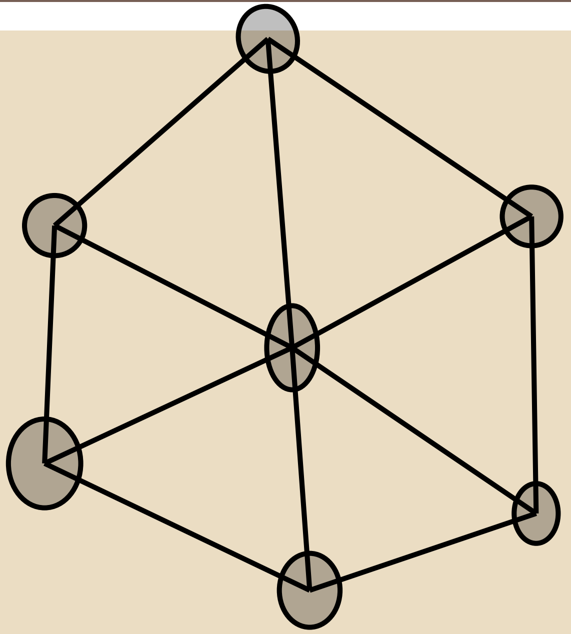
COL702: Advanced Data Structures and Algorithms

Thanks to Miles Jones, Russell Impagliazzo, and Sanjoy Dasgupta at UCSD for these slides.

SPANNING TREES

A spanning tree of an undirected graph $G = (V, E)$ is a subgraph $G' = (V, E')$ such that G' is a tree and all vertices in V are connected.

An output tree of DFS or BFS is a spanning tree.



EXAMPLE

Suppose you have a network of computers that were linked pairwise

Suppose each link has a positive maintenance cost.
Your job is to keep some links so that the cost of the network is minimized and the network stays connected.

Or each edge is a potential road, with the cost to build it, and you want to be able to drive to any location

THE MINIMIZED GRAPH IS A TREE

When is a connected undirected graph NOT a tree?

THE MINIMIZED GRAPH IS A TREE

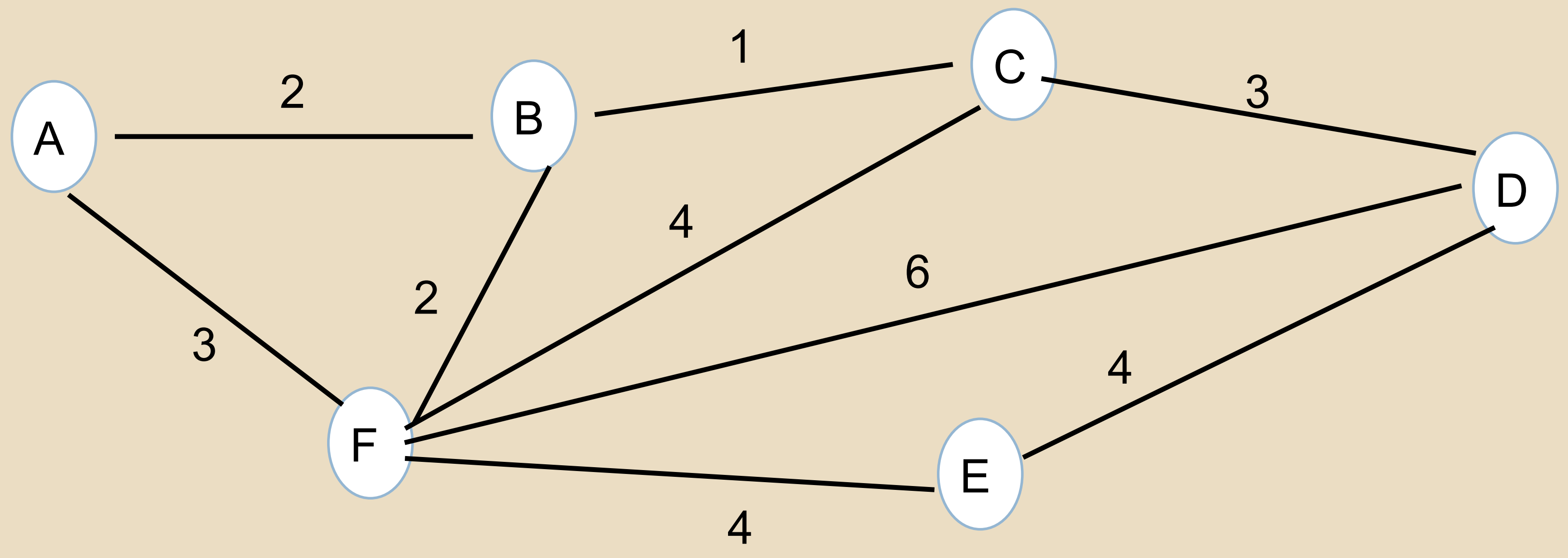
When is a connected undirected graph NOT a tree?

When it has a cycle.

If the subgraph has a cycle, dropping any edge makes a cheaper connected subgraph.

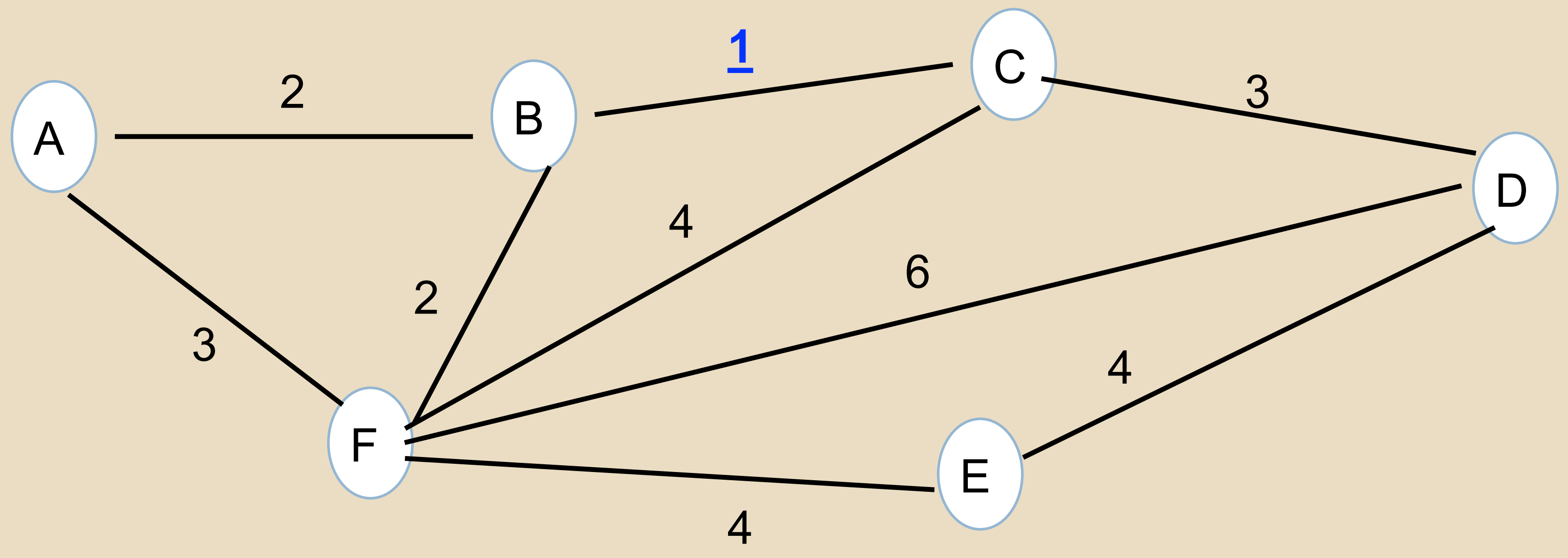
So the minimum cost connected sub-graph is always a tree

