Course: Introduction to Graph Algorithms

Professor: C Pandu Rangan

Department: Computer Science and Engineering

Institute: IISc

Week: 06

Lecture 27 Kruskal's Algorithm 3

Namaskara, we continue our discussions on the efficient implementation of Kruskal's algorithm. What we have seen is that the execution of the Kruskal's algorithm very closely resembles the act of handling a partition. The way in which we are building the edges in A and the way in which the corresponding partitions are processed have high level similarity that we were focusing on implementing a partition ADT, partition abstract data type okay. Towards that we have seen a name array representation. Partition of a set. V is the set 1, 2, 3, n S1, S2, Sk is a partition of the set, Si intersection, Sj is empty, union Si equal to V. This is a partition so this is a current partition the partition will dynamically change.

So we have two operations find operation and union operation, these two operations define the abstract data type.

$$V = \{1, 2, 3, \ldots n\}$$

$$S_1, S_2 \ldots S_k \qquad S_i \cap S_j = \phi$$

$$\cup S_i = V$$

Find (x)

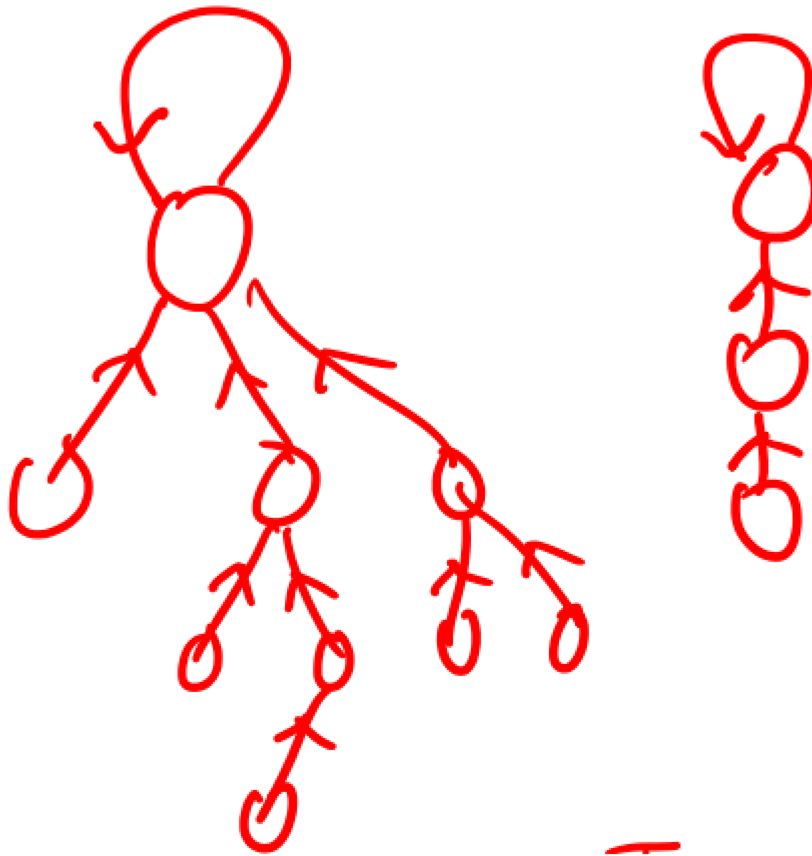
Union (A, B)

Abstract data type is a mathematical model together with certain set of operations the mathematical model is partition. and find returns the name of the set containing x. Union AB is combining two different sets and creates one bigger set which is the union of the two sets.

How do we implement these operations because our algorithm has been expressed in terms of these operations. So we have to discuss the way in which we would implement this. We have already seen a name array implementation and a name array representation. A name array representation, find x was taking order one time and union was taking order n time,

