Tarea 2: Costos Amortizados

Profesor: Gonzalo Navarro
Auxiliar: Ricardo Córdova
Ayudantes: Nicolás Canales, Gabriel Iturra, Yuval Linker, Natalia Quinteros.
Estudiantes: Pablo Benario, Valeria Burgos, Álvaro Morales

Noviembre 2022

1 Introducción

El análisis amortizado consiste en considerar el tiempo promedio de ejecución de una seria de operaciones en el peor de los casos. Este tipo de análisis es interesante cuando las operaciones costosas no se ejecutan mucho, lo que se puede dar al alterar el estado de tal manera que tome cierta cantidad de otras operaciones poder volver a realizar esta operación costosa, y en general, estas nuevas operaciones son menos costosas provocando un efecto de "amortización" en las menos eficientes. Para un algoritmo, el conocer las operaciones a realizar permite tomar mejores decisiones respecto a su implementación, por ejemplo, el usar una estructura de datos que permita tener el menor costo amortizado puede reducir de forma importante el tiempo de ejecución para algoritmos que realizan una gran cantidad de operaciones.

En este informe, se considera el problema de la distancia más corta en un grafo con pesos estrictamente positivos, que consiste en dado un grafo $G = (V, E)$ con $|V|$ vértices y $|E|$ aristas, cada una de ella con un costo W_i asociado, definimos como el camino más corto entre dos vértices las aristas que los conectan con el menor peso acumulado posible. De esta forma, el problema de la distancia más corta de u grafo es el de encontrar el árbol de aristas que conectan a un nodo raíz con todos los otros nodos del grafo con los caminos mas cortos. Para esto implementaremos 3 algoritmos:

Algoritmo 1: : Algoritmo de Dijsktra con Listas de Adyacencia: Para este algoritmo, se implementa la descripción dada considerando que los grafos vienen en una lista de adyacencia (cada nodo se almacena dentro de un arreglo, y cada nodo contiene un arreglo de aristas representadas por pares que contienen un puntero a uno de sus vecinos y el peso de esa arista). En este algoritmo se enfocó Álvaro Morales.
Algoritmo 2: : Algoritmo 2: Dijkstra con Heap: Este algoritmo modifica el primero, utilizando una cola de prioridad almacenando los nodos y sus distancias a la raíz como prioridad para poder reducir costos. En este algoritmo se enfocó Valeria Burgos.
Algoritmo 3: : Algoritmo 2: Algoritmo 3: Dijkstra con Colas de Fibonacci.: Para esta versión, se usa la misma variante que el algoritmo de Dijsktra que en el algoritmo 2, pero se cambia la cola de prioridad de un heap a una cola de Fibonacci. En este algoritmo se enfocó Pablo Benario.

Podemos notar que el tiempo de ejecución es altamente dependiente del tipo de estuctura que se ocupe para guardar las aristas. Dijkstra se inicializa siempre un arreglo para cada vértice, $O(V)$, donde guardamos las distancias (inicialmente todas infinito, y el origen 0). Luego en el loop while, hacemos $O(V)$ llamadas para encontrar aquel con $dist[v]$ mínimo (esto es FindMin), y $O(n)$ llamadas a borrar los mínimos encontrados (DeleteMin). Finalmente, cada arista será comparada al menos una vez, en tiempo constante (total $O(E)$), y durante estas comparaciones, habrán a lo mucho $O(E)$ llamadas a disminuir el $dist[v]$ (DecreaseKey). En total esto nos deja con que el tiempo de ejecución de Djikstra es $O(V(FindMin(V) + DeleteMin(V)) + EDecreaseKey(V))$, lo cual para el 1er algoritmo resulta ser $O(V^2 + E)$, para el 2do $O(VlogV + ElogV)$, y para el 3ero $O(VlogV + E)$. Notemos que si V fuera constante, los 3 serían $O(E)$, por lo que esperaríamos 3 plots lineales, solo que a diferentes alturas debido a cómo se manejan con V.

2 Desarrollo

2.1 Implementaciones previas al algoritmo

Antes de implementar los 3 algoritmos se crearon estructuras necesarias para esto y para el testeo de resultados. A continuación se explican:

2.1.1 Grafos: Estructura de la lista de adyacencia

Para poder implementar los 3 algoritmos, se requieren varias estructuras y funciones comunes. Todo fue hecho en C++:

Una lista de adyacencia corresponde a una estructura que contiene V listas, donde cada V[i] representa a un nodo. Cada una de estas listas es una *estructura* con un puntero a una "Lista nodo". Para un vértice v, la lista nodo indica sus aristas y, por lo tanto, esta estructura tiene la información del nodo con el cual se forma la arista, el peso de la arista y la id al nodo con que conecta. En detalle:

Implementación de una **lista nodo**: Representa una arista. Es una estructura con los campos:

1. Un entero (int) que llamando *dest* que indica el índice del nodo con el que se forma la arista.
2. Un entero (int) que llamamos *weight*, que indica el peso de la arista. Por enunciado, este debe ser un valor estrictamente positivo.
3. Un puntero a otra lista nodo, que llamamos *next* la cual es la siguiente arista de un determinado nodo. Para indicar que es la última arista, este campo se vuelve NULL.

Implementación de **lista**: Representa a un vértice. Contiene los campos:

1. Un puntero a una lista nodo, la cual llamamos *head*. Esta lista nodo es una arista para un determinado vértice, y dentro de su campo next se accede a las siguientes aristas del vértice, en caso de existir. Para tener un grafo conexo, es necesario que cada vértice tenga como mínimo una arista.

