Minimum Spanning Trees

CS 4231, Fall 2020 Mihalis Yannakakis

Minimum Spanning Tree

- Input: *Undirected*, connected weighted graph

  $G(N,E)$, weights $w: E \rightarrow R$ (≥ 0 or <0)
- Output: Spanning tree T of minimum weight

  (= Min weight subgraph that connects all nodes if weights ≥ 0

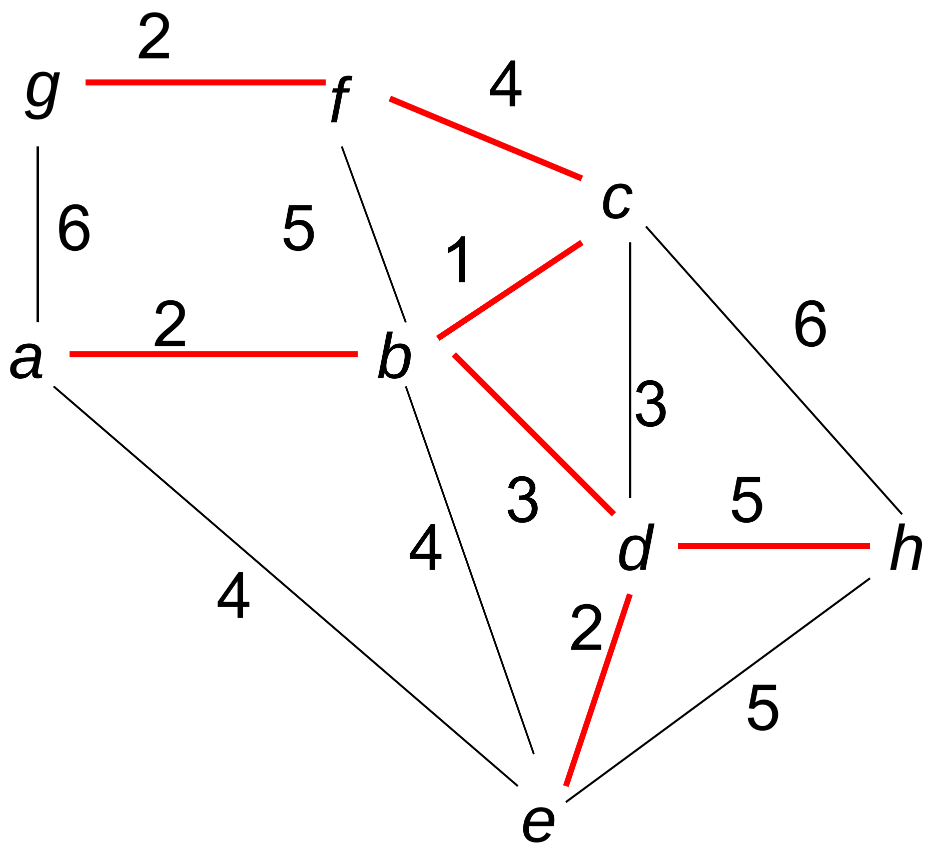
  Proof: If Cycle -> remove any edge of cycle)

- Applications: Power distribution network (Boruvka 1926), road-, phone-, TV cable-, computer- network

Maximum Spanning Tree $\Leftrightarrow$ Minimum Spanning Tree

- negate weights

Example



Weight of MST = 19

Minimum Spanning Tree - Algorithms

- Exhaustive: Enumerate all spanning trees
- Too many, in general exponential number
- Complete graph: #spanning trees = n^{n-2} (Caley)

Greedy algorithms: Prim, Kruskal

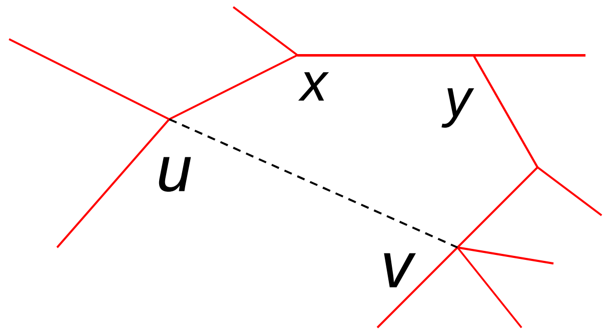
At each step choose minimum weight edge that satisfies a certain criterion

A general underlying mathematical structure ("matroids") in Chapter 16

- analysis of problem => structure of optimal solutions

Exchange argument

Tree T:



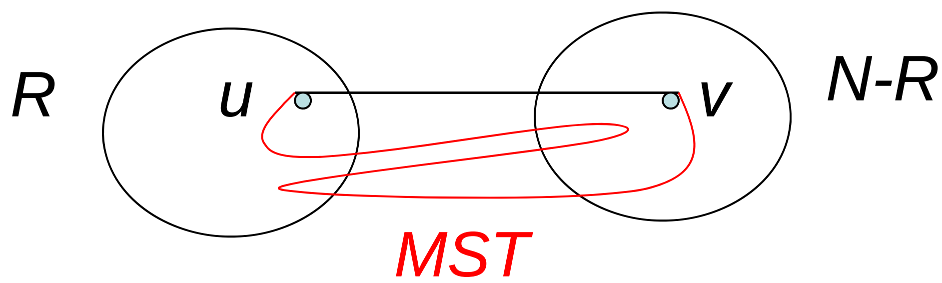
- Given a spanning tree T, adding an out-of-tree edge (u,v) forms a unique cycle: edge (u,v) + u-v path of tree.
  - Exchanging any edge (x,y) of the u-v path with edge (u,v) $\rightarrow$ another spanning tree T'. Cost lower if $w(u,v) < w(x,y)$
- T is a MST $\Rightarrow$ $\forall$ out-of-tree edge (u,v) has weight $\geq$ maximum weight of the edges along the u-v path in tree T
- Converse also true: HW exercise

Edges across partitions

Partition Theorem: For any set of edges A contained in some MST and for every partition (R,N-R) of the nodes such that no edge of A crosses the partition:

1. Every MST contains some min weight edge across partition (i.e. edge from R to N-R)
2. Every min weight edge across partition $\in$ some MST that contains also A

Proof: Exchange argument for both parts



If w(u,v) $\leq$ weight of MST edges across partition, replace one of these MST edges on u-v path with (u,v)

Some Optimality Conditions

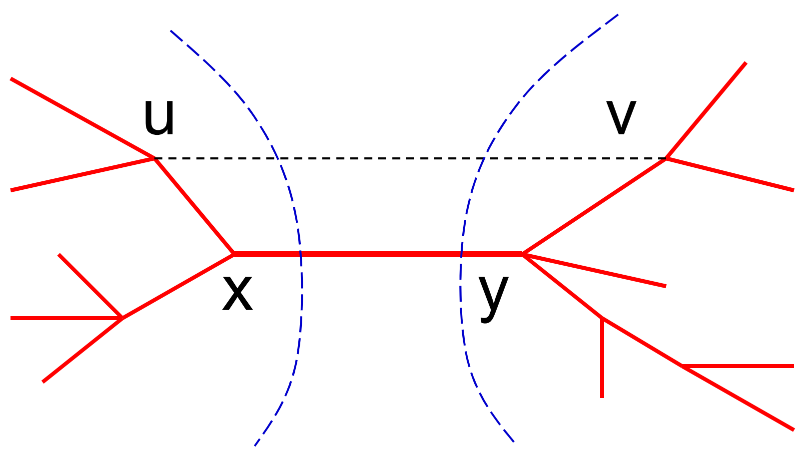
1. A spanning tree T has minimum weight

$\Leftrightarrow$

2. $\forall$ out-of-tree edge (u,v) has weight $\geq$ maximum weight of the edges along the u-v path in tree T

$\Leftrightarrow$

3. $\forall$ tree edge (x,y) has minimum weight among edges connecting the two components formed by removing the edge from T



Properties

- MST depends only on relative order of edge weights, not their actual values
- If edge weights are distinct $\Rightarrow$ unique MST
- Trick: Can perturb edge weights slightly to break any ties to prove optimality of a spanning tree
  (simplifies proofs)

HW Exercise:
Prove the optimality conditions and the properties

Prim's algorithm

- Graph Search algorithm from a source node s

  Start with R={s} and iteratively add nodes to R.

  Policy: in each iteration choose an edge from R to N-R that has minimum weight

- Can implement using a priority queue Q for nodes in N-R with priority of a node = min weight of an edge from a node in R

- Correctness follows from Partition Theorem

  Proof by induction on #selected edges that there exists an MST that contains all the edges selected so far.

Prim's Algorithm

Worst-Case Time Complexity
Heap: O(elogn)
Fibonacci Heap: O(e+logn)

MST-Prim(G,w,s)

for each $v \in N$ do {d[v]=∞; p[v]=$\perp$; mark[v]=0}
d[s]=0;
Q = N   [Q a priority queue with priority d[ ]]
[alternatively, Q={s} initially and insert nodes when reached]
while Q $\neq \emptyset$ do
{   u=Extract-Min(Q)   [Extract-Min operation; first time, u=s]
    mark[u]=1
    for each v $\in$ Adj[u] do
        if mark[v]=0 and d[v] > w(u,v) then
        {d[v]= w(u,v); p[v]=u}   [Decrease-Key(v) operation]
}

Time Complexity

Operations:	Extract-Min	Decrease-Key
# of ops:	n	e
Time/Op.		
Heap:	$O(\log n)$	$O(\log n)$
Fibonacci Heap: (*amortized*)	$O(\log n)$	$O(1)$

Total (Worst-Case) Time Complexity:

Heap: $O(e\log n)$

Fibonacci Heap: $O(e+n\log n)$

Amortized Analysis of Data Structures

- amortized complexity of operation: bounds the time per operation in any sequence of operations

  = average per operation = (total time) / #operations
- But no probability: we consider the worst sequence of ops
- An individual operation in the sequence may take lot of time but compensated by earlier cheaper ones

- The Fibonacci heap data structure has amortized complexity O(logn) for Extract-Min and O(1) for Decrease-Key.

