UNION-FIND

IIUWr. II rok informatyki. *Opracował: Krzysztof Loryś*

40 Definicja problemu

Dany jest skończony zbiór U oraz ciąg σ instrukcji UNION i FIND:

- UNION(A, B, C); gdzie A, B - rozłączne podzbiory U;
  wynikiem instrukcji jest utworzenie zbioru C takiego, że $C \leftarrow A \cup B$, oraz usunięcie zbiorów A i B;
- FIND(i); gdzie $i \in U$;
  wynikiem instrukcji jest nazwa podzbioru, do którego aktualnie należy i.

Problem polega na zaprojektowaniu struktury danych umożliwiającej szybkie wykonywanie ciągów σ. Początkowo każdy element U tworzy jednoelementowy podzbiór.

40.1 Uwagi i założenia

- Zbiór U jest mały ($|U| \ll$ pojemność pamięci wewnętrznej). Zwykle przyjmuje się, że $U = \{1, \ldots, n\}$.
- Bardzo często σ zawiera cn instrukcji (c-stała).
- Rozważa się dwa sposoby wykonywania ciągów σ:
  - *on-line* - wynik każdej instrukcji musi zostać obliczony przed wczytaniem kolejnej instrukcji;
  - *off-line* - ciąg σ może być wczytany całkowicie zanim zostanie obliczony wynik którejkolwiek instrukcji.

  Nas interesować będzie sposób *on-line*.
- Często nazwy podzbiorów są nieistotne, a instrukcja FIND służy jedynie do stwierdzenia czy dane elementy należą do tego samego podzbioru.

41 Przykład zastosowania

41.1 Konstrukcja minimalnego drzewa rozpinającego grafu

$T \leftarrow \emptyset$
$VS \leftarrow \emptyset$
for each $v \in V$ **do** wstaw zbiór $\{v\}$ do VS
while $|VS| > 1$ **do**
    wybierz $\langle u, w \rangle$ z E o najmniejszym koszcie
    usuń $\langle u, w \rangle$ z E
    $A \leftarrow FIND(u)$; $B \leftarrow FIND(w)$
    if $A \neq B$ **then** $UNION(A, B, X)$
                wstaw $\langle u, w \rangle$ do T

42 Rozwiązania

42.1 Proste rozwiązanie

Do reprezentowania rodziny zbiorów używamy tablicy $R[1..n]$ takiej, że

$$\forall_i \quad R[i] \text{ jest nazwą zbioru zawierającego } i.$$

Koszt: FIND - $\Theta(1)$; UNION -$\Theta(n^2)$.

42.2 Modyfikacja prostego rozwiązania

42.2.1 Idea

Oparta na dwóch trickach:

- Wprowadzamy nazwy wewnętrzne zbiorów (niewidoczne dla użytkownika).
- Podczas wykonywania UNION(A, B, C) zbiór mniejszy przyłączany jest do większego.

42.2.2 Realizacja

Używamy tablic: $R, ExtName, IntName, List, Next$ i $Size$ takich, że:

$R[i]$	=	nazwa wewnętrzna zbioru zawierającego i,
$ExtName[j]$	=	nazwa zewnętrzna zbioru o nazwie wewnętrznej j,
$IntName[k]$	=	nazwa wewnętrzna zbioru o nazwie zewnętrznej j,
$List[j]$	=	wska"xnik na pierwszy element w liście elementów zbioru o nazwie wewnętrznej j,
$Next[i]$	=	następny po i element w liście elementów zbioru $R[i]$,
$Size[j]$	=	liczba elementów w zbiorze o nazwie wewnętrznej j.

```
procedure Find(i)
    return (ExtName(R[i]))


procedure UNION(I, J, K)
    A ← IntName[I]
    B ← IntName[J]
    Niech Size[A] ≤ Size[B]; w p.p. zamień A i B rolami
    el ← List[A]
    while el ≠ 0 do  R[el] ← B
                     last ← el
                     el ← Next[el]
    Next[last] ← List[B]
    List[B] ← List[A]
    Size[B] ← Size[A] + Size[B]
    IntName[K] ← B
    ExtName[B] ← K
```

Twierdzenie 13 *Używając powyższego algorytmu można wykonać dowolny ciąg σ o długości $O(n)$ w czasie $O(n \log n)$.*

43 Struktury drzewiaste dla problemu Union-Find

43.1 Elementy składowe struktury danych

- Las drzew.
  Każdy podzbiór reprezentowany jest przez drzewo z wyróżnionym korzeniem. Wierzchołki wewnętrzne zawierają wska"xnik na ojca (nie ma wska"xników na dzieci!).

