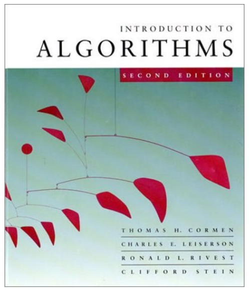
CS 445



Union/Find

Aka: Disjoint-set forest

Alon Efrat

Problem definition

Given: A set of atoms *S={1,2…n}*
E.g. each represents a commercial name of a drugs.
This set consists of different disjoint subsets.

Problem: suggest a data structures that efficiently supports two operations
•**Find(*i,j*)** – reports if the atom *i* and atom *j* belong to the same set.

•**Union(*i,j*)** – unify (merged) all elements of the set containing atom *i* with the set containing *j*.

•Example – on the board.

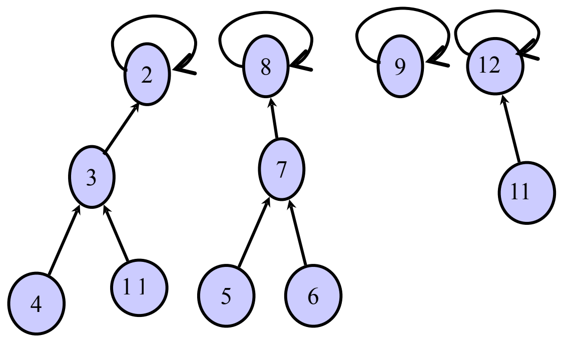
Naïve attempts

Idea: Each element "knows" to which set it belongs
(recall – each element belongs to exactly one set)

Bad idea: once two sets are merged, we need to scan all elements of one set and "tell" them that they belong to a different – lots of work if the set is large.

A Promising attempts

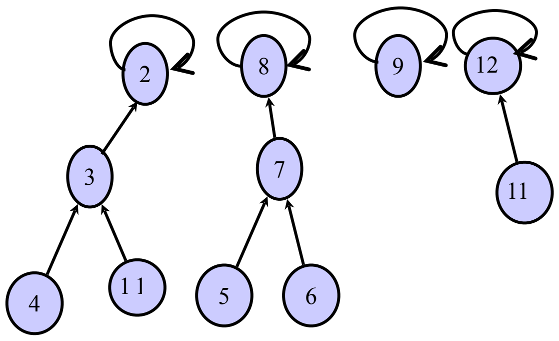
Idea: Store each set as a tree. Every node points to the parent
(different than standard trees)
Only the root "knows" the name of the set.



So the name of the set of *{2,3,4,1}* is **2.**
The name of the set of *{5,6,7,8}* is **8**.
The name of the set of *{9}* is **9**.
The name of the set of *{11,12}* is **12.**

A Promising attempts

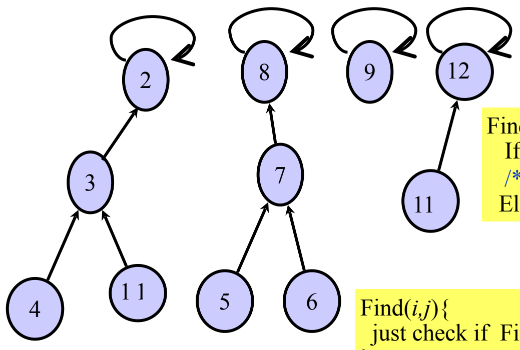
Idea: Store each set as a tree. Every node points to the parent (different than standard trees)
Only the root "knows" the name of the set.



So the name of the set of *{2,3,4,1}* is **2.**
The name of the set of *{5,6,7,8}* is **8**.
The name of the set of *{9}* is **9**.
The name of the set of *{11,12}* is **12.**

To find if two atoms belong to the same set, just check if they belong to same tree: Follow the parent pointers from each of them up all the way to the root. Check if this is the same root.

