/tikz/,/tikz/graphs/
conversions/canvas coordinate/.code=1 , conversions/coordinate/.code=1

Laboratorio 9 | Fibonacci Heaps

Geordie Quiroa

November 1, 2018

1 - Spanning Trees

Caso en el que el algoritmo de kruskal devolvería un spanning tree distinto para el mismo grafo

Esto sucedería bajo el escenario en el cual, en un grafo dirigido, existen dos o más aristas que relacionan distintos nodos con el mismo peso, siendo este peso el menor del grafo. Esto daría lugar a que se genere un MST distinto dependiendo de la relación que se utilice para iniciar el árbol.

Rapidez para computar un nuevo MST al agregar un vértice al grafo

Para determinar la rapidez es importante tomar en cuenta cuatro factores: la naturaleza del grafo (si es dirgido o no), el número total de vértices y nodos del grafo, el número de relaciones para este nuevo nodo; aunque este factor no influye mucho, pero sí incide en las comparaciones para determinar la de menor peso, y lo más importante: el algoritmo utilizado para graficar el MST. La importancia del último factor descrito radica específicamente en la complejidad de los algoritmos, siendo la complejidad de Kruskal $\mathcal{O}(E\log(V))$, y la de Prim (sin implementar fibonacci heaps) es $\mathcal{O}(V\log(V)+E\log(V))$.

Tomando en cuenta la complejidad de cada algoritmo y los factores mencionados, es posible concluir que el algoritmo de Kruskal es más rapido que el de Prim en el escenario donde el grafo es poco denso, es decir, la cantidad de nodos y vértices no varían significativamente, el grafo no es dirigido y el vértice nuevo tiene sólo una relación (con el fin de que "E" no tenga un impacto significativo en la complejidad). Siendo "E" el total de vértices del grafo previo a agregar el nodo, más la cantidad de relaciones que tiene el nuevo vértice; que en este caso es uno (E = E+1) y "V" es el número total de vértices que tenía el grafo anterior, más el vértice que se añadió (V = V +1). Estas condiciones permiten comparar la complejidad de cada algoritmo, y determinar que $\mathcal{O}(E\log(V)) < \mathcal{O}(V\log(V)+E\log(V))$, consecuentemete Kruskal es más rapido que Prim (sin fibonacci heaps).

Por otro lado, el algoritmo de Prim es más rapido que Kruskal en el escenario donde el grafo es bastante denso, es decir la cantidad de relaciones "E" es significativamente mayor que la cantidad de nodos "V", no es dirgido y el nodo a añadir tiene n cantidad de relaciones. En este escenario, la cantidad de relaciones "E" es el total que tenía el grafo previo a agregar el nodo, más las n rela-

ciones que presenta el nodo a añadir (E = E + n). En cambio, la cantidad de vertices "V" sólo varía en una unidad (V = V + 1). Por lo que estas condiciones permiten comparar la complejidad de cada algoritmo y determinar que $\mathcal{O}(V\log(V)+E\log(V)) < \mathcal{O}(E\log(V))$, consecuentemente el alogritmo de Prim (sin fibonacci heaps) es más rapido que Kruskal. En este escenario, es posible mejorar aún más la rapidez del algoritmo de Prim a través de la implementación de fibonacci heaps, esto haría que la complejidad pasara de ser $\mathcal{O}(V\log(V)+E\log(V))$ a ser $\mathcal{O}(V\log(V)+E)$.

