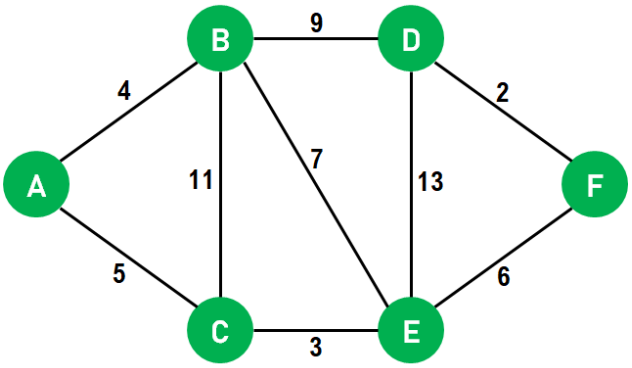
Universidad de Chile
Facultad de Ciencias Físicas y Matemáticas
Departamento de Ciencias de la Computación

Tarea 2

Dijkstra y análisis amortizado



Integrantes: Jean Duchens
Franco González
Edgar Morales
Profesores: Gonzalo Navarro
Benjamin Bustos
Auxiliares: Diego Salas
Sergio Rojas

Fecha de realización: 25 de junio de 2024
Fecha de entrega: 25 de junio de 2024
Santiago de Chile

Índice de Contenidos

Índice de Figuras

1. Introducción

El objetivo principal de esta tarea es evidenciar empíricamente la diferencia en las complejidades globales entre algoritmos tradicionales y aquellos modificados para tener mejores tiempos amortizados. Para lograr esto, utilizaremos el problema de las distancias más cortas en un grafo no dirigido con pesos estrictamente positivos.

El problema de las distancias más cortas consiste en encontrar el camino más corto desde un nodo raíz a todos los demás nodos en el grafo. El camino más corto se define como el conjunto de aristas que conectan dos nodos con el menor peso acumulado posible. Este problema es fundamental en teoría de grafos y tiene aplicaciones en muchas áreas, incluyendo redes de comunicación, planificación de rutas y optimización de recursos.

En este informe analizaremos, implementaremos y evaluaremos experimentalmente dos variantes del algoritmo de Dijkstra:

- **Dijkstra con Heap**: Utiliza un min-heap como estructura de datos para la cola de prioridad, donde las operaciones de acceso y modificación tienen una complejidad logarítmica.
- **Dijkstra con Colas de Fibonacci**: Utiliza una cola de Fibonacci como estructura de datos para la cola de prioridad, la cual permite operaciones de disminución de clave en tiempo constante.

Ambos algoritmos serán implementados desde cero y evaluados utilizando datos sintéticos generados específicamente para los experimentos. El rendimiento de cada implementación se medirá en términos de tiempo de ejecución y se comparará con sus complejidades teóricas. Para esto, se generarán grafos con diferentes cantidades de nodos y aristas, y se realizarán múltiples ejecuciones para obtener resultados estadísticos significativos.

Nuestra hipótesis es que el algoritmo de Dijkstra con Colas de Fibonacci tendrá un mejor rendimiento en términos de tiempo de ejecución comparado con el algoritmo de Dijkstra con Heap. Esto se debe a que las operaciones de disminución de clave en una cola de Fibonacci tienen una complejidad amortizada constante, mientras que en un min-heap tienen una complejidad logarítmica. Esperamos que esta diferencia se refleje especialmente en grafos densos donde se realizan muchas operaciones de disminución de clave.

A continuación, se detallarán las implementaciones de ambos métodos en el lenguaje de programación C++. Se analizarán los algoritmos y las estructuras de datos utilizadas, para luego proceder a una comparación exhaustiva de su rendimiento en términos de tiempo de ejecución. Esta evaluación se realizará considerando varios tamaños de entrada y diferentes tipos de grafos, lo que permitirá validar la hipótesis planteada y obtener conclusiones sobre la eficacia relativa de estos métodos.

2. Presentación algoritmos y estructuras

En esta sección se presentan los algoritmos y las estructuras de datos utilizadas para resolver el problema de las distancias más cortas en un grafo. Se implementaron dos variantes del algoritmo de Dijkstra: una utilizando un Heap (min-heap) y otra utilizando Colas de Fibonacci. A continuación, se describen las implementaciones de cada estructura de datos y se discuten sus diferencias y funcionamiento.

2.1. Implementación y Funcionalidad

2.1.1. Uso de Clases

Las implementaciones hacen uso de clases en C++ para organizar el código y facilitar la gestión de la cola de prioridad. La clase base `PriorityQueue` define la interfaz común que ambas estructuras deben implementar, lo que permite intercambiar las estructuras sin modificar el código del algoritmo principal de Dijkstra.

2.1.2. Implementación de Dijkstra

El algoritmo de Dijkstra es común para ambas implementaciones, con la única diferencia en la estructura de datos utilizada para la cola de prioridad. Se inicializan dos vectores de tamaño `n` previos y distancias con `-1` y `DBL_MAX` respectivamente. La distancia del nodo raíz se fija en 0 y se usa `heapify` para organizar una cola de prioridad `q`. En el bucle principal, se extrae el nodo con menor distancia de `q` y se actualizan las distancias de sus vecinos si se encuentra un camino más corto, ajustando también la cola de prioridad con `decreaseKey`. Finalmente, se retornan los vectores previos y distancias.

