Analysis and Implementation of Boruvka's Algorithm

James Calnan

15/10/2023

Abstract

This report presents a comprehensive analysis of Boruvka's algorithm, a fundamental approach in graph theory for finding a connected graph's minimum spanning tree (MST). The report covers the algorithm's principles, pseudo code, complexity, limitations, and applications, followed by an implementation in Python.

I certify that all material in this report which is not my own work has been identified.

Student Number: 700032368

Word Count: 1497

1 Principles of Boruvka's Algorithm

Developed in 1926 by Czech scientist Otakar Boruvka for optimising electrical networks in Moravia, Boruvka's algorithm was one of the earliest approaches to solving the Minimum Spanning Tree (MST) problem. Its initial aim was to connect towns efficiently with electrical cables, setting a foundation in combinatorial optimisation and graph theory.

The algorithm gained broader recognition in the 1950s and 1960s, parallel to similar developments by Kruskal and Prim. Initially overlooked due to language and communication barriers, it became a significant part of algorithmic design, particularly with advancements in parallel computing and data structures. Boruvka's algorithm remains relevant today, demonstrating its enduring impact on computer science and network design.

Boruvka's algorithm has evolved significantly in modern contexts, particularly in distributed and parallel computing. As the demands for processing large-scale networks and big data have grown, the algorithm's ability to independently handle graph components has made it an invaluable tool. Today, it is often integrated with other algorithms to optimise performance for specific applications, showcasing its adaptability and continued relevance in the era of high-performance computing. [2]

2 Principles of Boruvka's Algorithm

Boruvka's algorithm operates on the principle of progressive reduction of graph components. Initially, every vertex forms a separate component. In each iteration, the algorithm identifies the smallest edge connecting each component to a different component and adds these edges to the growing Minimum Spanning Tree (MST). This process merges the components connected by these edges, effectively reducing the number of components in the graph. The algorithm repeats this process until all vertices are spanned by a single connected component, resulting in the MST.

The strength of Boruvka's algorithm lies in its local optimisation strategy. At every stage, each component independently selects the least expensive edge that connects it to a different component without needing global knowledge of the entire graph. This makes Boruvka's algorithm inherently parallelisable and efficient, particularly for sparse graphs. The algorithm ensures that no cycles are formed during the construction of the MST, as it always connects separate components and never adds an edge that closes a loop within a single component.

3 Pseudocode

```
function Boruvka(nodes, edges):
    Initialise each node as a separate component in sets
    Initialise an empty dictionary mst for the Minimum Spanning Tree
    while the number of components in sets is greater than 1:
        Initialise an empty dictionary min_edges for tracking minimum
           edges for each component
        for each edge in edges:
            Determine the components of the nodes connected by the edge
            If the nodes belong to different components:
                Update min_edges for each component if the current edge
                   is smaller than the existing minimum
        for each edge in min_edges:
            If the edge connects two different components:
                Add the edge to mst
                Merge the two components into a single component in sets
        Update the number of components in sets
    return mst
```

4 Complexity Analysis

The time complexity of Boruvka's algorithm is $O(E \log V)$, where E represents the number of edges and V the number of vertices in the graph. This logarithmic complexity arises because the number of components is at least halved in each iteration, leading to a maximum of $\log V$ iterations. In each iteration, the algorithm examines all edges to find the minimum weight edge for each component, contributing to the E factor in the time complexity.

In large-scale networks, this complexity implies that the algorithm scales efficiently with the number of nodes. However, as the network density (the number of edges) increases, the performance can be impacted due to the linear relationship with E. In dense networks, where the number of edges approaches V^2, the time to process all edges in each iteration may become substantial, potentially affecting the algorithm's practicality in such scenarios. Nonetheless, the parallelisable nature of Boruvka's algorithm can mitigate this impact, making it suitable for distributed computing environments often used in handling large-scale network data.

The space complexity of Boruvka's algorithm is influenced by the graph representation used. When an adjacency list is employed, the space complexity is $O(V + E)$, as it stores a list of edges for each vertex. In contrast, an adjacency matrix representation will have a space complexity of $O(V^2)$, as it requires storage for every possible edge between vertices, regardless of whether an edge exists. The choice between these representations impacts the space required and the efficiency of accessing and updating edge information during the algorithm's execution.

5 Limitations and Advances

While Boruvka's algorithm excels in sparse graphs, its efficiency diminishes in dense graphs due to the increased number of edges, resulting in higher computational overhead in each iteration. However, recent advances, particularly in parallel computing, have mitigated this limitation. Parallel implementations exploit the algorithm's inherent ability to process multiple components independently, allowing for simultaneous edge selection and component merging. Additionally, algorithmic optimisations, such as using advanced data structures for edge and set management, have enhanced its performance across various graph types, expanding the algorithm's applicability in complex network analysis and large-scale graph processing.