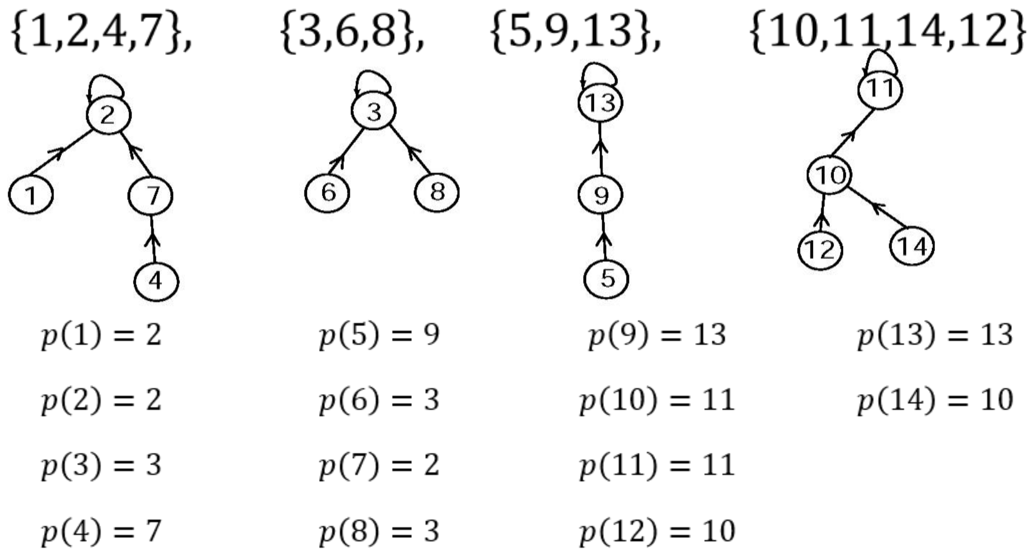
this what we have seen okay. Now we are going to look at inverted forest data structure and then finally conclude with the complexity analysis of Kruskal's algorithm, okay.

The inverted forest or in-forest is a collection of trees in which the edges are directed towards the root that is called in-tree. So it is a directed one if this is the root there is no restriction on the size or shape. If all edges are directed towards the route this is called in-tree, in-forest means we will have a collection of such entries and that is called in-forest or inverted forest okay. We will also have a self loop at every tree okay. So this is called inverted forest with the self loop at the root in every tree, this is a kind of a data structure that we would use okay we will see how to represent it.



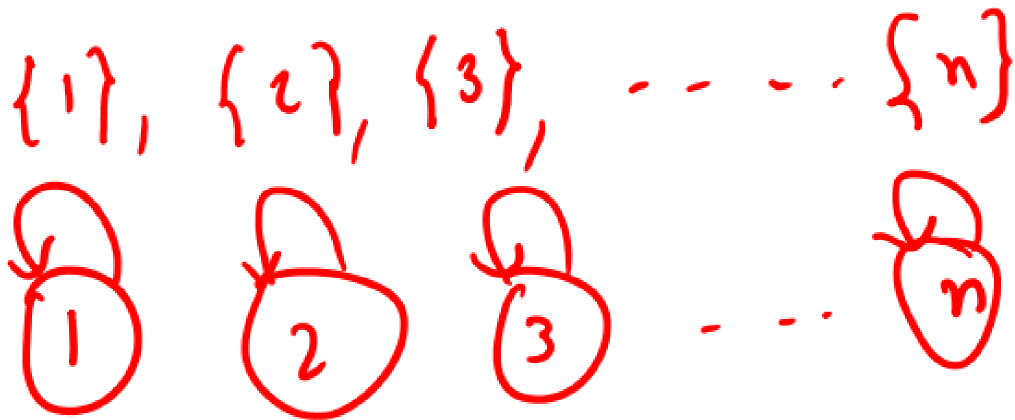
So first of all a k partition is going to be represented by an in forest with k trees okay a partition has got several sets, a k partition will have k sets, the corresponding representation will have k in-trees okay for each set one tree, and in the tree in each node one value of the set, that is how we are going to build the representation. We will place the name of the set in the root, okay the name of the set will be placed in the root , other elements are distributed arbitrarily in the tree, one value per node, root carrying the name of the set, one tree per set this is going to be the in-forest or inverted forest representation. So let us look at this example, this example will clearly, so you can see that there are 4 sets this is a partition of 1 to 14 I have split that set 1 to 14 in 4 sets. Each set is represented by a corresponding in-

tree 1 2 4 7, it has been put here in a tree 3 6 8, that is put in another tree 5 9 13, that is put in another tree 10 11 14 12, that is in another tree, 1 tree per set, each node has one value of the set. Now you can see the root is denoted by a self loop, the root is denoted by a self loop. So there are 4 sets and 4 trees, all the trees are going to be represented in a single array, that is the power of so called implicit representation Okay, implicit means hidden all these trees are hidden in one array and that is called parent array okay 7 is parent of 4 okay 7 is parent of 4, 2 is parent of 7. So now you can see I have filled that all the 4 trees.



