DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico 2

Implementacion de Algoritmos en Grafos

8 de junio de 2022 — Algoritmos y Estructuras de Datos III

Integrante	LU	Correo electrónico
Schwartzmann, Alejandro	390/20	a.schwartzmann@hotmail.com
Bitesnik, Iván	446/20	ivanbitesnik@gmail.com
Casco, Rocío Diana	512/20	rociodcasco@gmail.com
Dallegri, Pablo	445/20	pdallegri@dc.uba.ar
Memoli Buffa, Pedro	376/20	pedromemoli@hotmail.com

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

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

Índice

Cantidad de Caminos Mínimos en un Grafo no Pesado

Uno de los objetivos de este trabajo es verificar si, dado un grafo conexo y no pesado, la cantidad de caminos mínimos que hay dentro de ese grafo cumple que para todo par de vértices existe un único camino de longitud mínima entre ellos.

Definimos, un grafo geodésico como un grafo, no pesado, donde para todo par de vértices pertenecientes a él, existe un único camino de longitud mínima entre ellos. Denotamos la longitud para grafos no pesados como la suma de las aristas que hay que recorrer para llegar desde fuente a destino, es decir, sea $G = (V, E), \forall (s, t) \in V(G)$, $\exists! P_{s,t}$, donde el Path desde s hasta t es de longitud mínima.

Determinar si un grafo es geodésico

Dado un grafo conexo, determinar si es o no geodésico implica contabilizar la cantidad de caminos mínimos que existe para todo par $(s, t) \in V(G)$ y, en caso de haber uno por cada par, retornar que es geodésico. A partir de este concepto, la intuición nos lleva a pensar en el algoritmo BFS, donde yo inicio la exploración de mi grafo no pesado desde una fuente, la cual inicia en un nodo cualquiera, que llamaremos fuente e inspeccionamos los "niveles"del grafo. El nivel 0 será compuesto por el vértice fuente. El nivel 1 serán los vecinos del nodo fuente, el nivel 2 los vecinos de los vecinos y así sucesivamente. Con esta noción, a partir del nivel en el que se encuentra un vértice, yo se la distancia desde la fuente. Ahora BFS no encuentra la distancia de todos los paths posibles, sino que me da los paths mínimos desde un único vértice fuente. Por lo tanto, necesitamos utilizar BFS pero agregando dos pequeños detalles al algoritmo. Necesitamos que cuente la cantidad de caminos que hay desde una fuente s hasta v, $\forall v \in V(G) - s$ dentro del grafo, vértices que siempre sabemos que son alcanzables porque el grafo dado es conexo. Y además que cuente la distancia $d(s, v)$ como la definimos previamente, el número de aristas que tengo que saltar hasta llegar a v.

Implementación del algoritmo BFS* en $\mathcal{O}(nm)$

La implementación de nuestra solución consiste en aplicar una leve vuelta de tuerca al algoritmo BFS. Cuando lo iniciamos vamos a tener que modificar parents, donde será una lista que permita almacenar múltiples parents por posición de la lista, ya que un vértice en caso de tener más de un camino mínimo tendrá más de un parent para llegar a hasta él desde la fuente. Y necesito agregar dos estructuras de tamaño n, distancias y paths, donde en la primera almacenamos la distancia desde el vértice fuente s hasta un vértice v cualquiera y, en paths, la cantidad de caminos mínimos que encontramos desde la fuente hasta v, ambas estructuras inicializadas en ∞ en todas sus posiciones, excepto en el vértice utilizado como fuente, que será 0 para la distancia y paths, y vacío para el parent.

