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

Tarea 2

Dijkstra y análisis amortizado

Integrantes: Joaquín González
Lucas Llort
Franco Navarro
Profesor: Benjamín Bustos
Sección: 1
Santiago de Chile

1. Introducción

En este informe se abordará la implementación de dos variantes del algoritmo de Dijkstra utilizando estructuras de datos avanzadas como Heaps y Colas de Fibonacci, en el contexto de la resolución de problemas de búsqueda de caminos mínimos en grafos. El objetivo principal es analizar y comparar el rendimiento de ambas implementaciones frente a conjuntos de datos de diversos tamaños y características.

Para garantizar la correcta implementación y rendimiento de los algoritmos, se diseñaron pruebas detalladas que cubren aspectos como la correcta generación de grafos y la eficiencia del algoritmo de Dijkstra.

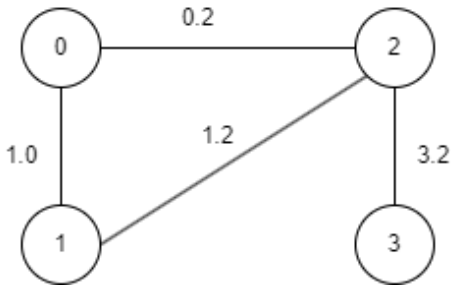
En el cuerpo de este informe se proporcionará una descripción detallada de cómo se implementó el algoritmo utilizando Heaps y Colas de Fibonacci, así como el proceso seguido para llevar a cabo las pruebas experimentales. Finalmente, se presentarán los resultados obtenidos y las conclusiones derivadas de estos análisis.

La hipótesis inicial es que la implementación del algoritmo con Colas de Fibonacci será más eficiente que con Heaps, esto debido a que Fibonacci permite realizar operaciones de inserción y reducción de llaves de manera lineal, lo que puede resultar en un mejor rendimiento general en Dijkstra.

2. Desarrollo

Con el fin de poder comprender correctamente la manera en la que se implementaron Dijkstra junto con Heap y cola de Fibonacci es necesario primero entender la manera en la que se concibió el grafo que se entrega al algoritmo de Dijkstra. Las columnas del grafo contienen las conexiones de cada i-esimo nodo con otros nodos y esta conexión esta dada por los pares, siendo unsigned int el indice del otro nodo al que esta conectado el nodo y double la distancia que hay entre los dos nodos.

Para que se entienda mejor la implementación de grafos, a continuación se mostrará de ejemplo el como representar uno, aportando el código necesario y la matriz de adyacencia.



(a) Grafo de ejemplo

Código 1: Ejemplo creación de grafo

```
int main() {
    Grafo grafo(4);

    unir_nodos(grafo, 0, 1, 1.0);
    unir_nodos(grafo, 0, 2, 0.2);
    unir_nodos(grafo, 1, 2, 1.2);
    unir_nodos(grafo, 2, 3, 3.2);

    return 0;
}
```

- Matriz de Adyacencia
  - Nodo 0: (1, 1.0) (2, 0.2)
  - Nodo 1: (0, 1.0) (2, 1.2)
  - Nodo 2: (0, 0.2) (1, 1.2) (3, 3.2)
  - Nodo 3: (2, 3.2)

Con esto claro, se procedera a explicar la manera en la que se implementaron las estructuras Heap y Fibonacci.

Para la implementación de la estructura Heap, se siguió una estructura basada en la del apunte del curso CC3001 Algoritmos y Estructuras de Datos. Esta estructura contiene la clase interna Nodo, que contiene como variable la llave que representa la distancia óptima y el índice del nodo dentro del grafo. A continuación se detallan ambas clases, sus métodos y atributos.

En primer lugar, para el Heap se implemento tal que:

