The Minimum Spanning Tree Problem

Minimum Spanning Tree (MST) - Introduction

- Informally,
  - We have some locations and we would like to build a transportation line such that it is least costly and can connect every location to one another -> connected.

Minimum Spanning Tree (MST) - Introduction

3

- Formally,
  - For certain pairs (v_i, v_j), we may build a direct link between v_i and v_j for a certain cost $c(v_i, v_j) > 0$.
  - Thus we can represent the set of possible links that may be built using a graph $G = (V, E)$, with a positive cost c_e associated with each edge $e = (v_i, v_j)$.
  - The problem is to find a subset of the edges $T \subseteq E$ so that the graph**/tree** (V, T) is connected, and the total cost $\sum_{e \in T} c_e$ is as small as possible.
  - **Goal:** find the cheapest spanning tree.

Application example

4

One example would be a telecommunications company laying cable to a new neighborhood. If it is constrained to bury the cable only along certain paths (e.g. along roads), then there would be a graph representing which points are connected by those paths.

Some of those paths might be more expensive, because they are longer, or require the cable to be buried deeper; these paths would be represented by edges with larger weights.

A spanning tree for that graph would be a subset of those paths that has **no cycles** but still connects to every house; there might be several spanning trees possible. A minimum spanning tree would be one with the lowest total cost, thus would represent the least expensive path for laying the cable.

Exam PYQ

5

Once upon a time there was a city that had no roads. Getting around the city was particularly difficult after rainstorms because the ground became very muddy cars got stuck in the mud and people got their boots dirty. The mayor of the city decided that some of the streets must be paved, but did not want to spend more money than necessary because the city also wanted to build a swimming pool. The mayor therefore specified two conditions:

1. Enough streets must be paved so that it is possible for everyone to travel from their house to anyone else's house only along paved roads, and

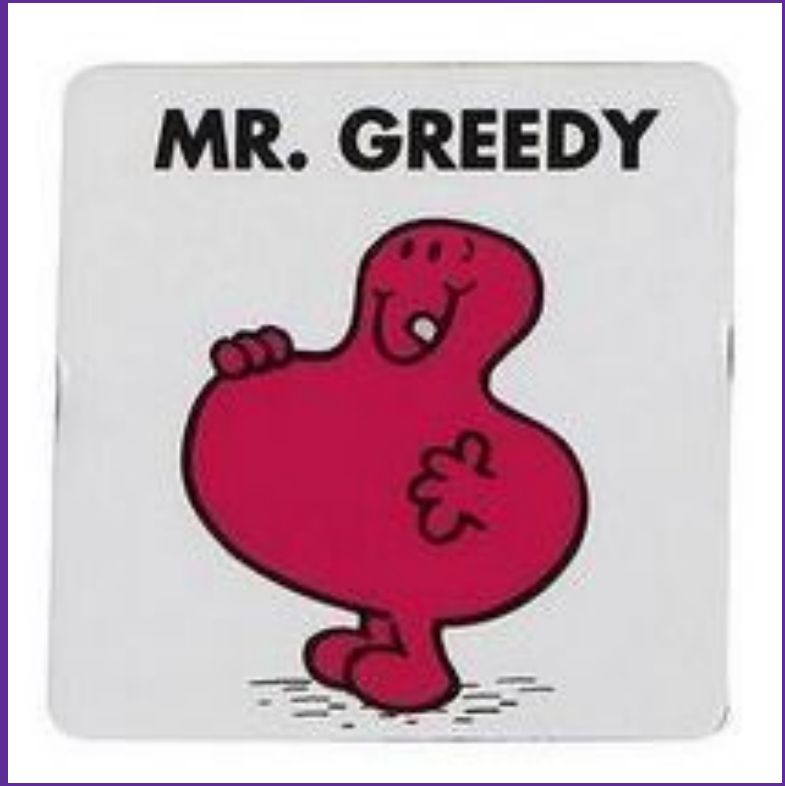
2. The paving should cost as little as possible.

Continued…

(V, T) is a tree

- Should be connected + no cycles
- Connected -> by definition
- No cycle -> proof by contradiction
- Suppose it contained a cycle C, and let e be any edge on C. We claim that (V , T − {e}) is still connected, since any path that previously used the edge e can now go “the long way” around the remainder of the cycle C instead. It follows that (V , T − {e}) is also a valid solution to the problem, and it is **cheaper**—a contradiction.

