Design and Analysis of Algorithms CSCI 5454
Lecture 15 14 March 2011
Nicholas Jeanette and Wei-Te Chen

1 Disjoint Set

In mathmatics, two sets are said to be *disjoint sets* if they have no elements in common. For example, 1, 2, 3 and 4, 5, 6 are disjoint sets. The formal definition of disjoint set is

$$S = \{S_1, S_2, \cdots, S_k\}, S_i \bigcap S_j = \emptyset, \forall 1 \leq i < j \leq k$$

where S is a collection of sets, and any two sets in S are nonoverlapping.
In computer science, *disjoint sets data structure* maintains this collection of sets. A bunch of applications and algorithms use this data structure to improve their work. For example, Kruskal's algorithm uses disjoint sets data structure to find the minimun spanning tree. It checks any cycles would occur or not when it wants to add a new edge to the tree. The key point for *disjoint sets data structure* is to assign a **representative** to each set. No matter how the representative is selected, we only care the representative fulfills the following: 1) it is different from other representatives 2) each time we ask for the representative of a set, without modifying the set, we get the same answer (representative). For convenience, we choose one of the elements in the set as the representative. Figure 1 shows an example for the representative. We select the left most elements in each set as representative.

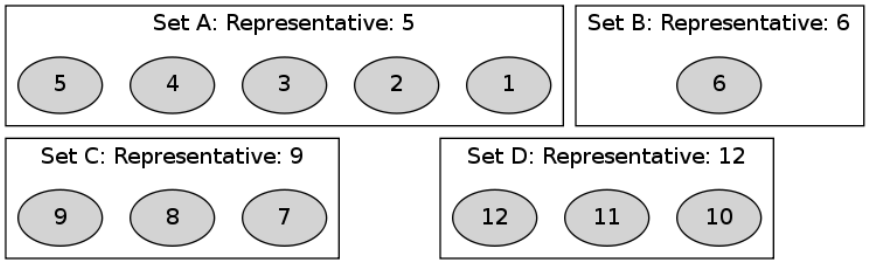


Figure 1: Disjoint set and its representative

A disjoint sets data structure should support the following three operations:

- **Make-Set**(x) creates a new set whose only member is x. Since every elements should belong to one set, we create sets which own exactaly one element at first.
- **Union**(x, y) unites the sets contain x and y, say S_x and S_y, into a new set that is the union of these two sets. The representative of the new set $S_{x \cup y}$ is any member of $S_x \cup S_y$.

- **Find-Set(x)** returns a pointer to the representative of the set containing x.

When we want to create a new disjoint sets with n elements, we need these three operations. At first we run the Make-Set operations n times to generate n disjoint sets which own exactaly one element. Then we run the Union operation a couple of times to merge sets and finally generate our disjoint sets data structure.

1.1 Application - Determining the Connected Components

Disjoint sets data structure models the partitioning of a set. One application is to keep track of the connected components of an undirected graph. A connected component of an undirected graph is a subgraph in which any two vertices are connected to each other by paths, and which is connected to no additional vertices. Figure 2 shows a graph with four connected components. At first we use the three operations in disjoint set data structure to create the connected components. We call this preprocessing step as ConnectedComponents. Since all elements in the same connected component are in the same set, which means these elements all have the same representative. Then we can check whether two elements, say V_a and V_b, are in the same connected component or not by using Find-Set operation. The running time for the Find-Set operation is quick ($O(1)$, which we will discuss later), so it makes the whole process faster. If Find-Set returns the same representitive, it means V_a and V_b are in the same connected component. Otherwise, they're not. This procedure is called SameComponent. The procedure for ConnectedComponents and SameComponents are as followed:

```
ConnectedComponent(G)
for all v ∈ V[G] do
  Make-Set(v)
end for
for all edge(u, v) ∈ E[G] do
  if Find-Set(u) ≠ Find-Set(v) then
    Union(u,v)
  end if
end for
```

```
SameComponents(u,v)
if Find-Set(u) = Find-Set(v) then
  return TRUE
else
  return FALSE
end if
```

Table1 illustrates how the disjoint sets in Figure 2 are generated by ConnectedComponents. The running time to check weather two vertices are connected or not depends on the running time of Find-Set.

