并查集

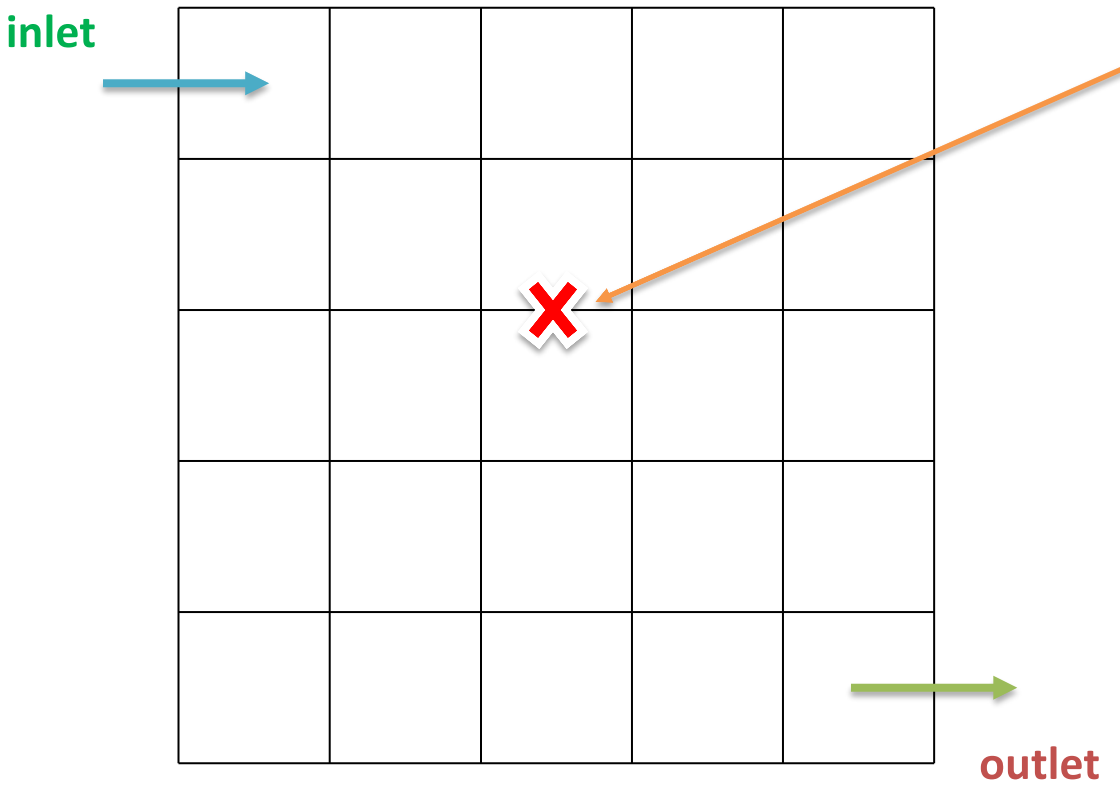
南京大学计算机系

赵建华

Union-Find

- Dynamic Equivalence Relation
  - Examples
  - Definitions
  - Brute force implementations
- Disjoint Set
  - Straightforward Union-Find
  - Weighted Union + Straightforward Find
  - Weighted Union + Path-compressing Find

Maze Generation



Select a wall to pull down randomly

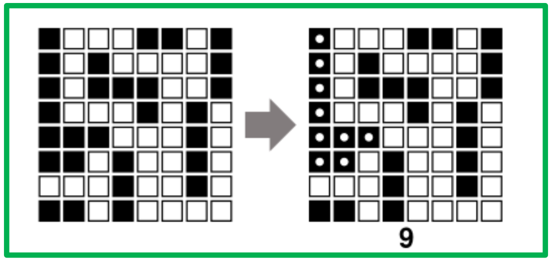
If *i* and *j* are in same *equivalence class*, then select another wall to pull down.

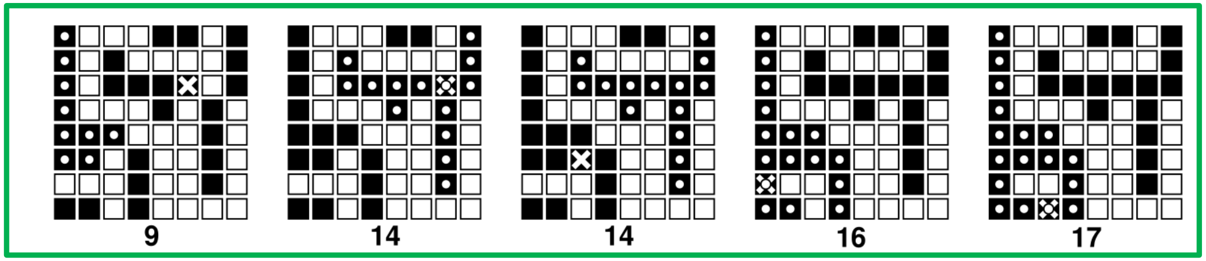
Otherwise, joint the two classes into one.

The maze is complete when the inlet and outlet are in one equivalence class.

Black Pixels

- Maximum black pixel component
  - Let α be the size of the component
- Color one pixel black
  - How α changes?
  - How to choose the pixel, to accelerate the change in α





In Minimum Spanning Tree

- Kruskal算法生成图的最小生成树
- Kruskal's algorithm, greedy strategy:
  - Select one edge
    - With the minimum weight
    - Not in the tree
  - Evaluate this edge
    - This edge will **NOT** result in a cycle
- Critical issue:
  - How to know "NO CYCLE"?
  - The two nodes of this tree should not be connected by the edges selected previously.

