Ayudantía 9 - Algoritmos y Complejidad
Union Find - Hashing

Sassy complexes

1. Union Find

El algoritmo de Union Find o Disjoint Set se utiliza generalmente cuando se trabaja con clases de equivalencia, en donde definimos una relación de equivalencia y particionamos los elementos en conjuntos implementados mediante árboles de acuerdo a esta relación. En esta representación el nodo raíz del árbol es el "representante" de la clase, es decir, lo utilizamos para identificar los elementos pertenecientes a la clase.

1.1. Algoritmo

Utilizamos un arreglo de elementos, cada elemento debe tener una referencia al padre. Al inicio el padre es el mismo nodo, lo que indica que es la raíz de ese árbol (al inicio solo hay árboles de un nodo). Posteriormente se utilizan las operaciones de *Union* y *Find*. *Find* retorna el padre del arbol y *Union* une dos árboles, por lo que se usan para comparar y unir clases de equivalencia respectivamente. Para esta ultima hay que tomar en cuenta el *rank* o profundidad máxima de cada árbol, ya que es deseable evitar que este valor crezca mucho, y para esto se debe procurar

1.1.1. Primera Implementación

```
function Find(x)
   if x.parent != x
     x.parent := Find(x.parent)
   return x.parent

function Union(x, y)
   xRoot := Find(x)
   yRoot := Find(y)

   // x and y are already in the same set
   if xRoot == yRoot
       return

   // x and y are not in same set, so we merge them
   if xRoot.rank < yRoot.rank
     xRoot.parent := yRoot
   else if xRoot.rank > yRoot.rank
     yRoot.parent := xRoot
```

```
else
  //Arbitrarily make one root the new parent
  yRoot.parent := xRoot
  xRoot.rank   := xRoot.rank + 1
```

1.2. Path Compression

El problema con este algoritmo es que en su versión básica tanto el *Union* como el *Find* tienen complejidad lineal, por lo que si debo realizar muchas de estas operaciones el tiempo de ejecución crece mucho. Esto puede remediarse con una técnica llamada *Path Compression*. Esta técnica consiste en que en cada *Find* se cambia el padre de todos los nodos que se recorren por la raíz del árbol.

2. Hashing

Una tabla hash es una estructura de datos que nos permite determinar si un item esta o no esta en un conjunto. La idea es agarrar una funcion h que mapee nuestro elemento del conjunto a un pequeño entero $h(x)$. Entonces guardamos ese elemento en el casillero $h(x)$ de la tabla. Notó lo que paso aqui?

Ahora la idea es tener una tabla donde podamos acceder a un item en tiempo "amortizado" constante (como el diccionario en Python), hay varias formas de implementar esto.

Primero, queremos guardar n items, cada item es un elemento de un universo U, siendo u el tamaño del universo. Una tabla hash es un arreglo $A[1..m]$ donde m es el "tamaño de la tabla". Es normal que m sea muuucho mas pequeño que u. Una funcion Hash es una funcion de la forma:

$$h : U \to 0, 1, \ldots, m-1$$

Que mapea todos los posibles elementos de U a un slot de la tabla hash. Un item x se "hashea" al slot $T[h(x)]$.
Que pasa cuando $u = m$? que pasa cuando $u > m$? existe $u < m$?? Obviamente el caso trivial es $u = m$, donde $h(x) = x$, una tabla de acceso directo, pero esto requiere mucho espacio, y pocas veces necesitamos manejar mas que una pequeña fraccion de U. En cambio, al usar una tabla mas pequeña tenemos que lidiar con las colisiones, decimos que x e y colisionan si sus hashes son iguales $h(x) = h(y)$, es necesario diseñar una buena funcion hash que reduzca las colisiones y sea veloz de ejecutar (poco overhead).

Hay que tomar decisiones:

Direccionamiento Cerrado Los items que colisionan se almacenan en una estructura de datos secundaria, como una lista.

Direccionamiento Abierto Al colisionar, uno de los elementos se va a alguna posicion libre

Hashing Perfecto Si conocemos los itemes a almacenar, elejimos un h conveniente.

