calcular las listas de adyacencia, pero eso se realiza en $O(m)$ por lo que no suma complejidad. Dentro del algoritmo, nuestra modificación se realiza adentro del if que se ejecuta n veces (parte de la demostración de Kruskal), por lo que la complejidad que agreguemos va a multiplicarse con el n. Las operaciones agregadas que se realizan en $O(1)$ u $O(\alpha^{-1}(n))$ no nos interesan ya que no modifican la complejidad actual. Por fuera de eso, hay 2 tipos de for que realizamos. Unos que recorren todos los nodos, por ende $O(n)$. Y otros que recorren las listas de adyacencia, que podríamos decir que en el global tienen un costo amortizado de $O(m)$ pero como ya tenemos los otros ciclos $O(n)$, simplemente vamos a decir que estos también son $O(n)$. Teniendo esto en cuenta, las n veces en las que el if da true ahora tienen un costo $O(n\alpha^{-1}(n))$. Finalmente, entonces, la complejidad del algoritmo queda $O(mlog(n) + n^2\alpha^{-1}(n))$.

## 2.4. Demostración

Para demostrar la correctitud del algoritmo nos basamos en el siguiente lema:

*Lema:* Si se ejecuta *Kruskal* para buscar un árbol generador máximo, toda componente candidata va a ser, en alguna iteración, una de las componentes conexas del bosque que mantiene el algoritmo.

*Demostración:* Sea $C$ una componente candidata, entonces todas sus aristas internas son mayores que sus aristas borde. Corriendo el algoritmo de *Kruskal* para encontrar árboles generadores máximos, éste va a considerar las aristas por peso de mayor a menor. Esto quiere decir que va a seleccionar las aristas internas antes que las externas. Por lo tanto, esa componente va a aparecer en una iteración de *Kruskal*. Entonces, para toda componente candidata, podemos afirmar que ésta va a aparecer en alguna iteración de la ejecución del algoritmo de *Kruskal*.

Entonces, chequeando si una componente es candidata para toda componente conexa del bosque que mantiene el algoritmo en cada iteración, estaremos chequeando *todas* las componentes candidatas. Es decir, si corremos el algoritmo de *Kruskal* para buscar un árbol generador máximo, y cada vez que agregamos una arista al árbol chequeamos si la nueva componente que se genera es candidata, estaremos considerando a *todas* las componentes candidatas. La forma en la que chequeamos si una componente es candidata es fijándonos si la arista interna de menor peso es mayor a la arista saliente de mayor peso. Esta comparación la podemos hacer usando los algoritmos y estructuras mencionados anteriormente sin exceder la complejidad. Habiendo encontrado todas las componentes candidatas, sumando la cantidad de nodos de cada una de ellas encontramos la solución.

# 3. Camino mínimo

## 3.1. Usher

En este problema un sacerdote le pasa al portero una caja vacía, la cual puede contener hasta $c$ monedas. Luego, empezando por el portero, la caja se va pasando entre distintos feligreses, los cuales agregan monedas. Cada vez que la caja vuelve a pasar por el portero, este roba una moneda. El portero tiene un conjunto de feligreses a los que les puede pasar la caja, y cada feligrés tiene un conjunto de reglas que consisten en la cantidad de monedas que pone en la caja, y la persona (feligrés o portero) a la que se la pasa. Cuando la caja se llena con $c$ monedas, inmediatamente se devuelve al sacerdote y se deja de pasar. Debemos determinar la ganancia máxima que puede obtener el portero, asumiendo que podemos controlar el comportamiento de los feligreses.

## 3.2. Modelado

Para resolver este problema armamos un digrafo pesado que llamaremos $G$. Supongamos que tenemos $n$ feligreses, entonces $G$ tendrá $n+1$ nodos: uno representando cada feligrés, y uno representando al portero.