结点之间的连同关系实际上就是一个动态的等价关系

Dynamic Equivalence Relations

- Equivalence
  - Reflexive, symmetric, transitive
  - Equivalent classes forming a partition
    - S的一个分划：S的一组互不相交的子集，且子集的并集就是S
- Dynamic equivalence relation
  - Changing in the process of computation
  - **IS** instruction: *yes* or *no* (in the same equivalence class)
  - **MAKE** instruction: combining two equivalent classes, by relating two unrelated elements, and influencing the results of subsequent IS instructions.
  - Starting as equality relation

Implementation: How to Measure

- The number of basic operations for processing a sequence of m **MAKE** and/or **IS** instructions on a set S with n elements.
- An Example: $S=\{1,2,3,4,5\}$
  - 0. [create] {{1}, {2}, {3}, {4}, {5}}
  - 1. **IS** 2≡4? No
  - 2. **IS** 3≡5? No
  - 3. **MAKE** 3≡5. {{1}, {2}, {3,5}, {4}}
  - 4. **MAKE** 2≡5. {{1}, {2,3,5}, {4}}
  - 5. **IS** 2≡3? Yes
  - 6. **MAKE** 4≡1. {{1,4}, {2,3,5}}
  - 7. **IS** 2≡4? No

Union-Find based Implementation

- The maze problem
  - 初始状态： Each cell as a set
  - Randomly delete a wall and union two cells
  - Loop until you find the inlet and outlet are in one equivalent class
- The Kruskal algorithm
  - 初始状态： Each node as a set
  - Choose the least weight edge (u,v)
  - Find whether u and v are in the same equivalent class
  - If not, add the edge and union the two nodes

Implementation: Choices

- Matrix (relation matrix)
  - Space in $\Theta(n^2)$, and worst-case cost in $\Omega(mn)$ (mainly for row copying for MAKE)
- Array (for equivalence class ID)
  - Space in $\Theta(n)$, and worst-case cost in $\Omega(mn)$ (mainly for search and change for MAKE)
- Disjoint Set
  - A collection of disjoint sets, supporting *Union* and *Find* operations
  - Not necessary to traverse all the elements in one set

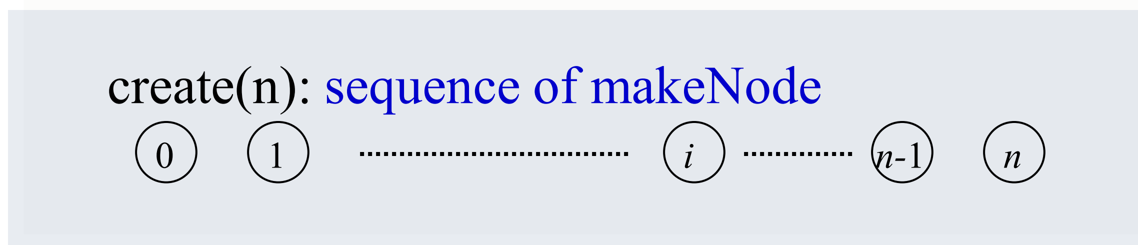
Union-Find ADT

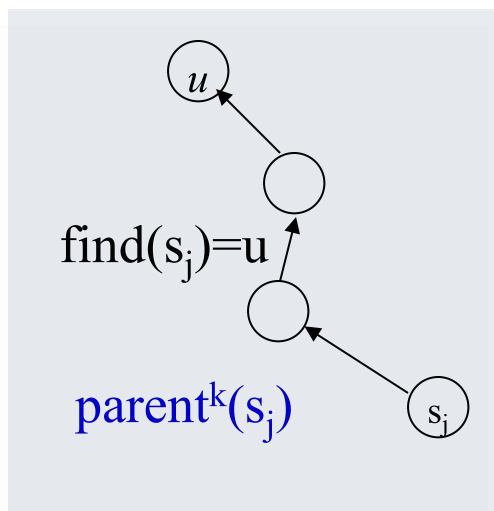
- Constructor: Union-Find create(int n)
  - sets=create(*n*) refers to a newly created group of sets {1}, {2}, ..., {*n*} (*n* singletons)
- Access Function: **int** find(UnionFind sets, *e*)
  - find(sets, *e*)=<*e*>
- Manipulation Procedures
  - **void** makeSet(UnionFind sets, **int** *e*)
  - **void** union(UnionFind sets, **int** *s*, **int** *t*)

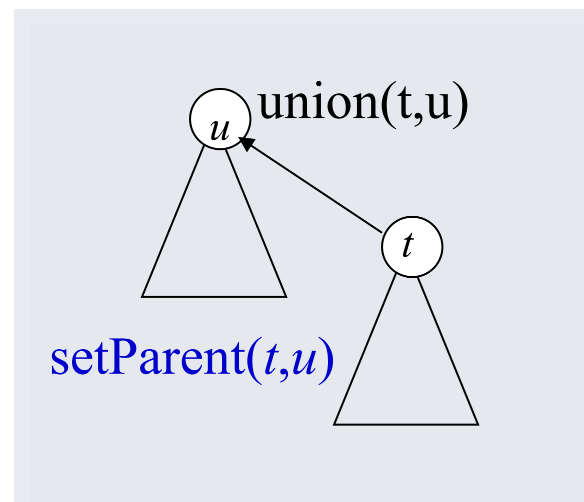
Using Union-Find (as inTree)

- **IS** $s_i \equiv s_j$:
  - t=find(s_i);
  - u=find(s_j);
  - (t==u)?
- **MAKE** $s_i \equiv s_j$:
  - t=find(s_i);
  - u=find(s_j);
  - union(t,u);

implementation by inTree



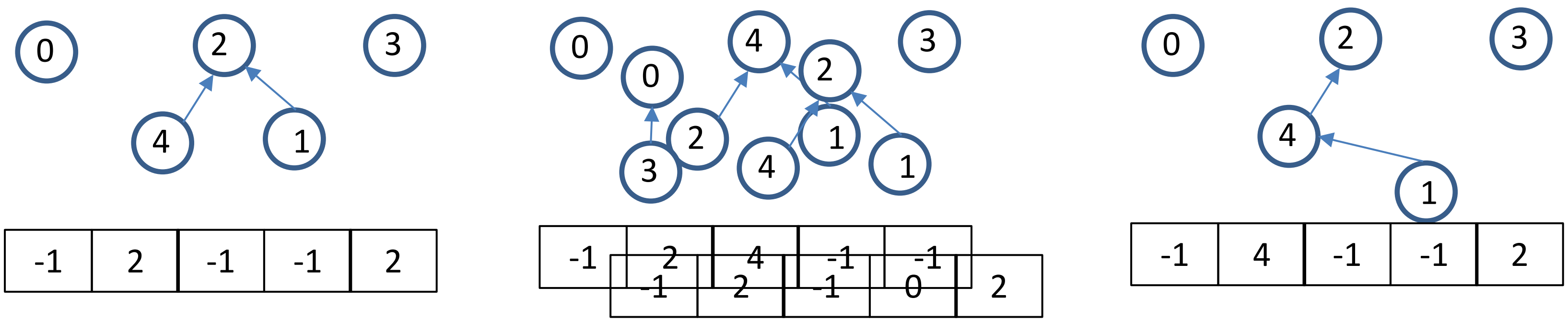




Union-Find的数据结构

- 逻辑上
  - 各个元素编号为0, 1, 2, •••, n−1
  - 使用一棵树表示一个子集，子集中的元素相互等价。
  - 整个集合划分成为互不相交的子集（树）
- 实现上
  - 使用数组记住每个元素（结点）的父亲结点(的编号)
  - 树的根节点的父亲结点为−1.