- Tablica $Element[1..n]$:

$$Element[i] = \text{ wska"xnik na wierzchołek zawierający } i.$$

- Tablica *Root*:

$$Root[I] = \text{ wska"xnik na korzeń drzewa odpowiadającego zbiorowi } I$$

  (nazwy zbiorów są dla nas nieistotne; będą one liczbami z $[1, .., n]$).

43.2 Realizacja instrukcji

$Union(A, B, C)$ polega na połączeniu drzew odpowiadających zbiorom A i B w jedno drzewo i umieszczeniu w jego korzeniu nazwy C.

$Find(i)$ polega na przejściu ścieżki od wierzchołka wskazywanego przez $Element(i)$ do korzenia drzewa i odczytaniu pamiętanej tam nazwy drzewa. Przy wykonywaniu tych instrukcji stosujemy następującą strategię:

1. instrukcję $Union$ wykonujemy w sposób zbalansowany - korzeń mniejszego (w sensie liczby wierzchołków) drzewa podwieszamy do korzenia drzewa większego (a dokładniej drzewa nie większego do korzenia drzewa nie mniejszego),

2. podczas instrukcji $Find(i)$ wykonujemy *kompresję ścieżki* prowadzącej od i do korzenia - wszystkie wierzchołki leżące na tej ścieżce podwieszamy bezpośrednio pod korzeń.

43.3 Implementacja

Każdy wierzchołek v zawiera pola:

- $Father[v]$ - wska"xnik na ojca (równy NIL, gdy v jest korzeniem),
- $Size[v]$ - liczba wierzchołków w drzewie o korzeniu v,
- $Name[v]$ - nazwa drzewa o korzeniu v

Zawartość pól $Size[v]$ i $Name[v]$ ma znaczenie tylko wówczas, gdy v jest korzeniem.

```
procedure InitForest
    for i ← 1 to n do v ← Allocate − Node()
                       Size[v] ← 1
                       Name[v] ← i
                       Father[v] ← NIL
                       Element[i] ← v
                       Root[i] ← v
```

```
procedure Union(i, j, k)
   Niech Size[Root[i]] ≤ Size[Root[j]]; w p.p. zamień i oraz j rolami
   large ← Root[j]
   small ← Root[i]
   Father[small] ← large
   Size[large] ← Size[large] + Size[small]
   Name[large] ← k
   Root[k] ← large
```

```
procedure Find(i)
    list ← NIL
    v ← Element[i]
    while Father[v] ≠ NIL do wstaw v na list
                          v ← Father[v]
    for each w na list do Father[w] ← v
    return Name[v]
```

43.4 Analiza algorytmu

Lemat 3 *Jeśli instrukcje Union wykonujemy w sposób zbalansowany, to każde powstające drzewo o wysokości h ma co najmniej 2^h wierzchołków.*

Definicja 18 *Niech $\tilde{\sigma}$ będzie ciągiem instrukcji Union powstałym po usunięciu wszystkich instrukcji Find z ciągu σ.* Rzędem *wierzchołka v względem σ nazywamy jego wysokość w lesie powstałym po wykonaniu ciągu $\tilde{\sigma}$.*

Lemat 4 *Jest co najwyżej $\frac{n}{2^r}$ wierzchołków rzędu r.*

Wniosek 3 *Każdy wierzchołek ma rząd co najwyżej r.*

Lemat 5 *Jeśli w trakcie wykonywania ciągu σ wierzchołek w staje się potomkiem wierzchołka v, to rząd w jest mniejszy niż rząd v.*

Definicja 19

$$\log^*(n) \stackrel{df}{=} \min\{k \mid F(k) \geq n\},$$

gdzie $F(0) = 1$ i $F(i) = 2^{F(i-1)}$ dla $i > 0$.

Twierdzenie 14 *Niech c będzie dowolną stałą. Wówczas istnieje inna stała c' (zależna od c) taka, że powyższe procedury wykonują dowolny ciąg σ złożony z cn instrukcji Union i Find w czasie $c'n \log^* n$.*

Twierdzenie 15 *Algorytm realizujący ciągi instrukcji Union i Find przy użyciu powyższych procedur ma złożoność większą niż cn dla dowolnej stałej c.*

UWAGA: na ćwiczeniach pokażemy, że przy pomocy struktur drzewiastych można w czasie $O(n \log^* n)$ realizować ciągi σ, które oprócz instrukcji *Union* i *Find* zawierają także instrukcje *Insert* i *Delete*.