2 - Fibonacci Heaps

2.1 - Intuición detrás del potencial de las operaciones

$$\Phi(H) = t(H) + 2(m(H))$$

El potencial es el costo que podría llegar a representar cualquiera de las operaciones del fibonacci heap, y por lo tanto debe poder cubrir el costo del trabajo de las operaciones. Este potencial se basa en los dos estados del fibonacci heap, un estado incial (previo a realizar cualquier operación), y uno final (el estado resultante de la operación realizada), consecuentemente cada operación da lugar a una diferencia de potencial. La función potencial se deriva a partir de los cambios observados en dos elementos/características del fibonacci heap a lo largo de las operaciones, siendo estos dos elementos: el root list; ya que luego de cada operación el número de nodos en el root list t(H) se ve afectado por la operación, y el número de nodos marcados en el árbol m(h); debido a que estos cambian en cada operación luego de que el nodo padre cambié de posición en el heap o cambie el número de nodos hijos. En la función potencial, t(H) hace referencia al número inicial de nodos en el root list, y el máximo trabajo (potencial) resultante, es decir, los nodos en el root list resultantes, y cómo este número cambia en cada operación. El cambio de t(h) en cada operación sucede de la siguiente forma: inserting a node (t(H) = t(H)+1), extract-min (t(H) = D(n) + t(H) + D(n) + 1) donde D(n) es el número de nodos hijos que tiene el nodo n, y decrease-key (t(h) = t(h)+c) en donde c es la cantidad de veces que se llamó recursivamente la función cascade-cut la cual corta los nodos marcados que pertenecen al árbol del nodo que se extrajo y los coloca en la raíz. Por este motivo, t(H) forma parte de la función potencial, ya que en cada una de las operaciones del fibonacci-heap el trabajo a realizar está relacionado al número de nodos en el root list. Por otro lado, el 2(m(H)) se incluye con el fin de cubrir el trabajo que se lleva a

cabo en decrease-key ya que a lo largo de las llamadas recursivas de cascade-cut u otras operaciones puede que existan nodos marcados a los que haya que limpiarle la marca, lo que reduce el potencial por 2, una unidad paga por el corte y limpieza del flag "marked" (corta la relación entre nodo padre y nodo hijo marcado y lo pasa al root con el flag limpio), y la otra unidad compensa el incremento en el potencial generado por el hecho de pasar el nodo hijo al root list. Por estos motivos, la función potencial está compuesta por los elementos t(h) y la constante que duplica el total de los nodos marcados.

