Minimum Spanning Trees: Prim's and Kruskal's Algorithms

Step 1

Imagine you are the engineer responsible for setting up the intranet of UC San Diego: given a set of locations ***V*** and a set of costs ***E*** associated with connecting pairs of locations with cable, your task is to determine the networking layout that minimizes the overall cost of a network where there exists a path *from* each location *to* each location. If your design has even a single pair of locations that cannot reach one another, this can be problematic in the future because individuals at either location will not be able to send each other files. If your design does not minimize the overall cost of such a network, you will have wasted valuable money that could have been spent elsewhere. As such, these two requirements are quite strict.

Notice that we can trivially transform the Network Design Problem above into a formal computational problem that looks quite similar to other problems we solved previously in this chapter. We can represent UC San Diego as a **graph** in which the locations are represented as **nodes** (or **vertices**) and the pairwise location connection possibilities are represented as **undirected weighted edges**, where the weight associated with an edge is simply the cost of the cables needed to connect the two locations connected by the edge.

With this graph representation, the Network Design Problem transforms into the formal computational problem: given a graph ***G***, defined by a set of vertices ***V*** and a set of edges ***E***, return a collection of edges such that the following two constraints are maintained:

- For each pair of vertices in the graph, there exists a path, constructed only from edges in the returned collection, that connects the two vertices in the pair
- The sum of the weights of the edges in the returned collection is minimized

Notice that, since it would be redundant to have multiple paths connecting a single pair of **vertices** (if the paths are equal weight, you don't need to keep them all; if not, you would only want to keep the shortest-weight path), our **subgraph** that we return must not contain any cycles. In other words, our subgraph will be a **tree**. Specifically, we call a tree that hits all nodes in a graph *G* as a **Spanning Tree** of *G*. Because we want the Spanning Tree with the smallest overall cost, we want to find a **Minimum Spanning Tree (MST)** of *G*.

In this section, we will discuss two algorithms for finding **Minimum Spanning Trees**: **Prim's Algorithm** and **Kruskal's Algorithm**.

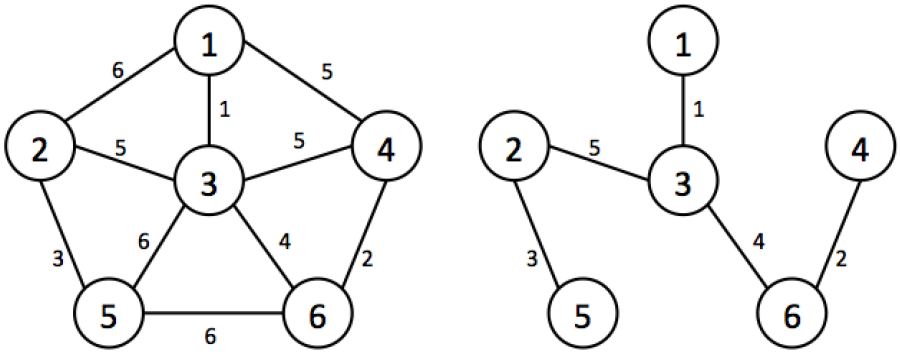
Step 2

A "**spanning tree**" is a **tree** (what a surprise) that *spans* across the entire graph (i.e., for any pair of nodes *u* and *v* in the graph *G*, there exists some path along the edges in the MST that connects *u* and *v*).

More formally, a **spanning tree** for an **undirected graph** *G* is an **undirected graph** that

- contains all the vertices of *G*
- contains a subset of the edges of *G*
- has no cycles
- is connected (this implies that only connected graphs can have spanning trees)

Below is an example of a graph with its spanning tree:

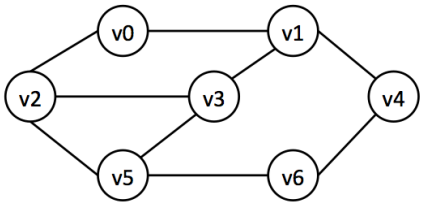


Note that a **spanning tree** with *N* vertices will always have *N-1* edges, just like any other tree!

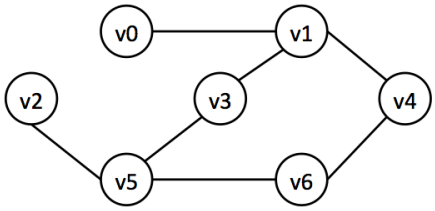
STOP and Think: Can a graph have more than one **spanning tree** that fulfills the four **spanning tree** properties mentioned above?

