CSE 421: Introduction to Algorithms

Yin Tat Lee
Guest Lecturer: Jeremy Lin (TA)
Lecture Outline:

- Spanning Trees / Minimum Spanning Trees
- Cut Property
- Cycle Property
- Kruskal’s Algorithm
  - Union Find Data Structure
- (Briefly) talk about Prim’s and Reverse-Delete algorithms
Spanning Trees

• A tree $T$ is a spanning tree of a graph $G$ if:
  • $T$ is a valid tree (obviously)
  • $T$ includes all vertices in $G$
  • $T$ includes only edges in $G$ (but possibly not all edges in $G$)

• More formally, if $G = (V, E)$ and $T = (V', E')$ then:
  • $V' = V$
  • $|E'| = |V'| - 1$
  • $E' \subseteq E$

http://www.youtube.com/watch?v=Qw4w9WgXcQ
Minimum Spanning Trees (MST)

- An MST is the lowest-cost spanning tree of a graph
Minimum Spanning Trees (MST)

• A graph may have multiple possible MSTs!
• Trivial example:
Yesterday: Dijkstra’s Algorithm

• Find the shortest path from vertex S to all other vertices in G
  • Guaranteed non-negative edges, etc.

• If you draw out all the shortest paths calculated on G, do they always form a (spanning) tree?
  • (Assume no two paths from S to T “tied” for shortest)
Yesterday: Dijkstra’s Algorithm

• Find the shortest path from vertex S to all other vertices in G
  • Guaranteed non-negative edges, etc.

• If you draw out all the shortest paths calculated on G, do they always form a spanning tree? Yes!

• Proof Sketch: (Contradiction)
  • Suppose that the graph G’ formed by connecting the shortest paths as described above is not a tree -> it must have a cycle by definition)
  • Let T be a vertex in a cycle. Therefore, there must be two paths from S to T
    • One of them is not used for any shortest paths, since for any of T’s neighbors T’ we will always path from S to T’ by taking the shorter path from S to T first, then the path from T to T’
    • But then this contradicts how we constructed G’!
Yesterday: Dijkstra’s Algorithm

• Find the shortest path from vertex S to all other vertices in G
  • Guaranteed non-negative edges, etc.

• If you draw out all the shortest paths calculated:
  • Do they form a (spanning) tree? Yes!
  • Do they form a minimum spanning tree?

https://www.youtube.com/watch?v=dQw4w9WgXcQ
Yesterday: Dijkstra’s Algorithm

- Find the shortest path from vertex $S$ to all other vertices in $G$
  - Guaranteed non-negative edges, etc.
- If you draw out all the shortest paths calculated:
  - Do they form a minimum spanning tree? No!
  - Proof: (counterexample)
Why MSTs?

• **LOTS** of applications
  • **Network Design:**
    • Roads, TV cables, Electrical wires, etc.
  • **Approximations for (NP-) hard problems**
    • Travelling Salesperson
  • And many more!
Properties of MSTs

- Cut Property
- Cycle Property
Cuts

- A cut is any partition of the vertices in $G$ into two disjoint sets of vertices (denoted by $(A, B)$)
  - The vertices in each set don’t need to be connected to each other
  - This will come up again later! (~Lecture 18 on flows and cuts)
Cut Property

• The lightest (least weight) edge connecting the two sets of vertices in each cut must be in every MST
  • If there are multiple edges tied for the lowest weight then all MSTs must contain at least one of them
Cut Property

• **General Proof**: (contradiction)
  - Say a cut \((A, B)\) results in the two sets of vertices A and B. Say an MST includes an edge E going across the cut (connecting A and B).
  - If there exists a *lighter* edge \(E'\) going across the cut, then we would get a “better” MST by removing E and adding \(E'\) instead.
  - But this is a contradiction!
**Cut Property**

- **General Proof:** (contradiction)
  - Say a cut \((A, B)\) results in the two sets of vertices A and B. Say an MST includes an edge E going across the cut (connecting A and B).
  - If there exists a *lighter* edge \(E'\) going across the cut, then we would get a “better” MST by removing E and adding \(E'\) instead.
  - But this is a contradiction!
  - Minor details to think about:
    - Prove that replacing E with \(E'\) creates a valid tree
      - (Hint: prove it doesn’t create a cycle first)
Cycle Property

- The **heaviest** (most weight) edge in every cycle in $G$ cannot be in any MST
  - If there are multiple edges tied for the highest weight then all MSTs can contain at most all but one of them
Cycle Property

• **General Proof: (contradiction)**
  • Say that while constructing the MST we keep (all of the) heaviest edges in a cycle $C$ and remove one of the lighter edges $E'$ instead.
  • But then we will be able to construct a “better” MST by removing one of the heaviest edges and adding $E'$ back in!
  • But this is a contradiction!
Cycle Property

• **General Proof:** (contradiction)
  • Say that while constructing the MST we keep (all of the) heaviest edges in a cycle C and remove one of the lighter edges E′ instead
  • But then we will be able to construct a “better” MST by removing one of the heaviest edges and adding E′ back in!
  • But this is a contradiction!