6 Applications

Boruvka's algorithm is helpful in various fields for constructing minimum spanning trees (MSTs), especially in electrical engineering for designing efficient circuits and power networks. It optimises cable layouts, reducing costs. In network design, it enhances communication network efficiency and reliability. The algorithm also aids in road network planning and pipeline layouts in industrial projects. In computer science, it is used in clustering algorithms for data analysis, utilising MSTs to identify data groupings. Boruvka's broad applicability in areas requiring efficient connectivity highlights its practical importance.

7 Algorithm Implementation

The implementation of Boruvka's algorithm in Python is demonstrated in Figure 3. The script includes functions for graph representation, Boruvka's algorithm execution, and visualisation.

7.1 Graph Representation

```
nodes = ['A', 'B', 'C', 'D', 'E', 'F']
edges = {
    ('A', 'B'): 3,
    ('A', 'E'): 1,
    ('A', 'C'): 6,
    ('B', 'C'): 5,
    ('B', 'D'): 9,
    ('C', 'D'): 7,
    ('C', 'F'): 11,
    ('D', 'E'): 2,
    ('E', 'F'): 3,
    ('F', 'B'): 2
}
```

Figure 1: Simple graph

7.2 Visualisation

```
# Need a way to print the graph so that it is easy to visualise using
   networkx
def print_graph(nodes, edges, layout=nx.shell_layout, save=False,
   filename='graph.png'):
    G = nx.Graph()
    G.add_nodes_from(nodes)
    for (u, v), weight in edges.items():
        G.add_edge(u, v, weight=weight)

    pos = layout(G)  # Get the position layout for nodes
    nx.draw(G, pos, with_labels=True, node_size=500, node_color='
       lightblue', font_size=8, font_weight='bold')

    # Draw edge labels
    edge_labels = nx.get_edge_attributes(G, 'weight')
    nx.draw_networkx_edge_labels(G, pos, edge_labels=edge_labels,
       font_color='red')

    if save: # If save is true, save the graph
        plt.savefig(f'mst_graphs/{filename}')

    plt.show()
```

Figure 2: Code to output the graph

7.3 Boruvka's Algorithm Function

```
def boruvka(nodes, edges):
    # Initialise each node as its own separate component
    sets = {node: node for node in nodes}
    mst = {}  # Dictionary to store the minimum spanning tree
    frames = []  # List to store frames for animation
    # Heap to store all edges in the order of their weights
    edge_heap = []
    for edge, weight in edges.items():
        heapq.heappush(edge_heap, (weight, edge))

    # Run the algorithm until the MST has n-1 edges
    while len(mst) < len(nodes) - 1:
        min_edges = {}  # Store minimum edge for each component

        # Iterate through the heap to find the minimum weight edge for
           each component
        for weight, (node1, node2) in edge_heap:
            set1, set2 = sets[node1], sets[node2]
            if set1 != set2:
                # Update minimum edge for the component if a smaller one
                   is found
                if set1 not in min_edges or weight < min_edges[set1][0]:
                    min_edges[set1] = (weight, (node1, node2))
                if set2 not in min_edges or weight < min_edges[set2][0]:
                    min_edges[set2] = (weight, (node1, node2))

        # Merge components and add edges to the MST
        for weight, (node1, node2) in min_edges.values():
            set1, set2 = sets[node1], sets[node2]
            if set1 != set2:
                edge = (node1, node2)
                mst[edge] = weight
                # Update the component set for the merged components
                new_set = min(set1, set2)
                for node in sets:
                    if sets[node] == set1 or sets[node] == set2:
                        sets[node] = new_set

    return mst  # Return the minimum spanning tree
```

Figure 3: Boruvka's Algorithm

8 Results and Discussion

The algorithm was tested on various graphs, demonstrating its ability to find the MST efficiently. The visualisation component provides an intuitive understanding of the algorithm's operation; the MST calculated from Figure 4 can be seen in Figure 5. The graph represented in Figure 1 can be seen in Figure 4, created by the code in Figure 2.

