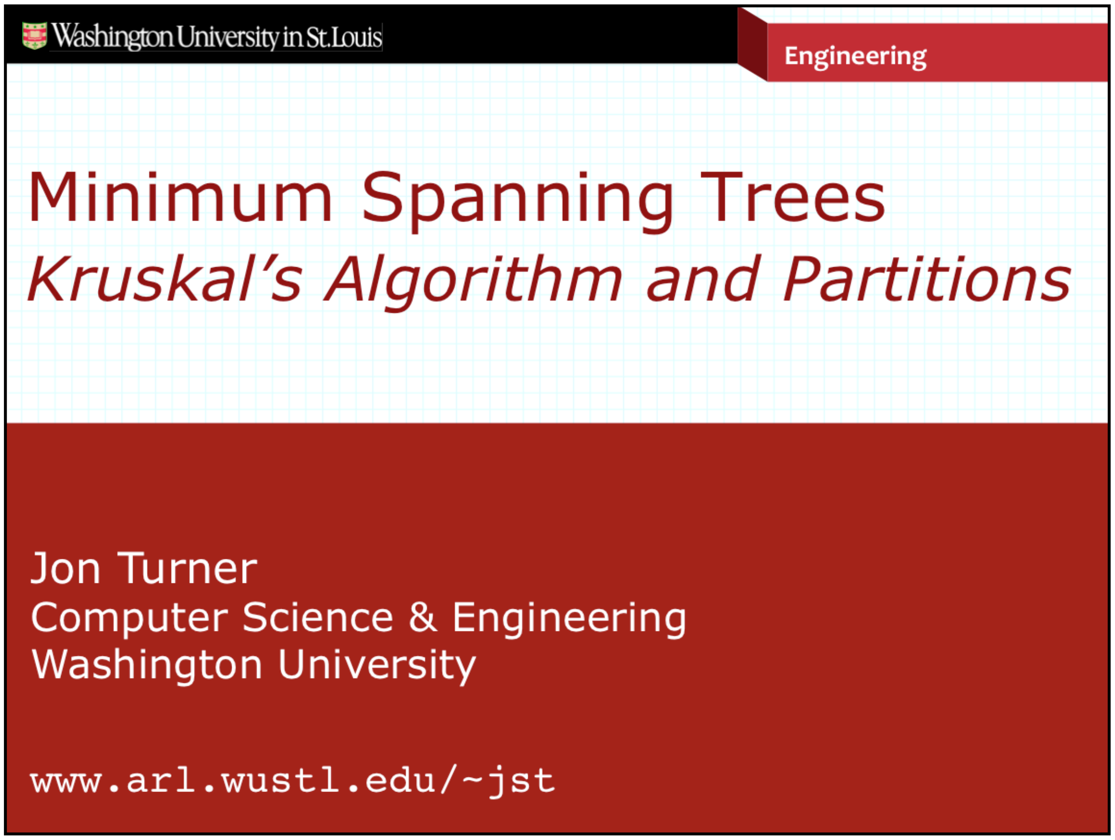


Washington University in St. Louis

Engineering

Minimum Spanning Trees

Kruskal’s Algorithm and Partitions

Jon Turner
Computer Science & Engineering
Washington University

www.arl.wustl.edu/~jst

Kruskal's Algorithm

- Kruskal's algorithm applies following rule to the edges in non-decreasing order of cost
  - » *Coloring Rule 1*: if the current edge *e* has both ends in the same blue tree, color it red; otherwise color it blue
- The algorithm can also be expressed as

```
procedure minspantree(graph  $G=(V,E)$, modifies set $blue$)
  vertex $u,v$; set $edges$;
  $blue := \{\}$; $edges := E$;
  Sort $edges$ by cost;
  for $\{u,v\} \in edges \Rightarrow$
     if $u$ and $v$ are in different blue trees $\Rightarrow$
        $blue := blue \cup \{u,v\}$;
     fi;
  rof;   // edges not added to blue are implicitly red
end;
```

Partition Data Structure

- To make Kruskal's algorithm fast, need a fast way to determine if two vertices are in same blue tree
  - » use a data structure that maintains a *partition* on set of all vertices, with a separate set for the vertices in each blue tree
  - » requires operations that allow us to merge two sets and to determine whether or not two vertices are in the same set
- Operations on partition data structure
  - » *partition*(*S*): create a partition on the set *S* with each element of *S* forming a separate subset
  - » *find*(*x*): return the *canonical element* of the set containing *x*
  - » *link*(*x*,*y*): merge the two sets with canonical elements *x* and *y*
    - original sets replaced with new set; returns new canonical element
- Efficient, easy to implement and has many applications
  - » often referred to as *union-find* data structure