Can you think of some greedy approaches to build a MST?



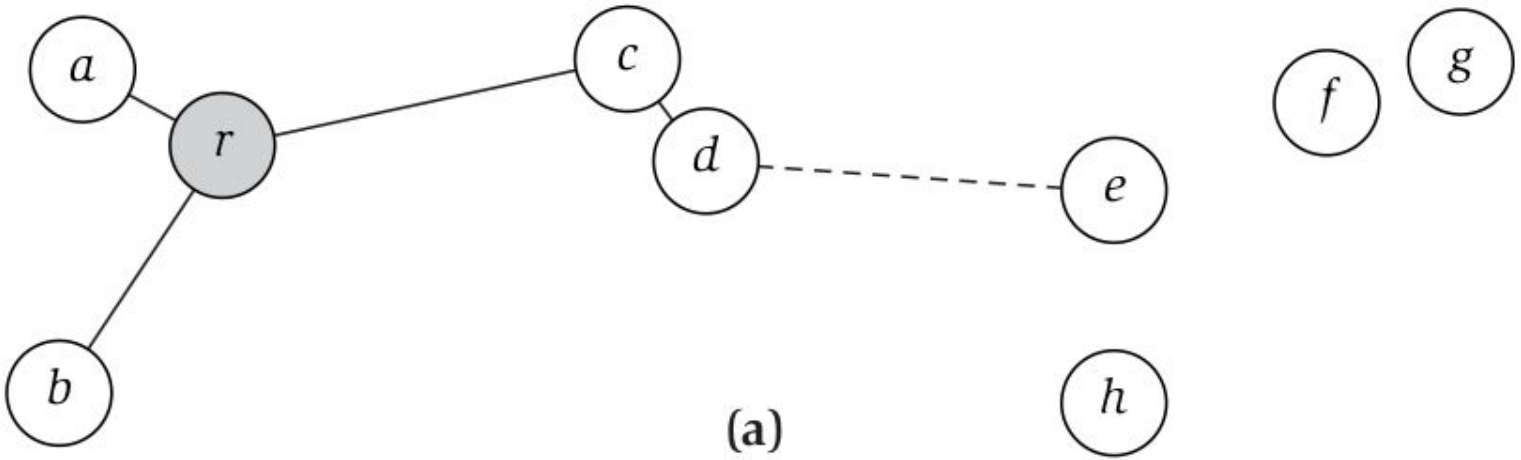
MST Algorithms - Greedy Approaches

- **Approach 1:** insert edges from E in order of increasing cost => include the edge first which has a lesser cost as long as no cycle are created. [**Kruskal's Algorithm**]

- **Approach 2:** (Similar to Dijkstra's) Include edge 'e' to the partial tree starting at s if the new node can be attached as cheaply as possible. [**Prim's Algorithm**]

- **Approach 3:** Reverse of kruskal. Take complete graph and start removing edges with the largest cost as long as the graph is connected. [**Reverse-Delete Algorithm**]

Example



cost of each edge is proportional to the geometric distance in the plane.

