Universidad de Chile
Facultad de Ciencias Físicas y Matemáticas
Departamento de Ingeniería en Ciencias de la Computación
CC4102 Diseño y Análisis de Algoritmos

# Algoritmo de Dijkstra

Tarea 2

**Integrantes:** Amaro Zurita
Javier Andrés Facondi
Rodolfo Salgado
**Profesores:** Benjamín Bustos
**Auxiliar:** Sergio Rojas H.

# 1. Introducción

En teoría de grafos, encontrar el camino óptimo entre dos vértices es uno de los problemas más estudiados de esta área, ya que tiene importantes aplicaciones en otras disciplinas. Entre las técnicas más usadas para resolverlo está el Algoritmo de Dijkstra, debido a su rapidez en comparación con otros algoritmos, como el de Bellman-Ford. Como parte de la implementación de este algoritmo, se hace uso de una cola de prioridad, por lo tanto existen distintas variantes de este con diferentes tipos de estructuras que buscan implementar esta funcionalidad. Esto trae consecuencias para la rapidez del Algoritmo de Dijkstra, ya que una cola de prioridad más eficiente hará que disminuya el tiempo de ejecución.

En el presente informe se describirá la implementación de dos variantes del Algoritmo de Dijkstra para encontrar los caminos más cortos en grafos no dirigidos y con pesos estrictamente positivos. La primera hará uso de un Heap como cola de prioridad, mientras que la segunda utilizará una Cola de Fibonacci. La teoría señala que la implementación con Colas de Fibonacci es la versión más eficiente, con un tiempo de $O(V \log V + E)$, en donde $V$ y $E$ es la cantidad de vértices y aristas respectivamente, mientras que la implementación con un Heap es más lenta con un tiempo de $O(V^2)$. Para comprobar esta hipótesis se medirán los tiempos de ejecución de ambas variantes para grafos aleatorios de distintos tamaños y así determinar cual es efectivamente más rápido.

# 2. Desarrollo

Para esta tarea, se tuvo que programar el algoritmo de Dijkstra y aplicarlo con 2 estructuras, un Heap y una cola de Fibonacci como colas de prioridad.

A continuación se detallan estas implementaciones.

### 2.0.1. Estructura MinHeap

Métodos de la clase:

- *heapifyDown*: Método privado que mantiene la propiedad del MinHeap hacia abajo. A partir de un índice `idx`, se compara el nodo con sus hijos izquierdo y derecho, y se intercambia con el menor de ellos si es necesario. Este proceso se repite recursivamente hasta que la propiedad del MinHeap se restablezca en esa rama del árbol.
- *heapifyUp*: Método privado que mantiene la propiedad del MinHeap hacia arriba. A partir de un índice `idx`, se compara el nodo con su padre y se intercambia si el nodo es menor que su padre. Este proceso se repite recursivamente hasta que la propiedad del MinHeap se restablezca hacia la raíz.
- *isEmpty*: Verifica si el MinHeap está vacío. Devuelve `true` si está vacío y `false` en caso contrario.
- *insert*: Inserta un nuevo nodo en el MinHeap. El nodo se añade al final del vector y luego se ajusta hacia arriba para mantener la propiedad del MinHeap.
- *extractMin*: Extrae el nodo con la distancia mínima del MinHeap. El nodo raíz (mínimo) se reemplaza con el último nodo del MinHeap, se ajusta hacia abajo para restaurar la propiedad del MinHeap, y se devuelve el nodo extraído.
- *decreaseKey*: Disminuye la distancia de un nodo en el MinHeap. Actualiza la distancia del nodo especificado y ajusta hacia arriba para mantener la propiedad del MinHeap.
- *contains*: Verifica si el MinHeap contiene un vértice específico. Devuelve `true` si el vértice está presente en el MinHeap y `false` en caso contrario.

### 2.0.2. Estructura Cola de Fibonacci

**Clase NodeInfo** La clase `NodeInfo` representa la información de un nodo vecino en el grafo.

- `neighbor`: Identificador del vecino.
- `weight`: Peso del borde hacia el vecino.

**Clase Node2** La clase `Node2` representa un nodo en el Fibonacci Heap.

- `key`: Clave del nodo (utilizada para comparar nodos).
- `info`: Información del nodo.
- `degree`: Grado del nodo (número de hijos).
- `mark`: Marca para operaciones de corte en cascada.
- `child`: Puntero al primer hijo.
- `parent`: Puntero al nodo padre.
- `prev`: Puntero al nodo anterior en la lista de hermanos.
- `next`: Puntero al nodo siguiente en la lista de hermanos.