Using Partitions in Kruskal's Algorithm

```
procedure minspantree(graph $G = (V,E)$, modifies set $blue$)
  vertex $u,v$; set $edges$; partition($V$ );
  $blue := \{\}$; $edges := E$;
  Sort $edges$ by cost;
  for $\{u,v\} \in edges \Rightarrow$
       if find($u$) $\neq$ find($v$) $\Rightarrow$
          link(find($u$),find($v$));  $blue := blue \cup \{u,v\}$
       fi;
  rof;
end;
```

- Sorting can be done in $O(m \log m) = O(m \log n)$ time
- Remaining time determined by the partition operations
  - » $n-1$ *links* and at most $4m$ *finds*
  - » these operations can be done in $O(m \log n)$ time
- So, overall time is $O(m \log n)$

C++ Version

```
void kruskal(Wgraph& wg, Wgraph& mstree) {
   edge e, e1; vertex u,v,cu,cv; weight w; int i = 0;
   Partition vsets(wg.n());
   edge *elist = new edge[wg.m()+1];
   for (e = wg.first(); e != 0; e = wg.next(e))
      elist[i++] = e;
   sortEdges(elist,wg);
   for (e1 = 1; e1 <= wg.m(); e1++) {
      e = elist[e1];
      u = wg.left(e); v = wg.right(e);
      w = wg.weight(e);
      cu = vsets.find(u); cv = vsets.find(v);
      if (cu != cv) {
         vsets.link(cu,cv);
         e = mstree.join(u,v); mstree.setWeight(e,w);
} } }
```

so, *wg*.*weight*(elist[*i*]) ≤*wg*.*weight*(elist[*i*+1])

e is *i*-th "lightest" edge

Implementing Partition

- Represent each set as a tree
  - » each tree node contains a set element *x* and a pointer to its parent *p*(*x*) in the tree; the root points to itself
- Link operation limits depth of trees using auxiliary variable *rank*(*x*) for each node *x*

```
procedure partition(set $S$)
    for $x \in S \Rightarrow p(x) := x$; $rank(x) := 0$; rof; end;
int function find(integer $x$);
    do $p(x) \neq x \Rightarrow x := p(x)$ od; return $x$;
end;
int function link(integer $x,y$);
    if $rank(x) > rank(y) \Rightarrow p(y) := x$; return $x$;
     | $rank(x) = rank(y) \Rightarrow rank(y) := rank(x) + 1$;
    fi;
    $p(x) := y$; return $y$;
end;
```

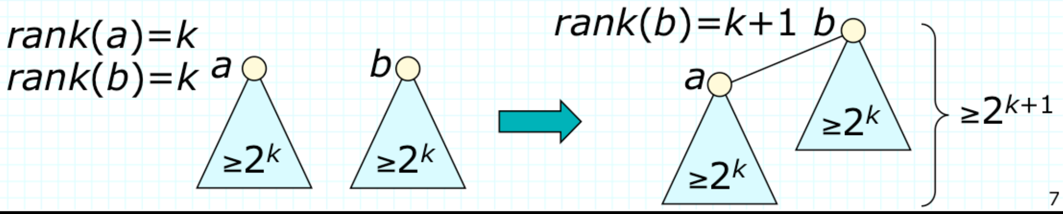
Basic Analysis of Partition

- **Lemma 2.1**. If x is any node, $rank(x) \le rank(p(x))$ with inequality strict if $p(x) \neq x$; $rank(x)=0$ initially and increases with time until $p(x)$ is assigned value other than x; after that, $rank(x)$ does not change; $rank(p(x))$ is nondecreasing function of time

  Proof. By induction on number of *find* and *link* ops ■

- **Lemma 2.2**. Number of nodes in tree with root x is $> 2^{rank(x)}$

  Proof. By induction on number of *link* operations ■



- **Lemma 2.3**. For any integer $k \geq 0$, the # of nodes with rank k is at most $n/2^k$; so, every node has rank $\leq \lg n$

  Proof. Suppose that when a node x is assigned a rank of k, all nodes in the tree with root x are labeled (x,k)

  By Lemma 2.2, at least 2^k nodes are labeled when x gets rank k

  If root of tree containing x changes, new root must have rank at least $k+1$, so no node is labeled more than once with a label of the form (y,k) for any particular value of k

  Since there are n nodes, there are $\leq n$ labels of form (y,k) for any particular k and there are at least 2^k for each node of rank k

  Hence, at most $n/2^k$ nodes can be assigned a rank of k. ■
- Running time
  - » initialization takes $O(n)$ time and each *link* operation takes $O(1)$
  - » by Lemmas 2.1–2.3, the height of any tree is $\leq \lg n$; so *find* takes $O(\log n)$ time
  - » so any sequence of m ops takes at most $O(m \log n)$ time