EXAMPLE



Greedy rule: which edge should we pick first?

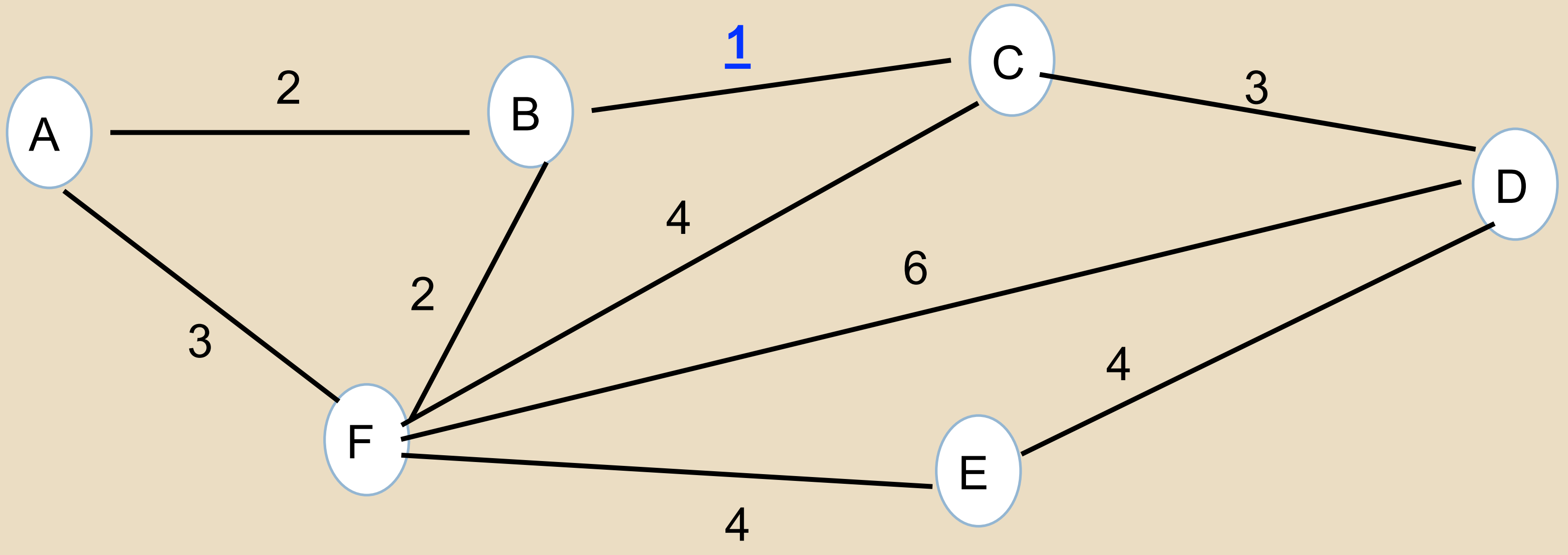
EXAMPLE



Greedy rule: which edge should we pick first?

Pick the smallest weight edge

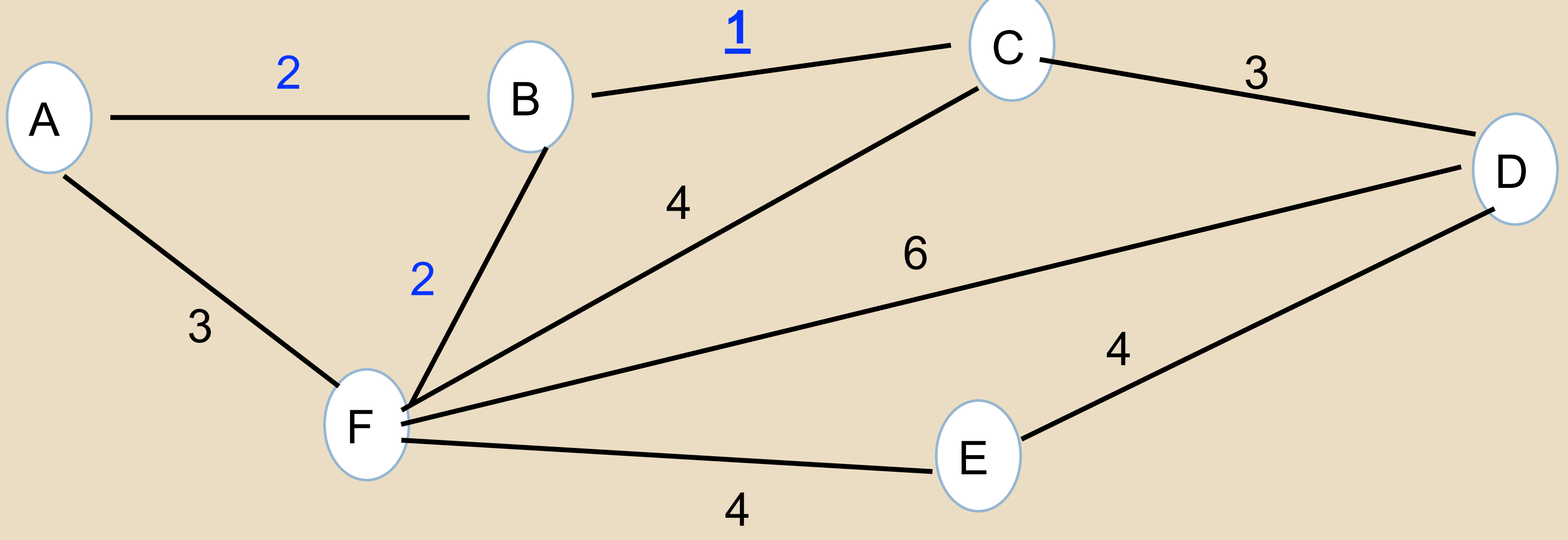
EXAMPLE



Greedy rule: which edge should we pick first?