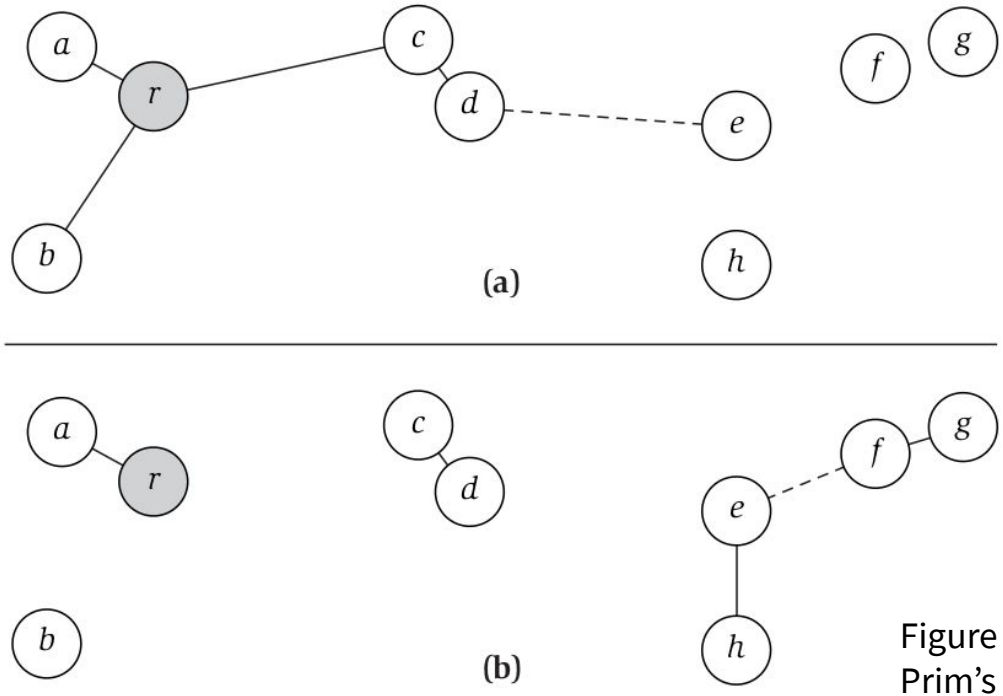


Figure 4.9 shows the first four edges added by Prim's and Kruskal's Algorithms respectively, on a geometric instance of the Minimum Spanning Tree Problem in which the **cost of each edge is proportional to the geometric distance in the plane**.

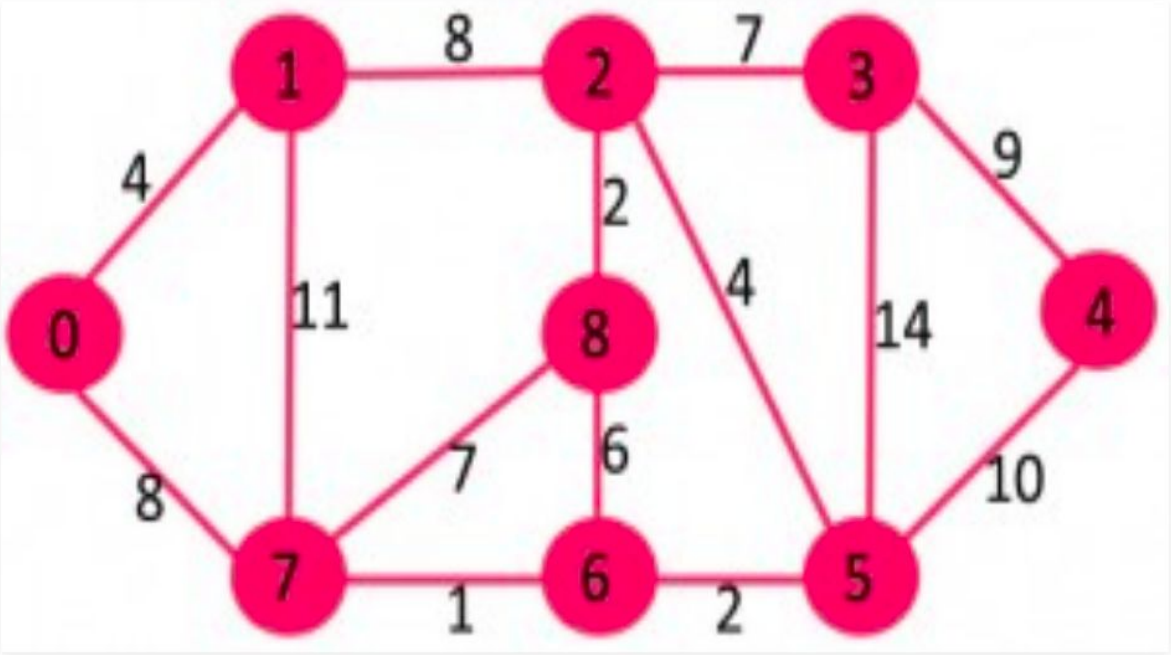
Kruskal's Algorithm

Starts without any edges at all and builds a spanning tree by successively inserting edges from E in order of increasing cost.

As we move through the edges in this order, we insert each edge 'e' as long as it does not create a cycle when added to the edges we've already inserted.