1. **Métodos Públicos**
   a) `agregarPar(llave_t llave, valor_t valor)`: Se agrega un nodo a la estructura heap que contiene la distancia y el índice que representa la posición de ese nodo en la estructura grafo.
   b) `llaveMenor()`: Devuelve la distancia del nodo que tiene la distancia mas baja dentro del heap.
   c) `valorMenor()`: Devuelve el indice del nodo dentro del grafo que tiene la distancia mas baja dentro del heap.
   d) `vacio()`: Devuelve 'true' si el Heap está vacío.
   e) `extraerMinimo()`: Devuelve el índice del nodo en el grafo tiene la menor distancia y elimina el nodo del heap.
   f) `reducirLlave(Nodo* nodo, llave_t llave)`: Actualiza el valor de la distancia del nodo explicitado en el argumento y se realiza los respectivos cambios de posiciones en el Heap.
2. **Métodos Privados**
   a) `intercambiar(unsigned int i, unsigned int j)`: Intercambia entre si a los nodos en las posiciones i y j del vector '_heap'.
   b) `subir(unsigned int indice)`: Hace que escale el nodo dentro del Heap acorde a la distancia que tenga.
   c) `bajar(unsigned int indice)`: Hace que baje el nodo dentro del Heap acorde a la distancia que tenga.
3. **Datos Privados**
   a) `_heap`: Vector que almacena los nodos del heap.

La clase Heap tiene una clase interna llamada "Nodo", esta corresponde a cada uno de los pares (llave, dato) que se insertan al Heap, la clase se llama "Nodo.ª pesar de que la clase

heap utiliza un vector y no un árbol, esto es para que la interfaz de la clase Heap sea la misma que la interfaz de la cola de Fibonacci, la cual si utiliza arboles.

La clase Nodo tiene los siguientes campos:

1. **Métodos Públicos**
   a) `llave()`: Devuelve la distancia óptima del nodo.
   b) `valor()`: Devuelve el índice del nodo en el grafo.
2. **Datos Privados**
   a) `_llave`: Almacena la distancia óptima del nodo.
   b) `_valor`: Almacena el índice del nodo en el grafo.
   c) `_pos`: Almacena el índice del nodo en el vector '_heap'.

Para la implementación de la cola de Fibonacci, seguimos la explicación proporcionada en clase. Para esto, se decidió crear la clase interna 'Nodo' para representar los nodos internos de un árbol binomial. No confundir con la estructura Nodo que se definió para heap. Esta estructura almacena nodos .

La estructura 'Nodo' implementa tanto métodos públicos como privados. Esta decisión permite que el nodo gestione su propio acceso y modificación de manera adecuada, garantizando la encapsularon y el correcto funcionamiento de la cola de Fibonacci.

1. **Métodos Públicos**
   a) `llave()`: Devuelve la distancia óptima del nodo.
   b) `valor()`: Devuelve el indice del nodo en el grafo.
2. **Métodos Privados**
   a) `Nodo(llave_t llave, valor_t valor)`: Constructor del nodo que lo inicializa con una distancia y el índice del nodo en el grafo, y establece los valores iniciales de los otros atributos.
   b) `anadir_hijo(Nodo* nuevo_hijo)`: Añade un nuevo nodo hijo al nodo actual.
3. **Datos Privados**
   a) `_llave`: Almacena la distancia del nodo.
   b) `_valor`: Almacena el índice en el grafo del nodo.
   c) `_marcado`: Indica si el nodo ha perdido un hijo desde la última vez que se convirtió en hijo de otro nodo.
   d) `_orden`: El número de hijos que tiene el nodo.

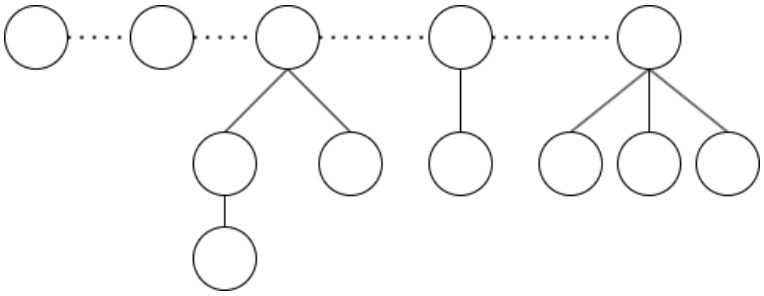
e) `_padre`: Puntero al nodo padre.

f) `_hijo`: Puntero a uno de los hijos del nodo.

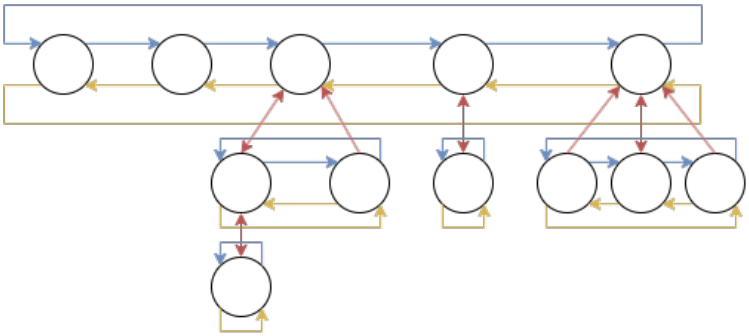
g) `_hermano_i`: Puntero al nodo hermano izquierdo.

h) `_hermano_d`: Puntero al nodo hermano derecho.