Implementación de **grafo**: Representa al grafo. Contiene:

1. Un entero **V** que indica los nodos del grafo.
2. Un puntero a un arreglo de listas de adyacencia. Llamémoslo **array**. Este array contiene exactamente V listas, una por cada nodo. Su posición nos indica de que vértice se trata, por lo que para sus aristas representadas por una lista nodo dentro de la lista solo se vuelve necesario conocer el destino de la arista, pues su ubicación entrega el origen.
3. Un puntero a una matriz $V \times V$ que representa las aristas que ya existen con un 1, y las que no con un 0. Funciona a modo de *look-up table* para ver qué aristas ya existen. Útil a la hora de agregar nuevas aristas.

Con eso tenemos el grafo representado. A continuación detallamos algunas de las funciones de utilidad:

1. Funciones para crear la estructura:
   (a) new_Adj_ListNode(int dest, int weight): Función que dado un entero que indica el destino de la arista y el peso, retorna un puntero a una lista nodo. El campo next se inicializa como nulo.
   (b) createGraph(int V) : Dada la cantidad de nodos V, esta función genera un grafo de este tamaño, pidiendo espacio con malloc. Se crean V inicializadas con su campo head nulo y se retorna un puntero a un grafo.
2. Funciones para liberar la memoria pedida por un grafo.
   (a) freeNode_list(struct Node_list* node): Dado un nodo, libera la memoria.
   (b) destroyGraph(struct Graph* graph): Dado un grafo, libera la memoria (Libera la memoria de los nodos también).
3. Funciones para crear grafos funcionales:

(a) addEdge(struct Graph* graph, int orig, int dest, int weight) : Esta función permite añadir una arista a un grafo. Se le debe indicar los dos vértices (orig y dest), además del peso de la arista.

(b) fillInGraphRandomly(struct Graph * graph, int E, int wtRange) : se encarga de llenar un grafo inicializado de aristas elegidas al azar, con pesos al azar, tomando en cuenta la estructura que debe tener el grafo. Tomamos la decisión de agregar exactamente E/V aristas por nodo, con tal de simplificar las cosas.

(c) addEdgesRandomly(struct Graph * graph, int E, int wtRange) : similar a fillInGraphRandomly, pero simplemente agrega aristas nuevas donde sea. Útil cuando pasamos de 2^i aristas a 2^{i+1} aristas.

(d) shuffleEdgeWeights(struct Graph* graph) : Cambia los pesos de todas las aristas al azar.

La idea de estas funciones es poder generar y alocar un grafo al principio, modificarlo para correr cada iteración, y solo destruirlo una vez que se termine todo. ¿Por qué no simplemente destruir y volver a hacer un grafo? Como los grafos van a ser muy grandes, esto sería demasiado costoso, y francamente tomaría horas (o incluso días) de computación si lo hiciéramos así. Con hacer las conexiones al azar solo cuando necesitamos aristas nuevas, y haciéndole *shuffle* a los pesos antes de cada iteración, obtenemos grafos suficientemente distintos entre sí para obtener resultados significativos, y en una fracción del tiempo que tomaría alocar y liberar la memoria que ocuparían varios grafos uno después de otro.

2.1.2 Retorno: Estructura para retornar

Para los tres algoritmos, se pide retornar un par con dos arreglos. Para facilitar esto, se crea una estructura que es utilizada para todos los algoritmos. Esta estructura es llamada **intArrPair** y tiene los siguientes tres campos:

1. Un puntero a un arreglo de tipo int que llamamos *first*.
2. Un puntero a un arreglo de tipo int que llamamos *second*.
3. Un int llamado *size*, que indicará el tamaño de los arreglos de los otros dos campos. Este valor será igual a la cantidad de vértices en u gráfico.

Sobre esta estrucutra, se crean las siguientes funciones para facilitar su manipulación:

- *createIntArrPair(int size)*: función que dado el tamaño de los arreglos, los crea pidiendo memoria con malloc. Retorna una estructura intArrPair.
- *destroyIntArrPair(struct intArrPair pair)*: Función que dada una estructura intArrPair libera la memoria que fue pedida para su creación.

Esta estructura se usa para guardar en su primer campo el arreglo dist y en el segundo campo el arreglo prev.

2.1.3 Experimentos: explicación de como se implementan los experimentos

Los 3 experimentos tienen una interface muy parecida de funciones. La diferencia siendo qué estructura ocupan para guardar las aristas de cada vértice.

En cuanto al experimento en sí, para los 3 algoritmos, dejamos constante la cantidad de vértices como $V = 2^{14}$. La cantidad de aristas $E = 2^i$ va en el rango de $i = 16...24$. Para cada i, se corre el respectivo Dijkstra 50 veces, midiendo el tiempo que se demora, y se rescatan el tiempo promedio y la desviación estándar para cada i.

La memoria del grafo solo se aloca 1 vez, y se libera 1 vez, al principio y al final del experimento. Entre medio, se hace uso de las funciones definidas anteriormente para llenarlo de aristas al azar, cambiar todos los pesos en cada iteración, y agregar las aristas necesarias cuando se pasa a $i + 1$. Esto nos entrega resultados significativos, sin la necesidad de destruir y volver a crear un grafo en cada iteración, lo cual puede ser muy lento para grafos de este tamaño.

