

Minimum Spanning Tree Algorithms: Comparing Architecture Performance

Team 1: Monica Mooney, Chris Clayton, Logan Muir, Davis Pham, Brittany Pham

INTRODUCTION

Look outside. Minimum spanning trees are everywhere! The task of finding and calculating minimum spanning trees is an integral problem in today's world. As our most pertinent technological problems are solved and we work towards problems facing the analysis, calculations, and manipulations behind larger data sets, networks, and general graph theory problems, it is necessary for modern day computer scientists to focus their attention on the optimization of Minimum Spanning Tree algorithms. Within this project, we sought out to uncover the structure and usage behind common MST algorithms. Our focus originally extended towards testing the performance of the serial versions of Prim's, Kruskal's and Boruvka's algorithms before migrating our approach to test optimized versions of these algorithms.

Boruvka's algorithm is a MST algorithm in which the MST is produced through a recursive search and selection of the cheapest outgoing edge. In Kruskal's algorithm, a more cautious approach is taken as all of the graph edges are sorted and then minimally selected and added to a new graph, ensuring that no cycles are created. In Prim's algorithm, a random node is selected,

and a tree is built by adding the cheapest outgoing edge with each iteration.

As a whole, we were interested in seeing the performance results these algorithms would produce through serial and optimization test processes because while each succeeds in their production of a minimum spanning tree, their methodologies were varied and could potentially respond differently to our manipulation and optimization procedures.

LITERATURE REVIEW

Before fully committing to our project goals and outline, we conducted a thorough literature review to better understand current development in this area. In the most general definition, the Minimum Spanning Tree problem seeks to find the minimum weight spanning tree when connecting all vertices of a connected, undirected graph.

We wanted to learn what behaviors can be expected of these algorithms and conduct our tests with this information in mind. Thus, we researched our three algorithms to understand under what circumstances would each perform most efficiently.

We knew that we wanted to challenge ourselves and test these algorithms on the EECIS GPUs, and so we

delved into literature that discussed parallelized versions of these algorithms. Much of the research crowned Boruvka's algorithm as the best for parallelization in comparison to Prim's and Kruskal's. The Bulk Synchronous Parallel (BSP) model is a model for designing parallelized algorithms that uses "either a vertex centric or edge centric method to compute graph data in sequential consecutive steps" (Chu). This model explains parallelized Boruvka's performance. Because each vertex is independent from the other in this algorithm, parallelized Boruvka's can tackle the sequential version as a search and merge (Chu).

On the other hand, Prim's algorithm, we learned, is most difficult to parallelize. The inner loop of the algorithm is used to find the cheapest outgoing edge from a given node, and it is this loop that most researchers have found difficult to parallelize (Wang et al). One study found that even loop unrolling did little to improve its performance (Mariano). Furthermore, speedup of Prim's on GPUs was limited with a max 3x speedup (Mariano).

Prim's algorithm, while bad for parallelization, was reported to be best suited for dense graphs and complete graphs. Furthermore, the Johnson implementation of Prim's algorithm that uses d-heaps works well on both dense and sparse graphs (Bazlamaçci). However, the most common implementation uses an adjacency matrix. Contrasting the behavior of Prim's algorithm, Kruskal's performs best on sparse graphs. Because the algorithm sorts the edges of the graph, it has better

performance when edges are passed into it already sorted or in an order than can be sorted quickly (Bazlamaçci).

In terms of scalability, Kruskal's outperforms Prim's because Prim's has a high communication cost (Lončar). When comparing Kruskal's and Boruvka's, the literature shows that Kruskal's algorithm is more efficient than parallelized Boruvka's when used on a smaller set of data since parallelized algorithms require more time to share messages and status updates and to recolor (Chu).

We also came across literature that discussed the effect memory access patterns have on performance. For example, Prim's irregular memory access patterns affect its overall performance. The connection between memory and performance was influenced some of the testing methods we found. In one paper, researchers used memory as their limited factor since the amount of memory directly affects runtime.