Parent of 4 is 7 you can see that 4 is directed towards 7, parent of 4 is 7, parent of 8 is 3 you can see that parent of 8 is 3, parent of 5 is 9 where is parent of 5 you can see parent of 5 is 9 because 5 is directed towards, parent of 9 is 13 you can see that parent of 9 is 13. sets parent of that value to be the same value, parent of 13 is 13, parent of 11 is 11 because at 11 you have a self loop, parent of 2 is 2 because 2 is at the root, root is pointing to itself, so the parent of that node is itself, it is not a different node, okay so you can see that this single array represents all the 4 trees, 1 array okay. So the current partition is represented by 1 array, any partition is going to be represented by the same array by changing the values.

For example you want to represent the partition 1, 2, 3 n. Each set is to be represented by one tree, therefore this has to be represented by a tree and this is the only one node because the set has got only one element, this is the root and therefore it is a self loop, 2 a self loop, 3 a self loop, n there will be a self, So parent of 1 is 1 parent of 2 is 2 in general parent of i is i and so on.

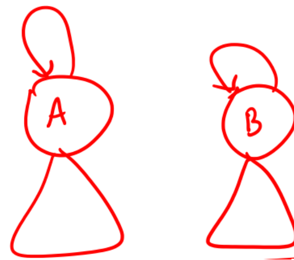


So this partition is pictorially represented like this by a parent array like P of i equal to i 1 less than or equal to i less than or equal to n.

$$p(i) = i \cdot$$
$$1 \leq i \leq n$$

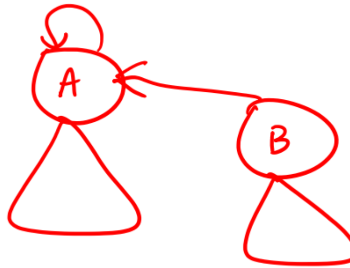
So you can see how a partition is represented by an array. Keep in your mind about the picture as well as the corresponding array representation. So, that the mapping is very helpful in understanding what is happening in the algorithms ok.

So, how do you find the union of two sets ok. Union of two sets is done by what is known as tree hooking. okay. So if I have a tree like this, if I have another tree like this, it is A, it is B. Suppose you would like to merge these two sets, okay. So I will show you how the two trees are merged by tree hooking. okay



So let me show that in let A be the root of a tree so there is some picture and B is another set, you want to do union A B, when you take union A B, both the sets are put together into a single set. and there is only one element at the root which is the name of the resulting set. So tree hooking is going to be done like this, let us say you are keeping A as it is, then what you do is instead of B pointing to itself, let B point to A, that is parent of B was earlier B

this is changed to A. When parent of B is A, we have edge from B to A right, that is how the parent array is defined.



Look at the picture why parent of 5 is 9 if you look at 5 it is here 9 is here from 5 to 9 there is a directed edge. So when parent of B equal to A, then B to A there will be an edge. So this structure is transformed to this structure, here you have two independent trees here there is only one structure A is the root B is attached to A in order to do that, what is that you have to do very simple only one step parent of B equal to A, that is all you have to do. Alright therefore union AB is implemented as parent of B equal to A

Union (A, B)
p(B) = A

You can decide to make parent of A to B that is also possible if you do that A will be hooked to B that is why this is called tree hooking. Union is implemented by tree hooking means you should visualize pictorially this one, in your program it is implemented by only one step.

