CI2613: Algoritmos y Estructuras III

Blai Bonet

Universidad Simón Bolívar, Caracas, Venezuela

Enero-Marzo 2015

Estructura de datos para conjuntos disjuntos (Union-Find)

Conjuntos disjuntos

Dado un universo de elementos $U = \{u_1, u_2, \ldots, u_n\}$

Se quiere tener una estructura de datos que almacene una colección $\{S_1, S_2, \ldots, S_m\}$ de subconjuntos **no vacíos y disjuntos** de U

Dicha colección es **dinámica**; i.e. cambian a lo largo de la ejecución del programa

Cada S_i es representado por un elemento del subconjunto al que llamamos el **representante de** S_i

ED para conjuntos disjuntos

La ED debe soportar la siguientes operaciones:

- `make-set`(x): crea el conjunto $\{x\}$ en la colección cuyo representante es x
- `union`(x, y): dados dos elementos x, y del universo, **reemplaza** los **subconjuntos disjuntos** S_i and S_j que contienen a x y y respectivamente, por el subconjunto $S_i \cup S_j$. El representante del nuevo subconjunto $S_i \cup S_j$ es un elemento arbitrario del mismo
- `find`(x): retorna un apuntador al único subconjunto en la colección que contiene a x

ED para conjuntos disjuntos: Ejemplo

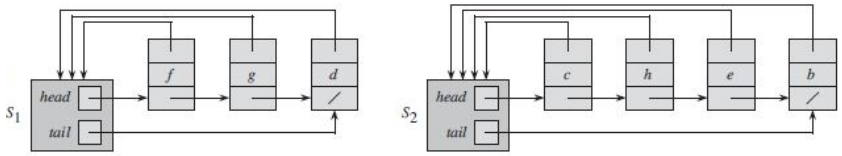
$$U = \{a, b, c, d, e, f, g, h, i, j\}$$

`make-set`(a)	
`make-set`(f)	
`make-set`(c)	
`union`(a, f)	
	$\{\mathbf{a}, f\}$ $\{\mathbf{c}\}$
.....	
	$\{\mathbf{a}, f\}$ $\{\mathbf{b}, h, j\}$ $\{\mathbf{c}\}$ $\{\mathbf{d}, e\}$ $\{\mathbf{g}\}$ $\{\mathbf{i}\}$
`union`(e, h)	
`union`(g, i)	
	$\{\mathbf{a}, f\}$ $\{b, d, e, h, \mathbf{j}\}$ $\{\mathbf{c}\}$ $\{\mathbf{g}, i\}$

Primera implementación: Listas

Podemos utilizar listas enlazadas para implementar la ED

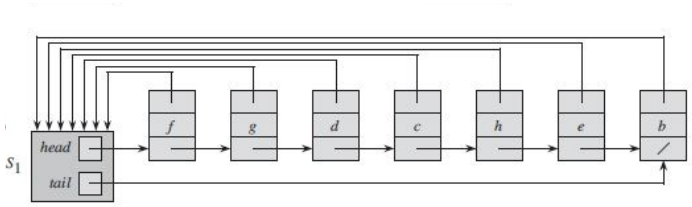
Ejemplo: $\{b, \mathbf{c}, e, h\}$ $\{d, \mathbf{f}, g\}$



Primera implementación: Listas

Podemos utilizar listas enlazadas para implementar la ED

Ejemplo: $\{b, c, d, e, \mathbf{f}, g, h\}$



Análisis de implementación con listas

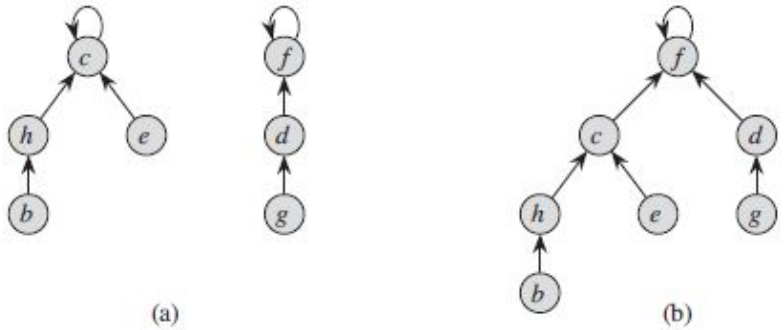
Para un secuencia σ arbitraria de $O(n)$ operaciones, se puede necesitar hasta $\Theta(n^2)$ unidades de tiempo para ejecutar σ dando un **tiempo amortizado** de $O(n)$ unidades de tiempo por operación