**Clase FibonacciHeap** La clase `FibonacciHeap` representa un Fibonacci Heap.

- `nodes`: Vector que almacena los nodos.
- `minNode`: Puntero al nodo con la clave mínima.
- `totalNodes`: Total de nodos en el heap.

**Metodos**

- *removeNodeFromList*: Remueve un nodo de la lista de hermanos actualizando los punteros `prev` y `next` de los nodos adyacentes.
- *addNodeToList*: Añade un nodo a la lista de hermanos actualizando los punteros `prev` y `next` de los nodos involucrados.
- *push*: Inserta un nuevo nodo en el heap. Crea un nuevo nodo con el vecino y peso proporcionados, lo inserta en el vector de nodos y en el heap, actualizando el total de nodos.
- *insertIntoHeap*: Inserta un nodo en el heap, añadiéndolo a la lista de raíces y actualizando el `minNode` si es necesario.
- *unionHeaps*: Une dos Fibonacci Heaps combinando sus listas de raíces y actualizando el `minNode` y el total de nodos.
- *isEmpty*: Verifica si el heap está vacío. Devuelve `true` si el heap está vacío (`minNode` es `nullptr`), `false` en caso contrario.
- *pop*: Extrae el nodo con la clave mínima del heap. Extrae el nodo con la clave mínima, lo elimina del heap, añade sus hijos a la lista de raíces y llama a `consolidate()` para mantener la estructura del heap.

- *consolidate*: Consolida el heap para mantener la estructura del Fibonacci Heap. Consolida el heap combinando nodos de la misma orden y reconstruyendo la lista de raíces para mantener la estructura del Fibonacci Heap.
- *linkNodes*: Enlaza dos nodos en el heap, haciendo que `y` sea hijo de `x` y actualizando los punteros y el grado de `x`.
- *decreaseKey*: Disminuye la clave de un nodo específico, realiza cortes si es necesario y actualiza el `minNode` si es necesario.
- *cutNode*: Corta un nodo de su padre y lo añade a la lista de raíces, actualizando los punteros y el grado del padre.
- *cascadingCut*: Realiza el corte en cascada de un nodo, cortando repetidamente nodos marcados y actualizando las marcas.

### 2.0.3. Dijkstra

El algoritmo de Dijkstra se utiliza para encontrar las rutas más cortas desde un nodo origen a todos los demás nodos en un grafo ponderado sin aristas de peso negativo. Inicialmente, se establecen las distancias desde el nodo origen a todos los nodos como infinito, excepto para el nodo origen, cuya distancia es 0. Se inserta el nodo origen en una estructura $Q$ (que puede ser un `MinHeap`, un `FibonacciHeap`. Mientras $Q$ no esté vacía, se extrae el nodo con la menor distancia almacenada. Para cada vecino de este nodo, se calcula la distancia potencial a través del nodo actual. Si esta distancia es menor que la distancia previamente almacenada para el vecino, se actualiza la distancia y el predecesor del vecino, y se inserta o ajusta la prioridad del vecino en $Q$. El proceso se repite hasta que todos los nodos hayan sido procesados y se hayan encontrado las distancias mínimas desde el nodo origen a todos los demás nodos.

### 2.0.4. Test

Se crean tests para probar la eficacia de Dijkstra con cada estructura y posteriormente medir el tiempo de ejecución en la construcción, para esto se crearon grafos conexos con aristas y pesos aleatorios, se crean grafos aleatorios con pesos entre 0 y 1 con cantidad de nodos $v = \{2^{10}, 2^{12}, 2^{14}\}$ y número de aristas $e = \{2^{16}, ....., 2^{22}\}$. Esto se realiza 50 veces por cada par $(v, e)$ y se ejecuta el algoritmo de Dijkstra haciendo uso de ambas estructuras, guardando un archivo de texto con la duración promedio de cada ejecución.

# 3. Resultados

Para la ejecución del código se utilizo un PC con las siguientes especificaciones:

- Procesador: Intel Core i5-2415M
  - Frecuencia: 2.3 GHz a 2.9 GHz
  - Cores: 2
  - Threads: 4
- Memoria principal (RAM): 8 GB a 1333 MT/s
- Cache de CPU:
  - L1: 64 KB
  - L2: 512 KB
  - L3: 3 MB
- Sistema operativo: Manjaro Linux (Kernel 6.6.32-1)
- Compilador: GCC 14.1.1

Como se mencionó en la parte anterior, para comparar las variantes del algoritmo se midió el tiempo de ejecución promedio para distintos valores de $v$ y $e$. A continuación se muestran las comparaciones del tiempo de ejecución de ambos métodos para distintos valores de $v$ junto con un ajuste lineal realizado sobre las curvas.