Look at the surprising feature of this, you may have two sets one set may have 1000 elements another set may have 1500 element. Finding the union of the set the union will have 2500 elements in spite of all of that the union is done by one operation. Conceptually it is a very powerful representation, in this representation in one step I can find a union of any two sets of any size. It is independent of the size, earlier the union operations cost depends on its size because every elements name must be changed, I do not have to do that here right, simply tree hooking, automatically throws the old two sets, brings in a new tree, representing the union of the two sets. Union is done very efficiently

What about find, find is little tricky here because the name of the set is in the root, find operation, find x, you start from x, x is in some tree, But what you want is the root of the tree and the value that is in the root of the tree, for example find 14, 14 is in this tree the root is 13 but how do I know the root of the tree. So from 14 I keep going up and up using the parent pointer. This is called parent pointer chasing or ancestor chasing, go to the ancestor parent of parent of, so when you keep going that way because all the edges are

directed towards the root you will reach the root, when you reach the root output the value. How do you know you have reached the root? Root has a property that p of x equal to x. p of x equal to x if x is root because only root has self loop all other values will take to a different value.

$$p(x) = x \text{ iff } x \text{ is at the root}$$

So p of x will not be equal to x if the node is not a root node, if it is a root node p of x equal to x. So you know you have reached to the root, when you are finding an x such that p of x is also x. That is the reason why here is the code for the find operation.

$y = p(x)$

while $(y \neq p(y))$

$y = p(y)$

Return (y)

You start from the parent of x and keep going to the parent of the x, as long as the parent is not the original value. If the parent equal to original value you got the root and you output that.

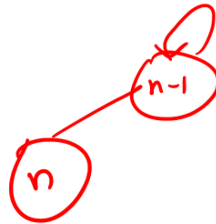
So while y not equal to parent of y that is as long as you are in a node, which is not a root y equal to p of y. y is updated to its parent so y is a moving variable x is a fixed node, y will keep moving y will become p of y, that means y will first be parent of x then it will be its parent that means grandparent of x and that way y will go through the ancestor list and y will reach the root. How do you know y has reached the root? y will be equal to p of y when y is equal to p of y the while loop will terminate and the next statement is returned y. let us look at this example, what is find 14 x equal to 14 what is y? y is parent of x, y equal to p of x and y is 10, for y is y equal to parent of y no parent of y is 11 y is 10. This is not equal so I am in the while loop in the while loop I update y to parent of y so this y will be updated here, is y equal to parent of y, no because it is not the while loop says update y to parent of y, this is a step y equal to parent of y, y is updated to this is y equal to parent of y yes parent of 13 is 13. Therefore you terminate the while loop and return 13 find 14 will return 13, that is because now you are at a y where y equal to parent of y alright. Therefore that single array is enough for you to find the name of the set containing x.

What is the complexity of this method? The complexity of this method is depth of x, how far x is away from the root. For example, in this case, you start from y and then applied once and applied again and the third time you found out the answer and you have output.

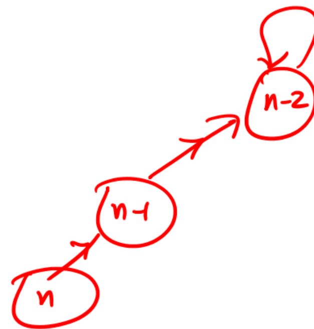
Therefore, 3 jumps it has to make to get the final answer. So if you have a tree, a very long tree and x is somewhere in the bottom the ancestor chasing will keep moving y up and up until y reaches the root. Therefore the complexity of find x is proportional to the depth of x depth means the distance from root to that node okay.

So here is an example that results in a very tall tree okay. So start with the partition 1, 2, n start with this partition.

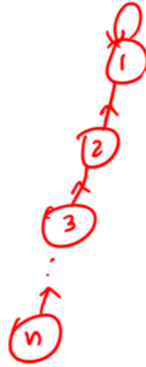
Hook the tree right n and n minus 1 union. So this is going to hook n to n minus 1 so you will have n here which is hooked to n minus 1.



Suppose the next operation is union n minus 1, n minus 2 this is going to hook this to n minus 2. So, I will have n minus 1 hooked to n minus 2, n minus 1 already has n to it.



So, finally union 2, 1, when you do this series of operation you are going to have 1 to which 2 is attached, to which 3 is attached and so on finally n is attached it is a very tall tree.