Usando esta idea, empezamos a explorar el grafo conexo dado. Defino a v como el nodo actual que estoy explorando su vecindario, por lo tanto, el que acabamos de descolar, y a w como uno de los vértices que pertenece a la lista de adyacencia dé v. Cuando lleguemos a w van a ocurrir 3 casos

- Si w nunca había sido visitado, BFS* actúa normalmente, lo marco como visitado y asigno a v como su parent.
- Si w fue visitado, vemos su distancia hasta la fuente en `distancias[w]`. Si `distancias[w]` es menor a `distancias[v]+1`, entonces tengo un nuevo camino mínimo desde fuente a w, minimizo la distancia y le asigno a w la cantidad de caminos mínimos que pasaban por v.
- Si w fue visitado, y su distancia desde la fuente, distancia[w], es igual a distancia[v]+1, agrego a v como parent de w y debo sumarle los paths de v a w, porque los caminos mínimos de v compartirán la misma distancia que los caminos mínimos de w luego de añadir la arista $v \to w$. Y la distancia desde el nodo fuente a w sigue igual.

Una vez que termina esta secuencia de pasos habremos encontrado la distancia para todo vértice desde una fuente dada. Por lo tanto, este BFS* necesita ser ejecutado n veces y, en cada ejecución, utiliza un vértice distinto de fuente, para así obtener todas las distancias mínimas del grafo.

Para asegurar que nuestro algoritmo cumple con la complejidad solo hace falta ver los costos de esta implementación. Notemos que el costo de ejecutar una vez BFS es de $\mathcal{O}(n+m)$ pero como nuestro grafo es conexo, sabemos que $m \geq n-1$, por ende, la cantidad de aristas de nuestro grafo domina en el cálculo de complejidad.

Correr n veces BFS nos costará $\mathcal{O}(n*(n+m)) = \mathcal{O}(n^2+nm)$ que esta incluido en $\mathcal{O}(nm)$, porque m domina a n.

Si para $(s,t) \in V(G)$ existe un único camino mínimo, el algoritmo imprimirá la distancias de todos a todos y, en caso de no ser un grafo geodésico, imprimirá dos líneas con dos paths mínimos desde un s a t.

Testing

Para testear este algoritmo generamos dos tipos de entradas a testear:

- Input donde el grafo descripto es un árbol, el cúal como sabemos para cada par de vértices existe un único camino mínimo que los conecta.
- Input donde el grafo descripto es un árbol, pero editamos el input para agregar al menos un camino mínimo que una por lo menos un par de vértices.

La siguiente tabla representa los tiempos de ejecución en un procesador Ryzen 5 5600X, pero nos abstuvimos de imprimir la salida, ya que en varios test cases M tiene valores mayores a 100:

Test	**Tiempo (en segundos)**	**Geodésico**
tree-100-true	00.02	True
tree-100-false	00.01	False
tree-500-true	02.32	True
tree-500-false	00.03	False
tree-1000-true	18.56	True
tree-1000-false	00.10	False
tree-2000-true	147.6	True
tree-2000-false	00.32	False

Algoritmo de Johnson

Otro de los problemas que debemos encarar en este trabajo es como calcular de manera más eficiente el camino mínimo para todo par de vértices dado un digrafo conexo y pesado. Claramente, tenemos una solución para este problema, podemos utilizar el algoritmo Floyd-Warshall que en $\mathcal{O}(n^3)$ nos retorna una matriz donde $M[i][j]$ representa la distancia desde el i-ésimo vértice hasta el j-ésimo vértice. El problema que surge en este algoritmo es que la cota de complejidad es muy cara para algoritmos de poca cantidad de aristas, donde $m < n$ o, mejor dicho, cuando el grafo es ralo, ya que la cantidad de aristas no incide en la complejidad del algoritmo.

En cambio, a partir del paper leído, dado un digrafo pesado y conexo, para este algoritmo la complejidad ($\mathcal{O}(n * m + n * m * log(n))$) si depende de la cantidad de aristas, ya que ejecuta una única vez Bellman-Ford, $\mathcal{O}(nm)$ y modifica el grafo para permitir la ejecución de Dijkstra, $\mathcal{O}(m * log(n))$.