Disjoint sets forests - cont

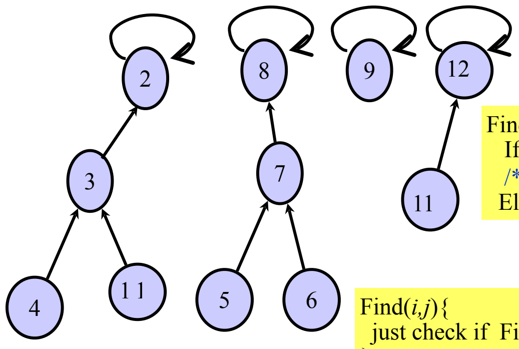


```
Find_root(j){
  If (p[j] ≠j) return Find_root(p[j]);
  /* p[j]  - points to the parent */
  Else return j ; }
```

```
Find(i,j){
  just check if  Find_root(i) == Find_root(j)
}
```

```
Union(i,j){
      Let r = Find_root(j)
      p[r]=i
}
```

Disjoint sets forests - cont



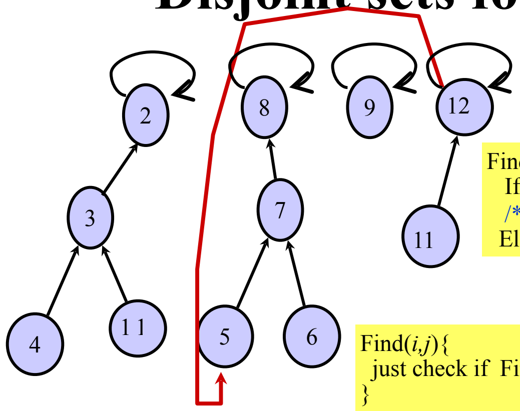
```
Find_root(j){
  If (p[j] ≠j) return Find_root(p[j]);
  /* p[j]  - points to the parent */
  Else return j ; }
```

```
Find(i,j){
  just check if  Find_root(i) == Find_root(j)
}
```

```
Union(i,j){
      Let r = Find_root(j)
      p[r]=i
}
```

Example – Union(5,11)

Disjoint sets forests - cont



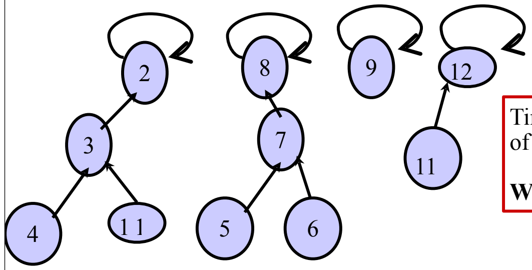
```
Find_root(j){
  If (p[j] ≠j) return Find_root(p[j]);
  /* p[j]  - points to the parent */
  Else return j ; }
```

```
Find(i,j){
  just check if  Find_root(i) == Find_root(j)
}
```

```
Union(i,j){
      Let r = Find_root(j)
      p[r]=i
}
```

Example – Union(5,11)

It this efficient?



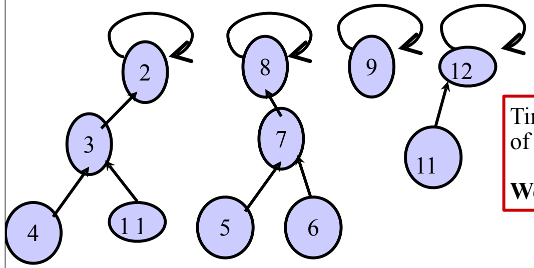
Time per operation depends on the height of the tree. Can be linear in the worst case.

We want short trees.

Improved union operation – version 1

```
Union(i,j){
        Let r = Find_root(j)
        p[r]=Find_root(i)
  /* rather than p[r]= i ;  */
}
```

It this efficient?



Time per operation depends on the height of the tree. Can be linear in the worst case.