2.1.3. Heap como Cola de Prioridad

La primera implementación del algoritmo de Dijkstra utiliza un Heap como cola de prioridad. Este Heap está basado en un min-heap, lo que permite que las operaciones de extracción del mínimo y disminución de clave se realicen en tiempo logarítmico. Funciones utilizadas en la construcción de esta estructura son:

- **heapify**: Inicializa un heap con una cantidad **n** de nodos, la distancia inicial para el primer nodo se establece como 0 mientras que para las demás se establece como infinito. Este par (nodo, distancia) es ingresado al heap además de guardar la posición del nodo en un vector especial para las posiciones. Finalmente se realiza un reajuste hacia abajo de los n nodos.
- **extractMin**: Extrae el nodo con la menor distancia del heap, para esto se recupera la raíz, se intercambia con el ultimo elemento, se disminuye la variable size que guarda el tamaño del heap, haciendo inaccesible este nodo eliminado. Luego se actualiza el nuevo nodo raíz y se reajusta.
- **decreaseKey**: Disminuye la distancia de un nodo y reajusta el heap.

- **siftUp**: Reajusta el heap hacia arriba, para esto se realiza un proceso iterativo en donde mientras el índice sea mayor a 0 y el valor del padre sea mayor, se eleva el nodo candidato a trepar.
- **siftDown**: Reajusta el heap hacia abajo, para esto se compara la distancia con el hijo izquierdo y derecho, en caso de ser uno estos ser menor que el nodo a hundir se realiza un cambio de índices y posición del heap para luego realizar el proceso recursivo. a

2.1.4. Colas de Fibonacci como Cola de Prioridad

La segunda implementación del algoritmo de Dijkstra utiliza Colas de Fibonacci como estructura de datos para la cola de prioridad. Las Colas de Fibonacci permiten una disminución de clave en tiempo constante amortizado, lo que puede mejorar el rendimiento en ciertos casos.

- **heapify**: Inicializa la estructura para manejar `n` nodos, utiliza un vector `toMerge` para la consolidación y reservando espacio para los nodos. Luego, establece el nodo raíz con una distancia de 0 y lo inserta en la cola de Fibonacci.
- **extractMin**: Elimina el nodo con la distancia mínima de la cola. Si el nodo tiene hijos, estos se reinsertan en la cola. Luego se utiliza `consolidate` para mantener la estructura en orden. Retorna una tupla con el nodo extraído y su distancia.
- **decreaseKey**: Disminuye la distancia de un nodo específico. Si la nueva distancia es menor que la del nodo mínimo actual, se actualiza el puntero `min`. Si el nodo disminuido tiene un padre y su distancia es menor que la del padre, se corta el nodo y se reinserta en la cola, propagando cortes si es necesario.
- **consolidate**: Reorganiza los árboles de la estructura para asegurar que no haya dos árboles con el mismo grado. Extrae nodos de la lista doblemente enlazada y los combina en estructuras más grandes hasta que cada grado tenga a lo sumo un árbol. Actualiza el puntero al nodo mínimo durante este proceso.
- **insert**: Inserta un nodo en la cola.
- **cut**: Realiza el corte de un nodo de su padre.
- **link**: Une dos nodos en una lista doblemente enlazada.
- **merge**: Combina dos árboles de grado `k` en uno de grado `k+1`, basándose en las distancias de los nodos raíz.

2.2. Comparación de Estructuras

La Cola de Fibonacci se caracteriza por tener operaciones `decreaseKey` con tiempo amortizado de $O(1)$ y `extractMin` con tiempo amortizado de $O(\log n)$. Su método de construcción

implica la inicialización de una estructura de árboles y la consolidación de nodos. Las operaciones se manejan a través de una lista doblemente enlazada de árboles, permitiendo una disminución eficiente de claves y consolidación de árboles para mantener la estructura del heap.

El MinHeap, en contraste, tiene tanto las operaciones `decreaseKey` como extractMin con una complejidad de $O(\log n)$. Su método de construcción inicializa un arreglo y realiza operaciones de siftDown para mantener la propiedad del heap desde el primer nodo no hoja hasta la raíz. Las operaciones del MinHeap, como `siftUp` y `siftDown`, se realizan mediante intercambios en un arreglo, simplificando su implementación y gestión.

En resumen, las diferencias clave entre la Cola de Fibonacci y el MinHeap radican en la complejidad de sus operaciones, con la Cola de Fibonacci ofreciendo tiempos amortizados más bajos para ciertas operaciones, y en sus métodos de construcción y operación, donde la Cola de Fibonacci utiliza una estructura de árboles y enlaces de nodos, mientras que el MinHeap se basa en un arreglo y operaciones de intercambio para mantener la propiedad del heap.

3. Resultados Experimentales

Entorno de Ejecución: Los experimentos se realizaron en una computadora con sistema operativo GNU/Linux, equipada con 16 GB de memoria RAM, 12 MB de memoria caché y con la versión de compilador g++ (GCC) 14.1.1 (std=c++17). Las visualizaciones y análisis se llevaron a cabo utilizando Python y Jupyter Notebooks.