- $\{\{0\}, \{1,2,4\}, \{3\}\}$的三种可能的表示



Union(0,3)之后得到什么样的数据？

并查集的数组实现（Slow）

- 用长度为n的数组s表示n个元素
  - S[i]表示第i个元素的父节点
  - S[i] == -1表明i是某棵树的根节点

```
public DisjSetsSlow( int numElements )
{
      s = new int [numElements ];
      for( int i = 0; i < s.length; i++ )
            s[ i ] = -1;
}

public int find( int x )
{
     if( s[ x ] < 0 )
          return x;
     else
          return find( s[ x ] );
}
```

```
public void union( int x, int  y)
{
     if(root1 != root2)
           s[ root1] = root2;
}
```

Union-Find Program

- A union-find program of length *m*
  - is (a *create*(*n*) operation followed by) a sequence of *m* union and/or find operations in any order
- A union-find program is considered an input
  - The object on which the analysis is conducted
- The measure: number of accesses to the ***parent***
  - assignments: for union operations
  - lookups: for find operations

  } **link operation**

- Union-Find Program用于union-find数据结构访问/操作的效率分析：
  - 如果一个算法A使用了union-find作为基础数据结构，那么A的一次运行过程中对这个数据结构的操作序列就是一个Union-Find Program.
  - 我们可以整体地分析这个Union-Find Program所需要的时间，也就是算法A花费在union-find上的总时间。

Union-find Program的例子

总共n个元素

1. Union(1,2)
2. Union(2,3)

⋮

n-1. Union(n-1,n)

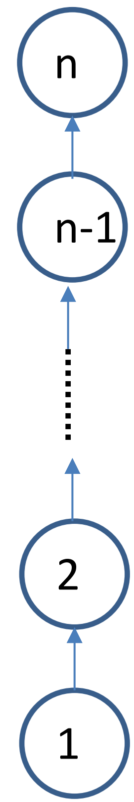
n. Find(1)

⋮

m. Find(1)

Example

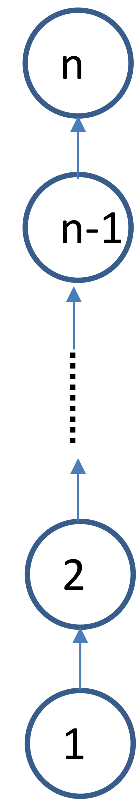
```
public int find( int x )
{
    if( s[ x ] < 0 )
        return x;
    else
        return find( s[ x ] );
}

public void union( int x, int  y)
{
    if(root1 != root2)
        s[ root1] = root2;
}
```



2	3	… … … …	n	-1

Worst-case Analysis for Union-Find Program

- Assuming each lookup/assignment take O(1).
- Each makeSet or union does one assignment, and each find does d+1 lookups, where d is the depth of the node.



1. Union(1
2. Union(2

⋮

n-1. Union(n-1,n)

n. Find(1)

⋮

m. Find(1)

Example

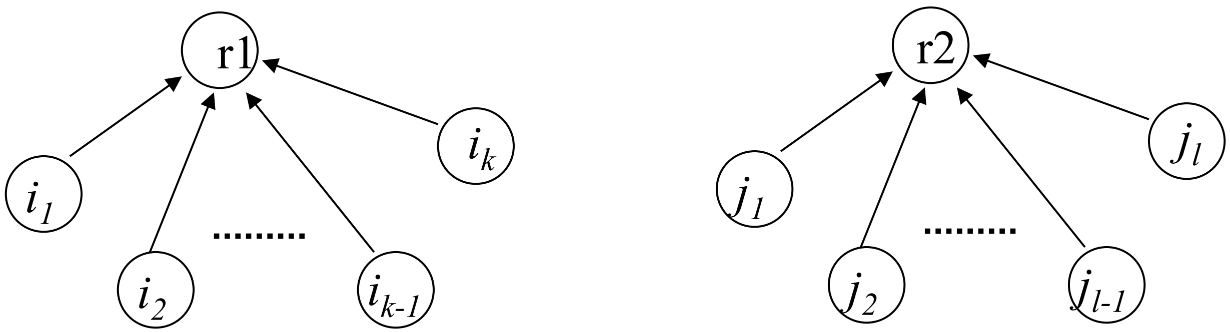
显然，union构建树的过程不够聪明

operations done:

$n+(n-1)+(m-n+1)n$ $\Theta(mn)$

Find(1) needs *n* array lookups

Union的改进



- Union(r1,r2)实际上既可以把r2作为合并后的树的根，也可以把r1作为合并后的树的根
- 如何选择新的根，使得寻找的效率比较高？

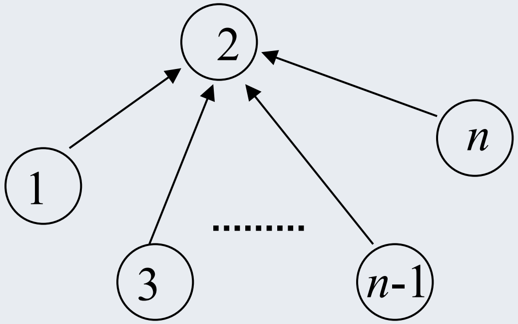
Weighted Union: for Shorter Trees

- Weighted union (*wUnion*)
  - always have the tree with **fewer nodes** as subtree

To keep the *Union* valid, each *Union* operation is replaced by:

t=find(i);
u=find(j);
union(t,u)