Idea detrás del algoritmo

El algoritmo de Johnson utiliza como idea principal la técnica "*reweighting*", la cual consiste básicamente en, sí existen aristas de peso negativo, tenemos que computar un nuevo peso para cada arista tal que permita correr Dijkstra en el grafo. Esta nueva función de peso la denotamos como $\bar{w}$

Si nosotros generamos un nuevo conjunto de vértices partiendo de aristas que tenían ciclos negativos, este nuevo conjunto debe cumplir dos propiedades:

- $\forall (s,t) \in G(V)$, un camino p es mínimo usando una función de pesos $w : e \to R$ desde s hasta $t \iff p$ es mínimo desde s hasta t usando la función de peso $\bar{w} : e \to R$
- $\forall (s,t) \in G(V)$, la nueva función de pesos cumple que $\bar{w}((s,t)) \geq 0$

Para satisfacer la primera propiedad utilizamos el siguiente teorema:

Teorema 1. *Repesar las aristas no modifica caminos mínimos.*
Dado un grafo pesado $G = (V, E)$, con una función de pesos $w : E \to R$, sea $h : V \to R$ una función que asigne números reales a los vértices de un grafo, $\forall (s,t) \in E(G)$, definimos una nueva función de pesos:
$\bar{w}(s,t) = w(s,t) + h(u) - h(v)$.
Sea $P = \{v_0, v_1, ..., v_k\}$ sea cualquier camino entre v_0 y v_k, luego P es el camino mínimo entre ellos con una función de peso $w \iff P$ es un camino mínimo con una función de peso $\bar{w}$. Esto es, sí $w(P) = \delta(v_0, v_k) \iff \bar{w}(P) = \bar{\delta}(v_0, v_k)$. Más aun, G tiene un ciclo negativo con una función de peso $w \iff G$ tiene un ciclo negativo con una función de peso $\bar{w}$

Con este teorema podemos afirmar que si encontramos un camino mínimo a partir de repesar las aristas, entonces es un camino mínimo. Pero esto no basta para que el algoritmo sea correcto, ya que para poder implementar Dijkstra necesitamos también evidencia que si repesamos los pesos, estos deben ser no negativos. Debido a esto surge el siguiente lema:

Lema 1. *Producir pesos no negativos repesando las aristas*
Dado un grafo pesado $G = (V, E)$ con una función de pesos $w : E \to R$, creamos un nuevo grafo $G' = (V', E')$, donde $V' = V \cup \{s\}$, con s un vértice tal que $s \notin V$ y $E' = E \cup \{(s, v) : v \in V\}$ y extendemos la función de peso w para que $w(s, v) = 0, \forall v \in V$ y como no existe ninguna arista donde s sea la cabeza de susodicha arista, porque todas las aristas nuevas de E' tienen siempre a s como cola, ningún camino mínimo de G' puede contener a s, excepto los que tienen a s como primer vértice. Más aún G' tiene ciclos negativos, $\iff$ G tiene ciclos negativos. Ya que no podemos generar nuevos ciclos negativos con las aristas nuevas porque en un ciclo q, $v_0 = v_k$ y, por lo tanto, con la nueva definición de peso: $\bar{w}(q) = w(q) + h(v_0) - h(v_k) = w(q)$.

Suponiendo que G no tiene ciclos negativos y, por lo tanto, G' tampoco, definimos $h(v) = \delta(s, v) \forall v \in V'$. Por la desigualdad triangular tenemos que $h(v) \leq h(u) + w(u, v), \forall (u, v) \in E'$

Entonces, al definir los nuevos pesos $\bar{w}$ a partir de la definición $\bar{w}$(s,t) = w(s,t) + h(u) - h(v), tenemos que:
$\bar{w}(s,t) = w(s,t) + h(u) - h(v) \geq 0$.

Entonces ahora, si podemos afirmar que Johnson corrige correctamente los pesos de tal manera que permite la ejecución de Dijkstra en grafos con aristas de pesos negativos.

