CSCI 5454, CU-Boulder **Dhivya Mylsamy & Karuna Raja Reddy**

Lecture 15 **21 March 2012**

Disjoint Sets and Union Find

History:

The idea of Disjoint sets seems to be in the field for a long time. There have been many literature for worst case analysis. In 1975, Robert Tarjan proved the best upper bound in terms of inverse Ackermann function. In 2005, Raimund Seidel and Micha Sharir proposed the new analysis for the worst case of "union-find" algorithm which runs in the inverse Ackermann function. In 2007, Sylvain Conchon and Jean-Christophe Filliatre developed a persistent version of disjoint-set data structure.

Structure:

Disjoint set is a data structure which works on set of disjoint sets. The data structure provides the method for maintaining the set of elements in those sets and provides the following operations

MakeSet: It creates a single element set for all elements under consideration. Emphasizing the concept of disjoint sets, there should not be same element present in two sets.

Find: Find determines which set the particular element belongs to.

Union: Union combines or merges the two sets in to a single set.

Disjoint Sets data structure is also called as Union Find data structure due to the above operations.

Each set has a leader and remaining elements in that set are linked to the leader. So, when we want to find the set to which the particular belongs, it returns the leader of the set. Similarly, Union takes leader of the two sets and combines them so that all the elements in both sets follow a single leader.

Why Disjoint Sets?

Intuition is whenever we combine two disjoint sets, the time required will be addition of size of the sets involved. The purpose of this data structure is to reduce the running time and give the asymptotically faster operation. The effective tree representation of disjoint is proven to have running time of inverse Ackermann function.

Problem Domain

Disjoint sets can be used in many applications. It can be used for all path problems. Precisely, it is used in minimum Spanning Tree(MST) to combine the two components and choose leader for both components. It is also used to maintain connected components of a graph and to check whether any addition of edges and vertices form loops. The data-Structure can also be used for other path finding problems in networks or in games like Maze, JigSaw puzzles,etc.

Amortization Analysis

This method analyses algorithms by considering the entire sequence of operations of the program.

Amortization analysis is used in the problems where when a worst case happens, it alters the operation such that there won't be a next worst case for a long time. In other words, the frequency of the extremely costly operations is less.

This analysis requires the knowledge of entire sequence of operation. The idea behind this analysis is that costly operations which occur rarely should not be allowed to determine the fate of the entire program when those operations time can be normalized over large number of cheap operations.

Let S be a set of all the operations. $S = \{s_1, s_2, s_3, .. s_k\}$ represents the sequence of all the k operations on the disjoint-sets. Let the cost for a particular operation s_i be represented as cost(s_i) which gives the asymptotic cost of the operation. If cost(s_n) is O(n) and cost($s_{i\ to\ n-1}$) is O(1). Then the amortized cost for each operation is given as

$$\frac{1}{n} * \sum_{i=1}^{k} cost(s_i) = \frac{1}{n} * O(n) + (n-1) * O(1) = O(1)$$

Amortization Analysis is applicable to union-find where each union operation changes the cost of subsequent unions.

Array Representation of disjoint sets

Elements in all sets are represented in the single array. The index of the array represents the element and content of the array contains its immediate predecessor. If the element is the leader then the content of the array will be -1.

Makeset

MakeSet will make all the elements as leader to itself. Thus, all the elements in the array will contain the value -1. Makeset takes a running time of O(1) for each operation. If there are m operations, then it takes O(m).

-1	-1	-1	-1
7	8	9	10

Find

Find operation returns the leader of the element. To obtain leader of the particular element, find operation has to traverse through the array until it finds the value as -1. Once it returns the value as -1, then the index of that array represents its leader.

Consider the following example,

-1	7	8	9
7	8	9	10

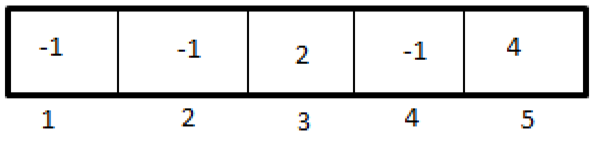
Here to find the leader of the element 10, first it checks the content 10, gets 9. Then checks the content of 9, gets 8. Then checks the content of 8 and gets 7. Then checks the content of 7 and gets -1. At this point, the leader is returned as 7 for element 10.

At the worst case if there are n elements in a set, then the running time will be O(n) as you need to check all n elements to find the leader of that set.