Para que los nodos formen una cola de Fibonacci, se tratan de la siguiente manera: se implementan como una lista doblemente enlazada, por lo que cada nodo tiene una referencia a su hermano izquierdo y a su hermano derecho. Esto tiene dos propósitos. Primero, permite identificar cuál es el siguiente árbol en la cola. Segundo, facilita la obtención de todos los hijos de un nodo, ya que un nodo puede tener referencia solo a uno de sus hijos. Esta referencia será a uno de sus hijos de manera arbitraria, por lo que, para acceder a los demás hijos, se deben usar las referencias de sus hermanos. Si un nodo no tiene hermanos, se referencia a sí mismo, y si tiene hermanos en una sola dirección, se referenciará al nodo hermano del otro extremo. Esta implementación facilita la eliminación de nodos, ya que simplemente se deben concatenar sus referencias.



(a) Cola de Fibonacci



(b) Cola de Fibonacci con las referencias de cada nodo, las flechas en azul representan los punteros hermano_i, en amarillo hermano_d y en rojo los padres e hijos

Además de la estructura 'Nodo', la clase ColaFibonacci define métodos públicos y privados que permiten la manipulación y gestión del heap de Fibonacci.

1. **Métodos Públicos**

   a) `ColaFibonacci()`: Constructor de la clase ColaFibonacci.

b) `agregarPar(llave_t llave, valor_t valor)`: Agrega un valor con una distancia al heap y devuelve el indice en el grafo.

c) `llaveMenor() const`: Devuelve la distancia mas baja sin eliminar el nodo del heap.

d) `valorMenor() const`: Devuelve el indice del nodo con la distancia mas baja en la cola.

e) `vacio() const`: Devuelve true si el heap está vacío.

f) `extraerMinimo()`: Devuelve el indice con la distancia mas baja y lo elimina de la estructura.

g) `reducirLlave(Nodo* nodo, llave_t llave)`: Al nodo se le actualiza la distancia con un nueva distancia.

2. **Métodos Privados**

a) `moverRaiz(Nodo* nodo)`: Mueve un nodo a la raíz del heap. Asume que el nodo no es una raíz.

b) `consolidar()`: Reorganiza el heap para mantener su propiedad de Fibonacci.

3. **Datos Privados**

a) `_minimo`: Puntero al nodo con la clave mínima.

b) `_tamano`: Almacena el tamaño del heap.

Ahora bien, respecto al algoritmo de Dijkstra, decidimos implementar los pasos de la siguiente manera:

- Paso 1: Los arreglos son parámetros de la función, por lo tanto, este paso se realiza al ejecutar el proyecto en la experimentación.
- Paso 2: Se recibe una estructura que cumpla con la template de cola. Tanto el Heap como la Cola de Fibonacci son estructuras que cumplen con los requisitos de este paso, por lo tanto, luego se especifican las plantillas para ambas.
- Paso 3: Se inicializa el vector de distancias con el valor máximo de tipo double, indicando que inicialmente no se conoce ninguna distancia a los nodos. Se establece la distancia desde la raíz en 0 y el valor previo de todos los nodos como -1.
- Paso 4: Como se explicó en el paso anterior, aunque no podemos representar cada nodo como infinito, podemos hacerlo utilizando el valor máximo de double. Del mismo modo, utilizamos -1 para identificar que un nodo previo está indefinido. Ambas definiciones ya se hicieron en el paso anterior, por lo que en el resto del código se trabajará con esas suposiciones.

- Paso 5: Se añaden los valores a la cola, guardando los nodos que retorna la función 'Nodo* agregarPar(llave_t llave, valor_t valor)' para utilizarlos luego cuando se necesite reducir sus llaves, la operación *Heapify* se realiza añadiendo los valores uno por uno en un ciclo for, esto tiene complejidad $\mathcal{O}(n)$ para ambos tipos de cola, en el caso de la cola de Fibonacci ver esto es trivial ya que la inserción de un único valor tiene complejidad $\mathcal{O}(1)$, por otro lado, la inserción de un valor en la cola basada en heap tiene complejidad $\mathcal{O}(\log n)$ en el caso general, sin embargo, como se comentó en el paso 4, todos los valores que se están insertando tienen la misma llave por lo que la operación 'subir' nunca se realiza, por lo que añadir un nodo se termina realizando en tiempo constante y la operación *Heapify* se realiza en tiempo lineal.