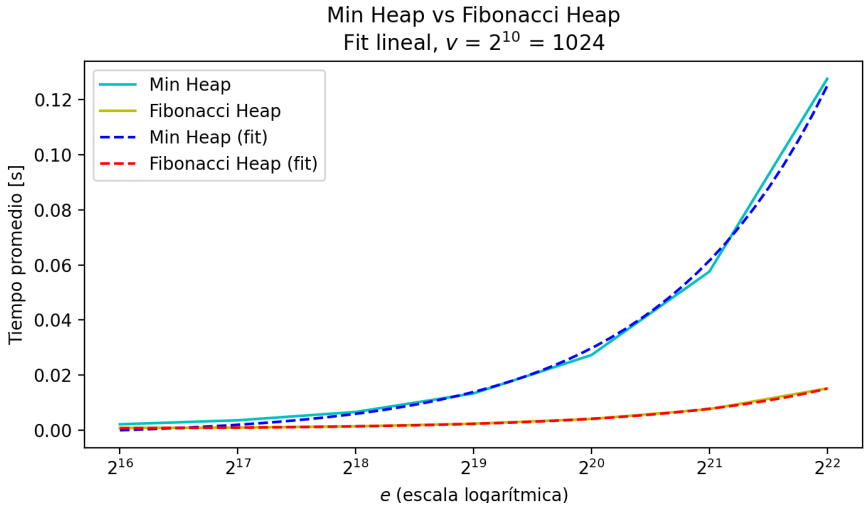


Figura 3.1: Comparación para $v = 2^{10}$

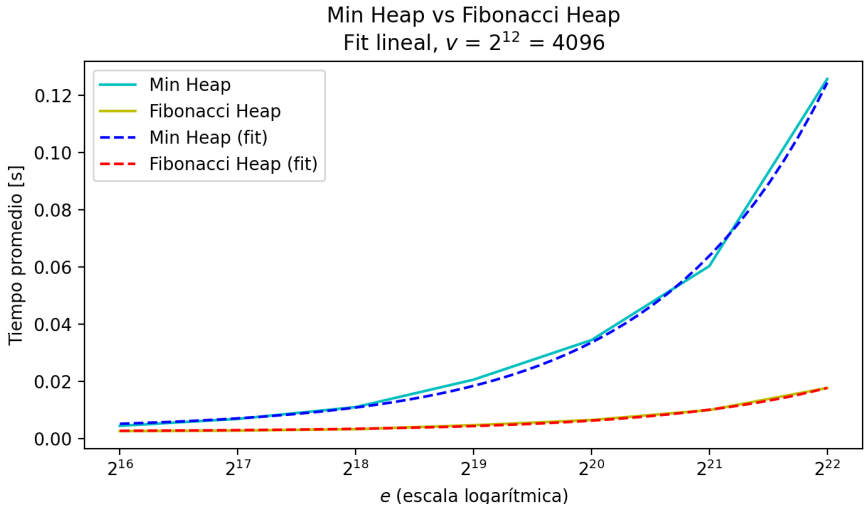


Figura 3.2: Comparación para $v = 2^{12}$

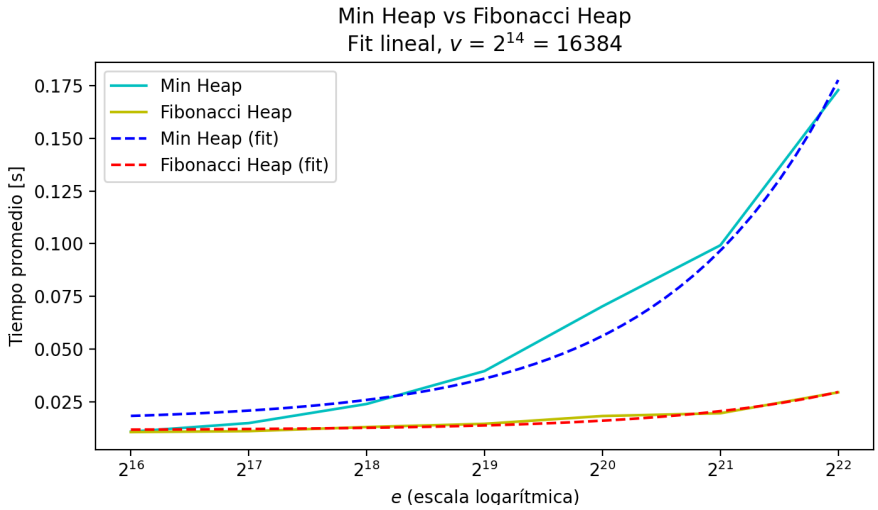


Figura 3.3: Comparación para $v = 2^{14}$

En los tres gráficos se puede apreciar un comportamiento similar, en donde los tiempos de ejecución para ambos tipos de Heap tienen un crecimiento lineal según el número de vértices (notar que el eje horizontal está en escala logarítmica). También se aprecia que el tiempo de ejecución del Fibonacci Heap es menor y de crecimiento más gradual que el del Min Heap.

