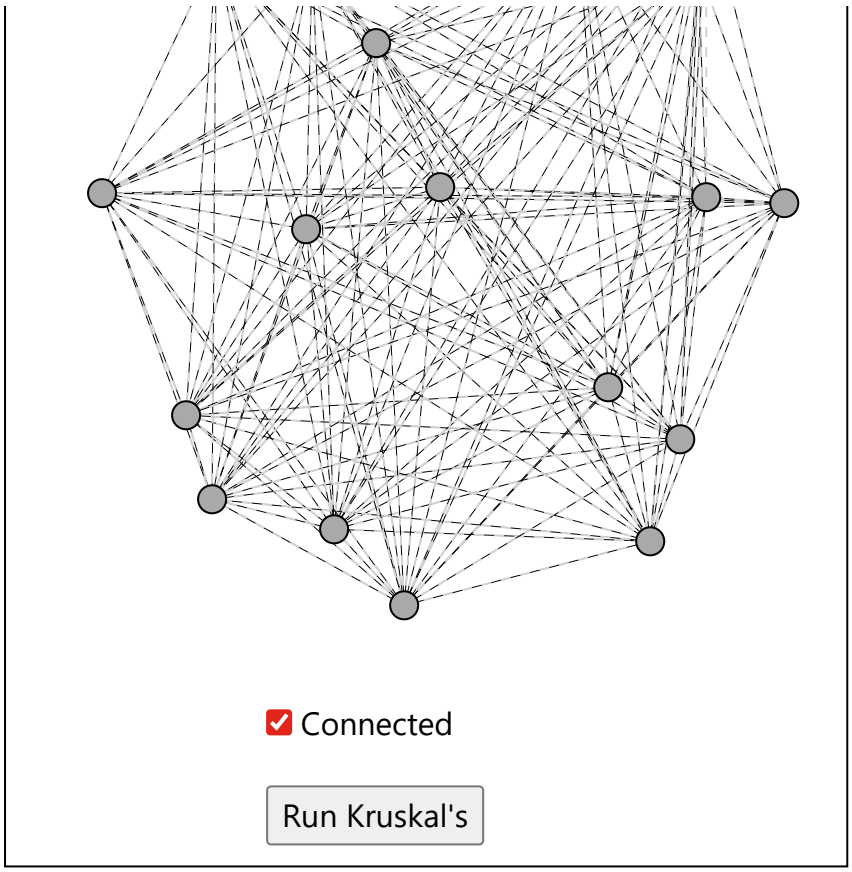
DSA Kruskal's Algorithm

‹ Previous Next ›

Kruskal's Algorithm

Kruskal's algorithm finds the Minimum Spanning Tree (MST), or Minimum Spanning Forest, in an undirected graph.



The MST (or MSTs) found by Kruskal's algorithm is the collection of edges that connect all vertices (or as many as possible) with the minimum total edge weight.

Kruskal's algorithm adds edges to the MST (or Minimum Spanning Forest), starting with the edges with the lowest edge weights.

Edges that would create a cycle are not added to the MST. These are the red blinking lines in the animation above.

Kruskal's algorithm checks all edges in the graph, but the animation above is made to stop when the MST or Minimum Spanning forest is completed, so that you don't have to wait for the longest edges to be checked.

Minimum Spanning Forest is what it is called when a graph has more than one Minimum Spanning Tree. This happens when a graph is not connected. Try it yourself by using the checkbox in the animation above.

To find out if an edge will create a cycle, we will use Union-Find cycle detection inside Kruskal's algorithm.

How it works:

1. Sort the edges in the graph from the lowest to the highest edge weight.
2. For each edge, starting with the one with the lowest edge weight:
    a. Will this edge create a cycle in the current MST?
        - If no: Add the edge as an MST edge.

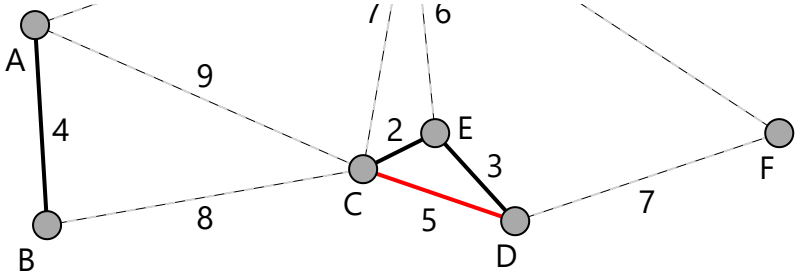
Manual Run Through

Let's run through Kruskal's algorithm manually on the graph below, so that we understand the detailed step-by-step operations before we try to program it.