Pick the smallest weight edge

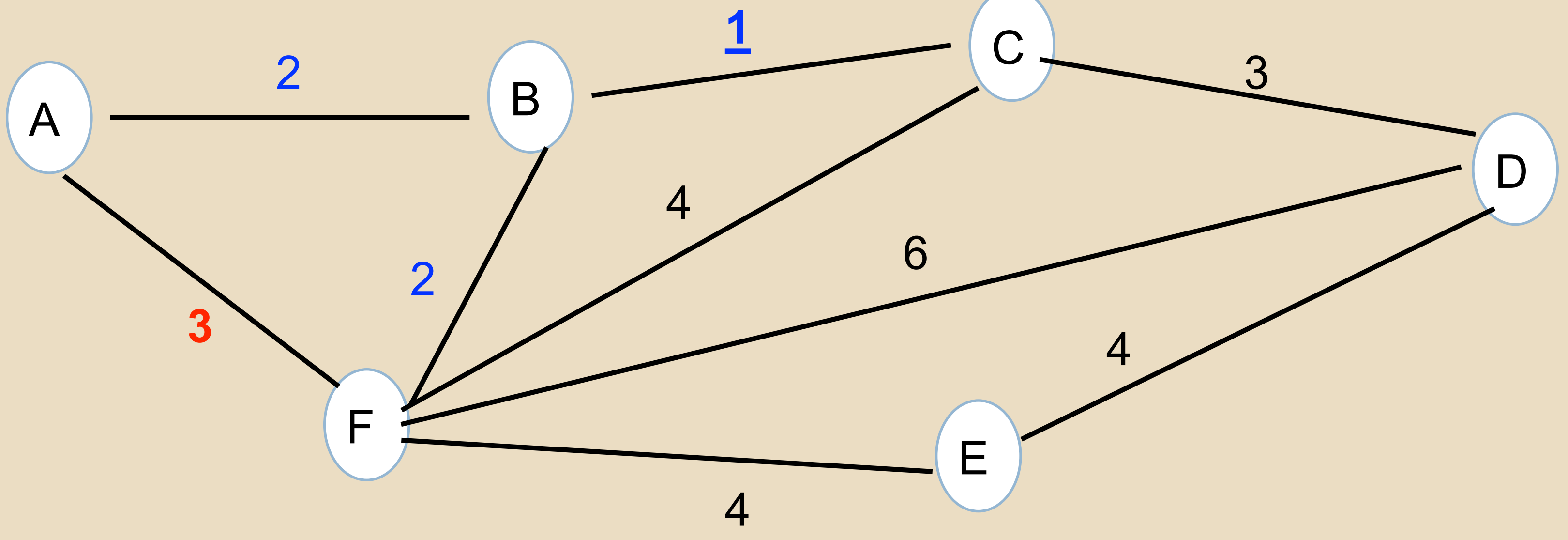
EXAMPLE



Greedy rule: which edge should we pick first?

Pick the smallest weight edge

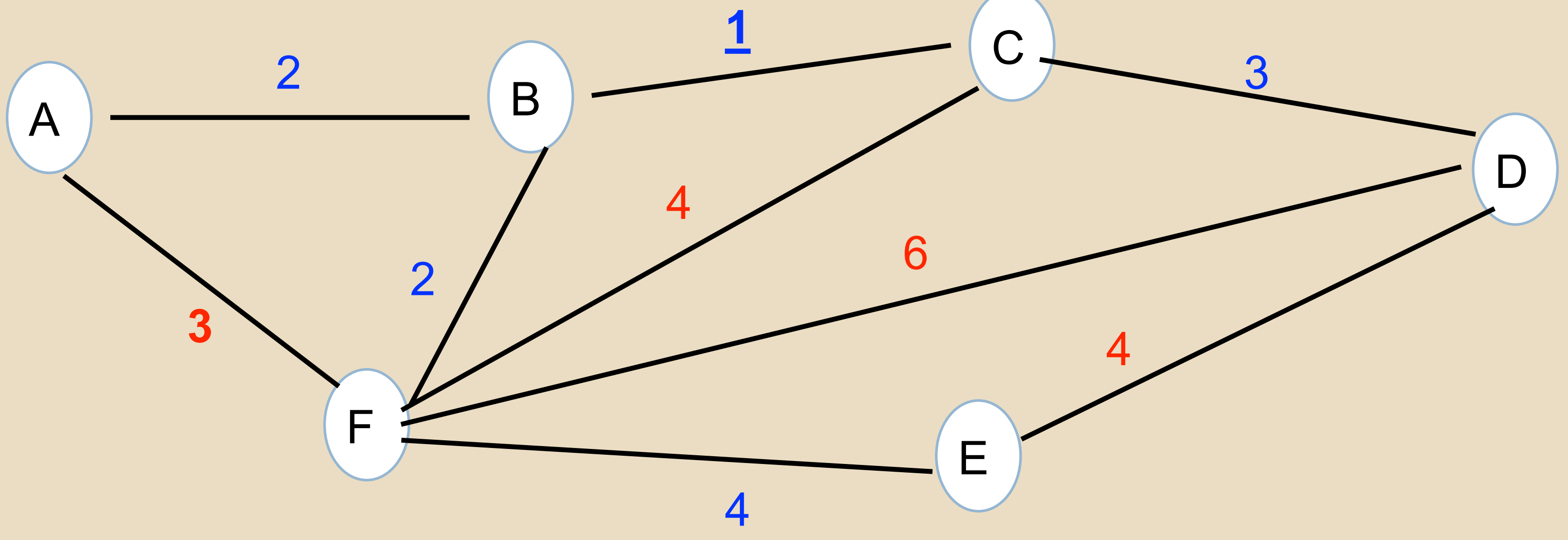
EXAMPLE



Greedy rule: which edge should we pick first?