Union

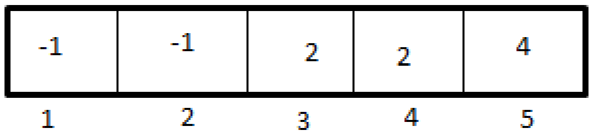
To perform union of two sets, first we have to find the leader of the two sets. This is done by calling find. Once we find the leaders, we need to make one of them as leader of the union and make the other leader point to the chosen leader.

For example,



-1	-1	2	-1	4
1	2	3	4	5

The aim is to find Union(3,4) . The leader of 3 is 2 and the leader of 4 is itself. Now, 2 is arbitrarily chosen as the leader of the union. So, 4 is made point to the new leader 2.



-1	-1	2	2	4
1	2	3	4	5

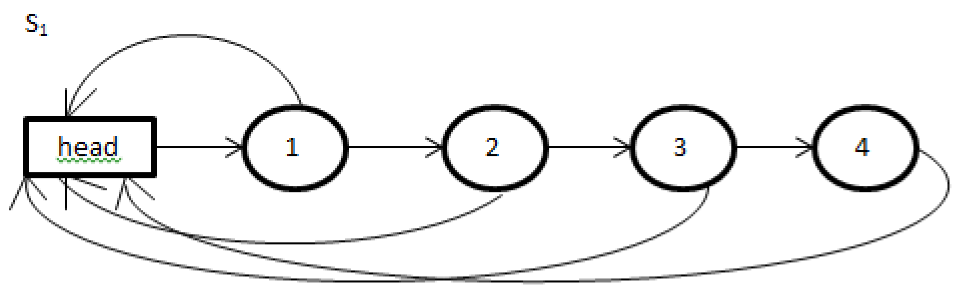
This union operation takes O(1) as it has to update only one of its leader.

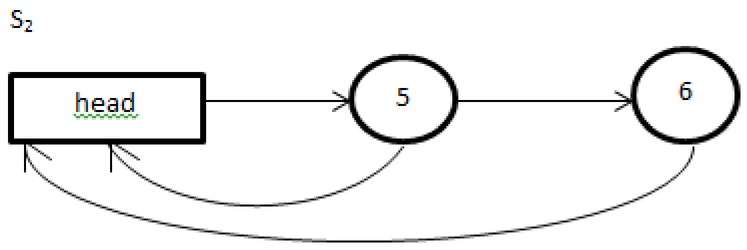
This array representation of the data structure is not effective as the find operation has running time of O(n).

Linked List Representation of disjoint sets

In linked list every set is represented by its own list. Head of the list is the representative of the set. Each element in the set has 2 pointers. One pointer points to the next element and the other points to the representative of the set.

Consider the following sets represented by linked lists:





MakeSet

MakeSet puts each element in its own set, creating m lists each of which contains a single element. It takes O(1) for each operation. If there are m elements, then running time becomes O(m).

Find

Find operation returns the leader of the element. Here as all elements are pointed to its leader directly, it takes O(1) time to return its leader.

Union

To find the union of two elements say x and y, first we retrieve the leaders of x and y. One of the leaders is arbitrarily chosen as the leader of the union. All the elements in the other list are made to point to the chosen leader. The running time of the union depends on the size of the list for which we update the pointer.

The worst case will occur when we make a list of size n point to the list of size 1. Thus the running time of union for n elements will be O(n).

Union based on size

To improve the running time, the smaller list is made to point to the larger list. Here you need to keep track of the list size in the leader node. When the lists are unified by this method, the running time will be O(logn) for n elements. A detailed analysis is presented below.

Amortized analysis for running time:

After object’s representative pointer has been changed once, set has >= 2 members
After object’s representative pointer has been changed twice, set has >= 4 members
.
.
After object’s representative pointer has been changed logk times, set has >= k members

For K <= n, so x’s representative pointer can be changed at most logn times.

Hence the running time is O(logn) for n elements. If there are n union operations, the running time is given as O(nlogn).

If there are m makeset operations and n elements, then total running time of the algorithm is O(m+nlogn). Amortized cost for makeset and find is O(1). Hence the amortized cost per Union is O(logn) since there can be at most n-1 union operations.

Tree Representation of disjoint sets

Each set is represented by a single tree. The node of the tree represents a single element. Each node points to its parent and the root of the tree points to itself.

MakeSet puts each element in its own set, creating n 'trees' each of which contains a single node. Find gives the representative or leader of the set by traversing the parent pointers. Union makes one leader of the set to point to the other.