Step 3

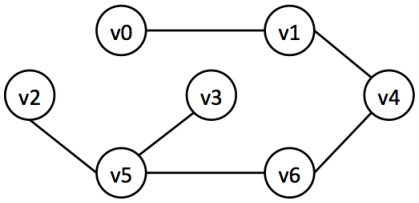
EXERCISE BREAK: Select all valid **spanning trees** of the graph below:



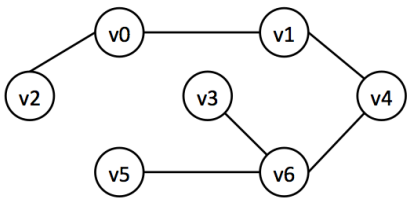
Choice A:



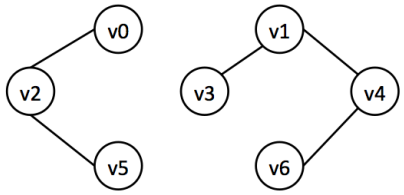
Choice B:



Choice C:



Choice D:

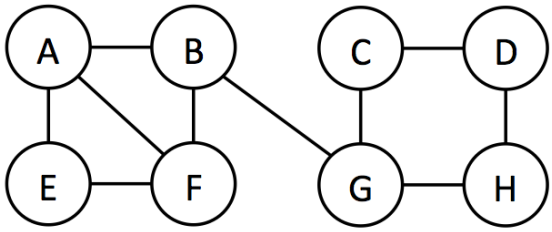


To solve this problem please visit https://stepik.org/lesson/28948/step/3

Step 4

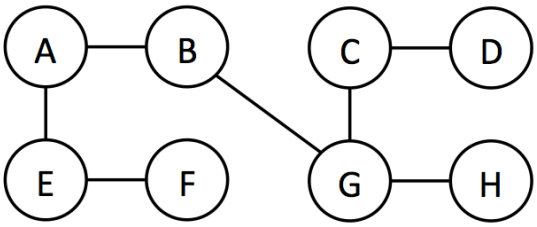
So how do we go about algorithmically finding spanning trees? This problem is actually very easy! All we need to do is run a single-source shortest-path algorithm, such as **Breadth First Search**. We can pick any vertex to run the shortest-path algorithm on. Once the algorithm is finished executing, all the "previous" fields of the vertex would lead us through a path that would create a **spanning tree**.

For example, take the graph below as an example:



Arbitrarily starting at vertex E and breaking ties by giving priority to the smallest letter of the alphabet, **BFS** would uncover the vertices in the order: E (previous null) → A (previous E) → F (previous E) → B (previous A) → G (previous B) → C (previous G) → H (previous G)→ D (previous C).

Following the discovered pathway above, we would thus get an assembled **spanning tree** that looks like this:



So why is **BFS** able to produce a valid spanning tree? The answer lies behind the idea that **BFS** is in charge of finding a single shortest path from one vertex to all other vertices and therefore will produce no cycles in its output —otherwise there would be more than 1 path to get to a vertex— and thus a **tree** that covers all vertices.

Step 5

It is important to note that in weighted graphs, we can also run **BFS** to find an *arbitrary* **spanning tree**. However, in weighted graphs, we often care about actually factoring the weights of the edges (cost of the paths). For example, in our original Networking Problem, an *arbitrary* spanning tree would translate into any network that successfully connects all of the computers at UCSD, but a *minimum* spanning tree will specifically translate into a valid network that has the *lowest cost*. Consequently, it can be useful to find the **Minimum Spanning Tree (MST)** for a graph: "the least-cost version" of the graph that is still connected. Formally, an **MST** is a spanning tree that

minimizes the total sum of edge weights.

Now that we are factoring in *minimum edge weights* in our algorithm, we should remember that **BFS** didn't always work well in these cases, which is why we originally introduced **Dijkstra's Algorithm**. In this case, we will also use a concept from **Dijkstra's Algorithm** to achieve what **BFS** couldn't.