Pick the smallest weight edge unless its vertices are already connected

EXAMPLE



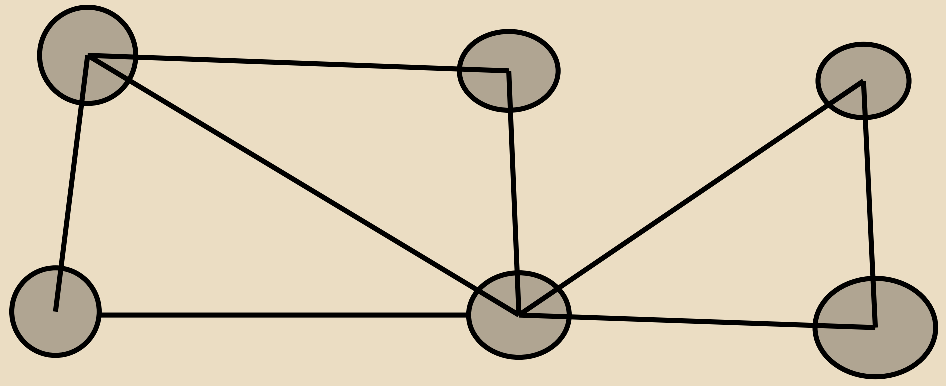
Greedy rule: which edge should we pick first?

Pick the smallest weight edge unless its vertices are already connected

KRUSKAL’S ALGORITHM FOR FINDING THE MINIMUM SPANNING TREE

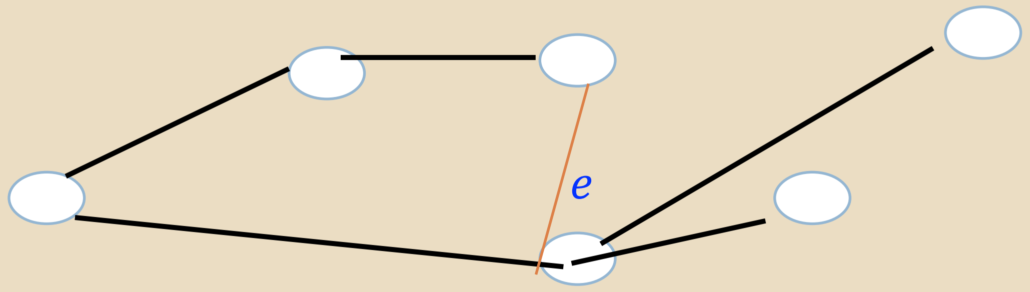
Start with a graph with only the vertices.

Repeatedly add the next lightest edge that does not form a cycle.



CORRECTNESS PROOF, MODIFY-THE-SOL'N

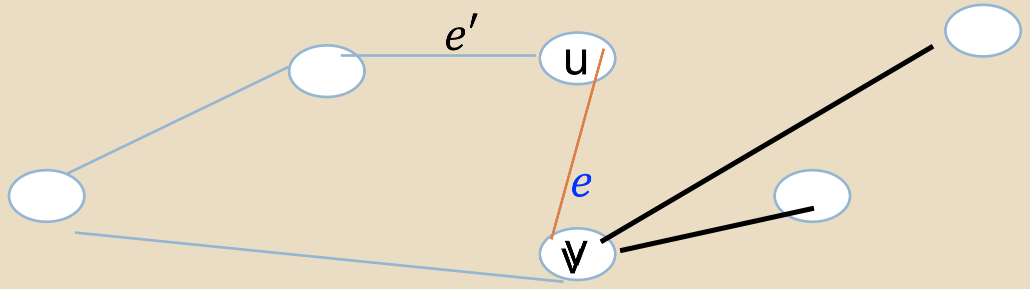
Let e be the smallest weight edge, OT a spanning tree that does not contain e. Then there is another spanning tree OT' that contains e, with $Cost(OT') \leq Cost(OT)$



CORRECTNESS PROOF, MODIFY-THE-SOL'N

Let e be the smallest weight edge, OT a spanning tree that does not contain e. Then there is another spanning tree OT' that contains e, with $Cost(OT') \leq Cost(OT)$

$e = \{u, v\}$, u must be connected to v in OT

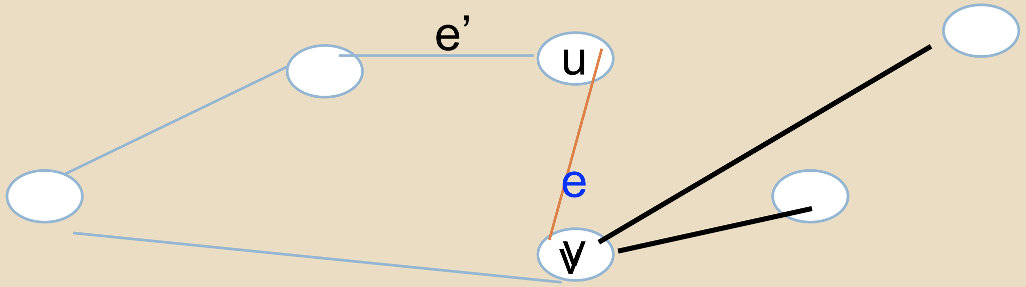


Let $\boldsymbol{p}$ be the path from u to v in OT, and let e' be any edge in that path

CORRECTNESS PROOF, DEFINE OT'

Let e be the smallest weight edge, OT a spanning tree that does not contain e. Then there is another spanning tree OT' that contains e, with $Cost(OT') \leq Cost(OT)$

$e = \{u, v\}$, u must be connected to v in OT



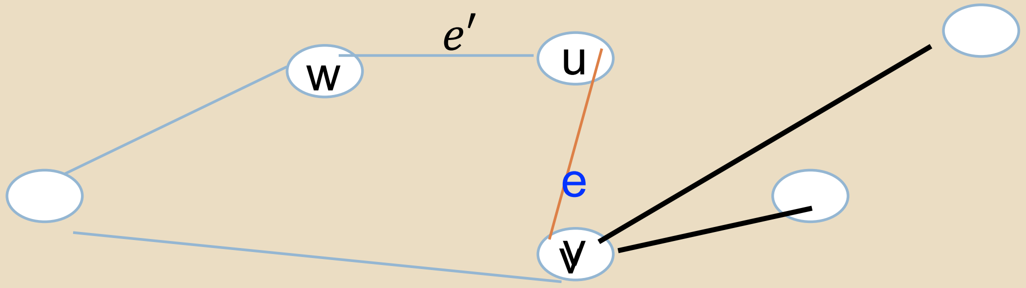
Let $OT' = OT + e - e'$

CORRECTNESS PROOF, OT' IS SPANNING TREE

Let $OT' = OT + e - e'$

Let $e' = \{u, w\}$. Then the path p is of the form e', p'

where p' is a path from w to v



To simulate e', we can take the following path in OT': e, p' in reverse

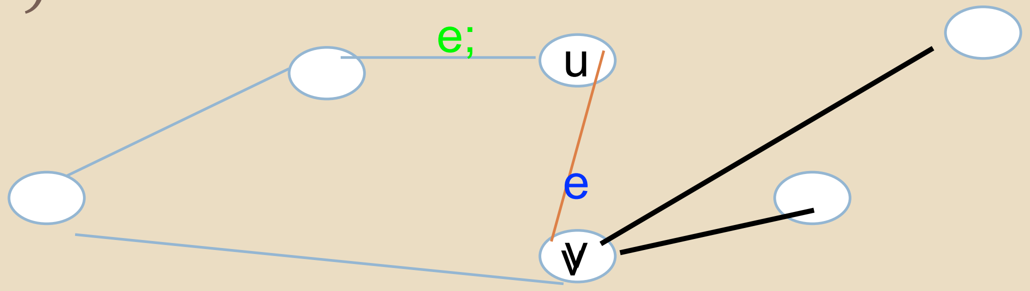
So since we can simulate any path in OT with a path in OT', and OT was connected, OT' is still connected