MakeSet creates tree for every elements. Let the elements be from 1 to 8. Since ever tree has only one element, it acts as a root and hence points to itself

```
$\underline{MakeSet(x)}$
$parent(x) \leftarrow x$
```

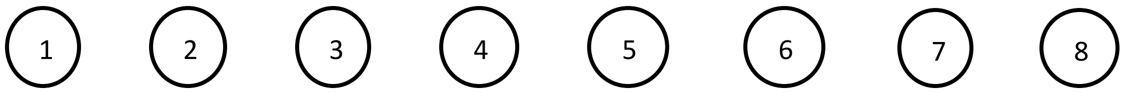


Figure 1: Tree and their pointer representation after MakeSet

Arbitrary Union and Simple Find

Let us take the union in the order of union(7,8), union(5,6), union(5,7) and union(4,5).

We can choose the leaders arbitrarily. At each union we have to find the leaders of the element and make any one of the leader to point to the other leader. Here, leader of the first set is chosen as leader of the union. Now there will be four trees as follows.

```
$\underline{MakeSet(x)}$
$parent(x) \leftarrow x$
```

```
$\underline{Find(x)}$
    $while\ x \neq parent(x)$
        $x \leftarrow parent(x)$
    $return\ x$
```

```
$\underline{Union(x,y)}$
    $\hat{x} \leftarrow Find(x)$
    $\hat{y} \leftarrow Find(y)$
    $parent(\hat{y}) \leftarrow \hat{x}$
```

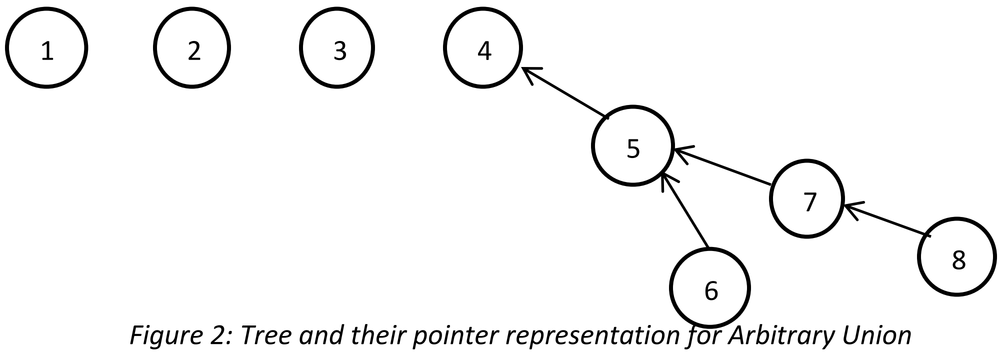


Figure 2: Tree and their pointer representation for Arbitrary Union

MakeSet takes constant time of $O(1)$ for each operation. For m elements it will take $O(m)$. Union operation also takes constant time of $O(1)$ without the find operation. Find operation is proportional to the depth of the element in that tree. In the worst case it takes $\theta(n)$ time.

Union by depth

In order to reduce the running time of find operation, we have to reduce the depth of the tree as far as possible. Whenever we perform the union operation, we can always make the leader of shorter depth tree point to leader of the deeper depth tree. In this way, the depth of the tree can be kept in check and the choosen leader of the deeper depth tree does not increase in depth.

Increase in the depth of the leader happens only when we have to union the same depth tree, in that case we have to choose the leader arbitrarily making the depth increase by 1.

```
$\underline{MakeSet(x)}$
$parent(x) \leftarrow x$
```

```
$\underline{Find(x)}$
    $while\ x \neq parent(x)$
        $x \leftarrow parent(x)$
    $return\ x$
```

```
$\underline{Union(x,y)}$
    $\hat{x} \leftarrow Find(x)$
    $\hat{y} \leftarrow Find(y)$
    $if(depth(\hat{x}) > depth(\hat{y}))$
        $parent(\hat{y}) \leftarrow \hat{x}$
    $else$
        $parent(\hat{x}) \leftarrow \hat{y}$
        $if(depth(\hat{x}) = depth(\hat{y}))$
            $depth(\hat{y}) \leftarrow depth(\hat{y}) + 1$
```

Now in the above example while doing last operation union(4,5), we can make 5 as the leader as the depth of tree 5 is larger than tree 4.