Los valores de la pendiente resultante al realizar el ajuste lineal se presentan en la siguiente tabla.

| | $v = 2^{10}$ | $v = 2^{12}$ | $v = 2^{14}$ |
|---|---|---|---|
| Min Heap | $3.03 \cdot 10^{-8}$ | $2.89 \cdot 10^{-8}$ | $3.86 \cdot 10^{-8}$ |
| Fibonacci Heap | $3.48 \cdot 10^{-9}$ | $3.63 \cdot 10^{-9}$ | $4.31 \cdot 10^{-9}$ |

Tabla 3.1: Pendiente obtenida para los ajustes lineales

En la tabla anterior se puede apreciar que para los tres valores de $v$ las pendientes asociadas al ajuste lineal del Fibonacci Heap son aproximadamente 1 orden de magnitud menor que las del Min Heap.

# 4. Análisis

De acuerdo a los resultados obtenidos, se puede afirmar que la estructura mas eficiente con respecto al tiempo es la cola de Fibonacci. Los valores mostrados en la tabla 3.1 pueden interpretarse como la cantidad promedio de segundos adicionales que demora el algoritmo cuando el número de vértices del grafo aumenta en 1, en otras palabras, estamos obteniendo el tiempo amortizado del algoritmo.

Con respecto a que Fibonacci sea mas eficiente esto es dado que en el algoritmo de Dijkstra se tienen 2 operaciones claves las cuales son:

1. Decrease key: Actualizar la distancia de un nodo en la estructura de datos cuando se encuentra un camino más corto.

2. Extraer el mínimo: Obtener y eliminar el nodo con la menor distancia desde el conjunto de nodos no visitados.

De estas 2 operaciones podemos notar que cola de Fibonacci destaca en sus costes en comparación con Heap

| | Heap | Cola de Fibonacci |
|---|---|---|
| Decrease key | O(log n) | O(1) amortizado |
| Extraer el mínimo | O(log n) | O(log n) amortizado |

Tabla 4.1: Comparación costes

Ahora si aplicamos esto en el contexto del algoritmo de Dijkstra en donde se aplica una operación de extracción del mínimo para cada nodo en el grafo y una operación de disminución de clave para cada arista. Si V es el número de nodos y E es el número de aristas.

| | Heap | Cola de Fibonacci |
|---|---|---|
| Decrease key | O(ElogV) | O(E) amortizado |
| Extraer el mínimo | O(VlogV) | O(VlogV) amortizado |

Tabla 4.2: Comparación costes en Dijkstra

Con esto podemos afirmar que nuestros resultados son correctos, ya que para grafos en donde el numero de aristas sea mayor que el numero de nodos el costo amortizado de O(1) en colas de Fibonacci significa una optimización importante para el algoritmo.

# 5. Conclusión

Luego de realizar la implementación de dos estructuras para ser usadas como colas de prioridad y la implementación de Dijkstra haciendo uso de estas, analizando el tiempo y complejidad de, podemos concluir que para grafos mas densos la estructura optima a utilizar como cola de prioridad es la cola de Fibonacci

Recapitulando, se ha observado que la estructura Cola de Fibonacci destaca por su rapidez en la acción de decrease key, mientras que la estructura Heap logra igualar a la anterior en la extracción del mínimo, tomando la misma complejidad temporal sin la necesidad de considerar un coste amortizado.

Como conclusión podemos decir que aunque las colas de Fibonacci sean mas difíciles de implementar que un Heap valdrá totalmente la pena, dado que se demostró que resultan ser mas eficientes en especial en grafos densos.

Finalmente este hallazgo apoya la premisa inicial de nuestra hipótesis, que predijo que las colas de Fibonacci serian la estructura mas óptima para Dijkstra. Por lo tanto, podemos concluir con confianza que nuestra hipótesis se cumplió en este estudio debido a que tanto los resultados obtenidos de manera experimental como teórica afirman lo predicho.