2.2 Algoritmo 1: Algoritmo de Dijsktra con Listas de Adyacencia

Este es el algoritmo más directo de los 3. La estructura del grafo solo se implementa con listas de adyacencia, que no son más que simples listas enlazadas, que representan, para cada vértice, todas las aristas que conectan con este vértice. Cada nodo de la lista contiene los índices de origen y destino, y el peso de la arista.

Esto no es particularmente eficiente, ya que nunca tenemos noción de dónde está la lista con la distancia mínima. Estamos obligados a recorrer la lista entera para cada nodo.

Solo ocupamos 3 arreglos, **dist**, para guardas las distancias, **prev**, para guardar el nodo anterior a cada nodo en el camino óptimo, y un arreglo auxiliar **aux**, que será una especie de proxy a lo que sería una cola donde se van guardando los nodos no visitados (0 = no visitado, -1 = visitado). Se procede de la siguiente forma:

1. se inicializan los arreglos (todos de tamaño V): dist se llena con infinitos, prev con -1's (Representando un indefinido), aux con 0's. Por último se marca dist[src] con 0, donde src es el vértice de inicio.
2. también declaramos un bool auxIsEmpty y lo inicializamos con false. Este es nuestro proxy a saber que aux está "vacío" (lo cual equivale a que todos sus valores sean -1).
3. mientras aux no esté vacío:
   (a) Definimos la distancia mínima hasta ahora como infinito.
   (b) Buscamos el mínimo dist[u], y guardamos el nodo u, lo marcamos como vistitado en aux y guardamos el nuevo mínimo dist[u]
   (c) Luego, para todos los vecinos v de u, que sigan en aux como "no vistos", si la distancia guardada para v en dist, es mayor que el peso de la arista que los conecta, cambiamos dist[u] por ese valor, y guardamos a v como prev[u]. (lo cual implica revisar la lista de adyacencia completa)
   (d) checkeamos si aux está lleno de -1's (lo que equivale a que esté vacío).
4. Se aloca memoria para guardar y retornar dist y prev, se copian y se retorna una estructura para devolver 2 arreglos (C++ te obliga a implementar todo).

2.3 Algoritmo 2: Dijkstra con Heap

2.3.1 Implementación heap

Para implementar la cola de prioridad, primero se creó la estructura que la define. En general, se tiene una estrucura Heap que contiene Heap_nodo. Cada una de estas estructuras se detalla de la siguiente forma:

Los **Heap_nodo**, que representan un elemento dentro de una cola de prioridad, se definen como una estructura de C++ con los siguientes dos campos:

1. Un int llamado *v*, el cual representa un índice de un vértice de un grafo.
2. Un int llamado *dist*, el cual representa la prioridad del nodo, que para efectos del algoritmo corresponde a la distancia a la raíz.

Y la estructura del **Heap** se define con los siguientes 4 campos:

1. Un int llamado *size*, que indica la cantidad de Heap_nodo que contiene.
2. Un int llamado *capacity*, que indica la cantidad máxima de Heap_nodo que puede contener la cola de prioridad.
3. Un puntero a un int llamado *pos*,
4. Un puntero a un array de Heap_nodo llamado *array*

Una vez definida la estructura para el heap, se deben crear las funciones que facilitarán su manipulación para usarlo dentro del algoritmo de Dijsktra.

- *nuevo˙heap˙(int capacity)*: dado la capacidad máxima de un heap, lo crea. Se pide espacio con malloc para los parámetros pos y array. El campo size se inicializa en 0, debido a que el heap se crea vacío. Retorna un puntero a un heap.

- *swap˙nodes(struct Heap˙nodo** a, struct Heap˙nodo** b)* : Dado dos heap nodos (Con doble puntero a cada uno de estos), intercambia sus contenidos. Esta corresponde a una función auxiliar necesaria para la implementación de las más complejas.
- *Heap˙Heapify(struct Heap˙* heap˙, int i)*: Función que dado una cola de prioridad (heap) y un índice i hace heapify.
- *isEmpty(struct Heap˙* heap)* : Función auxiliar que dado un heap revisa si este se encuentra vacío.
- *extract˙Min(struct Heap˙* heap)* : Función que dado un heap, extrae el mínimo. Retorna un puntero al Heap_nodo que tiene la menor prioridad en ese momento, de donde se puede recuperar justamente el índice del vértice y su respectiva distancia a la raíz.
- *decreaseKey(struct Heap˙* heap, int v, int dist)*: Función para disminuir el valor dist de un Heap_nodo, y por como se define la prioridad, implica aumentarla dentro del Heap.
- *isInHeap(struct Heap˙ *heap, int v)* : Función auxiliar que permite determinar si un determinado vértice v se encuentra en el Heap.

2.3.2 Implementación algoritmo de Dijkstra

A continuación, se explicará la implementación del algoritmo de Dijkstra usando colas de prioridad. Se creó una función que recibe como parámetro un grafo como lista de adyacencia, y un int que indica el índice del vértice que se desea usar como raíz. Retorna una estructura intArrPair, igual que el algoritmo 1. Los pasos son los siguientes:

1. Del grafo se recupera el número de vértices y se guarda en una variable *V*. Se definen dos arreglos de tipo int tamaño V: dist y prev. También se crea un nuevo heap de tamaño máximo V, que inicialmente está vacío.
2. Se llena el arreglo dist con infinito (En código, representado con el valor de INT_MAX) y el arreglo prev con indefinido (Representando con -1, un valor negativo)
3. Se llena el heap con un elemento por nodo, con prioridad igual a dist. Posteriormente, para el elemento que se usa como raíz se hace que su valor dist en la cola de prioridad sea 0, usando la función decreaseKey para poder cambiar la prioridad en el heap. Se le indica al Heap que ahora contiene V elementos (A través de su campo size).
4. Con esto, se hizo la inicialización de las variables necesarias para calcular la distancia más corta a partir de cierto nodo. Ahora se debe calcular.
5. Mientras el heap no esté vacío, se hace lo siguiente:
   (a) Se usa la función extract_Min(heap) para obtener el Heap_nodo que indica el vértice con la menor prioridad. Para la primera iteración, se extrae el vértice cuyo índice fue indicado como un parámetro a la función para ser la raíz.
   (b) Se obtiene el índice del vértice accediendo al campo v del Heap_nodo obtenido. Este es guardado en una variable de tipo int llamada *u*.
   (c) Se marca el nodo en el arreglo de previos. Esto se hace poniendo en la posición u un valor positivo, en este caso, el valor es 1.
   (d) Se actualiza la distancia a cada vecino de v del nodo de índice u que lo requiera. Para esto, se utiliza el grafo dado como parámetro al a función, y se accede a su campo array[u], que corresponde a la lista de adyacencia del vértice u de donde se pueden recuperar las aristas. Se accede al campo head de la lista de adyacencia, accediendo a la primera de las aristas. Se recorre esto, utilizando el campo next de las listas nodo, hasta que este tenga valor nulo. Para cada una de estas aristas, se hace lo siguiente:
      i. Se recupera el índice del vértice v con el que se une el vértice u a través de una arista.
      ii. Se comprueba que efectivamente v se encuentre en el heap. Si v está en el heap, entonces se comprueba que la distancia del vértice u no sea infinita.

iii. De haber pasado los chequeos, se evalúa si la distancia del nodo con índice v es mayor a la distancia para el vértice de índice u más el peso de la arista que los une. Si es así, se cambia el valor de la distancia de v al de la distancia de u más el peso de la arista.

iv. A continuación, se guarda u como el nodo previo de v. Para esto, se cambia en el arreglo prev la posición v al valor del índice u.

v. Luego, se actualiza dis en el heap usando decreaseKey(heap, v, dist[v]).

vi. Si existe otro vecino para el nodo u, se continúa con este.

(e) Se prepara el par de retorno. Se crea una estructura intArrPair, donde el primer campo corresponde al arreglo dist y el segundo al prev. Se retorna esta estructura.

2.4 Algoritmo 3: Dijkstra con Colas de Fibonacci

2.4.1 Implementación Colas de Fibonacci

Para implementar las Colas de Fibonacci, también llamadas Heap de Fibonacci, se siguió una estructura parecida a la anterior, es decir primero se creó la estructura fundamental, que vendrían siendo los nodos, también se necesita un puntero al nodo de mínima prioridad, que lo implementamos con una variable global llamada mini, que inicialmente apunta a NULL, ya que se tiene un árbol vacío. Asimismo se cuenta con la variable global no_of_nodes = 0; que tal como su nombre indica representa el numero de nodos que tiene el árbol actualmente, esta está inicializada en cero.

Los **node**, que representan un elemento dentro del heap, se definen como con una estructura de C++ con los siguientes campos:

1. Un puntero al nodo padre llamado *parent*.
2. Un puntero al nodo hijo llamado *child*.
3. Un puntero al nodo izquierdo llamado *left*.
4. Un puntero al nodo derecho llamado *right*.
5. Un int llamado *key*, el cual representa la prioridad del nodo.
6. Un int llamado *n*, el cual representa el vertice del grafo.
7. Un int llamado *degree*, el cual representa el grado del nodo, que es la cantidad de hijos que tiene.
8. Un char llamado *mark*, que nos indica el color del nodo, Black or White. Esto sirve para implementar la interfaz del Heap.
9. Un char llamado *c*, el cual sirve de Flag para asistir en la implementación de la interfaz del heap, siendo mas específicos sirve para la implementación de la función Find.

Una vez definida la estructura de la Cola de Fibonacci, debemos crear la funciones que implementaran la interfaz del Heap.

- insertion(int n, int val): Dado un n, que será el numero del vértice en este caso, y un val, se inserta a la Cola de Fibonacci un nodo con esos atributos, luego retorna un puntero al nodo insertado.
- Fibonacci_link(struct node* ptr2, struct node* ptr1): Dados dos punteros a nodos, ptr2 y ptr1, deja a los nodos a los que apuntan en una relacion padre e hijo, queda el *ptr1 como padre y *ptr2 como hijo. Retorna void.
- Consolidate(): Consolida el heap para que ningún nodo raiz tenga el mismo grado que el otro. Retorna void.
- Extract_min(): Extrae el mínimo del heap. Retorna void ya que solo lo extrae, no retorna nada, pero mantiene la estructura del heap.
- Cut(struct node* found, struct node* temp): Función que corta un nodo en el heap y lo coloca en la lista de raíces del árbol. Es fundamental para implementar Decrease_key.

- Cascade_cut(struct node* temp): Función que realiza un corte en cascada al heap de manera recursiva. Es fundamental para implementar la operación Decrease_key.
- Decrease_key(struct node* found, int val): El método mas importante, dado un puntero al nodo del heap y una nueva prioridad, cambia la prioridad antigua por la nueva y mantiene la estructura de este.