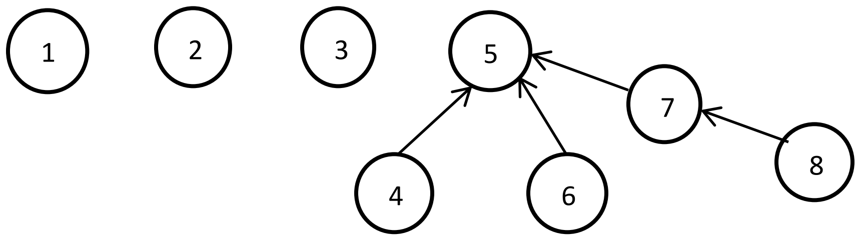


Figure 3: Tree and their pointer representation for union by depth

With this smarter union, we can see that size of the tree $\hat{x}$ will be atleast $2^{depth(\hat{x})}$. We can prove this statement by induction.

If depth($\hat{x}$) = 0, then it contains only single element being the root.

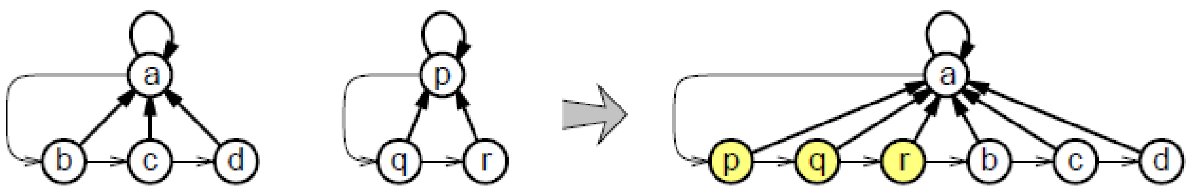
For d>0, when the depth of the tree $\hat{x}$ increases for the first time, it becomes the union of two sets whose depth is d-1. By, inductive hypothesis the size of those two trees will be 2^{d-1}. Hence, after the union, size of tree $\hat{x}$ = 2^{d-1} + 2^{d-1} = 2^{d}.

Later, when smaller depth tree is union with this tree, the number of elements will increase. Hence we can say that size of tree $\hat{x} \geq 2^{d}$.

By above argument, if there are n elements and all the elements are in one tree, still the depth of the tree will be $log\ n$. As find is directly proportional to depth of tree, it will run in $\theta(\log n)$. Both union and find will run in $\theta(\log n)$ and as usual MakeSet will take constant time $\theta(1)$ for each operation. The total time taken by this algorithm will be $O(m + n \log n)$.

Threaded Trees

We can still decrease the time taken by find operation by linking all the element to the root directly. Hence whenever union happens, we have to extra step to directly attach it to the leader. For this we need a thread to attach all element in the set. This might make union step longer, but find operation will always take only constant time.



$Figure\ 4$: $Union\ of\ two\ threaded\ trees.\ Taken\ from\ Jeff\ Erickson's\ lecture\ notes$

```
$\underline{MakeSet(x)}$
$leader(x) \leftarrow x$
$next(x) \leftarrow x$
```

```
$\underline{Find(x)}$
    $return\ leader(x)$
```

```
$\underline{Union(x,y)}$
    $\hat{x} \leftarrow Find(x)$
    $\hat{y} \leftarrow Find(y)$
    $y \leftarrow \hat{y}$
    $leader(y) \leftarrow \hat{x}$
    $while(next(y) \neq null)$
        $y \leftarrow next(y)$
        $leader(y) \leftarrow \hat{x}$
    $next(y) \leftarrow next(\hat{x})$
    $next(\hat{x}) \leftarrow \hat{y}$
```

Now find(x) gives constant time for every find operation. But union can take time $\theta(n)$ especially when you choose the leader from one element smaller tree, then you have to perform $n-1$ operations on larger tree. With n MakeSet and $n-1$ union operations, it might lead to $\theta(n^2)$.

Here we have to choose smaller size of the tree to connect to the leader of larger tree size. This gives only lesser number of pointers to be changed during union operation. For this, we need to have extra parameter to maintain the size of the tree.

```
$\underline{MakeWeightedSet(x)}$
$leader(x) \leftarrow x$
$next(x) \leftarrow x$
$size(x) \leftarrow 1$
```

```
$\underline{WeightedUnion(x,y)}$
    $\hat{x} \leftarrow Find(x)$
    $\hat{y} \leftarrow Find(y)$
    $if(size(\hat{x}) > size(\hat{y}))$
        $union(\hat{x},\hat{y})$
        $size(\hat{x}) \leftarrow size(\hat{x}) + size(\hat{y})$
    $else$
        $union(\hat{y},\hat{x})$
        $size(\hat{y}) \leftarrow size(\hat{x}) + size(\hat{y})$
```

As we require m MakeWeightedSet operations and n WeightedUnion operations, this algorithm takes $O(m + n\log n)$ in the worst case.

