Minimum Spanning Trees: Prim-Jarnik & Kruskal

CS16: Introduction to Data Structures & Algorithms
Seny Kamara - Spring 2018
Outline

- Minimum Spanning Trees
  - Analysis
  - Proof of Correctness
- Prim-Jarnik Algorithm
  - Analysis
  - Proof of Correctness
- Kruskal’s Algorithm
  - Union-Find
  - Analysis
  - Proof of Correctness
Spanning Trees

- A **spanning tree** of a graph is
  - subset of edges that form a tree that spans every vertex
Minimum Spanning Trees

- A **minimum spanning tree** (MST) is a spanning tree with minimum total edge weight.
Applications

- Networks
  - electric
  - computer
  - water
  - transportation

- Computer vision
  - Facial recognition
  - Handwriting recognition

- Low-density parity check codes (LDPC)
Minimum Spanning Tree Algos

- Prim-Jarnik’s Algorithm
- Kruskal’s Algorithm
Prim-Jarnik Algorithm

- Traverse $G$ starting at any node
  - Maintain priority queue of nodes
  - set priority to weight of the edge that connects them to MST
- Un-added nodes start with priority $\infty$
- At each step
  - Connect the node with lowest cost
  - Update ("relax") neighbors as necessary
- Stop when all nodes added to MST
Example

\[\text{PQ} = [(0, A), (\infty, B), (\infty, C), (\infty, D), (\infty, E), (\infty, F)]\]
Example

Dequeue from PQ and update neighbors

\[
PQ = [(4, B), (5, D), (\infty, C), (\infty, E), (\infty, F)]
\]
Example

$PQ = \{(4,C), (4,D), (6,E), (8,F)\}$

Deque from $PQ$ and update neighbors.
Example

PQ = [(2, E), (4, D), (8, F)]
Example

$$PQ = [(4, D), (4, F)]$$

Dequeue from PQ and update neighbors
Example

Dequeue from PQ and update neighbors

\[ \text{PQ} = [(3, F)] \]
Example

\[ PQ = \begin{bmatrix} \end{bmatrix} \]

Dequeue from PQ and update neighbors.
Example
function `prim(G)`:
    // Input: weighted, undirected graph G with vertices V
    // Output: list of edges in MST
    for all `v` in `V`:
        `v`.cost = ∞
        `v`.prev = null
    source = a random `v` in `V`
    source.cost = 0
    MST = []
    PQ = PriorityQueue(V) // priorities will be `v`.cost values
    while PQ is not empty:
        `v` = PQ.removeMin()
        if `v`.prev != null:
            MST.append((`v`, `v`.prev))
        for all incident edges `(v, u)` of `v`:
            if `u`.cost > `(v, u).weight`:
                `u`.cost = `(v, u).weight`
                `u`.prev = `v`
                PQ.replaceKey(u, u.cost)
    return MST
function `prim(G)`:
   for all `v` in `V`:
      `v`.cost = ∞
      `v`.prev = null
   source = a random `v` in `V`
   source.cost = 0
   MST = []
   PQ = PriorityQueue(`V`) // priorities will be `v`.cost values
   while PQ is not empty:
      `v` = PQ.removeMin()
      if `v`.prev != null:
         MST.append((`v`, `v`.prev))
      for all incident edges `(v,u)` of `v`:
         if `u`.cost > `(v,u)`.weight:
            `u`.cost = `(v,u)`.weight
            `u`.prev = `v`
            PQ.replaceKey(`u`, `u`.cost)
   return MST
Simulate Prim-Jarnik

function prim(G):
    for all v in V:
        v.cost = ∞
        v.prev = null
    source = a random v in V
    source.cost = 0
    MST = []
    PQ = PriorityQueue(V) // priorities will be v.cost values
    while PQ is not empty:
        v = PQ.removeMin()
        if v.prev != null:
            MST.append((v, v.prev))
        for all incident edges (v,u) of v:
            if u.cost > (v,u).weight:
                u.cost = (v,u).weight
                u.prev = v
                PQ.replaceKey(u, u.cost)
    return MST
Simulate Prim-Jarnik