HYPOTHESIS

Once we felt that we were up to date on the current state of our problem and understood the significance of these types of algorithms, we began to develop our hypothesis for the project. Our projected scope was large, with goals of testing the serial versions of each MST algorithm; Prim's, Kruskal's and Boruvka's, before developing parallelized implementations of these algorithms for testing purposes. In the span of our ten week time frame, we eventually scaled back our scope and refocused our efforts to testing the most commonly used MST algorithms.

	M. Mooney	B. Pham	D. Pham	C. Clayton	L. Muir
OS	Windows	Windows	Mac	Windows	Mac
Year of Production	2014	2017	2011	2014	2017
CPU	Intel Core i7	Intel Core i5	Intel Core i5	Intel Core i7	Intel Core i5
Clock Speed	2.6 GHz	2.3 GHz	2.4 GHz	2.4GHz	2.3 GHz
CORES	2	2	2	2	2
THREADS	4	4	4	4	4
RAM	8GB DDR3	8GB DDR3	4GB DDR3	8GB DDR3	8GB DDR3

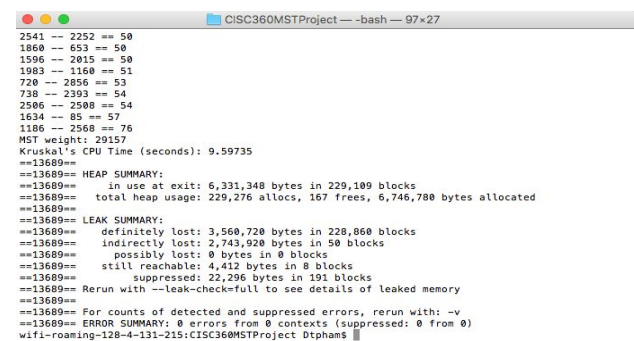
Figure 1: Architecture Specifics

Through our literature review and general analysis of the structure of each algorithm, we chose to prioritize our testing on Boruvka's and Kruskal's as the loop structure of these algorithms were such that it would be easier to optimize the structure of the serial version and later translate into parallelized code. After analyzing and comparing our different architectures, we predicted that the generation of the CPUs, namely i7 versus i5, would most influence the run time of the MST algorithms on our architectures. Furthermore, we also predicted that the parallelized versions of our MST algorithm implementations would run fastest on the GPU. Additionally, we believed that Kruskal's would run faster overall than Boruvka's given the nature of the algorithm.

TESTING

Following the development of our hypothesis, we divided our testing plan into

two sections: serial and optimized testing. For serial testing, we used our serial Boruvka and Kruskal algorithms and measured CPU execution time on each of our group members respective architectures (refer to Figure 1). For optimized testing, we planned to implement parallelized Kruskal and Boruvka algorithms in CUDA and test the execution time on EECIS NVIDIA Tesla GPU.



```

CISC360MSTProject --bash -- 97x27
2541 -- 2252 == 50
1860 -- 653 == 50
1596 -- 2615 == 50
1983 -- 1168 == 51
720 -- 2856 == 53
738 -- 2393 == 54
2586 -- 2588 == 54
1634 -- 85 == 57
1186 -- 2568 == 76
MST weight: 29157
Kruskal's CPU Time (seconds): 9.59735
==13689==
==13689== HEAP SUMMARY:
==13689==    in use at exit: 6,331,348 bytes in 229,189 blocks
==13689==   total heap usage: 229,276 allocs, 167 frees, 6,746,788 bytes allocated
==13689==
==13689== LEAK SUMMARY:
==13689==    definitely lost: 3,568,720 bytes in 228,860 blocks
==13689==    indirectly lost: 2,743,920 bytes in 50 blocks
==13689==    possibly lost: 0 bytes in 0 blocks
==13689==    still reachable: 4,432 bytes in 8 blocks
==13689==    suppressed: 22,296 bytes in 191 blocks
==13689== Rerun with --leak-check=full to see details of leaked memory
==13689==
==13689== For counts of detected and suppressed errors, rerun with: -v
==13689== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
wifi-roaming-128-4-131-215:CISC360MSTProject Dtpah$