So now if you find n, this is x, this is y, y equal to parent of y, y equal to parent of y, in this way the while loop will execute n times finally it will reach the root and you return 1. You return 1, but the find is doing ancestor chasing along a very long path. Therefore the complexity of the find operation could be as high as order n, it is of order depth of x, union is 1, this is tree hooking whereas find which is done by ancestor chasing, this can happen on a very tall tree and that is the reason why the complexity could be order n okay.

Now we have seen the worst case example already, we are going to see a way to improve this, can we cut down the cost of find operation. There is a very simple trick to reduce the cost of the find operation, okay that is called weighted merge for union.

When you do the union operation, merge the small tree with the bigger tree, you have to take the union, you have a choice of merging A with B or B with A, the tree hooking can be done, A can be hooked to B or B can be hooked to A, it is always advantageous if you hook smaller tree with a larger tree. We hook the smaller tree with a larger tree, if you do that you will never have a tall tree any tree, any tree will have height only order log n. Therefore any find operation is not going to take more than log n steps because you start from some node, keep going up and up, but the whole tree is log n height. Therefore you are not going to go more than log n step for any find operation, there are no tall trees. That is the advantage of this union but the union can be found out by simple additional bookkeeping again in order one time.

All we need is we have to maintain the size of the set as an additional information because we want to merge the smaller with larger. So here is the way in which union is implemented. If size of A is less than or equal to size of B, hook the smaller tree with the larger tree, have this picture in mind A is a smaller tree B is a bigger tree, size of A is less than or equal to size of B, there is a self loop here but A will be hooked to B. If the size of B is smaller else means if B is smaller so if B is smaller but A is bigger hook B to A P of B is A okay. So after this the size must be updated size of A is size of A plus size of B here size of B is because this picture.

Now only B is there as a tree that will have size A and B added, that is all, very simple okay.

Union (A, B)

If size $(A) \leq$ size (B)

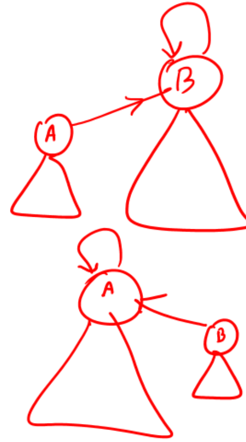
$p(A) = B$

size $(B) =$ size $(B) +$ size (A)

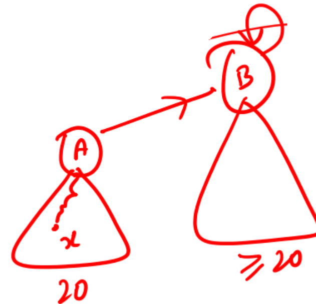
else

$p(B) = A$

size $(A) =$ size $(A) +$ size (B)



It has got either these 3 steps or these 3 steps done whichever step that you do alright it is order 1. Union complexity is still order 1 but the complexity of find is reduced from n to log n, dramatic reduction. For example if you are working with 1 million node, what is the effect of going from n to log n, from 1 million to go to 20 steps, what was taking 1 million steps you will take only 20 steps, that is the dramatic improvement if anything reducing from n to log n from a very large number to very small manageable number things will get reduced, very efficient method. There is no change in the find procedure the procedure is same you are doing the ancestor chasing but the number of times you will chase the ancestor is upper bound by log n because when you hook the tree the depth of certain node is increased maximum by 1, right the depth because you are attaching let us look at a picture.



This is A, consider this x how deep is x from A, I have drawn a picture. Suppose I attach A to B now how deep is x from the root, now B is the root, earlier A was the root now B is the root, the depth of x is increased by 1. Let us say from A to x there were 10 edges from B to x there will be 11 edges because this one more edge added therefore the depth is increased by 1. For each increase in depth by 1, the size of the set containing x doubles. Earlier let us say x was in a set with the cardinality 20, now the union, this will be greater than or equal to 20 because smaller trees are merged with larger trees so together this will be greater than or equal to 40.

So earlier x was in a set with 20 elements now x is in a set with 40 elements it is in a bigger set but the size doubles, so initially x was in a set with cardinality 1, by itself ,then by hooking it will be in a set with cardinally greater than or equal to 2, in second hooking it will go to a set with size at least 4. Then at least 8 you can see that the size of the set doubles so when will you hit n after log n doublings, after log n doubling you are going to hit n, therefore no tree is taller than log n units, that means every find operation is going to take only log n.