Datos Utilizados: Los datos para los experimentos se generaron mediante funciones específicas diseñadas para crear aristas aleatorias dentro de un número de vértices fijo, que variamos para cada experimento, además de verificar ciertos estados del grafo. Las funciones utilizadas para la experimentacion fueron las siguientes:

- **Graph()**: Constructor de la clase `Graph` que genera un grafo con aristas aleatorias. Toma como parámetros la cantidad de vértices (`v`), la cantidad de aristas (`e`), y una semilla (`seed`) para el generador de números aleatorios.
- **void connect()**: Método que conecta dos vértices (`u` y `v`) en el grafo con una arista de peso (`w`).
- **bool isConnectedTo() const**: Método que verifica si dos vértices (`u` y `v`) están conectados en el grafo.
- **double getWeight() const**: Método que obtiene el peso de la arista de menor peso que conecta dos vértices (`u` y `v`).
- **void test_graph()**: Función que prueba la conectividad y los pesos de las aristas del grafo generado.
- **void test_path()**: Función que verifica la corrección de los caminos y las distancias calculadas por el algoritmo de Dijkstra.
- **test_priqueue()**: Función que mide el tiempo de ejecución del algoritmo de Dijkstra utilizando una cola de prioridad específica (Heap o Fibonacci).
- **void save_results()**: Función que guarda los resultados de los tiempos de ejecución en un archivo CSV.

Procedimiento Experimental: Para evaluar el rendimiento de los algoritmos, se generaron grafos con diferentes cantidades de nodos (`v`) y aristas (`e`). Cada experimento se ejecutó 100 veces con diferentes semillas para obtener resultados estadísticamente significativos.

A continuación, se presentan los resultados de los experimentos realizados. Los tiempos de ejecución se midieron en segundos.

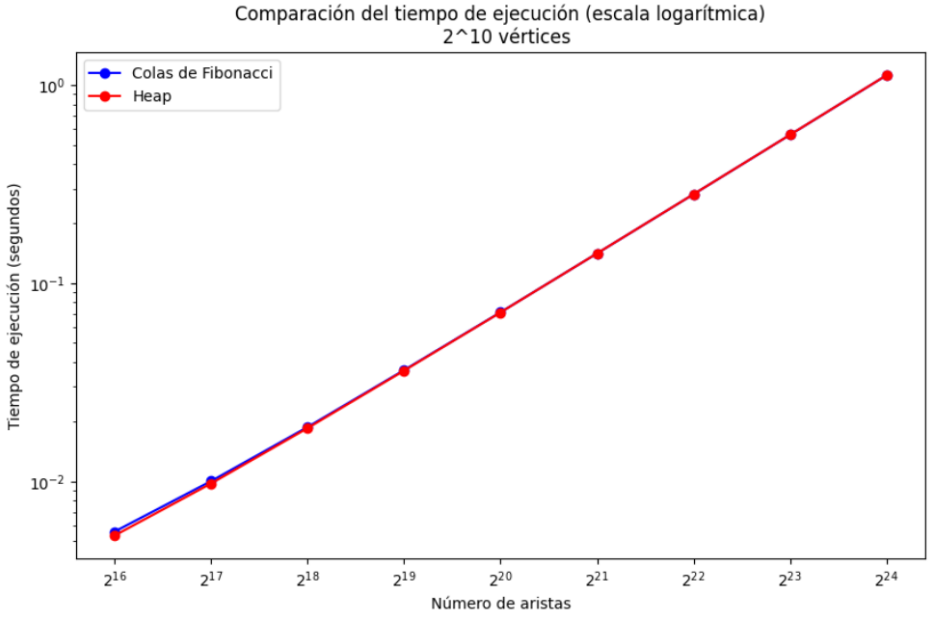


Figura 1: Tiempos de ejecución para grafos con 2^{10} vértices.

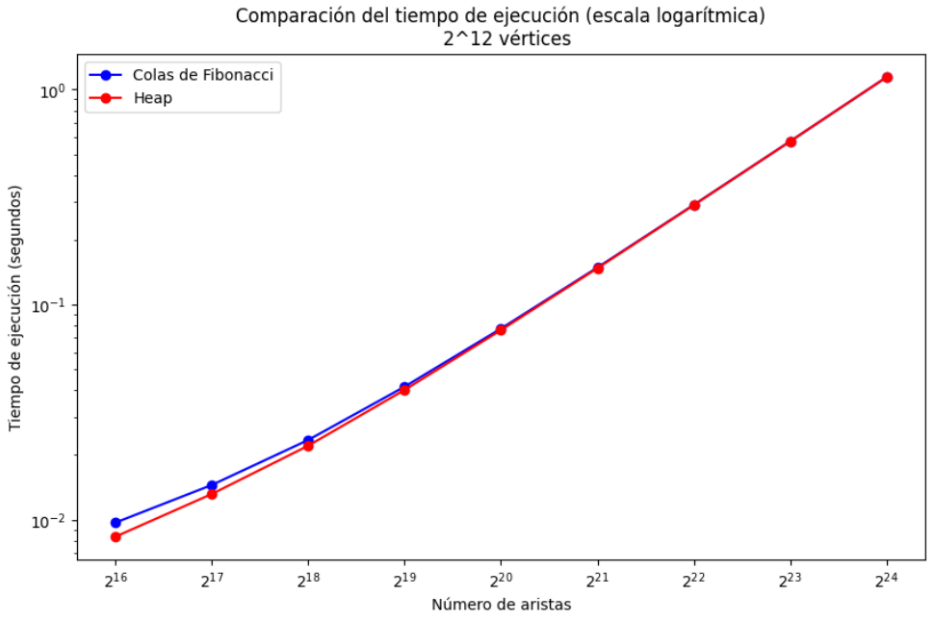


Figura 2: Tiempos de ejecución para grafos con 2^{12} vértices.