Implementación de Johnson en complejidad temporal $\mathcal{O}(nm * log(n))$ y $\mathcal{O}(m)$ espacial

Nuestra implementación de Johnson recibe un digrafo pesado, y en costo temporal y espacial $\mathcal{O}(m)$ construye la instancia a partir de las aristas, ya que el grafo, al ser conexo como en el primer punto de este trabajo, la cantidad de aristas domina a los vértices. Por lo tanto, al cargar el digrafo en memoria, la lista de adyacencias cuesta $\mathcal{O}(m)$ que es mayor o igual a la cantidad de vértices.

A partir de este digrafo, le agrego un vértice s que actúa como fuente, del cual salen n aristas con cabeza igual v, para todo vértice v distinto a s en el digrafo. El costo de agregar n aristas tanto espacial como temporalmente está acotado por $\mathcal{O}(m)$. Y luego ejecuta Bellman-Ford para intentar detectar ciclos negativos en el digrafo. Este algoritmo está acotado por $\mathcal{O}(nm)$ el costo espacial de vuelta se ve acotado por O(m). En caso de encontrar algún ciclo de peso negativo, retorna un falso y Johnson termina su ejecución.

En caso contrario, retorna una lista de distancias de s a todo vértice, con complejidad espacial $\mathcal{O}(n)$. Luego, utilizando esta lista, para cada vértice del digrafo G y para cada una de las aristas de susodicho vértice, la rebalanceamos usando la nueva función de peso $\bar{w}$, como la definimos previamente. El costo de este rebalanceo, temporalmente, es $\mathcal{O}(nm)$ y espacial desdeñable, ya que utilizamos estructuras que ya están cargadas en memoria.

A partir de recrear el digrafo G rebalanceado simplemente ejecutamos Dijkstra utilizando como fuente cada uno de los vértices del digrafo. El costo temporal de esto es $\mathcal{O}(n * Dijkstra) = \mathcal{O}(n*m*log(n)$, ya que implementamos Dijkstra utilizando la priority queue de C++, donde el costo de inserción de elementos [1] y eliminación del primer elemento [2] son logarítmicos, pero, hay que tener en cuenta que, si ejecutamos n veces Dijkstra y mantenemos en memoria cada lista de distancias que retorna, el costo espacial seria de $O(nm) \nsubseteq O(m)$. Por lo tanto, luego de cada ejecución de Dijkstra imprimimos la lista de distancias desde el vértice que se corrió y luego eliminamos esa lista para así poder acotar el costo espacial a $O(m)$ y así pudimos implementar el algoritmo de Johnson con las complejidades requeridas. Más aún, la complejidad temporal de este algoritmo termina siendo $\mathcal{O}((Bellamn-Ford) + Rebalancear + N * Dijkstra \subseteq \mathcal{O}(N * Dijkstra) = \mathcal{O}(n*m*log(n)$, como se pide.

Testing

Los casos de test que ejecutamos para este algoritmo son digrafos pesados que o tienen un ciclo negativo, para verificar la detección correcta, o simplemente digrafos para cuantificar el camino mínimo para todo par de vértices. Estas instancias normales fueron compartidas comunalmente. La siguiente tabla representa los tiempos de ejecución en un procesador Ryzen 5 5600X, pero nos abstuvimos de imprimir la salida, ya que en varios test cases M tiene valores mayores a 100:

[1] `https://en.cppreference.com/w/cpp/container/priority_queue/push`
[2] `https://en.cppreference.com/w/cpp/container/priority_queue/pop`

Test	**Tiempo (en milisegundos)**	**Tiene ciclo negativo**
tree-100-false	00.003	False
tree-100-true	00.002	True
tree-500-false	00.17	False
tree-500-true	00.0003	True
tree-1000-false	00.3	False
tree-1000-true	00.0004	True
tree-10000-false	10.00	False
tree-10000-true	00.2	True
tree-50000-false	260.00	False
tree-50000-true	01.00	True
tree-100000-false	670.00	False
tree-100000-true	02.00	True

Conteo de componentes conexas

El grafo G(M) asociado a una matriz M $\in \{0,1\}^{m \times n}$ tiene un vértice por cada posición de M que tiene valor 1, donde dos vértices de G(M) son adyacentes si y solo si sus posiciones son adyacentes en M.

