Analysis and Implementation of Boruvka's Algorithm

James Calnan

13/10/2023

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: 1478 (not incuding title page and references)

1 Principles of Boruvka's Algorithm

Developed in 1926 by Otakar Boruvka for optimising electrical networks, Boruvka's algorithm was a pioneering method for solving the Minimum Spanning Tree (MST) problem. It aimed to efficiently connect towns with electrical cables, thus laying a foundational role in combinatorial optimisation and graph theory. [2]

The algorithm's essence lies in its approach to progressively reduce graph components. Each vertex starts as an independent component, and in each iteration, the smallest edge connecting each component to a different one is selected and added to the growing MST. This procedure merges these components, continually lessening the graph's component count until a single connected component spans all vertices.

This unique local optimisation strategy, where each component independently chooses the least expensive connecting edge without global graph knowledge, gives Boruvka's algorithm inherent parallelizability and effectiveness, especially in sparse graphs. It guarantees a cycle-free MST, solely connecting distinct components avoiding any loop formation within a component.

Gaining broader recognition in the mid-20th century alongside developments by Kruskal and Prim, Boruvka's algorithm overcame initial overlooks due to language barriers and became integral in algorithmic design. Today, its application in distributed and parallel computing is significant, particularly for handling large-scale networks and big data. Its integration with other algorithms for performance optimisation underlines its adaptability and ongoing relevance in modern high-performance computing contexts.

