CS 341: ALGORITHMS

Lecture 13: graph algorithms IV – minimum spanning trees

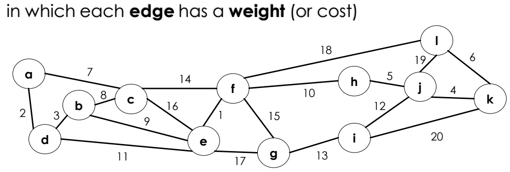
Readings: see website

Trevor Brown
https://student.cs.uwaterloo.ca/~cs341
trevor.brown@uwaterloo.ca

WEIGHTED UNDIRECTED GRAPH

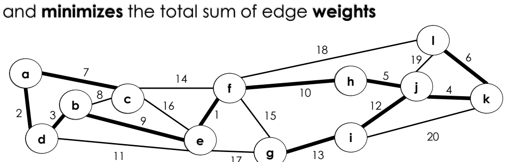
Problem can also be defined for directed graphs…

- Consider an **undirected** graph in which each **edge** has a **weight** (or cost)



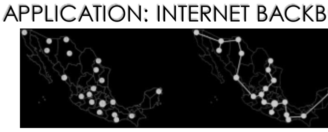
MINIMUM SPANNING TREE (MST)

- A tree (connected acyclic graph) that includes every node, and **minimizes** the total sum of edge **weights**



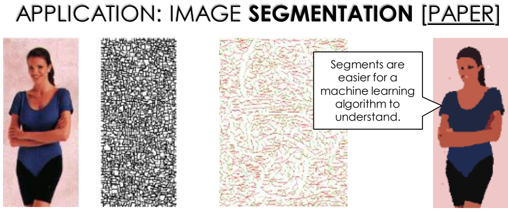
Problem can also be defined for minimum spanning **forest**. Algorithm taught here works.

APPLICATION: INTERNET BACKBONE PLANNING



- Want to connect n cities with internet backbone links
  - Direct links possible between each pair of cities
  - Each link has a certain dollar cost (excavation, materials, distance & time, legal costs…)
  - Want to **minimize total cost**

APPLICATION: IMAGE **SEGMENTATION** [PAPER]



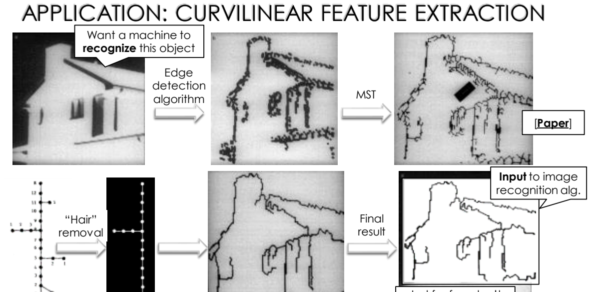
break image into **regions** by colour similarity via other techniques

turn regions into nodes, and add edges between them with weights = "dissimilarity," then build MST

break MST into large, highly similar **segments**, and assign the dominant colour to each **segment**

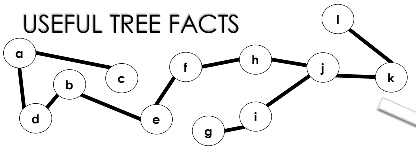
Just for fun, don't need to know this

APPLICATION: CURVILINEAR FEATURE EXTRACTION



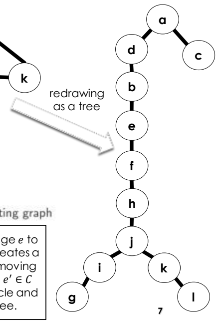
Just for fun, don't need to know this

USEFUL TREE FACTS



- A tree on n vertices has $n-1$ edges.
- There is a unique path between any two vertices in a tree.
- If T is a tree and an edge $e \notin T$ is added to T, then the resulting graph contains a unique cycle C.
- If $e' \in C$ then $T \cup \{e\} \setminus \{e'\}$ is a tree.

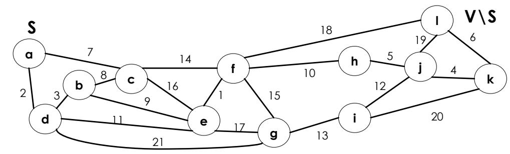
If you add an edge e to a tree and this creates a cycle C, then removing any other edge $e' \in C$ will break the cycle and produce a tree.