function `prim(G):`
  for all v in V:
    v.cost = ∞
    v.prev = null
  source = a random v in V
  source.cost = 0
  MST = []
  PQ = PriorityQueue(V)  // priorities will be v.cost values
  while PQ is not empty:
    v = PQ.removeMin()
    if v.prev != null:
      MST.append((v, v.prev))
    for all incident edges (v,u) of v:
      if u.cost > (v,u).weight:
        u.cost = (v,u).weight
        u.prev = v
        PQ.replaceKey(u, u.cost)
  return MST
Simulate Prim-Jarnik

```python
function prim(G):
    for all v in V:
        v.cost = ∞
        v.prev = null
    source = a random v in V
    source.cost = 0
    MST = []
    PQ = PriorityQueue(V)  // priorities will be v.cost values
    while PQ is not empty:
        v = PQ.removeMin()
        if v.prev != null:
            MST.append((v, v.prev))
        for all incident edges (v,u) of v:
            if u.cost > (v,u).weight:
                u.cost = (v,u).weight
                u.prev = v
                PQ.replaceKey(u, u.cost)
    return MST
```
Simulate Prim-Jarnik

function **prim**(*G*):
    for all *v* in *V*:
        *v*.cost = ∞
        *v*.prev = null
    source = a random *v* in *V*
    source.cost = 0
    MST = []
    PQ = PriorityQueue(*V*)  // priorities will be *v*.cost values
    while PQ is not empty:
        *v* = PQ.removeMin()
        if *v*.prev != null:
            MST.append((*v*, *v*.prev))
        for all incident edges (*v*,*u*) of *v*:
            if *u*.cost > (*v*,*u*).weight:
                *u*.cost = (*v*,*u*).weight
                *u*.prev = *v*
                PQ.replaceKey(*u*, *u*.cost)
    return MST
Runtime of Prim-Jarnik

Activity #2
Runtime of Prim-Jarnik

2 min

Activity #2
Runtime of Prim-Jarnik

Activity #2
Runtime of Prim-Jarnik

0 min

Activity #2
Runtime Analysis

- Decorating nodes with distance and previous pointers is $O(|V|)$
- Putting nodes in PQ is $O(|V| \log |V|)$ (really $O(|V|)$ since $\infty$ priorities)
- While loop runs $|V|$ times
  - removing vertex from PQ is $O(\log |V|)$
  - So $O(|V| \log |V|)$
- For loop (in while loop) runs $|E|$ times in total
  -Replacing vertex’s key in the PQ is $\log |V|$
  - So $O(|E| \log |V|)$
- Overall runtime
  - $O(|V| + |V| \log |V| + |V| \log |V| + |E| \log |V|)$
  - $= O((|E| + |V|) \log |V|)$
Proof of Correctness

- Common way of proving correctness of greedy algos
  - show that algorithm is always correct at every step
- Best way to do this is by induction
  - tricky part is coming up with the right invariant
Graph Cuts

- A cut is any partition of the vertices into two groups

- Here $G$ is partitioned in 2
  - with edges $b$ and $a$ joining the partitions
Proof of Correctness

- \( P(n) \)
  - first \( n \) edges added by Prim are a subtree of some MST
- Base case when \( n=0 \)
  - no edges have been added yet so \( P(0) \) is trivially true
- Inductive Hypothesis
  - first \( k \) edges added by Prim form a tree \( T \) which is subtree of some MST \( M \)
Proof of Correctness

- Inductive Step
  - Let \( e \) be the \((k+1)\)th edge that is added
  - \( e \) will connect \( T \) (green nodes) to an unvisited node (one of blue nodes)
  - We need to show that adding \( e \) to \( T \)
    - forms a subtree of some MST \( M' \)
    - (which may or may not be the same MST as \( M \))
Proof of Correctness