Se presenta un algoritmo que, dada una matriz M, determina la cantidad de componentes conexas de G(M). El algoritmo consume espacio O(n) y tiempo O(mn), sin contar el espacio que ocupa la matriz M en el disco, solo contando lo que se consume de memoria. Notar que M no se carga enteramente en memoria, sino que solo se lee una cantidad constante de filas en simultáneo.

Interfaz de entrada y salida

Cada instancia empieza con una línea con los valores m y n que representan la cantidad de filas y columnas de la matriz M. Luego sigen m líneas, cada una de las cuales tiene n valores en {0,1}. El j-ésimo valor de la i-ésima fila se corresponde a M_{ij}.

Se devuelve una sola línea con la cantidad de componentes conexas de G(M).

Planteo de solución

Se propone el siguiente invariante para un ciclo que procesó la matriz hasta la fila i inclusive:
Sea R_i el subgrafo inducido por G(M) al quitar todos los vértices correspondientes a filas de M posteriores a la fila i, cada vértice de la fila i tiene asignado un identificador exclusivo a la componente conexa en R_i a la que pertenece.
Además, se conoce el conteo K_i de componentes conexas de G(M) contenidas en R_i, que inicialmente vale 0.

Con dicho invariante, se propone el siguiente método para procesar la fila i+1:
Sea F el subgrafo inducido por G(M) al quitar todos los vértices no correspondientes a la fila por procesar, asignar a cada vértice de la fila i+1 un identificador exclusivo a la componente conexa en F a la que pertenece. Sea I el subgrafo inducido por G(M) al quitar todos los vértices no correspondientes a las filas i e i+1, y sea C una compresión de I al reemplazar cada conjunto de vértices con mismo identificador y misma fila por un único vértice, realizar un único recorrido del grafo C para identificar sus componentes conexas.
Incrementar el conteo K según la cantidad de vértices de C en la fila i que estén aislados.
Almacenar para cada vértice de G(M) en la fila i+1, un identificador exclusivo a la componente conexa en C a la que pertenece el vértice dentro del cual fue comprimido, pues será de utilidad en la próxima iteración.

El recorrido del grafo C es realizable en complejidad temporal y espacial O(n), siendo n el largo de cada fila, pues los números de vértices y aristas de C son menores a los de I (por ser C una compresión de I), e I tiene a lo sumo 2*n nodos y 3*n aristas por ser un subgrafo inducido por dos filas de G(M), un grafo de grilla reticulado.
Tras iterar sobre las m filas de la matriz, se logra una complejidad temporal de O(n*m).

Como para efectuar cada iteración solo es necesario conocer los identificadores de las componentes conexas en R_i de los vértices correspondientes a la última fila procesada, cuya cantidad no puede exceder n/2+1, se logra una complejidad espacial O(n) para todo el algoritmo, sin incluir el espacio donde inicie almacenada la matriz de entrada M.

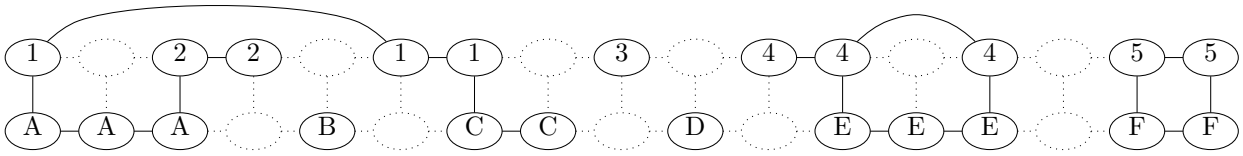

Figura 1: en subgrafo I, se identifican las componentes conexas (A a F) incluidas en el subgrafo F