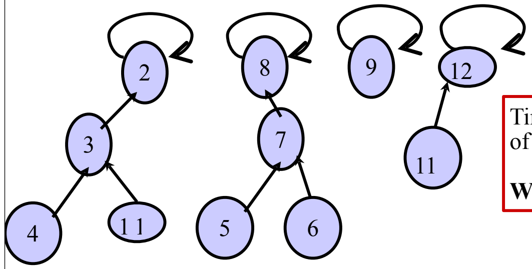
We want short trees.

Improved union operation – version 1

```
Union(i,j){
        Let r = Find_root(j)
        p[r]=Find_root(i)
  /* rather than p[r]= i ;  */
}
```

Note that we can also do

It this efficient?



Time per operation depends on the height of the tree. Can be linear in the worst case.

We want short trees.

Improved union operation – version 1

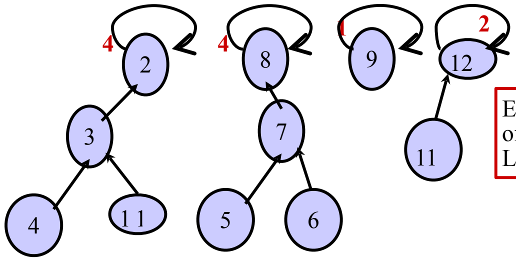
```
Union(i,j){
        Let r = Find_root(j)
        p[r]=Find_root(i)
  /* rather than p[r]= i ;  */
}
```

Note that we can also do

```
Union(i,j){
        Let r = Find_root(i)
        p[r]=Find_root(j)

}
```

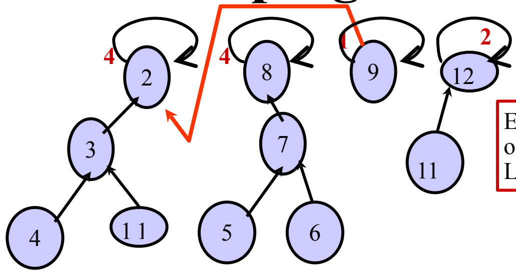
Keeping tracks of # nodes



Every **root** (only roots) stores the number of nodes in its tree
Let ***r.n*** denote this field in the root *r*.

```
Union(i,j){
  Let r1 = Find_root(i);  Let r2 = Find_root(j);
  /* connect the root of the small tree as a child of the root of       the larger tree */
  if (r1.n< r2.n )  {    p[r1]=r2 ;    r2.n += r1.n ; }
  else  {   p[r2]=r1 ;     r1.n += r2.n     }
}
```

Keeping tracks of # nodes



Every **root** (only roots) stores the number of nodes in its tree
Let ***r.n*** denote this field in the root *r.*

```
Union(i,j){
  Let r1 = Find_root(i);  Let r2 = Find_root(j);
  /* connect the root of the small tree as a child of the root of       the larger tree */
  if (r1.n< r2.n )  {    p[r1]=r2 ;    r2.n += r1.n ; }
  else  {   p[r2]=r1 ;     r1.n += r2.n     }
}
```

Example: Union(9,3)

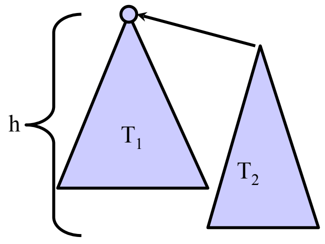
Proving bounds on the height

Assume we start from a forest where each node is a singleton
(a set of one element), and we perform a sequence of union operations.

Lemma: The height of every tree is $\leq \log_2 n$.   (n – number of atoms)

Proof: Show by induction that every tree of height ***h*** has $\geq 2^h$ nodes.

Assume true for every tree of height $h' < h$, and assume that after merging trees T_1, T_2, we obtained a tree of height exactly h.



T_2 has height exactly $h-1$, so it has $\geq 2^{h-1}$ nodes.

T_1 must have more nodes (why ?) so it also has $\geq 2^{h-1}$ nodes

Together they have $2^{h-1} + 2^{h-1} = 2^h$ nodes.

Further improvement: path compression

So far we know that every tree has height O(log n), so this bounds the time for each operation.