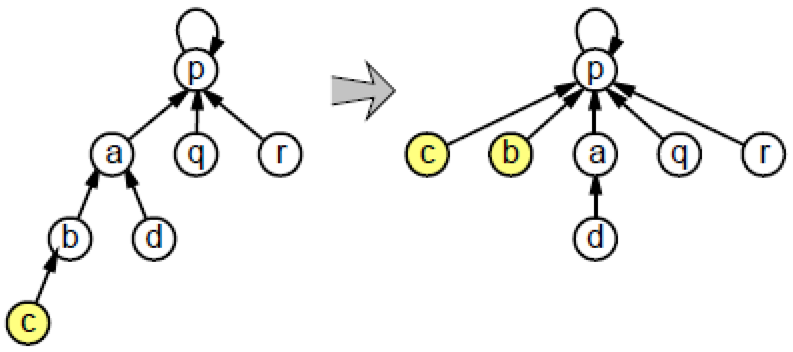
Whenever the leader of element x changes, the size of set containing x increases by factor of 2. Thus if the leader has changed k times, then there will be at least 2^k elements.

Hence if there are n elements in that set, then we can say leader has changed by lg n times. We can tell all the n element had lg n time leader changes, hence total time for union operation for all elements will be $O(n\ log\ n)$. Amortized cost for WeightedUnion will be $O(\log n)$.

Path Compression

We can take the Union by Depth and modify it with path compression. Path Compression ensures both find and union to run in constant time. The idea is whenever we find the leader of the element, we can make the that element directly point to its leader thus making the further find operation for that element faster. This is done by assigning the leader to the element in find while traversing for its leader.

As find operation in Union by Depth is $\theta(log\ n)$. This must give reduced running time. Here, instead of depth we use the term "rank". Rank here refers to the depth of the tree.



$Figure\ 5$: $Path\ compression\ during\ find(c).\ Taken\ from\ Jeff\ Erickson's\ lecture\ notes$

```
MakeSet(x)
  $parent(x) \leftarrow x$
  $rank(x) \leftarrow 0$
```

```
Find(x)
  $if\ x \neq parent(x)$
      $parent(x)$
                $\leftarrow Find(parent(x))$
  $return\ parent(x)$
```

```
Union(x, y)
    $\hat{x} \leftarrow Find(x)$
    $\hat{y} \leftarrow Find(y)$
    $if(rank(\hat{x}) > rank(\hat{y}))$
        $parent(\hat{y}) \leftarrow \hat{x}$
    $else$
        $parent(\hat{x}) \leftarrow \hat{y}$
        $if(rank(\hat{x}) = rank(\hat{y}))$
            $rank(\hat{y}) \leftarrow rank(\hat{y}) + 1$
```

Path compression increases the cost only by constant factor. Find still can run in $O(log\ n)$ in the worst case. Splay trees[1] have similar strategy which brings the frequently accessed element to the top which has running time as $O(log\ n)$. Our tree which can have more than one child should perform lot better than that.
There are two approximations to perform the analysis. First is $\boldsymbol{log^*n}$ and second is **inverse Ackermann function.**

To give those proofs, we have to use following properties of the algorithm

1. If node x is not a set leader then the rank of x must be smaller than the rank of its parent.

[1] Splay trees are binary search trees which stores such that recently accessed elements are easy to access again. It performs basic operations of union, search and find in $O(\log n)$. Source: http://en.wikipedia.org/wiki/Splay_tree

2. Whenever the leader of x's set changes, the new leader has larger rank than the old leader.
3. If there are n elements, then the highest possible rank is $lg\ n$.
4. The size of any set is exponential to the rank of its leader.

   $$size(\hat{x}) \geq 2^{rank(\hat{x})}.$$

   This is similar to property we have proven by induction in Union by depth.
5. For any integer r, there can be at most $\frac{n}{2^r}$ nodes with rank r.

   By intuition, we can tell if there are n elements, then number of sets containing 2^r elements will be atmost $\frac{n}{2^r}$.