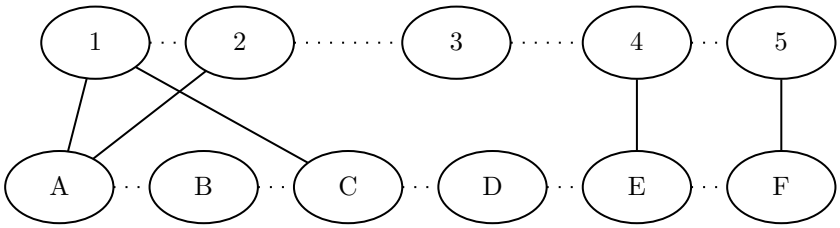

Figura 2: la compresión C del subgrafo inducido I, con un vértice de la fila superior (3) aislado

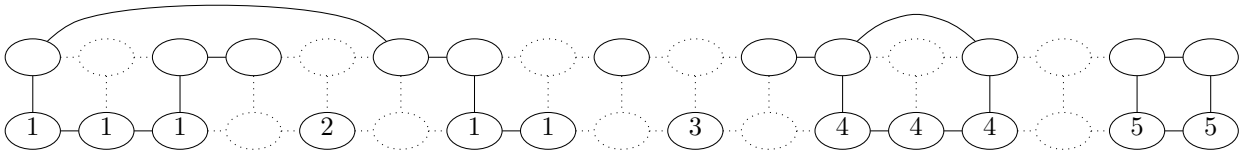

Figura 3: identificadores finales de la fila R_{i+1}, no vinculados a los que supo tener la fila previa

Pruebas

Luego de implementar el algoritmo explicado, se realizaron pruebas para corroborar su correcto funcionamiento y su desempeño.
Para lo primero, se utilizaron casos de prueba pequeños y creados manualmente, pero interesantes y representativos. Para lo segundo, se utilizaron casos de prueba de mayor tamaño y generados automáticamente. En ambos casos, encontrados en el repositorio de La Plebe[1] (grupo de estudiantes).
A continuación se adjuntan los tamaños para algunos de ellos, junto a sus tiempos de corrida (Debian 11, x86_64, Intel i3-3240 (4) @ 3.400GHz) a fin de mostrar la linearidad del algoritmo.

Nombre de instancia	Tamaño (m×n celdas)	Tiempo (segundos)	Tamaño/Tiempo (celdas/μ segundos)
big-frantst16	296.208	0.09	3.2912
big-frantst19	446.688	0.14	3.1906
big-frantst5	562.264	0.17	3.3074
big-frantst10	2.542.782	0.78	3.2600
big-frantst2	3.131.545	0.99	3.1632
big-frantst3	3.691.030	1.12	3.2956
big-frantst17	3.838.262	1.16	3.3088
big-frantst20	5.843.648	1.82	3.2108
big-frantst18	6.624.995	2.04	3.2475
big-frantst14	13.680.900	4.22	3.2419
big-frantst11	15.551.222	4.75	3.2739
big-frantst7	18.975.740	5.71	3.3232

Conjunto dominante total en grafos de intervalos

Los grafos de intervalos son aquellos que sus vértices representan intervalos y existe la arista (i, j) si y solo si el intervalo i se solapa con el intervalo j.

Dado una familia de intervalos I, decimos que $D \subseteq I$ es un *conjunto total dominante* (CDT) si para todo $i \in I$ existe $i' \in D$ tal que $i' \neq i$ e $i' \cap i \neq \emptyset$

En este trabajo se resuelve este problema usando el algoritmo de Bertossi que reduce el problema de dominación total a uno de camino mínimo. La complejidad de este algoritmo es $\mathcal{O}(n^2)$.

Algoritmo de Bertossi

Sea $I = \{I_1, ..., I_n\}$ el conjunto de intervalos al cual se le quiere encontrar un *conjunto total dominante* (CDT). Donde cada $I_i = \{a_i, b_i\}$.

