Rendimiento del algoritmo de Dijkstra utilizando montículos binarios y de Fibonacci

David Escobar Rupalí López Alfonso Valderrama

12 de mayo de 2021

Resumen

El algoritmo de Dijkstra puede optimizarse mediante el uso de colas de prioridad. En este informe se presenta un estudio donde se comparan dos implementaciones de colas de prioridad para ejecutar dicho algoritmo: el montículo binario y el montículo de Fibonacci. Los resultados muestran que, pese a que el montículo de Fibonacci tiene menores costos amortizados de sus operaciones, el montículo binario demora menos en la mayoría de los casos estudiados. Se postula que esto ocurre debido al coste de implementar concretamente el primero de éstos. Por consecuencia, se recomienda privilegiar el uso de un montículo binario para implementar un algoritmo de Dijkstra que busque manipular hasta 10000 elementos.

1. Introducción

El algoritmo de Dijkstra, también llamado algoritmo de caminos mínimos, resuelve el problema de determinar el camino más corto desde un vértice origen hacia el resto de los vértices de un grafo que tiene pesos en cada arista.

En el presente informe se realiza una implementación de dicho algoritmo, empleando colas de prioridad. El objetivo de éste consiste en comparar dos variantes de colas de prioridad: el montículo binario y el montículo de Fibonacci, y determinar su efecto en el rendimiento del algoritmo de Dijkstra.

Utilizando grafos conexos generados aleatoriamente, se ejecuta el algoritmo y se evalúan los tiempos de duración de cada implementación, en términos de la cantidad de nodos y de la densidad de aristas del grafo.

1.1. Complejidad de las distintas operaciones en las dos variantes de colas de prioridades

A continuación se presentan dos tablas que exhiben la complejidad de las distintas operaciones que el algoritmo de Dijkstra utilizará, tanto para el caso del montículo binario como para el montículo de Fibonacci.

En la primera tabla se pueden ver las complejidades de peor caso mientras que en la segunda se ven las complejidades del análisis amortizado. Las demostraciones de estas complejidades pueden encontrarse en la sección C del Anexo.

	Montículo binario	Montículo de Fibonacci
extract-min()	$O(log(n))$	$O(n)$
insert(*x, k*)	$O(log(n))$	$O(1)$
empty()	$O(1)$	$O(1)$
decrease-key(*x, k'*)	$O(log(n))$	$O(log(n))$

Cuadro 1: Complejidad de las distintas operaciones en términos de peor caso en las dos variantes de colas de prioridad

	Montículo binario	Montículo de Fibonacci
extract-min()	$O(log(n))$	$O(log(n))$
insert(*x, k*)	$O(log(n))$	$O(1)$
empty(1)	$O(1)$	$O(1)$
decrease-key(*x, k'*)	$O(log(n))$	$O(1)$

Cuadro 2: Complejidad de las distintas operaciones en términos de análisis amortizado en las dos variantes de colas de prioridad

1.2. Pseudo-código implementación algoritmo de Dijkstra

Más abajo se expone el pseudo-código correspondiente a la implementación del algoritmo de Dijkstra que se desarrolló. En él se puede observar a grandes rasgos el comportamiento de dicho algoritmo.

Algoritmo 1 Pseudo-código algoritmo de Dijkstra

```
1:  procedure DIJKSTRA(GRAPH, START, QUEUE)
2:      distances ← {}
3:      for node ∈ graph do
4:          distances[node] ← ∞
5:          queue.insert(node, ∞)
6:      distances[start] ← 0
7:      queue.decreaseKey(start, 0)
8:      while !queue.empty() do
9:          currentNode, currentDistance ← queue.extractMin()
10:         if currentDistance > distances[currentNode] then
11:             continue
12:             for neighbor, weight ∈ graph[currentNode].items() do
13:                 distance ← currentDistance + weight
14:                 if distances[neighbor] > distance then
15:                     distances[neighbor] ← distance
16:                     queue.decreaseKey(neighbor, distance)
17: return distances
```

1.3. Complejidad total del algoritmo de Dijkstra

Se pretende calcular la complejidad total del algoritmo de Dijkstra, según el montículo usado. Para ello, se debe tener en cuenta que el costo del algoritmo recae en las operaciones que éste realiza.

Sea $G = \{E, V\}$ un grafo de $|E|$ aristas y $|V|$ nodos sobre el cual se aplica el algoritmo de Dijkstra. Se tiene que el total de operaciones realizadas corresponde a: $|V|$ inserciones, $|V|$ extracciones de mínimo y $|E| - |V|$ decrementos de clave.

Por lo tanto, la complejidad total del algoritmo estará dada por la ecuación que sigue:

$$C_D = |V| \cdot O(insert) + |V| \cdot O(extract - min) + (|E| - |V|) \cdot O(decrease - key) \tag{1}$$

Cabe mencionar que la cantidad esperada de aristas corresponde a:

$$|E| = (|V| - 1) + \rho \cdot (\frac{|V|^2 - |V|}{2} - (|V| - 1)) \tag{2}$$

Donde ρ corresponde a la probabilidad de generar una arista entre dos nodos arbitrarios no previamente conectados en un grafo. Más adelante se explica cómo se obtuvo dicho valor, en la sección 3.2.

1.3.1. Complejidad del algoritmo de Dijkstra usando montículo binario

En el caso del montículo binario, cada operación toma $O(log|V|)$, por lo que a partir de la ecuación 1 se llega a que $C_D = O((|E| + |V|) \cdot log(|V|))$.

Se reemplaza $|E|$ con la fórmula 2, lo que resulta en: $C_D = O(\frac{\rho}{2} \cdot (|V|^2 - 3|V| + 2) + 2|V| - 1) \cdot log(|V|))$.

Despreciando los términos de bajo orden, se puede concluir que la complejidad del algoritmo de Dijkstra usando montículo binario es:

$$C_{D-binario} = O(\rho \cdot |V|^2 \cdot log(|V|) + |V| \cdot log(|V|)) \tag{3}$$

1.3.2. Complejidad del algoritmo de Dijkstra usando montículo de Fibonacci

Como se vio previamente en la sección 1.1, el montículo de Fibonacci tiene costo $O(1)$ en insertar, $O(log|V|)$ amortizado en extracción de mínimo y $O(1)$ amortizado para el decremento de claves. Entonces, la complejidad total amortizada es $O(|V| \cdot log|V| + |E|)$.