2.4.2 Implementación algoritmo de Dijkstra

El algoritmo de Dijkstra usando colas de Fibonacci es idéntico al usando Heap. Los únicos cambios en el código a realizar son las partes relacionadas al a cola de prioridad, que en la parte anterior se empleaban las funciones relacionadas al heap y en esta nueva implementación se utilizan sus equivalentes, pero para una cola de Fibonacci (Función para crear y llenar una cola, encontrar el min, cambiar prioridades).

3 Resultados

3.1 Hardware utilizado

Los tres experimento se realizaron en el mismo dispositivo para poder compararlos. A continuación, se entregan las especificaciones de las características de hardware utilizado, y del lenguaje de programación:

- Capacidad RAM: 16 GB (Aunque ningún experimento superó los 10 GB)
- CPU: Intel Core i7-7700HQ @ 2.80GHz
- SO: Windows 10
- Lenguaje de programación: C++
- Compilador: g++.exe (GCC) 12.1.0 (de w64devkit)

3.2 Algoritmo 1: Algoritmo de Dijsktra con Listas de Adyacencia

En la siguiente tabla se muestran los resultados (Error % = $\sigma \times 100$/Tiempo):

E	**Tiempo** $[\mu s]$	σ	**Error%**
2^{16}	1.32045e+06	± 10540.6	0.798
2^{17}	1.34186e+06	± 12080.7	0.9
2^{18}	1.40227e+06	± 11547.8	0.824
2^{19}	1.51582e+06	± 23284.6	1.536
2^{20}	1.67122e+06	± 10224.5	0.612
2^{21}	1.88194e+06	± 18202.2	0.967
2^{22}	2.30277e+06	± 69028.8	2.998
2^{23}	3.50573e+06	± 97389.6	2.778
2^{24}	6.16724e+06	± 128313.0	2.081

Los resultados se muestran gráficamente en las siguientes dos imágenes, incluyendo la desviación estándar calculada. Se muestra en escala lineal y en escala logarítmica.

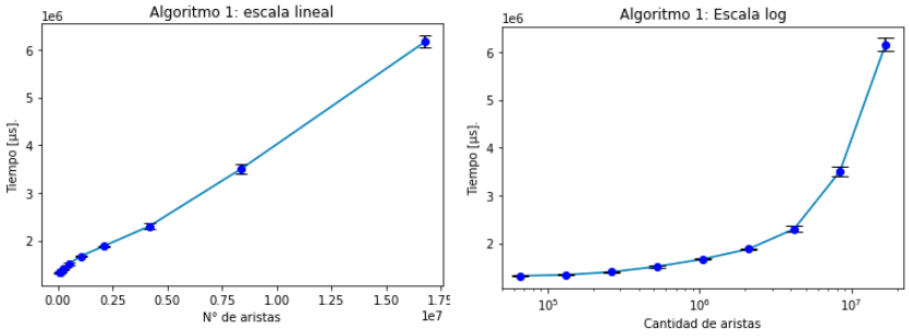

Figure 1: Resultados para algoritmo 1: escala lineal y log

3.2.1 Análisis

Podemos ver que efectivamente crece de forma lineal. Como vimos anteriormente, al dejar V fijo, la complejidad sería $O(E)$. En detalle esto ocurre porque las listas de adyacencia te obligan a revisarlas completas. Al ser de $O(V)$ tamaño, y al tener que revisarlas $O(V)$ veces, terminamos con un factor $O(V^2)$. En específico, esto equivale a que FindMin sea $O(V)$, mientras que DeleteMin y DecreaseKey son $O(1)$, ya que solo son acceder al arreglo auxiliar con el vértice dado. Recordando que en total, la complejidad de Dijkstra es $O(V(findMin(V) + deleteMin(V)) + EdecreaseKey(V))$ quedamos con $O(VV + E) = O(V^2 + E)$ lo cual con V cte. nos da $O(E)$.

También notemos que los Error % son bastante bajos, lo cual es de esperarse, ya que este algoritmo está obligado a revisar las listas de adyacencia completas en cada iteración. No habrán muchos casos en que a veces se demore mucho y otras veces poco.

3.3 Algoritmo 2: Dijkstra con Heap

E	**Tiempo** $[\mu s]$	σ	**Error%**
2^{16}	42310.2	± 3196.15	7.554
2^{17}	56497	± 3187.74	5.642
2^{18}	84809.9	± 3198.45	3.771
2^{19}	141401	± 3894.67	2.754
2^{20}	259718	± 7620.49	2.934
2^{21}	502708	± 12725.9	2.531
2^{22}	1.02721e+06	± 17620.5	1.715
2^{23}	2.31253e+06	± 114630	4.957
2^{24}	5.06004e+06	± 28792.4	0.569

Gráficos (lineal y log):

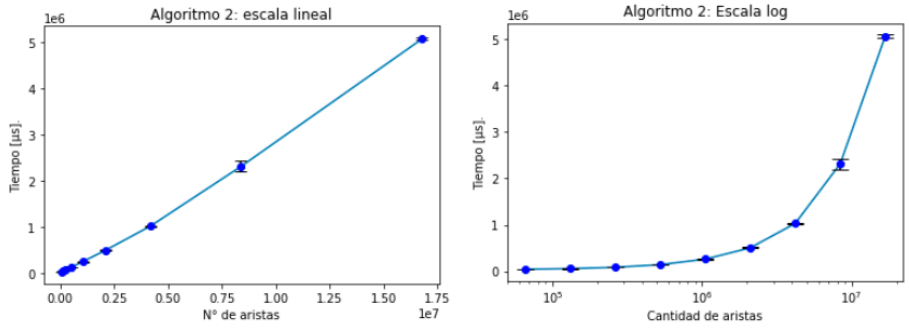

Figure 2: Resultados para algoritmo 2: escala lineal y log

3.3.1 Análisis

Para iniciar el análisis , se comenzará analizando la estructura del heap creado y los tiempos que esta toma.