Let us recall the pseudo code that we have written implementing Kruskal's algorithm. This is the complete description of the Kruskal's algorithm in terms of the union find operations. Here you see that find operations are there and here you see union operations are performed here. How many find operations are done? for each edge okay let us assume that cardinality of V is n and cardinality of E, number of edges is m, m is the number of edges n is the number of vertices.

```
Process:
  While (NOT END OF L)
        Let e = (x, y)
       X = Find (x);
       Y = Find (y);
       If (X ≠ Y)
          Add e to A
          Union (X, Y)
      e = Next (e, L)
Output:
   T = (V, A)
      \\ T is a MST of G
```

$|V| = n, \ |E| = m$

How many find operations are done for each edge you are doing to find operations therefore 2m find operations. How many unions are performed, for each edge you add you perform one union operation. You know that A is going to have only n minus 1 edges, therefore there will be n minus 1 union operations. So the complexity of the Kruskal's algorithm is, once the sorted edge list is found you have to implement it and when you implementing this you are going to perform 2m find operations and n-1 union operations.

Cost of constructing L is you have to sort it is order m log m, which is same as order m log n, this is cost of building L, for constructing L this is the cost. So the total cost is of constructing L plus 2m find operation plus n minus 1 union operation, this is the total complexity.

Cost of constructing L + 2m Find + (n-1) union.

The total complexity is cost of constructing L, cost of constructing L is done for any implementation, therefore cost of constructing L is order m log n anyway. Suppose I use the name array, If I use name array data structure, each find operation is going to cost only 1 unit, so 2m find operations will cost 2m, each union is going to cost n, in find operation the complexity of the union in name array is n, so this will be n minus 1 times n, therefore the total complexity is order m log n plus m order m plus n square. This m is included in this so this can be written as order m log n plus n square.

$$O(m \log n + n^2)$$

This is the cost of implementing Kruskal's algorithm if you use name array data structure. Suppose I use inverted forest representation and perform union find, if I do that the complexity is going to be this is in-forest.

If I do in-forest, this part is order m log n is fixed, this is for sorting and getting L, each find operation is log n, therefore that will be 2m log n, each union is order 1 that will be n minus 1 times 1, therefore n is included in m log n, this is m log n, therefore this will be order m log n. So one algorithm has a complexity m log n right plus n square and another one is m log n, 2m find operations is going to cost 2m log n, this is the in forest representation okay. If we ignore the m log n part because m log n is a pre-processing cost, right m log n is a pre-processing cost, since m log n is a pre-processing cost if we ignore the processing cost, when you ignore m log n the processing cost for this would be order m plus n square. The processing cost this is ignored, only order m log n, in other words okay.

$$\text{In-forest} \quad O(m \log n) + 2m \log n + (n-1)\cdot 1 = O(m \log n) - O(m \log n)$$

Kruskal's algorithm performs n minus 1 union this is the pre-processing cost okay. This is pre-processing cost, ignoring pre-processing the actual execution of the algorithm the cost of execution of the algorithm depends on the data structure you use. You are using n minus 1 union and 2m find operation. The cost of n minus 1 union and 2m find operation in name array is n minus 1, each find operation is in-forest this will be order each union is order 1 but find is log n, that is m log n, which is better, is n square plus m is better or m log n is better, well it depends on whether the graph is sparse or dense sparse graph means number of edges m will be somewhat close to n, n log n kind of a thing, dense graph means m equal to n square right, so dense graph means m is n square. So for dense graph name array is n square plus n square that is n square, whereas m log n is n square log n asymptotically this will be more. Therefore for dense graph name array is n square in-forest array will be n

square log n that is the reason why for dense graph name array is a good choice of data structure.

For sparse graph let us say m is order n then this will be more like n log n whereas this will be n square. right that will be more asymptotically larger than this therefore for sparse graph in forest representation is a good choice. So here is a summary name array based implementation is better for dense graph, parent array based implementation or in-forest representation is better for sparse graph. Okay, this concludes our discussions on Kruskal's algorithm. Thank you.