CORRECTNESS PROOF: COMPARE COSTS

Let e be the smallest weight edge, OT a spanning tree that does not contain e. Then there is another spanning tree OT' that contains e, with $Cost(OT') \leq Cost(OT)$

Let $OT' = OT + e - e'$

$w(e) \leq w(e')$



So $Cost(OT') = Cost(OT) - w(e') + w(e) \leq Cost(OT)$

INDUCTION STEP

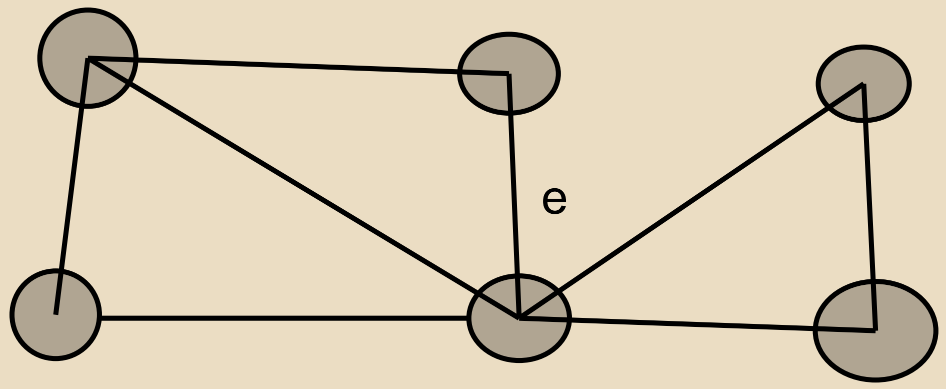
If G has at most two vertices, any solution is optimal

Assume Kruskal is optimal for any graph with $n-1$ vertices

Let e be the smallest weight edge

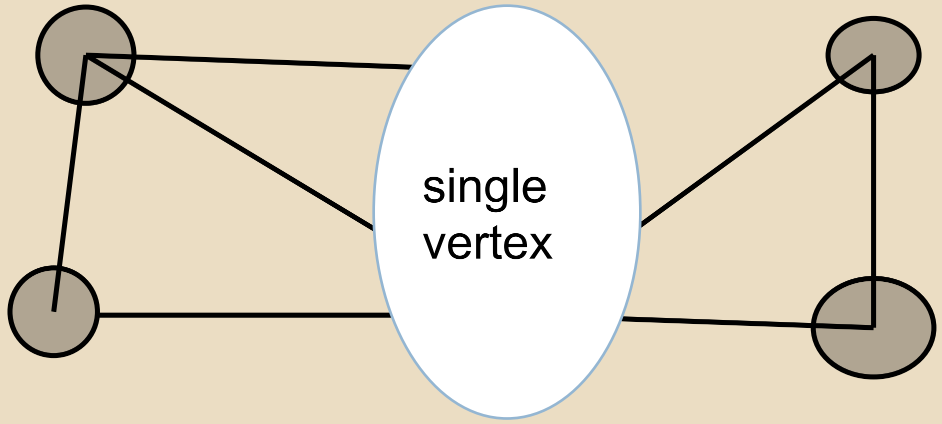
G': Contract the edge e in G, treating its two vertices as one vertex

CONTRACTION



CONTRACTION

Contracted graph is not necessarily simple



INDUCTION STEP

If G has at most two vertices, any solution is optimal

Assume Kruskal is optimal for any graph with $(n-1)$ vertices

Let e be the smallest weight edge

G': Contract the edge e in G, treating its two vertices as one vertex

Kruskal(G) = e + Kruskal(G')

OT' = e +some spanning tree in G'

Therefore,

Cost (Kruskal(G)) =

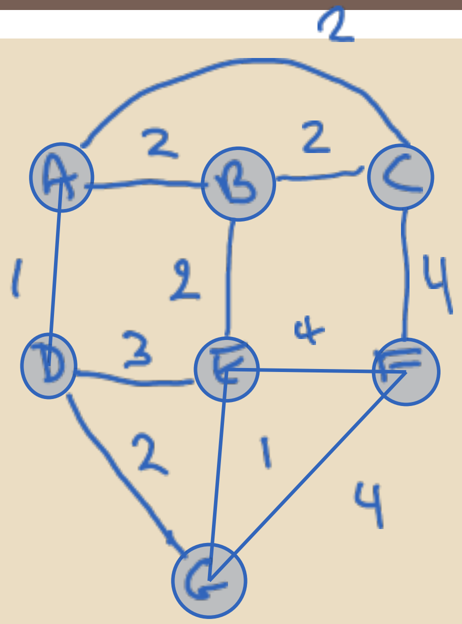
Cost(Kruskal(G')) + $w(e)$ ≤

Cost(spanning tree in G') + $w(e)$ =

Cost(OT')≤ Cost(OT)

EXAMPLE

AD: 1
EG: 1
BC: 2
DG: 2
BE: 2
AC: 2
DE: 3
EF: 4
GF: 4
CF: 4



HOW TO IMPLEMENT KRUSKAL’S

Sort edges by weight, go through from smallest to largest, and add if it does not create cycle with previously added edges

How do we tell if adding an edge will create a cycle?

Naive : DFS every time

Need to test for every edge, m times

DFS on a forest: only edges added to MST searched

Thus, each DFS is $O(n)$.

Total time $O(nm)$

NEXT TIME

How to implement Kruskal's algorithm faster

Data structures for disjoint sets

Leading into : Amortized analysis.