Reemplazando $|E|$ con la ecuación 2 se tiene que: $C_D = O(|V| \cdot log(|V|) + \frac{\rho}{2} \cdot (|V|^2 - 3|V| + 2) + |V| - 1))$

Despreciando los términos de bajo orden, se puede concluir que la complejidad del algoritmo de Dijkstra usando montículo de Fibonacci es:

$$C_{D-Fibonacci} = O(\rho \cdot |V|^2 + |V| \cdot log(|V|)) \tag{4}$$

2. Hipótesis

Se desea plantear una hipótesis referente al efecto del uso de un montículo binario versus con un montículo de Fibonacci en el rendimiento del algoritmo de Dijkstra. Con este propósito en mente, es claro que se requiere comparar los resultados obtenidos para la complejidad total del algoritmo en cada caso, es decir, las ecuaciones 3 y 4 de la sección 1.3.

Lo anterior significa restar las complejidades de ambas implementaciones, lo cual se traduce en la siguiente fórmula: $\rho \cdot |V|^2 \cdot (log(|V|) - 1) > 0$. De ésta, se deduce que $\rho > 0$ y $|V| > 2$.

Entonces, según el análisis teórico realizado, es pertinente postular que el algoritmo de Dijkstra tendrá mejor rendimiento utilizando el montículo de Fibonacci para $\rho \in (0, 1]$ y cualquier $|V| > 2$.

3. Diseño experimental

3.1. Generación aleatoria de grafos conexos

Para poder llevar a cabo los experimentos, es necesario generar varios grafos aleatorios conexos. A continuación se describe una forma simple de conseguirlo.

Sean $v_1, ..., v_n$ los vértices del grafo, se comienza construyendo aristas para cada par (v_i, v_{i+1}) con un peso aleatorio entre $[1, 10^9]$, con $i \in [1, n-1]$. Esto asegura que el grafo sea conexo.

Luego, para cada par de vértices $\{v_i, v_j\}$ (en donde $v_i \neq v_j + 1$ y $v_i + 1 \neq v_j$) se decide si se agregará una arista entre ellos, usando una variable aleatoria binaria de peso $0 \leq \rho \leq 1$ (llamada densidad del grafo). Si se decide agregar la arista, su peso es escogido uniformemente en el intervalo $[1, 10^9]$.

El pseudo-código de este apartado se puede encontrar en la sección E del Anexo.

3.2. Número esperado de aristas en función de ρ y $|V|$

El algoritmo descrito en la sección anterior siempre genera $|V| - 1$ aristas al inicio. Además, con probabilidad ρ genera una arista entre dos nodos no generada previamente.

El total de aristas generadas de este modo es $\frac{|V|^2-|V|}{2} - (|V| - 1)$, donde el primer término corresponde al total de aristas generables entre $|V|$ vértices, y el segundo es el total de aristas ya generadas.

Luego, el valor esperado de aristas totales en función de ρ y $|V|$ es $(|V| - 1) + \rho \cdot (\frac{|V|^2-|V|}{2} - (|V| - 1))$.

3.3. Experimentos para comprobar la hipótesis

Debido a que la complejidad del algoritmo de Dijkstra depende tanto de la densidad del grafo como de la cantidad de nodos, se procederá a generar un listado con distintos valores para la cantidad de nodos, para luego fijar uno. Después, se iterará sobre una serie de diferentes densidades de grafo, generando 3 experimentos distintos para cada valor, promediando el tiempo de ejecución de éstos para ambos montículos. Luego, se fijará la siguiente cantidad de nodos en la lista y se repetirá el proceso, hasta que se hayan consumido todos.

El listado con la cantidad de nodos corresponderá a $[10, 100, 1000, 10000]$ debido a que son valores de orden distinto.

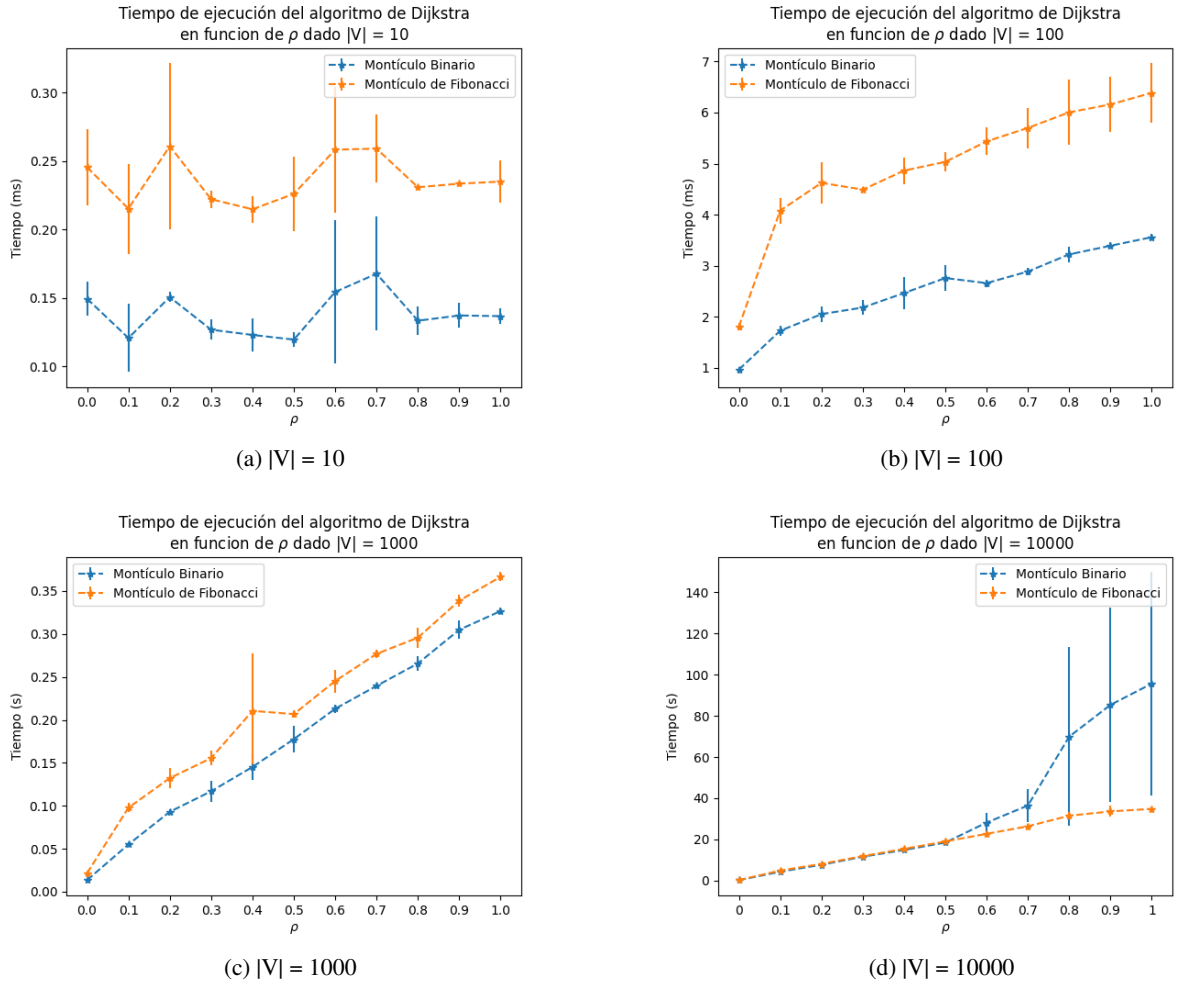
Con respecto a ρ, se producirá una lista de [0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1], ya que la hipótesis plantea que el algoritmo de Dijkstra usando el montículo de Fibonacci será mejor para todo $\rho > 0$.