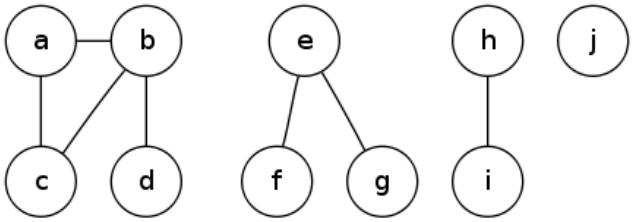


Figure 2: A graph with four connected components: {a,b,c,d},{e,f,g},{h,i}, and {j}

Table 1: The collection of disjoint sets after each edge is processed

Edge Processed	Collection of disjoint sets									
initial sets	{a}	{b}	{c}	{d}	{e}	{f}	{g}	{h}	{i}	{j}
(b,d)	{a}	{b, d}	{c}		{e}	{f}	{g}	{h}	{i}	{j}
(e,g)	{a}	{b, d}	{c}		{e,g}	{f}		{h}	{i}	{j}
(a,c)	{a,c}	{b, d}			{e,g}	{f}		{h}	{i}	{j}
(h,i)	{a,c}	{b, d}			{e,g}	{f}		{h,i}		{j}
(a,b)	{a,b,c,d}				{e,g}	{f}		{h,i}		{j}
(e,f)	{a,b,c,d}				{e,g,f}			{h,i}		{j}
(b,c)	{a,b,c,d}				{e,g,f}			{h,i}		{j}

1.2 Application - Maze Generation Algorithm

Another interesting application for disjoint sets data structure is the maze generation algorithm. Figure 3 shows a simple maze.

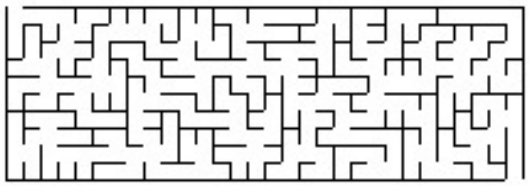


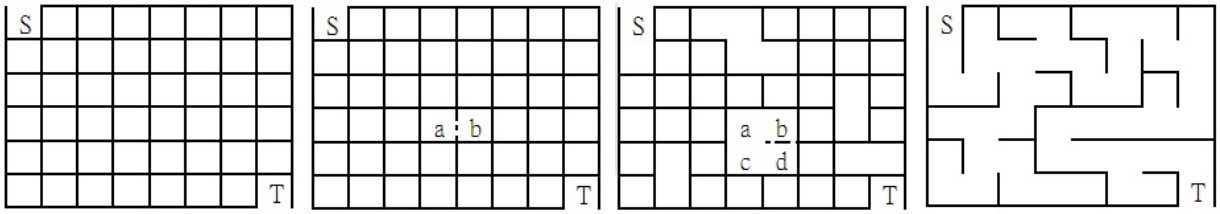
Figure 3: A simple maze

Our generated maze should fulfill these following requirement:

- All blocks in the maze can be reached by any other blocks (no closed space.)
- No cycle path in the maze.

The maze can be generated by following steps:

1. Generate a grid which is full of walls. Then choice the start(S) and terminal(T) cells on the border. (Figure 4(a))
2. Arbitrarily select a wall $Wall_{(i,j)}$ from maze, where $Wall_{(i,j)}$ is the wall between $cell_i$ and $cell_j$.
   (a) If there is no path between $cell_i$ and $cell_j$, then we remove $Wall_{(i,j)}$ ($Wall_{(a,b)}$ in Figure 4(b).)
   (b) If $cell_i$ and $cell_j$ have already been connected by a path, then keep $Wall_{(i,j)}$ ($Wall_{(b,d)}$ in Figure 4(c).)
3. Repeat Step 2 until all cells are connected. After that, we have an arbitrarily generated maze (Figure 4(d).)



(a) Full Walls Maze (b) Remove $Wall_{(a,b)}$ (c) Can't Remove $Wall_{(b,d)}$ (d) Complete Maze

Figure 4: Steps for generating maze

With the previous steps, we can use disjoint set data structure to generate the maze. Initially all cells are placed in their own set by Make-Set. Then we arbitrary choose a wall between $cell_u$ and $cell_v$. If $cell_u$ and $cell_v$ are in different sets, then we remove this wall and then do operation Union(u, v). We repeat previous steps until all cells are in the same set. The pseudo-code is listed as followed:

```
GenerateMaze(C) {
  for cell in C {
    Make-Set(cell);
  }

  W = ConstructFullWall(C);
```

```
  while(|S| > 1) {
    wall(u,v) = ArbitrarilySelectWall(W);
    if(Find-Set(u) != Find-Set(v)) {
      Union(u,v);
      RemoveWall(W, wall(u,v));
    }
  }
}
```

2 Linked-list Representation of Disjoint Sets Data Structure

A simple way to implement disjoint-set data structure is to represent each set by a linked list. The idea is all elements in same set are linked together with a linked list. A head pointer points to the representative of the set. Then the representative element links to following element and finally all elements in the same set are on the same linked path. Every element in the set link to their representative element. Also, a tail pointer points the last element in the set to make it more efficiency when we want to get the tail element in Union operation. Besides, we maintain an array to keeps the links of elements point to their containing sets. An example of a linked-list representation is shown in Figure 5(a).
Then we show how the linked-list representation runs Find-Set and Union operations. Find-Set is simple. Since each element maintain a link to their representative, Find-Set is completed by following this link and return the representative. For Union operation, a simple implement is shown in Figure 5(b). First we use the Find-Set operation to obtain the two representatives, say x and y. Then we find x's list tail element by the tail pointer. Then we append y's list to x's list, which means the tail of S_x update its link from null to y. Since S_y has been appended to S_x, all nodes in S_y need to be updated to point to x. The new set still uses x as representative.
In the next section, we will cover the analysis of the running time.