We will introduce two different algorithms that can efficiently produce **Minimum Spanning Trees**: **Prim's** and **Kruskal's.**

In general,

- **Prim's Algorithm** starts with a single vertex from the original graph and builds the **MST** by iteratively adding the least costly edges that stem from it (very similar to **Dijkstra's Algorithm)**.
- **Kruskal's Algorithm** starts with a forest of vertices without any edges and builds the **MST** by iteratively adding the least costly edges from the entire graph.

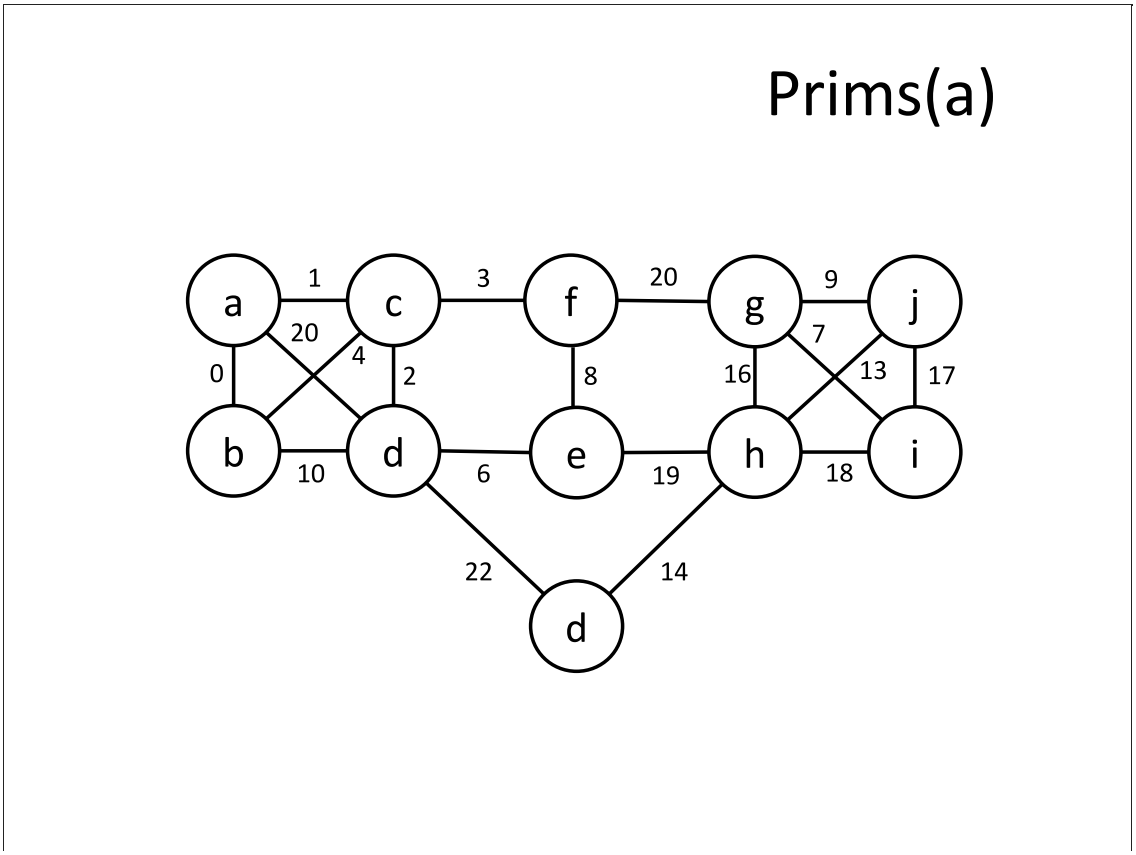
Step 6

As mentioned, **Prim's Algorithm** starts with a specified starting vertex. While the growing **Minimum Spanning Tree** does not connect *all* vertices, **Prim's Algorithm** adds the edge that has the least cost from any vertex in the spanning tree that has been built so far.