- Two cases
  - **e** is in original MST **M**
  - **e** is not in **M**

- Case 1: **e** is in **M**
  - there exists an MST that contains first **k+1** edges
  - So **P(k+1)** is true!
Proof of Correctness

- Case 2: e is not in M
  - if we add \( e=(u,v) \) to M then we get a cycle
  - why? since M is span. tree there must be path from u to v w/o e
  - so there must be another edge \( e' \) that connects T to unvisited nodes
  - We know \( e\.weight \leq e'\.weight \) because Prim chose e first
Proof of Correctness

- So if we add $e$ to $M$ and remove $e'$
  - we get a new MST $M'$ that is no larger than $M$ and contains $T$ & $e$

- $P(k+1)$ is true
  - because $M'$ is an MST that contains the first $k+1$ edges added by Prim’s
Proof of Correctness

- Since we have shown
  - \( P(0) \) is true
  - \( P(k+1) \) is true assuming \( P(k) \) is true (for both cases)
  - The first \( n \) edges added by Prim form a subtree of some MST
Outline

- Minimum Spanning Trees
  - Analysis
  - Proof of Correctness
- Prim-Jarnik Algorithm
- Kruskal’s Algorithm
  - Union-Find
  - Analysis
  - Proof of Correctness
Kruskal’s Algorithm

- Sort edges by weight in ascending order
- For each edge in sorted list
  - If adding edge does not create cycle...
  - ...add it to MST
- Stop when you have gone through all edges
Example

edges = [ (C,E), (D,F), (B,C), (E,F), (B,D), (A,B), (A,D), (B,E), (B,F) ]
Simulate Kruskal

Activity #3

2 min
Simulate Kruskal

Activity #3

2 min
Simulate Kruskal

Activity #3
Simulate Kruskal
Kruskal

- How can we tell if adding edge will create cycle?
  - could run BFS/DFS to detect a cycle
  - but that’s slow!
- Start by giving each vertex its own “cloud”
- When edge is added to MST
  - \texttt{union( )} or merge clouds of the endpoints
  - If both ends of edge are in same cloud
  - we know that adding the edge will create a cycle!
Example

edges = [(C,E), (D,F), (B,C), (E,F), (B,D), (A,B), (A,D), (B,E), (B,F)]
Example

edges = [(D,F), (B,C), (E,F), (B,D), (A,B), (A,D), (B,E), (B,F)]
Example

edges = [(B,C), (E,F), (B,D), (A,B), (A,D), (B,E), (B,F)]
Example

edges = [(E,F), (B,D), (A,B), (A,D), (B,E), (B,F)]
Example

edges = [(B,D), (A,B), (A,D), (B,E), (B,F)]
Example

edges = [(A, B), (A, D), (B, E), (B, F)]

BD cannot be added because it would lead to a cycle
Example

edges = [(A,D), (B,E), (B,F)]
Example

AD cannot be added because it would lead to a cycle

edges = [(B,E),(B,F)]
Example

BE cannot be added because it would lead to a cycle

\[
\text{edges} = [(B,F)]
\]
Example

edges = []

BF cannot be added because it would lead to a cycle
function kruskal(G):
    // Input: undirected, weighted graph G
    // Output: list of edges in MST
    for vertices v in G:
        makeCloud(v) // put every vertex into its own set
    MST = []
    Sort all edges
    for all edges (u,v) in G sorted by weight:
        if u and v are not in same cloud:
            add (u,v) to MST
            merge clouds containing u and v
    return MST
Merging Clouds (Naive way)

- Assign each vertex a different number
  - that represents its initial cloud
- To merge clouds of \( u \) and \( v \)
  - Find all vertices in each cloud
  - Figure out which of the clouds is smaller
  - Redecorate all vertices in smaller cloud with bigger cloud’s number
Merging Clouds (Naive way)