ANALYSIS OF THE UNION-FIND ALGORITHM

In this handout we are going to analyze the worst case time complexity of the UNION-FIND algorithm that uses a forest of ''up trees'' (*i.e.* trees where each node has only a pointer to its parent) with weight (or size) balancing for UNION and path compression for FIND. (Note: The book uses UNION by rank, which is another valid approach.) The algorithms are:

procedure *Make-Set* (x)
1. $size[x] \leftarrow 1$
2. $parent[x] \leftarrow x$

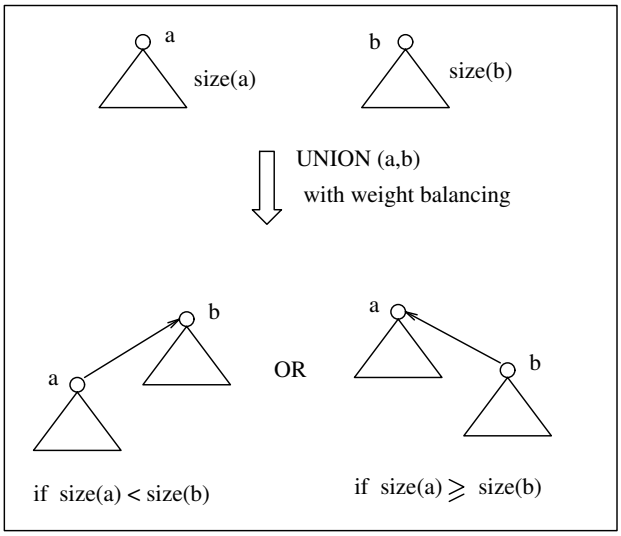
end.

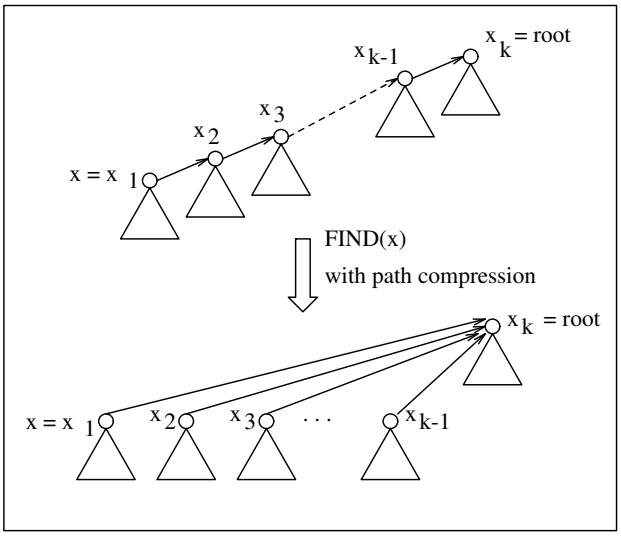
procedure *UNION*(a , b) { with weight balancing }
{a and b are roots of two distinct trees in the forest.}
{Makes the root of the smaller tree a child of the root of the larger tree.}
1. **if** $size[a] < size[b]$ **then** $a \longleftrightarrow b$
2. $parent[b] \leftarrow a$
3. $size[a] \leftarrow size[a] + size[b]$

end.

function *FIND*(x) { with path compression }
{Returns the root of the tree that contains node x.}
1. **if** $parent[x] \neq x$ **then**
2. $parent[x] \leftarrow FIND(parent[x])$
3. **return** $parent[x]$

end.





Throughout the handout, n is the number of different elements in all the sets (*i.e.* the total number of nodes in all the trees in the forest that represents the sets. In other words, the total number

of *Make-Set* operations), and m denotes the total number of *Make-Set*, *UNION* and *FIND* operations in a sequence of such operations. Note that there can be at most $n-1$ UNION operations, since each such operation reduces the number of trees in the forest by one.

Lemma 1: Assume, starting with the initial forest, we perform a number of UNION operations. If we use weight balancing when merging trees, any node in the forest with height h will have $\geq 2^h$ descendents.