operación	unidades de tiempo requeridos
`make-set`(x_1)	1
`make-set`(x_2)	1
...	...
`make-set`(x_n)	1
`union`(x_2, x_1)	1
`union`(x_3, x_2)	2
`union`(x_4, x_3)	3
...	...
`union`(x_n, x_{n-1})	$n - 1$

Segunda implementación: Bosque de árboles

Ejemplo: (a) $\{b, \mathbf{c}, e, h\}$ $\{d, \mathbf{f}, g\}$

(b) $\{b, c, d, e, \mathbf{f}, g, h\}$



Segunda implementación: Pseudocódigo

Asociamos a cada elemento un "apuntador" al padre en el árbol:

```
1  void make-set(x)
2      π[x] = x
3
4  void union(x,y)
5      link(find(x), find(y))
6
7  void link(x, y)
8      π[x] = y
9
10 T find(x)
11     if x == π[x]
12         return x
13     else
14         return find(π[x])
```


Análisis de implementación con bosque de árboles

Nada evita que se forme un árbol con una sóla rama de profundidad n, lo que da un tiempo amortizado de $\Theta(n)$ unidades por operación

Mejoras (heurísticas):

- Unión por rango
- Compresión de caminos

Unión por rango

Para cada nodo en el árbol mantenemos un entero `rank` que acota la altura del nodo en el árbol:

```
1  void make-set(x)
2      π[x] = x
3      rank[x] = 0
4
5  void union(x,y)
6      link(find(x), find(y))
7
8  void link(x, y)
9      if rank[x] > rank[y]
10         π[y] = x
11     else
12         π[x] = y
13         if rank[x] == rank[y]
14             rank[y] = rank[y] + 1
```

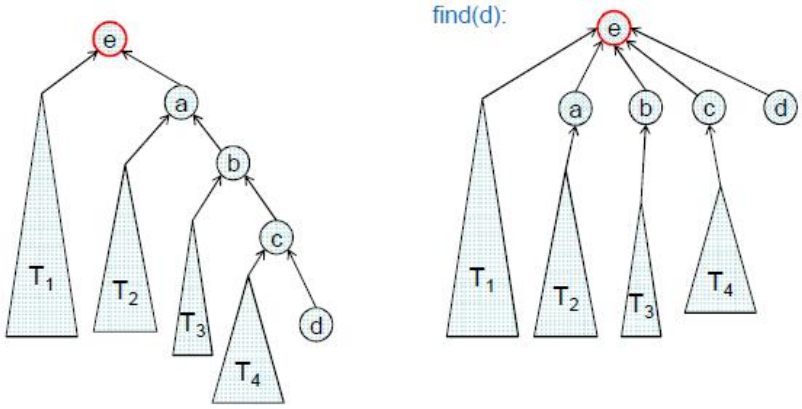

Compresión de caminos

Cada vez que hacemos un `find`(x), colocamos a x (y todos sus **ancestros**) como hijos de la raíz del ábol al cual pertenece x

La idea es que los proximos `find` de x tomen menos tiempo

```
1  T find(x)
2      if x != π[x]
3          π[x] = find(π[x])
4      return π[x]
```


Compresión de caminos: Ejemplo



Implementación final: Pseudocódigo

```
1  void make-set(x)
2      π[x] = x
3      rank[x] = 0
4
5  void union(x,y)
6      link(find(x), find(y))
7
8  void link(x, y)
9      if rank[x] > rank[y]
10         π[y] = x
11     else
12         π[x] = y
13         if rank[x] == rank[y]
14             rank[y] = rank[y] + 1
15
16 T find(x)
17     if x != π[x]
18         π[x] = find(π[x])
19     return π[x]
```


Análisis: Resumen

Para una secuencia de m operaciones, con f `find`, sobre un universo de n objetos, se puede mostrar lo siguiente:

- Al utilizar sólo unión por rango: $\Theta(m \log n)$
- Al utilizar sólo compresión de caminos: $\Theta(n + f \cdot (1 + \log_{2+f/n} n))$
- Al utilizar las dos heurísticas: $O(m\,\alpha(n))$

donde la función $\alpha(n)$ es de crecimiento muy muy . . . lento

Crecimiento de $\alpha(n)$

Considere $f(n) = \begin{cases} 1 & \text{si } n = 0 \\ 2 \uparrow\uparrow n & \text{si } n > 0 \end{cases}$ donde $2 \uparrow\uparrow n = \underbrace{2^{2^{2^{\cdot^{\cdot^{\cdot}}}}}}_{n\ 2\text{'s}}$

n	$f(n) = 2 \uparrow\uparrow n$
0	$2^0 = 1$
1	$2^1 = 2$
2	$2^2 = 4$
3	$2^4 = 16$
4	$2^{16} = 65536$
5	2^{65536}

La función $\alpha(n)$ es la **inversa** de $f(n)$ (también denotada por $\log^*(n)$)

Para cualquier propósito práctico, $\alpha(n)$ es constante!

Aplicación: Componentes conexas

Considere un grafo **no dirigido** $G = (V, E)$

Una **componente conexa** de G es un subconjunto de vértices $C \subseteq V$ tal que:

- para todo $x, y \in C$, existe un camino de x a y (y viceversa porque G es no dirigido)
- C es maximal

Como en el caso de las componentes fuertemente conectadas para grafos dirigidos, el conjunto de vértices se particiona de forma $\{C_1, C_2, \ldots, C_n\}$ donde cada C_i es una componente conexa

Cálculo de la componentes conexas

```
1  void componentes-conexas
2      % inicialización
3      foreach vértice x
4          make-set(x)
5          componente[x] = new ArregloDinamico
6
7      % cálculo de componentes
8      foreach arista (u,v)
9          if find(u) != find(v)
10             union(u,v)
11
12     % guardar componentes
13     foreach vértice x
14         componente[find(x)].push-back(x)
```


Análisis de Union-Find

Observaciones preliminares sobre rangos

Invariantes sobre los rangos para un universo U con n objetos:

1. Si x no es raíz, su rango es menor estricto al rango de su padre

---

Prueba: Por inducción en el # operaciones en $\sigma = (op_1, op_2, \ldots, op_k)$

1 es cierto antes de op_1. Asuma 1 cierto después de ejecutar op_{i-1}

Si $op_i = \texttt{make-set}(x)$, 1 se cumple después de op_i

Si $op_i = \texttt{find}(x)$, 1 se cumple después de op_i porque ciertos nodos son hechos hijos de la raíz la cual tiene el mayor rango en el árbol

Si $op_i = \texttt{union}(x, y)$ y $rank[x] < rank[y]$, x es hecho hijo de y y 1 se cumple después de op_i

Si $op_i = \texttt{union}(x, y)$ y $rank[x] > rank[y]$, similar al caso anterior

Si $op_i = \texttt{union}(x, y)$ y $rank[x] = rank[y]$, x es hecho hijo de y y se incrementa $rank[y]$. Por lo tanto, 1 se cumple después de op_i

□

Observaciones preliminares sobre rangos

Invariantes sobre los rangos para un universo U con n objetos:

1. Si x no es raíz, su rango es menor estricto al rango de su padre
2. Para cualquier x y $S_x =$ "items en subárbol de x", $|S_x| \geq 2^{rank[x]}$

Prueba: Por inducción en el # operaciones en $\sigma = (op_1, op_2, \ldots, op_k)$

2 es cierto antes de op_1. Asuma 2 cierto después de ejecutar op_{i-1}

Si $op_i = \texttt{make-set}(x)$, 2 se cumple después de op_i

Si $op_i = \texttt{find}(x)$, 2 se cumple porque ningún subconjunto cambia

Si $op_i = \texttt{link}(x, y)$ y $rank[x] < rank[y]$, x es hecho hijo de y y

$$|S'_y| = |S_x| + |S_y| \geq 2^{rank[x]} + 2^{rank[y]} \geq 2^{rank[y]}$$

Si $op_i = \texttt{link}(x, y)$ y $rank[x] = rank[y]$, x es hecho hijo de y y

$$|S'_y| = |S_x| + |S_y| \geq 2^{rank[x]} + 2^{rank[y]} = 2^{rank[y]+1} = 2^{rank'[y]}$$ □

