Trabajo Práctico II: Algoritmos sobre grafos

Algoritmos y Estructuras de Datos III

Integrante	LU	Correo electrónico
Morales Pessacq, Joaquín	430/20	joacomp489@gmail.com
Prida, Fabrizio Gabriel	31/20	pridafabrizio@gmail.com
Santesteban, Martín	397/20	martin.p.santesteban@gmail.com
Muñoz, Thomas Nahuel	1821/21	thomm88@hotmail.com

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2610 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (++54 +11) 4576-3300
https://exactas.uba.ar

Contenidos

1. Ejercicio 1

1.1. Presentación del problema - UVa Hedge Mazes

La Reina de Nlogonia es fanática de los laberintos y sus arquitectos le construyen varios alrededor de su castillo. Cada laberinto está formado por R habitaciones conectadas por C pasillos. Cada día se le propone un desafío que consiste en ir de una habitación de inicio a una final por un camino que no repita habitaciones (*i.e,* un camino simple). Un desafío es *bueno* cuando, entre todos los recorridos que unen las habitaciones de inicio y fin, exactamente uno de ellos es simple. Queremos determinar, dado un laberinto y dados ciertos desafíos, si los mismos son buenos o no.

1.2. Resolución del problema

1.2.1. Modelo

Para resolver este problema, lo primero que vamos a hacer es abstraernos un poco del enunciado y quedarnos con lo que realmente nos interesa.
En particular, vamos a modelar el problema mediante un grafo $G = (V, E)$ tal que:

- $V = \{1, ..., R\}$ de forma tal que cada habitación es representada por un nodo.
- $E = \{(u, v) : u, v \in G \wedge \text{existe un pasillo que conecta las habitaciones } u, v\}$

Sobre este modelo, podemos pensar que el problema consiste en, dados dos nodos u, v, determinar si existe exactamente un camino simple que los una.

1.2.2. Ideas previas al algoritmo

¿Qué implica que exista un único camino simple entre dos nodos? De alguna manera, eso nos dice que no se puede tomar ninguna ruta alternativa. Para cualquier arista del camino, ocurre que no hay otra opción más que tomar dicha arista para llegar de un nodo al otro. Esta idea en un árbol DFS se traduce en que para ir de u a v sólo se pueden tomar *tree-edges*, y que en ningún momento nos cruzamos con un *back-edge* que nos permita "saltear" una parte del camino sin repetir nodos. Un poco más formalmente hablando, lo que estamos diciendo es que un camino simple entre u y v es único si y sólo si todas sus aristas son *puentes*.

Lema. *Sea $P = w_1, ..., w_k$ un camino simple entre dos nodos u, v de un grafo G. Entonces, P es el único camino simple entre u y v si y sólo si todas las aristas de P son puentes.*

Demostración. $\Longrightarrow$) Por contrarrecíproco. Supongamos que (w_i, w_{i+1}) no es puente. Consideremos un árbol DFS sobre G enraizado en u, y asumamos que $nivel(i)$ indica el nivel del nodo i en el árbol. Como P es un camino simple que parte de la raíz u, se cumple que $nivel(w_i) < nivel(w_{i+1})$. Como (w_i, w_{i+1}) no es puente, existe un *back-edge* (x, y) tal que $nivel(x) \leq nivel(w_i)$ y $nivel(y) \geq nivel(w_{i+1})$. En consecuencia, sea $Q = w_1 \overset{Q_1}{\rightsquigarrow} x \to y \overset{Q_2}{\rightsquigarrow} w_k$ tal que Q_1, Q_2 son los caminos simples de w_1 a x y de y a w_k respectivamente, compuestos únicamente por *tree-edges*. Podemos afirmar que Q_1 y Q_2 no repiten nodos ya que máx$\{nivel(w_1), nivel(x)\} <$ mín$\{nivel(y), nivel(w_k)\}$ y en consecuencia todos los nodos de Q_1 están en un nivel menor a los nodos de Q_2.
$\Longleftarrow$) Tomemos la arista (w_{k-1}, w_k). Consideremos G' el grafo que resulta de eliminar de G la arista en cuestión. Como la arista considerada es puente, sabemos que w_1 y w_k están en componentes conexas distintas para G'. Luego, si queremos ir de w_1 a w_k, seguro que deberemos usar la arista (w_{k-1}, w_k), cualquiera sea el camino que tomemos. Ahora consideremos (w_{k-2}, w_{k-1}). Con el mismo análisis, llegamos a que w_1 y w_{k-1} están en distintas componentes conexas en el grafo G' que resulta de eliminar la arista en cuestión de G, por lo que si queremos ir a w_{k-1} (lo cual es necesario para llegar a w_k), seguro que deberemos pasar por la arista (w_{k-2}, w_{k-1}). Si seguimos con la misma idea hasta llegar a w_2, tenemos que dado un camino simple cualquiera entre w_1 y w_k, seguro que dicho camino contiene las aristas $(w_1, w_2), (w_2, w_3), ..., (w_{k-1}, w_k)$.