- Paso 6: Antes de entrar al ciclo que evalúa si la cola está vacía, se crea un arreglo que indica si un nodo ya fue extraído o no. Luego, utilizando la función 'extraerMinimo()' de las estructuras mencionadas, se extrae el valor mínimo y se marca el índice correspondiente al nodo como verdadero en el arreglo de extraídos. Después, para cada vecino u del nodo v, se realiza la comprobación para verificar si se puede reducir el valor de la distancia, es decir, que el nodo u aún no se ha extraído y que se cumple la condición de distancia especificada. Si la condición se cumple, se actualiza la distancia y el valor del nodo previo; luego, se aplica la función 'reducirLlave(Nodo* nodo, llave_t llave)' implementada en ambas estructuras para reducir la prioridad del nodo u en la cola.

- Paso 7: Teniendo en cuenta que nuestro grafo está implementado como una matriz, no se retorna nada en esta función. En el archivo de experimentación se maneja el uso del grafo ya modificado.

3. Resultados

Para realizar la experimentación se utilizaron inputs generados de manera aleatoria, gracias a la función `crear_grafo(i, j)`. Se ejecutó el algoritmo para inputs con $v = 2^i$ nodos, donde $i \in [10, \ldots, 20]$. Para cada grafo de v nodos, se ejecutó con $e = 2^j$ aristas, donde $j \in [16, \ldots, 22]$. Para cada par de valores (i, j), se crearon y ejecutaron el algoritmo de Dijkstra 50 veces.

La ejecución de la tarea se realizó en un computador con las siguientes especificaciones.

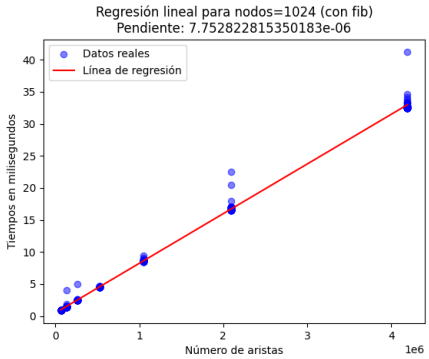
Sistema operativo: Debian 12

Procesador: Intel Core 3-1115G4

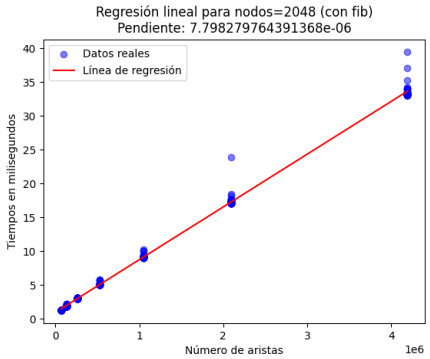
Cache:

1. L1d: 96 MB (2 instancias)
2. L1i: 64 KB (2 instancias)
3. L2: 2.5 MB (2 instancias)
4. L3: 6 MB (1 instancia)

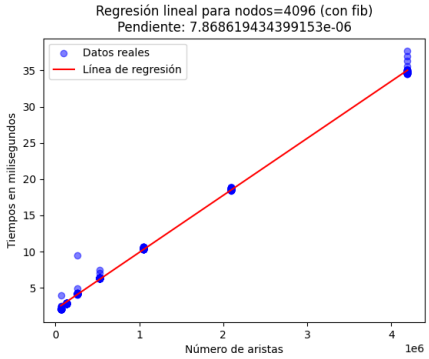
Memoria RAM: 4 GB



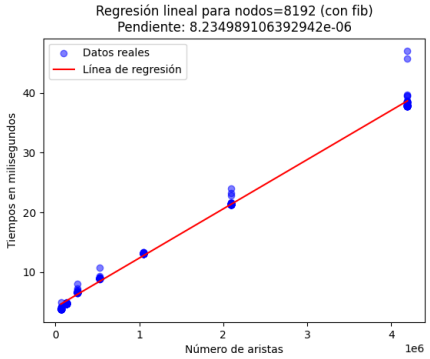
(a) 1024 Nodos Fibonacci



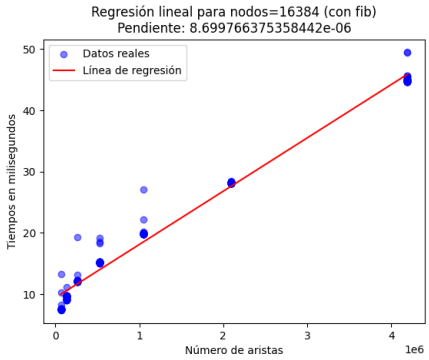
(b) 2048 Nodos Fibonacci



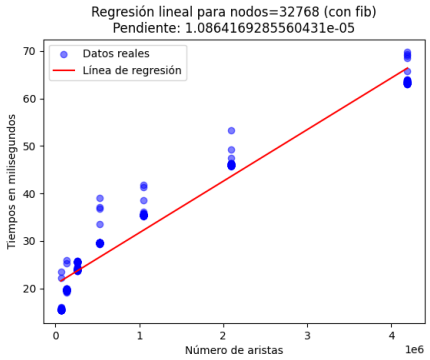
(c) 4096 Nodos Fibonacci



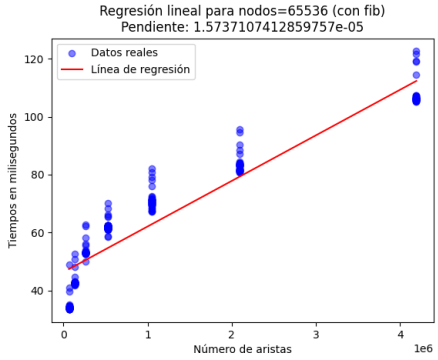
(d) 8192 Nodos Fibonacci



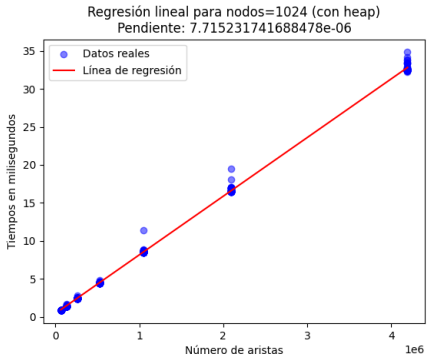
(e) 16384 Nodos Fibonacci



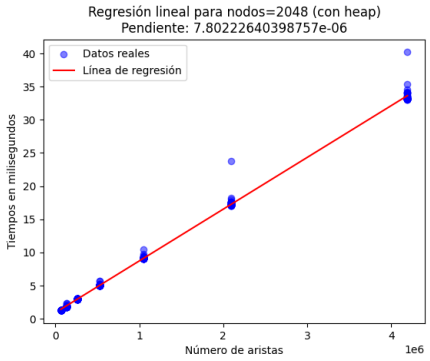
(f) 32768 Nodos Fibonacci



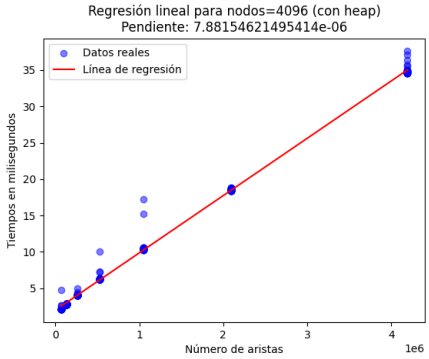
(g) 65536 Nodos Fibonacci



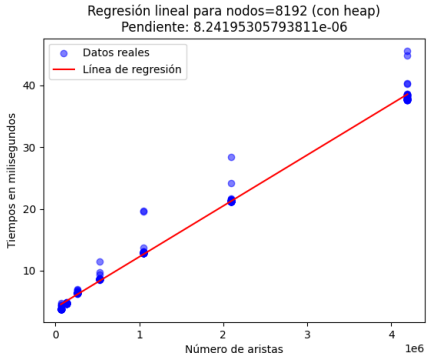
(a) 1024 Nodos Heap



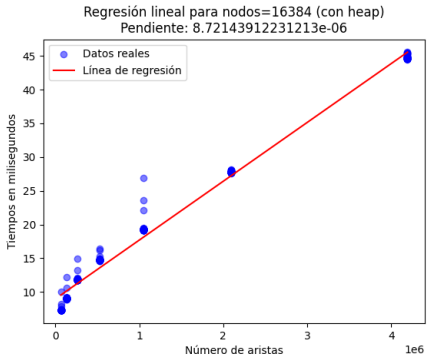
(b) 2048 Nodos Heap



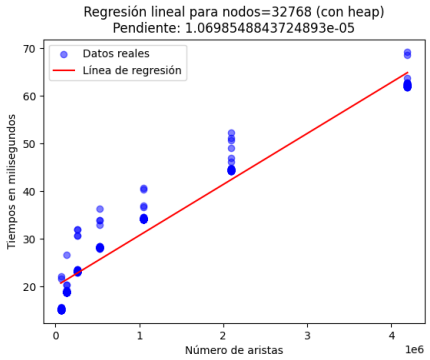
(c) 4096 Nodos Heap



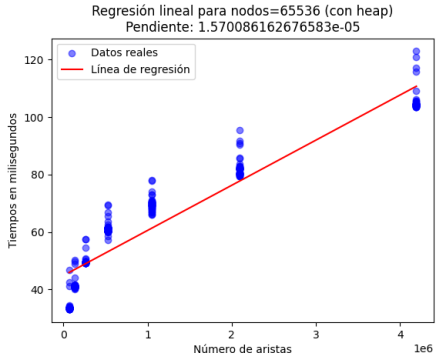
(d) 8192 Nodos Heap



(e) 16384 Nodos Heap



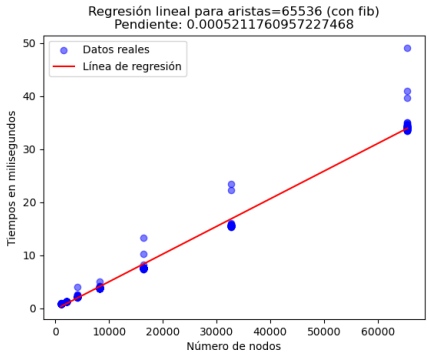
(f) 32768 Nodos Heap



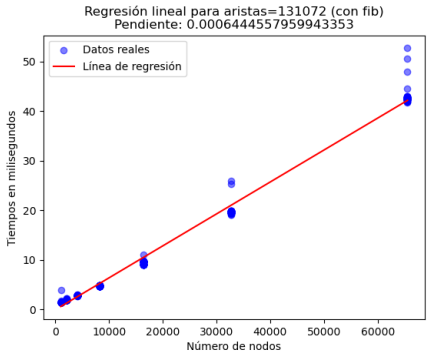