```

Figure 2: Example Valgrind Output

As recommended, we executed our code using the Valgrind Tool. Valgrind is a system for debugging and profiling Linux programs. The Valgrind Tool Suite can automatically detect memory management

and threading bugs in order to maintain a more stable program. The most commonly used tool from Valgrind, which we utilized ourselves, is the Memory Check. Figure 2 is the result we retrieved after running our program through Memory Check. This provided us the information about the Heap Summary and Leak Summary for our code. The summary informed us that there is a massive memory leak within our program. In an ideal program, there would be “0 bytes lost in 0 blocks” for each listed category. However, our program resulted in “3,560,720 bytes lost in 228,860 blocks” that were definitely lost and “2,743,920 bytes lost in 50 blocks” that were possibly lost. We have acknowledged that the reason for the huge memory leak was due to the lack of manual garbage collection in our code. For each new data type we created, we did not delete each type to clear the heap/stack. Our intention was to modify our code to mitigate the memory leak, however, since we had issues creating a graph generator that was compatible for each of the algorithms, we decided to spend the majority of our time fixing that instead. Refer to Appendix A for code.

SERIAL TESTING

For testing the serial versions of each algorithm, our time was split between code generation and manual tests. We first located the serial version of each algorithm and after several meetings discussing structural analysis, input and output, and the general steps of each program, we were able to devise a plan of action for ensuring that

these algorithms were producing proper output.

One of our biggest challenges before going into serial testing was modifying each program so that each algorithm would take in the same graph be able to analyze it properly. This, along with the large number of graphs that would be required for substantial data, led to the idea of a random graph generator that would return a graph object, which would then be passed into each of the algorithms for analysis. We discovered a random graph generator and modified it to fit our requirements. The code we found for Boruvka’s algorithm and Kruskal’s algorithm both used the same graph class, so we extracted that into a separate file. The problem that arose was that Prim’s algorithm takes in an adjacency matrix, which, with our graph class, was not possible for us to extrapolate given our timeline and implementation, so we were not able to test that algorithm.

Furthermore, density was not able to be kept uniform, as our computers could handle a maximum of 100,000 edges (which, when creating a graph of 3,000 vertices, results in a density of a mere 2.22%). To maintain consistent results, we tested on a range of edges that each machine could successfully process for every execution.

Another issue we ran into was that both Boruvka’s and Kruskal’s algorithm assumed that the graph passed in contained a minimum spanning tree. Not realizing this, we invested a large amount of time in searching for a (non-existent) bug causing the algorithms to run in an infinite loop.