2.2 - Potencial Fibonacci heaps

$$\Phi(H) = t(H) + 2(m(H))$$

Potencial de fibonacci heap figure 1

- t(H) = 5
- m(H) = 3
- $\Phi(H) = 5 + 2(3) = 11$

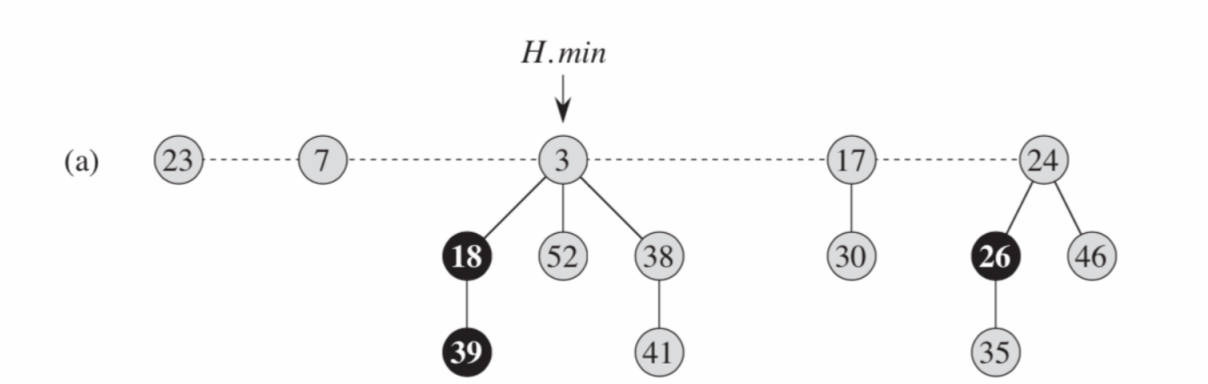

Figure 1: Pseudo código para operar decrease key en Fibonacci heap

Potencial de fibonacci heap figure 2

- t(H) = 3
- m(H) = 3
- $\Phi(H) = 3 + 2(3) = 9$

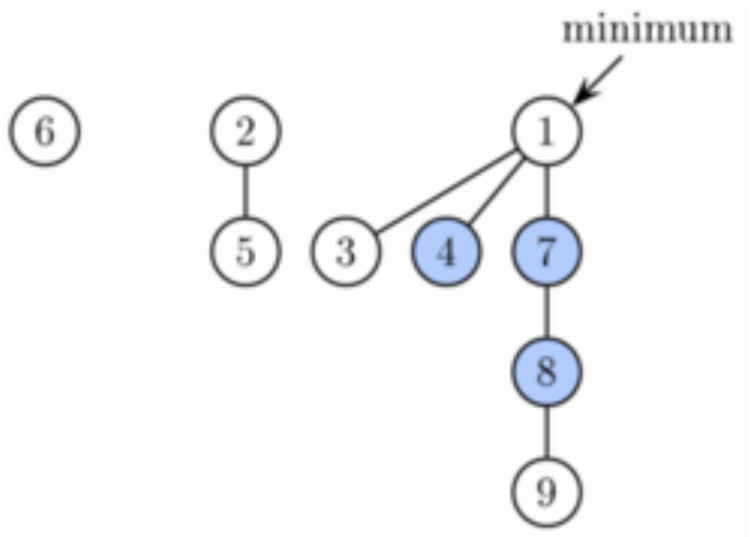

Figure 2: Fibonacci heap H para segundo cálculo de potencial

Potencial de fibonacci heap figure 3

- t(H) = 4
- m(H) = 1
- $\Phi(H) = 4 + 2(1) = 6$

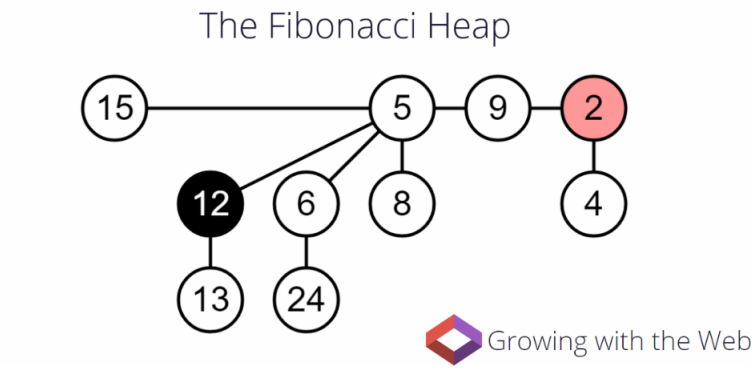

Figure 3: Fibonacci heap H para tercer cálculo de potencial

3 - Trace de decrease key para fibonacci heap

```
FIB-HEAP-DECREASE-KEY(H, x, k)
1  if k > x.key
2      error "new key is greater than current key"
3  x.key = k
4  y = x.p
5  if y ≠ NIL and x.key < y.key
6      CUT(H, x, y)
7      CASCADING-CUT(H, y)
8  if x.key < H.min.key
9      H.min = x

CUT(H, x, y)
1  remove x from the child list of y, decrementing y.degree
2  add x to the root list of H
3  x.p = NIL
4  x.mark = FALSE

CASCADING-CUT(H, y)
1  z = y.p
2  if z ≠ NIL
3      if y.mark == FALSE
4          y.mark = TRUE
5      else CUT(H, y, z)
6          CASCADING-CUT(H, z)
```