El pseudo-código del experimento se exhibe en la sección F del Anexo.

Adicionalmente, las especificaciones técnicas del equipo en el que se llevaron a cabo los experimentos se encuentra disponible en la sección D del Anexo.

4. Resultados de los experimentos

A continuación se presentan los resultados de los experimentos.



(a) |V| = 10

(b) |V| = 100

(c) |V| = 1000

(d) |V| = 10000

Figura 1: Tiempo de ejecución del algoritmo de Dijkstra para cantidad de nodos |V|.

Es importante aclarar que las sub-figuras (a) y (b) de la figura 1 miden el tiempo en milisegundos mientras que las sub-figuras (c) y (d) lo hacen en segundos.

Se aprecia que en las sub-figuras (a), (b) y (c) el tiempo de ejecución del algoritmo de Dijkstra utilizando el montículo binario es siempre menor que el tiempo de ejecución usando el montículo de Fibonacci. No obstante, en la sub-figura (d) se observa que los tiempos de ejecución de ambos montículos son idénticos para todos los $\rho \leq 0{,}5$. Para el resto, en cambio, el algoritmo de Dijkstra con el montículo de Fibonacci logra mejores tiempos de ejecución.

También, es posible advertir que el montículo de Fibonacci presenta desviaciones estándar más pequeñas a medida que la cantidad de nodos crece. No se puede establecer una relación clara para el montículo binario.

A medida que crece la cantidad de nodos $|V|$, los tiempos de ejecución del algoritmo tienen una dependencia lineal con la densidad de grafo ρ. La única excepción es el montículo binario para $|V| = 10000$ (figura 1.d), la cual además presenta una dispersión inusual de los tiempos de duración para los mayores valores de ρ.

5. Discusión

Los resultados muestran une dependencia lineal entre la duración del algoritmo con cada implementación de la cola de prioridad y la densidad de aristas, salvo para $|V| = 10000$. La dependencia lineal de dicha densidad era esperada y el caso excepcional muestra una desviación estándar muy grande de los datos. Se presume que esto ocurrió por un problema de ejecución, por ejemplo, asociado a la búsqueda de espacio en memoria, pues los grafos más grandes ($\approx$100000000 de aristas) requirieron porcentajes cercanos a los $3GB$ (ver Anexo, figura G (d)).

En general, para el algoritmo de Dijkstra, la implementación de una cola de prioridad con un montículo binario se muestra más conveniente en términos de duración que un montículo de Fibonacci para una gran cantidad de nodos, y en los casos en que no, la diferencia entre ambos es pequeña. Esto parece contradecirse con el razonamiento del análisis de costos, el cual muestra que Fibonacci debería demorar menos en la mayoría de los casos. Si bien eventualmente dicho montículo supera al binario, lo hace con una diferencia muy pequeña en duración.

5.1. Posible explicación de resultados contrarios a la hipótesis

Una posible explicación de lo sucedido requiere estudiar la cantidad exacta de operaciones que realiza cada función. Por ejemplo, si bien el crecimiento de una función $O(\log n)$ es infinitamente más rápido que una función $O(1)$, si esta última ejecuta una cantidad fija de 1000 operaciones, mientras la primera realiza exactamente $\lceil \log_2 n \rceil$ operaciones, la función en tiempo constante comienza a ser más eficiente recién cuando $n = 2^{1000}$, lo cual es, usualmente, demasiado grande. Se procede en los siguientes párrafos a analizar de manera cualitativa la posibilidad de que en este problema ocurra algo similar.

Una distinción relevante es que el algoritmo de Dijkstra ejecuta las operaciones **insert** consecutivamente, agregando a la cola de prioridad elementos cuya clave es siempre la misma (∞). Esto significa que el montículo binario nunca es sometido a una reorganización de sus elementos, dado que nunca se inserta un elemento con mayor prioridad que otro. Por lo mismo, en realidad el coste amortizado de esta operación en el *heap* binario es simplemente $O(1)$.

Lo anterior hace que ambos montículos difieran únicamente en el orden de complejidad de la operación **decrease-key**, la cual, además, es la operación que se llama más a menudo (pues la cantidad de veces que se realiza es $|E| - |V|$).

Sin embargo, un análisis muestra que, suponiendo que la frecuencia de un peor caso es semejante para ambas (lo cual ocurrirá seguido, pues todos los nodos decrecerán su clave al menos una vez), el número de operaciones que realiza cada algoritmo favorece al montículo binario en una gran cantidad de casos: en el peor caso, decrecer una clave en el montículo de Fibonacci toma alrededor de 15 operaciones (10 reasignaciones de punteros multiplicadas por 1.5, lo cual es el total de elementos cortados por aplicación de la función en el peor caso según la contabilidad de costos), mientras que el montículo binaria toma alrededor de $2 \cdot log(n)$ operaciones.

