Aug 2025, India

Kruskal's Algorithm

Tutorial

Tutorial By

Vikas Awadhiya

Invented By

Joseph Kruskal

Tutorial By

Vikas Awadhiya

LinkedIn profile: https://in.linkedin.com/in/awadhiya-vikas

1 Introduction

The Kruskal's algorithm finds minimum spanning tree (MST) of an undirected weighted graph. It finds minimum spanning tree of a simple/multi graph, connected/disconnected graph. Kruskal's algorithm has ability to find minimum spanning forest (spanning tree of each component) of a disconnected graph in a single run as compare to Prim's algorithm which requires a separate run for each component of a disconnected graph.

Kruskal's algorithm sorts the edges by their weights in acceding order and iterate over them to find minimum spanning tree.

Algorithm has $O(E \log V)$ time complexity and $O(E + V)$ space complexity, assuming Union-Find data structure is used to detect the cycle. Here E is number of edges $|E|$ and V is number of vertices $|V|$.

Kruskal's algorithm belongs to a special category of algorithms called greedy algorithms.

2 The Algorithm

Kruskal's algorithm finds the minimum spanning tree of an undirected weighted graph $G = (V, E, \omega)$, where V is finite set of vertices, E is finite set of edges and ω is a weight function where $\omega : E \to \mathbb{R}$ or simply $\forall e \in E,\ \omega(e) = \mathbb{R}$ (here ω is a Greek alphabet omega in lower case, $\to$ means "map to" / "is a function from", $\mathbb{R}$ is real numbers, $\forall$ is read as "for all"/ "for every", $\in$ is read as "element of"/"belongs to" and e represents an edge).

As a greedy algorithm it makes decision best/optimal at a step level rather then best/optimal decision at a problem level. To understand the algorithm it is first required to understand what is a minimum spanning tree and that also requires to understand few other concepts of graph-theory like tree, cut-edge and spanning tree.

Tree is an acyclic graph which doesn't contains cycle (or self loop) and every edge of the graph is a cut-edge as shown in fig 1 below on the left. So removing any edge disconnects the graph and creates two components. The cut-edge also called bridge or bond. The bond is minimum number of edges requires to remove, to divide the graph into two components. In a tree, bond is only one edge and removing a single edge creates two components as shown below in fig 2 and in fig 3.

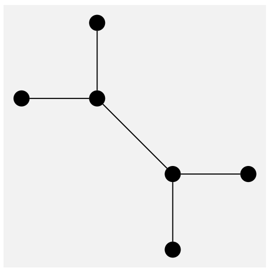
Fig 1

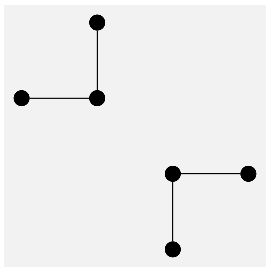
Fig 2

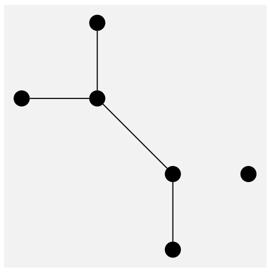
Fig 3

In other words, tree is a graph in which, there is only a single path between any two vertices. A graph shown in fig 4 below on the left is not a tree because it has multiple paths between the vertices for example multiple paths between vertex a and vertex d are highlighted by bold lines in fig 5 and fig 6 below.

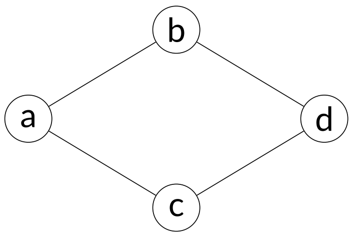

Fig 4

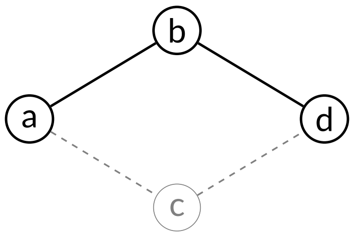

Fig 5

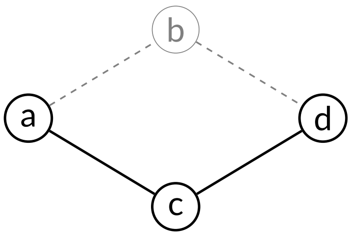

Fig 6

So the graph shown in fig 4 is not a tree and due to this removing an edge can't divide graph in two components.