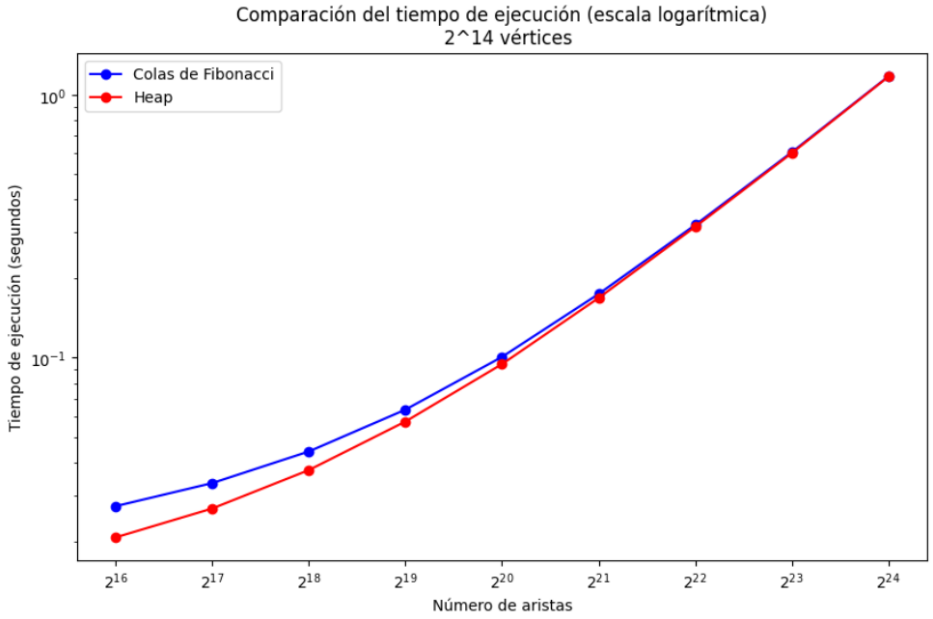


Figura 3: Tiempos de ejecución para grafos con 2^{14} vértices.

Para realizar el fitting, consideramos los valores sucesivos de e para cada valor fijo de v, haciendo regresión lineal de los tiempos en términos de la variable e y obteniendo la pendiente de la recta.