The first three edges are added to the MST. These three edges have the lowest edge weights and do not create any cycles:

- C-E, weight 2
- D-E, weight 3
- A-B, weight 4

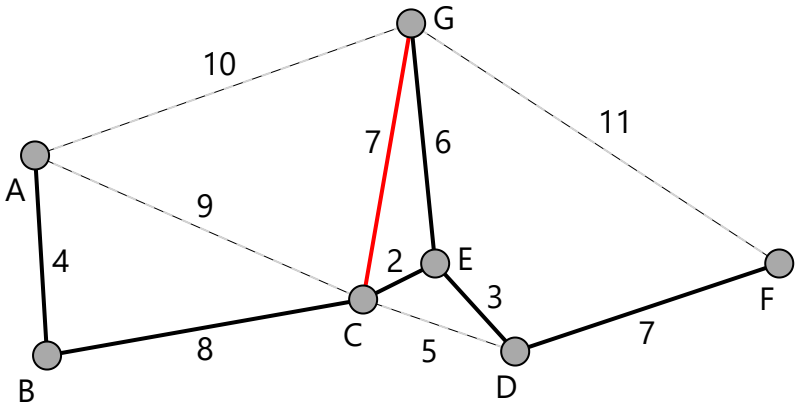
After that, edge C-D (indicated in red) cannot be added as it would lead to a cycle.



The next four edges Kruskal's algorithm tries to add to the MST are:

- E-G, weight 6
- C-G, weight 7 (not added)
- D-F, weight 7
- B-C, weight 8

Edge C-G (indicated in red) cannot be added to the MST because it would create a cycle.

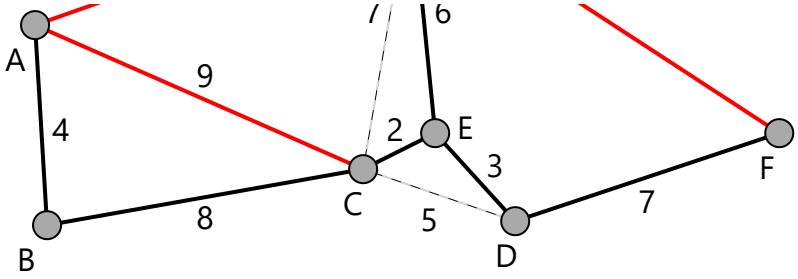


As you can see, the MST is already created at this point, but Kruskal's algorithm will continue to run until all edges are tested to see if they can be added to the MST.

The last three edges Kruskal's algorithm tries to add to the MST are the ones with the highest edge weights:

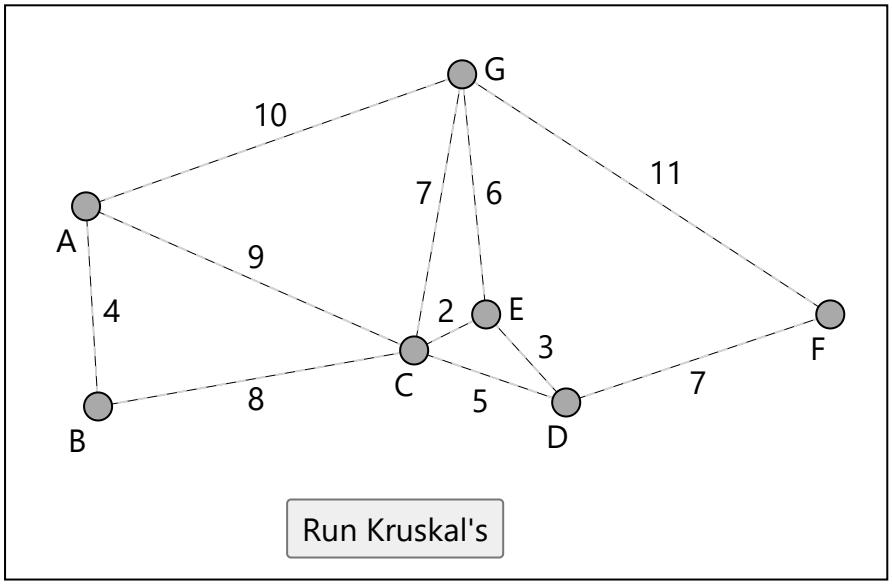
- A-C, weight 9 (not added)
- A-G, weight 10 (not added)
- F-G, weight 11 (not added)

Each of these edges would create a cycle in the MST, so they cannot be added.



Kruskal's algorithm is now finished.

Run the simulation below to see Kruskal's algorithm doing the manual steps that we have just done.



Note: Although Kruskal's algorithm checks all edges in the graph, the animation at the top of this page stops right after the last edge is added to the MST or Minimum Spanning Forest so that we don't have to look at all the red edges that can't be added.

This is possible because for a connected graph, there is just one MST, and the search can stop when the number of edges in the MST is one less than there are vertices in the graph ($V - 1$). For the unconnected graph, there are two MSTs in our animation, and the algorithm stops when the MSTs have reached a size of $V - 2$ edges in total.