The order of (t,u) satisfying the requirement



Tree made by wUnion

Cost for the program example:
$n+3(n-1)+2(m-n+1)$

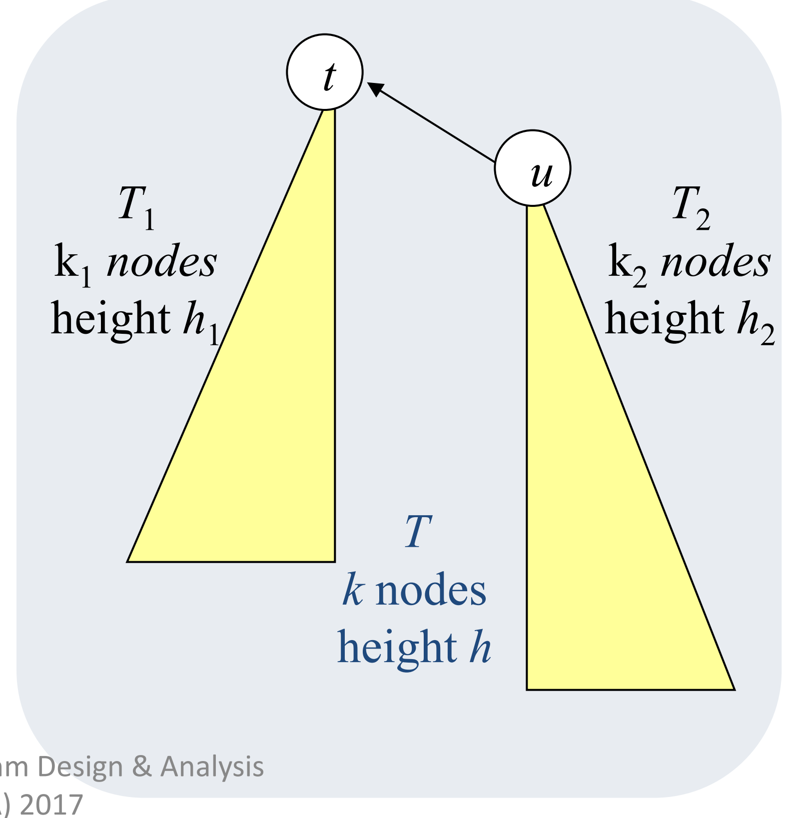
Weighted Union的实现

```
public DisjSetsSlow( int numElements )
{
        s = new int [numElements ];
        weights = new int[numElements];
        for( int i = 0; i < s.length; i++ )
        {
                s[ i ] = -1;    weights[i] = 1;
        }
}

public int find( int x )
{
        if( s[ x ] < 0 )
                return x;
        else
                return find( s[ x ] );
}
```

```
//注意这里仍然假设root1和root2是树的根
public void wUnion( int root1, int  root2)
{
        if(root1 == root2)
                return;
        if(weights[root2]  > weights[root1])
        { int tmp = root1; root1 = root2; root2 = tmp;}

        s[ root2] = root1;
        weights[root1] = weights[root1] + weights[root2];
}
```

Upper Bound of Tree Height

- After any sequence of *Union* instructions, implemented by *wUnion*, any tree that has k nodes will have height at most $\lfloor \log k \rfloor$
- Proof by induction on k:
  - base case: k=1, the height is 0.
  - by inductive hypothesis:
    - $h_1 \leq \lfloor \lg k_1 \rfloor$, $h_2 \leq \lfloor \lg k_2 \rfloor$
  - h=max(h1, h2+1), k=k1+k2
    - if $h=h_1$, $h \leq \lfloor \lg k_1 \rfloor \leq \lfloor \lg k \rfloor$
    - if $h=h_2+1$, note: $k_2 \leq k/2$
      so, $h_2+1 \leq \lfloor \lg k_2 \rfloor + 1 \leq \lfloor \lg k \rfloor$



Upper Bound for Union-Find Program (With Weighted Union)

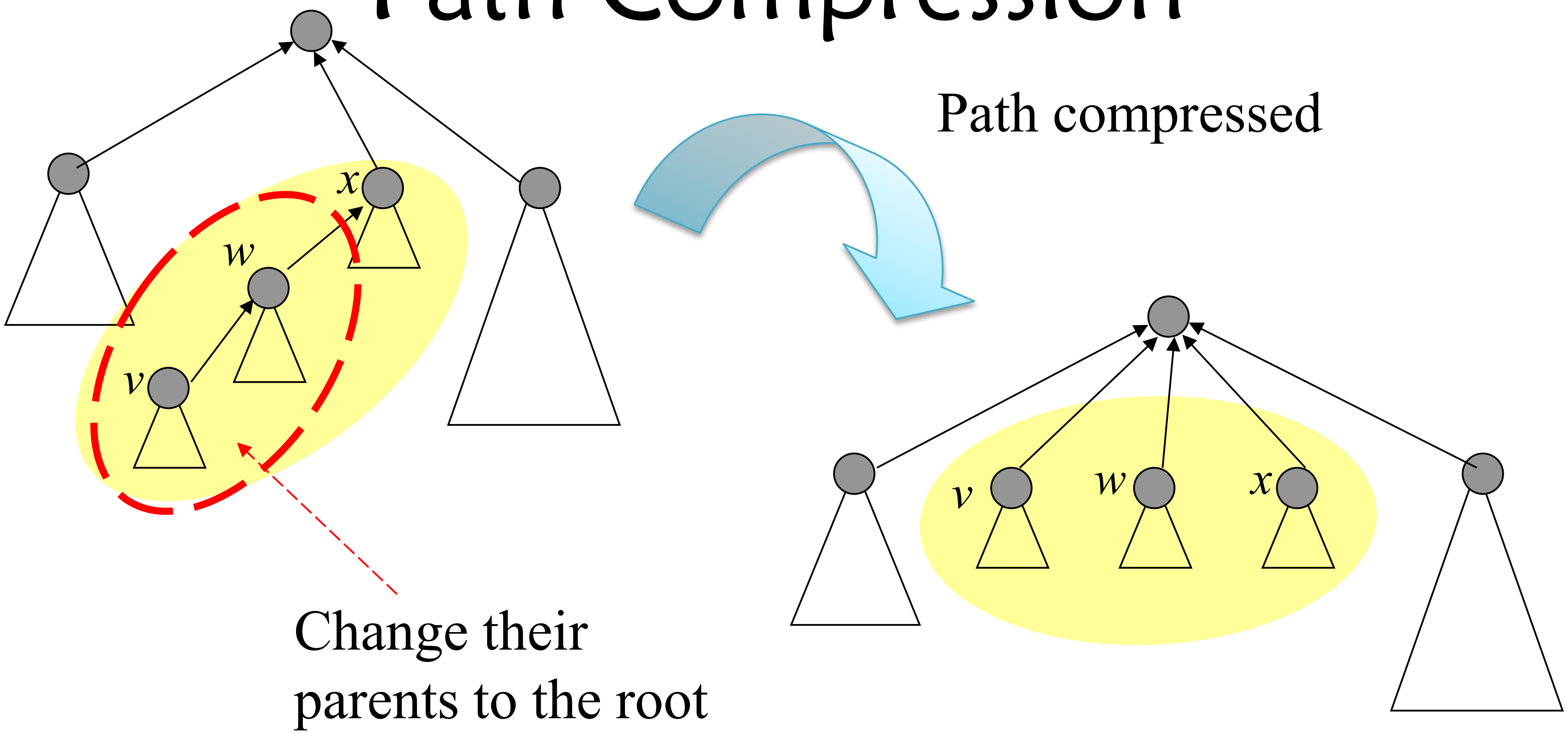
- A Union-Find program of size m, on a set of n elements, performs **$O(n+m\log n)$** link operations in the worst case if *wUnion* and straight *find* are used
- Proof:
  - At most n-1 *wUnion* can be done, building a tree with height at most $\lfloor \log n \rfloor$,
  - Then, each *find* costs at most $\lfloor \log n \rfloor+1$.
  - Each *wUnion* costs in $O(1)$, so, the upper bound on the cost of any combination of m *wUnion/find* operations is the cost of m *find* operations, that is $m(\lfloor \log n \rfloor+1) \in O(n+m\log n)$