If, on the other hand, inserting 'e' would result in a cycle, then we simply discard 'e' and continue.

Kruskal's Algorithm - Example

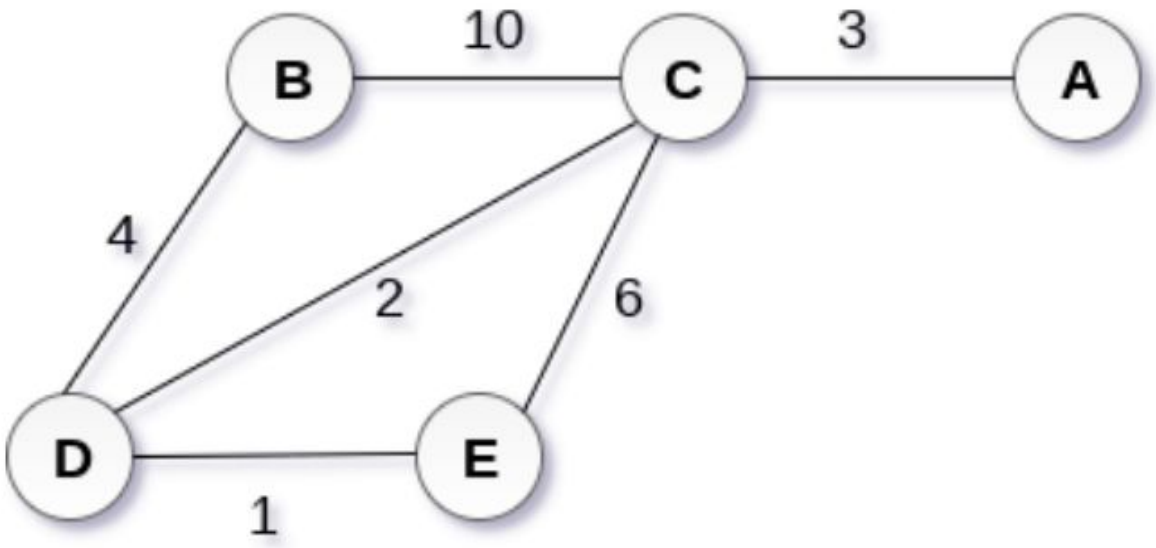


Prim's Algorithm

- We start with a root node s and try to greedily grow a tree from s outward. At each step, we simply add the node that can be attached as cheaply as possibly to the partial tree we already have.
- Maintain a set $S \subseteq V$ on which a spanning tree has been constructed so far. Initially, $S = \{s\}$. In each iteration, we grow S by one node, adding the node v that minimizes the "attachment cost" $\min_{e=(u,v):u \in S} c_e$, and including the edge e = (u, v) that achieves this minimum in the spanning tree.

Prim's Algorithm - Example

Although rare, if starting vertex not given, choose one randomly and **MENTION** it in your solution.



14

Note

The algorithms work by repeatedly inserting edges to a partial solution.

Big question:

When Is It Safe to Include an Edge in the Minimum Spanning Tree?

Answered by the “Cut Property”.

Cut Property - Poll time

16

Assume that all edge costs are distinct. Let S be any subset of nodes that is neither empty nor equal to all of V, and let edge e = (v, w) be the minimum-cost edge with one end in S and the other in V − S.

Then, ‘e’ will be:

A. Never part of MST
B. Might be a part of MST
C. Always part of MST
D. Not sure.

Proof

Let T be a spanning tree that does not contain e; we need to show that T does not have the minimum possible cost.

We'll do this using an exchange argument: we'll identify an edge e' in T that is more expensive than e, and with the property exchanging e for e' results in another spanning tree. This resulting spanning tree will then be cheaper than T, as desired.

How can you identify this edge e'?