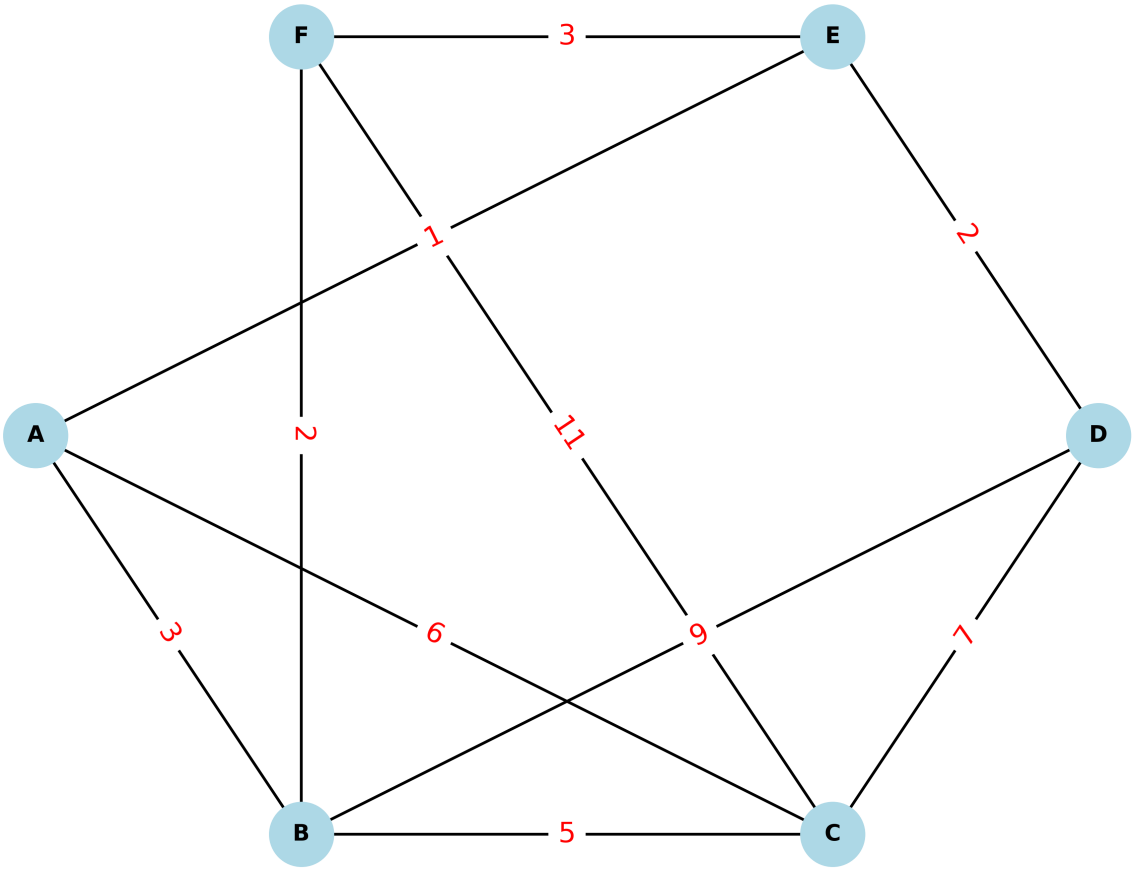

Figure 4: Graph representation

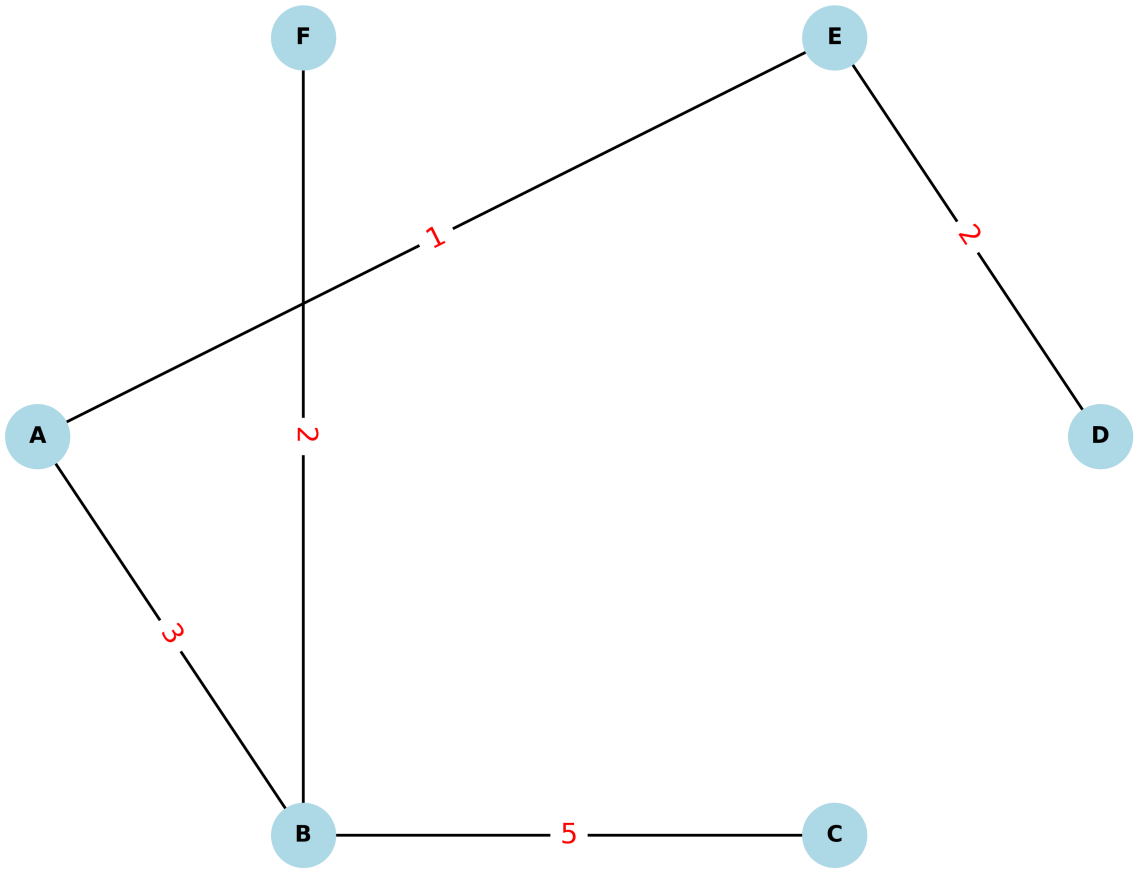

Figure 5: MST for graph from Figure 1

9 Application to Road Network Data

Borůvka's algorithm was applied to the road network data of San Joaquin County, sourced from the Spatial Dataset Repository [1]. The objective was to construct an MST for the region's road system, optimising the total road length and potentially reducing construction and maintenance costs.

9.1 Results

The application of Borůvka's algorithm to the road data significantly reduced in the total weight of the network. The original graph had a total weight of 831,572.31, while the MST had a total weight of 531,061.62, marking a reduction of 300,510.69.

Original Graph Weight	Minimum Spanning Tree (MST) Weight
831,572.31	531,061.62

Table 1: Results of Applying Borůvka's Algorithm to San Joaquin County Road Data

9.2 Graphical Representations

Graphical representations of the original road network (Figure 6) and the resulting MST (Figure 7) are provided below, illustrating the optimisation achieved through the algorithm.

These visualisations demonstrate the practical application of Borůvka's algorithm to real-world data, underscoring its potential in infrastructure planning and optimisation.