   We can also prove this by using induction.

   Initially when r=0, then the number of nodes can be atmost n.

   The node with rank k comes after the union of two nodes with rank k-1. After the union, one of the two nodes gets higher rank and other with k-1 is no longer a leader. Hence, it creates two no longer active nodes of k-1.

   By induction, we will have at most $\frac{n}{2^{k-1}}$ such nodes.

   Hence, for nodes with rank k, the number of nodes will be $\leq \left(\frac{n}{2^{k-1}}\right)/2$

   $$\leq \left(\frac{n}{2^k}\right)$$

$O(log^*n)$ Amortized time

log^*n denotes the number of times one must take log on the number n before the value becomes less than 1.

$$lg^*n = \begin{cases} 1 & if\ n \leq 2 \\ 1 + lg^*(\lg n) & otherwise \end{cases}$$

Seidel and Sharir have proved this by using recursive decomposition in 2005. Previous proofs are based on the charging schemes where they analyzed block and path charges separately.

Here the proof is given according to the analysis of Seidel and Sharir.

Analysis given by them has included two more operations on set forests. Compress is the general operation that compresses any directed path not just paths that lead to the root. Shatter operation makes every node from root to leaf as its own parent.

```
$\underline{Compress(x, y)}$
    $\ll y\ must\ be\ an\ ancestor\ of\ x \gg$
    $If\ x \neq y$
        $Compress(parent(x), y)$
        $Parent(x) \leftarrow parent(y)$
```

```
$\underline{Shatter(x)}$
    $If\ parent(x) \neq x$
        $Shatter(Parent(x))$
        $Parent(x) \leftarrow x$
```

Running time of Find(x) operation is dominated by the running time of compress(x, y), where y is the leader of set containing x. Thus we can give the upper bound by analyzing arbitrary sequence of union and compress operations. We assume that arguments to Union operations are set leaders, so that each Union takes only constant worst case time.

As top node in the path to be compressed has specified to call compress operation, we can reorder the sequence of operations, so that every Union occurs before compress, without changing the number of pointer assignments.

Union operation takes only constant time in the worst case. We only need to analyze the running time of compress. The running time of compress is proportional to the number of pointer assignments plus O(1) overhead.

Let $T(m,n,r)$ denotes the number of pointer assignments in the worst case with any sequence of at most m compress operations.

n denotes the number of nodes in the forest that those compress operations are worked on.

r denotes the maximum rank of the forest.

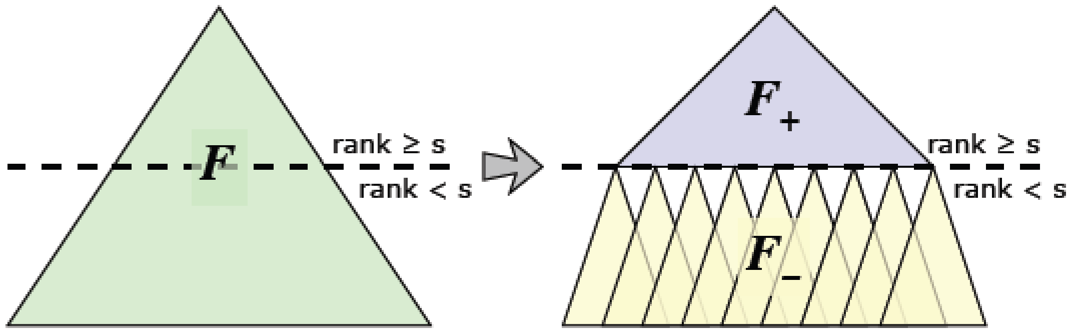
Each node can change parents at most r times as each new parent has higher rank than the previous one.

Thus, $\boldsymbol{T(m,n,r) \leq nr}$

Consider a forest with n nodes and maximum rank r. Let it have a sequence C of m compress operations performed on it.

Let T(F,C) denotes the total number of pointer assignments executed on this sequence.

Let s be the positive rank which divides the forest in to two sub-forests. The 'low' forest F_{-} contains all nodes with rank at most s, and 'high' forest F_{-} contains all nodes with rank greater than s. Since rank increases as we follow the parent pointers, every ancestor of high node is another high node.



$Figure\ 5$: $Splitting\ the\ Forest\ F.\ Taken\ from\ Jeff\ Erickson's\ lecture\ notes$

Let n_+ and n_- denotes the number of nodes in the forest F_+ and F_- respectively. Let m_+ and m_- denotes the number of compress operations occurs in the forest F_+ and F_- respectively.