2 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
```

See Figure 8 for a flowchart.

3 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 complexity is due to two things: firstly, the algorithm examines all edges in each iteration to find the minimum weight edge connecting different components, accounting for the E factor. Secondly, the number of components is at least halved in every iteration, leading to a logarithmic factor of $\log V$. This halving is due to each component independently selecting a minimum-weight edge, resulting in the progressive merging of components.

In densely connected networks, where E approaches V^2, the time complexity becomes significant, as each iteration involves processing many edges. This can impact the algorithm's efficiency, particularly in scenarios with highly dense networks. However, the inherent parallelism of Boruvka's algorithm allows for distributed

computation, making it well-suited for modern high-performance computing environments, especially in handling large-scale and complex network structures.

The chosen graph representation influences the space complexity. Using an adjacency list results in a space complexity of $O(V + E)$, suitable for sparse graphs as it stores edges adjacent to each vertex. On the other hand, an adjacency matrix, with a space complexity of $O(V^2)$, is less space-efficient for sparse graphs but facilitates quicker edge access, beneficial in scenarios where rapid edge lookups are crucial. Boruvka's algorithm typically utilises an adjacency list representation due to its efficiency in sparse graphs, which is common in many real-world applications.

4 Limitations and Advances

While effective in sparse graphs, Boruvka's algorithm faces challenges in dense graphs where the profusion of edges increases computational overhead in each iteration. This limitation is increasingly addressed through parallel computing techniques, leveraging the algorithm's capability for processing multiple graph components independently. Simultaneous edge selection and component merging in a parallel framework substantially improve efficiency. Additionally, employing advanced data structures for managing edges and sets further optimises performance, making the algorithm more versatile for complex network analysis and large-scale graph processing.

5 Applications

Boruvka's algorithm is pivotal in constructing minimum spanning trees (MSTs), finding extensive applications across various fields. Electrical engineering is instrumental in designing optimal circuits and power networks, reducing material and operational costs through efficient cable layouts. In network design, the algorithm enhances the efficiency and reliability of communication networks. It also plays a crucial role in road network planning and pipeline layouts for industrial projects, optimising connectivity and resource allocation. Furthermore, in computer science, Boruvka's algorithm underpins clustering algorithms for data analysis, employing MSTs to discern natural groupings in complex data sets. These diverse applications underscore the algorithm's significance in domains necessitating efficient and optimised connectivity solutions.