Path compression: during either union or find operation, we scan a sequence of nodes on our way from a node *j* to the root.

Idea: set the parent pointer of all these node to points to the root.

```
Find_root(j){
   If p[j] ≠ j  then p[j]=Find_root(p[j]);
   return p[j]
}
```

Make sense – but how fast is it ?

Thm: Any sequence of m U/F operations takes on a set of n atoms takes $O(m\,\alpha(n))$.

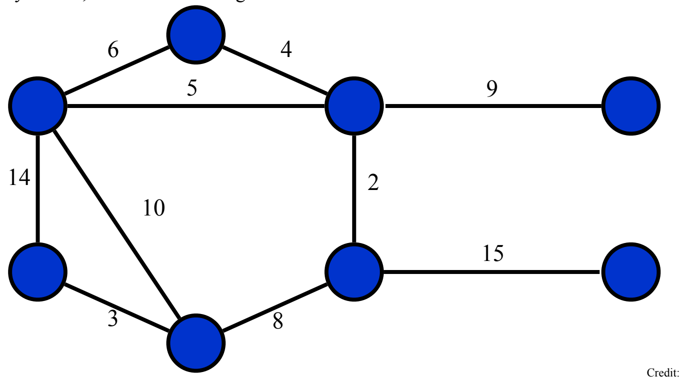
Here $\alpha(n)$ is the inverse function of Ackerman function, and is approaching infinity as n approaching infinity.

However, it does it very slowly.

$\alpha(n) < 4$ when $n < 10^{80}$.

Minimum Spanning Tree

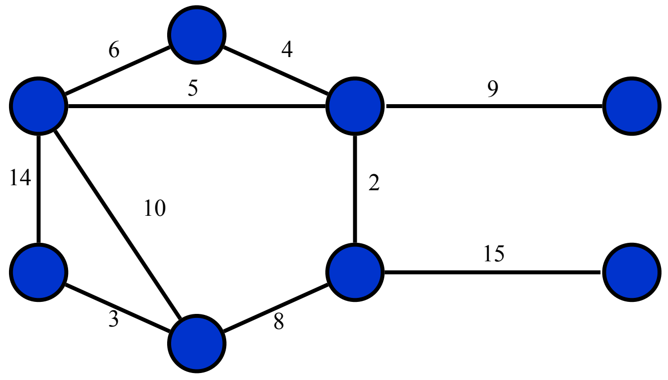
- Problem: given a connected, undirected, weighted graph $G(V, E)$. Each edge is assigned a positive cost (weight):
- We say that a set $F \subseteq E$ spans the graph if between every two vertices $s, t \in V$ there is a path (at least one) consists of edges of F.
- The goal is to find such a subset that minimizes the sum of costs of its edges. Obviously, it is a tree (if there is a cycle in F, then there is an edge that we can delete.



Credit: David Luebke

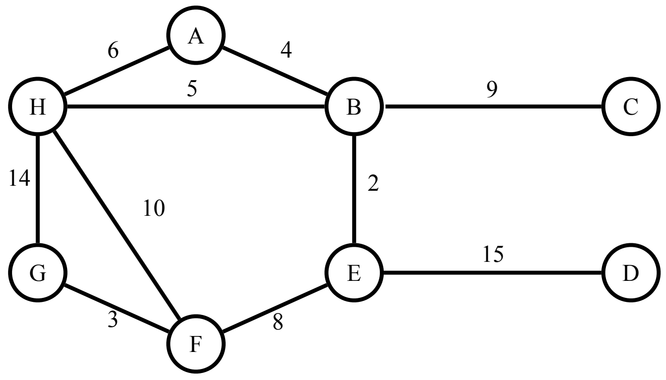
Minimum Spanning Tree

- Each edge is a possible road or a link that we need to buy. Want to buy enough edges to keep the network connected, but pay as little as possible.
- The resulting set of edges is called a *Minimum Spanning Tree* (**MST**).
- For simplicity of the analysis, lets assume that the MST is unique. Other we need to talk (though the algorithm does not care if there is another spanning tree with the same cost)