Primero consideramos dos intervalos, I_0 y I_{n+1}, donde estos cumplen:

- $b_0 < \min\limits_{1 \leq k \leq n} \{a_k\}$
- $a_{n+1} > \max\limits_{1 \leq k \leq n} \{b_k\}$

Definimos $I' = I \cup \{I_0, I_{n+1}\}$. Consideramos que en I' los intervalos están ordenados por su comienzo, es decir $a_0 < a_1 < ... < a_n < a_{n+1}$.

Luego definimos el digrafo $D(N, A)$, donde:

- Los vértices en N corresponden a los intervalos en I' que no están totalmente incluidos en algún otro intervalo.
- Las aristas de A están divididas en dos tipos, es decir, podemos definir $A = B \cup C$, donde $B \cap C = \emptyset$.
  - Las aristas en B unen a dos intervalos si estos se solapan, es decir, $(i, j) \in B$ si y solo si $a_i < a_j < b_i < b_j$
  - Las aristas en C unen a dos intervalos si estos no se solapan y no tienen un intervalo entre ellos que tampoco se solape con ninguno de los dos, es decir, $(i, j) \in C$ si y solo si $b_i < a_j$ y no existe un intervalo $I_h \in I'$ tal que $b_i < a_h$ y $b_h < a_j$.

Todas las aristas de A tienen peso 1, excepto aquellas del tipo $(0, i)$, estas tendrán peso 0.

Podemos notar que el grafo resultante es acíclico, ya que en todas las aristas $(i, j) \in A$ tenemos que $a_i < a_j$

Para armar los conjuntos B y C lo hacemos de la siguiente forma, por cada intervalo $I_i \in N$ recorremos todo intervalo $I_j \in I'$ que cumpla que $a_i < a_j$, luego:

- Si $I_j \in N$ y se solapa con él I_i, agregamos a B la arista (i, j).
- Si los intervalos no se solapan, hay que ver si existe un intervalo $I_h \in I'$ tal que $b_i < a_h$ y $b_h < a_j$. Para saber esto, nos guardamos en una variable cuál es el min_b de los intervalos que ya recorrimos y no se solapaban con I_i, de esta forma luego si $I_j \in N$ nos fijamos que $a_j < min_b$, si esto se cumple significa que no hay ningún intervalo I_h tal que $b_i < a_h$ y $b_h < a_j$.

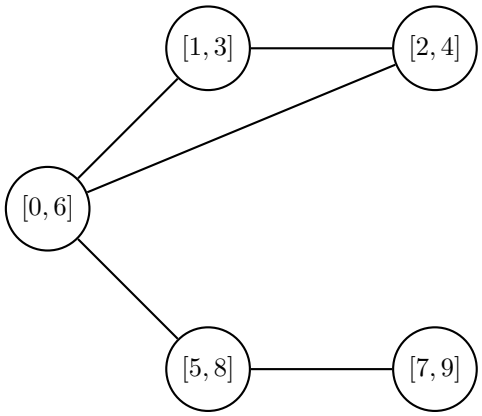

Figura 4: Grafo de intervalos para $I = \{[0,6],[5,8],[1,3][2,4],[7,9]\}$

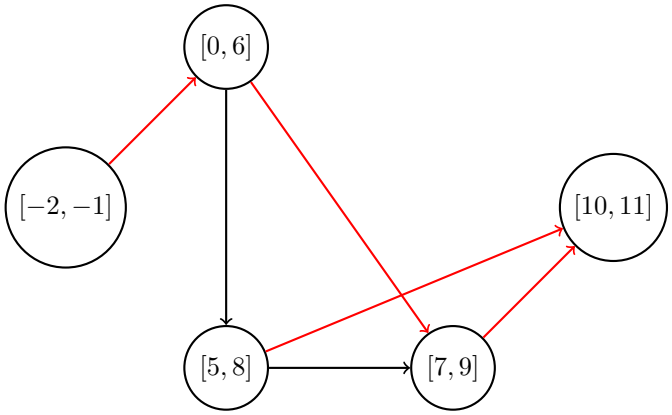

Figura 5: Grafo D para $I = \{[0,6],[5,8],[1,3][2,4],[7,9]\}$, donde $N = \{[0,6],[5,8],[7,,9]\}$, $I_0 = \{-2,-1\}$, $I_{n+1} = \{10,11\}$ y las aristas rojas pertenecen a C.