7

A CUT OF A GRAPH

- Definition: a **cut** in a graph G = (V,E) is a partition of V into two non-empty subsets **S** and **V \ S**

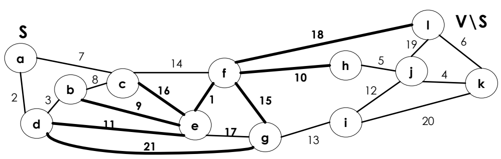


8

THE CUTSET OF A CUT

Edges in the cutset are also said to "**cross the cut**"

- Definition: given a cut (S, V\S), the **cutset** is the **set of edges** with one endpoint in **S** and the other in **V\S**

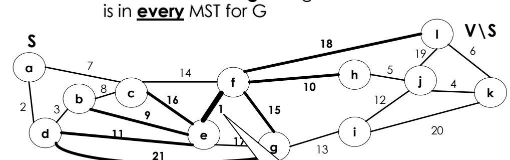


9

THE CUT PROPERTY

The minimum weight edge is also called the "**lightest edge**"

- Theorem: for **any cut** (S, V\S) of a graph G, the **minimum weight** edge in the **cutset** is in **every** MST for G

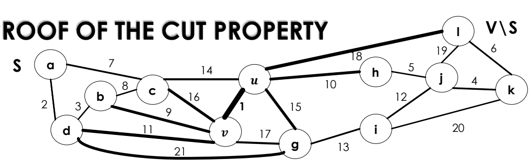


In **every** MST

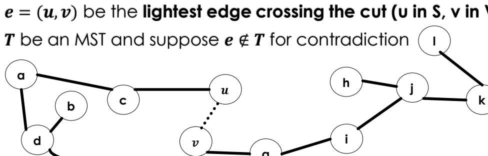
This can also be referred to as **the lightest edge crossing the cut**

10

PROOF OF THE CUT PROPERTY

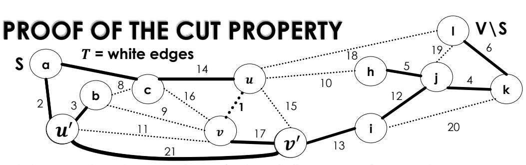


- Let $e = (u, v)$ be the **lightest edge crossing the cut (u in S, v in V\S)**
- Let T be an MST and suppose $e \notin T$ for contradiction



11

PROOF OF THE CUT PROPERTY



- We construct spanning T' s.t. $w(T') < w(T)$ for contra.
- T is spanning, so exists path $u \rightsquigarrow v$
- Path starts in S and ends in V\S so contains an edge $e' = (u', v')$ with $u' \in S, v' \in V\setminus S$

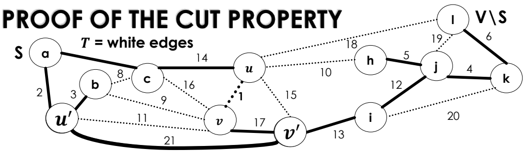
This edge **crosses the cut**

- Let $T' = T - \{e'\} + \{e\}$

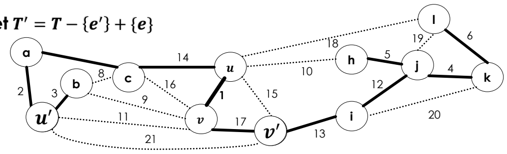
Exchanging edges that cross the cut

12

PROOF OF THE CUT PROPERTY

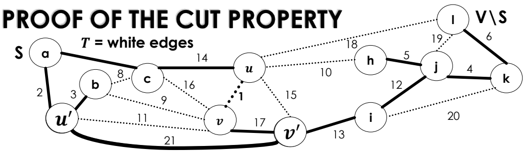


Let $T' = T - \{e'\} + \{e\}$

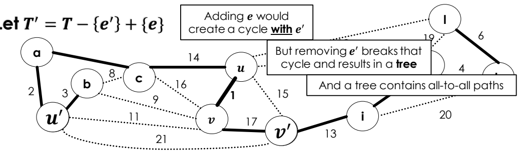


13

PROOF OF THE CUT PROPERTY



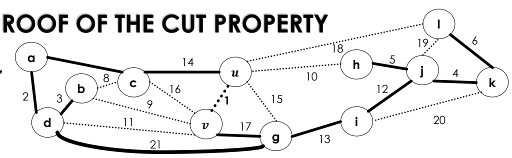
Let $T' = T - \{e'\} + \{e\}$



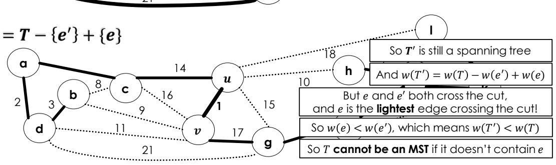
14

PROOF OF THE CUT PROPERTY

T



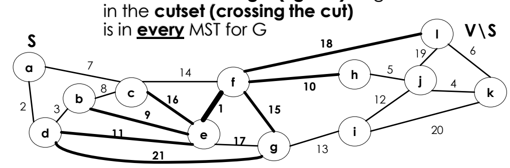
$T' = T - \{e'\} + \{e\}$



15

RECAP: THE CUT PROPERTY

- Theorem: for **any cut** (S, V\S) of a graph G, the **minimum weight (lighest)** edge in the **cutset (crossing the cut)** is in **every** MST for G