(g) 65536 Nodos Heap

Los gráficos que yacen arriba son los solicitados en el enunciado. Cabe constatar que el tiempo esta medido en milisegundos. Se puede apreciar que la diferencia entre pendientes para los nodos 1024, 4096 y 16834 es 3,7591e-08 , 1,2927e-08 y 2,1673e-08 respectivamente. Cuando los nodos son 1024 se tiene que la pendiente obtenida a partir de la regresión lineal de Fibonacci es mas grande que la que se obtiene con Heap mientras que, para 4096 y 16834, se tiene que la pendiente generada en la regresión lineal de Heap es mas grande que la que se consigue en los gráficos de regresión lineal de Fibonacci.

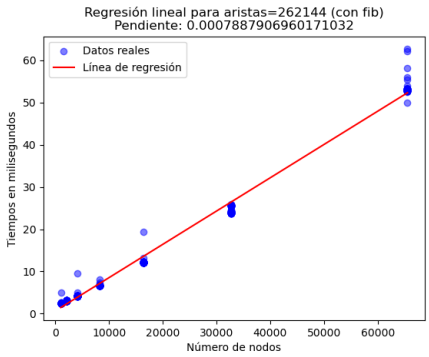
Mas abajo están los gráficos de regresión lineal que se calculan dejando una cantidad de aristas fija y una cantidad de nodos variables mostrando para cada cantidad de nodos los respectivos tiempos de ejecución de cada estructura. Los nodos en este caso van desde 2**10 hasta 2**20. Para estos gráficos también se dejo el tiempo en milisegundos.