Luego, si existiese otro camino distinto entre w_1 y w_k, seguro que repitiría nodos, y en consecuencia no sería simple. Por lo tanto, podemos afirmar que P es el único camino simple entre w_1 y w_k. □

A partir de este lema surge el puntapié inicial para un algoritmo que resuelva el problema.

1.2.3. Algoritmo

Supongamos que detectamos todos los puentes del grafo G. Consideremos ahora el subgrafo $P = (V_P, E_P)$ tal que $V_P = V$ y $E_P = \{(u, v) \in E : (u, v) \text{ es puente}\}$. Dados dos nodos cualesquiera, sabemos que van a estar en la misma componente conexa de P si y sólo si existe un camino simple entre ellos que utilice aristas puente. Y por el Lema anterior sabemos que si existe un camino simple que utiliza únicamente aristas puente, es único. Luego, si obtenemos las aristas puente de G, armamos el subgrafo P y sabemos a qué componente conexa pertenece cada nodo en P, podemos responder la consulta que plantea el problema. En consecuencia, el siguiente algoritmo resuelve el problema:

1. Detectar las aristas puente de G.
2. Armar el subgrafo P como aquel que tiene los mismos nodos que G pero únicamente las aristas puente.
3. Detectar las componentes conexas de P.
4. Para cualquier consulta que llegue, responder que hay un único camino simple entre los nodos, únicamente si ocurre que ambos pertenecen a la misma componente conexa.

Concretamente, para detectar las aristas puente de un grafo, lo que hacemos es ejecutar DFS e ir guardando para cada nodo el mínimo nivel del árbol al que puede acceder utilizando *back-edges* que se encuentren en su subárbol. Esto se puede hacer al recorrer la lista de adyacencia de un nodo u: si nos encontramos con un nodo v tal que está en un nivel superior y no es el padre de u entonces tenemos un *back-edge*, así que en caso de poder llegar a un nivel menor con dicha arista, actualizamos la información relacionada al nodo u. Si por el contrario v es descubierto por u, entonces esperamos que se termine de procesar el nodo v y, en caso de que el mínimo nivel al que pueda llegar v sea menor al de u, actualizamos el valor relacionado a u. Por otra parte, también podemos detectar componentes conexas de un grafo por medio de DFS. En nuestra implementación, tenemos la función `DFS` que fundamentalmente se encarga de recorrer los nodos y ejecutar `dfs_visit` por cada uno que todavía no haya sido descubierto. En consecuencia, cada vez que comienza a ejecutar `dfs_visit` desde `DFS` sucede que inicia la exploración de una nueva componente conexa. Luego, basta diferenciar los nodos en base a esto para saber a qué componente conexa pertenecen.

1.2.4. Demostración de correctitud

Para demostrar la correctitud del algoritmo basta ver que, dados dos nodos $u, v \in V$, existe un único camino simple entre ellos si y sólo si se encuentran en la misma componente conexa del grafo P definido previamente. Veamos por qué esto es cierto, basándonos fundamentalmente en el Lema demostrado anteriormente:

- Si ambos nodos se encuentran en la misma componente conexa, significa que existe un camino entre ellos que utiliza únicamente aristas que son puente. En consecuencia, por el Lema enunciado previamente, sabemos que existe un único camino simple entre ellos.
- Si se encuentran en componentes conexas distintas, se puede deber a dos razones. O bien originalmente se encontraban en distintas componentes conexas en el grafo G, o bien para ir de un nodo a otro se pasaba por una arista que no es puente, y por lo tanto ahora quedan en distintas componentes. En el primer caso, como no existe ningún camino entre u y v, en particular podemos afirmar que no existe ningún *camino simple* entre ellos. En el segundo caso, tenemos que existe un camino P tal que no todas sus aristas son puente y entonces por el Lema podemos afirmar que P no es único, y en consecuencia no existe exactamente un único camino simple.

1.2.5. Análisis de complejidad del algoritmo

En primer lugar, el costo de detectar las aristas puente es el costo de ejecutar DFS, puesto que la única diferencia con un DFS básico es que además estamos manteniendo un vector para indicar el mínimo nivel al que puede llegar cada nodo utilizando *back-edges* de su subárbol, a partir del cual se realizan comparaciones a medida que se recorre el árbol. Posteriormente, el costo de armar el subgrafo P compuesto únicamente por aristas puente es $O(n+m)$ puesto que estamos utilizando listas de adyacencias para representar los grafos. Por otra parte, para detectar las componentes conexas se tiene un costo que también es igual al de DFS, ya que lo único que hacemos es agregar un vector que indica para cada nodo a qué componente conexa pertenece. Como dicho vector se actualiza cada vez que se llama a `dfs_visit` desde `DFS` podemos afirmar que la complejidad no empeora respecto a un DFS sin esta variante.