16

BUILDING AN MST

- **Kruskal's** algorithm [introduced **in this 3-page paper from 1955**]
- Greedy
  - Sort edges from lightest to heaviest
  - For each edge e in this order
    - Add e to T if it does not create a cycle

17

EXAMPLE EXECUTION

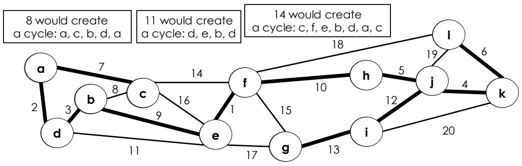
How can we test for cycles as we go?

Increasing edge weights: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20

8 would create a cycle: a, c, b, d, a

11 would create a cycle: d, e, b, d

14 would create a cycle: c, f, e, b, d, a, c



15 would create a cycle: g, f, h, j, l, g

16 would create a cycle…

17 would create a cycle…

18 would create a cycle…

19 would create a cycle…

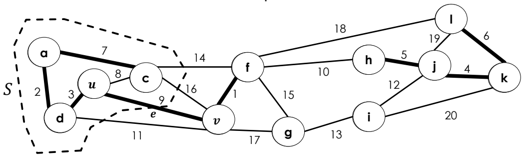
20 would create a cycle…

Done!

18

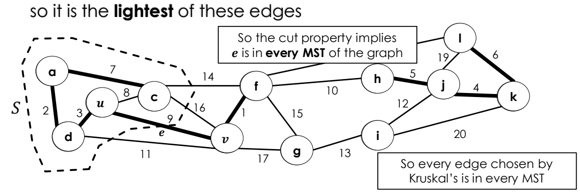
PROOF

- Let T be partial spanning tree just before adding $\boldsymbol{e} = (\boldsymbol{u}, \boldsymbol{v})$, the lightest edge that does not create a cycle
- Let S be the connected component of T that contains u



PROOF

- Note $\boldsymbol{e} = (\boldsymbol{u}, \boldsymbol{v})$ crosses the cut $(S, V \backslash S)$ or it would create a cycle
- Out of all edges crossing the cut, $\boldsymbol{e}$ is considered first, so it is the **lightest** of these edges



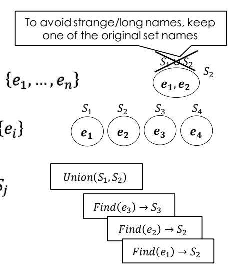
IMPLEMENTING KRUSKAL'S

- Sort edges from lightest to heaviest
- For each edge e in this order
  - Add e to T if it **does not create a cycle**

How can we determine whether adding e would create a cycle?

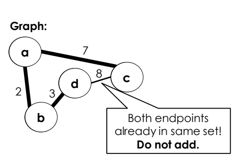
UNION FIND

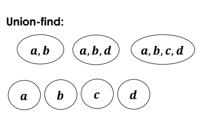
- Represents a **partition** of set $S = \{e_1, \ldots, e_n\}$ into **disjoint subsets**
  - Initially n disjoint subsets $S_i = \{e_i\}$
- Operations
  - $Union(S_i, S_j)$ replaces S_i and S_j by their union $S_i \cup S_j$
  - $Find(e_i)$ returns the **label** of the set containing e_i



KRUSKAL'S USING UNION-FIND

- Each graph node is initially in its own subset
- Add an edge → union two subsets
- An edge **creates a cycle IFF** its endpoints are in the **same subset**





PSEUDOCODE FOR KRUSKAL'S USING UNION-FIND

```
Kruskal(V[1..n], E[1..m])
    sort E[1..m] in increasing order by weight
    uf = new UnionFind data structure
    mst = new List
    for j = 1..m
        set_a = uf.find(E[j].source)
        set_b = uf.find(E[j].target)
        if set_a != set_b
            mst.add(E[j])
            uf.merge(set_a, set_b)
    return mst
```

TIME COMPLEXITY?

```
Kruskal(V[1..n], E[1..m])
    sort E[1..m] in increasing order by weight
    uf = new UnionFind data structure
    mst = new List
    for j = 1..m
        set_a = uf.find(E[j].source)
        set_b = uf.find(E[j].target)
        if set_a != set_b
            mst.add(E[j])
            uf.merge(set_a, set_b)
    return mst
```

Need to know runtime for union find…

For an efficient union-find algorithm (with union by rank and path compression), we get a total running time for Kruskal's algorithm of $O\big(\alpha(m+n)(m+n)\big)$, where $\alpha(x)$ is the inverse Ackermann function. For all practical x, $\alpha(x) \leq 5$, so this is ***pseudo*-linear**.