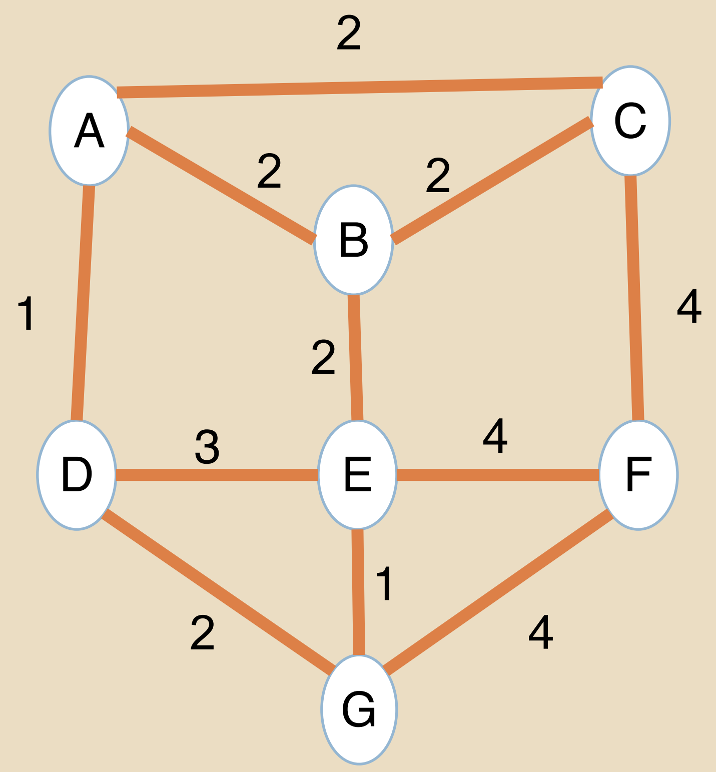
COL702: Advanced Data Structures and Algorithms

Thanks to Miles Jones, Russell Impagliazzo, and Sanjoy Dasgupta at UCSD for these slides.

KRUSKAL'S ALGORITHM FOR FINDING THE MINIMUM SPANNING TREE

Start with a graph with only the vertices.

Repeatedly add the next lightest edge that does not form a cycle.



HOW TO IMPLEMENT KRUSKAL’S

Start with an empty graph R. (Only vertices, no edges.)

Sort edges by weight from smallest to largest.

- For each edge e in sorted order:
  - If e does not create a cycle in R **then**
    - Add e to R
  - **otherwise**
    - do not add e to R

How do we tell if adding an edge will create a cycle?

LET'S ASK OUR STANDARD DS QUESTIONS

What kind of object do we need to keep track on in Kruskal's algorithm?

What do we need to know in one step?

How does the structure change in one step?

LET'S ASK OUR STANDARD DS QUESTIONS

What kind of object do we need to keep track on in Kruskal's algorithm? We need to keep track of the way the edges added to the MST divide up the vertices into components.

What do we need to know in one step? Are two vertices in the same component?

How does the structure change in one step? If we add an edge, it merges the two components into one.

DSDS MATCHES OUR REQUIREMENTS

DSDS stands for Disjoint Sets Data Structure.

What can it do?

Given a set of objects, DSDS manage partitioning the set into disjoint subsets.

It does the following operations:

- Makeset(S): puts each element of S into a set by itself.
- Find(u): it returns the name of the subset containing u.
- Union(u, v): it unions the set containing u with the set containing v.

KRUSKAL'S ALGORITHM USING DSDS

```
procedure Kruskal($G, w$)
▪ Input: undirected connected graph $G$ with edge weights $w$
▪ Output: a set of edges $X$ that defines an MST of $G$
▪ Makeset($V$)
▪ $X = \{\}$
▪ Sort the edges in $E$ in increasing order by weight.
▪ For all edges $(u, v) \in E$ until $X$ is a connected graph
    ▪ if find($u$) $\neq$ find($v$):
        ▪ Add edge $(u, v)$ to $X$
        ▪ Union($u, v$)
```

KRUSKAL'S ALGORITHM USING DSDS

```
procedure Kruskal($G, w$)
- Input: undirected graph $G$ with edge weights $w$
- Output: A set of edges $X$ that defines a minimum spanning tree
- for all $v \in V$
    - Makeset($v$)                                          |V|*(makeset)
- $X = \{\ \}$
- sort the set of edges E in increasing order by weight       sort(|E|)
- for all edges $(u, v) \in E$ until $|X| = |\mathrm{V}| - 1$              2*|E|*(find)
    - if find($u$) $\neq$ find($v$):
        - add $(u, v)$ to $X$
        - union($u, v$)                                         (|V|-1)*(union)
```

SUBROUTINES OF KRUSKAL'S

- makeset (u): This creates a set with one element, u
- find(u): finds the set to which u belongs
- union(u,v): merges the sets containing u and v

Runtime of Kruskal's:

$$|V| makeset + 2|E| find + (|V| - 1) union + sort(|E|)$$

DSDS VERSION 1 (ARRAY)

Keep an array Leader(u) indexed by element

In each array position, keep the leader of its set

Makeset(u):

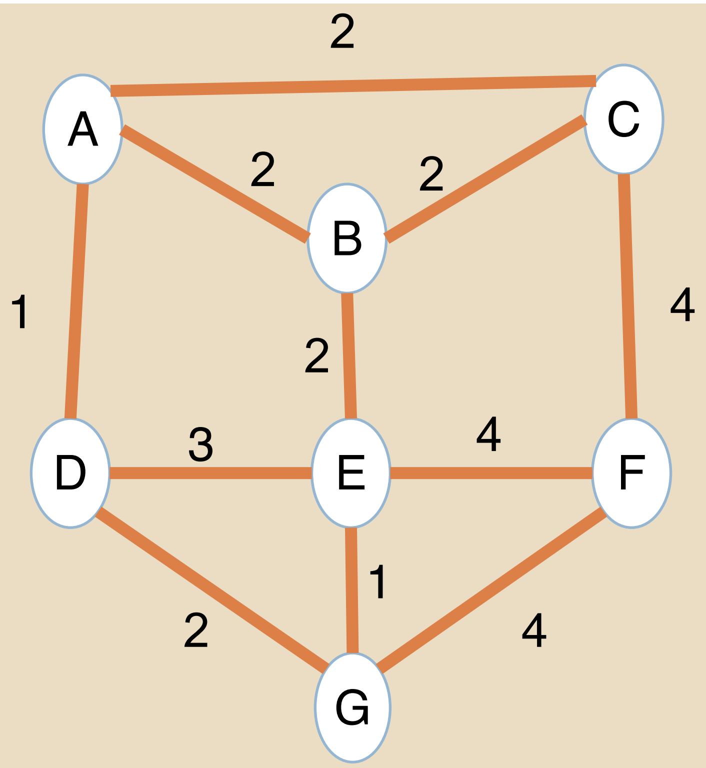
Find(u) :

union(u, v) :

