Minimal Spanning Trees in Graph Theory: Applications in resource optimization.

Jean Batista
Juniata College

Introduction to Minimal Spanning Trees

A minimum spanning tree (MST) is a fundamental concept in graph theory and combinatorial optimization. The MST problem is one of the oldest and most studied problems in computer science, with a history dating back to the 1920s (Graham & Hell, 1985). In an MST problem, we are given a set of points (called vertices) connected by links (called edges) that have associated weights or costs. The goal is to select a subset of these links that connects all the points together with the smallest possible total cost. In other words, an MST is a tree (a cycle-free connected subgraph) that spans all vertices of a weighted graph and has the minimum achievable sum of edge weights (Cormen, Leiserson, Rivest, & Stein, 2009).

Several classic algorithms exist for finding an MST in a graph. In this paper, we focus on two well-known greedy algorithms: Prim's algorithm (Prim, 1957) and Kruskal's algorithm (Kruskal, 1956). Both algorithms build the minimum spanning tree by incrementally adding edges that preserve the minimality and connectivity properties, but they do so in different ways. We will describe each algorithm in detail with step-by-step explanations. We also provide a case study demonstrating how each algorithm works on an example graph, and we discuss some real-world applications of MSTs.

Graph Theory Background

Graphs and Spanning Trees: In graph theory, a *graph* represents a collection of objects and the connections between them. The objects are usually called **vertices** (or nodes), and the connections are called **edges**. For example, imagine a set of cities with roads between them – each city can be thought of as a vertex, and each road as an edge connecting two cities. A graph can be visualized as a set of points (vertices) with lines (edges) drawn between certain pairs of points. When each edge has an associated number (for instance, a distance or cost), we call the graph a **weighted graph**. An edge's weight typically represents how expensive or long that connection is.

A **spanning tree** of a graph is a subset of the edges that connects *all* the vertices together without any cycles. *Connecting all vertices* means every vertex in the graph is reached by the edges in the spanning tree. Having *no cycles* means that there is no way to start at one vertex and follow a series of edges that eventually loops back to the starting point – in other words, there is exactly one path between any two vertices in a tree. A spanning tree thus "spans" the graph (touches all vertices) and is a "tree" in the graph-theoretic sense (it has no loops). For a given graph with N vertices, any spanning tree will have exactly N-1 edges (since adding any additional edge would create a cycle). There can be many different spanning trees possible for the same graph, especially if the graph is highly connected.

Minimum Spanning Tree (MST) Definition: When a graph has weights on its edges, each spanning tree has a total weight which is the sum of the weights of all the edges in that tree. A **minimum spanning tree** is defined as a spanning tree whose total weight is the smallest among all possible spanning trees of the graph (Cormen et al., 2009). In plain terms, it is the cheapest way to connect all the vertices. If you think of the vertices as cities and edges as possible roads with construction costs, a minimum spanning tree would be the network of roads that connects all cities for the lowest total cost.

It is important to note that the minimum spanning tree of a graph is not necessarily unique – there can be more than one spanning tree tying for the same minimum total weight if the graph has multiple edges with equal weights. However, if all edge weights are distinct (no ties), then there will be exactly one unique minimum spanning tree for the graph (Kruskal, 1956). Finding an MST is a classic optimization problem, and many algorithms have been developed to solve it efficiently. Next, we describe two primary algorithms for constructing an MST, known for their greedy strategy (they build the solution step by step, always choosing the next optimum local step): Prim's algorithm and Kruskal's algorithm.

Prim's Algorithm

Prim's algorithm is a greedy method that builds a minimum spanning tree by expanding outward from a starting vertex. The idea is to grow a single tree by always adding the least expensive edge that connects a vertex in the current tree to a vertex outside the tree, until all vertices are included. This approach was originally discovered by Vojtěch Jarník in 1930 and later independently by Robert Prim in 1957 (Graham & Hell, 1985). It is commonly known simply as **Prim's algorithm**. We will explain Prim's algorithm in a step-by-step manner:

1. **Start with an arbitrary vertex:** Begin with an empty set of edges and pick any one vertex to start the tree. At the start, the spanning tree consists of just this single vertex and no edges. (Because a single vertex by itself has no connecting edges yet, it is trivially a tree on its own.)
2. **Find the smallest edge from the tree to a new vertex:** Look at all the edges that connect the vertices already in the tree to those vertices not yet in the tree. Out of all these candidate edges, find the one with the smallest weight (the least cost). Because the tree initially has only one vertex, this step will simply pick the shortest edge emanating from that starting vertex to any other vertex.