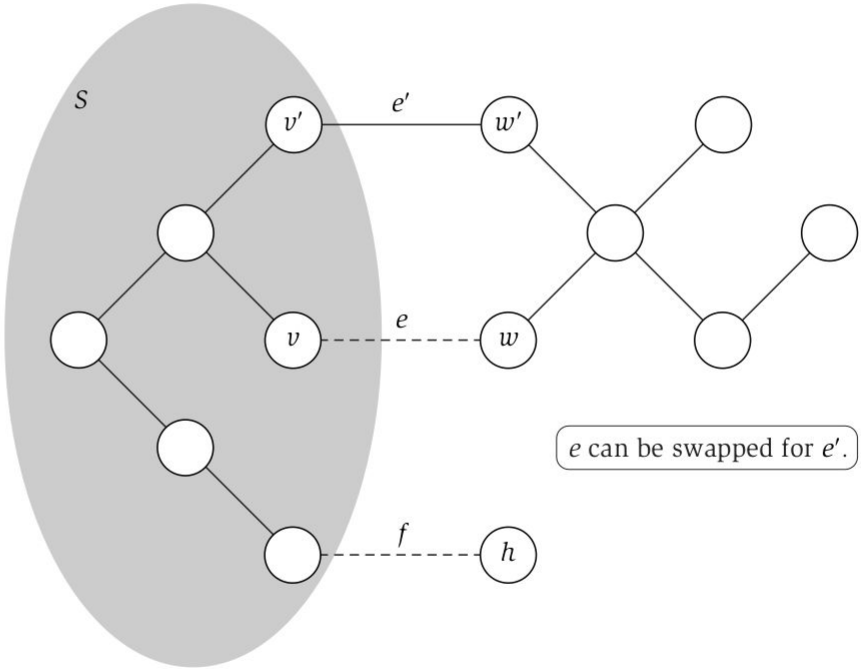
The edge “e’”

The crux is therefore to find an edge that can be successfully exchanged with e.

Recall that the ends of e are v and w. T is a spanning tree, so there must be a path P in T from v to w.

Starting at v, suppose we follow the nodes of P in sequence; there is a first node w’ on P that is in V − S. Let v’ ∈ S be the node just before w’ on P, and let e’ = (v’ , w’) be the edge joining them.

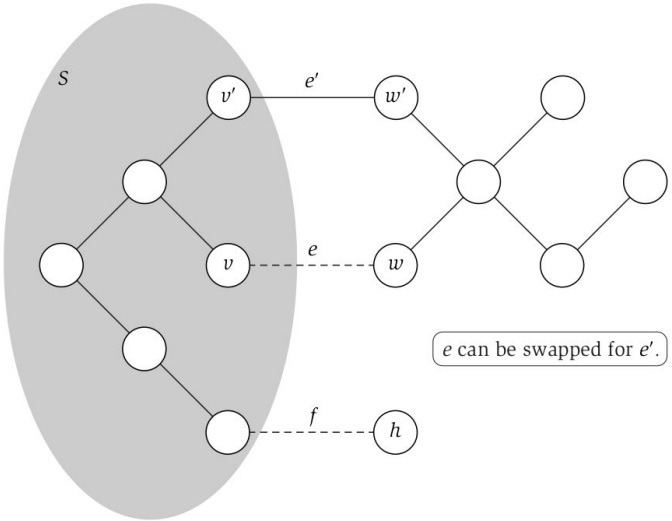
Thus, e’ is an edge of T with one end in S and the other in V − S.



Exchange e for e'

If we exchange e for e', we get a set of edges $T' = T - \{e'\} \cup \{e\}$. We claim that T' is a spanning tree. Clearly (V, T') is connected, since (V, T) is connected, and any path in (V, T) that used the edge $e' = (v', w')$ can now be "rerouted" in (V, T') to follow the portion of P from v' to v, then the edge e, and then the portion of P from w to w'. To see that (V, T') is also acyclic, note that the only cycle in $(V, T' \cup \{e'\})$ is the one composed of e and the path P, and this cycle is not present in (V, T') due to the deletion of e'.

We noted above that the edge e' has one end in S and the other in $V - S$. But e is the cheapest edge with this property, and so $c_e < c_{e'}$. (The inequality is strict since no two edges have the same cost.) Thus the total cost of T' is less than that of T, as desired. ■



19

Optimality of Kruskal and Prim

The point is that both algorithms only include an edge when it is justified by the Cut Property

Optimality of Kruskal's Algorithm

To prove: Kruskal's Algorithm produces a minimum spanning tree of G.

Part 1: Prove that every edge added by Kruskal's is safe

Part 2: The result (V, T) is a spanning tree.