La cola de prioridad llama principalmente a 2 funciones: *extract_Min(heap)* y *decreaseKey(heap, v, dist[v])*. A continuación, se analiza cada una de estas:

- *extract_Min(heap)*: Cumple la función de encontrar el mínimo en el heap, borrarlo del mismo, y retornarlo. Esta función toma tiempo $O(Log(V))$, ya que se mezclan 3 operaciones básicas de los heaps en una función con ayuda de otras funciones auxiliares. Encontrar el mínimo toma $O(1)$, pero al extraerlo se debe corregir la cola de prioridad, tomando tiempo $O(Log(V))$, y el retornarlo toma tiempo O(1). Con esto, se obtiene la complejidad total de $O(Log(V))$ para esta operación.

  Dentro del algoritmo de Dijkstra se hacen $O(V)$ llamadas a esta función, puesto que en el heap se guardan las prioridades para cada uno de los V nodos y se van obteniendo uno a uno aquellos con la mejor prioridad, en este caso, representando como la menor distancia a la raíz.

- *decreaseKey(heap, v, dist[v])*: Esta función toma tiempo $O(Log(n))$, por un ciclo while en su implementación en el cual los nodos se van intercambiando con sus nodos padres.

  En la implementación del algoritmo de Dijkstra, se hace $O(E)$ llamadas a esta función, pues por cada uno de los nodos se llama una vez para cada uno de sus vecinos, es decir, por cada arista en el grafo.

Recordando la hiótesis entregada para la complejidad, que era:

$$O(VFindMin(V) + DeleteMin(V)) + EDecreaseKey(V))$$

Al reemplazar esto por los costos conocidos nos queda una complejidad de

$$O(VLog(V)) + ELog(V)) = O((V + E) \cdot Log(V)) = O(ElogV + VlogV)$$

Ahora, es posible notar que esto es consistente con los resultados obtenidos experimentalmente. Debido a que para los experimentos el número de nodos se mantuvo fijo y solo se cambió la cantidad de aristas en el grafo, se esperaba observar una relación directamente proporcional entre el tiempo que toma el algoritmo y el número de aristas, lo que en los gráficos generados es observado claramente.

También notemos que las desviaciones estándar no son muy distintas al algoritmo 1. Esto también tiene sentido, ya que nuevamente no hay razones por las cual deberían variar mucho, fuera de la estructura del grafo.

3.4 Algoritmo 3: Dijkstra con Colas de Fibonacci

En la siguiente tabla se muestran los resultados obtenidos para el tercer algoritmo ejecutando 50 iteraciones por cada largo de arista:

E	**Tiempo** $[\mu s]$	σ	**Error%**
2^{16}	40768	± 10401	25.513
2^{17}	49602.9	± 17992.7	36.273
2^{18}	42283.9	± 11364.7	26.877
2^{19}	41338.8	± 11449	27.696
2^{20}	42494.6	± 11698.6	27.53
2^{21}	44032	± 14116.9	32.061
2^{22}	42956.3	± 10788.2	25.114
2^{23}	49760.3	± 11807.6	23.729
2^{24}	46755.3	± 13647.9	29.19

Gráficos (linear y log):

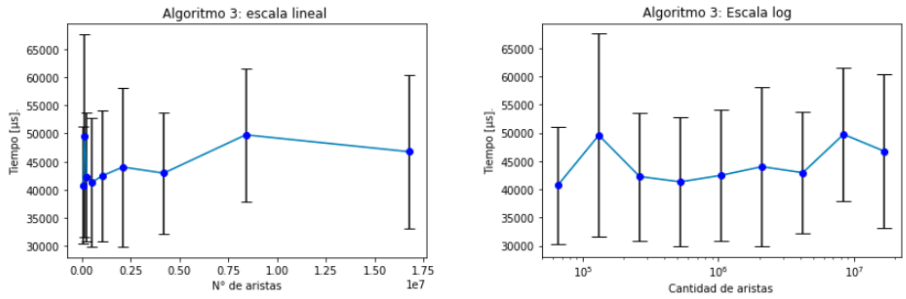

Figure 3: Resultados para algoritmo3: escala lineal y log

Acá hay 2 cosas que destacar. 1) Con colas de Fibonacci resulta ser mucho más rápido que los otros 2 y 2) Los errores resultan ser mucho más grandes proporcionalmente. Por esto, decidimos volver a probar el algoritmo 3, pero con 500 iteraciones, con el objetivo de obtener errores más chicos:

E	**Tiempo** $[\mu s]$	σ	**Error%**
2^{16}	44074	$\pm$ 7969.69	19.549
2^{17}	43582.1	$\pm$ 5322.16	10.73
2^{18}	43565.3	$\pm$ 4581.35	10.835
2^{19}	43660.8	$\pm$ 4514.55	10.921
2^{20}	44927.4	$\pm$ 5930.98	13.957
2^{21}	44609	$\pm$ 5785.93	13.14
2^{22}	44811.5	$\pm$ 5986.89	13.937
2^{23}	44422.4	$\pm$ 6223.04	12.506
2^{24}	47785.6	$\pm$ 9251.97	19.788