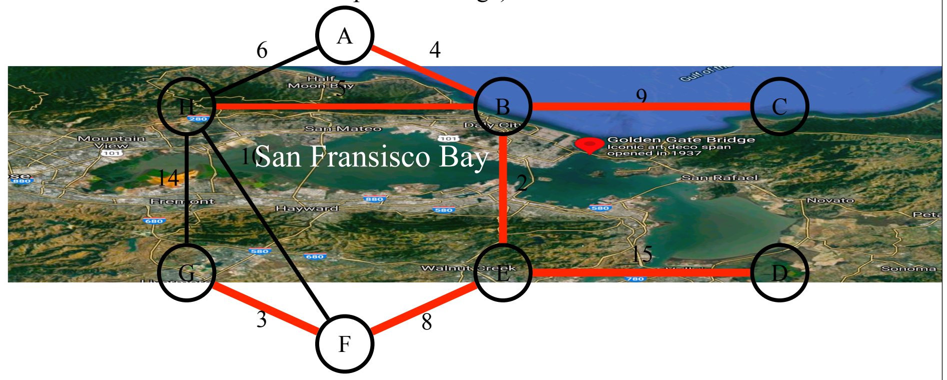
Minimum Spanning Tree

- Which edges form the minimum spanning tree (MST) of the below graph?

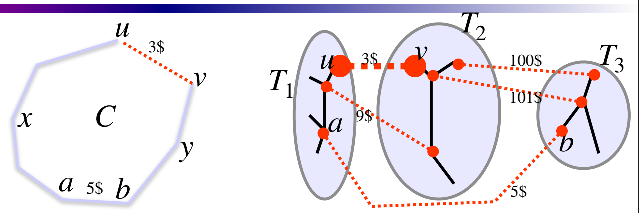


Minimum Spanning Tree

- Solution (heavy blue edges)
- Now, if we think about a river separating the city, and edges as potential bridges crossing it, then the **cheapest** bridge will be used.
- (if we don't use the cheapest bridge we either have now way to cross the city, we we need to use a more expensive bridge)



A Theorem about the structures of MST. Which edge is it safe to add?



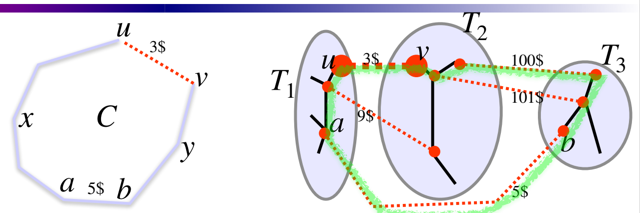
- Let T denote the MST(G).
- In a typical step of the algorithm, there are some edges of T that we committed to so far. The form a forest of subtrees.
- Assume that T_1, T_2 and T_3 are disjoint subtrees, and they are all included in T.
- Consider the set F all edges of E that connect two different subtrees. (in the figure, these are the red edges)
- Let $(u,v) \in F$ be the cheapest edge in F.

Claim: (u,v) is also in T

Proof: Assume that $(u,v) \notin T$. Since *MST(G)* is connected, there is a path π from $u \rightsquigarrow v$ in T. This path contain at least one edge of F (since u and v are in different subtrees). Lets call this edge (a,b). If we add (u,v) to π we obtain a cycle C. (in the example, this is the **green** cycle)

- Since (u,v) is the cheapest edge in F, it follows that $cost((u,v)) < cost((a,b))$.
- Lets generate a new tree T' from T, by adding (u,v) to T, and then remove (a,b).
- It is easy to see that the cost of T' is cheaper than the cost of T.
- It is a bit harder to see that T' is still connected. Prove this claim (**homework**). Lets prove it for the vertices of π.
- Assume $x, y \in \pi$ is a pair of vertices of $x, y \in \pi$. We can still use them in T', by using (u,v) instead of (a,b).
- So T' still spans G (covers all vertices and connected), and is cheaper than T. QED

A Theorem about the structures of MST. Which edge is it safe to add?