2.1 Running Time Analysis

The running time for Find-Set is $O(1)$ (the time to find out the set which holds the element plus the time to trace the head link). The running time for Union depends on the number of elements updated. Again we use Figure 5(b) as an example. All elements in S_c have to update their head links to the new representative, f. Therefore, the running time for Union is $O(max(|S_u|, |S_v|))$.
Next, we analyze the running time of generating a disjoint-set data structure. Suppose that we have n elements $x_1, x_2, \cdots, x_n$. Suppose also, that the total number of Make-Set, Union, and Find-Set operations is m. A sample sequence of operations in Table 2 is used to illustrate the running time. We first execute Make-Set for n times and run Union operations $n-1$ times. Then $m = (n) + (n-1) = 2n-1$. The ith Union operations updates i elements, which means we always

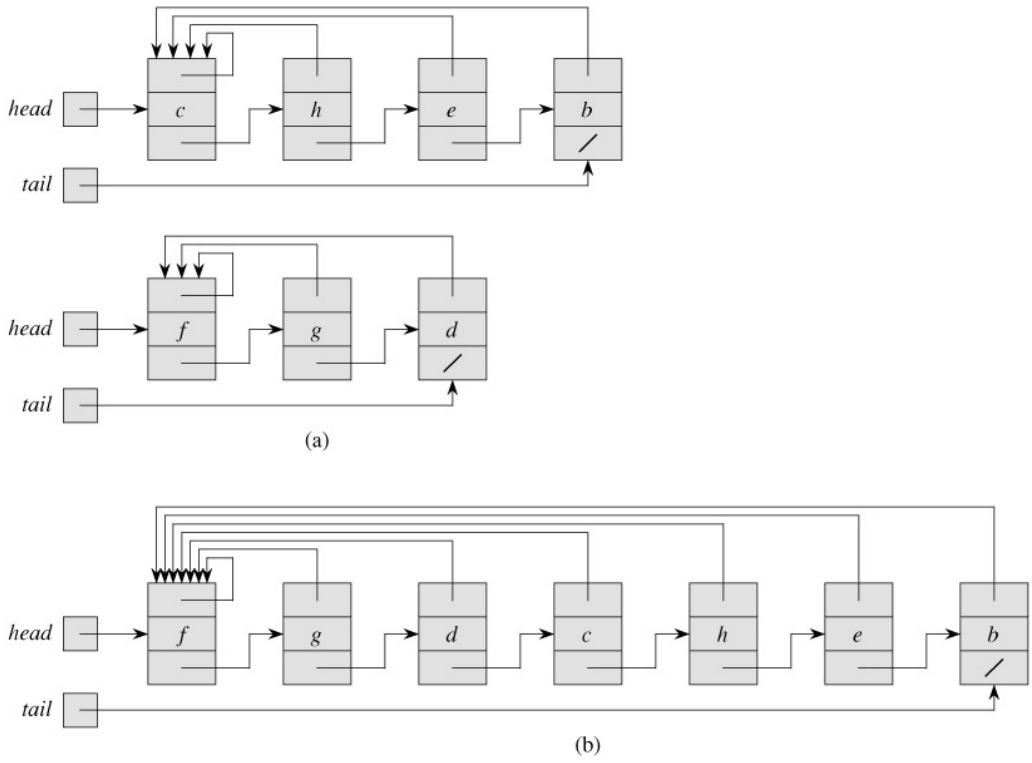


Figure 5: (a)Linked-list representations of two sets. (b) An example of Union(S_c, S_f)

append the longest list to the one-element set. So the worst running time (total number of elements updated by all $n-1$ Union operation) is

$$\sum_{i=1}^{n-1} i = \Theta(n^2)$$

Since the total number of operations is $m = 2n-1$, each operations on average requires $\Theta(n)$ time.

2.1.1 Improvement with Weighted-Union Heuristic

In the previous example, the Union operation always takes the worst case to run. A heuristic method to improve the running time is we always append a shorter list onto a longer list. The running time for constructing a linked-list representation of disjoint sets with this heuristic method is $O(m + n \lg n)$ time.

Theorem 1. *Using the linked-list representation of disjoint sets and the weighted-union heuristic,*

Table 2: A sequence of $2n-1$ operations on n elements.

Operation	Number of elements updated
Make-Set(x_1)	1
Make-Set(x_2)	1
$\vdots$	$\vdots$
Make-Set(x_n)	1
Union(x_1, x_2)	1
Union(x_2, x_3)	2
Union(x_3, x_4)	3
$\vdots$	$\vdots$
Union(x_{n-1}, x_n)	$n-1$

a sequence of m Make-Set, Union, and Find-Set operations, n of which are Make-Set operations, take$O(m + n \lg n)$ time.