El portero tiene un conjunto de feligreses a los que les puede pasar la caja. Esto lo representaremos con una arista entre el nodo del portero y cada uno de los feligreses de dicho conjunto. A estas aristas les asignaremos un peso de valor 0. Tendría sentido que el valor de dicha arista sea $-1$, representando que el portero se roba una moneda, pero mantener las aristas positivas nos servirá luego para obtener una solución más eficiente.

Para el feligrés $i$ tenemos un conjunto de reglas. Cada regla consiste de una tupla de valores, la cual contiene la cantidad de monedas que pone en la caja, y la persona a la que se la pasa. Por cada regla $(p, j)$ del feligrés $i$, agregamos la arista $(i \to j)$ con $w((i \to j)) = p$, representando que el feligrés $i$ le puede pasar la caja al feligrés $j$, luego de agregar $p$ monedas.

## 3.3. Solución

Para facilitar la explicación de la solución, supongamos que el portero es el nodo 0. Una vez creado el digrafo pesado $G$, para encontrar la ganancia máxima que puede obtener el portero basta con encontrar el ciclo de menor peso que contenga al nodo 0. En la sección 3.4 demostraremos por qué esto lleva a encontrar una solución óptima.

Para encontrar el ciclo de menor peso que contenga al nodo 0, primero corremos el algoritmo de caminos mínimos de *Dijkstra* desde el nodo 0. Este algoritmo requiere que todas las aristas sean positivas, lo cual se cumple ya que las aristas salientes del portero pesan 0, y las aristas salientes de los feligreses son siempre positivas.

Luego, recorremos cada arista $(v, w)$ de $G$, y preguntamos si $w = 0$, es decir, si es una arista saliente de un feligrés hacia el portero. Si se cumple esa condicion, entonces encontramos un ciclo que consiste en ir desde el portero hasta $v$, y luego tomar la arista $(v, 0)$. El peso de dicho ciclo será $d(0, v) + w(v, 0) - 1$. La distancia de 0 a $v$ la obtuvimos corriendo *Dikjsta*, a eso le sumamos el peso de la arista $(v, 0)$, y le restamos uno ya que el portero se roba una moneda en cada vuelta.

Una vez ubicado el ciclo de menor peso, para obtener la ganancia máxima del portero solo debemos recorrer el ciclo, sumando uno cada vez que pasamos por el portero. Mientras recorremos el ciclo mantenemos un contador con el valor actual de la caja, y cuando este valor llega o supera $c$, cortamos el recorrido.

### 3.3.1. Complejidad

La complejidad del algoritmo está compuesta por:

1. Armar un grafo de $p+1$ nodos, siendo $p$ la cantidad de feligreses, y $r+f$ aristas, siendo $r$ la cantidad de reglas totales de los feligreses, y $f$ el cardinal del conjunto de feligreses a los que el portero les puede pasar la caja. El portero a lo sumo le podrá pasar la caja a todos los feligreses, por lo tanto $f$ será a lo sumo $p$. La complejidad del armado será $O(p + r + f) = O(p + r + p) = O(p + r)$.

2. Luego, correr el algoritmo de caminos minimos de Dijkstra desde el portero tendrá una complejidad de $O((r + f)\ log(p)) = O((r + p)\ log(p)) = O(r\ log(p))$. Cada feligrés tiene al menos una regla, por lo tanto $p \leq r$.

3. Finalmente, recorriendo las aristas podemos determinar si forman un ciclo con el portero y determinar la longitud de dicho ciclo en $O(1)$. Por lo tanto, este paso costará $O(r + f) = O(r + p)$

Entonces, la complejidad total del algoritmo queda en $O(p + r + r\ log(p) + r + p) = O(r\ log(p))$

### 3.4. Demostración

Para demostrar la correctitud del algoritmo decidimos probar dos teoremas por separado, que luego combinándolos llevarán a un argumento final que demuestra lo buscado.

Si en $G$ no existe un ciclo que pasa por el portero, entonces sin importar el comportamiento de los feligreses, el portero no va a ganar nada, ya que la caja nunca volverá a él. Entonces entre dos recorridos, si uno es un ciclo que pasa por el portero y el otro no, siempre se obtendrá mayor ganancia en el primero. Esto es lo que demostraremos en el siguiente teorema.