Kruskal's Algorithm

Worst-Case Time Complexity
O(elogn)

(1. Sort the edges in nondecreasing order of weight)*

2. Process the edges in order of weight: each edge (u,v) is included in the tree if its nodes u,v are in different connected components in subgraph defined by selected edges so far.

```
for each u in N do Comp(u)={u}
T = ∅
 for each edge (u,v) of E in sorted order do
   if ( Comp(u) ≠ Comp(v)) then
      {   T = T ∪ {(u,v)}
          Union sets Comp(u) and Comp(v)
      }
```

*Instead of sorting the edges, we could use a priority queue and extract-min in each iteration, until we have n-1 edges in T

Kruskal's Algorithm

1. Sort the edges in nondecreasing order of weight
2. Process the edges in order: each edge (u,v) is included in the tree if its nodes u,v are in different connected components in subgraph defined by selected edges so far.

Correctness: By Partition Theorem

Time Complexity: $O(e\log n)$

Implementation: Need to maintain components

Operations: Find component of a node
Union two components

At most 2e Find operations (2 for each edge)
n-1 Union operations (1 for each edge of the tree)

Union-Find (Disjoint Sets) Data structure

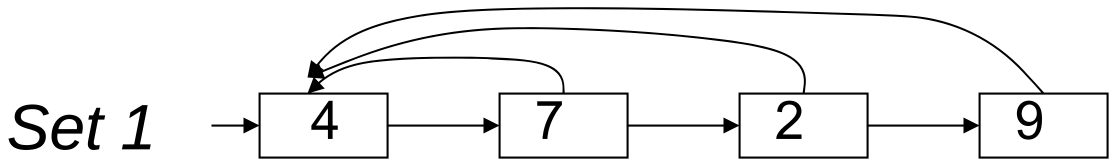
- Maintain a family of *disjoint sets* over a set N of n elements

- Initially each element in singleton set by itself

- Sequence of operations:
- FIND(x): return (pointer to) set that contains x
- UNION(S,T): union sets S,T

One simple approach: Linked Lists

- Linked list for each set, with pointers from all elements to the head



- FIND: O(1) time

•UNION: Concatenate the lists and update pointers of the elements of one list

- Key idea: Update the smaller set

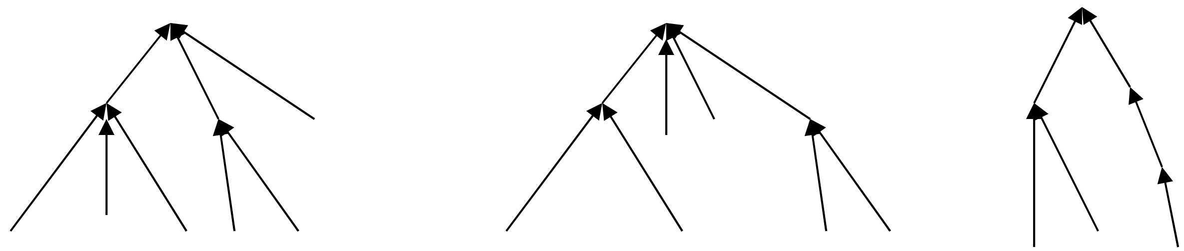
- m FINDS, n UNIONS: Time O(m + nlogn)

Reason: Every node has its pointer updated $\leq$ logn time, because this happens only when the size of its set doubles

Faster: Forest Data structure

- Rooted Tree for each set, with elements at the nodes

(the tree for a set is *not* the same as tree for a component)



FIND(x): Trace path from x to the root of its tree, and hang all nodes of the path from the root ("path compression")

UNION(S,T): Hang the root of one tree (the one with smaller "rank") from the root of the other

Amortized time per operation: $\alpha(n)$ an *extremely* slowly growing function: $\alpha(n) \leq 4$ for all realistic n (eg. for $n \leq 10^{80}$)

See Chapter 21 for more details

More advanced MST Algorithms

- If edges sorted $\Rightarrow$ Kruskal almost linear time
- Sorting is *not* required for MST
- Many further algorithms, improving the running time
- Randomized linear time (Karger-Klein-Tarjan'95)
- O(e α(n)) deterministic time (Chazelle'2000)
- Optimal time (Petie-Ramachandran'2002)
- Open : Deterministic Linear time?

Applications – Clustering

- Complete Graph
- Vertices = objects (eg. documents, images, dna seqs…)
- Edge weights = dissimilarity/distance measure
- Clustering problems: Want similar objects in same cluster, dissimilar in different clusters
- Various metrics to evaluate clusterings

- One metric: Partition objects into k clusters to maximize the minimum distance between any two objects in different clusters
- Run Kruskal's algorithm till k components

  - HW Exercise: Prove that Kruskal computes the optimal k-clustering under this metric

Applications – Bottleneck paths

- Undirected Graph = network
- edge weights = bandwidth of links
- Bandwidth of path = min bandwidth of an edge on path
- Find max bandwidth path from s to t (or from s to all nodes, or between all pairs)

- Undirected graphs: Maximum Weight Spanning Tree gives max bandwidth paths between all pairs of nodes

  - HW Exercise: prove it