Speeding Up Partition

- Simple optimization makes partition significantly faster

```
int function find(integer $x$);
    integer $r$ ; $r := x$;
    do $p(r) \neq r \Rightarrow r := p(r)$; od;
    do $p(x) \neq r \Rightarrow x, p(x) := p(x), r$; od;
    return $r$;
end;
```

first pass finds root
second pass re-directs
parent pointers

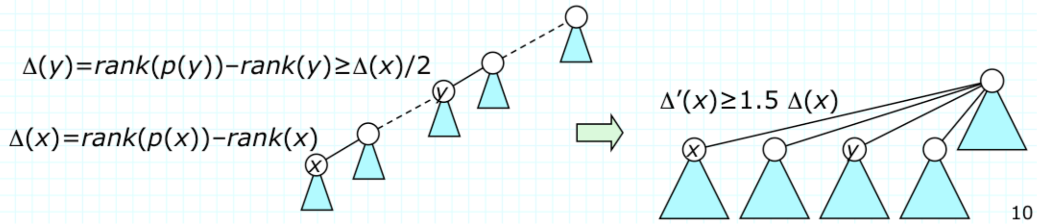
- Called *path compression* – speeds up later *finds.*
- Note: earlier lemmas remain true when path compression is used
- Recursive version (used in analysis)

```
int function find(integer $x$);
    if $x \neq p(x) \Rightarrow p(x) :=$ find($p(x)$) fi;
    return $p(x)$;
end;
```

each recursive call referred to as a "*find step*"

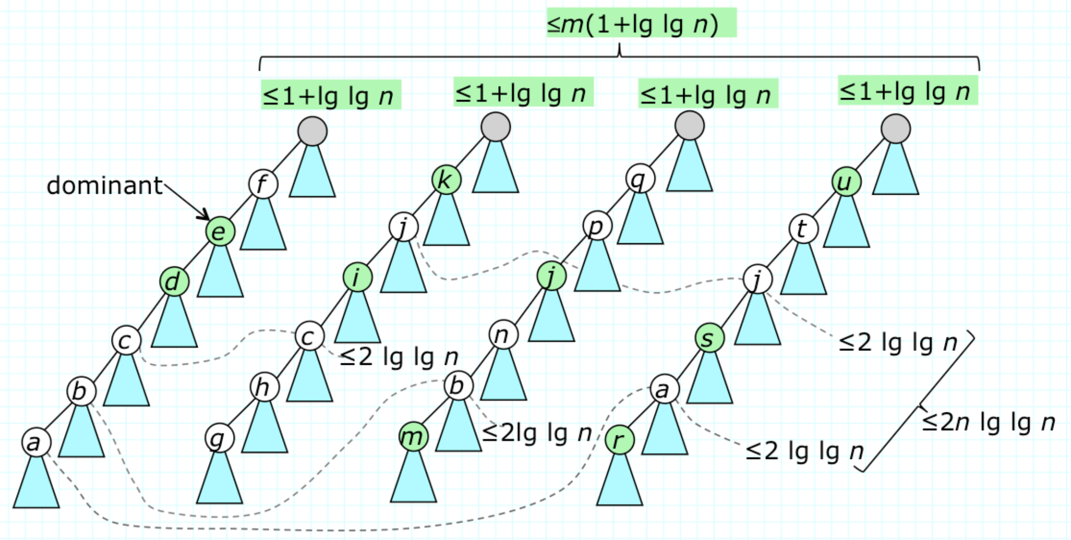
Bounding Number of Find Steps

- For a node x, let $\Delta(x)=rank(p(x))-rank(x)$
  - » $\Delta(x)=0$ initially and may then increase but is always $\leq \lg n$
- Call x *dominant* if $\Delta(x)>2\Delta(y)$ for all ancestors y of x
  - » there are at most $1+\lg\lg n$ dominant nodes along a "find path"
  - » so, total number of find steps at dominant nodes is $O(m \lg\lg n)$
- If x not dominant, it has ancestor y, with $\Delta(y)\geq\Delta(x)/2$
  - » a *find* involving x increases $\Delta(x)$ by a factor of at least 1.5
  - » this can happen to x at most 2 lg lg n times
  - » so $O(n \lg\lg n)$ find steps at nodes that are not dominant



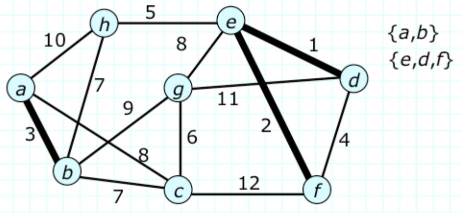
$\Delta(y)=rank(p(y))-rank(y)\geq\Delta(x)/2$
$\Delta(x)=rank(p(x))-rank(x)$
y
x
$\Delta'(x)\geq 1.5\ \Delta(x)$
x
y

Understanding Partition Analysis

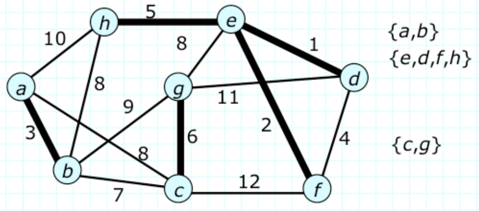


Exercises

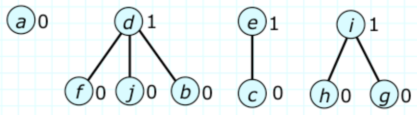
1. The figure below shows an intermediate state in Kruskal's algorithm. The tree edges are shown in bold and the non-singleton sets in the partition data structure are listed as sets.



