GR3: Minimum Spanning Trees

Notes for CS-8803-GA: Introduction to Graduate Algorithms

Georgia Tech (Dr. Eric Vigoda), Fall 2017

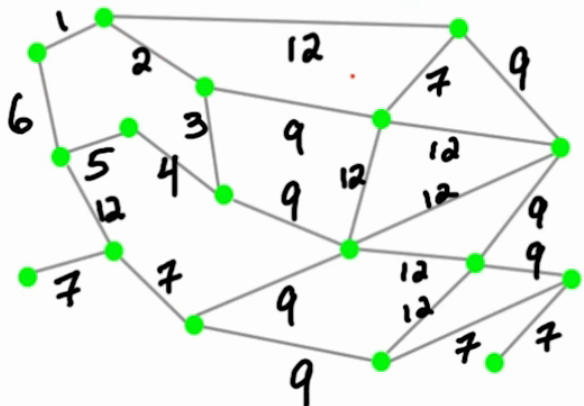
as recorded by Brent Wagenseller

Greedy Strategies

* Greedy strategies take the choice which looks locally optimal.
  + The problem is, what if choosing these local optimums does NOT lead to the **global optimum**?
* We can think of a greedy algorithm as: given the choices we have made so far, make the choice that provides the most benefit now.
* These work well for computing the minimum spanning tree of a graph.

Properties of Trees

* A tree is a connected, acyclic graph
* a tree on n vertices has exactly n-1 edges.
* In a tree, exactly one path between every pair of vertices
* Any connected G=(V, E) with |E| = (|V| - 1) is a tree
  + This is what will be used heavily in the MST
* NOTE: it appears that, for MST, the original graph does NOT have to be a tree, but the output from the MST is certainly a tree. Observe the example given in the lecture and its clear the original graph is not an acyclic tree:



Proof Of Spanning Trees

**Claim:**A minimal size subset of edges F\subseteq E that connects every pair of vertices is a spanning tree. That is,

1. Fhas no cycles
2. Fhas |V|-1=n-1edges; i.e. |F|=n-1.

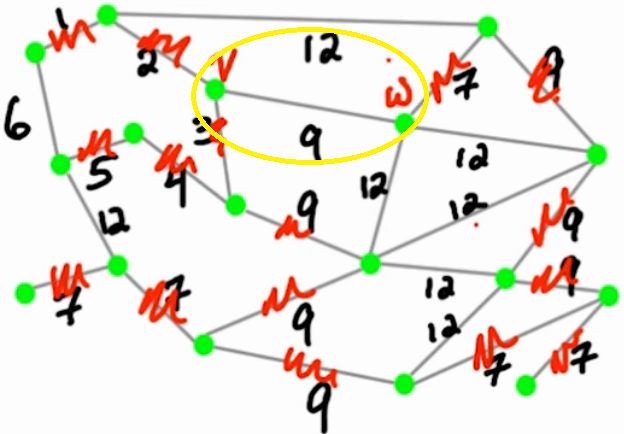
**Proof: (1)**Suppose Fhad a cycle. Then, we can remove any edge on the cycle while maintaining connectivity between every pair of vertices. Therefore, there can be no cycles if Fis minimal.

**(2)** We prove by induction that |F|=n-1. It’s clearly true for the base cases n=1,2. Now, for n>2, remove any edge in F. Then, F \backslash \{e\}has 2components (otherwise Fwas not minimal), and each component has n_1, n_2<nvertices with n_1 + n_2 = n. By induction,

|F| = (n_1-1) + (n_2-1) + 1 = n-1.