Nótese que el caso con mayor cantidad de nodos probado en el algoritmo es de 10000 nodos, lo cual es un poco más de 2^{13}. Esto implica que en el caso más costoso toma alrededor de 26 operaciones **decrease-key**, número que es mayor pero comparable con el total de operaciones del *heap* de Fibonacci.

Pese a lo que ocurre, esta comparación no es suficiente para explicar la excesiva demora de Fibonacci, pues experimentalmente los valores son recién comparables cuando $|V| = 5000$. Se aventura, entonces, que el exceso de demora está relacionado además con el uso de memoria para instanciar los elementos.

En el Anexo, en la sección G, se exponen los gráficos que muestran los puntos álgidos de uso de memoria principal para cada experimento. Un análisis cuantitativo de éstos permite notar que el montículo de Fibonacci siempre emplea más memoria que el montículo binario. Tiene sentido, pues si bien en cada caso se instancia una misma cantidad de nodos, los de Fibonacci tienen 8 atributos, cuatro de ellos punteros, y 9

métodos; mientras el *heap* binario tiene únicamente 3 atributos, todos de tipo primitivo, y solo 3 métodos, económicos en cantidad de operaciones.

Es oportuno notar que la diferencia entre los *peaks* de memoria no es considerable (la memoria ocupada por los nodos del montículo binario es consistentemente un 90 % de la ocupada por los nodos del montículo de Fibonacci). No obstante, se considera importante este factor al momento de decidirse por una implementación, pues puede, entre otras cosas, influir en los tiempos de localización en memoria de los elementos utilizados, el cual no es constante.

En este informe no se realiza un análisis exhaustivo del uso de memoria ni de la cantidad de operaciones, de manera que un planteamiento futuro consiste en verificar la influencia de estos factores en el tiempo de ejecución.

5.2. Otra posible explicación de resultados contrarios a la hipótesis

Otra posible explicación de por qué la implementación del montículo de Fibonacci demora tanto en esta implementación tiene que ver con el orden de ejecución de las funciones **decrease-key** y **extract-min**.

El algoritmo de Dijkstra, por cada nodo extraído, realiza una iteración sobre sus vecinos, los cuales son aproximadamente $\rho(|V| - 1) + 1 \approx \rho|V|$. En cada una de estas iteraciones, puede invocar a **decrease-key**, lo cual en el peor caso significa invocarla siempre.

Si la cantidad de veces que esta función es invocada por nodo es proporcional al peor caso (es decir $O(\rho|V|)$), lo que ocurre concretamente en el montículo de Fibonacci, es que desarma árboles y va dejando cada vez más nodos como raíces en el *heap*. Si el número de raíces generadas de esta forma resulta la mayoría de las veces $O(\rho|V|)$, entonces la operación **extract-min** se ejecuta siempre en el peor caso, de manera que toma $O(\rho|V|)$ igualmente.

Como el crecimiento de una función de esta índole es más rápido que una función $O(\log|V|)$, una cantidad suficiente de casos hacen que esta operación también influya en el costo total del algoritmo.

Concretamente, en el montículo de Fibonacci, la complejidad de una operación **extract-min** en una determinada iteración tiene un coste proporcional al del número de operaciones **decrease-key** en dicha iteración que supusieron una reorganización del *heap*. Esto, en el peor caso, significa que demora $O(n)$, siendo n el número de elementos al momento de extraer.

El algoritmo realiza $|V|$ extracciones una tras otra, sin inserciones entre medio, lo cual se traduce en un coste $O(\rho|V|^2)$ en Fibonacci, en contraste con el coste total de **decrease-key** en el montículo binaria, el cual es $O(\rho|V|^2 \log|V|)$. Si bien el crecimiento sigue siendo menor, **extract-min** realiza considerablemente más operaciones internas en el montículo de Fibonacci (alrededor de 12 reasignaciones multiplicadas por la complejidad de la operación) que lo que hace **decrease-key** en el montículo binario (solo 2 reasignaciones por la complejidad de la operación), de manera que es posible que influya en una gran cantidad de casos.

5.3. Reflexión final

Las explicaciones planteadas en las secciones anteriores no han sido comprobadas, si no que se plantean para estudiar como trabajo futuro.

En cualquier caso, los experimentos muestran que la implementación de un montículo de Fibonacci supone costos que la hacen comparable a la implementación de un *heap* estándar hasta el orden de los 1000 elementos. Sumando el hecho de que este montículo es particularmente tedioso de implementar (los autores de este informe pasaron al menos tres días implementándolo) se sugiere que un montículo binario resulta suficiente para una gran cantidad de casos.

6. Conclusiones

Tras haber analizado los diferentes resultados obtenidos, se concluye que el algoritmo de Dijkstra usando el montículo de Fibonacci presenta un rendimiento superior únicamente para una cantidad exageradamente grande de nodos y de aristas, puesto que solo entonces se hacen notorios los costos amortizados.

Lo anterior, sumado a su costosa implementación, sugiere que frente a algún problema en donde se deba implementar una cola de prioridad, se prefiera el montículo binario por sobre el de Fibonacci.

Sin ir en desmedro de la idea anterior, es pertinente declarar que el montículo de Fibonacci posee costos amortizados interesantes, pero poco útiles, por lo que su interés se remite más bien al ámbito teórico por sobre el práctico.

7. Anexo

A. Montículo binario

Considere que el montículo binario corresponde a un árbol binario, es decir, un árbol en el cual cada nodo tiene a lo más dos nodos hijos. También, satisface dos propiedades adicionales:

- Es un árbol binario completo, lo que significa que cada nivel excepto el último está completo, y los nodos del último nivel están posicionados tan a la izquierda como sea posible.
- La clave de cada nodo es menor o igual que la de sus nodos hijos.

B. Montículo de Fibonacci

Considere que el montículo de Fibonacci es una implementación específica de la estructura de datos *heap*[1], que hace uso de los números de Fibonacci[2].