Total time:

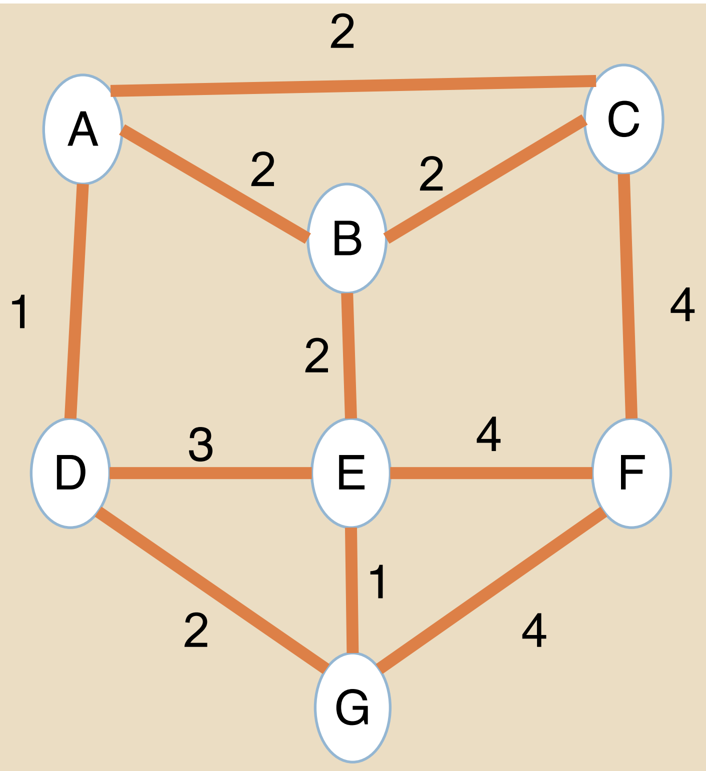
EXAMPLE DSDS VERSION 1 (ARRAY)

(A,D)=1
(E,G)=1
(A,B)=2
(A,C)=2
(B,C)=2
(B,E)=2
(D,G)=2
(D,E)=3
(E,F)=4
(F,G)=4
(C,F)=4



EXAMPLE DSDS VERSION 1 (ARRAY)

(A,D)=1
(E,G)=1
(A,B)=2
(A,C)=2
(B,C)=2
(B,E)=2
(D,G)=2
(D,E)=3
(E,F)=4
(F,G)=4
(C,F)=4



	A	B	C	D	E	F	G
	A	B	C	D	E	F	G
(A,D)=1							
(E,G)=1							
(A,B)=2							
(A,C)=2							
(B,C)=2							
(B,E)=2							
(D,G)=2							
(D,E)=3							
(E,F)=4							
(F,G)=4							
(C,F)=4							

EXAMPLE DSDS VERSION 1 (ARRAY)

(A,D)=1
(E,G)=1
(A,B)=2
(A,C)=2
(B,C)=2
(B,E)=2
(D,G)=2
(D,E)=3
(E,F)=4
(F,G)=4

```
A.   |. B   |.   C |.  D. | E.  |. F.  |G

A.   |. B   |.   C |.  A. | E.  |. F.  |G
A.   |. B   |.   C |.  A. | E.  |. F.  | E
B.   |. B   |.   C |.  B. | E.  |. F.  | E
C.   |. C   |.   C |.  C. | E. |. F.  | E

E.   |. E   |.   E |.  E. | E.|  F.  | E |.

E.   |. E   |.   E |.  E. | E. |. E.  | E
```

DSDS VERSION 1 (ARRAY)

Keep an array Leader(u) indexed by element

In each array position, keep the leader of its set

Makeset(u): $O(1)$

Find(u) : $O(1)$

union(u, v) : $O(|V|)$

Total time: $O(|E| \cdot 1 + |V| \cdot |V| + |E| \cdot \log |E|)$ =

$O\ (|V|^2 + |E| \log |E|)$. $\}$

VERSION 2: TREES

Each set is a rooted tree, with the vertices of the tree labelled with the elements of the set and the root the leader of the set

Only need to go up to leader, so just need parent pointer

Because we’re only going up, we don’t need to make it a binary tree or any other fixed fan-in.

VERSION 2A: DSDS OPERATIONS

Find: go up tree until we reach root

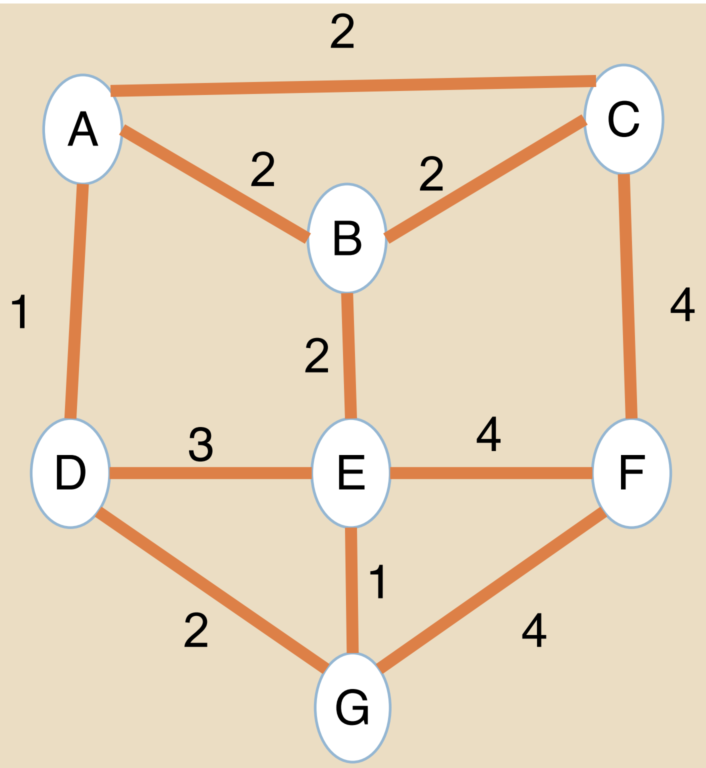
```
Find($v$).
    $L = v$.
    Until p(L) ==L, do: L=p(L)
```

Assume union is only done for distinct roots. We just make one root the child of the other.

```
Union(u, v)
    p(v)= u
```

EXAMPLE DSDS VERSION 1 (ARRAY)