More formally,

1. Let ***V*** be the set of all vertices, ***S*** be the empty set, and ***A*** be the empty set. Choose any vertex *r* to be the root of our spanning tree; set ***S*** = {*r*} and ***V*** = ***V*** - {*r*} (i.e., remove *r* from set ***V***).
2. Find the least weight edge, *e*, such that one endpoint, v_1 , is in ***S*** and another, v_2 , is in ***V*** \ ***S*** (i.e., an edge that stems from the spanning tree being built so far, but doesn't create any cycles). Note: The notation ***V*** \ ***S*** simply means ***V*** - ***S*** (i.e., removing all of ***S***'s elements from ***V***).
    - Set ***A*** = ***A*** + {*e*}, ***S*** = ***S*** + {v_2}, and ***V*** = ***V*** - {v_2} (i.e., Add that edge to ***A*,** add its other vertex endpoint to ***S***, and remove that endpoint from the set of all "untouched" vertices, ***V***).
3. If ***V*** \ ***S*** = NULL (i.e., once ***S*** contains all vertices), then output **(*S*,*A*)** as the two sets of vertices and edges that define our **MST**. Otherwise, our **MST** clearly doesn't cover all vertices and so we must repeat step 2 above.

Below is an animation of how **Prim's Algorithm** chooses which vertices and edges to use to build the **Minimum Spanning Tree**. Use the arrows in the bottom left to step through the slides.



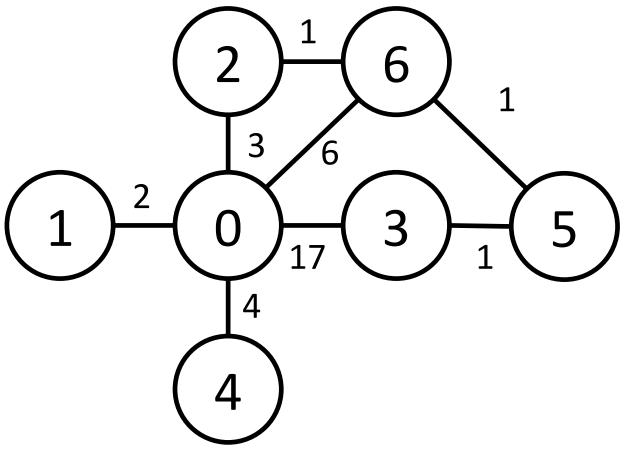
Step 7

Implementation-wise, **Prim's Algorithm** achieves the steps described previously by using a priority queue of edges sorted by edge weights. However, unlike **Dijkstra's Algorithm**, which had used the *total* path edge weights explored so far to the priority queue for comparison, **Prim's Algorithm** uses *single* edge weights.

Here is the pseudocode for **Prim's Algorithm**:

1. Initialize all the vertex fields of the graph — just like **Dijkstra's Algorithm**: Set all "done" fields to false. Pick a start vertex *r*. Set its "prev" field to -1 and its "done" field to true. Add all the edges adjacent to *r* as a tuple that includes the starting vertex, *v*, ending vertex, *w*, and cost, into the priority queue.
2. Is the priority queue empty? Done!
3. Else: Dequeue the top element, (*v*, *w*, cost) from the priority queue (i.e., remove the smallest cost edge).
4. Is the "done" field of vertex *w* marked true? If so, then this edge connects two vertices already connected in the **MST** and we therefore cannot use it. Go to step 2.
5. Else: Mark the "done" field of vertex *w* true, and set the "prev" field of *w* to indicate *v*.
6. Add all the edges adjacent to *w* into the priority queue.
7. Go to step 2.

Below is a visualization of how **Prim's Algorithm** takes advantage of the priority queue to choose which vertices and edges to use to build the **Minimum Spanning Tree**. Use the arrows in the bottom left to step through the slides.



Prims(v0)

Priority Queue:

Vertex	Prev	Done
V0	-1	F
V1	-1	F
V2	-1	F
V3	-1	F
V4	-1	F
V5	-1	F
V6	-1	F

So how well does **Prim's Algorithm** do with respect to time complexity? Hopefully by now you should be able to see all the similarities between **Prim's** and **Dijkstra's Algorithm** and can thus come to the intuitive conclusion that **Prim's Algorithm** is also **O(|V| + |E|* log(|E|))**.

Step 8

Below is a visualization of **Prim's Algorithm**, created by David Galles at the University of San Francisco.