- Finding all vertices in u & v's clouds is $O(|V|)$
  - because we have to iterate through each vertex...
  - …and check if its cloud number matches u or v’s cloud number
- Figuring out smaller cloud is $O(1)$
  - as long as we keep track of cloud size as we find vertices in them
- Changing cloud numbers of nodes in smaller cloud is $O(|V|)$
  - because cloud could be as big as $|V|/2$ vertices
- Total Runtime
  - $O(|V|) + O(1) + O(|V|) = O(|V|)$
Runtime of Naive Kruskal

- Finding all vertices in $u$ & $v$'s clouds is $O(|V|)$
  - because we have to iterate through each vertex...
  - ...and check if its cloud number matches $u$ or $v$'s cloud number
- Figuring out smaller cloud is $O(1)$
  - as long as we keep track of cloud size as we find vertices in them
- Changing cloud numbers of vertices in smaller cloud is $O(|V|)$
  - because cloud could be as big as $|V|/2$ vertices
- Total Runtime
  - $O(|V|) + O(1) + O(|V|) = O(|V|)$

Activity #4
2 min
Runtime of Naive Kruskal

- Finding all vertices in u & v's clouds is $O(|V|)$
  - because we have to iterate through each vertex...
  - ...and check if its cloud number matches u or v's cloud number
- Figuring out smaller cloud is $O(1)$
  - as long as we keep track of cloud size as we find vertices in them
- Changing cloud numbers of vertices in smaller cloud is $O(|V|)$
  - because cloud could be as big as $|V|/2$ vertices
- Total Runtime
  - $O(|V|) + O(1) + O(|V|) = O(|V|)$

Activity #4

2 min
Runtime of Naive Kruskal

- Finding all vertices in $u$ & $v$'s clouds is $O(|V|)$
  - because we have to iterate through each vertex...
  - ...and check if its cloud number matches $u$ or $v$'s cloud number
- Figuring out smaller cloud is $O(1)$
  - as long as we keep track of cloud size as we find vertices in them
- Changing cloud numbers of vertices in smaller cloud is $O(|V|)$
  - because cloud could be as big as $|V|/2$ vertices
- Total Runtime
  - $O(|V|) + O(1) + O(|V|) = O(|V|)$
Runtime of Naive Kruskal

- Finding all vertices in \( u \) & \( v \)'s clouds is \( O(|V|) \)
  - because we have to iterate through each vertex…
  - …and check if its cloud number matches \( u \) or \( v \)'s cloud number
- Figuring out smaller cloud is \( O(1) \)
  - as long as we keep track of cloud size as we find vertices in them
- Changing cloud numbers of vertices in smaller cloud is \( O(|V|) \)
  - because cloud could be as big as \( |V|/2 \) vertices
- Total Runtime
  - \( O(|V|) + O(1) + O(|V|) = O(|V|) \)
Kruskal Runtime

function kruskal(G):
    // Input: undirected, weighted graph G
    // Output: list of edges in MST
    for vertices v in G:
        makeCloud(v)
    MST = []
    Sort all edges
    for all edges (u,v) in G sorted by weight:
        if u and v are not in same cloud:
            add (u,v) to MST
            merge clouds containing u and v
    return MST

\[ O(|V|) \]
\[ O(|E| \log |E|) \]
\[ O(|E|) \]
\[ O(|V|) \]
Kruskal Runtime

- \( O(|V|) \) for iterating through vertices
- \( O(|E| \log |E|) \) for sorting edges
- \( O(|E|) \times O(|V|) \) for iterating through edges and merging clouds naively
- \( O(|V|) + O(|E| \log |E|) + O(|E|) \times O(|V|) \)
  - \( = O(|V||E|) = O(|V|^3) \)
- Can we do better?
Union-Find