Proof. We first compute an upper bound on the number of times the element's head link has been updated. An element x is selected. Since we always append shorter list to longer list, x is always in the shorter list. The first time when x needs updated its representation, the result list must have had a least 2 members. Similarly, the next time when x is upeated, the new result list must have had at least 4 members. Continuing on, we find out that when the new result list have at least k members, it means x has been updated $\lceil \lg k \rceil$ times where $k \leq n$. So the total number of updated for $k = n$ elements is $O(n * \lceil \lg k \rceil) = O(n \lg n)$.
Since the running time for Make-Set and Find-Set are both $O(1)$, and there are $O(m)$ of these operations, the total time for the entire generation is thus $O(m + n \lg n)$. □

3 Disjoint-Set Forests

Disjoint-set forests are disjoint-sets represented as rooted trees. In each node we hold one element of the set and the tree is the set. In this tree, each element points only to its parent. The root node of the tree points to itself, as its own parent. In the naive approach there are no speed advantages of this forest representation over the linked list implementation. We will briefly discuss this a little later.
Just like in the linked list version there are three operations, Make-Set, Find-Set, and Union. The Make-set operation creates a tree with only one node. This node is the root of the tree and thus points to itself. The Find-set operation just follows the path up the tree until it reaches the root node. The nodes that were visited along this operation create what is known as the find path. The final operation is Union, this just takes the pointer of one root node and points it to the root node of the tree being unioned with. Below is the psudocode for a naive implementation.

```
Make-Set(x){
  x.parent = x;
}
Find-Set(x){
  if (x.parent==x)
    return x
  else return Find(x.parent)
}
Union(x,y){
  xroot=Find-Set(x);
  yroot=Find-Set(y);
  x=y.parent;
}
```

We can see that these are all very similar to the linked list approach. In the Union function there is the possibility that the tree formed is very unbalanced if we continually append nodes to the same root. The unbalanced tree structure essentially becomes a linked list structure. When this is the case the Find-Set operation traverses the tree in $O(s)$ time where s is the size of the find path. Implemented in this way we see there are no real advantages over the linked list approach.

3.1 Improvements

There are two heuristics that can be applied to this implementation allowing an asymptotically optimal disjoint-set data structure. The first heuristic is union by rank. This is similar to the weighted-union in the linked list implementation. In this heuristic when unioning two sets together we would like to attach the root of the smaller tree to the root of the larger tree. In order to do this we need to know the rank of each node. The rank is an upper bound of the height of the node within a given tree. By keeping this rank the two roots can be compared and the one with a smaller rank can be attached to the root with a larger rank. After the union if the two roots being joined had equal ranks the new root nodes rank will be increased by 1.
The second method is path compression. This, like union by rank, is very simple and very effective. This is used while in the Find-set operation and it takes each node on the path and it changes its pointer from its parent to the root node. This does not change any of the rank information stored in the nodes, just the parent pointer. This is very useful when applied to longer trees as it flattens them. The decreases the time it takes to determine what sets a given node belongs to because every node points to the root. The Figure 6 gives an example of this flatting property. The tree on the left is before the path compression is performed. You can see if we want to know what set the bottom node belongs to we have to traverse all parents until we get to the root, this will take a considerable amount of time for very large trees. The image on the left is after the compression has been performed. Now determining what set any given node is in is one node away.
pathCompressor.jpg

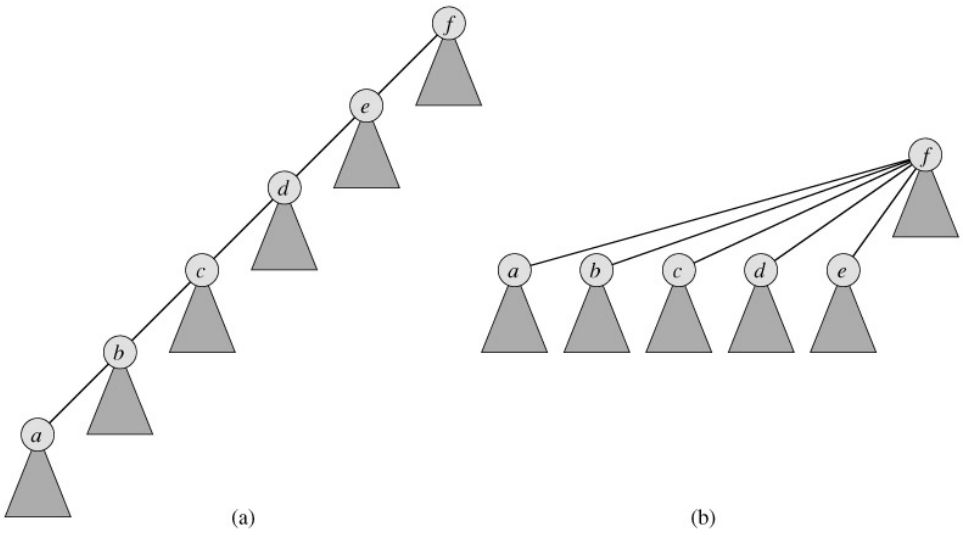


Figure 6: Path compression during the operation Find-Set

3.2 Implementation

Using the union by rank heuristic all we need to do is add a rank variable when creating sets and update it when unioning two sets together.

```
Make-Set(x){
  x.parent=x;
  x.rank=0;
}
```