Su implementación consiste en lo siguiente:

- Uso de listas doblemente enlazadas.
- Cada nodo contiene un puntero a su padre y uno de sus hijos.
- Los hijos están conectados entre ellos mediante una lista doblemente enlazada, llamada la lista de los hijos.
- Cada hijo en la lista de los hijos tiene punteros a su hermano izquierdo y a su hermano derecho.
- Para cada nodo, la lista enlazada también almacena el número de hijos que éste tiene, un puntero a la raíz conteniendo la mínima clave, y un indicador de si el nodo ha sido marcado o no (un nodo es marcado para indicar que ha perdido un único hijo, y un nodo no se encuentra marcado si no ha perdido ningún hijo).

En los montículos de Fibonacci, el grado de los nodos (la cantidad de hijos) está restringido. Cada nodo en el montículo tiene a lo más grado $O(log(n))$.

C. Demostraciones complejidades de las distintas operaciones de las dos variantes de colas de prioridad

C.1. Complejidades de peor caso

C.1.1. Complejidades de peor caso de las operaciones de montículo binario

Más adelante se detallan las demostraciones correspondientes a las complejidades de las distintas operaciones de montículo binario, las cuales aplican en términos de peor caso.

[1] Un *heap* es una estructura de datos basada en árboles, limitada por una propiedad que dictamina que los nodos padres siempre deben tener un valor menor o igual que el de sus hijos.

[2] La secuencia de Fibonacci es una secuencia de enteros definida por una relación lineal de recurrencia, en la cual cada término corresponde a la suma de los dos términos previos.

- **extract-min**(): Para obtener el mínimo de un montículo binario, se debe extraer la raíz del árbol, ya que en ésta se almacena el mínimo elemento. Dicha acción implica que se debe reordenar el montículo, de forma de asegurarse que cumpla las propiedades descritas en la sección A.

  Para llevar a cabo el reordenamiento requerido, se coloca como raíz momentáneamente al elemento que se encuentre más profundo en el extremo derecho del montículo. Luego, este elemento se "hunde" en el árbol, hasta encontrar el lugar que le corresponde, es decir, debe irse intercambiando por los nodos del lado derecho, cuando éstos sean menores que el elemento en cuestión.

  En el peor caso, este proceso recorre la altura completa del árbol, la cual es igual a $log(n)$, por lo tanto la operación toma $O(log(n))$, y de ahí se deduce que la complejidad de **extract-min**() para un montículo binario es $O(log(n))$. ■

- **insert**(*x, k*): Para insertar un par (*x, k*) en un montículo binario, se debe efectuar un procedimiento similar al explicado en la operación previa. Se agrega un valor al final del árbol y se "flota" hacia arriba, es decir, se va intercambiando por los nodos en el caso que corresponda, siempre que éstos sean mayores que el elemento en cuestión.

  En el peor caso, este proceso recorre la altura completa del árbol, por lo tanto la operación también toma $O(log(n))$, y de ahí se deduce que la complejidad de **insert**(*x, k*) para un montículo binario es $O(log(n))$. ■

- **empty**(): La operación determina si el montículo se encuentra vacío o no.

  Dependiendo de la implementación, podría comprobar que la raíz sea nula o no, o que el número de elementos del árbol sea mayor o igual a 0, pero cualquiera sea el caso tomará tiempo constante, por lo tanto, la complejidad de **empty**() para un montículo binario es $O(1)$. ■

- **decrease-key**(*x, k'*): Cuando se disminuye el valor de una llave, puede ocurrir que el nuevo valor todavía cumple con ser mayor que su nodo padre, en cuyo caso no se debe hacer nada más.

  Sin embargo, si el nuevo elemento infringe dicha propiedad, la operación debe comportarse similarmente a una inserción, esto es, "subir" el nodo hasta que éste se encuentre en la posición correcta.

  En el peor caso, esto tomará $O(log(n))$, y así la complejidad de **decrease-key**(*x, k'*) para un montículo binario es $O(log(n))$. ■

C.1.2. Complejidades de peor caso de las operaciones de montículo de Fibonacci

A continuación se describen las demostraciones correspondientes a las complejidades de las distintas operaciones de montículo de Fibonacci, las cuales aplican en términos de peor caso.

- **extract-min**(): Dada la estructura de los montículos de Fibonacci mencionada en la sección B, conocer el mínimo elemento toma tiempo constante, puesto que siempre se mantiene almacenado. Sin embargo, extraerlo conlleva inevitablemente tener que actualizar dicho valor.

  Para encontrar el segundo mínimo se deben revisar todas las raíces del montículo, las cuales en el peor caso pueden llegar a ser n, por lo tanto, el proceso tomará $O(n)$. Se concluye entonces que la complejidad de **extract-min**() para un montículo de Fibonacci es $O(n)$. ■

- **insert**(*x, k*): Para insertar un elemento (*x, k*) en un montículo de Fibonacci A, primero se crea un nuevo montículo B a partir de dicho elemento, y luego los dos montículos A y B son mezclados mediante una operación *merge*.

Esta operación se basa en concatenar las dos listas que contienen las raíces de los árboles. Se comparan las raíces de los dos montículos que se van a mezclar, y la que sea menor se convierte en la raíz del nuevo montículo combinado. El otro árbol se agrega como un subárbol a esta raíz.

Todos los pasos nombrados se pueden lograr en tiempo constante, lo que quiere decir que un *merge* puede llevarse a cabo en tiempo constante.

Después de realizar la operación descrita, se actualiza el puntero al último elemento, en caso de ser necesario, y la cantidad total de nodos en el árbol aumenta en 1. Esto también toma tiempo constante, por lo tanto, se demuestra que la complejidad de **insert**(*x, k*) para un montículo de Fibonacci es $O(1)$. ■

- **empty**(): Para determinar si el montículo se encuentra vacío o no, bastaría con revisar el puntero al mínimo elemento. Si éste es nulo, la respuesta es afirmativa, y en caso contrario es negativa.

  Dicha comprobación toma tiempo constante, luego, la complejidad de **empty**() para un montículo de Fibonacci es $O(1)$. ■

- **decrease-key**(*x, k'*): Al disminuir el valor de una llave, puede que esta acción no cause que el montículo en cuestión transgreda sus propiedades fundamentales. En caso de que sí lo haga, la operación seguirá un procedimiento específico, el cual se explica más adelante.