A simpler implementation with union-by-rank only yields $O(m \log n)$

25

OTHER NOTABLE MST ALGORITHMS

- Prim's algorithm
  - Incrementally extend a tree T into an MST, by:
  - Initializing T to contain any arbitrary node in G
  - Repeatedly selecting the lightest edge that crosses cut (T, V\T)
  - Visualization: https://www.cs.usfca.edu/~galles/visualization/Prim.html
- Borůvka's algorithm
  - Like Kruskal (merging components), but with **phases**
  - In each phase, select an outgoing edge for **every** component, and add **all** edges found in the phase

Use priority queue to store **outgoing** edges from T (and repeatedly extract the minimum weight one)

There is also a fast **parallel hybrid** of Prim and Borůvka

26

A FUN APPLICATION: MAZE BUILDING

- Create grid graph with
- edges up/down/left/right
- **Randomize** edge **weights** then run Kruskal's

[video clip]

27

VISUALIZING KRUSKAL'S (WITHOUT PATH COMPRESSION)

- https://www.cs.usfca.edu/~galles/visualization/Kruskal.html

28

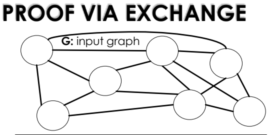
BONUS SLIDES

- Kruskal's proof via exchange argument instead

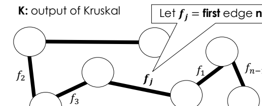
- Implementing union-find efficiently

29

PROOF VIA EXCHANGE



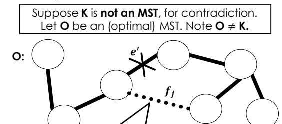
Suppose **K** is **not an MST**, for contradiction. Let **O** be an (optimal) MST. Note **O** ≠ **K.**



Let $\boldsymbol{f_j}$ = **first** edge **not in O**

Label edges so $\boldsymbol{w(f_1) < w(f_2) < \cdots < w(f_{n-1})}$. (we prove this for **distinct** weights)

Adding $\boldsymbol{f_j}$ to **O** would create cycle **C**



Let $\boldsymbol{O'}$ be same as O but with $\boldsymbol{e'}$ **and** $\boldsymbol{f_j}$ **swapped**

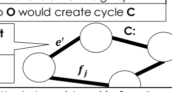
Note $w(O') = w(O) + w(f_j) - w(e')$

$w(O') \geq w(O)$ since O is optimal

So $w(f_j) - w(e') \geq 0$, so $\boldsymbol{w(f_j) > w(e')}$

Let $\boldsymbol{e'}$ = **smallest** edge in **C \ K**

(exists since no cycles in **K**)



Kruskal considers $\boldsymbol{e'}$ **before** $\boldsymbol{f_j}$, and **rejects** $\boldsymbol{e'}$ despite taking $f_1, \ldots, f_{j-1}$

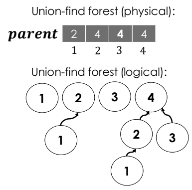
So, $f_1, \ldots, f_{j-1}, e'$ contains a cycle $\boldsymbol{C'}$

But $f_1, \ldots, f_{j-1}, e' \in O$. **Contradiction!**

30

UNION FIND IMPLEMENTATION

- Suppose we are partitioning set $\{1, \dots, n\}$ into **subsets** $S_1, \dots, S_n$
- Represent the partition as a **forest** of **trees**
  - Initially one single-node tree per subset
  - Each node has a **parent pointer**
- $Find(i)$ returns the **root** of the tree containing **element** i
- $Union(i, j)$ makes one root the parent of the other



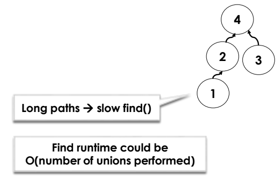
Let's union the sets containing elements 1 and 2
$find(1) \rightarrow 1$, $find(2) \rightarrow 2$
$Union(1,2)$: $parent[1] = 2$

How about elements 4 and 1?
$find(4) \rightarrow 4$, $find(1) \rightarrow 2$
$Union(4,2)$: $parent[2] = 4$

How about elements 3 and 1?
$find(3) \rightarrow 3$, $find(1) \rightarrow 4$
$Union(3,4)$: $parent[3] = 4$

31

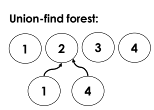
PROBLEM: SLOW FIND()



32

UNION-FIND WITH **UNION BY RANK**

- Keep track of **heights** of trees
- Make **root with greater height** be the **parent**
  - Union of two trees with height h has height $h+1$
  - Union of tree with height h and tree with height $< h$ has height h
- **Runtime** with union by rank?