The spanning tree is a tree which span across all vertices of a graph. The tree shown in fig 8 below on the middle is a spanning tree of the graph shown below on the left in fig 7 compared to tree shown below on the right in fig 9 which is not a spanning tree because it doesn't span across all the vertices of the graph and the one vertex remains isolated vertex.

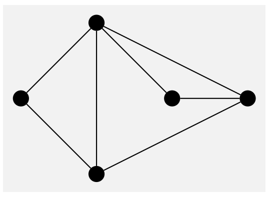
Fig 7

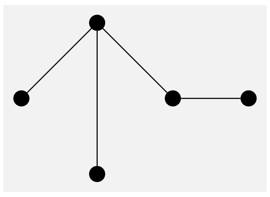
Fig 8

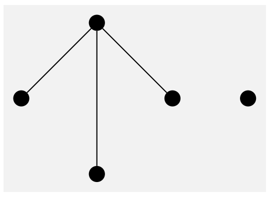
Fig 9

There is more than one spanning tree possible in a graph and number of spanning trees of a graph G is denoted by $\tau(G)$ (here τ is Greek alphabet tau in lower-case). Number of spanning trees of a complete graph K_n can be evaluated by $\tau(K_n) = n^{n-2}$, for example a triangle as a complete graph with three vertices and edges of weights 1, 1 and 2 has $\tau(K_3) = 3^{3-2} = 3$ number of spanning trees as shown below in fig 10, fig 11 and in fig 12.

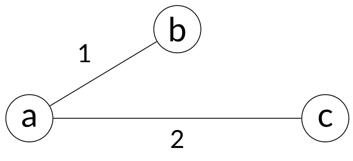

Fig 10

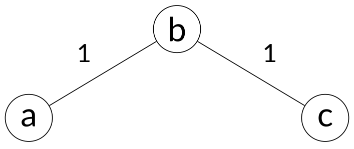

Fig 11

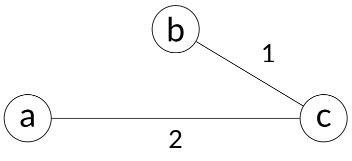

Fig 12

Among these spanning trees the tree shown in fig 11 above on the middle is minimum spanning tree because sum of weights of it's edges is minimum compare to sum of weights of edges of other two spanning trees. Tutorials explained minimum spanning tree and concepts related to it, now it proceeds to explain algorithm.

2.1 Algorithm's Steps

Kruskal's algorithm is viewed as the following steps,

1. Sort the edges by their weights in acceding order,
2. Iterate over the edges and select an edge if selecting this edge doesn't introduce a cycle (To find if selecting an edge introduces a cycle algorithm uses Union-Find / Disjoint-set data structure).

An example is required to explain these steps, so let consider an undirected weighted graph G as shown below in fig 13.

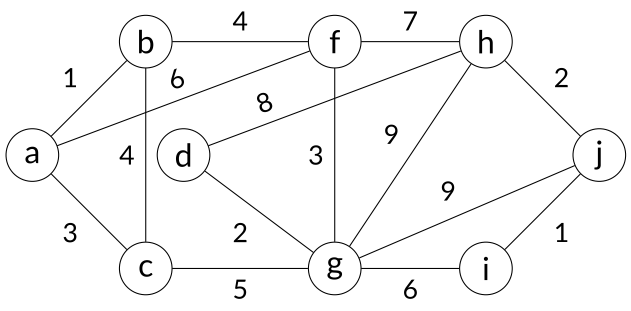

Fig 13 : Graph G

This graph G is a simple undirected weighted graph. A simple graph is a graph which doesn't have a vertex with self-loop or contain parallel edges between any two vertices. That's why in a simple graph a pair of vertices $\{u, v\}$ uniquely represents an edge between any vertex u and vertex v and $\omega(uv)$ represents edge's weight. Vertices are usually labeled by numbers in graph, but here English alphabet is used to distinguish vertices from weights of edges.