There do exist programs requiring $\Omega(n+(m-n)\log n)$ steps.

Path Compression

- 并查集中的树是用来发现各个结点所在的根节点的，只要根节点不变，这棵树的形状不影响find的结果。
- 树的形状越矮越好
- 在find(x)的过程中会遍历从x到达x所在树的根节点的路径上的全部结点
  - 这些结点和x在同一个子集（根节点相同）
  - 只需要一次操作就可以将这些结点直接链接到根节点
  - 虽然本次find多花了时间，但是后面的find可以省下很多时间

Path Compression



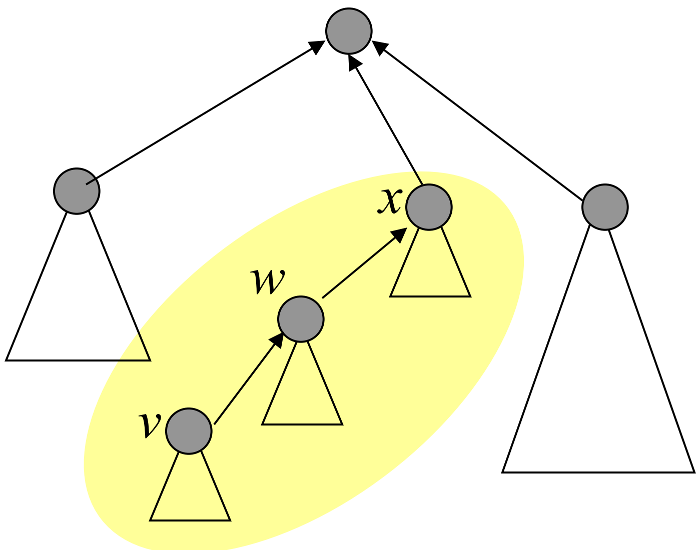
付出：
1、find(v)的过程增加了一倍的操作
收益：
1、之后再调用find(v),find(w),find(x)时，只需要2次漫游
2、对于v,w,x的子节点，find所需要的次数也减少了。

Find的路径压缩实现

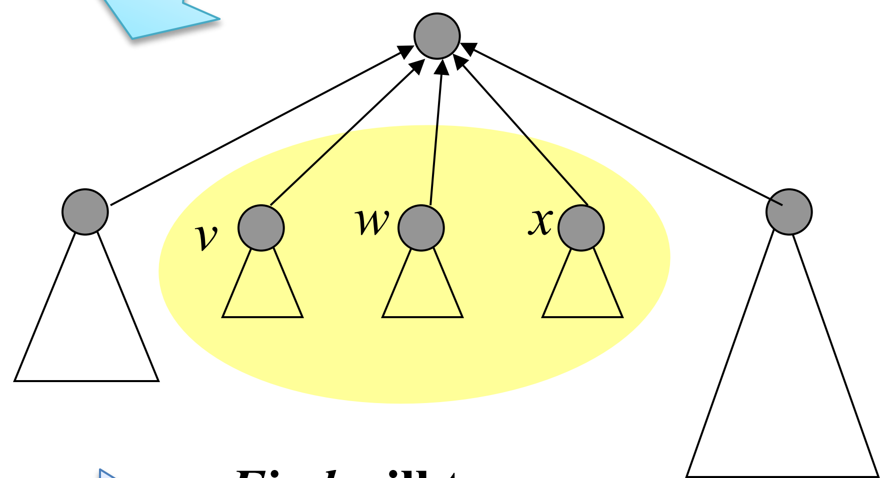
```
//递归实现
public int cFind( int x )
{
    if( s[ x ] < 0 )
        return x;
    else
    {
        int root = find( s[ x ] );
        s[x] = root;
        return root;
    }
}
```

```
//迭代实现
public int cFind( int x )
{
    int cur = x;
    while(s[cur] >= 0)
        cur = s[cur];
    root = cur;

    cur = x;
    while(s[cur] >= 0)
    {
        int tmp = s[cur];
        s[cur] = root;
        cur = tmp;
    }

     return root
}
```

Challenges for the Analysis



Path compressed



***cFind* does twice as many link operations as the *find* does for a given node in a given tree.**

but…

***cFind* will traverse shorter paths**

Analysis: the Basic Idea

- *c*Find may be an expensive operation
  - in the case that find(*i*) is executed and the node *i* has great depth.
- However, such *c*Find can be executed only for limited times
  - Union(r1,r2)使得r1或者r2的所有子结点离根节点的路径增加1，导致find这些子节点的时候可能需要增加一次compression操作.
  - Path compressions depends on previous unions
- So, *amortized analysis(均摊分析)* applies
  - 均摊分析通常用于m个操作之间相互促进，使得总体复杂度远低于m乘以单个操作最坏条件的情况

Co-Strength of *wUnion* and *cFind*

- **$O((n+m)\log^*(n))$**
  - Link operations for a *Union-Find* program of length m on a set of n elements is in the worst case.
  - Implemented with *wUnion* and *cFind*

What's $\log^*(n)$?