6 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.

6.1 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

6.2 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

6.3 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

7 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, which was created by the code in Figure 2.

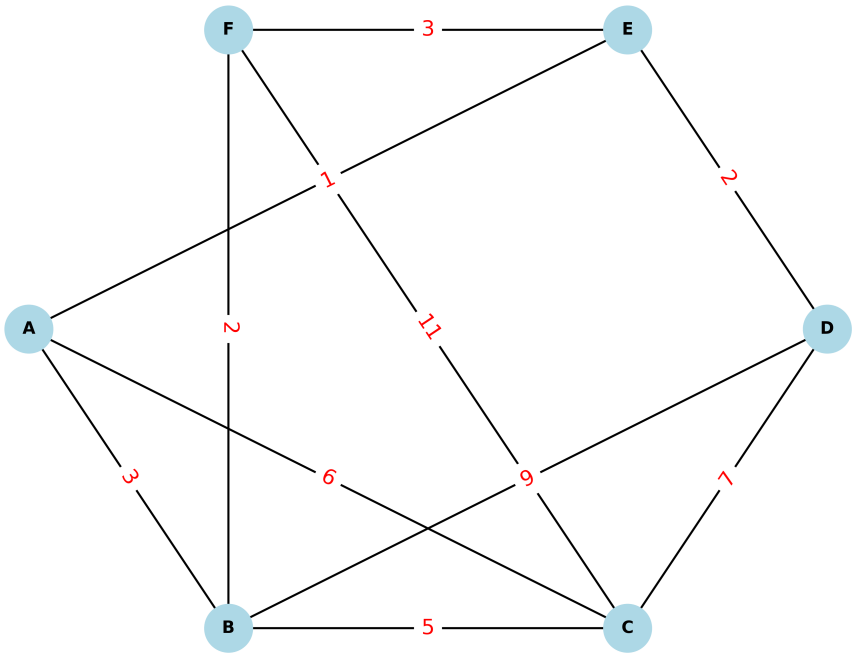


Figure 4: Graph representation

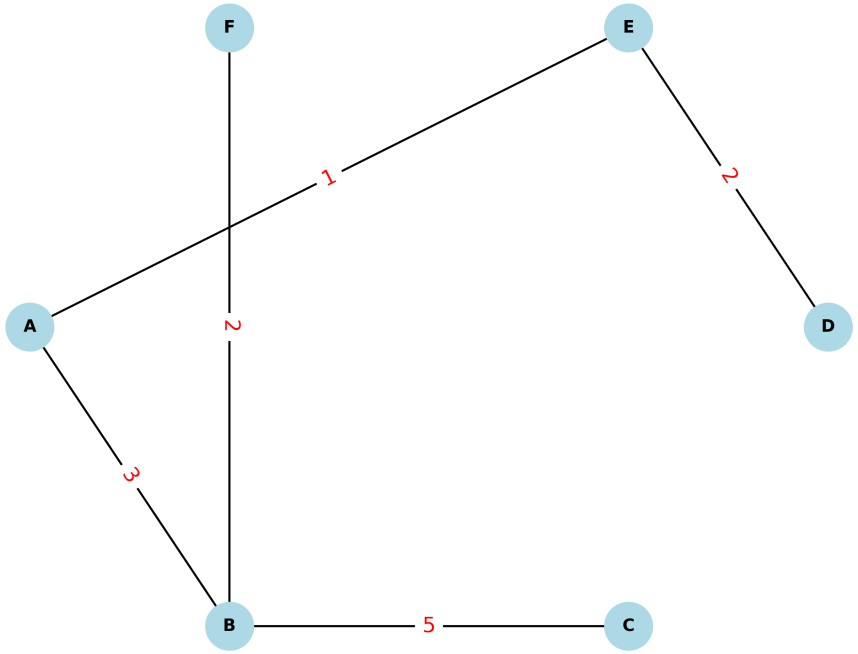


Figure 5: MST for graph from Figure 1

8 Another Example

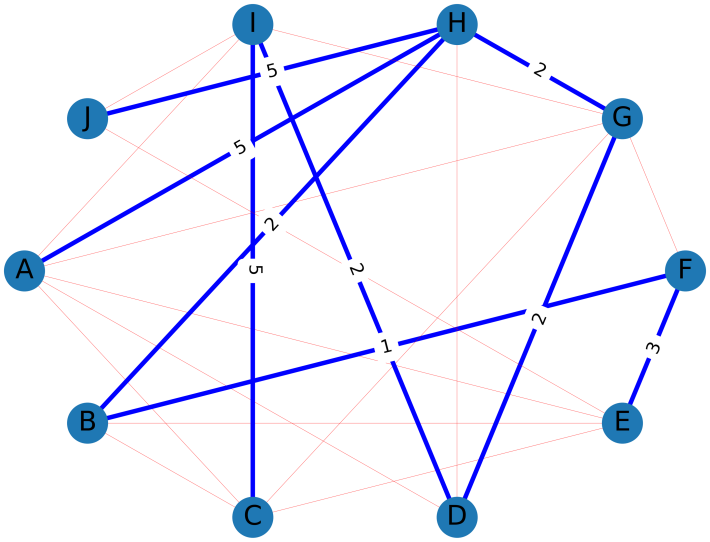


Figure 6: Larger Graph

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 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

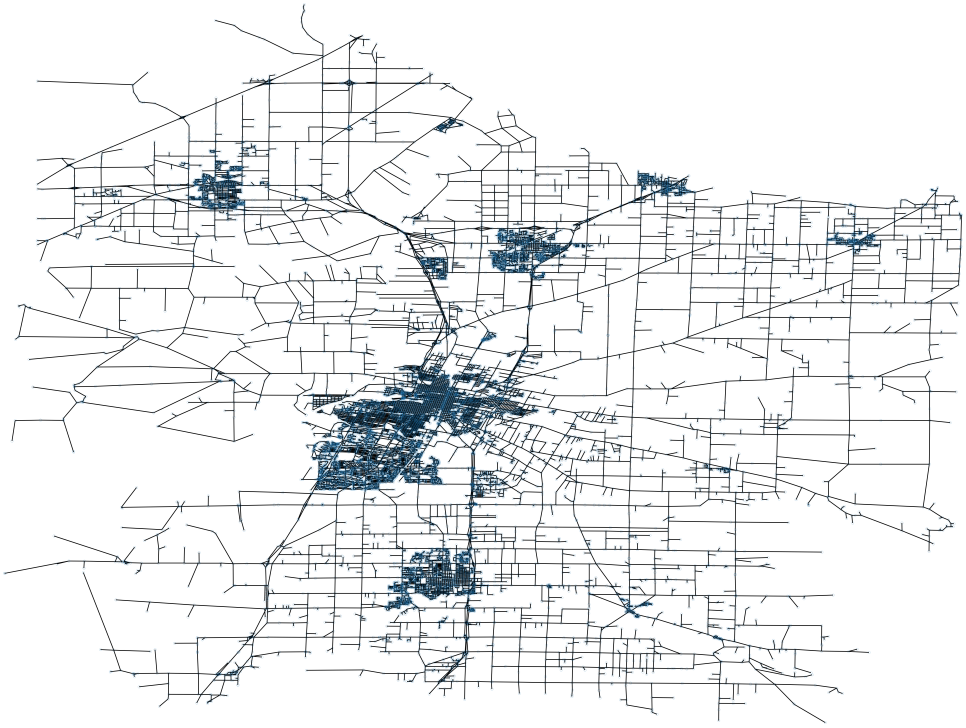
9.2 Graphical Representations

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

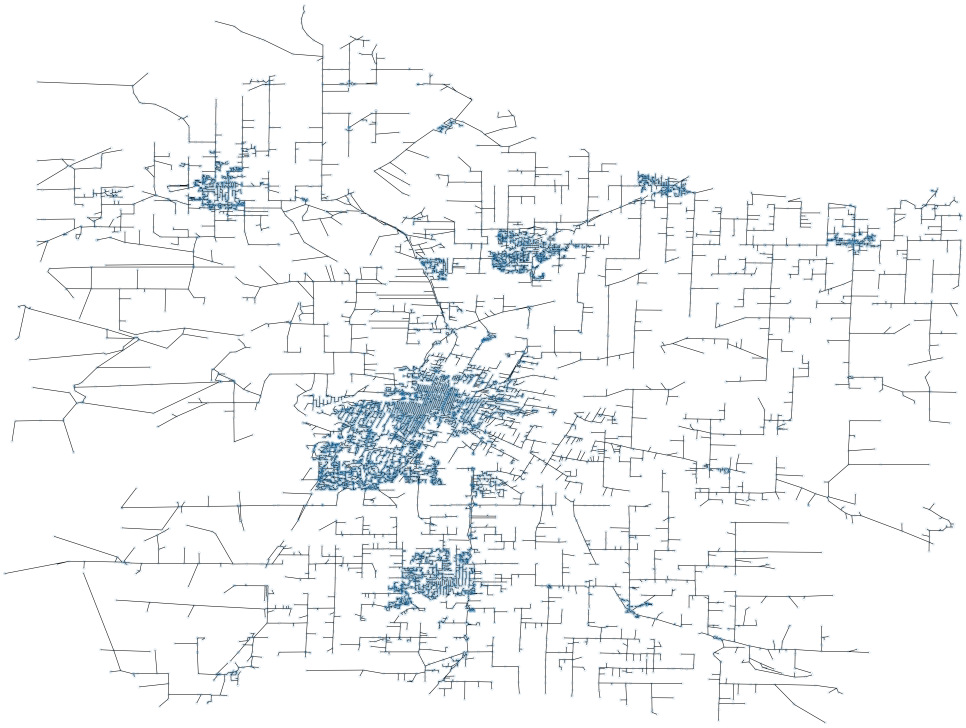
These visualisations demonstrate the practical application of Borůvka's algorithm to real-world data, underscoring its potential in infrastructure planning and optimisation.

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.



(a) Original road network of San Joaquin County, weight: 831,572.31



(b) Optimised MST of San Joaquin County road network, weight: 531,061.62

Figure 7: Road network of San Joaquin County

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. https://en.wikipedia.org/wiki/Bor%C5%AFvka%27s_algorithm, 2023. [Online; accessed 10-December-2023].

A Appendix

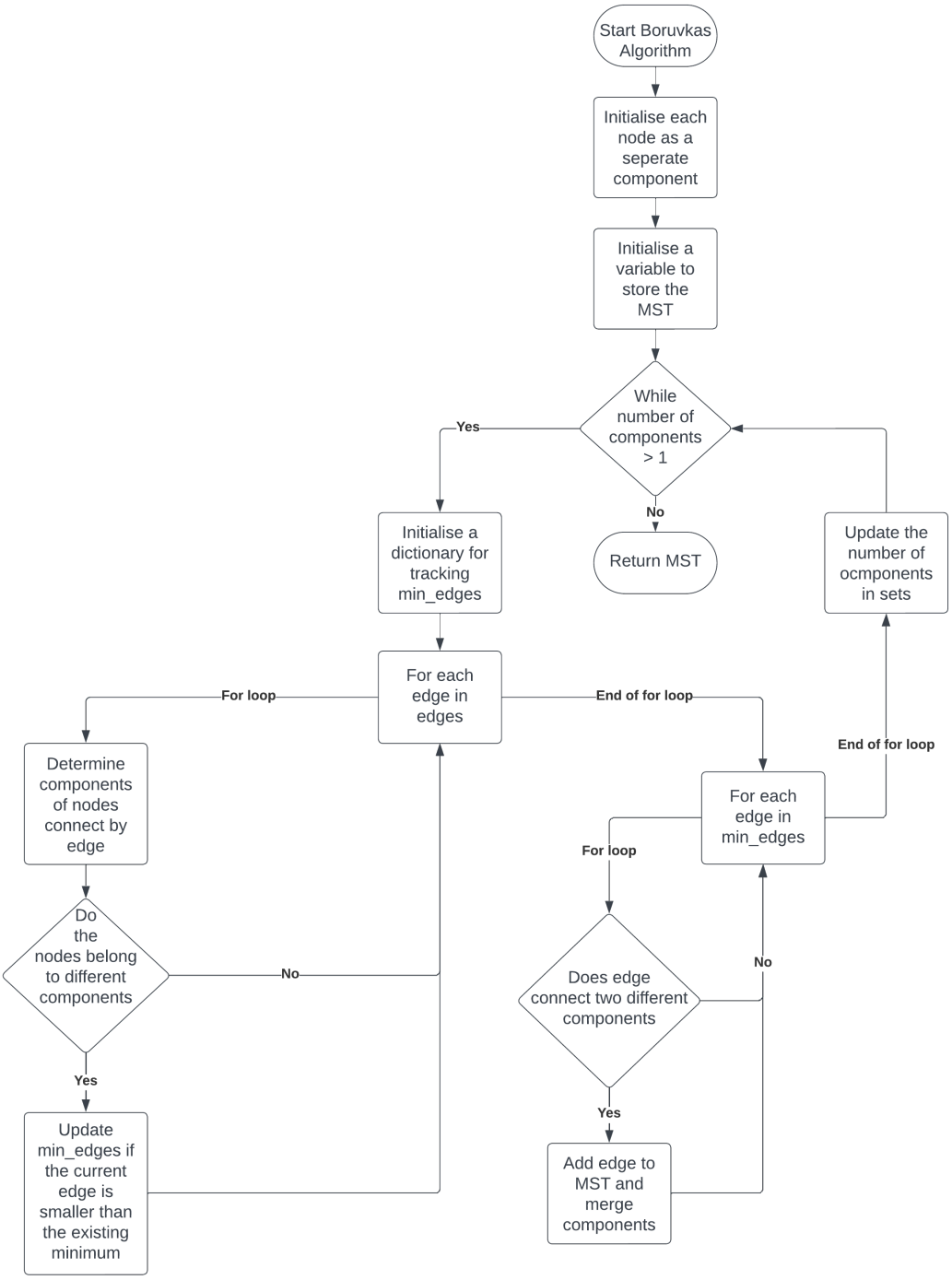


Figure 8: Algorithm Flowchart