Let’s assume the edges are sorted by their weights as a first step of algorithm. Now process moves to the next step where it iterates over the edges. The fig 14 as shown below on the left, represents initial state where no edge is selected yet.

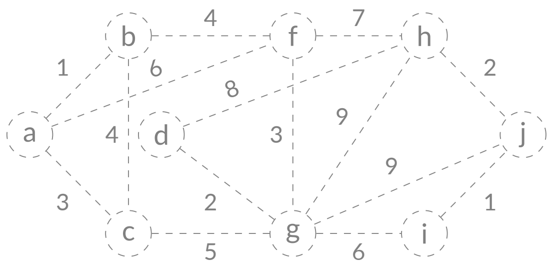

Fig 14

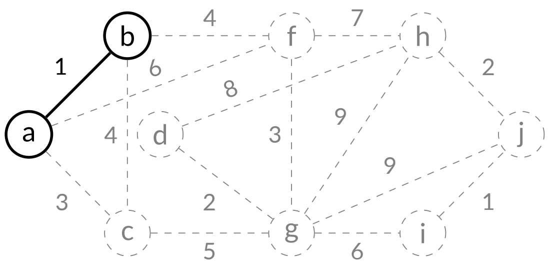

Fig 15

Now second step begins by iterating over the sorted edges. There are two edges, edge {a, b} and edge {i, h} with weight 1, so let’s consider algorithm first processes edge {a, b}. Edge {a, b} is selected because it doesn’t introduce a cycle as shown above on the the right in fig 15 and the selected edge {a, b} is highlighted by bold line.

As iteration continues, the next edge is {i, h} with weight 1 as shown above on the right in fig 15. Algorithm selects this edge because it doesn’t introduce cycle as shown below on left in fig 16 and edge {i, h} is highlighted by bold line.

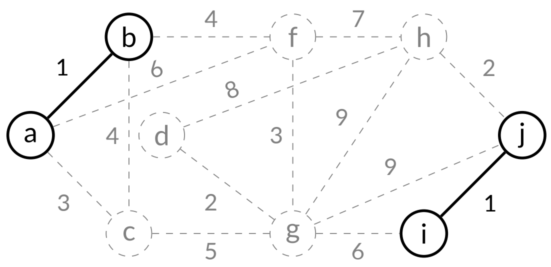

Fig 16

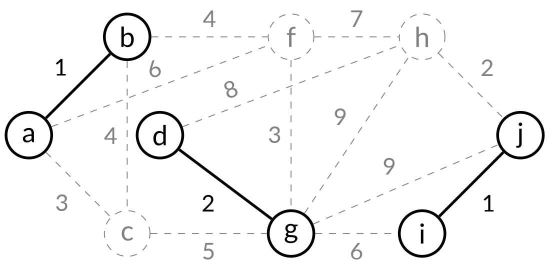

Fig 17

Iterate continues and there are two edges, edge {d, g} and edge {h, j} with weight 2 as shown above in fig 16. Let’s consider algorithm process edge {d, g} first and selects it because it doesn’t introduce cycle as shown in fig 17 above on the right and highlighted by bold line. Algorithm uses Union-Find / Disjoint-Set data structure to detect the cycle in each step.

Iteration continues and algorithm processes edge {h, j} with weight 2 next and selects it because it doesn’t introduce cycle as shown in fig 18 below on the left and edge is highlighted by bold line.

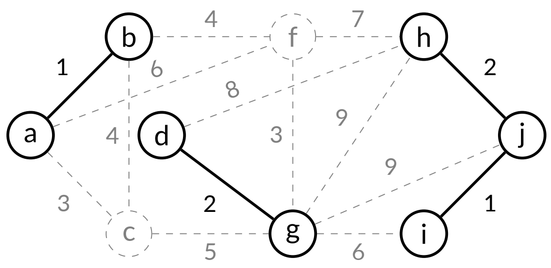

Fig 18

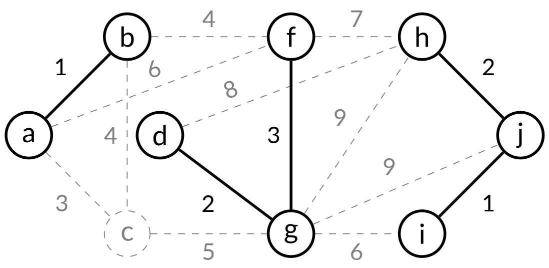

Fig 19

Next there are two edges, edge {a, c} and edge {f, g} with weight 3 as shown in fig 18 above.