*Teorema:* Dados dos recorridos $R_1$ y $R_2$ dentro del digrafo pesado $G$, donde $R_1$ es un ciclo que contiene al portero y $R_2$ no, entonces la ganancia obtenida cuando la caja sigue el primer recorrido siempre va a ser mayor o igual a la ganancia obtenida cuando recorre el segundo.

*Demostración:* Debido a la forma del problema, la ganancia del portero aumenta en 1 unidad únicamente cuando la caja llega hacia él después de un recorrido entre los feligreses. Como la caja comienza en el portero, si buscamos que la ganancia aumente al menos una vez, buscamos que el recorrido vuelva al portero, teniendo así un ciclo que pasa por el portero. En cambio, si el recorrido no incluye un ciclo que pasa por el portero, este nunca va a poder incrementar su ganancia. Entonces, la ganancia del portero cuando el recorrido es un ciclo que pasa por él siempre será mayor o igual. El caso en el que la ganancia es igual (0) se da cuando el peso del ciclo es mayor a $c$, ya que la caja llegará a $c$ antes de que pueda volver al portero, obteniendo así una ganancia nula, la misma que si no contiene un ciclo.

Ahora, supongamos que tenemos dos ciclos que contienen al portero. Empezando desde el portero, por cada vuelta que se da en un ciclo, el portero roba una moneda. Entonces, para obtener una mayor ganancia, debemos encontrar el ciclo en el que se puedan dar mas vueltas. La cantidad de vueltas que se pueden dar en un ciclo está determinada por $c$. Una vez que la cantidad de monedas en la caja alcanza dicho valor, se termina el recorrido. Si mantenemos un contador de las monedas en la caja hasta el momento, en cada vuelta al ciclo se le sumará el peso del ciclo. Es evidente entonces que en un ciclo de mayor peso se podrán dar menos vueltas que en uno de menor peso, ya que el contador alcanzará el valor $c$ más rápido, cortando el recorrido más temprano. Entonces, la ganancia que obtendremos si nuestro recorrido es el ciclo de menor peso será mayor o igual a la ganancia que obtendremos si el recorrido es el ciclo de mayor peso. Esta idea es la que formalizaremos en el siguiente teorema.

*Teorema:* Dados dos ciclos $C_1$ y $C_2$ dentro del digrafo pesado $G$, donde ambos ciclos contienen al portero, y el peso de $C_1$ (la suma de sus aristas) es menor al peso de $C_2$. Entonces, la ganancia del portero cuando el recorrido es $C_1$ siempre va a ser mayor igual a la ganancia cuando el recorrido es $C_2$.

*Demostración:* La caja que pasa por los feligreses tiene una capacidad máxima $c$. Empezando desde el portero, por cada vuelta que se da en $C_1$, se le sumará el peso de dicho ciclo al valor de la caja. Análogamente pasa lo mismo con $C_2$ y su peso. Como el peso de $C_1$ es menor al de $C_2$, entonces siempre que demos una vuelta por $C_1$ vamos a tener un monto en la caja menor o igual que dando una vuelta por $C_2$. Entonces, $C_2$ llegará antes a $c$, es decir, la cantidad de vueltas que se podrán dar en $C_1$ serán mayores o iguales que las que se podrán dar en $C_2$. Empezando desde el portero, cada vuelta que se da en el ciclo, se incrementa en una unidad la ganancia del portero, por lo tanto, si en $C_1$ se pueden dar más vueltas, siempre obtendrá una mayor ganancia quedándose recorriendo $C_1$.

En conclusión, con el primer teorema demostramos que entre un recorrido que es un ciclo que pasa por el portero y otro que no, la ganancia con el primero será mayor o igual que con el segundo. Entonces, va a ser conveniente quedarse dentro de un ciclo. Con el segundo teorema, demostramos que entre dos ciclos de distinto peso, se obtiene una mayor ganancia cuando la caja se queda girando dentro del ciclo de menor peso. Con esto queda demostrado que lo más conveniente será elegir el ciclo de menor peso que pase por el portero como el recorrido de la caja.