Implementation of Kruskal's Algorithm

```
class Graph:
    def __init__(self, size):
        self.size = size
        self.edges = []  # For storing edges as (weight, u, v)
        self.vertex_data = [''] * size  # Store vertex names

    def add_edge(self, u, v, weight):

            self.edges.append((weight, u, v))  # Add edge with weig

    def add_vertex_data(self, vertex, data):

            self.vertex_data[vertex] = data
```

Line 8 and 12: Checks if the input arguments `u`, `v`, and `vertex`, are within the possible range of index values.

To do Union-Find cycle detection in Kruskal's algorithm, these two methods `find` and `union` are also defined inside the `Graph` class:

```

```

Line 15-18: The `find` method uses the `parent` array to recursively find the root of a vertex. For each vertex, the `parent` array holds a pointer (index) to the parent of that vertex. The root vertex is found when the `find` method comes to a vertex in the `parent` array that points to itself. Keep reading to see how the `find` method and the `parent` array are used inside the `kruskals_algorithm` method.

Line 20-29: When an edge is added to the MST, the `union` method uses the `parent` array to merge (union) two trees. The `rank` array holds a rough estimate of the tree height for every root vertex. When merging two trees, the root with a lesser rank becomes a child of the other tree's root vertex.

Here is how Kruskal's algorithm is implemented as a method inside the `Graph` class:

```
    def kruskals_algorithm(self):
        result = []  # MST
        i = 0 # edge counter

        self.edges = sorted(self.edges, key=lambda item: item[2])

        parent, rank = [], []

        for node in range(self.size):
            parent.append(node)
            rank.append(0)

        while i < len(self.edges):
            u, v, weight = self.edges[i]
            i += 1

            x = self.find(parent, u)
            y = self.find(parent, v)
            if x != y:
                result.append((u, v, weight))
                self.union(parent, rank, x, y)

        print("Edge \tWeight")
```

Line 35: The edges must be sorted before Kruskal's algorithm starts trying to add the edges to the MST.

Line 40-41: The `parent` and `rank` arrays are initialized. To start with, every vertex is its own root (every element in the `parent` array points to itself), and every vertex has no height (`0` values in the `rank` array).

Line 44-45: Pick the smallest edge, and increment `i` so that the correct edge is picked in the next iteration.

Line 47-51: If the vertices `u` and `v` at each end of the current edge have different roots `x` and `y`, it means there will be no cycle for the new edge and the trees are merged. To merge the trees, the current edge is added to the `result` array, and we run the `union` method to make sure the trees are merged correctly, so that there is only one root vertex in the resulting merged tree.

Now let's create the graph from the "Manual Run Through" above and run Kruskal's algorithm on it:

Example

Python:

```
class Graph:
    def __init__(self, size):
        self.size = size
        self.edges = []  # For storing edges as (weight, u, v)
        self.vertex_data = [''] * size  # Store vertex names

    def add_edge(self, u, v, weight):
        if 0 <= u < self.size and 0 <= v < self.size:
            self.edges.append((u, v, weight))  # Add edge with we

    def add_vertex_data(self, vertex, data):
        if 0 <= vertex < self.size:
            self.vertex_data[vertex] = data

    def find(self, parent, i):
```

Time Complexity for Kruskal's Algorithm

For a general explanation of what time complexity is, visit this page.

With E as the number of edges in our graph, the time complexity for Kruskal's algorithm is

$$O(E \cdot logE)$$

We get this time complexity because the edges must be sorted before Kruskal's can start adding edges to the MST. Using a fast algorithm like Quick Sort or Merge Sort gives us a t

$O(E \cdot logE)$

After the edges are sorted, they are all checked one by one, to see if they will create a cycle, and if not, they are added to the MST.

Although it looks like a lot of work to check if a cycle will be created using the `find` method, and then to include an edge to the MST using the `union` method, this can still be viewed as one operation. The reason we can see this as just one operation is that it takes approximately constant time. That means that the time this operation takes grows very little as the graph grows, and so it does actually not contribute to the overall time complexity.

Since the time complexity for Kruskal's algorithm only varies with the number of edges E, it is especially fast for sparse graphs where the ratio between the number of edges E and the number of vertices V is relatively low.

‹ Previous Sign in to track progress Next ›

COLOR PICKER

PHP Tutorial
Java Tutorial
C++ Tutorial
jQuery Tutorial

Top References

HTML Reference
CSS Reference
JavaScript Reference
SQL Reference
Python Reference
W3.CSS Reference
Bootstrap Reference
PHP Reference
HTML Colors
Java Reference
AngularJS Reference
jQuery Reference

Top Examples

HTML Examples
CSS Examples
JavaScript Examples
How To Examples
SQL Examples
Python Examples
W3.CSS Examples
Bootstrap Examples
PHP Examples
Java Examples
XML Examples
jQuery Examples

Get Certified

HTML Certificate
CSS Certificate
JavaScript Certificate
Front End Certificate
SQL Certificate
Python Certificate
PHP Certificate
jQuery Certificate
Java Certificate
C++ Certificate
C# Certificate
XML Certificate