Let's consider edge {f, g} is preceded by edge {a, c} in iteration and it is processed next. Algorithm selects edge {f, g} because it doesn't introduce cycle as shown in fig 19 above on the right and selected edge is highlighted by bold line.

The iteration continues and algorithm selects next edge {a, c} because it doesn't introduce cycle as shown in fig 20 below.

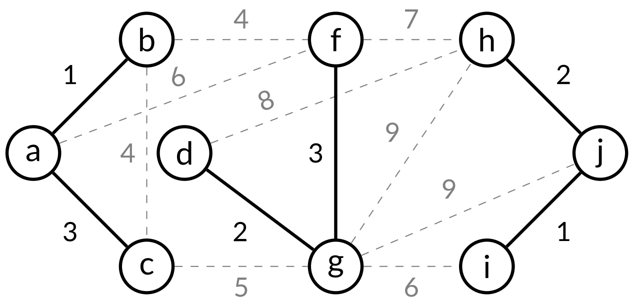

Fig 20

Now each vertex is connected and there is no vertex remain isolated. These components are not a tree (spanning tree) rather a group of 3 trees also called forest (as highlighted in bold lines) constructed in parallel with independently of each other. These components must be connected acyclically to form a spanning tree.

This demonstrate why an efficient technique is essential in Kruskal's algorithm to detect the cycles because in real world scenario individual components may contains a lot of elements and DFS/BFS is inefficient to detect the cycle and union-find / disjoint-set data structure is required. Two scenarios are shown below in fig 21 and in fig 22 when these components are connected in a why that introduce cycles **a, b, f, g, c** and **f, h, j, i, g** respectively and graph doesn't remain acyclic/tree. The union-find data structure helps to prevent such scenarios.

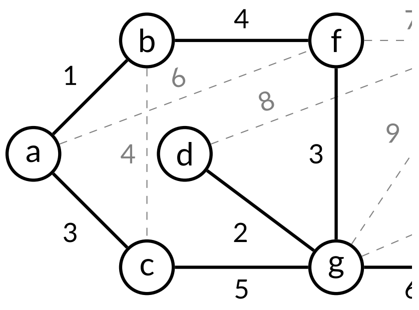

Fig 21

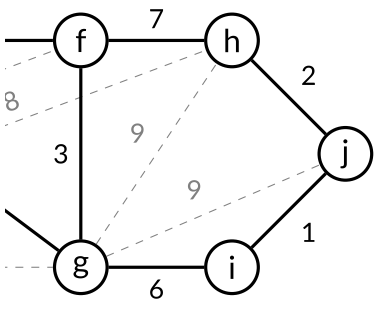

Fig 22

You can learn union-find data structure from anywhere or you can read a tutorial at https://www.github.com/vikasawadhiya/Union-Find-Or-Disjoint-Set written by me.

Iteration continues and there are two edges, edge {b, c} and edge {b, f} with weight 4 as shown above in fig 20. Let's consider edge {b, c} is processed next but as shown in fig 20 above, selection of it result in a cycle **a, b, c** therefor it can't be selected and iteration continue and process edge {b, f} and selects it because it doesn't introduce cycle as shown below on the left in fig 23.

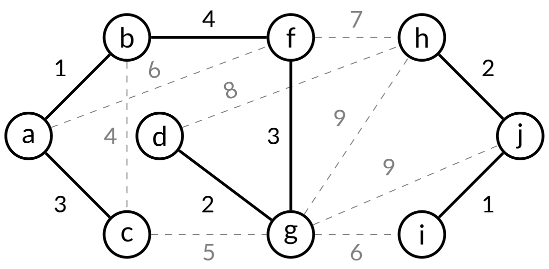

Fig 23

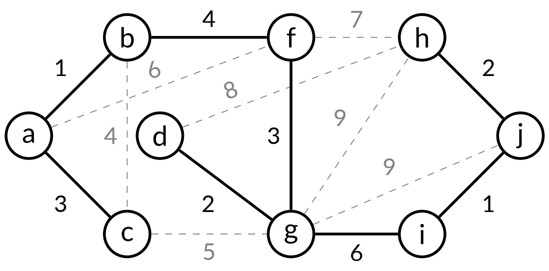

Fig 24

Iteration continues, and there are two edge, edge {a, f} and edge {g, i} with weight 6 as shown in fig 23 above on the left. Let's consider algorithm processes edge {g, i} first and selects it because edge {g, i} doesn't introduce a cycle as shown above on the right in fig 24.