There is no change in Find-Set.

```
Find-Set(x){
  if (x.parent==x)
    return x
  else return Find(x.parent)
}
```

The Union operation has the biggest change.

```
Union(x,y){
  xroot=Find-Set(x);//find the root of x
  yroot=Find-Set(y);//find the root of y
  if(xroot.rank>yroot.rank) {
```

```
    yroot.parent=xroot;
  }else{
    xroot.parent=yroot
    if(xroot.rank==yroot.rank){
      yparent.rank++;
    }
  }
}
```

The implementation of path compression changes the way Find-Set works by using a two pass method. The first pass occurs as the routine recurses up the tree, finding the path up to the root node. While the second pass occurs when the recursion returns setting the node to point at its parents parent. This change allows the node to point directly to the root node. Take note that this only changes the nodes that are on the find path. If this is called on a node half way down a tree, all of the nodes in the bottom half will still be several steps away from the root. Given a tree with height 10 (i.e the roots rank is 9) and we call Find-Set on a node with a rank of 5 all nodes with rank above 4 (ie 5-9) will have their parent point at the root. However, a node with a rank of 0 still must traverse 6 nodes to find what the root is, the path would be 0-¿1-¿2-¿3-¿4-¿5-¿9.

```
Find-Set(x){
  if(x!=x.parent){
    x.parent = Find-Set(x.parent);
  }
  return x.parent;
}
```

3.3 Application to Kruskal's Algorithm

Before continuing on to any analysis let us first look at Kruskal's Algorithm for finding Minimum Spanning Trees using the Disjoint-Set Forest structure. It is also important to note before we begin that in using this structure, it prevents the ability to delete edges.
If we recall the way we previously found MST, we checked every edge to see if it was safe to include in the tree. That is we did not add any cycles. If we wish to use an edge that would create a cycle we must first remove the edge with largest cost contained in that cycle before continuing. This is a very ineffective way of performing such a task.
We can use Kruskal's algorithm which will order all of the edge weights lowest to highest and then construct a set that contains all of the vertices in the graph. This construction can be done by starting with all vertices as individual disjoin sets and performing the Union operation until we have a set containing all of them.
Given a graph G(V,E) where V=a,b,c,d,e and weighted edges E=(a,b,1),(a,c,3),(b,c,4), (b,d,5),(c,d,4),(c,e,2),(d,e,8) where the last value is the weight associated with each edge. To start we create 5 disjoint sets a b c d

e and arrange our edges in order of weight from low to high (a,b,1),(c,e,2),(a,c,3),(b,c,4),(c,d,4),(b,d,5),(d,e,8). Notice that we have to edges with the same weight, it does not matter the order as they will both be looked at. First we look at the edge from a to b. We check to see what sets a and b belong to (using the Find-Set operation) we see they are their own. This allows us to union them together and we now have a,b c d e with the edge (a,b,1) being added to our MST list. We can continue in this fashion for several steps before we have a change. After looking at the first three edges we have a state that follows. The sets of vertices are a,b,c,e d and the path is (a,b,1),(c,e,2),(a,c,3). We now are on the edge that connects b and c. We check what set b belongs to and depending on how things were matched it would return the root node or a. This is the same node that c will return from the Find-Set operation. This means that b and c are in the same set so we can not perform a Union and as a result we do not add the edge. We continue and this time add the vertex d and we now have one set and a MST. With the final result being a,b,c,d,e and edges (a,b,1),(c,e,2),(a,c,3),(c,d,4).