Proof: Induction on the number of UNION operations.
Basis: (No UNION operations are performed.) Then each tree has $1=2^0$ nodes, as wanted.
Induction Step: Assume the induction hypothesis holds so far, and the next UNION operation makes the root of a tree T_1 a child of the root of another tree T_2. Let us call the resulting tree T. The height and number of descendents of all nodes in T remains the same as before, except for the root of T. Let us assume T_i, for $i = 1, 2$, has size (i.e., number of nodes) s_i, and height h_i. By the induction hypothesis we must have (i) $s_i \geq 2^{h_i}$, for $i = 1, 2$. And because of the weight balancing we must have (ii) $s_2 \geq s_1$. The root of T has $s = s_1 + s_2$ descendents and has height $h = \max (h_1 + 1 , h_2)$. Then from (i) and (ii) we conclude:

$$s = s_1 + s_2 \geq 2s_1 \geq 2^{1+h_1} \quad \& \quad s = s_1 + s_2 \geq s_2 \geq 2^{h_2}$$

Therefore, $s \geq \max (2^{1+h_1} , 2^{h_2}) = 2^h$. This completes the inductive proof. □

Corollary 0: Assume, starting with the initial forest, we perform an arbitrary number of UNION and FIND operations. If we use weight balancing when merging trees, any tree in the forest with height h will have $\geq 2^h$ nodes.

Proof: The claim follows from Lemma 1 by observing that a FIND operation does not change the number of nodes in a tree and it can not increase the height of a tree (it may decrease it). □

Corollary 1: In a forest created by using the weight balancing rule, any tree with n nodes has height $\leq \lceil lg\ n \rceil$. □

Corollary 2: The UNION-FIND algorithm using *only* weight balancing (but no path compression) takes $O(n + m\ lg\ n)$ time in the worst case for an arbitrary sequence of m UNION-FIND operations.

Proof: Each UNION operation takes $O(1)$ time. Each FIND operation can take at most $O(lg\ n)$ time. □

In the rest of this handout we analyze the UNION-FIND algorithm that uses both weight balancing (for UNION) and path compression (for FIND). To help us in our analysis, let us define the ''super-exponential'' and ''super-logarithmic'' functions.

The ''super-exponential'', $\exp^*(n)$, is defined recursively as follows: $\exp^*(0) = 1$, and for $i > 0$ $\exp^*(i) = 2^{\exp^*(i-1)}$; thus $\exp^*(n)$ is a stack of n 2's. (Let us also define the boundary case $\exp^*(-1) = -1$.)

The ''super-logarithm'', $lg^*(n) = \min i$ such that $\exp^*(i) \geq n$.

Remark: Note that $\exp^*$ grows *very* rapidly, whereas lg^* grows *very* slowly: $\exp^*(5) = 2^{65536}$, while $\log^*(2^{65536}) = 5$. We have $2^{65536} \gg 10^{120}$, where the latter quantity is the estimated number

of atoms in the universe. Thus $lg^*(n) \leq 5$ for all "practical" n. However, eventually $lg^*(n)$ goes to infinity as n does, but at an almost unimaginably slow rate of growth.

Fact 1: Assume $r \geq 0$ and $g \geq 0$ are integers. Then, $lg^*(r) = g$ if and only if $\exp^*(g-1) < r \leq \exp^*(g)$. □

Let s be a sequence of UNION-FIND operations.

Definition: The rank of a node x (in the sequence s), $rank(x)$, is defined as follows:

a. Let s' be the sequence of operations resulting when we remove all FIND operations from s.

b. Execute s', using weight balancing (since there are no FIND's there will be no path compression).

c. The rank of x (in s) is the height of node x in the forest resulting from the execution of s'.

Put another way. Perform sequence s in two different ways. One with path compression when doing FIND operations, one without path compression. (The UNION operations in both are done with weight balancing.) Call the resulting UNION-FIND forests, the *compressed forest* and the *uncompressed forest*, respectively. Then $rank(x)$ is the height of node x in the *final uncompressed forest*.

Lemma 2: For any sequence s, there are at most $n/2^r$ nodes of rank r.

Proof: Let s' be the sequence that results from s if we delete all the FIND operations. Consider the forest produced when we execute s'. By Lemma 1, each node of rank r has $\geq 2^r$ descendents. In a forest two nodes of the same height have distinct descendents. In particular, distinct nodes of rank r must have disjoint sets of descendents. Since there are n nodes in total, there are at most $n/2^r$ disjoint sets each of size $\geq 2^r$. Hence there are at most $n/2^r$ nodes of rank r. □

Lemma 3: If during the execution of sequence s, node x is ever a proper descendent of node y, then $rank(x) < rank(y)$ in s.

Proof: Simply observe that if path compression in the execution of s causes x to become a proper descendent of y, surely x will be a proper descendent of y at the forest resulting in the end of executing s' (that does *not* involve any path compression). Thus the height of x in that forest is less than the height of y and therefore $rank(x) < rank(y)$ in s, as wanted. □

We want to calculate an upper bound on the worst case time complexity to process a sequence s of m operations on a forest of size n. First of all, observe that each *Make-Set* and *UNION* takes only $O(1)$ time and we can have n *Make-Set* and at most $n-1$ *UNION* operations. So these operations will contribute at most $O(n)$ time. Let's now consider the complexity of at most m *FIND* operations.