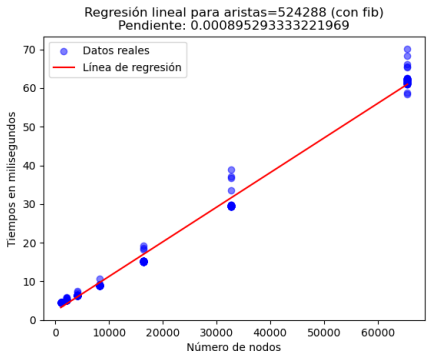
(a) 65536 Aristas Fibonacci



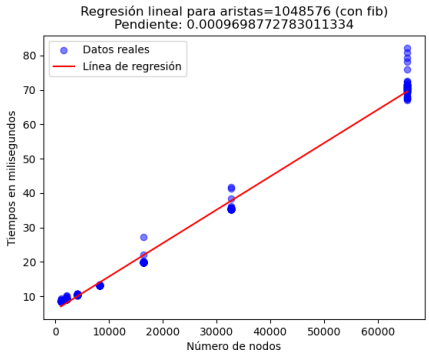
(b) 131072 Aristas Fibonacci



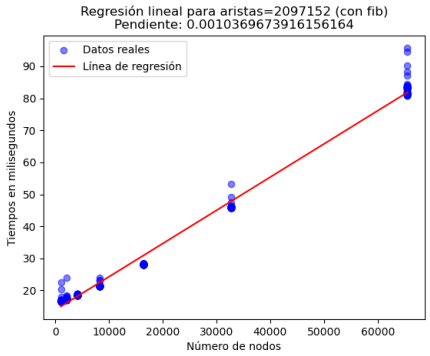
(c) 262144 Aristas Fibonacci



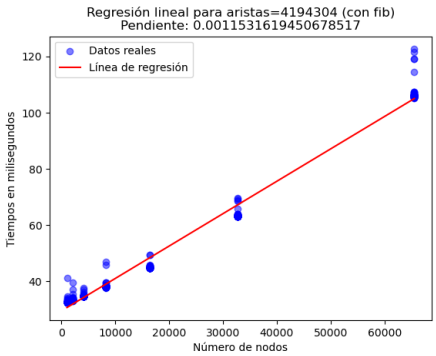
(d) 524288 Aristas Fibonacci



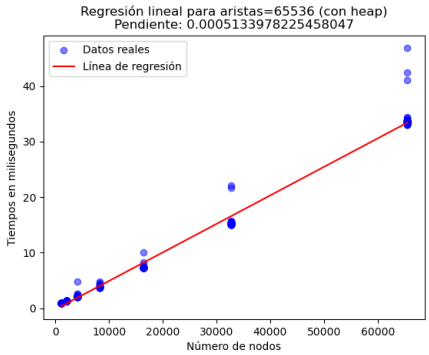
(e) 1048576 Aristas Fibonacci



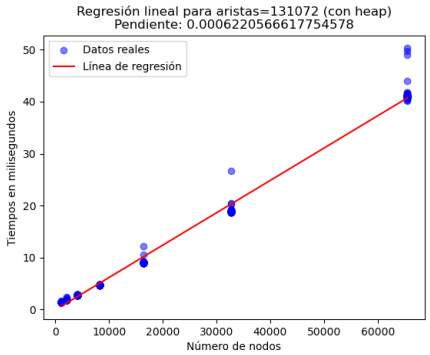
(f) 2097152 Aristas Fibonacci



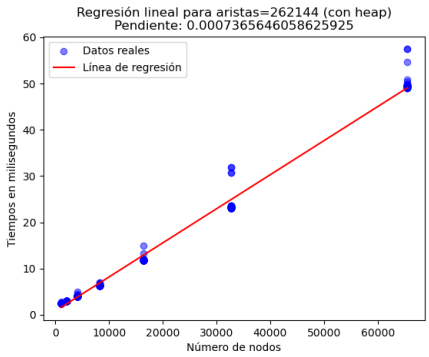
(g) 4194304 Aristas Fibonacci



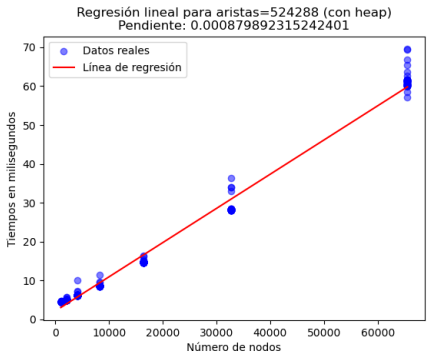
(a) 65536 Aristas Heap



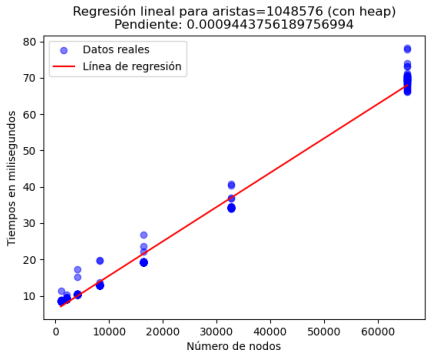
(b) 131072 Aristas Heap



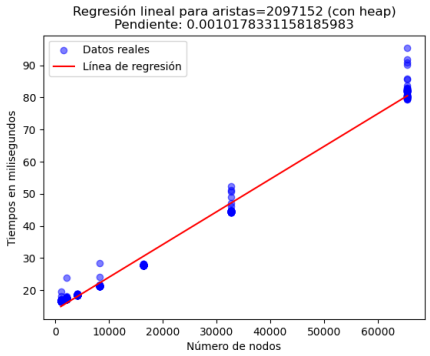
(c) 262144 Aristas Heap



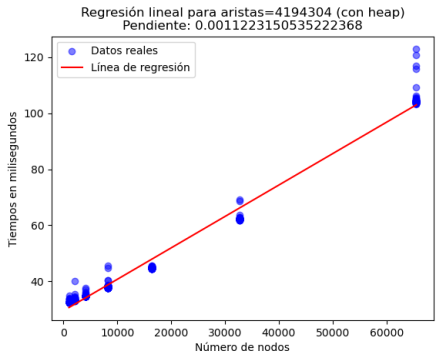
(d) 524288 Aristas Heap



(e) 1048576 Aristas Heap



(f) 2097152 Aristas Heap



(g) 4194304 Aristas Heap

La diferencia entre las pendientes de las regresiones lineales entre los gráficos de Fibonacci y Heap para 65536, 131072, 262144, 524288, 1048576, 2097152 y 4194304 aristas es 7,786e-06, 7,748e-06, 5,2226e-05, 1,5401e-05, 2,5502e-05, 2,51767e-05 y 3,085e-05 respectivamente. Se tiene que la pendiente de Heap es mayor que la pendiente de Fibonacci cuando se hace el análisis con 131072 aristas mientras que en todos los demás casos ocurre lo contrario.

4. Análisis

Antes de analizar los resultados obtenidos, es importante recordar la complejidad de las operaciones en cada tipo de cola.

Tipo de cola	Extraer minimo	Reducir una llave	Insertar
Fibonacci	$\mathcal{O}(\log\ n)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Heap	$\mathcal{O}(\log\ n)$	$\mathcal{O}(\log\ n)$	$\mathcal{O}(\log\ n)$