Luego, sumando las complejidades de detectar puentes, armar el subgrafo P y detectar sus componentes conexas, tenemos que la complejidad de este procesamiento es $O(n+m)$. Como tenemos guardado en un vector a qué componente conexa pertenece cada nodo, podemos responder todas las consultas que lleguen en $O(1)$.

2. Ejercicio 2

2.1. Presentación del problema - UVa Tour Belt

Dado un conjunto de islas de un archipiélago, se desea conformar un Tour Belt. Para esto, contamos con los efectos de sinergia entre algunos pares de islas, que denota el valor turístico que tiene la presencia de ambas islas en el tour, este será denotado para el par de islas u, v como $SE(u,v)$. Un subconjunto de islas B es candidato a ser Tour Belt si todos los efectos de sinergia entre las islas que pertenecen a B son mayores a los efectos que tienen estas con las islas que no pertenecen al subconjunto, el conjunto es completamente conexo y además tiene dos o más vértices.

2.2. Resolución del problema

2.2.1. Modelo

Modelamos el problema utilizando un grafo pesado $G = (V, E)$, donde cada vértice corresponde con una isla y la arista $(u,v) \in E$ si y solo si el efecto de sinergia fue calculado para el par de islas u y v. Además, contamos con la función de peso $w : E \to Z$, que a cada arista (u,v) le atribuye un peso igual al efecto de sinergia calculado entre u y v. Mas formalmente, tenemos que:

- $V = \{1, ..., n\}$, donde cada isla es representada por un nodo.
- $E = \{(u,v) : u, v \in G \wedge \text{fue calculado el efecto de sinergia del } u, v\}$
- $w : E \to \mathbb{Z}$, $w((u,v)) = SE(u,v)$, $\forall (u,v) \in E$

2.2.2. Ideas generales

El esquema general gira en torno a la ejecución del algoritmo de Kruskal. Vamos agregando la i-ésima arista más pesada en la i-ésima iteración solo si ésta une conjuntos disjuntos. Cada vez que se arma un conjunto de vértices conexos nuevo, nos fijamos si es candidato, viendo si la arista de peso mínimo interna tiene peso mayor a la arista de mayor peso externa, y si es el caso le sumamos a la respuesta final el cardinal de la unión.

Para poder determinar la arista mínima interna y la máxima externa, usamos estructuras de datos auxiliares. Contamos con dos matrices que nos permitirán, dado dos conjuntos A y B, determinar el peso de la arista (u,v) de peso mínimo/máximo donde $(u \in A \wedge v \in B) \vee (u \in B \wedge v \in A)$. Adicionalmente, guardamos para cada conjunto disjunto el peso de la arista de peso mínimo interna.

Notar que usamos la versión máxima del algoritmo de Kruskal debido a que nos interesan los conjuntos cuyas aristas internas tienen peso mayor a las aristas externas, y de esta forma nos encargamos de ir armando

subconjuntos cuyas aristas internas son máximas, que pueden llegar a ser candidatos. Si usáramos la versión mínima en casi todas las iteraciones tendríamos subconjuntos que seguro no son candidatos.

2.2.3. Algoritmo

1. Inicialización de todas las estructuras de datos, $res = 0$.
2. Llenar la cola de prioridad Q con todas las aristas.
3. Para (u,v) en Q:
   - Si u y v pertenecen a conjuntos disjuntos A, B distintos:
     - Unión de A, B.
     - Actualización de estructuras de datos.
     - Si w(arista mínima interna de $A \cup B$) $>$ w(arista máxima externa de $A \cup B$) , $res = res + \#(A \cup B)$.
4. Retornar res.

2.2.4. Demostración de correctitud.

Para demostrar que el algoritmo devuelve la respuesta correcta cualquiera sea el grafo de entrada, nos interesa demostrar el siguiente teorema. **Teorema.** *Sea C una componente de G. C es candidata $\iff$ El algoritmo reconoce a C como una componente candidata.* Decir que el algoritmo reconoce a C como una componente candidata refiere a que al unir dos conjuntos durante la ejecución de Kruskal, si el peso de la arista mínima interna es mayor a la externa, entonces el algoritmo evalúa que se trata de una componente candidata.

Demostración. $\Rightarrow$) Para demostrar la ida, nos interesa ver que existe una iteración del algoritmo de Kruskal en la que la componente candidata C es una componente conexa del bosque que se mantiene. Nos interesa ver esto porque nuestro algoritmo sigue el mismo esquema que el de Kruskal. Adicionalmente, deseamos demostrar que los chequeos que hace nuestro algoritmo para ver si una componente conexa es candidata son suficientes.