# 4. SRD

## 4.1. Fishburn

En este ejercicio contamos con $n$ incógnitas $x_1 \dots x_n$ junto a un sistemas de $k$ inecuaciones de la forma $x_i - x_j \leq c$, y un conjunto $D$ con $m$ números enteros ordenados. El problema consiste en saber si existe una asignación de elementos del conjunto $D$ a las incógnitas de forma tal que se satisfagan todas las inecuaciones del sistema.

Sabemos que un sistema de inecuaciones cualquiera, sin restringir los valores posibles de las soluciones a un conjunto finito $D$, se puede resolver con el algoritmo de *Bellman-Ford*. Sin embargo, *Fishburn* [1] planteó un algoritmo que hace uso de esta nueva restricción para obtener una forma más eficiente de resolverlo. El método de *Fishburn* es el que vamos a implementar y discutir en las siguientes secciones.

## 4.2. Ideas principales del algoritmo

Supongamos que tenemos acceso a una matriz con todas las asignaciones posibles de elementos de $D$ a cada una de las incógnitas. El tamaño de esta matriz sería la cantidad de incógnitas por la cantidad de elementos en $D$, es decir $n \times m$. Ahora lo que se podría hacer es ir chequeando para qué valores de la matriz el sistema de inecuaciones se cumple. Una iteración de este algoritmo consistiría en realizar una asignación y comprobar si las $k$ inecuaciones del sistema se cumplen. Esto sería orden de $k$ por cada iteración. Sumado a esto, la cantidad de iteraciones que se realizan sería igual a la cantidad total de combinaciones de posibles asignaciones. Esto daría un orden de complejidad factorial muy elevado.

Quedándonos inconformes con la complejidad anterior, quisiéramos encontrar una forma de recorrer la matriz tal que pase a lo sumo una vez por cada posición. Es decir, que una vez que una incógnita $x$ con un valor asignado $d$ no cumpla con una ecuación, podamos asegurar que dicha asignación *nunca* va a llevar a una solución, pudiendo así descartarla. La complejidad obtenida así sería $n \times m$ posibles asignaciones. Por consiguiente, la complejidad total quedaría en $k$ por $n \times m$ iteraciones. El método propuesto por *Fishburn* cumple con esta idea y la justificaremos con más detalle en la sección 4.4.

## 4.3. Algoritmo

Nuestro algoritmo toma como input un vector $D$ de $m$ enteros ordenados, un vector de $k$ tuplas $(a, b, c)$, donde cada tupla representa la inecuación $x_a - x_b \leq c$. El output del algoritmo es un booleano que indica si se puede resolver el sistema.

Para resolver el problema, creamos un vector $S$ de tamaño $n$ (cantidad de incógnitas), el cual tendrá valores entre 1 y $m$ (tamaño del conjunto $D$). Los valores de este vector son "punteros" que indican, por cada incógnita, su valor representado como un índice en el conjunto $D$. Es decir, si $S_i$ tiene valor $j$, quiere decir que $x_i = D_j$. Cada valor de $S$ comienza con valor $m$, lo que equivale a que cada incógnita comience valiendo el valor más grande de $D$.

Ahora empezaremos a cambiar los valores de las incógnitas dentro de un ciclo, buscando que se cumplan todas las inecuaciones. Para esto, creamos un booleano que nos indica si el valor de alguna incógnita fue modificado en la iteración actual. El ciclo principal sigue iterando hasta que este booleano sea falso (cuando no cambió el valor de ninguna incógnita). Dentro de este ciclo principal, primero ponemos el booleano en *false*, y luego introducimos un ciclo que recorre todas las inecuaciones. Para cada inecuación $x_a - x_b \leq c$, si los valores actuales no la cumplen, entonces estamos en el caso $x_a > x_b + c$. Para solucionar este caso debemos disminuir el valor de $x_a$, entonces le vamos restando 1 al puntero $S_a$, hasta que se cumpla la inecuación. Como $D$ está ordenado, restarle uno al puntero hará que el valor de la incógnita sea menor. En el caso de que $S_a$ llegue a cero, significa que incluso cuando $x_a$ era igual al valor más chico de $D$ la inecuación no se cumplió, por lo tanto en ese caso no hay solución y retornamos *false*. En el caso de encontrar un valor de $x_a$ para el cual se cumple la inecuación simplemente ponemos el booleano en *true* y continuamos el ciclo.