Como se puede observar, las colas de Fibonacci en ocasiones son mas eficaces que Heap mientras que a veces resulta tener un mejor rendimiento Heap lo que contraviene lo que uno se podría esperar tras un análisis rápido de la tabla de arriba. Esto se justifica ya que Fibonacci, a pesar de tener una mejor complejidad en las operaciones insertar y reducir una llave y de tener la misma complejidad en extraer mínimo, resulta que esta ultima operación es mas costosa en términos generales que el extraer mínimo de Heap lo que, en cierta medida, compensa el costo de las otras operaciones a la larga. Quizá si se tratara de un algoritmo diferente a Dijkstra las colas de Fibonacci tendrían un mejor rendimiento que Heap.

5. Conclusión

A modo de conclusión, implementamos el algoritmo Dijkstra junto con las estructuras Heap y Fibonacci de manera exitosa obteniendo resultados que contradijeron la hipótesis inicialmente planteada. Se intentó dentro de lo posible modularizar los diversos componentes de la solución separando por carpetas los archivos y también creando funciones con un rol bien definido lo que tiene como efecto añadido una mayor facilidad a la hora de entender el código.

Por un lado, a raíz de los resultados entregados por la experimentación, se tiene que la cola de Fibonacci es más eficaz que Heap en algunos casos, mientras que en otros no se cumple esta afirmación, lo cual contradice la hipótesis inicialmente planteada. Esto se debe en parte al análisis que hicimos anteriormente fue basado en el orden de las operaciones, pero como estas son tan solo cotas pueden variar en tiempo de ejecución.