1. Sea T_C el árbol generador máximo de la componente candidata C (que tendrá una única componente conexa porque C es completamente conexa), al agregar aristas válidas en orden decreciente durante la ejecución de Kruskal, sabemos que las aristas internas de C serán evaluadas antes que las externas debido a que los pesos internos son mayores a los externos. Como consecuencia se agregaran primero las aristas internas que no generan ciclos, que corresponden a las aristas en T_C. Podemos afirmar que durante la ejecución de Kruskal, en alguna iteración, T_C forma parte del bosque que se mantiene.

   Es importante notar que para conseguir el árbol generador máximo de G, en iteraciones posteriores se evaluará alguna arista saliente de T_C, formando una componente producto de unir a T_C con alguna otra.

   Como T_C y C forman la misma componente incluida en G, si se forma T_C como árbol del bosque durante la ejecución de Kruskal, podemos asegurar que la componente candidata C es en alguna iteración una componente conexa.

   Debido a que el algoritmo analiza si se formó una componente candidata cada vez que se tiene una nueva componente conexa, podemos asegurar que todas las componentes candidatas son evaluadas durante la ejecución del mismo.

2. Ahora nos resta ver si, al formar una componente conexa nueva con Kruskal, el chequeo que hace nuestro algoritmo para ver si se trata de una componente candidata es adecuado. Sea U la componente conexa, queremos ver si chequear únicamente que el peso de la arista mínima interna de U es mayor al de la arista máxima externa es suficiente para definir si U tiene dos o más vértices, es completamente conexo y además todas las aristas internas tienen peso mayor a todas las aristas externas:

- En primer instancia todos los vértices forman componentes conexas de un único elemento, como solo se evalúa si una componente es candidata al unir dos componentes anteriores agregando una arista válida, podemos asegurar que la suma de los vértices de las componentes siempre es dos o más. Por lo tanto, no hace falta chequearlo en el algoritmo.
- Por el invariante de Kruskal, sabemos que todas las componentes conexas del algoritmo son árboles y como consecuencia son grafos completamente conexos acíclicos. Al ser U una unión de dos componentes, seguro U es un árbol también y como consecuencia será completamente conexo.
- Determinar que todas las aristas internas tienen peso mayor a las externas es equivalente a ver si la arista interna de peso mínimo tiene peso mayor a la máxima externa. Como U es una unión de dos componentes conexas que llamaremos A y B, el peso de la arista de peso mínimo interna de U será igual al peso mínimo entre la arista mínima interna de A, la mínima interna de B, y la mínima que tienen en común. Más rigurosamente, sea ϕ el peso de la arista mínima interna de $U = A \cup B$, definimos la función recursiva:

$$\phi(U) = \begin{cases} \infty \text{ si } \#V(U) = 1 \\ \\ min\{\phi(A), \phi(B), min\{w((u,v))/(u,v) \in E(G) \wedge (u \in A \wedge v \in B) \vee (u \in B \wedge v \in A)\}\} \quad cc \end{cases}$$

  Adicionalmente, definimos la función ζ(U) para definir el peso de la arista máxima externa de $U = A \cup B$:

$$\zeta(U) = \begin{cases} 0 \text{ si } \#V(U) = 1 \\ \\ max\{max\{w((u,v))/u \in A \otimes v \in A\}, max\{w((u,v))/u \in B \otimes v \in B\}\} \text{ cc.} \quad cc \end{cases}$$

  Como todas las aristas internas tienen peso mayor o igual a la de peso mínimo interna, y todas las aristas externas tienen peso menor a la de peso máximo externa, por transitividad si $\phi(U) > \zeta(U)$, entonces todas las aristas internas tendrán peso mayor a las externas.

Finalmente, podemos asegurar que dada una componente conexa de Kruskal los chequeos que hacemos son suficientes para ver si esta es candidata.

Si todas las componentes candidatas son en alguna iteración del algoritmo de Kruskal una componente conexa, y cada vez que se forma una componente nueva hacemos el chequeo correspondiente, podemos asegurar que dado un grafo G el algoritmo reconoce todas las componentes candidatas del mismo. Más rigurosamente, si C es una componente candidata del grafo, entonces el algoritmo la evalúa correctamente.