3.4 Run time analysis

Both of these heuristics improve the running time for the disjoint-set forests when used individually. Together, they improve the run time over that of the disjoint-set forest with just one of the heuristics implemented. The run time of just union by rank s $O(m \lg n)$. This can be seen by the fact that we are building fatter trees. Trees with lower rank are joined to trees with larger rank. In the worst case, given n Make-Set operations, the result of which is n nodes with rank zero. If we perform Union on nodes that have rank zero the result is $\frac{n}{2}$ Unions half of the nodes now have rank 1 while the other have a rank of zero. If we continue performing this operation with nodes that share rank we essentially have built a b-tree. B-trees have the property of having the height bounded by $\lg n$. This upper bound on a single tree is then applied to m number of trees in the forest so the run time of union by rank is $O(m \lg n)$.
The run time improvement of just path-compression is a lot more complicated with a worst-case run time of $O(n + f * (1 + log_{2+f/n} n))$ where f are Find-Set operations. We have n Make-Set operations and thus at most $n - 1$ Unions. The time it takes to perform a Find-Set depends on the time it takes to locate the root of the tree. This changes proportionally with how many times find-set has already been performed. Recall the example above with the Find-Set call being performed on the node with rank 5. The next call to Find-Set that performs this compression will be proportional to how compressed the tree already is. So if we make the second call to Find-Set on the node with rank 0 it only needs to travel half as far as it would have had it been the first Find-Set operation. The running time based on the compression level is for just one of the find-operations, so we need to multiply that by the total number already performed, f.

3.5 Union by Rank with Path Compression using Amortized Analysis

If we combine the two the time per operation is only $O(m\alpha(n))$, where $\alpha(n)$ is the inverse of the Ackermann function. The Ackermann function is an extremely fast growing function so the inverse grows extremely slowly, in fact it is slower than any algebraic function. In any conceivable application of a disjoint-set data structure $\alpha(n) \leq 4$ and we can therefor see the running time is effectively linear in m, again where m is the number of disjoint-set operations on the n elements. [1] We can define the inverse of the Ackermann function as such

Definition. $\alpha(n) = \min\{k : A_k(1) \geq n\}$

This is the lowest level of k such that $A_k(1)$ is at least n. Only values that are so large, on the order of all atoms in the observable universe, that $\alpha(n) > 4$ so we can just use $\alpha(n) \leq 4$.

Lemma 1. *For all nodes x we have $x.rank \leq x.parent.rank$ with strict inequalities if $x \neq x.parent$. The value of $x.rank$ is initially 0 and increase through the time until $x \neq x.parent$; from then on $x.rank$ does not change. The value of $x.parent.rank$ monotonically increases over time.*

Proof. Each nodes rank begins with a value of 0 and following Make-Set is also set to be its own parent. When we perform Union on two trees the rank of one root gets incremented, this node will be the new root, while the other root has its parent pointer update to the new root. This nodes rank does not change. For each Union call the parent, the new root, has its rank incremented so the rank of $x.parent$ will continue to increase. □

Corollary 1. *As we follow any simple path toward the root, node rank strictly increase.*

Lemma 2. *Every node has rank at most $n - 1$*

Proof. Make-Set creates each node with $rank = 0$ and each Union increases the rank of one of two nodes in the operation by one. There can be at most $n - 1$ Union operations, two distinct trees per call. This lemma provides a weak bound on the ranks. □

[1] The Ackermann function is defied as

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

Cormann and Leiserson use the functional iteration-notation

$$A_k(j) = \begin{cases} j+1 & \text{if } k = 0 \\ A_{k-1}^{j+1}(j) & \text{if } k \geq 1 \end{cases}$$

We can show with induction that for any integer $j \geq 1$ $A_1(j) = 2j + 1$ and that $A_2(j) = 2^{j+1}(j+1) - 1$. If we just look at the first 4 levels of m with n we can see how fast this really grows.

$A_0(1) = 2$ $A_1(1) = 3$ $A_2(1) = 7$ $A_3(1) = A_2^2(1) = A_2(A_2(1)) = A_2(7) = 2047$ and $A_4(1) = A_3^2(1) = A_3(2047) \gg A_2(2047) > 2^{2048} \gg 10^{80}$ and that is larger than the atoms in the observable universe.

2

Before proceeding with the analysis we need to defined two helper functions. The first is a level function. This is the highest level that can be used, on the Ackermann function when applied to a node's rank, such that the value produced does not exceed the rank of its parent.

Definition. $level(x) = max\{k : x.parent.rank \geq A_k(x.rank)\}$

The second is the function iter. This is the maximum number of times we can iterate the Ackerman function, when applied to a node's rank at a given level, before we have a value that exceeds the rank of its parent. The level used is the level produced from the function level.

Definition. $iter(x) = max\{i : x.parent.rank \geq A^i_{level(x)}(x.rank)\}$

3

We can use a potential function to assign a potential $\phi_q(x)$ to each node in the forest after q operations. The sum of all potentials is the potential for the entire forest, $\phi_q = \sum_x \phi_q(x)$ is the potential of the forest after q operations. The forest is empty before the first operation and therefor can be set $\phi_q = 0$, this prevents the potential from ever being negative. The value of ϕ depends on if the node is a root after q operations. As long as the rank of the node is 0 or the node is a root, $\phi = \alpha(n) * x.rank$. If the node is not the root and the rank is greater than zero the potential of the node is $\phi_q(x) = (\alpha(n) - level(x)) * rank - iter(x)$. Using this potential function we can calculate the cost of each individual operation and amortize it to get the overall running time of the improvements.