Let's union the sets containing elements 1 and 2
$find(1) \rightarrow 1$, $find(2) \rightarrow 2$
$Union(1,2)$: **same height** → $parent[1] = 2$

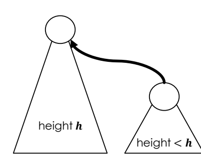
How about elements 4 and 1?
$find(4) \rightarrow 4$, $find(1) \rightarrow 2$
$Union(4,2)$: **2's height is greater** → $parent[4] = 2$

33

RUNTIME OF UNION BY RANK

- Can prove the following **lemma** by induction:
  - Each tree of height h contains at least 2^h nodes

Case 1: trees of different height



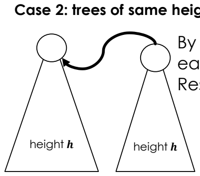
By I.H.,
left tree already has $\geq 2^h$ nodes.
So result has height h and $\geq 2^h$ nodes

34

RUNTIME OF UNION BY RANK

- Can prove the following **lemma** by induction:
  - Each tree of height h contains at least 2^h nodes

Case 2: trees of same height



By I.H.,
each tree has $\geq 2^h$ nodes.
Result has height $h+1$ and $\geq 2^h + 2^h$ nodes

And $2^h + 2^h = 2^{h+1}$. QED

35

RUNTIME OF UNION BY RANK

- How does the **lemma** help?
  - Each tree of height h contains at least 2^h nodes
- There are only $\boldsymbol{n}$ **nodes** in the graph
  - So **height** is at most $\mathbf{log}\, \boldsymbol{n}$
  - (Lemma: a tree of height $\log n$ contains at least $2^{\log n}$ nodes and $2^{\log n} = n$)
- So the longest path in the union-find forest is $\log n$
  - So all union-find operations run in $\Theta(\log n)$ time!

36

TIME COMPLEXITY **USING UNION BY RANK**

```
Kruskal(V[1..n], E[1..m])
    sort E[1..m] in increasing order by weight
    uf = new UnionFind data structure
    mst = new List
    for j = 1..m
        set_a = uf.find(E[j].source)
        set_b = uf.find(E[j].target)
        if set_a != set_b
            mst.add(E[j])
            uf.merge(set_a, set_b)
    return mst
```

$O(m \log m)$

$O(n)$

$O(1)$

$O(\log n)$

$O(1)$

$O(\log n)$

$O(m \log n)$

$O(\log n)$

Total $O(m \log n + m \log m)$

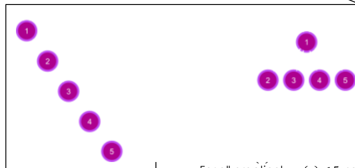
Trick: $\log m \leq \log n^2 = 2 \log n \in O(\log n)$

So runtime is in $O(m \log n)$

MAKING THIS EVEN FASTER

In addition to union by rank, union-find can be implemented with **path compression**

This variant is introduced <u>**in this paper**</u>



compression, we get a total
m of $O(\alpha(m+n)(m+n))$,
kermann function.
For all practical x, $\alpha(x) \leq 5$, so this is ***pseudo*-linear**.

EFFICIENT UNION-FIND

```
class UnionFind {
    int * parent
    int * rank;
    UnionFind(int n) {
        parent = new int[n];
        rank = new int[n];
        for (int i=0; i<n; i++) {
            rank[i] = 0;
            parent[i] = i;
        }
    }
    ~UnionFind() {
        delete[] parent;
        delete[] rank;
    }
    int find(int u) {
        if (u != parent[u]) parent[u] = find(parent[u]);
        return parent[u];
    }
    void merge(int x, int y) {
        x = find(x), y = find(y);
        if (rank[x] > rank[y]) parent[y] = x;
        else parent[x] = y;
        if (rank[x] == rank[y]) rank[y]++;
    }
};
```

Initialization

Free memory at end

Path compression

Union by rank