3. **Add the smallest edge to the tree:** Take the minimum-weight edge identified in the previous step and include it in the spanning tree. This will bring one new vertex (the one that was connected by that edge) into the tree. Now the spanning tree has one more vertex than before, and one edge connecting that new vertex to the rest of the tree.
4. **Repeat the process:** Now that the tree has grown, repeat the selection process. At each iteration, consider all edges that connect any vertex in the current tree to any vertex outside the current tree. Again, choose the edge with the smallest weight among these. Add that edge and the new vertex it leads to into the spanning tree. Ensure that no cycles are formed – by always connecting to a vertex that was previously "outside" the tree, Prim's algorithm inherently avoids creating cycles.

5. **Continue until spanning tree is complete:** Keep repeating step 4 until all vertices are included in the spanning tree. When the algorithm terminates, we will have added N1 edges for N vertices, and all vertices will be connected in a tree structure.

Kruskal's Algorithm

Kruskal's algorithm is another greedy strategy for finding a minimum spanning tree, but it operates in a different manner. Instead of growing a single tree from a starting vertex, Kruskal's method builds the spanning tree gradually by considering all the edges in order of increasing weight and choosing the cheapest edges that do not form a cycle. This algorithm was introduced by Joseph Kruskal in 1956 and has since become a standard approach to the MST problem (Kruskal, 1956). The outline of Kruskal's algorithm in plain English is as follows:

1. **Sort all edges by weight:** Begin by examining all the edges in the graph and sorting them from the smallest weight to the largest weight. This gives an ordered list of candidate edges, starting with the most inexpensive connections.
2. **Start with an empty edge set:** Initially, the spanning tree is empty – no edges have been chosen yet. We will gradually add edges from the sorted list to this set, making sure we never form a cycle.
3. **Add edges in increasing order:** Iterate through the sorted list of edges, from lowest weight to highest:
   - For each edge, check if including that edge in the current set would create a cycle among the vertices that are already connected by the chosen edges.
   - If adding the edge does *not* form a cycle, accept this edge and add it to the spanning tree. If adding the edge *would* form a cycle, then skip this edge (do not add it), because including it would violate the tree structure.
4. **Use a union-find structure to detect cycles (conceptually):** As we add edges, we keep track of which vertices are connected together. One efficient way to do this in practice is by using a disjoint set union–find data structure, which can quickly tell whether two vertices are already in the same connected component (Cormen et al., 2009). In simple terms, the algorithm keeps track of groups of vertices that are connected by the chosen edges so far. When considering a new edge, we can determine if its two endpoints are already connected indirectly through the existing chosen edges. If they are, then adding this edge would create a loop. If they are not, the edge is safe to add.

5. **Continue until spanning tree is complete:** Continue scanning through the sorted edges, adding those that are safe (non-cycling). Eventually, once we have added enough edges such that all vertices are connected, the process stops. At that point, we will have a spanning tree. Just like before, for N vertices we will stop when we have N-1 edges chosen. Any remaining edges (usually the heavier ones or those creating cycles) can be disregarded.

By the end of Kruskal's algorithm, we have built a minimum spanning tree by always taking the next smallest edge that does not violate the tree conditions. The greedy choice at each step (taking the next smallest possible connection) is guaranteed to produce an MST due to the cut and cycle properties of spanning trees (Kruskal, 1956). In essence, Kruskal's algorithm exploits the fact that any valid MST must include the lightest edge connecting any partition of the vertices; conversely, a heaviest edge in a cycle can never be in the MST. Thus, by always choosing the globally smallest remaining edge and avoiding cycles, Kruskal's method ensures an optimal result.

In summary, Prim's and Kruskal's algorithms both find the minimum spanning tree by a series of greedy steps, but Prim's builds from a single starting point and grows a tree, while Kruskal's picks edges in global sorted order and builds a forest that eventually becomes a single tree. Both will be illustrated in the following case study.