Observaciones preliminares sobre rangos

Invariantes sobre los rangos para un universo U con n objetos:

1. Si x no es raíz, su rango es menor estricto al rango de su padre
2. Para cualquier x y $S_x =$ "items en subárbol de x", $|S_x| \geq 2^{rank[x]}$
3. El mayor rango posible es $\lfloor \log_2 n \rfloor$

Prueba: Suponga que existe x con rango $\geq \lfloor \log_2 n \rfloor + 1$

Si $\log_2 n$ es entero, $2^{\lfloor \log_2 n \rfloor + 1} = 2^{1 + \log_2 n} > n$

Si $\log_2 n$ no es entero, $2^{\lfloor \log_2 n \rfloor + 1} = 2^{\lceil \log_2 n \rceil} > n$

En ambos casos, por 2, $|S_x| > n$ que es imposible ya que $|U| = n$ □

Observaciones preliminares sobre rangos

Invariantes sobre los rangos para un universo U con n objetos:

1. Si x no es raíz, su rango es menor estricto al rango de su padre
2. Para cualquier x y $S_x =$ "items en subárbol de x", $|S_x| \geq 2^{rank[x]}$
3. El mayor rango posible es $\lfloor \log_2 n \rfloor$
4. Sólo las raíces pueden cambiar rango

Prueba: Trivial ya que la única operación que cambia rangos es `link` y lo cambia sobre raíces □

Observaciones preliminares sobre rangos

Invariantes sobre los rangos para un universo U con n objetos:

1. Si x no es raíz, su rango es menor estricto al rango de su padre
2. Para cualquier x y $S_x =$ "items en subárbol de x", $|S_x| \geq 2^{rank[x]}$
3. El mayor rango posible es $\lfloor \log_2 n \rfloor$
4. Sólo las raíces pueden cambiar rango
5. Existen a lo sumo $\frac{n}{2^r}$ objetos con rango igual a r

---

Prueba: No es difícil ver que dos objetos distintos x y y con el mismo rango tienen descendientes disjuntos; i.e., $S_x \cap S_y = \emptyset$ si $rank[x] = rank[y]$

Sean $x_1, x_2, \ldots, x_k$ los elementos en U con rango r:

$$n \;\geq\; \sum_{i=1}^{k} |S_{x_i}| \;\geq\; \sum_{i=1}^{k} 2^{rank[x_i]} \;=\; k2^r$$

Por lo tanto, $k \leq n/2^r$ □

Observaciones preliminares sobre rangos

Invariantes sobre los rangos para un universo U con n objetos:

1. Si x no es raíz, su rango es menor estricto al rango de su padre
2. Para cualquier x y $S_x =$ "items en subárbol de x", $|S_x| \geq 2^{rank[x]}$
3. El mayor rango posible es $\lfloor \log_2 n \rfloor$
4. Sólo las raíces pueden cambiar rango
5. Existen a lo sumo $\frac{n}{2^r}$ objetos con rango igual a r

Partición de objetos en bloques

Particionamos los objetos en bloques asignando el objeto x al bloque $\log^*(rank[x])$

Como los rangos varían en $\{0, 1, \ldots, \lfloor \log_2 n \rfloor\}$ (por 3), los índices de bloques varían en $\{0, 1, \ldots, \log^*(\lfloor \log n \rfloor) = \log^*(n) - 1\}$

Por lo tanto, existen $\log^*(n)$ bloques distintos

Cota sobre el número total de objetos en bloque b (usando 5):

$$\sum_{r=2\uparrow\uparrow(b-1)+1}^{2\uparrow\uparrow b} \#\text{obj c/rango } r \;\leq\; \sum_{r=2\uparrow\uparrow(b-1)+1}^{2\uparrow\uparrow b} \frac{n}{2^r} \;<\; \sum_{r=2\uparrow\uparrow(b-1)+1}^{\infty} \frac{n}{2^r}$$

$$= \frac{n}{2^{2\uparrow\uparrow(b-1)}} \;=\; \frac{n}{2 \uparrow\uparrow b}$$

Secuencias de operaciones

Dada una secuencia σ de operaciones sobre la ED, podemos reemplazar cada `union` por dos `find` y un `link`