$\Leftarrow$) Si el algoritmo reconoce a una componente C como candidata, entonces sabemos que la misma surge como unión de dos árboles del bosque que mantiene Kruskal, y que adicionalmente el peso de la arista mínima interna es mayor al peso de la arista máxima externa. Debido a esto, podemos afirmar que la componente es conexa, pues la unión de dos árboles resulta en una componente conexa. Por otra parte, también podemos asegurar que la componente tiene al menos dos nodos, ya que en cada paso de $Kruskal$ unimos conjuntos que tienen al menos un nodo cada uno.
Si la arista mínima interna $a \in C$ tiene peso mayor a la arista de mayor peso externa $b \in C$, por definición todas las aristas internas tienen peso mayor o igual a a y todas las externas tienen peso menor o igual a b, entonces podemos asegurar que si $w(a) > w(b)$ entonces todas las aristas internas tienen peso mayor a las externas.
Luego, si el algoritmo reconoce a una componente como candidata, podemos afirmar que efectivamente lo es. □

2.2.5. Análisis de complejidad

Inicializar las estructuras de datos tiene costo $O(n^2)$ debido a que se requieren de dos matrices para poder determinar las aristas de costos máximos y mínimos entre distintas componentes, junto a dos vectores de

tamaño n para mantener los representantes de las componentes conexas de Kruskal y para guardar las aristas de costo mínimo internas de cada componente. Lo siguiente que se hace es llenar la cola de prioridad usando la estructura `std::priority_queue` con las aristas del grafo, que es equivalente a hacer m inserciones con costo $O(log(m))$ en peor caso, dando como resultado un costo de $O(m \cdot log(m)) = O(m \cdot log(n^2) = O(m \cdot 2log(n)) = O(m \cdot log(n))$. Sabemos que al iterar sobre todas las aristas de la cola de prioridad solo se agregan las aristas que están en el árbol generador máximo de G. Por lo tanto, se agregan $n-1 aristas$ con complejidad $O(n)$ debido a que se requiere de la actualización de todas las estructuras de datos. Adicionalmente se evalúa en $O(n)$ si la última componente formada es candidata. Luego, las aristas que no pertenecen al árbol generador máximo tienen complejidad $O(1)$ y el costo del ciclo termina siendo $O((n-1) \cdot 2 \cdot O(n) + (m-n-1) \cdot O(1)) = O(n^2 + m + n) = O(n^2)$. Sumando los términos, tenemos una complejidad final igual a $O(n^2 + m \cdot log(n))$.

3. Ejercicio 3

3.1. Presentación del problema - UVa Usher

En una iglesia, después de misa, el sacerdote le pasa al portero una caja que puede contener hasta c monedas. El portero se la pasa a algún feligrés, y cada uno de estos agrega una cantidad de monedas para luego pasársela a otra persona. Cada vez que la caja pasa por las manos del portero, este roba una moneda. Cada persona tiene definido un comportamiento, que consiste en la cantidad de monedas que agrega y a quién se la pasa. Cuando la caja llega a tener c monedas, la misma se devuelve inmediatamente al sacerdote.

El problema consiste en determinar la ganancia máxima que puede obtener el portero.

3.2. Resolución del problema

3.2.1. Ideas fundamentales

El objetivo que tenemos es maximizar la ganancia del portero. La primer observación que hacemos es que esto es equivalente a maximizar la cantidad de veces que la caja pasa por las manos del mismo. Luego, quisiéramos que la caja de muchas "vueltas" (*i.e*, algún camino que salga del portero, pase por algunos feligreses y vuelva al portero) antes de llenarse, pues cada vez que se completa una vuelta se puede robar una moneda. Como sabemos que la caja puede circular mientras no esté llena, quisiéramos dar vueltas que minimicen la cantidad de monedas agregadas, para así maximizar la cantidad de veces que podemos pasar por el portero. Observemos también que, como los feligreses agregan al menos 2 monedas, seguro que al terminar una vuelta habrá monedas en la caja para ser robadas.

Intuitivamente, de todas las formas posibles que hay de que la caja salga del portero, pase por algunos feligreses y vuelva a él, a priori parecería que la que más lo favorece es aquella en la que se agrega la mínima cantidad de monedas, porque eso permitiría que se den la mayor cantidad de vueltas alrededor de él. A partir de esta idea vamos a desarrollar nuestro algoritmo.

3.2.2. Modelo

Sea c la capacidad de monedas de la caja y sea p la cantidad de feligreses. Para resolver el problema vamos a considerar el siguiente digrafo pesado $G = (V, E, w)$, donde:

- $V = 0, 1, ..., p, p+1$. Por un lado, los nodos $1 \leq i \leq p$ representan a cada uno de los p feligreses. Por otra parte, los nodos 0 y $p+1$ representan al portero. En particular, el nodo 0 se usará para representar cuando el portero *pasa* la caja, mientras que el nodo $p+1$ para representar cuando la *recibe*.
- $E =$
  $\{(u, v) : 1 \leq u, v \leq p \ \wedge$ el feligrés u le puede pasar la caja al feligrés $v\} \cup$
  $\{(0, u) : 1 \leq u \leq p \wedge$ el portero puede pasarle la caja al feligrés $u\} \cup$
  $\{(u, p+1) : 1 \leq u \leq p \ \wedge$ el feligrés u le puede pasar la caja al portero$\}$
- El peso de cada arista (u, v) se define como la cantidad de monedas que agrega un feligrés u antes de pasarle la caja a v. En caso de que $u = 0$ entonces el peso es 0 (pues el portero no agrega monedas). Más

formalmente, definimos la función de pesos $w : E \to \mathbb{R}$ tal que $w(u, v)$ vale 0 si $u = 0$, y en caso contrario vale la cantidad de monedas que agrega u antes de pasarle la caja a v.