Consider any edge e = (v, w) added by Kruskal's Algorithm, and let S be the set of all nodes to which v has a path at the moment just before e is added. Clearly $v \in S$, but $w \notin S$, since adding e does not create a cycle. Moreover, no edge from S to $V - S$ has been encountered yet, since any such edge could have been added without creating a cycle, and hence would have been added by Kruskal's Algorithm. Thus e is the cheapest edge with one end in S and the other in $V - S$, and so by Cut-Property it belongs to every minimum spanning tree.

Optimality of Kruskal's Algorithm

To prove: Kruskal's Algorithm produces a minimum spanning tree of G.

Part 1: Prove that every edge added by Kruskal's is safe

Part 2: The result (V, T) is a spanning tree.

Clearly (V , T) contains no cycles, since the algorithm is explicitly designed to avoid creating cycles. Further, if (V , T) were not connected, then there would exist a nonempty subset of nodes S (not equal to all of V) such that there is no edge from S to V − S. But this contradicts the behavior of the algorithm: we know that since G is connected, there is at least one edge between S and V − S, and the algorithm will add the first of these that it encounters.

Optimality of Prim's Algorithm

To prove: Prim's Algorithm produces a minimum spanning tree of G.

Homework 2

Is it necessary that all edge costs be distinct?

- Simply take the instance and perturb all edge costs by different, extremely small numbers, so that they all become distinct.
- any two costs that differed originally will still have the same relative order, since the perturbations are so small; and since all of our algorithms are based on just comparing edge costs, the perturbations effectively serve simply as “tie-breakers” to resolve comparisons among costs that used to be equal.
- **Using this argument, can you claim that the spanning tree resulting after these perturbations is still a MST?**

Implementing Prim's Algorithm

- We need to be able to decide which node v to add next to the growing set S, by maintaining the attachment costs $a(v) = \min_{e=(u,v):u \in S} c_e$ for each node $v \in V - S$.
- keep the nodes in a priority queue with these attachment costs a(v) as the keys; we select a node with an ExtractMin operation, and update the attachment costs using ChangeKey operations.
- There are $n - 1$ iterations in which we perform ExtractMin , and we perform ChangeKey at most once for each edge.
- Using a priority queue, Prim's Algorithm can be implemented on a graph with n nodes and m edges to run in O(m) time, plus the time for n ExtractMin , and m ChangeKey operations resulting in an effective O(mlogn) complexity.