(A,D)=1
(E,G)=1
(A,B)=2
(A,C)=2
(B,C)=2
(B,E)=2
(D,G)=2
(D,E)=3
(E,F)=4
(F,G)=4
(C,F)=4



(A,D)=1

(E,G)=1

(A,B)=2

(A,C)=2

(B,C)=2

⋮

EXAMPLE DSDS VERSION 2A (TREE)

A. |. B |. C |. D. | E. |. F. |G

(A,D)=1
(E,G)=1
(A,B)=2
(A,C)=2
(B,C)=2
(B,E)=2
(D,G)=2
(D,E)=3
(E,F)=4
(F,G)=4

A. |. B |. C |. A. | E. |. F. |G

A. |. B |. C |. A. | E. |. F. | E

B. |. B |. C |. A. | E. |. F. | E

B. |. C |. C |. A. | E. |. F. | E

B. |. C |. E |. A . | E.| F. | E |.

B. |. C |. E |. A . | E. |. E. | E

VERSION 2A: DSDS OPERATIONS

Find: go up tree until we reach root :

Time= depth of tree, could be O(|V|)

```
Find(v).
   L=v.
   Until p(L) ==L, do: L=p(L)
```

. We just make one root the child of the other. O(1)

```
Union(u, v)
   p(v)= u
```

DSDS VERSION 2A (TREE)

Keep an array parent(u) indexed by element

In each array position, keep parent pointer

Makeset(u): $O(1)$

Find(u) : $O(|V|)$

union(u, v) : $O(|1|)$

Total time: $O(|E| \cdot |V| + |V| \cdot 1 + |E| \cdot \log |E|) = O(|V||E|)$

Seems worse. But can we improve it?

DSDS VERSION 2A (TREE)

Find(u) : $O(|V|)$

union(u, v) : $O(|1|)$

Total time: O(|E|*|V|+|V|*1+ |E|log|E|) = O(|V||E|)

Seems worse. But can we improve it? Bottleneck: find when depth of tree gets large. Solution: To keep depth small, choose smaller depth to be child.

VERSION 2B(UNION-BY-RANK)

vertices of the trees are elements of a set and each vertex points to its parent that eventually points to the root.

The root points to itself.

The actual information in the data structure is stored in two arrays:

- p(v): the parent pointer (roots point to themselves)
- rank(v): the depth of the tree hanging from v.   (Note: in later versions, we’ll keep rank, but it will no longer be the exact depth, which will be more variable.) Initially, rank(v)=0

VERSION 2B: DSDS OPERATIONS

```
 Find(v).
   L=v.
   Until p(L) ==L, do: L=p(L)
.  We make smaller depth root the child of the other.
Union(u, v)
   If rank(u) > rank(v) Then: p(v)= u;
   If rank(u) < rank(v). Then: p(u)=v
   If rank(u)=rank(v). Then p(v)=u, rank(u)++
```

LEMMA:

If we use union-by-rank, the depth of the trees is at most l$og(|V|)$.

Proof: We show as loop invariant, that if for the leader of a set u, rank(u)=r, the tree rooted at u has size at least 2^r

True at start, each rank(u)=0, tree size is $1 = 2^0$.

Only could change with union operation. If roots are different ranks, rank doesn't change, set size increases.

If roots have same rank, rank increases by 1, set sizes add.

If we merge two sets of rank r, each had at least 2^r elements, so merged set has at least $2^{\{r+1\}}$ elements, and new rank is r+1

LEMMA:

If we use union-by-rank, the depth of the trees is at most log(|V|).

Proof: Invariant: if for the leader of a set u, rank(u)=r, the tree rooted at u has size at least 2^r. $Therefore\ r \leq \log|V|$

Second invariant: rank(u) is the depth of the tree at u

So depth is at most log |V|.

VERSION 2B: DSDS OPERATIONS

```
Find(v).  Time = O(depth ) = O(log |V|)
  L=v.
  Until p(L) ==L, do: L=p(L)
. We make smaller depth root the child of the other.
Union(u, v). Still O(1) time
  If rank(u) > rank(v) Then: p(v)= u;
  If rank(u) < rank(v). Then: p(u)=v
  If rank(u)=rank(v). Then p(v)=u, rank(u)++
```

DSDS VERSION 2B (TREE)

Find(u) : O(log |V|)

union(u,v) : O(|1|).

Total time: O(|E|*log |V|+|V|*1+ |E|log|E|) = O(|E| log |V|)

Rest of algorithm matches sort time!

SHOULD WE TRY FOR BETTER?

Why continue? Can’t improve since bottleneck is sorting.

Many times, sorting can be done in linear time, e.g., when values are small, can use counting or radix sort

Many times, inputs come pre-sorted

Because we want to optimize DSDS for other uses as well

Because it’s fun (for me, at least)

PATH COMPRESSION

We can improve the runtime of find and union by making the height of the trees shorter.

How do we do that?

every time we call find, we do some housekeeping by moving up every vertex.

PATH COMPRESSION

new find function

```
function find(x)
  ▪ if $x \neq p(x)$ then
        $\mathrm{p}(x) \coloneqq find\big(p(x)\big)$
  ▪ return $\mathrm{p}(x)$
```

EXAMPLE DSDS VERSION 2C (TREE)

```
(A,D)=1
(E,G)=1
(A,B)=2
(A,C)=2
(B,C)=2
(B,E)=2
(D,G)=2
(D,E)=3
(E,F)=4
(F,G)=4
```

```
   A.  |. B  |.  C |.  D. | E.  |. F.  |G

   A.  |. B  |.  C |.  A. | E.  |. F.  |G  Rank(A)=1
   A.  |. B  |.  C |.  A. | E.  |. F.  | E   Rank(E) =1
   A.  |. A  |.  C |.  A. | E.  |. F.  | E
   A.  |. A  |.  A |.  A. | E. |. F.  | E

A.  |. A  |.  A |.  A . | A.|  F.  | E |.        Rank(A)=2

   A.  |. A  |.  A |.  A . | A.|  F.  | A |.

   A.  |. A  |.  A |.  A . | A  |. A.  | A
```

FIND (PATH COMPRESSION)

whenever you call find on a vertex v, it points v and all of its ancestors to the root.

Seems like a good idea, but how much difference could it make?

Since worst-case find could be first find, same worst-case time as before.

FIND (PATH COMPRESSION) (RANKS)

The ranks do not necessarily represent the height of the graph anymore and so will this cause problems?

AMORTIZED ANALYSIS

Amortized analysis: Bound total time of m operations, rather than worst-case time for single operation * m

Intuition: Fast operations make things worse in the future, but slow operations make things better in the future. (Merging might build up the height of the tree, but finding a deep vertex shrinks the average heights.).