- Define the function H as following:

$$\begin{cases} H(0) = 1 \\ H(i) = 2^{H(i-1)} \quad for \ \ i > 0 \end{cases}$$

- Then, $\log^*(j)$ for $j \geq 1$ is defined as:

  $\log^*(j) = \min\{ k \mid H(k) \geq j \}$

A Function Growing Extremely Slowly

- **Function H:**

$$\begin{cases} H(0)=1 \\ H(i+1)=2^{H(i)} \end{cases}$$

that is: $H(k)=\underbrace{2^{2^{\cdot^{\cdot^{2}}}}}_{k\ 2\text{'s}}$

Note:

H grows extremely fast:

$H(4)=2^{16}=65536$

$H(5)=2^{65536}$

- **Function Log-star**

log*(j) is defined as the least i such that:

$H(i)\geq j$ for $j>0$

- **Log-star grows extremely slowly**

$$\lim_{n\to\infty}\frac{\log^*(n)}{\log^{(p)} n}=0$$

***p* is any fixed nonnegative constant**

For any x: $2^{16}\leq x\leq 2^{65536}-1$, log*($x$)=5

Union-Find Program执行过程的性质

- wUnion会合并两棵树，并设定新的根节点，导致某一棵树的结点离根节点的距离加1
  - 但是wUnion不会改变子树中的结构
- cFind操作会改变树的结构，但是不会改变树的根节点。
  - 如果cFind(x)执行时x的根节点为y，那么它会将x到y的结点进行压缩，路径上的结点都会直接指向y
- 引入一个虚拟的函数ExploreAndCompress(x,y);
  - 要求y是x的祖先结点；从x遍历到y，并进行路径压缩
  - 当y是x所在树的根节点的时候，ExploreAndCompress(x,y)和cFind(x)执行相同的操作
  - 只要以y为根的子树的结构不变， ExploreAndCompress(x,y)执行的操作也不变。

```
…
cFind(x)      //返回值为y,
              //等价于ExploreAndCompress(x,y)
wUnion(…)  //将两棵树合并，不改变y
              //所在树的结构
…
```

```
…
wUnion(…)
ExploreAndCompress(x,y)
…
```

这两个程序所需要的操作（即代价），以及对树的结构的改变是等价的。

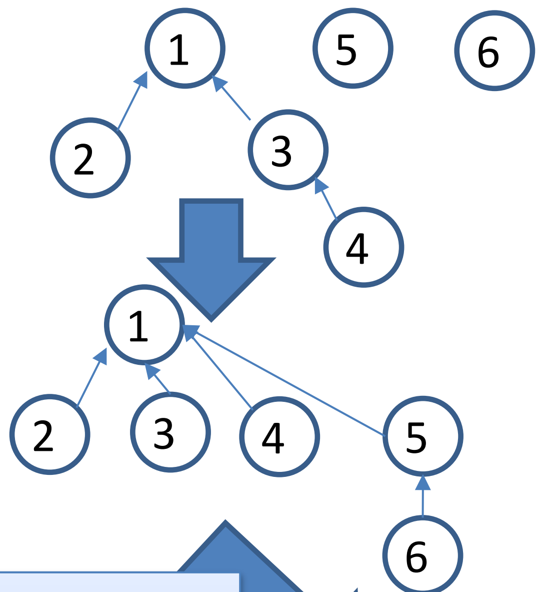
根据上面的转换，我们可以把一个Union-Find Program执行过程转换为一个等价的操作序列：

- 前面是一组wUnion操作,
- 之后是一组ExploreAndCompress操作.

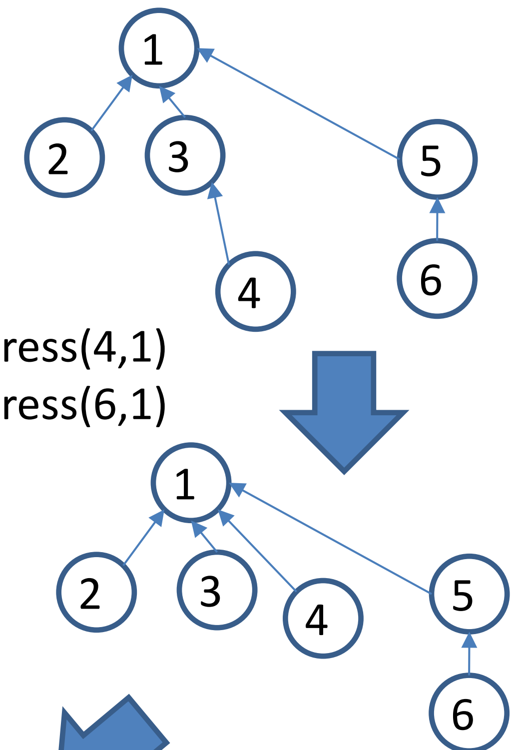
例子

- 对于6个元素的Union-Find

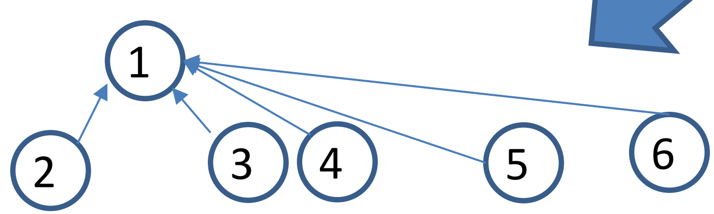
wUnion(1,2)
wUnion(3,4)
wUnion(1,3)
cFind(4)
wUnion(5,6)
wUnion(1,5)
cFind(6)



wUnion(1,2)
wUnion(3,4)
wUnion(1,3)
wUnion(5,6)
wUnion(1,5)
ExploreAndCompress(4,1)
ExploreAndCompress(6,1)



所以，一个Union-Find program可以看作是：
1、先使用一系列wUnion构造出一个森林
2、然后通过一系列压缩过程得到一个矮树森林



Definitions with a *Union-Find* Program *P*

- Forest *F*: the forest constructed by the sequence of *union* instructions in *P*, assuming:
  - *wUnion* is used;
  - the *finds* in the *P* are ignored
- Height of a node *v* in any tree: the height of the subtree rooted at *v*
- Rank of *v*: the height of *v* ***in F***
  - 静态的值 … 成具有更 …
  - 当v的父 … 会再被压 …