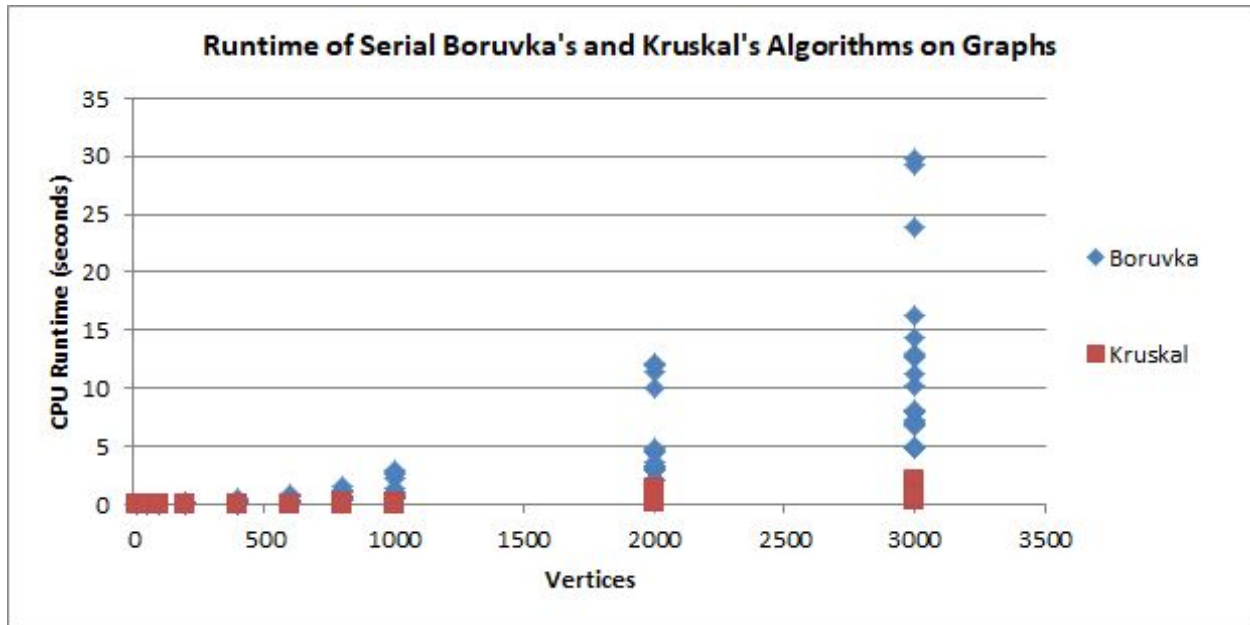


Figure 3: Serial Boruvka's vs. Serial Kruskal's

The fix was simple - we kept track of the number of trees in our solution, and if, after all edges were added, that value was not 1, we returned that there was no minimum spanning tree in the graph. This allowed us to run many iterations of our code on different input graphs without worrying about infinite loops.

The facts that the code for our algorithms did not all take in the same graph format, and that our machines could not handle large numbers of edges, made obtaining data and evaluating each algorithm very difficult.

SERIAL RESULTS

The serial testing for Kruskal's and Boruvka's algorithms on our varying respective architectures yielded interesting data about the algorithms themselves and the variances between the architectures.

Running both the serial Kruskal's and Boruvka's algorithms on each of our architectures, we found that Kruskal's overall ran much quicker than Boruvka's, as we hypothesized correctly. We ran both algorithms 10 times each on intervals of 10, 50, 100, 200, 400, 600, 800, 1000, 2000, and 3000 vertices, with the number of edges equal to $v(\sqrt{\log(\text{base } 2)(v^3)})$. This formula for calculating edges, although not ideal, was able to run on each of our architectures so we could yield results. We found that the disparity in run time between the two algorithms began to grow as the number of vertices increased. For example, while the algorithms differed in about 2.00 seconds for overall execution time on graphs of 1000 vertices, Kruskal's ran at times 25.00 seconds faster than Boruvka's on graphs of 3000 vertices, despite Kruskal's use of bubble sort.

Executing the algorithms on our different architectures led to varying

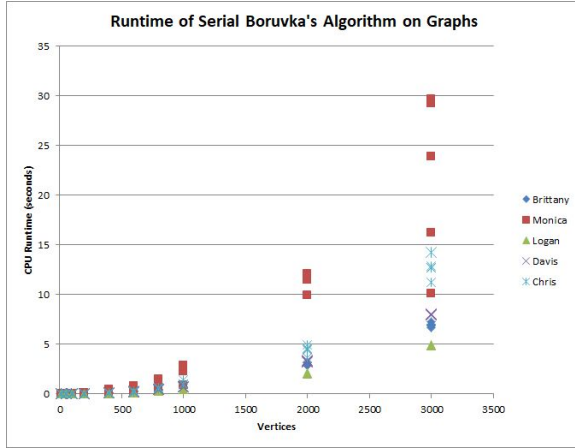


Figure 4: Serial Boruvka's across architectures

performance results. Logan's 2017 Macbook Pro ran the algorithm for all input graph sizes, followed by Brittany's, Davis's, Chris's and Monica's. Just as with comparing the two algorithms, the difference in execution times began to grow as the input graph sizes increased. For example, with graph sizes of 600 vertices and below, all of the architectures executed Boruvka's algorithm within 1.00 second of each other. When comparing the largest graph input size, 3000 vertices, Logan's