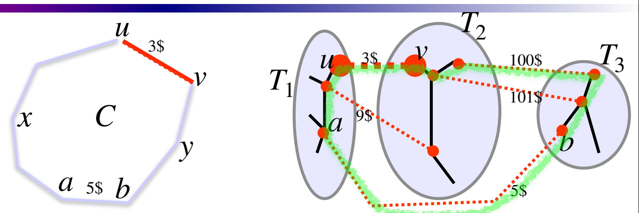
- Let T denote the MST(G).
- In a typical step of the algorithm, there are some edges of T that we committed to so far. The form a forest of subtrees.
- Assume that T_1, T_2 and T_3 are disjoint subtrees, and they are all included in T.
- Consider the set F all edges of E that connect two different subtrees. (in the figure, these are the red edges)
- Let $(u,v) \in F$ be the cheapest edge in F.

Claim: (u,v) is also in T

Proof: Assume that $(u,v) \notin T$. Since *MST(G)* is connected, there is a path π from $u \rightsquigarrow v$ in T. This path contain at least one edge of F (since u and v are in different subtrees). Lets call this edge (a,b). If we add (u,v) to π we obtain a cycle C. (in the example, this is the **green** cycle)

- Since (u,v) is the cheapest edge in F, it follows that $cost((u,v)) < cost((a,b))$.
- Lets generate a new tree T' from T, by adding (u,v) to T, and then remove (a,b).
- It is easy to see that the cost of T' is cheaper than the cost of T.
- It is a bit harder to see that T' is still connected. Prove this claim (**homework**). Lets prove it for the vertices of π.
- Assume $x, y \in \pi$ is a pair of vertices of $x, y \in \pi$. We can still use them in T', by using (u,v) instead of (a,b).
- So T' still spans G (covers all vertices and connected), and is cheaper than T. QED

A Theorem about the structures of MST. Which edge is it safe to add?



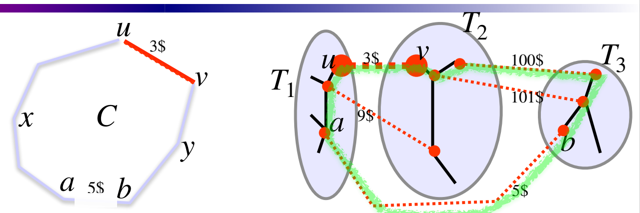
- Let T denote the MST(G).
- In a typical step of the algorithm, there are some edges of T that we committed to so far. The form a forest of subtrees.
- Assume that T_1, T_2 and T_3 are disjoint subtrees, and they are all included in T.
- Consider the set F all edges of E that connect two different subtrees. (in the figure, these are the red edges)
- Let $(u,v) \in F$ be the cheapest edge in F.

Claim: (u,v) is also in T

Proof: Assume that $(u,v) \notin T$. Since *MST(G)* is connected, there is a path π from $u \rightsquigarrow v$ in T. This path contain at least one edge of F (since u and v are in different subtrees). Lets call this edge (a,b). If we add (u,v) to π we obtain a cycle C. (in the example, this is the **green** cycle)

- Since (u,v) is the cheapest edge in F, it follows that $cost((u,v)) < cost((a,b))$.
- Lets generate a new tree T' from T, by adding (u,v) to T, and then remove (a,b).
- It is easy to see that the cost of T' is cheaper than the cost of T.
- It is a bit harder to see that T' is still connected. Prove this claim (**homework**). Lets prove it for the vertices of π.
- Assume $x, y \in \pi$ is a pair of vertices of $x, y \in \pi$. We can still use them in T', by using (u,v) instead of (a,b).
- So T' still spans G (covers all vertices and connected), and is cheaper than T. QED

A Theorem about the structures of MST. Which edge is it safe to add?