Iteration continues over the remaining edges but no further edges can be selected because spanning tree of graph G is already constructed and adding any edge further will introduce a cycle. So the spanning tree of graph G is shown belong in fig 25.

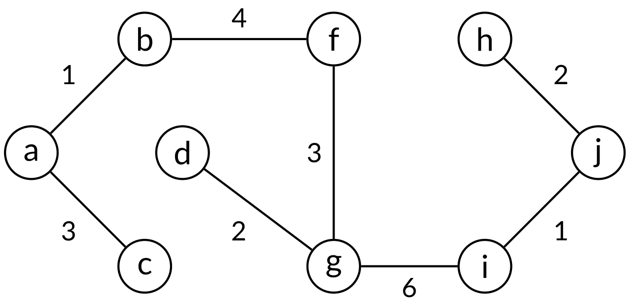

Fig 25 : Spanning Tree

The Kruskal's algorithm doesn't requires to iterate over all the edges rather iteration can be stopped immediately once $V - 1$ edges are selected because a tree of V vertices requires only $V - 1$ edges and adding any further edge introduces a cycle. This optimization doesn't work in disconnected graph because with disconnected graph number of selected edges never reach to $V - 1$.

But the Kruskal's algorithm usually iterate over all the edges. The reason is, Kruskal's algorithm performs well in sparse graphs where edges are in small number and iterating over all of them is not so expensive as compared to dense graphs like complete graphs has large number of edges $(V(V-1))$ and for a dense graph Prim's algorithm is preferred.

To detect the cycles algorithm initialize union-find data structure by providing number of elements that is equal to $|V|$ number of vertices. Initially all vertices are in different sets in union-find data structure. Iterating over the edges algorithm request union-find data structure to merge the sets both the vertices of an edge belong to. The merge only succeed if both the vertices belong to different sets that means both of the vertices don't belong to same component and connecting them with this edge doesn't introduce cycle. If succeeds algorithm selects the edge but if merge fails then algorithm rejects the edge and iterates over the next edge.

3 Complexity Analysis

The time complexity of the Kruskal's algorithm can be evaluated by evaluating the time complexity of sorting the edges and complexity of iterating over all the edges.

The n number of edges can be sorted in $O(n \log n)$ time complexity and here E the number of edges $|E|$ in a graph can be sorted in $O(E \log E)$ time complexity.

Iterating over E number of sorted edges requires $O(E)$ time complexity and assuming that union-find data structure is used to detect the cycles, requires constant (almost constant) time complexity at each step. So the overall time complexity of second step is $O(E) + c$, here c represent a constant. But in asymptotic analysis big O notation represents worst case and ignores the constant that means the time complexity of iterating over edges is $O(E)$.

Time Complexity = Complexity of iteration over all edges + Sorting complexity of edges
= $O(E) + O(E \log E)$
= $O(E)\ O(1 + \log E)$, but constant 1 is ignored in $O(1 + \log E)$
= $O(E)\ O(\log E)$
= $O(E \log E)$

In graph theory graphs are represented by vertices and edges as $G = (V, E)$ and in weighted graph weight function ω also is used as $G = (V, E, \omega)$ but still graph is expressed in terms of vertices and edges. In this sense time complexity of Kruskal's algorithm should be expressed in terms of both vertices and edges.

The minimum number of edges possible in connected graph (tree) is $V - 1$,

Time Complexity = $O(E \log E)$
= $O(E \log(V - 1))$, but big O notation ignores constant
= $O(E \log V)$

And the maximum number of edges possible in a K_n complete graph is $E = V(V - 1)$. A multigraph may have more then $V(V - 1)$ edges but it is a rare case scenario and time complexity mostly remains same because of power property of $\log$ (explained below). In big O notation constant is ignores and $O(V(V - 1))$ becomes $O(V^2)$.