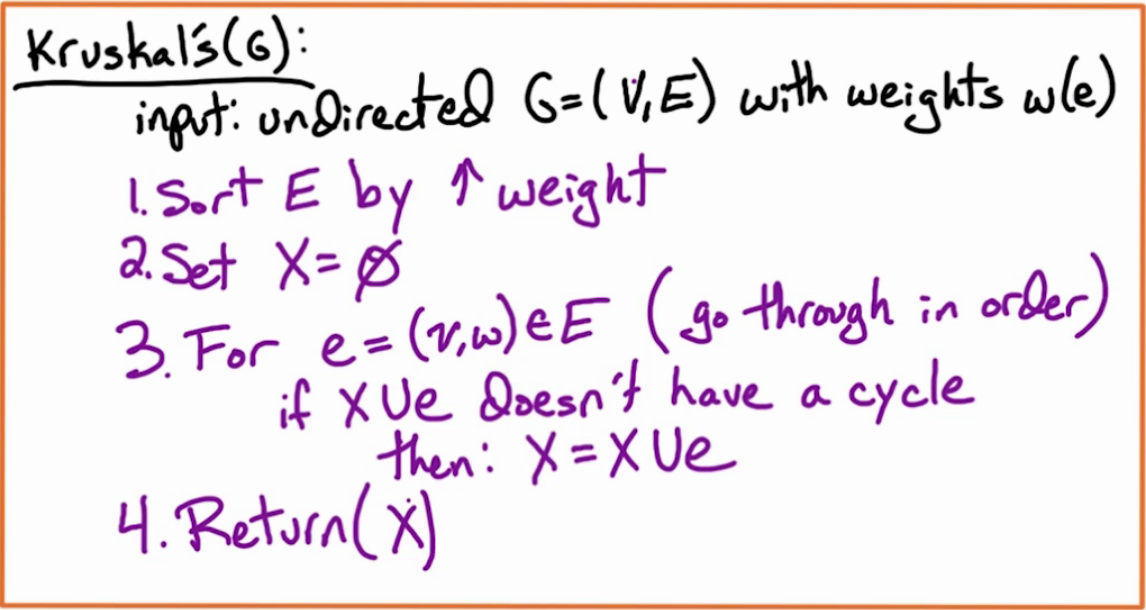
Basics of the MST

* Basic Layout
  + **Input:** An undirected graph G=(V,E), |V|=n, |E|=m with E=\{(u_1,v_1), \ldots, (u_m, v_m)\} where each u_i, v_i \in V. We also have weights on all the edges, given by w: E \rightarrow \mathbb{R}.
  + **Problem:**Find a subset of edges, vertices (i.e. a subgraph) of Gof minimum total weight that satisfies property P.
    - Property P could be one of many possibilities, e.g., “cycle” or “path” or “spanning”.
    - Some will have efficient solutions, others will not.
* In a MST, each edge now has a weight; we want to minimize the total weight of any given path in the undirected graph G=(V, E).
  + NOTE: these are absolutely trees, NOT forests!
* The general greedy algorithm for the MST orders ALL edges by weight FIRST, checking (and potentially adding) the lowest weights first to the graph, then cycles through all the rest of the edges
* If adding the edge would create a cycle, it is NOT added
* Example



* + In the above example, ALL edges with 1 are considered first; then 2, etc etc
  + When we get to 6, we cannot add that edge in as it creates a cycle
  + When we add in 9, we add almost all of them in, but when we get to the edge circled in yellow we cannot as it would create a cycle
  + NONE of the size 12 edges can be added in
  + This is the solved minimum spanning tree for this example; it’s a tree of the graph that has no cycles and used the minimum weights possible for edges
    - this uses a greedy approach because it takes the lowest weighted edges first, regardless of the distant future outcome

**Kruskal’s Algorithm for MST:**

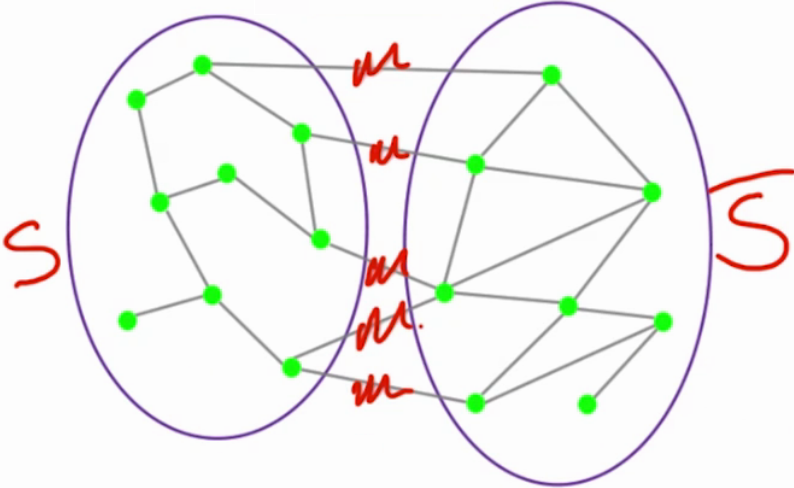


* Kruskal’s Algorithm is the algorithm mentioned in the previous example
* We consider the lower weights first, THEN the higher weights (in order)
* To determine if the edge exists, we use G=(V, X) and we figure out the components of v and w
  + ‘X’ keeps track of the edges used thus far (this is important to track if adding an edge would create a cycle).
* When X is returned, it’s a minimal spanning tree of the original graph!
* To check for cycles, we check for component numbers in G(V,X)
  + Initially, all vertices in this graph are in their own component as there are no edges
  + As edges are added the overall component counts start to decrease until eventually there is only one component (so long as the original graph was not a forest)
  + IF the component number (ccnums) are the same for v and w, they are in the same component so do not add that edge!
    - This check is done via the Union-Find data structure
      * The Union-Find data structure is in the text, but it uses rooted directed trees
  + If they are NOT the same, the Union-Find data structure merges the two components