  Se comienza removiendo el elemento al cual se le decrementó el valor de su clave, y éste se transforma en un nuevo árbol. Si el padre del nodo recién extraído no es una raíz, se marca. Si ya había sido marcado, también se convierte en un nuevo árbol, y su padre es marcado, y así sucesivamente. Se continua con este proceso a lo largo del árbol, ya sea hasta que se alcance la raíz o un nodo que no esté marcado.

  Después, se actualiza el puntero al mínimo si es que corresponde.

  En el peor caso, la operación implicará que se deba recorrer la altura del árbol, y ya que ésta puede ser a lo más $log(n)$, se infiere que la complejidad de **decrease-key**(*x, k'*) para un montículo de Fibonacci es $O(log(n))$. ■

C.2. Complejidades amortizadas

Cuando se utiliza análisis amortizado, se tiene como propósito estudiar el peor caso para una secuencia de operaciones.

Se define el costo amortizado de una secuencia de k operaciones como la suma de los costos de las operaciones dividido en el número total de operaciones, es decir:

$$\hat{C} = \frac{\sum_{i=1}^{k} c_i}{k} \tag{5}$$

Existen diferentes técnicas para demostrar costos amortizados, entre ellas la función de potencial. Ésta define ϕ_i como el "potencial" acumulado hasta la operación i. Además, se define una secuencia de costos amortizados como:

$$\hat{c_i} = c_i + \Delta\phi_i \tag{6}$$

donde c_i corresponde al costo real de la operación i, mientras que $\Delta\phi_i = \phi_i - \phi_{i-1}$, es decir, la diferencia de potencial que generó la operación i.

A partir de la ecuación 6 mencionada previamente, se obtiene:

$$\sum_{i}^{k} \hat{c_i} \geq \sum_{i}^{k} c_i \tag{7}$$

considerando que $\phi_i \geq \phi_0$ se cumple para todo i.

C.2.1. Complejidades amortizadas de las operaciones de montículo binario

Más adelante se detallan las demostraciones correspondientes a las complejidades de las distintas operaciones de montículo binario, las cuales aplican en términos de análisis amortizado.

En ellas se emplea la estrategia de función de potencial, definiéndola explícitamente como $\phi_i = |M|_i$, donde $|M|_i$ representa la cantidad de nodos que posee el montículo binario M en el momento i y $|M|_0 = m$.

- **extract-min**(): Se quiere calcular el costo amortizado para una secuencia de k operaciones de extracción de mínimo en un montículo binario M.

  Empleando la fórmula 6 vista en la sección C.2, se tiene que $c_i \leq log(n)$ para todo i y que $\Delta\phi_i = \phi_i - \phi_{i-1} = m - i - (m - (i-1)) = -1$. Entonces, $\hat{c_i} \leq log(n) - 1 \leq log(n)$.

  Dada la desigualdad 7 de la sección C.2, y que $\sum_i^k \hat{c_i} \leq log(n) \cdot k$, se deduce – aplicando la fórmula 5 de la sección C.2 – que el costo amortizado de **extract-min**() para un montículo binario es a lo más $\frac{O(log(n)) \cdot k}{k} = O(log(n))$. ■

- **insert**(x, k): Se pretende determinar el costo amortizado para una secuencia de k operaciones de inserción en un montículo binario M.

  Usando la fórmula 6, es sabido que $c_i \leq log(n)$ para todo i y que $\Delta\phi_i = \phi_i - \phi_{i-1} = (i+1) - i = 1$. Luego, $\hat{c_i} \leq log(n) + 1 \leq log(n)$.

  Considerando la ecuación 7, y que $\sum_i^k \hat{c_i} \leq log(n) \cdot k$, se concluye – utilizando la ecuación 5 – que el costo amortizado de **insert**(x, k) para un montículo binario es a lo más $\frac{O(log(n)) \cdot k}{k} = O(log(n))$. ■

- **empty**(): Se busca obtener el costo amortizado para una secuencia de k operaciones **empty** en un montículo binario M.

  Empleando la fórmula 6, se tiene que $c_i = 1$ para todo i y que $\Delta\phi_i = \phi_i - \phi_{i-1} = |M|_i - |M|_i = 0$. Por consecuencia, el resultado corresponde a $\hat{c_i} = 1$.

  Dada la desigualdad 7, y el hecho de que $\sum_i^k \hat{c_i} = 1 \cdot k$, es pertinente declarar – luego de aplicar la ecuación 5 – que el costo amortizado de **empty**() para un montículo binario es a lo más $O(\frac{k}{k}) = O(1)$. ■

- **decrease-key**(x, k'): Se desea calcular el costo amortizado para una secuencia de k operaciones de decremento de claves en un montículo binario M.

  Mediante el uso de la ecuación 6, se verifica que $c_i \leq log(n)$ para todo i y que $\Delta\phi_i = \phi_i - \phi_{i-1} = |M|_i - |M|_i = 0$. Entonces, $\hat{c_i} \leq log(n)$.

  Se tiene la desigualdad 7 y $\sum_i^k \hat{c_i} \leq log(n) \cdot k$, por lo que se deduce – aplicando la fórmula 5 – que el costo amortizado de **decrease-key**(x, k') para un montículo binario es a lo más $\frac{O(log(n)) \cdot k}{k} = O(log(n))$. ■

C.2.2. Complejidades amortizadas de las operaciones de montículo de Fibonacci

A continuación se describen las demostraciones correspondientes a las complejidades de las distintas operaciones de montículo de Fibonacci, las cuales aplican en términos de análisis amortizado.

En ellas se emplea la estrategia de función de potencial, definiéndola explícitamente como $\phi_i = t_i + 2 \cdot m_i$, donde t_i es el número de árboles en el montículo de Fibonacci, y m_i es el número de nodos marcados en el momento i.