Time Complexity = $O(E \log V^2)$
but the power property of $\log$ is, $\log_x(y^n) = n \log_x y$
= $O(E\ 2 \log V)$, but big O notation ignores constant
= $O(E \log V)$

It means time complexity of Kruskal's algorithm is $O(E \log V)$ for all types of undirected weighted graphs.

Sorting requires $O(E)$ space to store the edges in array and quick sort requires $O(E)$ space in a worst case, but $O(E) + O(E) = 2\ O(E) = O(E)$ and Union-Find data structure requires $O(V)$ space for V vertices plus $O(V)$ addition space for meta-data rank/size information and $O(V) + O(V) = 2\ O(V) = O(V)$.

The overall space complexity of Kruskal's algorithm is $O(E + V)$. In dense graph K_n where $E = V(V - 1)$ and space complexity becomes $O(V^2 + V) = O(V^2)$ but the $O(E + V)$ form is always used and it is applicable to all types of graphs.

4 Directed Graph Constraint

It is obvious question why Kruskal's algorithm can't be used with directed weighted graph to find something similar to a minimum spanning tree?

Analogue of a minimum spanning tree in a directed weighted graph is called minimum spanning arborescence (MSA). It is fundamentally differ from a minimum spanning tree.

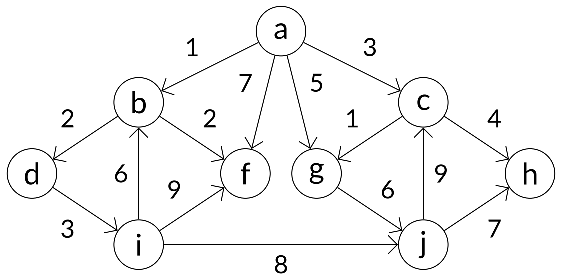

Fig 26

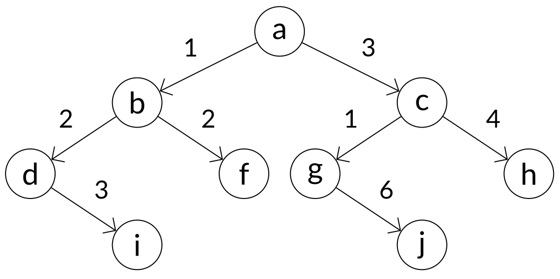

Fig 27

Arborescence (spanning arborescence) is an acyclic directed graph in which there is an unique path from a vertex (root vertex) to every other vertex of a directed graph. It is also called a directed rooted tree. The minimum spanning arborescence is a spanning arborescence with minimum sum of weight of edges. A directed graph shown above on the left in fig 26 and it's minimum spanning arborescence is shown above on the right in fig 27.

The kruskal's algorithm doesn't work with a directed graph because, Union-Find data structure can't detect the cycles in a directed graph but this is not the main problem because it can be replaced by other data structure to detect the cycles in directed graphs. The main problem is, Kruskal's algorithm iterates over sorted edges and at each step select an edge with minimum possible weight but this greedy approach only care about the minimum weight and can't select the edges by considering the direction as well.

An arborescence (spanning arborescence) is not always possible for directed graphs and it may happen that a directed connected weighted graph doesn't have a vertex such that there is a unique path from it to every other vertex. This makes finding of a minimum spanning arborescence more complex. The directed connected graph shown below in fig 28, there is no arborescence possible for it.

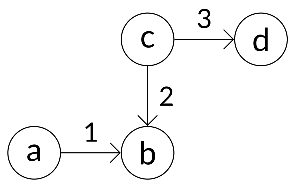

Fig 28

So if there is no arborescence possible for a directed connected graph then how Kruskal's algorithm can find minimum spanning arborescence? The kruskal's algorithm works with an undirected connected graph because an undirected connected graph always has at least one spanning tree due to undirected edges. Undirected edges are used in both direction and makes every vertex reachable from every other vertex.

5 Implementation

The C++ implementation of the Kruskal's algorithm is provided under the MIT License. It also contains the `main.cpp` file to demonstrate its usage.

Code available at: https://github.com/vikasawadhiya/Kruskal-Algorithm

August 2025, India