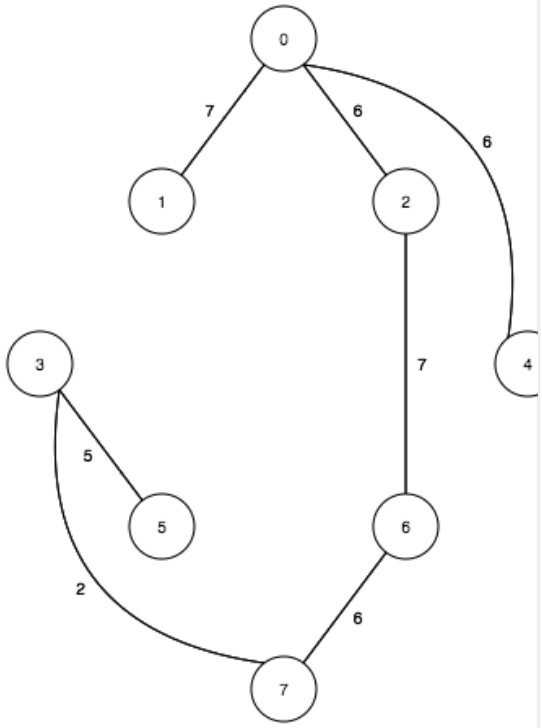
Prim Minimum Cost Spanning Treeh

Start Vertex: [ ] Run Prim New Graph

(•) Small Graph ( ) Large Graph

(•) Logical Representation ( ) Adjacency List Representation ( ) Adjacency Matrix Representation

Vertex	Known	Cost	Path
0			
1			
2			
3			
4			
5			
6			
7			



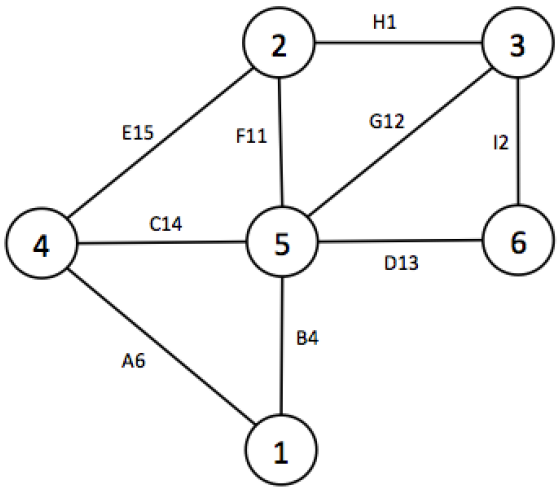
Animation Completed

Skip Back | Step Back | Pause | Step Forward | Skip Forward | Animation Speed | w: 1000 h: 500 | Change Canvas Size | Move Controls

Step 9

EXERCISE BREAK: For the graph below, rank the order of the edges (using the numbers 1 - 5) in the order that an **MST** would be built by **Prim's Algorithm**, starting at Vertex 1. For any edge that does not end up in the final **MST**, fill in the corresponding blank with the number 0.

Note: Each edge is labeled with its name, followed by its weight (i.e., A6 means edge A has a weight of 6).



To solve this problem please visit https://stepik.org/lesson/28948/step/9

Step 10

Kruskal's Algorithm is very similar to **Prim's Algorithm**. However, the major difference is that **Kruskal's Algorithm** starts with a "forest of nodes" (basically the original graph except with no edges) and interatively adds back the edges based on which edges have the lowest weight.