3.2.3. Algoritmo

Bajo el modelo planteado y en base a las ideas mencionadas anteriormente, el algoritmo va a consistir en encontrar un camino mínimo que salga del portero, pase por algunos feligreses y vuelva a él. Esto, en nuestro modelo, consiste en encontrar el camino mínimo entre los nodos 0 y $p + 1$. En particular, podemos ejecutar el algoritmo de *Dijkstra* sobre G puesto que todos los pesos son positivos.

Una vez que tenemos el peso mínimo entre 0 y $p + 1$ (de ahora en más d_{min}) queremos saber: ¿cuántas vueltas de costo d_{min} podemos dar hasta llenar la caja sabiendo que al finalizar cada una el portero roba una moneda?. Al finalizar cada vuelta se agregan $d_{min} - 1$ monedas en la caja (pues una es robada), y mientras haya menos de $c - d_{min}$ monedas, el portero va a poder volver a robar. Una vez que la caja tenga al menos $c - d_{min}$ monedas, el portero no podrá volver a robar ya que al terminar la siguiente vuelta podemos afirmar que la caja irá a parar directamente al sacerdote. Luego, la cantidad de monedas robadas es la cantidad de veces que podemos sumar $d_{min} - 1$ para llegar al mínimo valor mayor o igual a $c - d_{min}$. Es decir, la respuesta final será $\lceil c - d_{min}/(d_{min} - 1) \rceil$.

Así, el algoritmo planteado para resolver el problema es el siguiente:

1. Ejecutar el algoritmo de *Dijkstra* desde el nodo 0.
2. A partir de los resultados de *Dijkstra*, quedarnos con la distancia mínima $d_{min} = d(0, p + 1)$, que representa la mínima cantidad de monedas que se pueden agregar en un recorrido que salga del portero, pase por algunos feligreses y vuelva a él.
3. Retornar $\lceil c - d_{min}/(d_{min} - 1) \rceil$.

3.2.4. Demostración de correctitud

Para que el algoritmo planteado sea correcto, debemos ver que se maximiza la cantidad de veces que pasamos por el portero. Bajo nuestro modelo, esto es equivalente a maximizar la cantidad de veces que pasamos por el nodo $p + 1$. Luego, queremos ver que tomando el camino mínimo entre 0 y $p + 1$ maximizamos la cantidad de veces que pasamos por $p + 1$, sujetos a la restricción de la capacidad de la caja c.

Observación. Potencialmente podríamos dar vueltas alrededor del portero mediante caminos distintos, es decir, podríamos empezar yendo por un camino de 0 a $p + 1$ con costo $d_1 < c$, luego podríamos tomar otro camino distinto al anterior con costo d_2 y así sucesivamente hasta llenar la caja. Sin embargo, si tomamos caminos distintos, seguro que una mejor alternativa será repetir siempre el camino de peso mínimo entre los elegidos. En otras palabras, sean $d_1 < d_2 < ... < d_k$ los pesos de los caminos utilizados, si consideramos el peso mínimo d_1 entre ellos, entonces la cantidad de veces que pasamos por el nodo $p + 1$ tomando siempre el camino de costo d_1 es mayor o igual a la cantidad que se tiene utilizando los distintos caminos de costo $d_1, ..., d_k$.

A partir de esta observación podemos considerar que siempre se repite el mismo camino de 0 a $p + 1$ hasta llenar la caja, y en particular vamos a querer ver que el problema se resuelve tomando el camino mínimo.

Teorema. *Si se toma siempre un camino de peso mínimo, se maximiza la cantidad de veces que se pasa por el nodo $p + 1$.*

Demostración. Supongamos que se toma siempre el camino mínimo, de peso d_{min}, pero la cantidad de veces que se pasa por el nodo $p + 1$ no es máxima. Luego, seguro que se toma otro camino para dar vueltas y así maximizar el valor de interés. Sea entonces d la longitud del camino entre 0 y $p + 1$ que se utiliza para dar vueltas y maximizar la cantidad de veces que se pasa por $p + 1$. Sea i el mínimo número de vuelta tal que, usando el camino mínimo, al finalizar la i-ésima vuelta se cumple que la caja contiene al menos $c - d_{min}$ monedas. Entonces, como mediante el camino de longitud d_{min} se pueden hacer a lo sumo i vueltas pero mediante el de longitud d se pueden hacer más, sigue que $i \cdot (d_{min} - 1) \geq c - d_{min} \ \wedge \ i \cdot (d - 1) < c - d$, y en consecuencia $i \cdot (d_{min} - 1) + d_{min} \geq c \ \wedge \ i \cdot (d - 1) + d < c$. Luego,

$$
\begin{array}{lll}
i \cdot (d-1) + d & < & i \cdot (d_{min} - 1) + d_{min} \quad \Rightarrow \\
d \cdot (i+1) - i & < & d_{min} \cdot (i+1) - i \quad \Rightarrow \\
d \cdot (i+1) & < & d_{min} \cdot (i+1) \quad \Rightarrow \\
d & < d_{min} &
\end{array}
$$