[2]In the proof and analysis in the Cormen and Leiserson book they use a function Link that calls Find-Set on both trees to get the roots, in the pseudocode above all of this is done inside the Union function.

[3]A way to analyze the overall cost of the run time, with the heuristics, is by the potential method. The potential method is one of three methods used in amortized analysis. Amortized analysis averages the required time to perform a sequence of data-structure operations over all operations performed. An advantage to amortized analysis over average-case is, as we are averaging over a sequence of operations, probability can be taken out of the equation. This results in a guarantee of having the average performance of each operation, in a worst case situation. The potential method is similar to the accounting method in each operation has an amortized cost and may overcharge operations to compensate for undercharges later. Instead of counting this prepaid work as credit it is stored as "potential energy". This potential is applied to the whole rather than parts of the structure. Lets say we have n operations applied to a data structure D_i where i is the current state of the structure after i operations. Let $i = 0$ for the initial state. Then for each i there after let c_i be the actual cost for the ith operation. A potential function ϕ maps each data structure to a real number $\phi(D_i)$, resulting in the potential associated with the given data structure. The amortized cost $\hat{c}_i$ of the ith operation with respect to the potential function is given by the following definition.

Definition. $\hat{c}_i = c_i + \phi(D_i) - \phi(D_{i-1})$

Therefor the total amortized cost of the n operations is $\sum_{i=1}^{n} \hat{c}_i$.

If the potential difference between two states i and $i-1$ is positive we have an overcharge and the potential increases. On the other hand if the difference is negative, it is an undercharge and the decrease in the potential pays for the operation.

Recall the definition of our helper function level. $level(x) = max\{k : x.parent.rank \geq A_k(x.rank)\}$ We can show that $0 \leq level(x) < \alpha(n)$. First we show the lower bound by using the lemma on rank. The rank of the node's parent is at least that of its rank plus 1, and that is equal to $A_0(x.rank)$. This implies that the level is always non-negative. Showing the upper bound is harder. $A_{\alpha(n)}(x.rank) \geq A_{\alpha(n)}(1)$ because the Ackermann function is strictly increasing. By the definition of the inverse Ackermann function, $A_{\alpha(n)}(x.rank) \geq n$ and that is strictly larger than the rank of our nodes parent. We then have $level(x) < \alpha(n)$.
Recall the other function used iter, $iter(x) = max\{i : x.parent.rank \geq A^i_{level(x)}(x.rank)\}$. This is used in the potential for non-roots with rank larger than 0. We can show that when the rank is greater or equal to 1, the function is between 1 and the rank, ie. $1 \leq iter(x) \leq x.rank$. It can be shown by using the definition of level and functional iteration that iter is at least 1. $x.parent.rank \geq A_{level(x)}(x.rank)\ A^1_{level(x)}(x.rank)$. Using the definitions of Ackermann and the helper function level we can show that iter is at most the rank of our node. $A^{x.rank+1}_{level(x)}(x.rank) = A_{level(x)+1}(x.rank) > x.parent.rank$. As the rank of the node x's parent is monotonically increasing, level(x) must also increase in order for iter(x) to decrease.

Lemma 3. *For every node x, and for all operation counts q, we have $0 \leq \phi_q(x) \leq \alpha(n) * x.rank$*

Proof. By definition if x is a root or the rank is 0 we are done. In the other case we can get a lower bound if we maximize the two helper functions; level and iter. $((\alpha(n) - 1)$ is the upper bound on level and x.rank is the upper bound for iter. When we apply this to the potential function we get $\phi_q(x) \geq (\alpha(n) - (\alpha(n) - 1)) * x.rank - x.rank = 0$. Similarly the upper bound can be attained by minimizing the two functions and solving as before. 0 is the lower bound on level and 1 is the lower bound for iter. The result is $\phi_q(x) \leq (\alpha(n) - 0) * x.rank - 1 < \alpha(n) - x.rank$. □

Corollary 2. *If node x is not a root and its rank is greater than 0, the potential is less than its rank multiplied by the inverse Ackermann function.*

3.6 putting it all together

After all of the set up we can now see how each operation affects the node potentials. By looking at the changes in potential as a result of each operation we can figure the amortized cost.