- Let T denote the MST(G).
- In a typical step of the algorithm, there are some edges of T that we committed to so far. The form a forest of subtrees.
- Assume that T_1, T_2 and T_3 are disjoint subtrees, and they are all included in T.
- Consider the set F all edges of E that connect two different subtrees. (in the figure, these are the red edges)
- Let $(u,v) \in F$ be the cheapest edge in F.

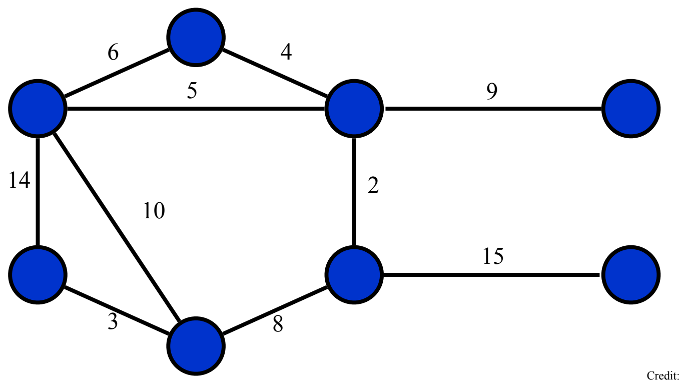
Claim: (u,v) is also in T

Proof: Assume that $(u,v) \notin T$. Since *MST(G)* is connected, there is a path π from $u \rightsquigarrow v$ in T. This path contain at least one edge of F (since u and v are in different subtrees). Lets call this edge (a,b). If we add (u,v) to π we obtain a cycle C. (in the example, this is the **green** cycle)

- Since (u,v) is the cheapest edge in F, it follows that $cost((u,v)) < cost((a,b))$.
- Lets generate a new tree T' from T, by adding (u,v) to T, and then remove (a,b).
- It is easy to see that the cost of T' is cheaper than the cost of T.
- It is a bit harder to see that T' is still connected. Prove this claim (**homework**). Lets prove it for the vertices of π.
- Assume $x, y \in \pi$ is a pair of vertices of $x, y \in \pi$. We can still use them in T', by using (u,v) instead of (a,b).
- So T' still spans G (covers all vertices and connected), and is cheaper than T. QED

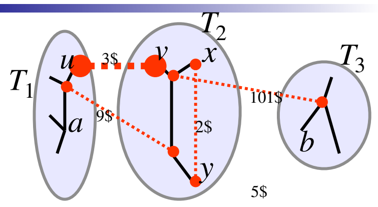
Application of the theorem. Kruskal Algorithm

- Kruskal Algorithm builds greedily the set F of the edge of MST(G).
- We start when F is empty. At each step, we add an edge to F.
- When we start, F is empty. Each vertex is a tiny subset.
- The Theorem implies that the first edge that we add to F is the cheapest edge of the graph



Credit: David Luebke

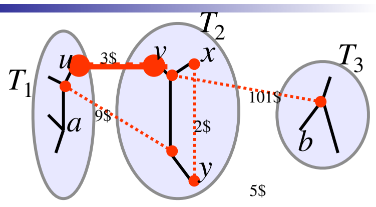
Kruskal Algorithm - cont



- In a general step of the algorithm, the edges of F form several subtrees $T_1, T_2, T_3 \ldots\ldots$ We need to add the cheapest edge, as long as it connects two different subtrees of F.
- Note that once (u, v) is added to F, the two subtrees are merged into one subtree.
- Considering a the cheapest edge (u, v) not used so far.
- We wonder if it should be added to F. If it connects two different subtrees - yes. If it connects two vertices of the same subtree - pass.
- Note that once (u, v) is added to F, the two subtrees are merged into one subtree.
- To answer this question, we will use the U/F data structure. The subsets are the vertices in each subtree.
- We will initialized it so each set contains a single vertex. $F = \varnothing$ (no edges in F yet)
- As the algorithm progress, each time that we add an edge (u, v), we perform **Union** of the vertices of the subtree T_1 and the vertices of T_2. This is easily done by the command *Union(u,v).*