Y hemos llegado a un absurdo ya que d_{min} es la longitud mínima entre 0 y $p+1$, por lo que no puede existir un camino entre dichos nodos de longitud menor. Luego, si se toma el camino mínimo entonces podemos afirmar que se maximiza la cantidad de veces que se pasa por el nodo $p+1$.

□

3.2.5. Análisis de complejidad

La complejidad de nuestro algoritmo va a estar dada por la complejidad que tenga ejecutar *Dijkstra*. En nuestra implementación utilizamos la estructura `std::priority_queue` que, al estar implementada con un *heap*, nos permite asegurar que la complejidad del algoritmo resulta ser $O(m \cdot log(n))$, donde m denota la cantidad de aristas y n la cantidad de nodos. Aquí, $n = p+2$ y $m = r$, donde r es la cantidad total de reglas (pues tenemos una arista por cada posible comportamiento de cada feligrés). Luego, la complejidad final del algoritmo es $O(r \cdot log(p))$.

4. Ejercicio 4: sistemas de restricciones de diferencias

4.1. Presentación del problema

Consideremos un sistema de k inecuaciones sobre n variables $x_1, ..., x_n$, donde cada desigualdad es de la forma $x_j - x_i \leq c$. Dada una lista D de números enteros ordenada de menor a mayor, queremos saber si existe una asignación de las variables a valores de D tal que se cumplan las restricciones del sistema.

4.2. Resolución del problema

A continuación analizaremos el algoritmo de Fishburn [1] que resuelve el problema planteado.

4.2.1. Algoritmo

El algoritmo comienza inicializando las variables con el valor máximo de la lista D. Posteriormente, se van realizando pasadas por las inecuaciones, verificando si las mismas se cumplen o no. En caso de que una inecuación $x_j \leq x_i + c$ no se cumpla, se intenta actualizar el valor de x_j al máximo valor posible (menor al actual) que haga que se cumpla la restricción. (*i.e,* se busca máx$\{D_k \in D : D_k \leq x_i + c\}$). Este proceso se repite hasta que al realizar una pasada no ocurran cambios en los valores de las variables, y esto puede ser por una de dos razones:

- Ya se satisfacen todas las inecuaciones y en consecuencia no es necesario actualizar el valor de ninguna variable.
- Existen inecuaciones $x_j \leq x_i + c$ que no se satisfacen, pero no existe ningún valor perteneciente a D por el que se pueda actualizar x_j de forma tal que la restricción pase a cumplirse.

Por eso es que al finalizar las pasadas correspondientes se chequea si alguna inecuación no se cumple. En caso afirmativo, el algoritmo indica que el sistema no tiene solución. En otro caso, el algoritmo retorna para cada variable un índice dentro de la lista D que corresponde al valor que toma dicha variable para solucionar el sistema.

4.2.2. Implementación del algoritmo

Para implementar el algoritmo partimos de un vector que representa la lista D, y mantenemos otro vector `var_index` tal que en la i-ésima posición contiene un índice j indicando que $x_i = D[j]$. Comenzamos haciendo que cada variable tenga un índice apuntando al mayor elemento de D y vamos realizando las pasadas mencionadas por las ecuaciones, decrementando el valor de los índices de ser necesario.

4.2.3. Ideas fundamentales

A continuación demostramos con nuestras palabras la correctitud del algoritmo a partir de lo explicado por el autor, y exponemos las ideas fundamentales del mismo. En primer lugar, mencionamos un Lema importante para demostrar la correctitud.

Lema. Sea $S_1, ..., S_n$ una solución al sistema de restricciones tal que $S_i \in D$ para $i = 1...n$. Considerando $x_1, ..., x_n$ como los valores que el algoritmo va asignando a las variables, entonces se mantiene el siguiente invariante a lo largo de la ejecución: $S_1 \leq x_1, ..., S_n \leq x_n$.