• **Minor detail:**
  • Prove that replacing the heaviest edge forms a tree
  • General sketch:
    • Doing this doesn’t create a cycle
    • Keep number of edges the same
    • -> by Pigeonhole Principle we form a valid tree still
Using Cuts and Cycles to build MSTs

- Kruskal’s Algorithm
- Prim’s Algorithm
- Reverse-Delete Algorithm
- (and more!)
Kruskal’s Algorithm:

- *Greedy Algorithm!*
  - *Greedy Rule: Add the lowest-cost edge that doesn’t create a cycle*
  - *Which property discussed previously does Kruskal’s use?*
Kruskal’s Algorithm:

• Greedy Algorithm!
  • Greedy Rule: Add the lowest-cost edge that doesn’t create a cycle
  • Which property discussed previously does Kruskal’s use?
    • Uses both Cut and Cycle Properties!
Kruskal’s Execution:

Original Graph

Minimum Spanning Tree
Kruskal’s Execution:

Original Graph

Minimum Spanning Tree
Kruskal’s Execution:

Original Graph

Minimum Spanning Tree
Kruskal’s Execution:

Original Graph

Minimum Spanning Tree

Adds a cycle
Kruskal's Pseudocode:

- Let $w_e$ denote the weight of edge $e$.

Kruskals(V, E):
    sort E in non-decreasing order ($w_0 \leq w_1 \ldots \leq w_m$)
    Initialize each vertex in its own “island”
    for $i = 1 \ldots m$:
        let $e_i = (u, v)$
        if $u$ and $v$ are in different connected components:
            add $e_i$ into the MST
            merge the connected components containing $u,v$
    return the MST
Kruskal’s Pseudocode:

• Let $w_e$ denote the weight of edge $e$.

Kruskals(V, E):

  - sort E in non-decreasing order ($w_0 \leq w_1 \ldots \leq w_m$)
  - Initialize each vertex in its own “island”
  - for $i = 1 \ldots m$:
    - let $e_i = (u,v)$
    - if $u$ and $v$ are in different “islands”:
      - add $e_i$ into the MST
      - merge the “islands” containing $u$ and $v$
  - return the MST

How to do this efficiently?
(easy $O(n \log n)$ implementation)
Union Find!!!

- Both a data structure and an algorithm
- Runtime:
  - $O(\log n)$ for checking if two nodes are in the same group 😊
  - $O(\log n)$ for merging two groups 😊
Union Find

• For each node, keep track of two things:
  • Pointer to its “parent”
  • “Depth” of its tree (length of longest path ending at that node)
• All pointers initially uninitializd, “depth” = 0
Union Find

• To check whether A and B are part of the same “island”:
  • Follow the pointers up to the root of the tree, check if identical
Union Find

• To merge two “islands”:
  • First find the root of each tree
  • Assign the lower-depth root to point to the higher-depth root
    • If roots are the same depth tiebreak arbitrarily
  • Adjust the depths if necessary

0 1 2 3 4 5 6

Depth
Nodes
Union Find Example

- \text{Merge}(2, 6)
Union Find Example

- \text{Merge}(4, 1)
Union Find Example

- `CheckSame(1, 2)`
- `CheckSame(6, 2)`

![Diagram of Union Find Example]
Union Find Example

• Merge(5, 4)
Union Find Example

- $\text{Merge}(2, 4)$
Union Find Example

- `CheckSame(5, 1)`
- `CheckSame(6, 2)`
Union Find Runtime Proof:

- **Claim**: If the label of a node is \( k \), then there must be \( \geq 2^k \) elements in the tree
  - Equivalently, if there are \( n \) nodes in a tree, the depth of the tree is at most \( \log(n) \)

- **General Proof: (Induction)**
  - **Base case**: True initially
  - **Inductive step**: Each step we merge a tree of depth at most \( \log(n) \)
    - From inductive hypothesis it also must contain at most \( n \) elements
    - Depth increases by at most 1, number of elements can double
      - \( \rightarrow \) Inductive hypothesis holds!

- As a consequence, union find is guaranteed to be \( \log(n) \)
  - Or better! (See Tarjan’s 1975 paper for details if you want)
Kruskal’s Runtime:

Kruskals(V, E):

\( O(M \log M) \) sort \( E \) in non-decreasing order

\( O(N) \) Initialize each vertex in its own “island”

\( O(M \log N) \) for \( i = 1 \ldots m \):

\( O(\log N) \) if \( u \) and \( v \) are in different “islands”:

\( \text{add } e_i \text{ into the MST} \)

\( O(\log N) \) merge the “islands” containing \( u \) and \( v \)

\( \text{return the MST} \)

Overall: \( O(M \log M) = O(M \log N) \)
Kruskal’s Proof of Correctness:

• Add the lowest-cost edge that doesn’t create a cycle

→ Equivalently:
  • If adding $e$ to $T$ creates a cycle, then delete it according to the cycle property
  • Otherwise, add it according to the cut property
Other MST greedy algorithms:

- Prim’s Algorithm: Similar to Dijkstra’s Algorithm
  - Start from an arbitrary vertex
  - At each step add the lowest-weight edge coming out of the tree
  - Straightforward application of cut property
- Reverse-Delete:
  - Keep deleting the highest-weight edge unless it disconnects the graph
  - (Somewhat) straightforward application of cycle property