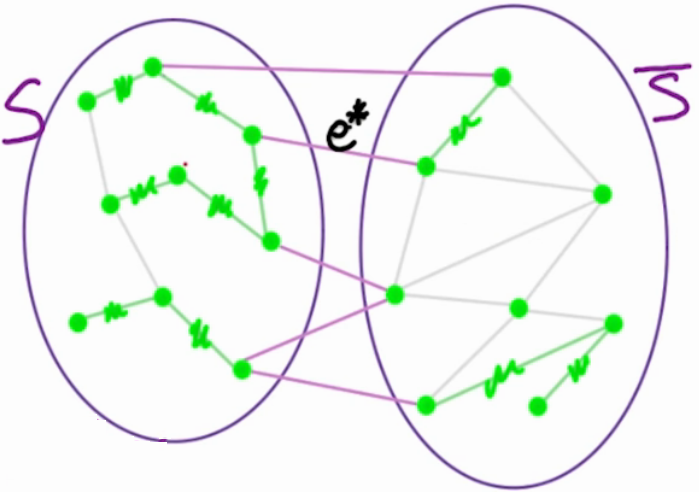
**Runtime of Kruskal’s algorithm**

* The runtime of the edge sort is O(m log n) (where m = |E| and n = |V|).
* To check the components it takes O(log n) time, as that’s the search time for the Union-Find data structure
  + since we are doing m operations, this also runs in O(m log n) time
* The entire algorithm runs in O(m log n) time

Cuts



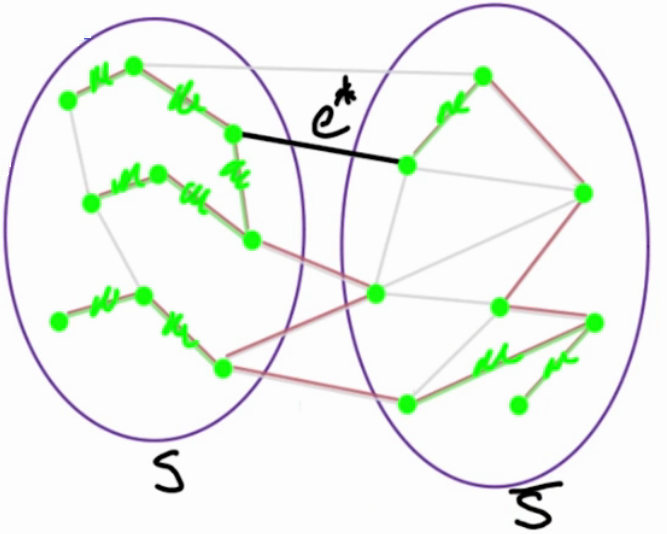
* Given graph G, a **cut** of the graph partitions the set of vertices V into two sets, (S,\bar S) (aka S and its compliment).
* Define a cut (S,\bar S)as the set of edges between a subset of vertices S \subseteq Vand its complement \bar S = V \backslash S. That is,
  + That is, (S,\bar S) = \{(u_i, v_j) \in E | u_i \in S, v_j \in \bar S\}.
    - In words, the **cut edges** are the edges crossing between S and its complement (S bar); see above example (edges scribbled over in red)
* Lemma: we can add edge e\* to our MST and X will still be a MST
  + For an undirected graph G=(V, E), take a subset of edges X where X is constitutes a MST called T (as above)
    - Mid-stream of the algorithm, X is a ‘partial’ solution, as we assume (by induction) that its correct so far
      * We are confident it’s a MST T, although we do not know what T is just yet
        + The graph above is a MST (as we have seen previously), although halfway through the algorithm we do not fully know this yet
        + Note that there can be multiple versions of T, all of which are valid MSTs for the graph G; we may use any one of them as the target T
  + We are assuming that the edge set we are adding to the graph is a cut of the graph
    - The assumption is that what we are adding to X, IF we are adding to X, is in fact a cut edge that will merge two connected components (S,\bar S).
  + We now look at all of the edges of this cut, and we find the minimal cut (we will call this **e\***):



* + - The assumption is the other edges have at least the weight of e\*
    - Thus, we can add e\* to X and X will still be a MST
      * Mathematically: X ∪ e\* ⊂ T` where T` is a MST
  + Any edge which is minimum weight across a cut is going to be a part of some MST
    - If we have MST T and T does not have the cut in question, we can add the edge and we will have a new tree
  + Extrapolating to the MST problem, the MST may have different Ts, but none of the different sums of weights (over edges) will be greater than that of T
    - If adding e\* would mean the overall weight of T` would be more than T, this would mean that T` is not a solution to the MST and would therefore be incorrect
      * This will not happen if we select e\* available (which by definition is the lightest edge)
    - This means, ultimately, adding e\* is ok