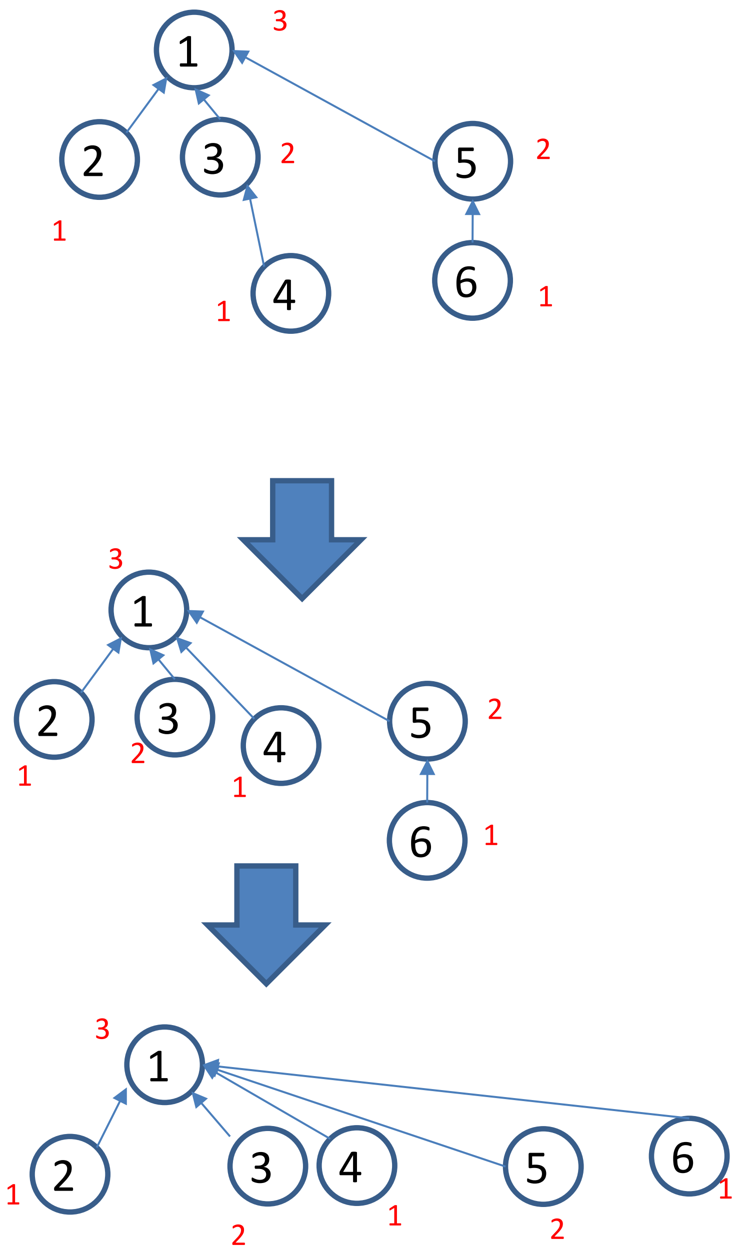
注意：

1、F是一个虚拟的森林。在P的运行中并不会真的出现，因为finds函数会改变树的结构。

2、Rank的主要用途是设置v被向上移动的上界。

如果我们把Union-Find等价地转换为wUnion + ExploreAndCompress序列，那么我们可以把Rank(v)看作是v压缩的起点

wUnion(1,2)
wUnion(3,4)
wUnion(1,3)
wUnion(5,6)
wUnion(1,5)
ExploreAndCompress(4,1)
ExploreAndCompress(6,1)



Constraints on Ranks in *F*

- The upper bound of the number of nodes with rank r ($r \geq 0$) is $\frac{n}{2^r}$
  - Remember that the height of the tree built by *wUnion* is at most $\lfloor \lg n \rfloor$, which means the subtree of height r has at least 2^r nodes.
  - The subtrees with root at rank r are disjoint.
- There are at most $\lfloor \log n \rfloor$ different ranks.
  - There are altogether n elements in S, that is, n nodes in F.

Increasing Sequence of Ranks

- The ranks of the nodes on a path from a leaf to a root of a tree in *F* form a strictly increasing sequence.
- When a *cFind* （*ExploreAndCompress*) operation changes the parent of a node, the new parent has higher rank than the old parent of that node.
  - Note: the new parent was an ancestor of the previous parent.
  - 当某个结点v的父节点是F中的某棵树的根节点时，不可能再被压缩。

Grouping Nodes by Ranks

- Node $v \in s_i$ ($i \ge 0$) iff. $\log^*(1+\text{rank of } v)=i$
  - which means that: if node v is in group i, then

    $r_v \le H(i)-1$, but not in group with smaller labels
- So,
  - Group 0: all nodes with rank 0
  - Group 1: all nodes with rank 1
  - Group 2: all nodes with rank 2 or 3
  - Group 3: all nodes with its rank in [4,15]
  - Group 4: all nodes with its rank in [16, 65535]
  - Group 5: all nodes with its rank in [65536, ???]

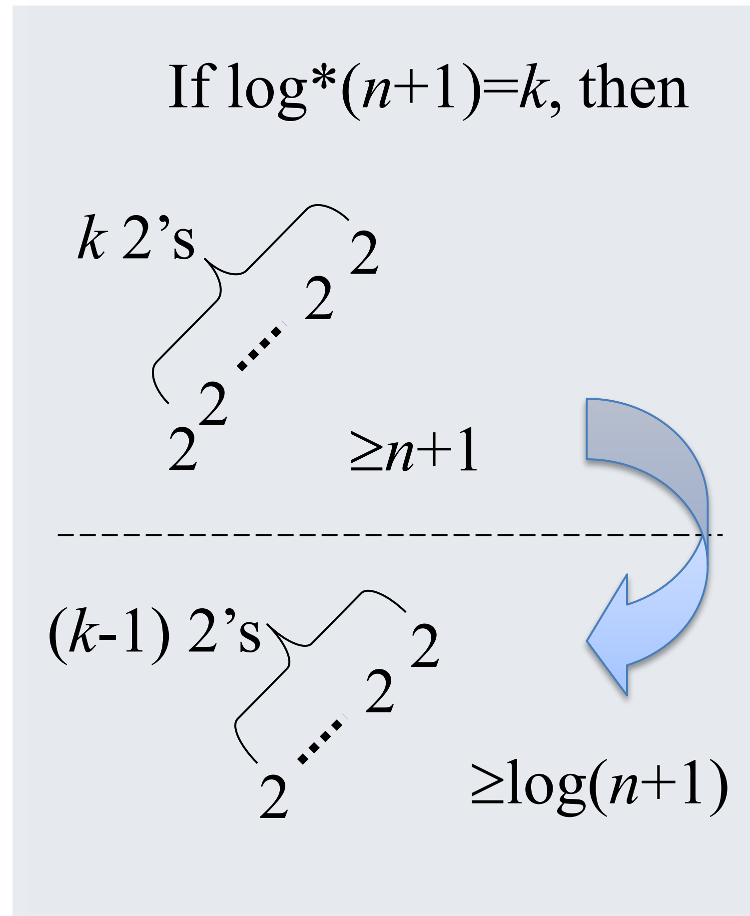
Group 5 exists only when n is at least 2^{65536}. What is that?