Demostración. Este Lema es relativamente directo de ver: como las variables se inicializan con el valor máximo de D, el invariante comienza cumpliéndose. Por otra parte, el valor de una variable x_j va a cambiar al ver que una restricción $x_j \leq x_i + c$ no se cumple. Como asumimos que $S_i \leq x_i$, sigue que al momento de analizar la desigualdad seguro que el valor factible para la j-ésima variable será menor al actual (pues si $x_j > x_i + c$ entonces $x_j > S_i + c$). Luego, el algoritmo actualiza la variable al máximo valor en D que hace cumplir la restricción, por lo que podemos asegurar que el nuevo valor x_j cumple ser mayor o igual a S_j. □

En otras palabras, el Lema nos dice que los valores que van tomando las variables a lo largo de la ejecución del algoritmo siempre son mayores o iguales a aquellos de una solución factible. Y esto hace que ir decrementando las variables al máximo valor que haga que se cumpla cada inecuación sea una estrategia correcta para resolver el problema. Intuitivamente, si no se cumple $x_j \leq x_i + c$, como tenemos asegurado que $x_i \geq S_i$, a pesar de que quizás el valor de x_i no sea su valor final, seguro que el valor que deba tomar x_j será menor al actual. Ahora bien, entre todos los valores en D que haga que se cumpla la inecuación, la mejor alternativa es considerar el mayor de todos. Esto es así porque si le asignásemos a x_j un valor que no fuese el máximo entre los mencionados y luego pasamos a considerar otra desigualdad $x_k \leq x_j + c'$ tal que x_j se encuentra del lado derecho, podría ocurrir que la misma no tenga solución, puesto que x_j fue acotado en exceso. Es decir, al asignarle a x_j el máximo entre los valores que hagan cumplir la inecuación $x_j \leq x_i + c$, estamos procurando "dejar el mayor espacio posible" en las demás ecuaciones donde x_j aparece del lado derecho.

Mencionado este Lema, pasamos a ver el Teorema que demuestra la correctitud del algoritmo.

Teorema. Si existe una solución factible, entonces el algoritmo encuentra alguna solución factible y retorna *True*. Si no hay solución factible, retorna *False*.

Demostración. Supongamos que existe una solución factible $S_1, ..., S_n$ pero el algoritmo retorna *False*. Esto implicaría que alguna inecuación $x_j \leq x_i + c$ no se cumple. Luego, al momento de haber analizado dicha ecuación, tenemos que $S_i + c \leq x_i + c$ y además $S_j \leq S_i + c$. Pero en ese caso, sigue que $S_j \leq x_i + c$, por lo que S_j era un candidato a ser el valor por el cual actualizar x_j, y entonces no puede ser que no haya sido actualizado. Así, no puede ocurrir que el algoritmo haya devuelto *False*.

Ahora asumamos que no hay solución factible. Entonces llegará un momento en el que algunas inecuaciones se cumplirán y otras no. Para las que no se cumplan, no existirá ningún valor al cual actualizar las variables de forma tal que pasen a cumplirse. Es decir, llegará un momento en el que se realice una pasada y no haya cambios en los valores de las variables. Como eso ocurre habiendo inecuaciones que no se cumplen, el algoritmo retornará *False*.

□

La demostración del Teorema hace notar fundamentalmente que, sabiendo que los valores que toman las variables siempre son mayor o iguales a los valores de una solución factible, no puede ocurrir que el algoritmo retorne una respuesta errónea cuando en realidad sí hay solución. Por otra parte, en caso de que no exista

solución factible, seguro que al salir del ciclo no se cumplirá alguna ecuación y, por lo tanto, se retorna *True*. Además, podemos asegurar que el programa sale del ciclo, puesto que, en el peor caso, se decrementan todas las variables hasta llegar a tomar el mínimo valor de la lista D, y a partir de ese momento no habrán más cambios, por lo que a lo sumo se termina de hacer pasadas por las ecuaciones al llegar ese momento.

4.2.4. Análisis de la complejidad

Como ya mencionamos, el algoritmo se encarga de hacer pasadas por las k ecuaciones. La cantidad de iteraciones depende de cómo se decrementen las variables. Por ejemplo, podría suceder que al realizar la primer iteración cada variable se decremente considerablemente, pasando a valer el mínimo valor de D. En ese caso, como mucho, se hará una pasada más, puesto que no hay más valores de D para seguir recorriendo. Es decir, no sólo que vamos haciendo pasadas por las ecuaciones, sino que también vamos recorriendo la lista D por cada variable, *pero sólo una vez*. Luego, la complejidad va a depender fundamentalmente de cuántas pasadas se realicen (*i.e,* cuántas iteraciones realice el ciclo *do-while*). En el peor caso, por cada pasada se decrementa una única variable, pasando a tomar el valor que se encuentra una posición a la izquierda del que tenía previamente en el conjunto D. En consecuencia, a lo sumo se realizan $O(n \cdot m)$ iteraciones. Como cada una de ellas cuesta $O(k)$ por recorrer todas las ecuaciones, tenemos que la complejidad final del algoritmo es $O(m \cdot n \cdot k)$.

Referencias

[1] John P. Fishburn. «Solving a system of difference constraints with variables restricted to a finite set». En: *Information Processing Letters* ().