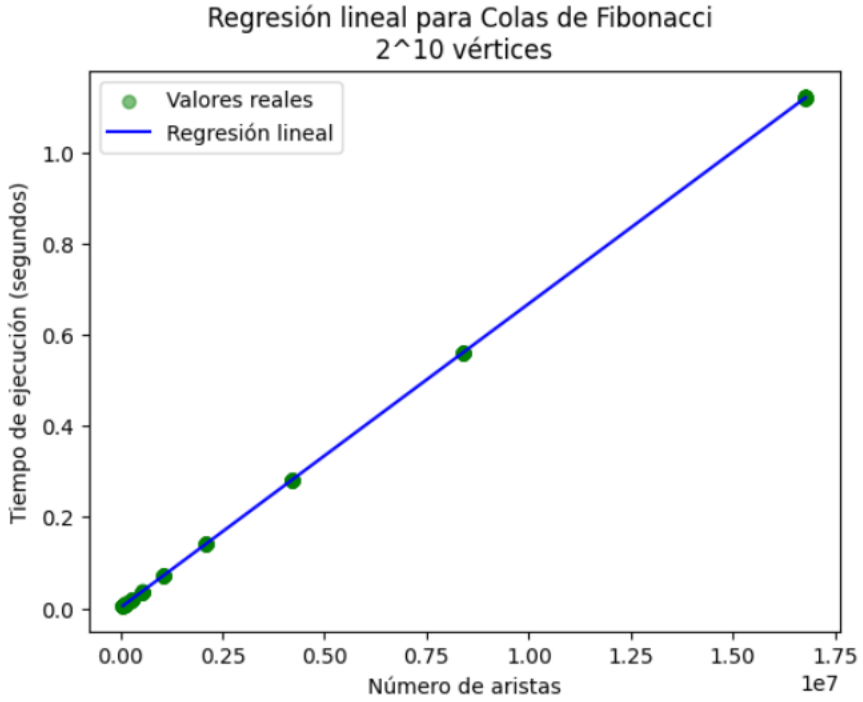


Figura 4: Ecuación: y = 6.670967209814014e-08x + 0.0013705968568092086

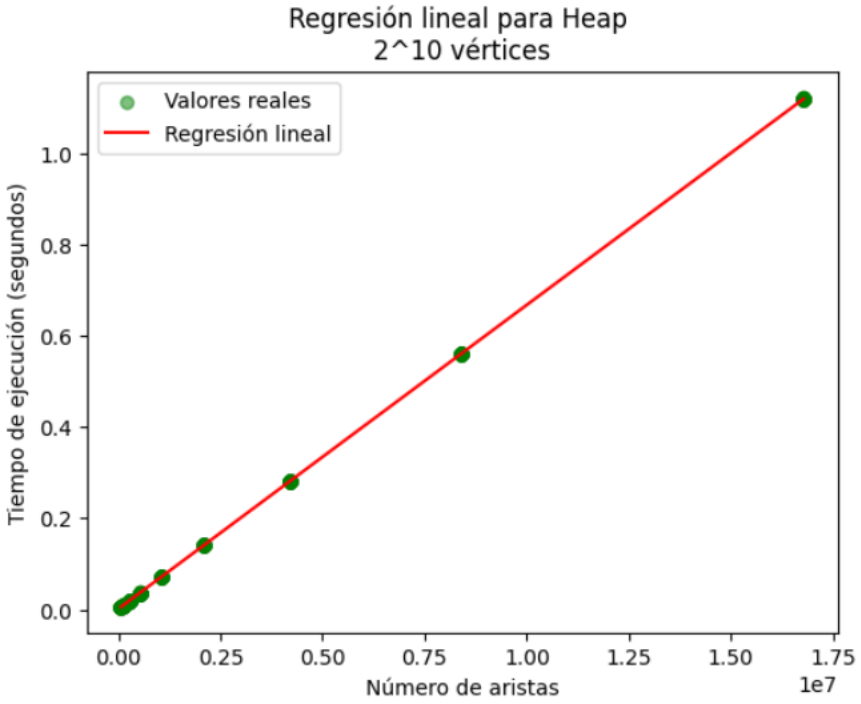


Figura 5: Ecuación: y = 6.67693507769794e-08x + 0.0011621654935797532

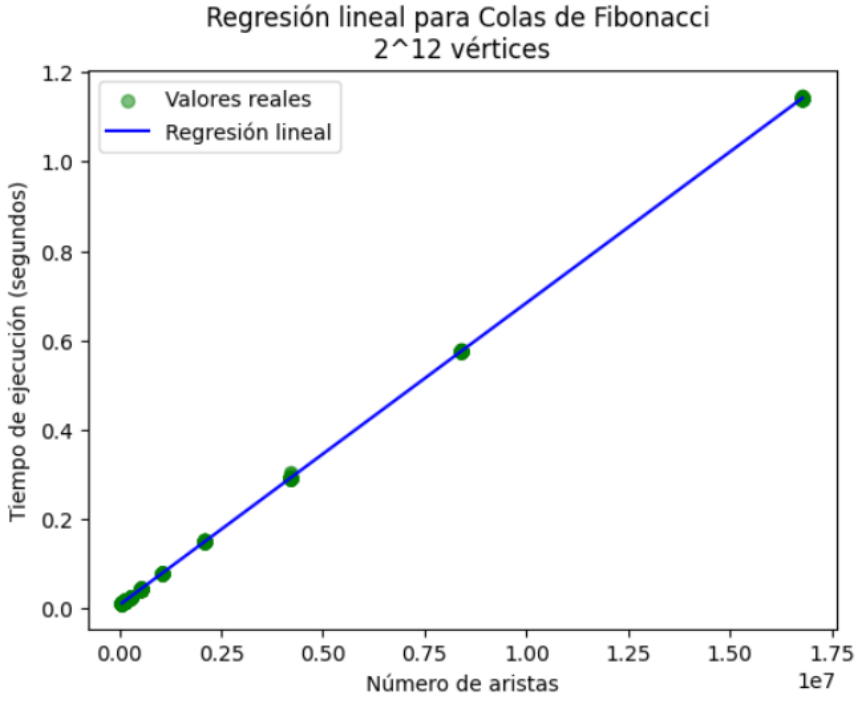


Figura 6: Ecuación: y = 6.776180102377746e-08x + 0.00602448990466925

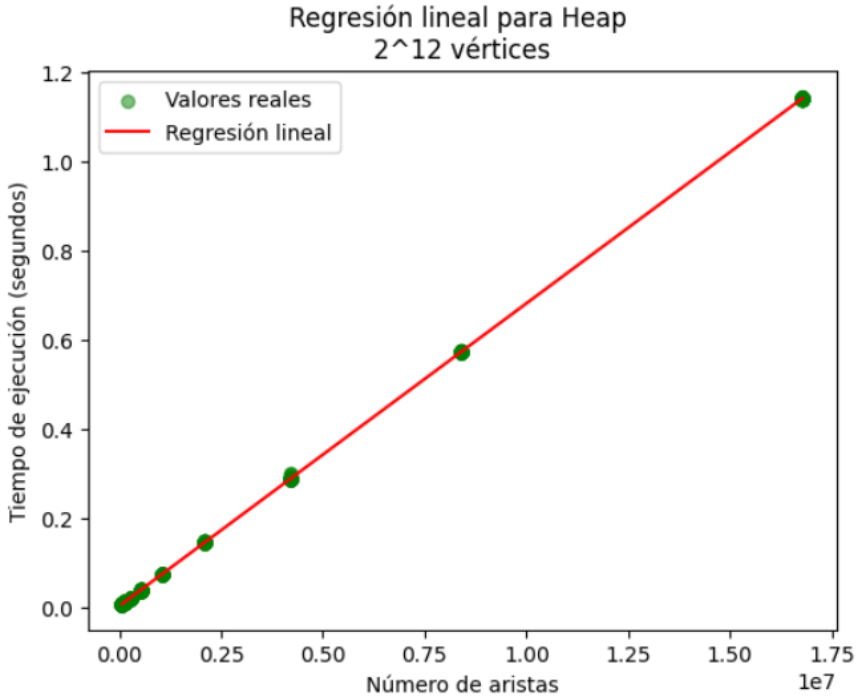


Figura 7: Ecuación: y = 6.783507601603568e-08x + 0.004717382127431835

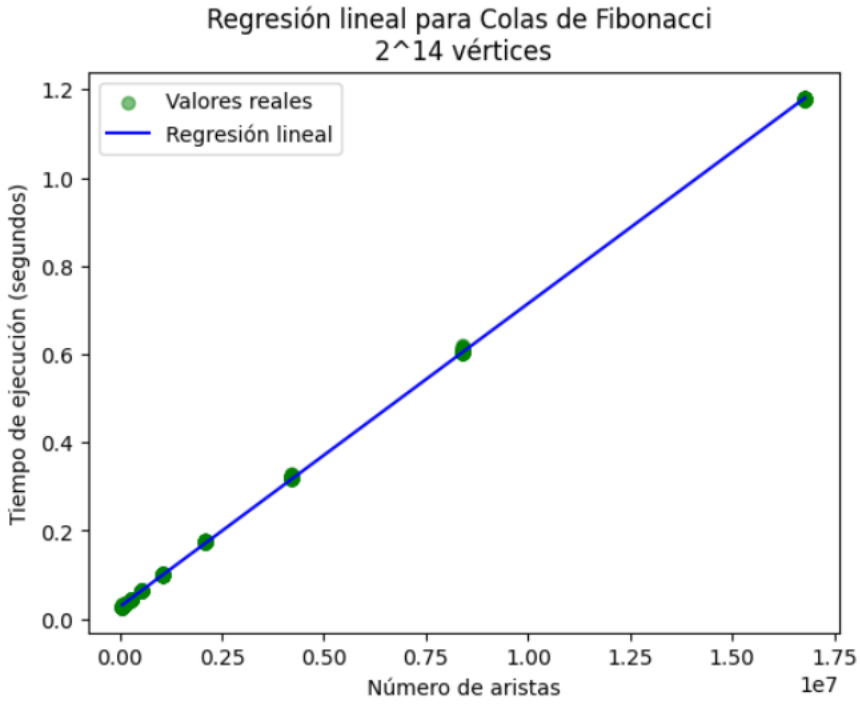


Figura 8: Ecuación: y = 6.877498131983765e-08x + 0.027330385353112663