Implementing Kruskal's Algorithm

Algorithm

Sort edges so that c1<c2<c3<c4<.....<cm

T = ϕ

For i = 1 to m

    If T U {i} has no cycles

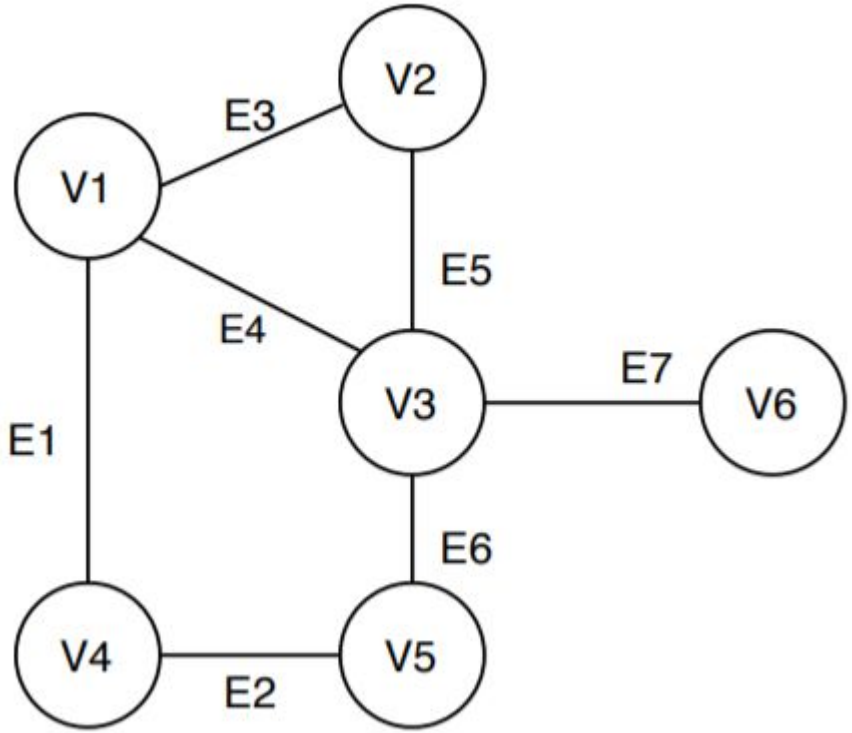
        Add i to T

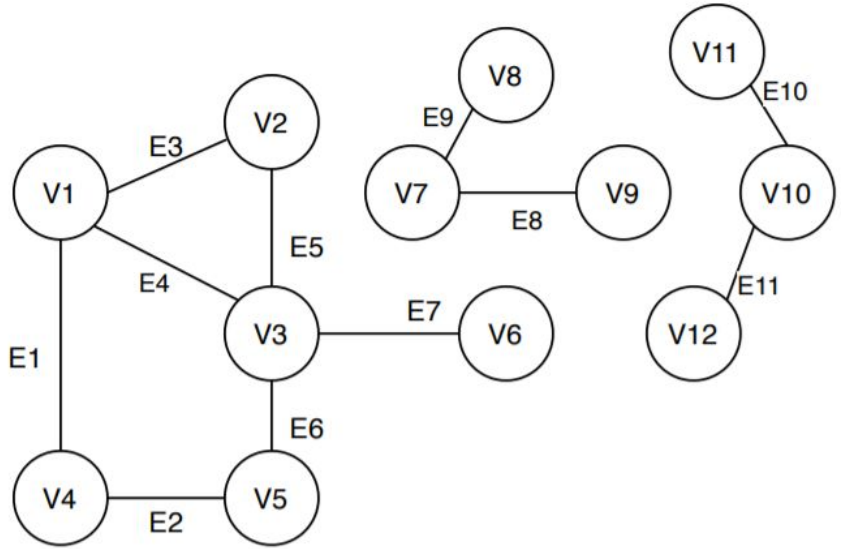
Return T

Connected Components

- A connected component or simply component of an undirected graph is a subgraph in which each pair of nodes is connected with each other via a path.
- Let’s try to simplify it further, though. A set of nodes forms a connected component in an undirected graph if any node from the set of nodes can reach any other node by traversing edges. The main point here is reachability.
- In connected components, all the nodes are always reachable from each other.
- More detail later in last unit of this course.

Connected Component Example





Implementing Kruskal's Algorithm

- Need of a new data structure: Union-Find (Why?)
- **Essence:** As each edge e = (v, w) is considered, we need to efficiently find the identities of the connected components containing v and w.
- If these components are different, then there is no path from v and w, and hence edge e should be included;
- but if the components are the same, then there is a v-w path on the edges already included, and so e should be omitted. In the event that e is included, the data structure should also support the efficient merging of the components of v and w into a single new component.

Union-Find Data Structure

Given a node u, the operation

- **MakeUnionFind(S)** for a set S will return a Union-Find DS where all elements are in separate sets.
- **Find (u)** will return the name of the set containing u. This operation can be used to test if two nodes u and v are in the same set, by simply checking if Find (u) = Find (v).
- The data structure will also implement an operation **Union (A, B)** to take two sets A and B and merge them to a single set.

A simple Union-Find DS

Let S be a set, and assume it has n elements denoted {1, . . . , n}. We will set up an array Component of size n, where Component [s] is the name of the set containing s.

- MakeUnionFind (S), we set up the array and initialize it to Component [s]= s for all s ∈ S. (TC?)
- This implementation makes Find (v) easy: it is a simple lookup and takes only O(1) time.
- However, Union (A, B) for two sets A and B can take as long as O(n) time, as we have to update the values of Component [s] for all elements in sets A and B.

Can we make optimizations?

A simple Union-Find DS

Optimization 1: choose the name for the union to be the name of one of the sets, say, set A: this way we only have to update the values Component [s] for s ∈ B, but not for any s ∈ A.

What will happen if size(B) >> size(A)?

Optimization 2: When set B is big, we may want to keep its name and change Component [s] for all s ∈ A instead. More generally, we can maintain an additional array size of length n, where size [A] is the size of set A, and when a Union (A, B) operation is performed, we use the name of the larger set for the union. This way, fewer elements need to have their Component values updated.