Sequence of Compress operations on *F* can be decomposed in to a sequence of Compress operations on F_+, plus a sequence of Compress and Shatter operations on F_-. There is one constraint we have to introduce, where any node in the lower forest is not allowed to have the parent in higher forest. Thus replacing all pointer assignments of $parent(x) \leftarrow y$ with $parent(x) \leftarrow x$.

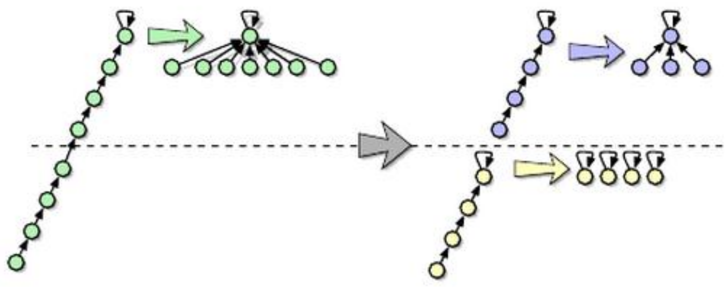


Figure 6: A compress operations in F splits in to Compress Function in F_+ and a shatter operation in F_- $. Taken\ from\ Jeff\ Erickson's\ lecture\ notes$

This reduction can be given in the form of code as follows

```
$\underline{Compress(x,y,F)}$
                    $\ll y\ must\ be\ an\ ancestor\ of\ x \gg$
$If\ rank(x) > r$
    $Compress(x,y,F_+)$        $\ll in\ C_+ \gg$
$else\ if\ rank(x) \leq r$
    $Compress(x,y,F_-)$        $\ll in\ C_- \gg$
$else$
    $z \leftarrow highest\ ancestor\ of\ x\ in\ F$
    $Compress(parent_F(z),y,F_+)$        $\ll in\ C_+ \gg$
    $Shatter(x,z,F_-)$
    $Parent(z) \leftarrow z$
```

The last assignment which although seems to be redundant, it is necessary for the analysis. This is used as a mirror to the assignment $parent(x) \leftarrow y$. This redundant assignment which happens immediately after the compress operation leads to m_+ redundant assignments.

Each node in x is touched by at most one Shatter operation. The number of all those pointer assignments is at most n.

Thus the sequence of C compress operations are partitioned in to C_+ and C_- such that

$$T(F,C) \leq T(F_+,C_+) + T(F_-,C_-) + m_+ + n$$

There are at most $\frac{n}{2^r}$ possible nodes with rank r. Hence,

$$n_+ \leq \sum_{i>s} \frac{n}{2^i}$$

$$\leq \frac{n}{2^s}$$

Hence, $T(F_+, C_+) < r\frac{n}{2^s}$ since $\boldsymbol{T(m,n,r) \leq nr}$

Let s = lg r**,** $(F_+, C_+) < n$

The recurrence relation becomes

$$T(F,C) \leq T(F_-, C_-) + m_+ + 2n$$

Substituting $m_+ = m - m_-$

$$T(F,C) - m \leq T(F_-, C_-) - m_- + 2n$$

Let T'(m,n,r) = T(m,n,r) – m

$$T'(m,n,r) \leq T'(m,n,\lg r) + 2n$$

Solving this recurrence will give $T'(m,n,r) \leq 2n\ lg^*r$

Hence,

$$\boldsymbol{T(m,n,r) \leq m + 2n\ lg^*r}$$

Thus, the compress function gives in the order of lg^*r which is a slow growing function and does not exceeds the value of 5.

The above

$$T(m,n,r) \leq m + 2n\ lg^*r$$

Gives us the more tighter bound than our earlier upper bound $T(m,n,r) \leq nr$. Solving with this, we will get another tight bound for compress which does not exceed the value of 4. That is inverse Ackermann function $\alpha(m,n)$.

The inverse Ackermann function which is defined for non – negative integers m and n as

$$\boldsymbol{A(m,n) = \begin{cases} n+1, \ \ if\ m = 0 \\ A(m-1,1), \ \ if\ m > 0\ and\ n > 0 \\ A\big(m-1, A(m, n-1)\big), \ \ if\ m > 0\ and\ n > 0 \end{cases}}$$

The path compression is said to have the tight upper bound of $O\big(\alpha(m,n)\big)$. The amortized cost of find algorithm is at least $\Omega(\alpha(m,n))$.