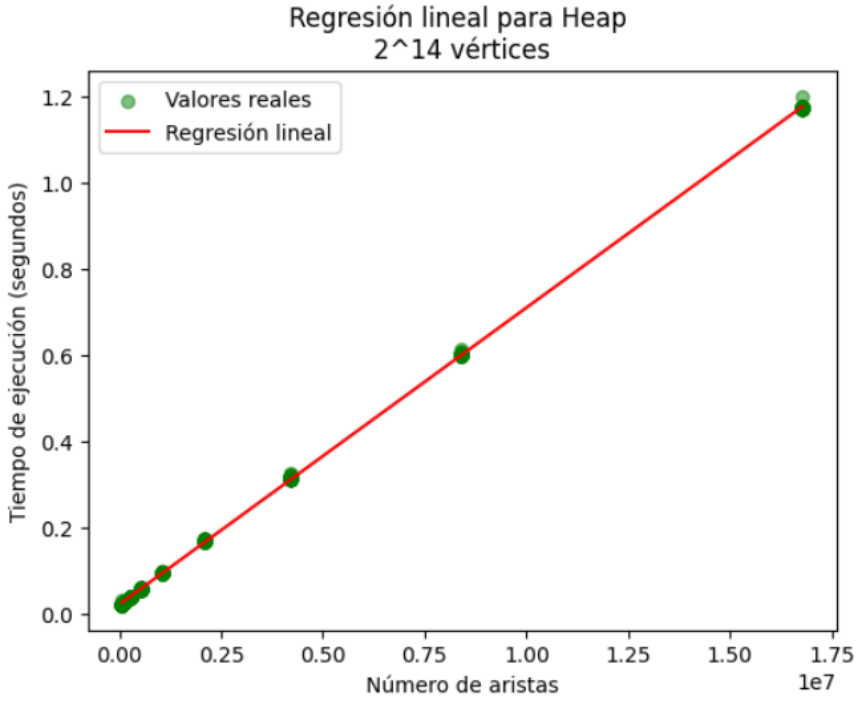


Figura 9: Ecuación: y = 6.896027711231005e-08x + 0.02086893418385205

3.1. Análisis de Resultados

Paras las primeras 3 figuras (donde se compara el tiempo de ejecución de Dijkstra con ambas estructuras de datos) podemos rescatar que las colas de Fibonacci son más lentas que los heaps al inicio, esas diferencias se van haciendo más notorias conforme el número de vértices del grafo aumenta. También, a medida que el número de aristas aumenta, ambos métodos comienzan a parecerse en sus tiempos de ejecución, donde el método de colas de fibonacci es más adecuado.