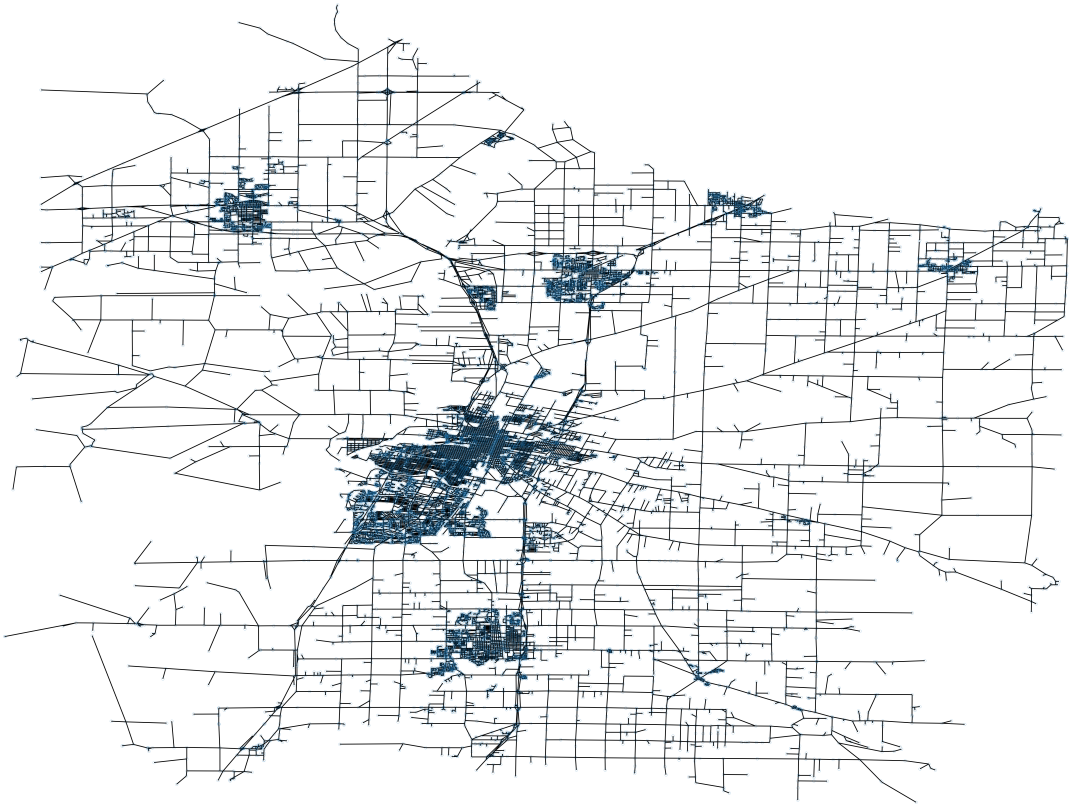

Figure 6: Original road network of San Joaquin County

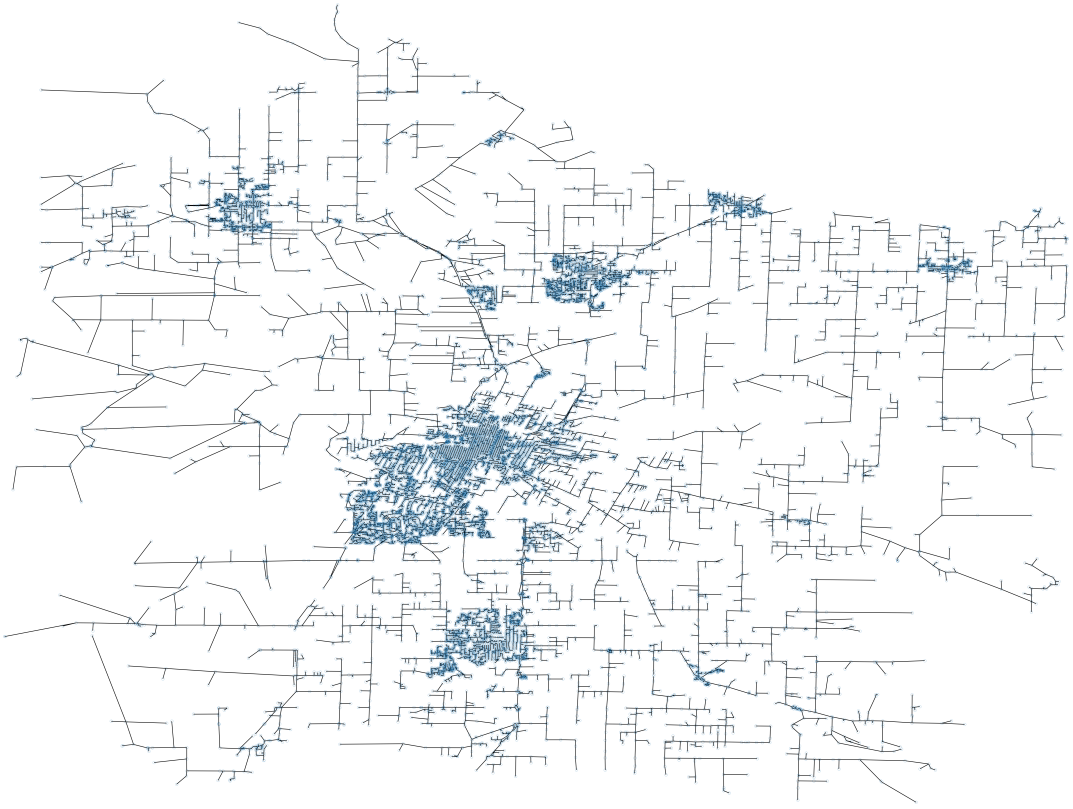

Figure 7: Optimised MST of San Joaquin County road network

10 Conclusion

Boruvka's algorithm remains a cornerstone in graph theory for efficiently solving minimum spanning tree problems. Its application extends from electrical engineering to network design, underscoring its lasting relevance and adaptability in diverse fields. Despite limitations in dense graphs, ongoing enhancements in computing continue to bolster its practical utility.

References

[1] Spatial dataset repository. https://users.cs.utah.edu/~lifeifei/SpatialDataset.htm. Accessed: 10-Dec-2023.

[2] Wikipedia. Borůvka's algorithm — Wikipedia, the free encyclopedia, 2023. [Online; accessed 10-December-2023].