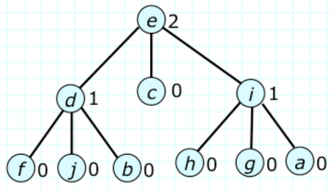
Show the state of the algorithm after the next three edges are considered for inclusion by the algorithm.



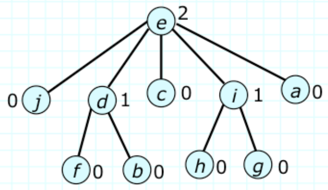
2. The figure below shows an instance of the partition data structure.



Show the data structure after the following operations are performed: *link*(*i*,*a*), *link*(*d*,*e*), *link*(*i*,*e*).



Then, show the data structure after the following operations are performed as well (show the effects of path compression): *find*(*j*), *find*(*a*).



3. The height of a node in a tree is the length of a longest path from the node to one of its descendants. Show that if we leave out the path compression feature of the *find* operation from the partition data structure, then the *rank* of a node is equal to its height.

   By induction on the number of link operations. Initially all nodes have 0 rank and are leaves (which have height 0), so initially all nodes have rank equal to their height. When a link operation is performed on two nodes of different rank (height), the node with the larger rank (height) becomes the tree root and neither its rank nor its height changes. On the other hand, if the link operation joins two nodes of equal rank, the new height of the new root node increases by 1, as does the rank. The heights (and ranks) of other nodes are unaffected by the link, so the link operation preserves the equivalence between ranks and heights.

Show that with path compression, the rank of a node is an upper bound on the height.

By induction on number of link and find operations. As before, the claim is true initially because all nodes have zero rank and height. For a link operation, there are two cases. If the tree roots have different ranks, the height of the resulting tree can be no larger than the height of the "higher" tree, which is bounded by the larger rank. So making the node with the larger rank the tree root preserves the property. If the nodes have equal rank, the height of the new tree is at most one plus the old height which is at most one plus the rank of the roots.

A find operation with path compression can only reduce the height of any node in the tree and does not change the rank of any node. So, for any node x, if rank(x)≥height(x) before the operation, then this remains true after the operation.

4. Lemma 2.2 was proved for the partition data structure without path compression. Explain why it remains true when path compression is included.

   The lemma states that a tree whose root has rank k contains at least 2^k nodes. Path compression does not change the value of any rank and does not change the number of nodes in any tree (although it can change the number of nodes in certain subtrees). Therefore, find operations with path compression will preserve the property stated in the lemma.

5. In the analysis of path compression, explain why $\Delta(x)$ cannot decrease over time.

$\Delta(x)$ is the difference between the rank of a node's parent and its rank. As long as a node is the root of its tree, this difference is zero. Once a node acquires a parent it becomes non-zero and its own rank can no longer change. If the parent of the node is a tree root, link operations can cause the rank of the parent to increase, causing $\Delta(x)$ to increase along with it. If the parent is not a tree root, then its parent's rank may increase as a result of a find operation that assigns it a new parent. But since its new parent was an ancestor of its old parent, and since ranks increase as you go up a tree, the rank of the new parent must be larger than the rank of the old parent. So once again, $\Delta(x)$ increases.

Can $\Delta(x)$ can decrease as you follow parent pointers in the tree?

Yes. While the ranks increase as you go up the tree, the differences between successive rank values can both increase and decrease as you go up the tree.

6. Show that if we implement the partition data structure without linking by rank, but with path compression, we can do a sequence of *m* operations in $O(m \log n)$ time.

   Redefine the rank of a node as follows. Initialize rank(x)=1. While x is a tree root, whenever x acquires a new child y, add rank(y) to rank(x). Do not change rank(x) once it becomes the child of some other node. With this definition of rank, define Δ as on page 11. Note that Δ is 0 for all nodes initially, it never decreases and it can never be larger than n.

   We can then adapt the argument on page 11 to show that the number of dominant nodes on any find path is at most lg n. So, there are at most m lg n find steps involving dominant nodes.

   Similarly, when a non-dominant node is involved in a find step, its Δ value increases by a factor of at least 1.5. This can happen at most 2 lg n times to any node. Hence the number of find steps at non-dominant nodes is at most 2n lg n.