Case Study: Example Construction of an MST

To better understand how Prim's and Kruskal's algorithms work in practice, let us consider a concrete example. Imagine we have a small network (graph) of five locations labeled A, B, C, D, and E. Suppose these locations are connected by roads with varying distances (or costs). We will describe the graph by listing the connections and their weights:

- A is connected to B with a road of cost 3.
- A is connected to C with a road of cost 1.
- A is connected to D with a road of cost 4.
- B is connected to C with a road of cost 2.
- B is connected to D with a road of cost 5.
- B is connected to E with a road of cost 6.
- D is connected to E with a road of cost 8.

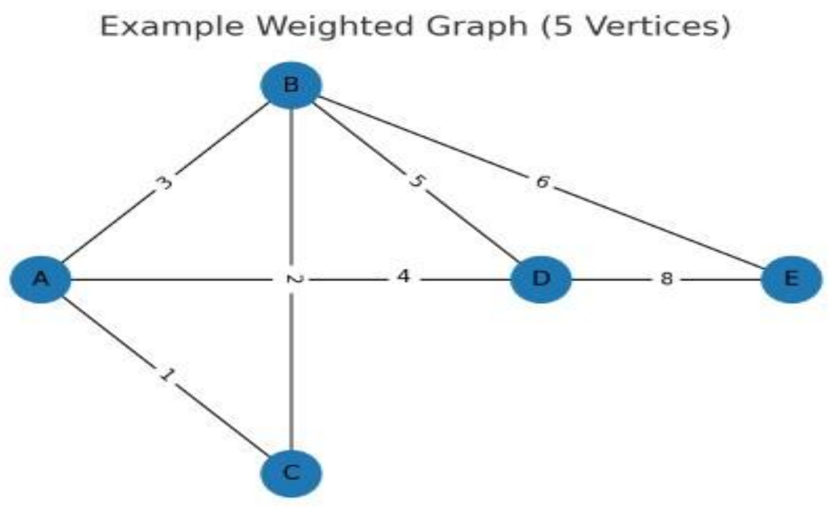

Figure 1 shows a diagram of this weighted graph with all five vertices and the seven edges connecting them (with the numbers indicating the weight or cost of each edge).

Our goal is to find the minimum spanning tree of this graph – that is, to select the subset of these roads that connects all five locations with the smallest total cost. We will apply Prim's algorithm and Kruskal's algorithm to this graph step by step, and verify that both methods arrive at the same final MST (as they should).

Prim's Algorithm on the Example:

Step	Edge Weight	Tree Vertices After Step	Total Weight
0 – Start	Start at A	{A}	0
1	A – C (1)	{A, C}	1
2	C – B (2)	{A, B, C}	3
3	A – D (4)	{A, B, C, D}	7
4	B – E (6)	{A, B, C, D}	13

Prim's algorithm has now finished, producing a spanning tree with edges {A–C, B–C, A–D, B–E}. It is easy to verify that this is indeed a tree (it connects all vertices and has no cycles) and that its total cost is 13. We should also check that this is minimum – there is no other way to connect all five vertices with a total cost less than 13. If we inspect the chosen edges, we see that at each step Prim's algorithm picked the smallest possible connecting edge, and it avoided adding the edges A–B and B–D because at the moments they were considered, those edges would have connected vertices that were already indirectly connected in the tree (which would have created a cycle). The final set of edges is the result of this greedy growth process.

Kruskal's Algorithm on the Example:

Now, let us apply Kruskal's algorithm to the same graph and see how the edges are selected in increasing order of weight:

- **Sorting edges by weight:** First, list all the edges with their costs and sort them from smallest to largest:
  1. A–C (cost 1) – smallest
  2. B–C (cost 2)
  3. A–B (cost 3)
  4. A–D (cost 4)
  5. B–D (cost 5)
  6. B–E (cost 6)
  7. D–E (cost 8) – largest

Step	**Edge Weight**	**Action**	**Edge Set**	**Running Total Cost**
0 – Start	**Start**	**N/A**	**N/A**	**0**
1	**A – C (1)**	**Add**	**{A – C}**	**1**
2	**B – C (2)**	**Add**	**{A – C, B – C}**	**3**
3	**A – B (3)**	**Skip (would form cycle A – B – C – A)**	**{A – C, B – C}**	**3**
4	**A – D (4)**	**Add**	**{A – C, B – C, A – D}**	**7**
5	**B – D (5)**	**Skip (Cycle)**	**{A – C, B – C, A – D}**	**7**
6	**B – E (6)**	**Add**	**{A – C, B – C, A – D, B – E}**	**13**
7	**D – E (8)**	**Skip (Tree already completed)**	**{A – C, B – C, A – D, B – E}**	**13**