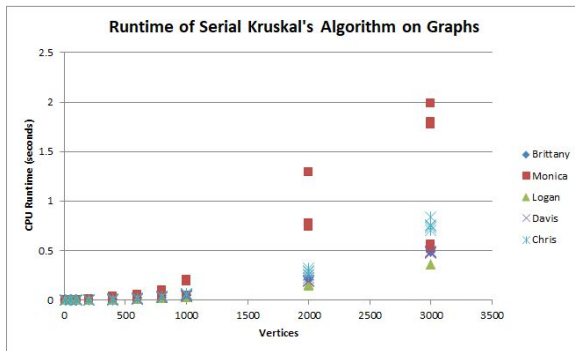


Figure 5: Serial Kruskal's across architectures

computer (fastest) and Monica's (slowest) differed by about 25.00 seconds for Boruvka's algorithm. Similar results occurred for Kruskal's algorithm, but given

the nature of the algorithm, the runtimes were still much lower (sub 3.50 seconds).

PARALLELIZATIONS

When we transitioned into the optimized algorithm testing, given the time constraint of our project, we set out to test a Boruvka CUDA implementation we had found in an open source GitHub repository. From our own experience with the Boruvka algorithm the understanding of other research within this area, Boruvka's algorithm has been notably performed the best when parallelized. We hypothesized that the parallel implementation of this algorithm paired with the high performance capabilities of the EECIS GPU machines would produce outstanding results for our data set.

Our first step in building our optimized algorithms was finding a code source that would utilize similar structures for graph generation and testing to those of our serial algorithms. Given our initial focus on the proposed optimizations of Boruvka's and Kruskal's, we decided to focus on testing the optimization of Boruvka's before Kruskal's seeing that Kruskal's has follows a "highly sequential workflow". With this in mind, we found a "parallel, platform, independent variant of Boruvka's algorithm" which we were to use for our first tests (Sousa). This program seemed to be highly suitable for our needs, in that it would require less refactoring than our original serial algorithms. In addition, this parallelized code used an adjacency list to represent the MST; a structural change that

would have further optimized our results had we implemented it in our serial algorithms. This optimized Boruvka's met our standards as it had the capabilities of loading in user generated graphs, allowing us to manipulate our test data.

When we moved further in compiling this project, we ran into several issues. Several of the main header files were not included in the repository, so while this code would require little refactoring to reflect our standards for output, it would require a lot of time to recreate certain header and other included files to run successfully.

GPU TESTING

Our next plan of action was to test the serial algorithms we had on the GPUs available through the EECIS department. We were granted access to the NVIDIA Tesla K40c machine for our testing purposes. Given the nature of the computing capabilities of a GPU machine; their ability to handle large datasets and perform the same calculations repeatedly is of good use to our Minimum Spanning Tree algorithms. Seeing as MST algorithms rely heavily on the same pattern; specifically in Boruvka's where the same search for the cheapest outgoing edge is captured.

Comparing several key architectural aspects of this architecture to our best performing CPU, several factors make it clear why the GPU outperformed the CPU. Specifically, the clock speed of the GPU was nearly double that of the CPU and its

massive 12GB memory allowed the GPU to tackle higher graphs of higher densities that crashed otherwise when testing on our CPUs (NVIDIA).

With our success in testing on a single GPU, we moved forward in testing on multiple GPUs. While our attempts at this were enthusiastic, our lack of a scheduling algorithm made our tests invalid as we were never able to schedule execution between multiple GPUs. Our research into dynamic load balancing on multiple GPUs gave us foresight into several problems that may have hindered our success in testing on multiple GPUs, specifically in that we were currently unable to efficiently allocate data across GPUs. It is unfortunate that we were unable to overcome this problem; however, the experience we did gain in our experimental testing on a singular GPU was perhaps more valuable in the final outcome of our project.