Figure 4: Pseudocódigo para operar decrease key en Fibonacci heap

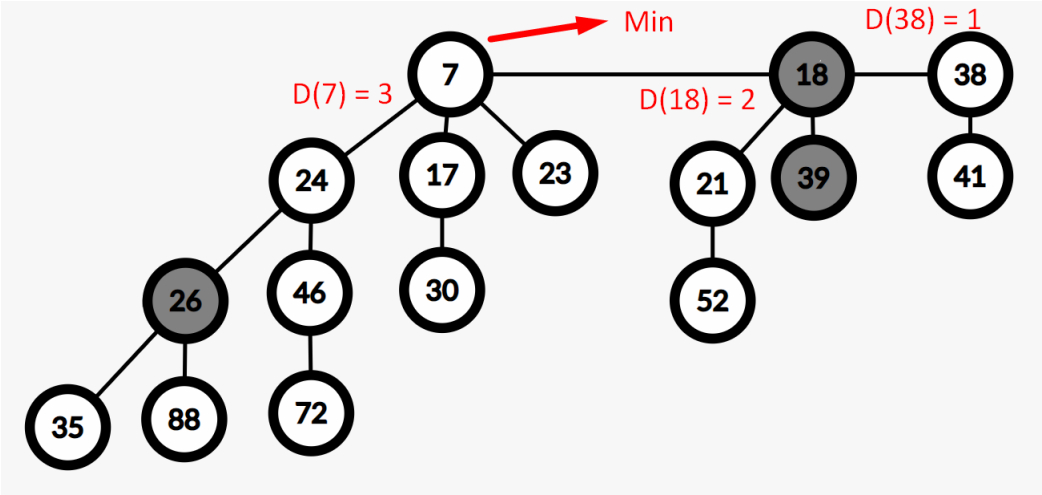

Figure 5: Fibonacci heap Inicial

Fib-heap-decrease-key(H, 24, 12)

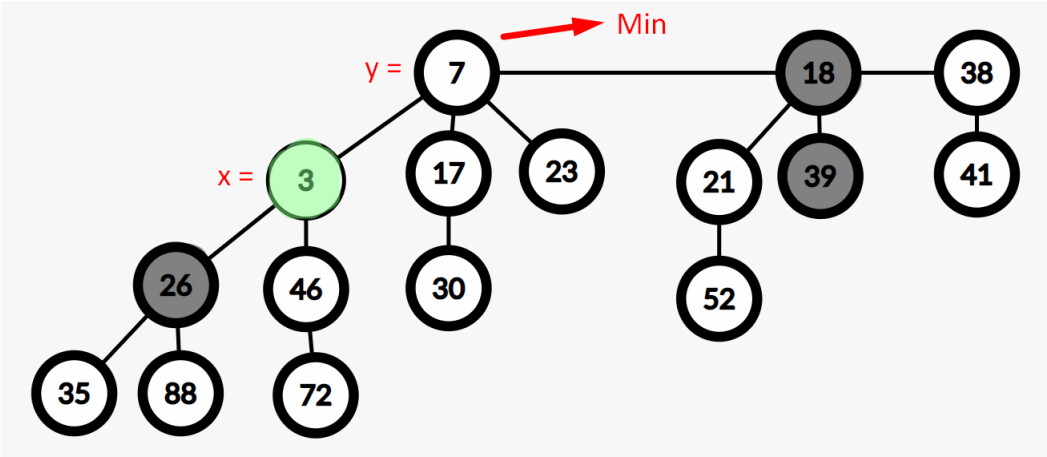

Figure 6: Cambio de key 24 a 3

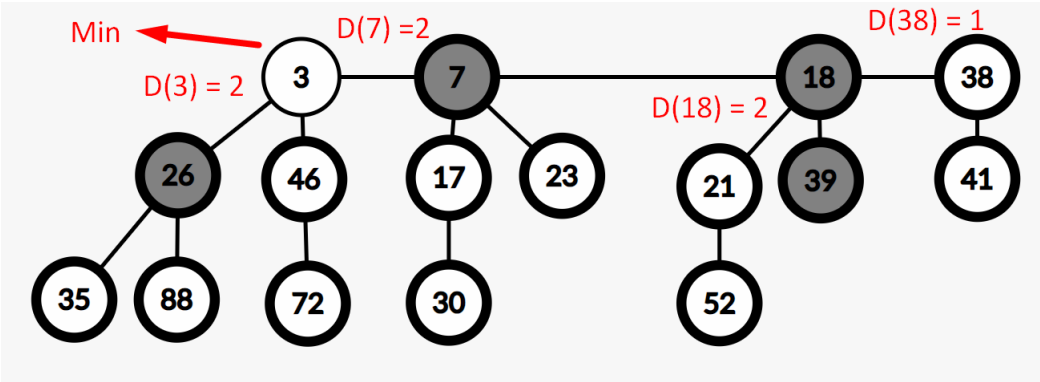

Figure 7: Cut, cascading-cut y H.min = x