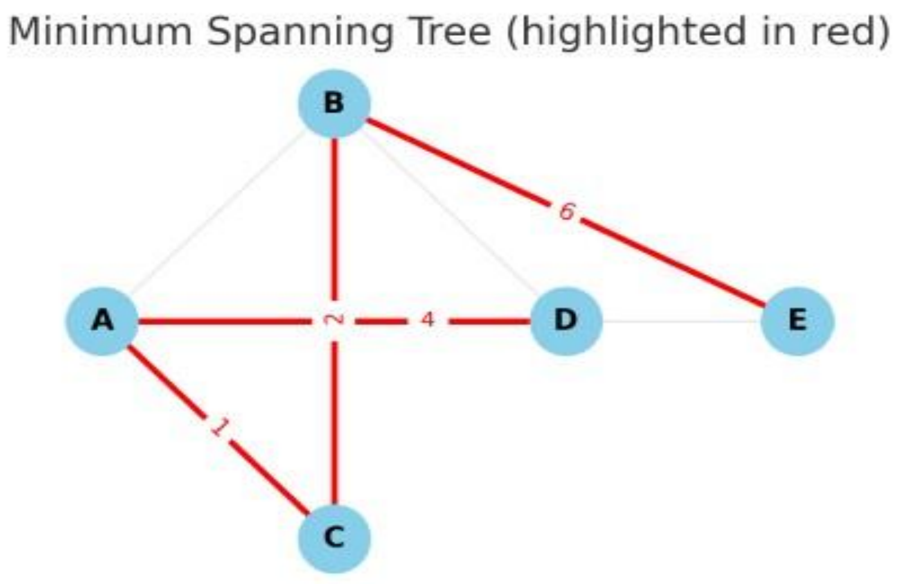

Figure 2 demonstrates that both algorithms, Prim's and Kruskal's, arrive at the same minimum spanning tree for the example graph, albeit through different processes.

Prim's algorithm added edges one by one by always extending the current tree with the cheapest outgoing edge, while Kruskal's algorithm picked edges in a global sorted order and avoided cycles. The final MST includes the connections from A to C, C to B (connecting A–B–C in a triangle but we exclude the heavier A–B edge), plus the connections from A to D and from B to E. Any other combination of roads connecting all cities would have equal or higher total cost than these selected ones. This example validates the correctness of both algorithms and provides a clear, step-by-step narrative of how each operates without the need for complex notation.

Warehouse Case Study

In a warehouse case study, Prim's minimum spanning tree (MST) algorithm was applied to design optimal picking routes connecting all required pick locations with the shortest possible total distance. By eliminating unnecessary travel through a minimal spanning network, the distance that order-picking trucks needed to drive was drastically reduced (e.g., a drop of 466 meters, from 693.3 m to 227.2 m, in one optimized picking round). This ~67% reduction in travel distance translated directly into faster picking times and lower energy consumption for the electric forklifts. Consequently, the optimized MST routes cut greenhouse gas emissions significantly – the case study reported an avoidance of approximately 234 kg of CO_2 for the consolidated picking scenario. In summary, using Prim's MST to streamline warehouse picking paths yielded substantial improvements in operational efficiency (shorter routes and times) while reducing energy usage and emissions, supporting more sustainable warehouse operations.

Conclusion

Minimum spanning trees are a fundamental concept in graph theory and have proven to be vitally important for efficient network design. They ensure that all required nodes are connected with the smallest possible total connection cost or distance, making them highly valuable for solving real-world infrastructure problems. Effective greedy algorithms such as Prim's and Kruskal's enable rapid computation of MSTs even for large graphs, each finding an optimal spanning network in roughly $O(E \log V)$ time. By using these algorithms, planners can obtain minimal-connectivity solutions that avoid wasteful routes or links. The case studies and examples discussed demonstrate that MST-based strategies consistently reduce costs and often decrease environmental impact across diverse domains – from optimizing warehouse pick paths to designing telecommunications, transportation, logistics, and utility networks. In summary, minimum spanning trees provide a powerful, concise framework for building efficient and sustainable connectivity in complex systems, combining theoretical optimality with tangible business and environmental benefits.