Highlights in Proving Kruskal’s Algorithm

* The proof of this uses induction, and seems to revolve around the fact that the two vertices (and their subgraphs, (S,\bar S)) in X about to be connected are either
  + Parts of two separate trees in a forest; it is ok to join them because they are two different connected components.
  + Part of the same tree, in which case the vertices in question would NOT be joined as they would be part of the same component and this thus the edge would not be considered for addition to the graph X
* If e\* is in T, its easy to prove that we will get T, because e\* is a part of T
* If e\* is in T` but not T
  + Observe this graph:



* + - The green is X; the red is T (for the point of the proof, its unknown), and the edge e\* is what we are adding
    - We know that e\* is not part of the solution – but if its equivalent to the other edges in the cut, it could be a part of the solution if it replaces an edge cut
      * ultimately any edge it would replace would have an equivalent weight, otherwise it wouldn’t even be considered
* Ideas Dr. Vigoda wishes to stress
  + **We can take a tree T and we can add an edge into the tree T ∪ e\*, but this creates a cycle; HOWEVER we can remove ANY edge of that cycle and we will get a new tree, T`**
  + **A minimum weight edge across a cut is part of a MST**
  + We should understand why Primm’s algorithm is correct by using the cut property (Primm’s algorithm not presented in lecture)

**Theorem(Kruskal):**Kruskal’s algorithm produces a minimum weight spanning tree of G=(V,E).

**Lemma 1:**Fix any edge ewhose weight is the minimum in some cut (S,\bar S). Then, there exists an MST Tcontaining e.

**Proof:**Suppose eis not in an MST. Consider an MST Tand add eto T. Then, einduces a cycle C(since T \cup \{e\}has two paths between endpoints of e).

Consider the cut (S,\bar S)for which eis a minimum weight edge. Then Cmust cross (S,\bar S)in some other edge fwith weight(e) \le weight(f).

Set T' = T \cup \{e\} \backslash \{f\}. Note that weight(T') \le weight(T), which means T'is an MST (since it connects every pair of vertices and is minimum weight).

**Lemma 2:**Let T_Kbe the tree found my Kruskal’s algorithm. For each edge e \in T_K, there exists a cut (S, \bar S)such that

1. eis the minimum weight edge of (S, \bar S)
2. eis the unique edge in T_K \cap (S, \bar S).

**Proof:**Consider T_K \backslash \{e\}, with e = (u,v). Then, u,vare in different components (S, \bar S)and eis the only edge of (S, \bar S)in T_K. Also, at the point when ewas added, uand vwere not connected, which means that eis the minimum weight edge on (S, \bar S)(otherwise, Kruskal’s algorithm would have added another edge before e).

**Proof (of Theorem):**Using Lemmas 1 and 2, we can now prove that Kruskal’s algorithm always produces an MST.

Suppose T_Kis not an MST, and let Tbe an MST. Order the edges of T_Kby increasing weight. Take the first edge eof T_Kthat is not in T. Thus T \cup \{e\}has a cycle C. Let (S, \bar S)be the cut for egiven by Lemma 2. Now, remove f \in C \cap Tthat crosses (S, \bar S. We have that weight(e) \le weight(f)and T \cup \{e\} \backslash \{f\}is a spanning tree.

Repeat this process for all edges of T_Knot in T, and thus T_Kis an MST.

The key idea of the proof is sometimes called the *cut property*.

**Lemma 3 (cut property).**Let Xbe a subset of edges of an MST of a graph G. For any edge esuch that (1) eis a minimum-weight edge of a cut (S, \bar{S})and (2) X \cap (S, \bar{S}) = \phi, there is an MST Ts.t. X \cup \{e \} \subseteq T.

Using the cut property, we can prove that the following algorithm, called Prim’s algorithm, also produces an MST.

**Running time of Kruskal’s algorithm:**O(m \log n).

Sorting the edges once up front takes O(m \log n).

However, we also have to keep track of whether an added edge creates a cycle, i.e. when an edge connects two vertices in the same connected component. We can do this via a simple data structure.

For each vertex, introduce a component number, which is initially just the index of that vertex. When Kruskal’s algorithm adds an edge which connects two vertices with the same component number, we skip that edge because it creates a cycle. If the component numbers of the two vertices are different, then we relabel the vertices in the *smaller* component to the label of the *larger* component. Since we relabel the smaller component, the size of a vertex’s component at least doubles each time it is relabeled. Therefore a vertex’s label changes at most O( \log n)times. Thus the total cost of this data structure is O(n \log n).

**Prim’s Algorithm:**

Another algorithm to find the MST of a graph is Prim’s algorithm.

* Start with any vertex, and mark it as visited.
* Add the lightest weight edge from some visited vertex to an unvisited vertex; repeat until all vertices are visited.

As with Kruskal’s algorithm, its correctness follows from the **cut property** given above.

**Running time of Prim’s algorithm:**O(m \log n).