- **extract-min**(): Se quiere calcular el costo amortizado para una secuencia de k operaciones de extracción de mínimo en un montículo de Fibonacci. La operación para extraer el mínimo se aplica en tres fases:

  - Primero, se remueve la raíz que contiene el mínimo elemento, y sus hijos se convierten en raíces de nuevos árboles. Dado que el grado de un nodo cualquiera no puede ser mayor que $log(n)$ (tal como se indica en la sección B), el costo real de dicha acción es menor o igual a $log(n)$ y el número de árboles t_i se incrementa en a lo más $log(n)$, por lo tanto se tiene lo siguiente:
    $\hat{c_i} = c_i + \Delta\phi_i \leq log(n) + (t_i + log(n) + 2 \cdot m_i - (t_i + 2 \cdot m_i))$
    $\Rightarrow \hat{c_i} \leq 2 \cdot log(n) \approx log(n)$
  - Luego, se debe actualizar el puntero a la raíz con la mínima clave. Tal como se mencionó en la sección C.1.2, para lograr esto se deben revisar todas las raíces del montículo, las cuales en el peor caso podrían ser n. Debido a esto, se hace imperativo emplear una estrategia astuta.
    Se disminuye la cantidad de raíces mediante la unión sucesiva de raíces del mismo grado. Cuando dos raíces u y v tienen el mismo grado, una de ellas se convierte en hija de la otra, de tal forma que la que tiene la menor clave se mantenga como raíz. Su grado aumentará en uno. Este proceso se repite hasta que todas las raíces tengan grados diferentes.
    Para encontrar raíces del mismo grado eficientemente, se usa un arreglo de largo $O(log(n))$ en el cual se almacena un puntero a una raíz de cada grado. Cuando se encuentra una segunda raíz del mismo grado, ambas se unen y se actualiza el arreglo.
    El costo real de esta fase será $O(log(n) + t_i)$, donde t_i corresponde al número de raíces existentes al comienzo del procedimiento. Al final se tendrán a lo más $O(log(n))$ raíces (ya que cada una tiene un grado diferente). Entonces:
    $\hat{c_i} = c_i + \Delta\phi_i \leq log(n) + t_i + (log(n) + 2 \cdot m_i - (t_i + 2 \cdot m_i))$
    $\Rightarrow \hat{c_i} \leq 2 \cdot log(n) \approx log(n)$
  - En la tercera fase, se revisa cada una de las raíces restantes y se encuentra el mínimo. Puesto que de la fase anterior quedaron a lo más $O(log(n))$ raíces, esta acción no toma más de $O(log(n))$. Por consecuencia, se tiene que:
    $\hat{c_i} = c_i + \Delta\phi_i \leq log(n) + (t_i + 2 \cdot m_i - (t_i + 2 \cdot m_i))$
    $\Rightarrow \hat{c_i} \leq log(n)$

  Finalmente, considerando que para las tres fases de una operación de extracción de mínimo en un montículo de Fibonacci se cumple $\hat{c_i} \leq log(n)$, se deduce que $c_i \leq log(n)$, y así el costo amortizado de **extract-min**() para un montículo de Fibonacci es a lo más $\frac{O(log(n)) \cdot k}{k} = O(log(n))$. ■

- **insert**(x, k): Se pretende determinar el costo amortizado para una secuencia de k operaciones de inserción en un montículo de Fibonacci.

  Para insertar un par x, k en un montículo A se debe crear un nuevo montículo B con un único elemento, para luego mezclar A y B. Esta unión de montículos se logra mediante la concatenación de las listas de raíces de árboles de ambos, y toma tiempo constante. Entonces:

$\hat{c_i} = c_i + \Delta\phi_i = 1 + (t_i + 1 + 2 \cdot m_i - (t_i + 2 \cdot m_i))$

$\Rightarrow \hat{c_i} = 2$

Se concluye que $c_i = 2$, por lo que el costo amortizado de **insert**(x, k) para un montículo de Fibonacci es $\frac{O(1) \cdot k}{k} = O(1)$. ■

- **empty**(): Se busca obtener el costo amortizado para una secuencia de k operaciones **empty** en un montículo de Fibonacci.

  Al ejecutar una de estas operaciones solo se verifica si el puntero al mínimo contiene una referencia vacía (pues no hay mínimo si no hay elementos), de manera que:

  $\hat{c_i} = c_i + \Delta\phi_i = 1 + (t_i + 2 \cdot m_i - (t_i + 2 \cdot m_i))$

  $\Rightarrow \hat{c_i} = 1$

  Se deduce que $c_i = 1$, entonces el costo amortizado de **empty**() para un montículo de Fibonacci es $\frac{O(1) \cdot k}{k} = O(1)$. ■

- **decrease-key**(x, k'): Se desea calcular el costo amortizado para una secuencia de k operaciones de decremento de claves en un montículo de Fibonacci.

  Recordando lo expresado en la sección C.1.2, se sabe que una operación de decremento de claves comienza disminuyendo el valor de una llave. En caso de que esto cause que el montículo en cuestión transgreda sus propiedades fundamentales, la operación seguirá un procedimiento específico, el cual se explica más adelante.

  Se comienza removiendo el elemento al cual se le decrementó el valor de su clave, y éste se transforma en un nuevo árbol. Si el padre del nodo recién extraído no es una raíz, se marca. Si ya había sido marcado, también se convierte en un nuevo árbol, y su padre es marcado, y así sucesivamente. Se continua con este proceso a lo largo del árbol, ya sea hasta que se alcance la raíz o un nodo que no esté marcado. Después, se actualiza el puntero al mínimo si es que corresponde.

  En el procedimiento descrito se crea una cierta cantidad t_{i*} de árboles nuevos. Cada uno de ellos – excepto el primero posiblemente – estaba marcado originalmente, pero al transformarse en raíz será desmarcado. Además, puede que el nodo padre del elemento decrementado haya sido marcado, por lo tanto, la cantidad de nodos marcados aumenta en $1 - (t_{i*} - 1)$. Luego:

  $\hat{c_i} = c_i + \Delta\phi_i = t_{i*} + (t_i + t_{i*} + 2 \cdot (m_i + 1 - (t_{i*} - 1)) - (t_i + 2 \cdot m_i))$

  $\Rightarrow \hat{c_i} = 4$

  A partir de ello es pertinente deducir que $c_i = 4$. Consecuentemente el costo amortizado de **decrease-key**(x, k') para un montículo de Fibonacci es $\frac{O(1) \cdot k}{k} = O(1)$. ■

D. Especificaciones técnicas del equipo usado para los experimentos

Los experimentos fueron llevados a cabo en un notebook Lenovo 14"330S con las siguientes especificaciones técnicas:

- Procesador: Intel i7-8550U
- RAM: 1x8GB DDR4 2400MHz + 1x4GB DDR4 2400MHz
- Almacenamiento: Crucial P1 500GB
- SO: Manjaro 20.0.3 Gnome x64
- Versión Python: 3.8.6

E. Pseudo-código generación de grafos

A continuación se presenta el pseudo-código correspondiente a la función que se implementó para la generación de grafos aleatorios dada una cantidad de nodos v y una densidad de grafo ρ.