En la figura 4 podemos ver el grafo de intervalo para los intervalos $I = \{[0,6],[5,8],[1,3][2,4],[7,9]\}$, en este grafo se puede ver que $D = \{[0,6],[5,8]\}$ es un *conjunto dominante total*.

Por el **Teorema 3.2**, descrito en el paper de Bertossi, el *conjunto dominante total* corresponde al camino entre I_0 y I_{n+1} que no tiene dos aristas seguidas dé C.

El problema de encontrar el *conjunto dominante total* se reduce a encontrar el camino mínimo en un grafo con ciertas restricciones en la secuencia de aristas. Definimos un digrafo D' en el que dividimos cada intervalo I_i en dos vértices, un vértice i_{in} y otro vértice i_{out}, que representan el comienzo y el final del intervalo, respectivamente. Estos dos vértices estarán unidos por una arista (i_{in}, i_{out}) con peso 0. El resto de las aristas se agregan de la siguiente forma:

- Por cada arista $(i,j) \in B$, agregamos la arista (i_{out}, j_{in}) con peso 1.
- Por cada arista $(i,j) \in C$, agregamos la arista (i_{in}, j_{out}) con peso 1.
- Por cada arista $(0,i) \in A$, agregamos la arista $(0, i_{out})$ con peso 0
- Por cada arista $(i, n+1) \in A$ agregamos la arista $(i_{in}, n+1)$ con peso 1.

Entonces el camino mínimo entre I_0 e I_{n+1} en D' corresponde a un conjunto total dominante de cardinalidad mínima en el cual un intervalo I_i es incluido si i_{in} o i_{out} o ambos se encuentran en el camino mínimo.

Complejidad del algoritmo

Tanto el digrafo D como el digrafo D' se pueden armar en $\mathcal{O}(n^2)$, ya que ver que aristas pertenecen a B y cuáles a C es $\mathcal{O}(n)$ por cada intervalo, es decir, $\mathcal{O}(n^2)$ en total.

Luego encontrar el camino mínimo entre I_0 e I_{n+1} se puede realizar usando el algoritmo para camino mínimo en digrafos acíclicos. En este algoritmo lo primero que hay que encontrar es el orden topológico de los vértices y, luego, recorrerlos en ese orden actualizando la distancia de sus vecinos, según corresponda. Por cada vértice se recorre a todos sus vecinos, y como cada vértice tiene como máximo $n-1$ vecinos, la complejidad final del algoritmo es $\mathcal{O}(n^2)$.

En la siguiente tabla podemos ver los casos de test:

Test	Tamaño	Cardinalidad	Tiempo (en segundos)
big-100	100	8	00.01
big-1000	1000	7	00.01
big-10000	10000	21	00.17
big-20000	20000	17	00.63
contained-1	4	2	00.01
contained-2	6	4	00.01
contained-3	6	2	00.01
enunciado-1	5	2	00.01
enunciado-2	5	2	00.01
solapan-todos	7	4	00.01

Los primeros 4 test fueron elegidos por el tamaño del conjunto, de esta forma se puede ver como el tiempo va subiendo cuando la instancia crece. `enunciado-2` fue seleccionado porque es un caso donde un solo intervalo contiene a todo el resto.[1]

El resto fueron agregados, ya que era posible chequear manualmente la solución.

Referencias

[1] Repositorio de los test: [Link]