- Let's rethink notion of clouds
  - instead of labeling vertices w/ cloud numbers
  - think of clouds as small trees
- Every vertex in these trees has
  - a parent pointer that leads up to root of the tree
  - a rank that measures how deep the tree is
edges = [(C, E), (D, F), (B, C), (E, F), (B, D), (A, B), (A, D), (B, E), (B, F)]
edges = [(D,F), (B,C), (E,F), (B,D), (A,B), (A,D), (B,E), (B,F)]
Example

edges = [(B,C), (E,F), (B,D), (A,B), (A,D), (B,E), (B,F)]
Example

\[\text{edges} = \{(E,F), (B,D), (A,B), (A,D), (B,E), (B,F)\}\]
edges = [(B,D), (A,B), (A,D), (B,E), (B,F)]
edges = [(A,D), (B,E), (B,F)]
Example

edges = [(A,D), (B,E), (B,F)]
Implementing Union-Find

- At start of Kruskal
  - every node is put into own cloud

```plaintext
// Decorates every vertex with its parent ptr & rank
function makeCloud(x):
    x.parent = x
    x.rank = 0
```

0

A

0

B
Implementing Union-Find

- Suppose \(A\) is in cloud 1 and \(B\) is in cloud 2
- Instead of relabeling \(B\) as cloud 1 make \(B\) point to \(A\)
  - Think of this as the union of two clouds
- Given two clouds which one should point to the other?
Implementing Union-Find

- Use the rank property!
- For clouds of size 1
  - root has rank 0
- For clouds larger than 1
  - rank is updated during a `union()` operation
    - +1 when merged with cloud of same size
Implementing Union-Find

- Merging trees with same rank
Implementing Union-Find

- Merging trees with same rank

Diagram showing the structure of trees with ranks and merge operations.
Implementing Union-Find

- Merging trees with different ranks
Implementing Union-Find

- Merging trees with different ranks
Implementing Union-Find

// Merges two clouds, given the root of each cloud
function union(root1, root2):
    if root1.rank > root2.rank:
        root2.parent = root1
    elif root1.rank < root2.rank:
        root1.parent = root2
    else:
        root2.parent = root1
        root1.rank++
Implementing Union-Find

- To find cloud of B
  - follow parent pointer to root

```plaintext
// Finds the cloud of a given vertex
function find(x):
    while x != x.parent:
        x = x.parent
    return x
```

![Diagram of A and B connected through a parent pointer](image)
Path Compression

- This approach to implementing `find` runs in $O(\log |V|)$
- We can bring this down to amortized $O(1)$ with path compression...
- ...a way of flattening the structure of the tree...
- ...whenever `find()` is used on it
Path Compression

- Instead of traversing up tree every time D's cloud is asked for
  - We only search for D's cloud once
  - As we follow chain of parents to A we set parents of D & C to A

Amortized $O(1)$
// Tweak find(...) to include path compression

function find(x):
    if x != x.parent:
        x.parent = find(x.parent)
    return x.parent
Runtime of Kruskal w/ Path Compression

Activity #5
Runtime of Kruskal w/ Path Compression

Activity #5

1 min
Runtime of Kruskal w/ Path Compression

Activity #5
function `kruskal(G)`:

// Input: undirected, weighted graph G
// Output: list of edges in MST

for vertices v in G:
    makeCloud(v)

MST = []

Sort all edges

for all edges (u,v) in G sorted by weight:
    if u and v are not in same cloud:
        add (u,v) to MST
        merge clouds containing u and v

return MST

- $O(|V|)$
- $O(|E| \log |E|)$
- $O(|E|)$
- $O(1)$ amortized
Kruskal Runtime

- $O(|V|)$ for iterating through vertices
- $O(|E| \log |E|)$ for sorting edges
- $O(|E| \times O(1))$ for iterating through edges and merging clouds with path compression

\[
O(|V|) + O(|E| \log |E|) + O(|E|) \times O(1)
\]

\[
= O(|E| \log |E|)
\]

- $O(|E| \log |E|)$ much better than $O(|V|^3)$
Readings

- Dasgupta Section 5.1
  - Explanations of MSTs
  - and both algorithms discussed in this lecture