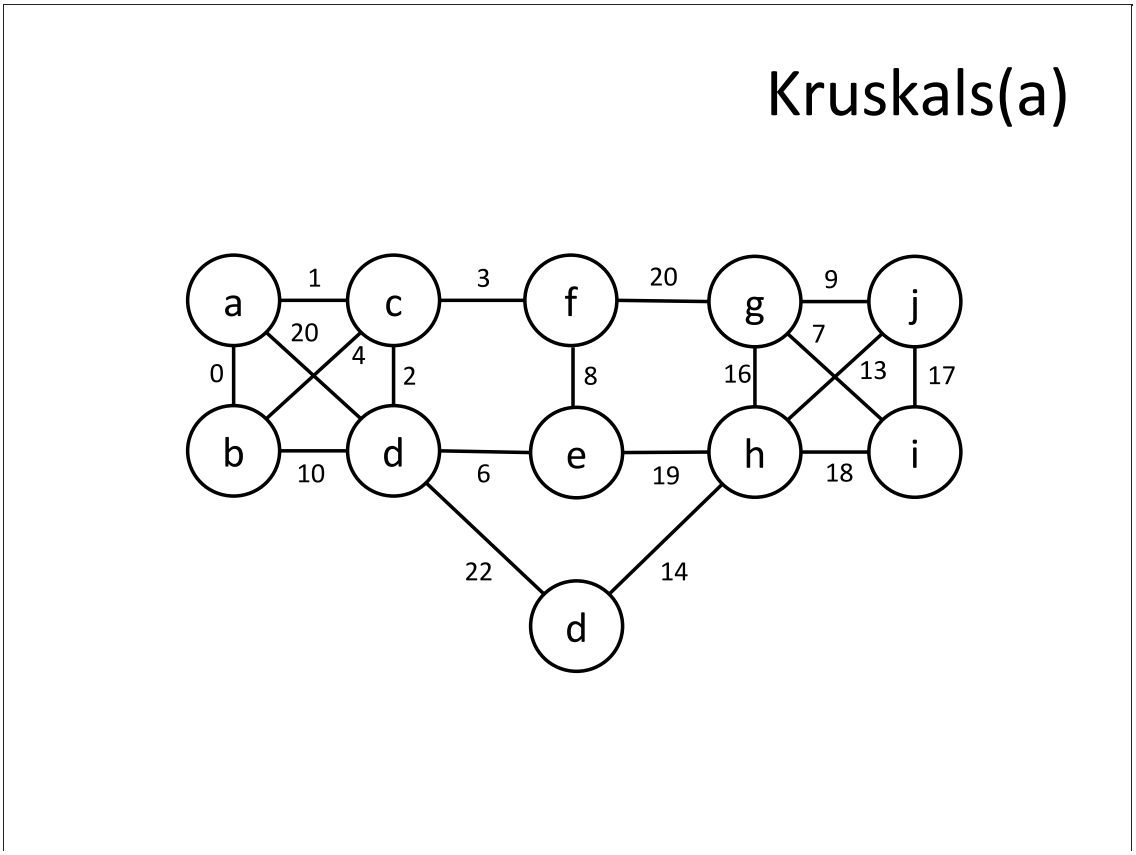
More formally, for a weighted undirected graph *G* = (***V***,***E***):

1. Let ***E*** be the set of all edges and ***A*** be the empty set.
2. Find the least weight edge, *e*.
    1. If ***A*** = ***A*** + {*e*} does *not* create a cycle; set ***E*** = ***E*** - {*e*} and ***A*** = ***A*** + {*e*}. In other words, if adding *e* to ***A*** does *not* create a cycle, add *e* to ***A*** (i.e., add the edge to the growing **MST**) and remove *e* from ***E***.
    2. Else set ***E*** = ***E*** - {*e*} (this edge creates a cycle and we should just ignore it)
3. If ***E*** = NULL (i.e., once we have checked all the edges), then output (***V***,***A***) as the two sets of vertices and edges that define our **MST**. Otherwise, we must repeat step 2 above.

STOP and Think: In **Prim's Algorithm**, we say that an edge *e* would create a cycle in our growing **MST** if both vertices of *e* were already in our growing **MST**. However, in **Kruskal's Algorithm**, even if both vertices of an edge *e* are already in our work-in-progress **MST**, *e* won't necessarily create a cycle. Why is this?

Hint: Pay attention to the last steps of the animation below.

Below is an animation of how **Kruskal's Algorithm** chooses which edges to use to build the **Minimum Spanning Tree**. Use the arrows in the bottom left to step through the slides.