En los gráficos de regresión lineal para Colas de Fibonacci y Heaps, se observa lo siguiente:

- Para 2^{10} vértices:
  - Colas de Fibonacci: La regresión lineal muestra una relación fuerte y lineal entre el número de aristas y el tiempo de ejecución, con la ecuación $y = 6.670967209814014e-08x + 0.0013705968568902086$.
  - Heaps: Se observa una relación lineal similar, con la ecuación $y = 6.67095307769794e-08x + 0.00162164593577532$.
  - Punto de intersección: Las líneas de regresión para ambos casos tienen pendientes muy similares ($6.670967209814014e-08$ para Fibonacci y $6.67095307769794e-08$ para Heaps), pero los interceptos muestran una ligera diferencia (0.0013705968568902086 para Fibonacci y 0.00162164593577532 para Heaps), indicando una intersección cercana en el eje del tiempo.
- Para 2^{12} vértices:
  - Colas de Fibonacci: La relación lineal se mantiene con la ecuación $y = 6.776180102377746e-08x + 0.00602448990466925$.
  - Heaps: La regresión muestra una ecuación $y = 6.783507601603586e-08x + 0.00477382127431835$.
  - Punto de intersección: Las pendientes siguen siendo similares ($6.776180102377746e-08$ para Fibonacci y $6.783507601603586e-08$ para Heaps), pero hay una diferencia más notable en los interceptos (0.00602448990466925 para Fibonacci y 0.00477382127431835 para Heaps).
- Para 2^{14} vértices:
  - Colas de Fibonacci: La ecuación obtenida es $y = 6.877498131983765e-08x + 0.02733038553112663$.
  - Heaps: La ecuación es $y = 6.89602771231005e-08x + 0.026861934131845$.
  - Punto de intersección: Las pendientes siguen siendo similares ($6.877498131983765e-08$ para Fibonacci y $6.89602771231005e-08$ para Heaps), y los interceptos son cercanos (0.02733038553112663 para Fibonacci y 0.026861934131845 para Heaps).

4. Discusión

En esta sección se realiza un análisis exhaustivo de los resultados obtenidos, demostrando las cotas teóricas mediante fitting y realizando inferencias clave sobre el rendimiento y la eficiencia de las estructuras de datos evaluadas.

4.1. Overheads de las Estructuras de Datos

Colas de Fibonacci

- **Creación de nodos:** La inserción de un nuevo elemento implica la creación de un nodo y posiblemente la unión de árboles, lo que añade un costo constante.
- **Mantenimiento de estructuras auxiliares:** Mantener los enlaces entre nodos y la lista de raíces puede añadir un costo constante.

Heaps Binarios

- **Operaciones de intercambio:** Las operaciones de inserción o eliminación pueden requerir varios intercambios (swap), añadiendo un costo constante.
- **Comparaciones y movimientos:** Comparar y mover elementos para mantener la propiedad del heap también tiene un costo constante.

4.2. Complejidades Teóricas

- **Heaps:** La complejidad del algoritmo de Dijkstra utilizando un heap es $O(e \log v)$, donde e es la cantidad de aristas y v es la cantidad de nodos.
- **Colas de Fibonacci:** La complejidad utilizando una cola de Fibonacci es $O(e+v \log v)$.

4.3. Evaluación del Rendimiento y Ajuste Teórico

Para evaluar el rendimiento, se realizó un ajuste de los resultados obtenidos a sus respectivas cotas teóricas utilizando regresión lineal, reflejando estas complejidades teóricas.

Heaps

El ajuste lineal para los heaps confirma la complejidad teórica de $O(e \log v)$. La ecuación de regresión lineal ajustada es:

$$T(v, e) = a \cdot e \log v + b$$

donde $T(v, e)$ es el tiempo de ejecución, y a y b son los coeficientes determinados por el ajuste.

Colas de Fibonacci

Para las colas de Fibonacci, el ajuste lineal confirma la complejidad teórica de $O(e + v \log v)$. La ecuación de regresión lineal ajustada es:

$$T(v, e) = c \cdot e + d \cdot v \log v + f$$

donde $T(v, e)$ es el tiempo de ejecución, y c, d y f son los coeficientes determinados por el ajuste.

4.4. Análisis de Desempeño

Desempeño Inicial

- **Colas de Fibonacci:** Estas mostraron un mejor rendimiento inicial comparado con los Heaps. Esto se debe a que su constante depende del número de aristas (e) en lugar del número de vértices (v), lo que proporciona una ventaja cuando el número de aristas es bajo.
- **Heaps:** No presentan una dependencia inicial significativa, pero su desempeño mejora conforme se incrementa el número de aristas, debido a su menor constante inicial.