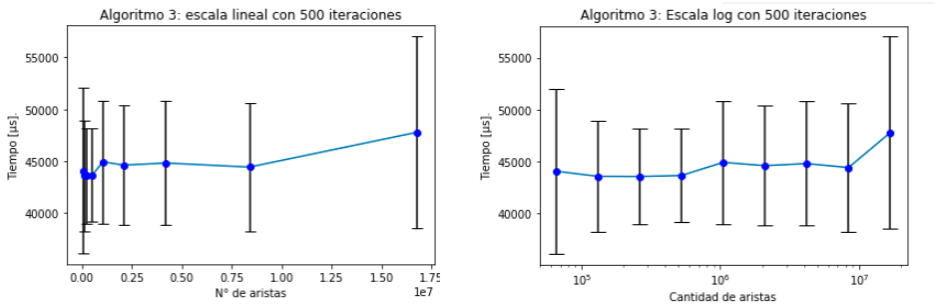

Figure 4: Resultados para algoritmo3 con 500 iteraciones: escala lineal y log

3.4.1 Análisis

Inmediatamente notamos lo rápido que es con colas de Fibonacci. Y a su vez, lo grandes que son las desviaciones estándar, el relación a el Tiempo. ¿Por qué ocurre ésto? Lo explicaremos más adelante.

En cuanto a por qué tiene sentido que con colas de Fibonacci, también sea de orden lineal, analicemos de la misma forma que hicimos con el algoritmo 2:

- *extract_Min(heap)*: Al igual que en el algoritmo 2, se hace uso de ésta función, pero para colas de Fibonacci, ésta es $O(logV)$ amortizado. Esto significa que, es aislación podría ser mayor a ésto, pero en medio de muchas operaciones de las cual depende, tiende a ser de orden $O(logV)$. En este caso estas operaciones son Insert, FindMin y DecreaseKey. Éstas 3 "pagan" por ExtractMin.

  Como se hacen $O(V)$ llamadas a extractMin, se aquí viene la parte de $O(VlogV)$ de la complejidad.

- *decreaseKey(heap, v, dist[v])*: Las colas de Fibonacci, al ser la implementación *lazy* de Colas Binomiales, tiene costo $O(1)$ para DecreaseKey, la cual se llama E veces, dándonos $O(E)$ y para un total de $O(EyVlogV)$.

3.5 Comparación de los 3 algoritmos

A continuación se muestran gráficos para comparar visualmente los resultados obtenidos para los tres algoritmos probados. El primero rmuestra los resultados en el mismo espacio, en escala lineal y logarítmica.

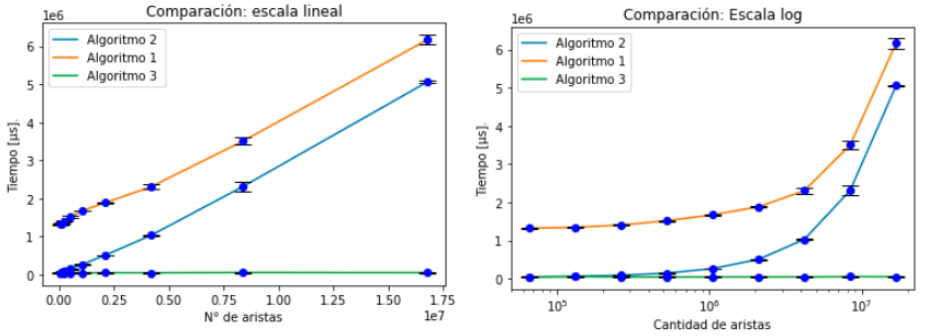

Figure 5: Comparación de los 3 algoritmos: escala lineal y log

En el segundo se muestra la comparación de la desviación estándar para cada uno de los algoritmos, incluyendo el caso de 500 iteraciones para el tercer algoritmo.

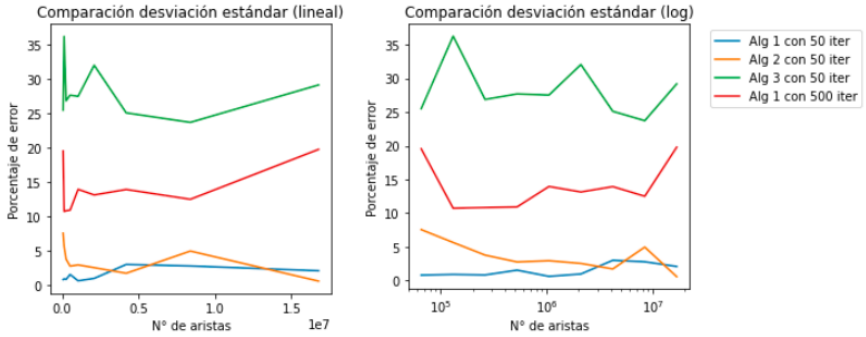

Figure 6: Comparación errores: escala lineal y log

3.5.1 Análisis

Sumado al análisis individual de cada algoritmo que se hizo previamente, también resulta interesante analizar los resultados experimentales en conjunto de los 3 algoritmos.

Del primer gráfico, es posible observar que los algoritmos 1 y 2 tienen un comportamiento lineal tal como se esperaba, pero es interesante notar que el algoritmo 2, es decir, utilizando un heap como cola de prioridad, toma menos tiempo que el primero. Esto se da justamente por las ventajas que proporciona el utilizar esta estructura de datos. Esta variación se observa únicamente al mantener constante el número de vértices del grafo y variar solamente las aristas, ya que de no ser así se espera un comportamiento distinto.