Si el ciclo principal termina, significa que en su última iteración no se cambió el valor de ninguna incógnita, por lo tanto se cumplen todas las inecuaciones. Si una no se cumpliera, el algoritmo seguiría intentando cambiar el valor de su $x_a$ correspondiente, siguiendo así con sus iteraciones, o retornando *false* si no se puede cumplir. Por lo tanto, si el ciclo terminó significa que encontramos una solución, entonces retornamos *true*. Además, en el vector $S$ nos quedaron los punteros a los valores de cada incógnita que cumplen todas las inecuaciones.

[1] *Solving a system of difference constraints with variables restricted to a finite set*, John P. Fishburn (2001).

### 4.4. Justificación

Como dijimos en la sección 4.2, si una incógnita $x$ con un valor asignado $d$ no cumple con una inecuación, entonces sabemos que dicha asignación *nunca* va a llegar a una solución, con lo que podemos descartarla. Esta idea se ve reflejada en el algoritmo cuando vamos iterando en el sistema de inecuaciones de la siguiente forma: Para cada inecuación $x_j \leq c_{ij} + x_i$, estando fijos $x_i$ y $c_{ij}$, buscamos el elemento $d \in D$ más grande que asignándolo a $x_j$ cumpla la inecuación. Ahora, si encontráramos otra inecuación $x_j \leq c_{lj} + x_l$ tal que, estando fijados $c_{lj}$ y $x_l$, $x_j$ no cumple nuevamente con la ecuación, entonces podemos decir que podemos descartar la asignación $x_j = d$. De hecho podemos afirmar que si existe una asignación $x_j = d^*$ que sea parte de la solución, entonces necesariamente $d^* < d$. En este apartado justificaremos esa idea final.

Sea $x_j = d_j$ y $x_i = d_i$, con $d_j, d_i \in D$. Supongamos que nos encontramos con la ecuación $x_j \leq c_i + x_i$ que no se está cumpliendo. Estando $x_i$ y $c_i$ fijos, encontramos a $d \in D$ tal que $d \leq c_i + x_i$. Ahora, seguimos iterando y nos encontramos con otra ecuación $x_j \leq c_l + x_l$. Entonces estamos en la situación de tener que buscar otra asignación $d^* \in D$ para $x_j$. Pueden pasar los siguientes casos.

1. $d^* > d$. Absurdo. Dijimos que $d$ era el elemento más grande en $D$ que satisface la ecuación $x_j \leq c_i + x_i$.
2. $d^* < d$. Entonces encontramos un $d^* \in D$ que cumple la inecuación, con lo que descartamos la anterior.
3. No hay elemento en $D$ asignable a $d^*$ que cumpla con la inecuación, con lo cual podemos decir que el problema no tiene solución.

Como el primer ítem es un absurdo, entonces solo puede pasar que no haya solución al problema, o que si haya, en ese caso necesariamente $d^* < d$. Con esta conclusión podemos decir que es factible descartar todos los elementos mayores a $d^*$, sin necesidad de chequear en el futuro si cumplen con las inecuaciones o no.

En conclusión, el algoritmo en el peor caso, para cada variable $x$ tendría que pasar a lo sumo una vez por cada elemento del conjunto $D$. En cada iteración del ciclo cambia al menos el valor de una incógnita, por lo tanto sería equivalente a recorrer una matriz de $m \times n$. Además, dentro de cada iteración se recorren todas las inecuaciones, por lo tanto la complejidad total termina siendo $O(kmn)$.