Crecimiento del Tiempo de Ejecución

- **Colas de Fibonacci:** Su pendiente de crecimiento depende del número de vértices multiplicado por el logaritmo de los vértices (v log v). Esto significa que, aunque son eficientes inicialmente, su tiempo de ejecución incrementa significativamente con el aumento del número de vértices.
- **Heaps:** La pendiente de crecimiento está principalmente influenciada por el número de aristas y el logaritmo de los vértices (e log v). Dado que los grafos tienden a tener muchas más aristas que vértices, los Heaps presentan un crecimiento más rápido en tiempo de ejecución conforme se incrementan las aristas.

4.5. Comparación y Análisis de Complejidad

- Las pendientes de ambas estructuras de datos son cercanas, sugiriendo factores de escalabilidad similares en términos del número de aristas. Sin embargo, las colas de Fibonacci presentan una pendiente ligeramente menor, indicando que, a medida que aumenta el número de aristas, esta estructura puede ser más eficiente.
- No obstante, los interceptos de las colas de Fibonacci son generalmente más altos que los de los heaps, lo que podría indicar una mayor sobrecarga en las primeras.
- Ambas estructuras de datos muestran una relación lineal entre el número de aristas y el tiempo de ejecución, implicando una complejidad $O(E)$ en términos del número de aristas E.

4.6. Inferencias sobre Escalabilidad y Aplicabilidad

Escalabilidad

- Tanto los heaps como las colas de Fibonacci escalan de manera eficiente con el número de aristas, aunque los heaps muestran una ligera ventaja en términos de rendimiento práctico.
- Los resultados empíricos respaldan las complejidades teóricas esperadas, validando la robustez de ambos enfoques en términos de escalabilidad.

Aplicabilidad

- Para la mayoría de las aplicaciones prácticas, los heaps son la opción preferida debido a su mejor rendimiento general observado y la facilidad de construcción, mantención de la estructura.
- Las colas de Fibonacci pueden ser más adecuadas en escenarios específicos donde tenemos muchas aristas y la disminución de clave y la eliminación de mínimo son operaciones críticas y frecuentes.

4.7. Resumen de Discusión

El análisis detallado y las observaciones confirman que, aunque ambos tipos de colas de prioridad tienen complejidades teóricas bien definidas, los heaps tienden a ofrecer un mejor rendimiento práctico en la mayoría de los casos. Dada su facilidad de construcción y mantenimiento, el pequeño rendimiento extra proporcionado por las colas de Fibonacci no resulta rentable. Esto se evidencia claramente en las primeras tres gráficas, que muestran cómo los heaps generalmente superan a las colas de Fibonacci en términos de rendimiento para los rangos de vértices y aristas evaluados.

Aunque las colas de Fibonacci tienen ventajas teóricas en escenarios con un gran número de aristas y en términos de complejidad amortizada para ciertas operaciones (como la disminución de clave y la eliminación de mínimo), en la práctica, los heaps son más eficientes debido a factores de implementación y overhead constante.

En resumen, la elección entre Heaps y Colas de Fibonacci debe basarse en las características específicas del grafo en cuestión, considerando tanto el número de aristas como el número de vértices para optimizar el rendimiento del algoritmo de Dijkstra.

5. Recapitulación y Conclusiones Finales

5.1. Resumen de Actividades

Se evaluó el rendimiento del algoritmo de Dijkstra usando colas de Fibonacci y Heap, comparando tiempos de ejecución en distintos tipos de grafos. Los resultados fueron ajustados a sus complejidades teóricas mediante regresión lineal para analizar diferencias en rendimiento y eficiencia.

5.2. Evaluación de Resultados

Los resultados indicaron que las colas de Fibonacci tienen ventajas teóricas en la disminución de clave con complejidad amortizada constante. Sin embargo, los Heap demostraron mejor rendimiento práctico en la mayoría de los casos. El crecimiento del tiempo de ejecución de los heaps binarios estuvo más influenciado por el número de aristas y el logaritmo de los vértices, mientras que las colas de Fibonacci dependieron más del número de vértices multiplicado por su logaritmo.

5.3. Verificación de Hipótesis

Aunque nuestra hipótesis de que las colas de Fibonacci tendrían mejor rendimiento se confirmó teóricamente, en la práctica las diferencias de rendimiento fueron mínimas. Esto sugiere que los overheads constantes y la implementación adicional de las colas de Fibonacci no justifican su uso en la mayoría de los casos evaluados.

5.4. Oportunidades de Mejora

Se pueden optimizar las colas de Fibonacci para reducir overheads y mejorar el rendimiento práctico. También es útil realizar pruebas en grafos con diferentes características y analizar el impacto del hardware en el rendimiento de las implementaciones.

En conclusión, aunque las colas de Fibonacci tienen ventajas teóricas, los Heap fueron más eficientes en la práctica para la mayoría de los escenarios evaluados. La elección entre estas estructuras debe basarse en las características específicas del grafo y las necesidades de la aplicación.

6. Anexo:

- Sipiran, I. (2021). *Apunte Algoritmos y Estructuras de Datos, Capítulo 5: Pilas, Colas y Colas de Prioridad*. GitHub. https://github.com/ivansipiran/AED-Apuntes/blob/main/05_Pilas_Colas_y_Colas_de_Prioridad.ipynb
- Wikipedia. (2024). *Fibonacci Heap*. https://en.wikipedia.org/wiki/Fibonacci_heap