Very Few Groups

- **Node $v \in S_i$ ($i \geq 0$) iff.**

  $\log^*(1+\text{rank of } v)=i$

- Upper bound of the number of distinct node groups is $\log^*(n+1)$
  - The rank of any node in F is at most $\lfloor \log n \rfloor$, so the largest group index is $\log^*(1+\lfloor \log n \rfloor)=\log^*(\lceil \log n+1 \rceil) = \log^*(n+1)-1$



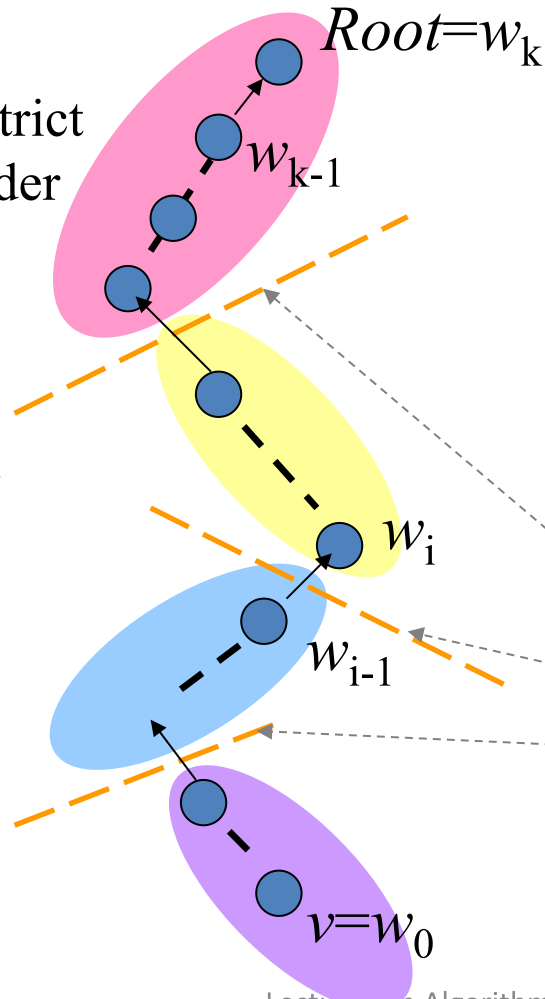
Amortized Cost of *Union-Find*

- Amortized Equation Recalled
  - amortized cost = actual cost + accounting cost

- The operations to be considered:
  - n makeSets
  - m union & find (with at most n-1 unions)

One Execution of *cFind*(w_0)

Groups in a strict increasing order



Group Boundary

Note: the ranks are not consecutive generally

Only when k=0,1, there is no parent change

For one *cFind* operation, the actual cost is $2k$ Not $2(k+1)$

Accounting cost is -2 for each pair of (w_{i-1}, w_i) for the the 2 nodes in the **same group only**, which we call a **withdrawal**.

Amortizing Scheme for *wUnion-cFind*

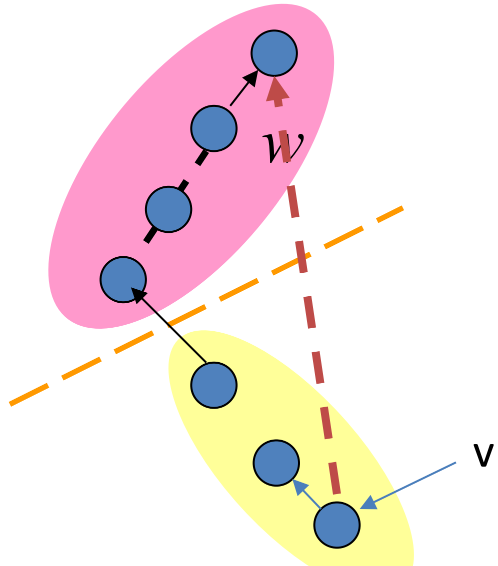
- makeSet
  - Accounting cost is $4\log^*(n+1)$ （即每个结点预存的钱）
  - So, the amortized cost is $1+4\log^*(n+1)$
- *wUnion*
  - Accounting cost is 0
  - So the amortized cost is 1
- *cFind*
  - Accounting cost is describes as in the previous page.
  - Amortized cost $\leq 2k-2((k-1)-(\log^*(n+1)-1))=2\log^*(n+1)$ (Compare with the worst case cost of *cFind*, $2\log n$)

Validation of the Amortizing Scheme

- We must be assure that **the sum of the accounting costs is never negative**.
- The sum of the negative charges, incurred by *cFind*, does not exceed $4n\log^*(n+1)$
  – We prove this by showing that at most $2n\log^*(n+1)$ withdrawals on nodes occur during all the executions of *c*Find.
  – 注意：每次withdrawal是-2.

Key Idea in the Derivation

- For any node, the number of withdrawal will be less than the number of different ranks in the group it belongs to
  - When a *cFind* changes the parent of a node, the new parent is always has higher rank than the old parent.
  - Once a node is assigned a new parent in a higher group, no more negative amortized cost will incurred for it again.
- The number of different ranks is limited within a group.



如果压缩v到w2的路径，路径上的每个结点accounting为-2; 每次压缩后，v的父节点的Rank必然增长。

压缩之后，v的父节点为w。
那么对于从v到它的某个祖先结点的路径第一条边是v->w，他们位于不同的group，accounting cost为0。

Derivation

- Bounding the number of withdrawals

The number of withdrawals from all $w \in S$ is:

$$\sum_{i=0}^{log^*(n+1)-1} H(i)(\text{number of nodes in group } i)$$

a loose upper bound of ranks in a group

The number of nodes in group i is at most:

$$\sum_{r=H(i-1)}^{H(i)-1} \frac{n}{2^r} \leq \frac{n}{2^{H(i-1)}} \sum_{j=0}^{\infty} \frac{1}{2^j} = \frac{2n}{2^{H(i-1)}} = \frac{2n}{H(i)}$$

So,

$$\sum_{i=0}^{log^*(n+1)-1} H(i)\frac{2n}{H(i)} = 2n \log^*(n+1)$$

Conclusion

- The number of link operations done by a *Union-Find* program implemented with *wUnion* and *cFind*, of length m on a set of n elements is in $O((n+m)\log^*(n))$ in the worst case.
  - Note: since the sum of accounting cost is never negative, the actual cost is always not less than amortized cost. The upper bound of amortized cost is: $(n+m)(1+4\log^*(n+1))$

Thank you!

Q & A

Yu Huang
http://cs.nju.edu.cn/yuhuang