It is useful to think of nodes as being in "groups" according to their rank. In particular we define the group number of node x, $group(x) = lg^*(rank(x))$.

The time for a FIND(x) operation, where x is a node, is proportional to the number of nodes in the path from x to the root of its tree. Suppose these nodes are $x_1 = x$, $x_2, \cdots, x_k = root$, where $x_{i+1} = parent(x_i)$, for $1 \leq i < k$. We will apportion the "cost" (i.e., time) for FIND(x) to the operation itself and to the nodes $x_1, x_2, \cdots, x_k$ according to the following rule:

For each $1 \leq i \leq k$:

(i) If $x_i = root$ (*i.e.* $i = k$) or if $group(x_i) \neq group(x_{i+1})$, then charge 1 unit (of time) to the operation FIND(x) itself.

(ii) If $group(x_i) = group(x_{i+1})$, then charge 1 unit (of time) to node x_i.

The time complexity of processing the FIND operations can then be obtained by summing the cost units apportioned to each operation and the cost units apportioned to each node.

From Lemma 2, the maximum rank of any node is $\lceil lg\, n \rceil$. Therefore the number of different groups is at most $lg^*(\lceil lg\, n \rceil)$. This is, then, the maximum number of units apportioned to any FIND operation. Therefore, for the total of at most m such operations the number of units charged to the FIND operations is at most $O(m\, lg^*(lg\ n)) = O(m\ lg^*\ n)$ (1).

Next consider the cost units apportioned to the nodes. Each time path compression causes a node x to ''move up'', *i.e.* to acquire a new parent, the new parent of x has, by Lemma 3, higher rank than its previous parent (the previous parent was a proper descendent of the new parent before the path compression). This means that x will be charged by rule (ii) at most as many times as there are distinct ranks in $group(x)$. After that, x *must* become the child of a node in a different group and thenceforth any further ''move ups'' of x will be accounted for by rule (i). (Note that, again by Lemma 3, once x has acquired a parent in a different group than it, all subsequent parents of x will also be in a different group than x, since they have progressively higher ranks.)

Let $g = group(x)$. By Fact 1, The number of different ranks in group g is $\exp^*(g) - \exp^*(g-1)$ (this is the number of (integer) ranks r such that $lg^*(r) = g$). Then, by the above discussion, the maximum number of units charged to any node in group g is $\exp^*(g) - \exp^*(g-1)$. Now let's calculate the number of nodes in group g, $N(g)$. By Lemma 2 we have:

$$N(g) \le \sum_{r=\exp^*(g-1)+1}^{\exp^*(g)} \frac{n}{2^r} \le \frac{n}{2^{\exp^*(g-1)+1}} [1 + \frac{1}{2} + \frac{1}{4} + \cdots]$$

$$= \frac{n}{2^{\exp^*(g-1)}} = \frac{n}{\exp^*(g)}$$

Thus, the total number of units charged to nodes of group g is at most

$$N(g) \cdot (\exp^*(g) - \exp^*(g-1)) \le \frac{n}{\exp^*(g)} \cdot (\exp^*(g) - \exp^*(g-1)) = O(n).$$

We have already seen that the number of different groups is $lg^*(\lceil lg\, n \rceil)$ and therefore the total number of cost units charged to all nodes by rule (ii) is $O(n \cdot lg^*(\lceil lg\, n \rceil)) = O(n\, lg^*(lg\, n)) = O(n\ lg^*\ n)$ (2).

By summing (1) and (2) we get that the worst case time complexity to process at most m FIND operations is $O((m+n)\, lg^*\, n)$. The latter is $O(m\ lg^*\ n)$ since $m \ge n$. Since, as we already pointed out, $O(n)$ UNIONs take only $O(n)$ time, we conclude:

Theorem 1: The total worst-case time complexity to process an arbitrary sequence of m *Make-Set*, *UNION* and *FIND* operations, n of which are *Make-Set*'s, using weight balancing and path compression is $O(\ m\ lg^*\ n\ )$. □

Bibliography

[CLRS] Chapter 21.

[Weiss] Chapter 8.

[Tar83] R.E. Tarjan, *"Data Structures and Network Algorithms,"* CBMS-NSF, SIAM Monograph, 1983, (Chapter 2).

[Tar79] R.E. Tarjan, *"Applications of path compression on balanced trees,"* Journal of ACM, Vol. 26, No. 4, Oct. 1979, pp. 690-715.