También resulta interesante analizar las desviaciones estándar obtenidas para cada algoritmo. Para facilitar la comparación entre los 3 valores, se calculó el siguiente valor para cada uno de los σ:

$$E = \frac{\sigma \cdot 100}{med}$$

Donde med corresponde a la respectiva medición en la cual se obtuvo la desviación estándar.

De este gráfico, es posible notar que los valores obtenidos para los algoritmos 1 y 2 son bastantes similares entre si y lineales a lo largo del tiempo, es decir, la cantidad de aristas no tienen un gran impacto en ellos. Además, tiene valores bastante pequeños en relación al tiempo de ejecución total.

Lo interesante ocurre en el caso de el algoritmo utilizando colas de Fibonacci. Acá se pueden observar valores mucho más altos en relación a los otros dos. ¿Por qué? por los tiempos amortizados. La mayor parte del tiempo no se ejecutan operaciones costosas, pero eventualmente se llamará a ExtractMin, que amortizadamente es $O(logV)$, pero esto quiere decir, que entre más operaciones se hagan antes de sí, más tendrá que pagar por eso. Entre más tiempo pase sin llamarse a ExtractMin, más tardará en ejecutarse la próxima vez que se llame, y este tiempo entre ExctractMin's es impredecible, lo que quiere decir que a veces se estará llamando frecuentemente, y otras veces no. Esta irregularidad produce tiempos bastante disparejos entre sí, creando ésta desviación estándar tan alta, aunque vale destacar que logramos más o menos reducir la desviación estándar promedio a mas o menos la mitad, tirando 10 veces más iteraciones.

4 Conclusión

Para la realización del informe se implementaron 3 algoritmos para resolver el mismo problema de encontrar la distancia más corta en un grafo con pesos estrictamente positivos, cada uno con variaciones únicamente en las estructuras de datos utilizadas.

El primer algoritmo utilizaba arreglos simples para guardar y recuperar información. Según el análisis realizado, se dio como hipótesis una complejidad de $O(V^2 + E)$, lo cual resultó ser consistente con los resultados encontrados. Este algoritmo resultó ser el más simple de implementar, pero también el más lento de los 3 de forma considerable.

Para el segundo algoritmo, donde se utilizó una heap para implementar el algoritmo de dijkstra, se esperaba una complejidad de $O(VlogV + ElogV)$, lo cual también resultó ser consistente con los resultados obtenidos. La implementación, un poco más compleja que en el caso anterior, permito que el tiempo de ejecución disminuyera en relación al algoritmo. A la vez, el análisis para este algoritmo es en el peor caso, ya que las operaciones ejecutadas no permitían un análisis amortizado interesante que pudiese dar una complejidad distinta al tradicional.

Para el tercer algoritmo se utilizó una cola de Fibonacci, esperando como hipótesis una complejidad de O(V logV + E), que los resultados experimentales respaldaron al ser consistentes y haber obtenido una gráfico lineal. Para este caso, se utilizó el análisis amortizado para determinar que el costo de las operaciones era baja amortizadamente, permitiendo obtener un tiempo de ejecución extremadamente más bajo en comparación a los otros dos algoritmos, aparentando casi ser constante.

La clave de la diferencia entre los algoritmos dos y tres se encuentra en la función decreaseKey(), que para el caso del heap toma tiempo $O(logV)$ y para el caso de la cola de Fibonacci toma tiempo $O(1)$ amortizado y se ejecuta de forma proporcional al número de aristas en el grafo.

Para la desviación estándar en el tercer algoritmo, se pueden observar valores mucho más altos en porcentaje que para los primeros dos. Este disminuye a mayor cantidad de iteraciones a realizar del experimento y se puede asociar justamente al estar hablando de tiempos amortizados, ya que ocasionalmente se ejecutan operaciones que toman una gran cantidad de tiempo las cuales aumentan notoriamente el valor, a diferencia de los otros algoritmos donde el valor se mantiene relativamente bajo y constante en el tiempo. Esto también apoya que las hipótesis planteadas son correctas, pues la diferencia se asocia justamente al tiempo amortizado del tercer algoritmo.

De los resultados obtenidos, se puede deducir la importancia de elegir bien la estructura de datos a utilizar en un determinado algoritmo. Es claro que en este caso, utilizar una cola de Fibonacci es la mejor opción ya que reduce el tiempo de ejecución notoriamente en relación a los otros dos casos. Esto también muestra la importancia de considerar los tiempos amortizados para algoritmos que buscan ejecutarse con una gran cantidad de variables, como es este caso, puesto que su correcto análisis permite tomar mejores desiciones a la hora de realizar implementaciones.

5 Fuentes

1. Wikipedia - Dijkstra
2. Himanshu Bhandoh - Lectura sobre costos amortizados y runtime de Dijkstra
3. geeksforgeeks - Dijkstra con Prim y usando Heap
4. geeksforgeeks - Dijkstra con Colas de Prioridad y STL
5. geeksdorgeeks - Heaps
6. geeksforgeeks - Grafos
7. CodeProject - sobre Dijkstra
8. Ejemplo de Dijsktra en C++
9. AdarshNaidu - Repositorio sobre Dijkstra con colas de Fibonacci
10. Clases de CC4102 - Diseño y Análisis de Algoritmos 2022-2