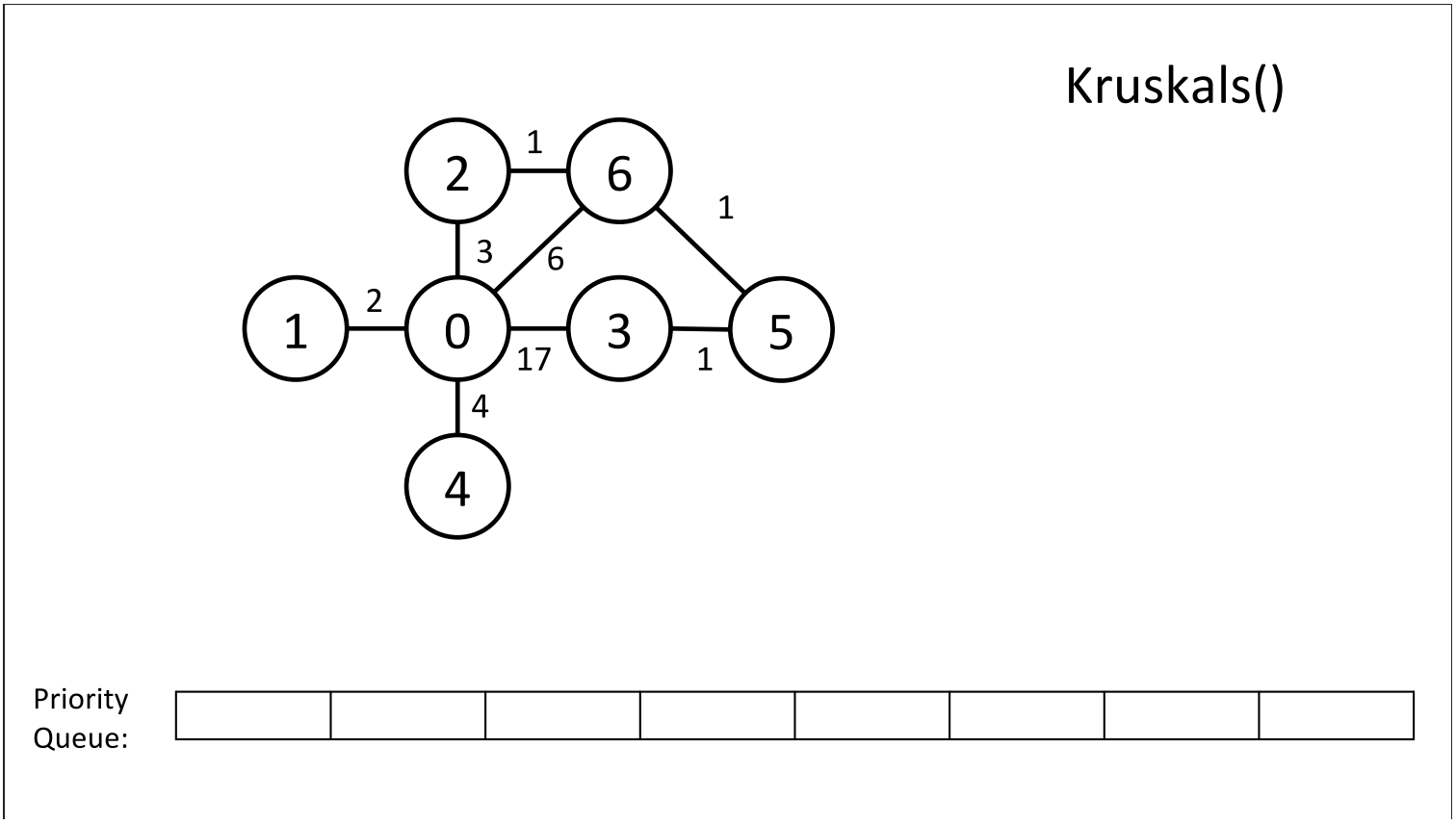
STOP and Think: Did **Kruskal's Algorithm** output the same **MST** as **Prim's Algorithm**?

Step 11

Implementation-wise, here is the pseudocode for **Kruskal's Algorithm** on a weighted undirected graph *G* = (***V***,***E***):

1. Create a forest of vertices
2. Create a priority queue containing all the edges, ordered by edge weight
3. While fewer than |***V***| - 1 edges have been added back to the forest:
    1. Deque the smallest-weight edge (*v*,*w*, cost), from the priority queue
    2. If: *v* and *w* already belong to the same tree in the forest, go to 3.1 (adding this edge would create a cycle)
    3. Else: Join those vertices with that edge and continue

Below is a visualization of how **Kruskal's Algorithm** takes advantage of the priority queue to choose which vertices and edges to use to build the **Minimum Spanning Tree**. Use the arrows in the bottom left to step through the slides.



It turns out that if **Kruskal's Algorithm** is implemented efficiently, it has a worst-case Big-O time complexity that is the same as **Prim's Algorithm**: **O(|*V*| + |*E*|*log(|*E*|)).**

Step 12

Below is a visualization of **Kruskal's Algorithm**, created by David Galles at the University of San Francisco.