Applications

Disjoint data structures could be used in partitioning of a set. For example, it could be used in an application which involves keeping track of the connected components (Like edges) in an undirected graph. The edges which could form a cycle could be avoided to be added to the graph.

Minimum Spanning Tree (MST)

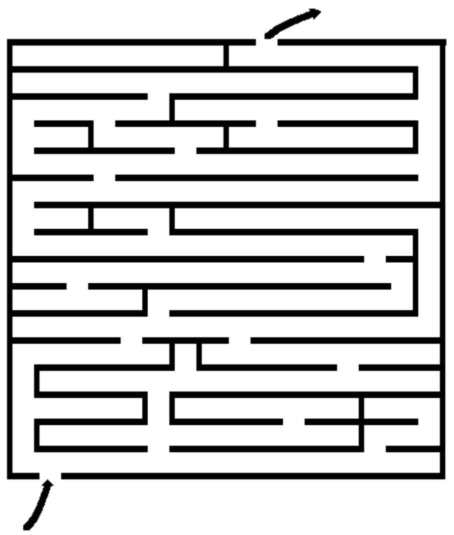
Given a connected, undirected graph, a spanning tree is a sub graph that is a tree and connects all the vertices together. A single graph can have many different spanning trees. Weights are assigned to each edge which is a number representing how unfavorable it is, and we use this to assign a weight to a spanning tree by computing the sum of the weights of the edges in that spanning tree. A minimum spanning tree is a spanning tree with weight less than or equal to the weight of every other spanning tree.

Kruskal's Algorithm

Kruskal's algorithm in graph theory finds a minimum spanning tree for a connected weighted graph.
Algorithm:
1. Create a set of trees and call it a forest F. Each node in a graph is a separate tree.
2. Create a set S containing all the edges in the graph
3. While S is nonempty and F is not yet spanning
    i. remove an edge which has minimum weight from S
    ii. if that edge connects two different trees, then add it to forest, combining two trees into a single tree
    iii. otherwise discard that edge

The result of the algorithm yields a single forest with minimum spanning tree of the input graph.

Maze problem



A maze could be created using various algorithms. Let us look at the algorithm which uses disjoint sets and union find for creating a maze.
Randomized Kruskal's Algorithm:
For creating a maze we use a randomized version of Kruskal's algorithm.

1. Create a list of all walls and create sets with each cell in a single set.
2. In a random order for each wall:
    i. If the cells divided by this wall belong to distinct sets:
        a. Remove the current wall.
        b. Join the sets of the formerly divided cells.

An efficient implementation of the algorithm above is by using disjoint-set data structure. In that case, each union and find operation can be performed in nearly constant amortized time O(lg *n). Running time of this algorithm is proportional to the number of walls in the maze.
The list of walls are either initially randomized or chosen randomly from a nonrandom set.

The maze generated by this method is fairly easy to solve as this algorithm produces a minimum spanning tree from a graph with equally-weighted edges. This leads to the production of regular patterns.

References

[1] http://en.wikipedia.org/wiki/Disjoint-set_data_structure

[2] Ananth. “Union – Find” Lecture notes for CPT s 223. Available at

http://www.eecs.wsu.edu/~ananth/CptS223/Lectures/UnionFind.pdf

[3] Robert Sedgewick and Kevin Wayn. ”Union-Find Algorithms” Lecture notes for the class AlgsDS07. Available at http://www.cs.princeton.edu/~rs/AlgsDS07/01UnionFind.pdf.

[4] Jeff Erickson. “Disjoint Sets”. Lecture notes. Available at the following link http://www.cs.uiuc.edu/~jeffe/teaching/algorithms/notes/16-unionfind.pdf

[5] Luca Trevisan. “Disjoint Set Union-Find” Lecture Notes for the class CS170. Available at www.cs.ucdavis.edu/~amenta/w10/trevisanNotes.pdf

[6] Robert E. Tarjan. “Efficiency of a good but not linear set union algorithm”. J. Assoc. Comput. Mach. 22:215–225, 1975

[7] Raimund Seidel and Micha Sharir. “Top-down analysis of path compression”. SIAM J.

Computing 34(3):515–525, 2005

[8] Andreas Klappenecker, “Disjoint Sets” Lecture notes for CPSC 411. Available at

faculty.cs.tamu.edu/klappi/cpsc411-f09/cpsc411-set7.ppt