- *To check if* (u, v) belongs to the same subtree or different subtrees, perform *Find(u,v)*
- To find the next cheapest edge, just sort them from cheapest to most expensive

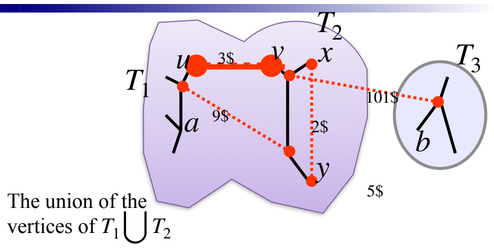
Kruskal Algorithm - cont



- In a general step of the algorithm, the edges of F form several subtrees $T_1, T_2, T_3 \ldots\ldots$ We need to add the cheapest edge, as long as it connects two different subtrees of F.
- Note that once (u, v) is added to F, the two subtrees are merged into one subtree.
- Considering a the cheapest edge (u, v) not used so far.
- We wonder if it should be added to F. If it connects two different subtrees - yes. If it connects two vertices of the same subtree - pass.
- Note that once (u, v) is added to F, the two subtrees are merged into one subtree.
- To answer this question, we will use the U/F data structure. The subsets are the vertices in each subtree.
- We will initialized it so each set contains a single vertex. $F = \varnothing$ (no edges in F yet)
- As the algorithm progress, each time that we add an edge (u, v), we perform **Union** of the vertices of the subtree T_1 and the vertices of T_2. This is easily done by the command *Union(u,v).*

- *To check if* (u, v) belongs to the same subtree or different subtrees, perform *Find(u,v)*
- To find the next cheapest edge, just sort them from cheapest to most expensive

Kruskal Algorithm - cont



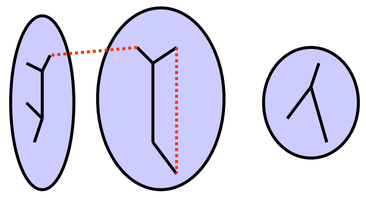
- In a general step of the algorithm, the edges of F form several subtrees $T_1, T_2, T_3 \ldots\ldots$ We need to add the cheapest edge, as long as it connects two different subtrees of F.
- Note that once (u, v) is added to F, the two subtrees are merged into one subtree.
- Considering a the cheapest edge (u, v) not used so far.
- We wonder if it should be added to F. If it connects two different subtrees - yes. If it connects two vertices of the same subtree - pass.
- Note that once (u, v) is added to F, the two subtrees are merged into one subtree.
- To answer this question, we will use the U/F data structure. The subsets are the vertices in each subtree.
- We will initialized it so each set contains a single vertex. $F = \varnothing$ (no edges in F yet)
- As the algorithm progress, each time that we add an edge (u, v), we perform **Union** of the vertices of the subtree T_1 and the vertices of T_2. This is easily done by the command *Union(u,v).*

- *To check if* (u, v) belongs to the same subtree or different subtrees, perform *Find(u,v)*
- To find the next cheapest edge, just sort them from cheapest to most expensive

Application: Kruskal algorithm

```
**Input**: Graph G(V,E).  Output: The set F of edges of the Minimal Spanning Tree for G.
**Init:** Set F=EmptySet. Init the U/F data structure where each vertex is its own set.
**Loop:** For each edge (u,v) (sorted from cheapest to most expensive)  {
        Perform Find(u,v).
            • If the answer is YES, then (u,v) connects vertices of the same
              subtree. Ignore it and continue
            • If the answer is NO, then {
                A.  Add (u,v)  to F  // it is an edge of MST(G).
                B.  Perform Union(u,v)  // merge the vertices of the two subtrees

            }
    }
```



If *E* is sorted, then the time is O(|*E*| α(|*E*|))

If we need to sort E, then naively it is O(|E| log |E|)