Kruskal Minimum Cost Spanning Treeh

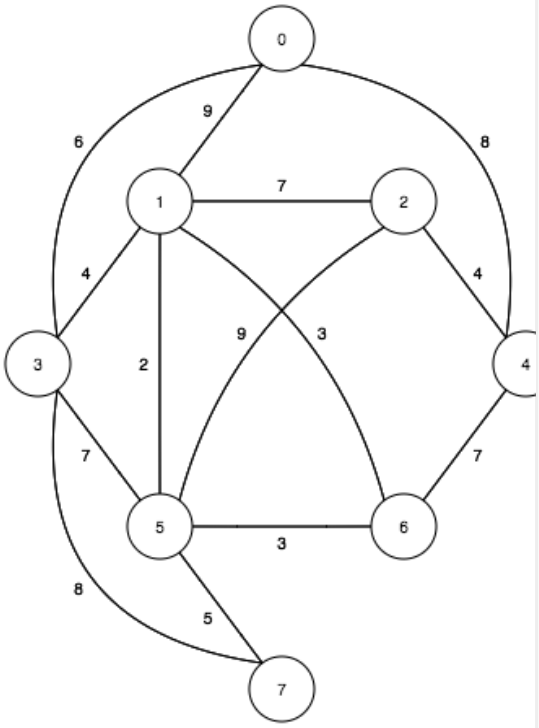
Run Kruskal | New Graph

Small Graph | Large Graph

Logical Representation | Adjacency List Representation | Adjacency Matrix Representation

Disjoint Set

0	-1
1	-1
2	-1
3	-1
4	-1
5	-1
6	-1
7	-1



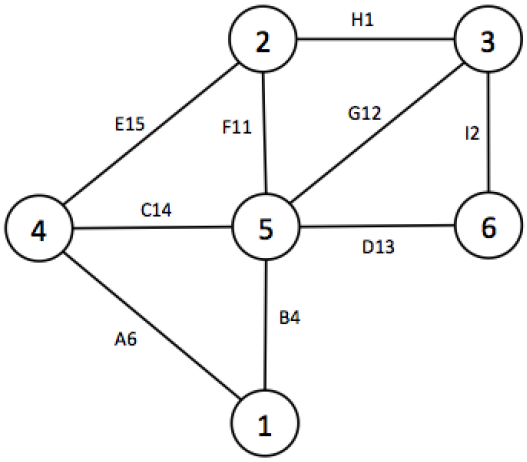
Animation Completed

Skip Back | Step Back | Pause | Step Forward | Skip Forward | Animation Speed | w: 1000 h: 500 | Change Canvas Size | Move Controls

Step 13

EXERCISE BREAK: For the graph below, rank the order of the edges (using the numbers 1 - 5) in the order that an **MST** would be built by **Kruskal's Algorithm**. For any edge that does not end up in the final **MST**, fill in the corresponding blank with the number 0.

Note: Each edge is labeled with its name, followed by its weight (i.e., A6 means edge A has a weight of 6).



To solve this problem please visit https://stepik.org/lesson/28948/step/13

Step 14

EXERCISE BREAK: In a dense graph, which **MST** algorithm do we expect to perform better **when implemented efficiently**?

To solve this problem please visit https://stepik.org/lesson/28948/step/14

Step 15

EXERCISE BREAK: True or False: Just like in **Dijkstra's Algorithm**, both **Prim's Algorithm** and **Kruskal's Algorithm** require the edges to be non-negative.

To solve this problem please visit https://stepik.org/lesson/28948/step/15

Step 16

We started this discussion by trying to come up with a good way to design the cabling of UC San Diego's intranet, and we were able to transform the original problem into the task of finding a **Minimum Spanning Tree** in an **undirected weighted graph**. Then, throughout this section, we explored two algorithms, **Kruskal's Algorithm** and **Prim's Algorithm**, both of which solve this computational problem efficiently in **O(|*V*| + |*E*|*log(|*E*|))** time. We saw that **Prim's Algorithm** was implemented very similarly to **Dijkstra's Algorithm**, yet didn't necessarily have all of the same restrictions (e.g. **Kruskal's Algorithm** and **Prim's Algorithm** can easily work with negative edges).

STOP and Think: For a graph with all unique edge weights, both **Kruskal's** and **Prim's Algorithm** will always return the same exact **MST**. Why might this be the case?