Por lo tanto, consideramos sólo secuencias σ con m operaciones de tipo `make-set`, `link` y `find`

`make-set` y `link` toman tiempo constance

Así que nos enfocamos en las $O(m)$ operaciones de tipo `find`

Análisis agregado en diferentes cuentas

Considere un operación `find`(x_0) y los objetos $x_0, x_1, \ldots, x_\ell$ en el camino desde x_0 a la raíz x_ℓ

El costo de dicha operacion es proporcional a $1 + \ell$ unidades de tiempo: 1 unidad por cada objeto x_i en el camino.

Dichos costos los **distribuimos en diferentes "cuentas"**:

- Cuenta ROOT
- Cuenta CHILD
- Cuenta BLOCK
- Cuenta PATH

Pagos

Camino $x_0, x_1, \ldots, x_{\ell-1}, x_\ell$ asociado a una operación `find`(x_0)

Pagos asociados a cada objeto x_i en el camino:

- x_ℓ paga 1 unidad a la cuenta ROOT
- $x_{\ell-1}$ paga 1 unidad a la cuenta CHILD
- Si x_i pertenece a un bloque distinto que su padre x_{i+1}, x_i paga 1 unidad a la cuenta BLOCK
- Si x_i pertenece al mismo bloque que su padre x_{i+1}, x_i paga 1 unidad a la cuenta PATH

Como todos los objetos pagan 1 unidad y el costo de `find`(x_0) es proporcional a $1 + \ell$, el **balance final** en las cuentas es proporcional al tiempo agregado utilizado en todas las operaciones `find`

Balance final en ROOT, CHILD y BLOCK

Pagos efectuados para una operación `find`(x_0):

- 1 unidad en la cuenta ROOT
- A lo sumo 1 unidad en la cuenta CHILD
- A lo sumo 1 unidad en la cuenta BLOCK por cada uno de los $\log^*(n)$ bloques

Después de $O(m)$ operaciones `find` tendremos:

- $O(m)$ unidades en la cuenta ROOT
- $O(m)$ unidades en la cuenta CHILD
- $O(m \log^*(n))$ unidades en la cuenta BLOCK

Total: $O(m \log^*(n))$ unidades en ROOT, CHILD y BLOCK

Balance final en PATH

Considere un objeto x_i en bloque b que paga en la cuenta PATH

- x_i no es raíz así que su rango nunca cambia (por ❹)
- Su padre x_{i+1} tampoco es raíz. Después de comprimir el camino, x_i obtiene un padre cuyo rango es $>$ al anterior (por ❶)
- Cada vez que x_i paga en PATH, el rango de su padre incrementa. Eventualmente, el padre cambia de bloque y x_i no vuelve a pagar en PATH
- Así, x_i paga en PATH a lo sumo una vez por cada rango (no objeto) perteneciente al bloque b. El número de dichos rangos es $\leq 2{\uparrow\uparrow}b$
- Como el bloque b contiene a lo sumo $n/2{\uparrow\uparrow}b$ objetos, y cada uno paga $\leq 2{\uparrow\uparrow}b$ unidades a PATH, el total de unidades en PATH pagadas por objetos del bloque b es a lo sumo n
- Existen $\log^*(n)$ bloques. El **balance final** en PATH es $\leq n \log^*(n)$

Costo amortizado por operación

Una secuencia de m operaciones la covertimos en una secuencia de $\Theta(m)$ operaciones de tipo `make-set`, `link` y `find`

El costo total de todas las operaciones en la secuencia es $O(m \log^*(n) + n \log^*(n))$

Para $m \geq n$ (el caso usual), el costo total es $O(m \log^*(n))$

Por lo tanto, el costo amortizado por operación es $O(\log^*(n))$ (**constante en la práctica!**)