Una familia universal de funciones basicas pero utiles : Considere un p primo, $1 \leq a < p$ y $0 \leq b < p$

$$h_{a,b}(x) = ((ax+b) \mod p) \mod m$$

La probabilidad de colision es:

$$P(h_{a,b}(x) = h_{a,b}(y)) \leq \frac{1}{m}$$

3. Ejercicios

3.1. Pregunta 1

Demuestre que después de una cantidad arbitraria de operaciones unión (sin *Path Compression*), la complejidad en el peor caso de *find* o de *union* es $O(\log n)$.

3.2. Pregunta 2

Suponga que se tiene una secuencia de bits $X[1..n]$, los cuales son inicialmente 1, sobre la que se pueden realizar sólo las siguientes operaciones:

- **Lookup**(i): Entrega el bit en la posición i.
- **Blacken**(i): Deja en 0 el bit en la posición i, no se puede aplicar en el último bit, cuando $i = n$.
- **NextWhite**(i): Entrega la posición $j \leq i$ del siguiente bit que vale 1 en la secuencia.

Describa una implementación en que el costo de cada operación sea mejor que $O(n)$ y demuestre que ese es el caso.

3.3. Pregunta 3

Un filtro de Bloom es una estructura probabilista eficiente para almacenar un conjunto. Dado un universo de elementos $\mathcal{U}$, un tamaño de tabla m (una tabla de m bits, numerados 0 a $m-1$, inicialmente todos 0) y k funciones hash $h_i : \mathcal{U} \to [0, m-1]$ (que suponemos ideales e independientes) para registrar que $x \in \mathcal{U}$ está en el conjunto ponemos en 1 los bits de la tabla en las posiciones $h_i(x)$ para $1 \leq i \leq k$.

1. Halle la probabilidad que luego de insertar n elementos, un bit dado sigue en 0.
2. Halle la probabilidad de un falso positivo (los bits en las posiciones $h_i(x)$ son todos 1, pero no se ha agregado x al conjunto) cuando el conjunto contiene n elementos.

4. Soluciones

4.1. Pregunta 1

La complejidad de una operación *find* está determinada por la altura del árbol que representa el conjunto, por lo tanto hay que demostrar que esta altura no supera $O(\log n)$.

En primer lugar hay que recordar que el *rank* de un nodo representa la altura que tiene un nodo (cuando no hay *Path Compression*) y sólo sube cuando se realiza una operación *union* entre dos árboles de igual *rank*. Para crear un árbol con raíz *rank* 1 es necesario unir dos árboles con raíz *rank* 0, para crear un árbol con raíz *rank* 2, es necesario unir dos árboles con raíz *rank* 1, y así sucesivamente.

Entonces, es posible demostrar por inducción, que se necesitan al menos 2^n nodos para lograr un árbol con raíz *rank* n.

- **Caso base**: Es fácil ver que se cumple para $n = 0$, pues para tener un árbol raíz *rank* 0 se necesita tener un nodo (la raíz).
- **Hipótesis inductiva**: Supongamos que se cumple que un árbol raíz *rank* p tiene al menos 2^p nodos.
- **Tesis inductiva**: La única forma de lograr un árbol raíz *rank* $p+1$ es uniendo dos árboles raíz *rank* p, por hip. inductiva, cada uno tiene al menos 2^p nodos, por lo que el nuevo árbol tiene al menos 2^{p+1} nodos, lo que ser quería demostrar.

Como un árbol no puede tener más de n nodos (este sería el caso en que todo el universo esté en el mismo árbol), la altura máxima que puede tener un árbol es $\log_2 n$, y esto limita el costo de una operación *find*.

Por otro lado, la operación *union* consta de dos operaciones *find* y el resto es $O(1)$, por lo tanto esta también es $O(\log n)$.

4.2. Pregunta 2

Básicamente UnionFind sin rank y con bits, donde blacken es Union y nextWhite es find. Todas las operaciones tienen complejidad temporal $O(1)$.