Algoritmo 2 Generador de gráficos aleatorios

```
1:  procedure GRAPH-GENERATOR(v, ρ)
2:      graph ← {}
3:      for i ∈ range(1, v) do
4:          weight ← random.randint(1, 10^9)
5:          graph[i][i + 1] ← weight
6:          graph[i + 1][i] ← weight
7:      for i ∈ range(1, v − 1) do
8:          for j ∈ range(i + 2, v + 1) do
9:              if random.uniform(0, 1) < ρ then
10:                 weight ← random.randint(1, 10^9)
11:                 graph[i][j] ← weight
12:                 graph[j][i] ← weight
        return graph
```

F. Pseudo-código generación de experimentos

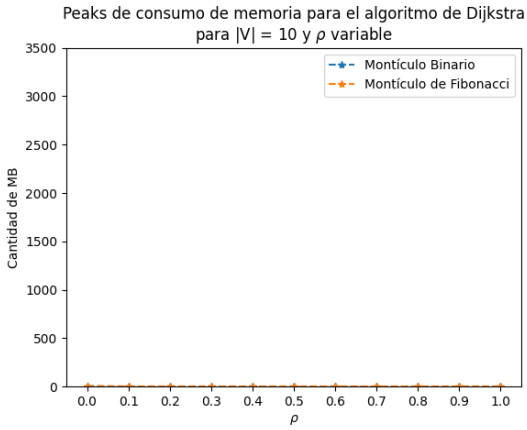
Más adelante se muestra el pseudo-código del método que fue implementado para generar los experimentos realizados.

Algoritmo 3 Generador de experimentos

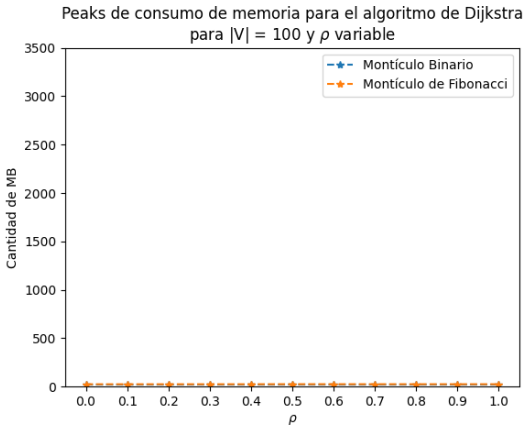
```
1:  procedure EXPERIMENTS-GENERATOR()
2:      results ← {fibonacci : {}, binary : {}}
3:      ρRange ← [0, 0,1, 0,2, 0,3, 0,4, 0,5, 0,6, 0,7, 0,8, 0,9, 1]
4:      vRange ← [10, 50, 100, 500, 1000, 5000, 10000]
5:      for v ∈ vRange do
6:          currentFibonacci ← {}
7:          currentBinary ← {}
8:          for ρ ∈ ρRange do
9:              fibonacciTimes ← []
10:             binaryTimes ← []
11:             for i ∈ range(3) do
12:                 graph, start ← generator(v, ρ)
13:                 ti ← time.time()
14:                 dijkstra(graph, start, FibonacciHeap(v))
15:                 tf ← time.time()
16:                 fibonacciTimes.append(tf − ti)
17:                 ti ← time.time()
18:                 dijkstra(graph, start, BinaryHeap(v))
19:                 tf ← time.time()
20:                 binaryTimes.append(tf − ti)
21:             fibonacciMean ← mean(fibonacciTimes)
22:             binaryMean ← mean(binaryTimes)
23:             currentFibonacci[ρ] ← fibonacciMean
24:             currentBinary[ρ] ← binaryMean
25:         results[fibonacci][v] ← currentFibonacci
26:         results[binary][v] ← currentBinary
        return results
```

G. Consumo de memoria del algoritmo de Dijkstra para ambos montículos

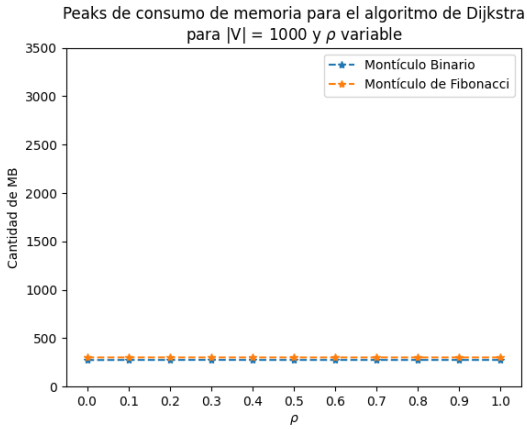
Se procede a mostrar un gráfico que detalla el consumo de memoria por parte del algoritmo de Dijkstra para distintos valores de ρ, $|V|$ y los montículos en estudio.



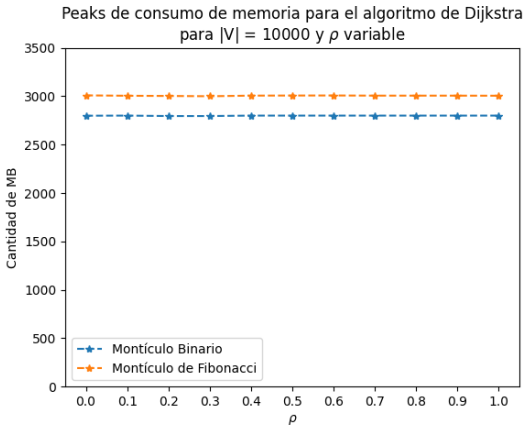
(a) |V| = 10



(b) |V| = 100



(c) |V| = 1000



(d) |V| = 10000

Figura 2: Peaks de consumo de memoria por parte del algoritmo de Dijkstra para cantidad de nodos |V|.

Es posible observar que el consumo de memoria explota para |V| = 10000 para ambos montículos.