Lemma 4. *Let x be a non-root node and the qth operation be Union of Find-Set. Then after the operation $\phi_q(x) \leq \phi_{q-1}(x)$. If the rank is at least 1 and there is a change to either level or iter, as a result of the qth operation, $\phi_q(x) \leq \phi_{q-1}(x) - 1$. There is no way for the potential of x to increase . Additionally if the rank is positive and there is a change in level or iter, the potential of x will decrease by at least 1.*

Lemma 5. *The amortized cost of each Make-Set is $O(1)$*

Proof. Let the qth operation be Make-Set(x). The result of this operation is the creation of a node with rank 0 such that its potential is also 0, $\phi_q(x) = 0$. This call has no affect on any other nodes so $\phi_q = \phi_{q-1}$. The cost of the operation is therefore $O(1)$. □

Lemma 6. *The amortized cost of each Union is $O(\alpha(n))$.*

Proof. Without a loss of generality we can suppose that y becomes the root of the new tree produced by the operation Union(x, y). The only nodes that may change their potential are x, y and all of y's children before the operation. If we use the lemma from the start of this section we can notice there is no way for the children of y to increase their potential due to a Union. The node x was a root before the qth operation. From the definition of a roots potential and the corollary 2 in the case the rank is non-zero, we can see x's potential can not increase.
Finally looking at the potential of node y we see it is the only one that could potentially increase. Prior to the Union operation y was a root, and after the operation it maintains its root status. In this operation one of two things could have happened to y's rank, it stayed the same or it increased by 1. Looking back at the definition for the potential for roots $\phi_{q-1}(y) = \alpha(n) * y.rank$, we can see that keeping a rank of 0 after the operation does not change the potential. If there is an increase in rank by 1 we have $\phi_q(y) = \phi_{q-1}(y) + \alpha(n)$. Therefore the amortized cost of Union is $O(\alpha(n))$. This can be seen by $\phi_q(y) - \phi_{q-1}(y) = (\alpha(n) * (y.rank + 1)) - (\alpha(n) * y.rank)$. □

Lemma 7. *The amortized cost of each Find-Set is $O(\alpha(n))$.*

Proof. Lets assume that the qth operation is a Find-Set and we have a find path of size s. The cost of Find-Set is the $O(s)$. We need to show that none of the nodes have an increase in potential and that in fact the maximum of either 0 or $s - (\alpha(n) + 2)$ nodes along the find-path actually have a decrease in their potential. From 4 we know that as long as the node is not a root it does not have an increase in potential. Looking again at the definition of the potential for a root node we can see its potential does not change, the rank remains the same for Find-Set.
Let x be a node on the find-path such that its rank is non-zero, and y be a non-root node along that same find-path such x occurs before y on the path. Additionally we must have two nodes x and y such that $level(x) = level(y)$ in the $q - 1$th operation. The nodes that do not satisfy this condition for x are the node whose rank is 0, the root node, and last node w that satisfies $level(w) = k$, where k is each value between 0 and $\alpha(n) - 1$. This results in a total of at most $\alpha(n) + 2$ nodes on the find-path. If we fix such a node x we can see the potential decreases by at least 1. Let $k = level(x) = level(y)$ then before the path compression by the definitions of iter and level we have, $y.parent.rank \geq A_k(y.rank)$, and $x.parent.rank \geq A_k^{iter(x)}(x.rank)$. Due to the fact that y comes after x on the find path and the strict increase of node rank along a simple path $y.rank \geq x.parent.rank$. Using these three inequalities together we can get that $y.parent.rank \geq A_k^{iter(x)+1}(x.rank)$.
Recall with path compression on Find-Set we have nodes x and y update their pointers to point to the same parent. This is done by the two-pass method of recursively finding the root and upon

returning updates the pointers. After this compression we know that the parents rank will be equal and therefore does not decrease y.parent.rank. Using the inequality recently established and the equality just established, the fact the rank of node x does not change we have $x.parent.rank \geq A_k^{iter(x)+1}(x.rank)$. Path compression causes iter to increase by at least $iter(x) + 1$ or $level(x)$ to increase. The later happens if iter becomes at least the rank+1. In either case we have the potential of x decreasing by at least 1, recall the definition of potential for non-root nodes.
So by 4 we get $\phi_q(x) \leq \phi_{q-1}(x) - 1$ and the decrease in the potential of x by at least 1. With the cost of Find-Set being $O(s)$ and the fact that the total potential decreases by $max(0, s-(\alpha(n)+2))$ so the amortized cost is $O(s) - s - (\alpha(n) + 2) = O(\alpha(n))$. □

From each of the last 3 lemmas we can form the following theorem

Theorem 2. *A sequence of m Make-set, Union, and Find-Set operations, n of which are Make-Set, can be performed on a disjoint-set forest with union by rank and path compression in worst case time of $O(m\alpha(n))$*

network connectivity good example

References

[1] T. H. Cormen, C. E., Leiserson, R. L. Rivest,and C. Stein. *Introduction to Algorithms, Third Edition, 2009.* The MIT Press: Basic Books.

[2] Disjoint Sets on WikiPedia: `http://en.wikipedia.org/wiki/Disjoint_sets`

[3] Disjoint-set data structure on WikiPedia: `http://en.wikipedia.org/wiki/Disjoint-set_data_structure`

[4] Maze generation algorithm on WikiPedia: `http://en.wikipedia.org/wiki/Maze_generation_algorithm`

[5] Professor Steve Allan course note: `http://digital.cs.usu.edu/~allan/DS/Notes/Ch24.pdf`