TC: Still, O(n)

Think like a vertex

Thinking like a vertex

If you are a vertex, how many times will your component name change?

Recall, the name of your component only updates when you are in the smaller set.

Options:

A. O(1)
B. O(logn)
C. O(n)

Thinking like a vertex

If you are a vertex, how many times will your component name change?

Recall, the name of your component only updates when you are in the smaller set.

Options:

A. O(1)
B. O(logn)
C. O(n)

Summary

The array implementation of the Union-Find data structure for some set S of size n, where unions keep the name of the larger set.

- The Find operation takes O(1) time,
- MakeUnionFind (S) takes O(n) time,
- and any sequence of k Union operations takes at most O(k log k) time.

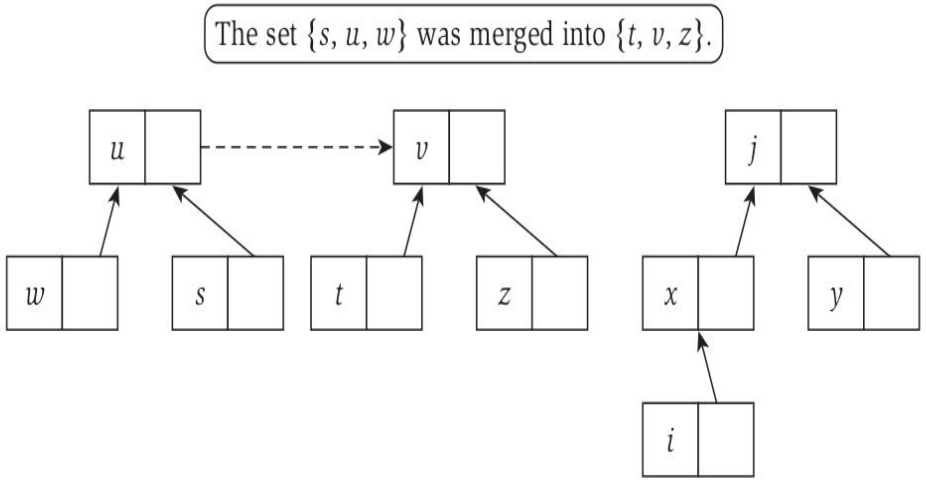
A less simple DS for Union-Find

- Uses pointers
- Each node v ∈ S will be contained in a record with an associated pointer to the name of the set that contains v.
- For the
- MakeUnionFind (S) operation, we initialize a record for each element v ∈ S with a pointer that points to itself (or is defined as a null pointer), to indicate that v is in its own set.
- Consider a Union operation for two sets A and B, and assume that the name we used for set A is a node v ∈ A, while set B is named after node u ∈ B. The idea is to have either u or v be the name of the combined set; assume we select v as the name. To indicate that we took the union of the two sets, and that the name of the union set is v, we simply update u's pointer to point to v. We do not update the pointers at the other nodes of set B.

Updating Pointers for Union

As a result, for elements w ∈ B other than u, the name of the set they belong to must be computed by following a sequence of pointers, first leading them to the “old name” u and then via the pointer from u to the “new name” v.

This pointer-based data structure implements Union in O(1) time: all we have to do is to update one pointer.
But a Find operation is no longer constant time, as we have to follow a sequence of pointers through a history of old names the set had, in order to get to the current name.



Pointer based union-find

Pointer-based implementation of the Union-Find data structure for some set S of size n, where unions keep the name of the larger set.

- A Union operation takes O(1) time,
- MakeUnionFind (S) takes O(n) time,
- and a Find operation takes O(log n) time.

Time complexity of Kruskal using Union-Find

- Sort the edges in increasing order of costs => O(mlogn)
- After the sorting operation, we use the Union-Find data structure to maintain the connected components of (V , T) as edges are added. As each edge e = (v, w) is considered, we compute Find (u) and Find (v) and test if they are equal to see if v and w belong to different components. We use Union (Find (u), Find (v)) to merge the two components, if the algorithm decides to include edge e in the tree T.
- #find operations = 2m
- #union operations = n-1
- Total complexity of Kruskal's using Union-Find DS is: O(mlogn)