GPU RESULTS

Evident in Figure 6, comparing the runtimes of serial MST algorithms on a CPU vs those of a GPU gave us substantial results. Running the serial algorithms on the GPU allowed us to view the importance of several key performance details, including memory allocation and clock speeds. As the range of our vertices increased with larger graphs, we watched as the performance on our CPUs faltered and latency was increased. These results also alluded to the importance of memory on a machine whereas in this case the 12GB memory of the GPU allowed the NVIDIA machine to test larger datasets.

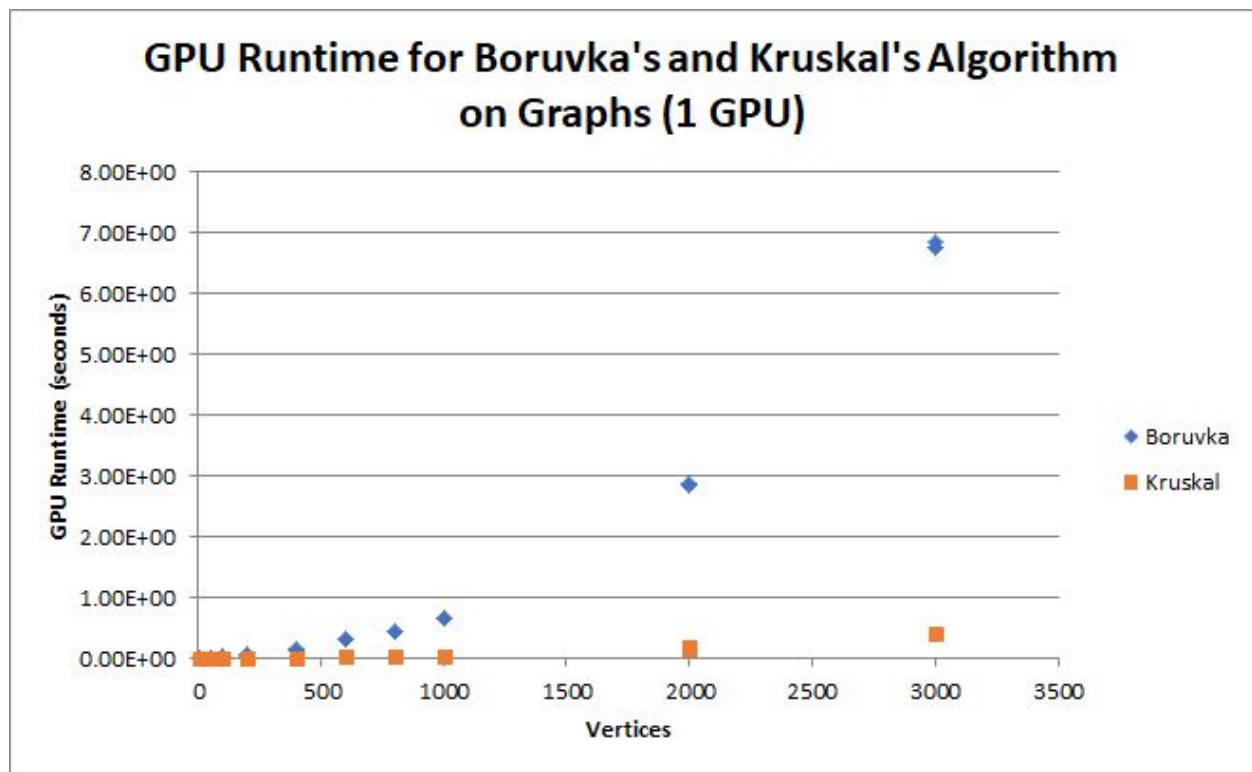


Figure 6: Serial algorithms on 1 GPU

These results met our original hypothesis and exceeded our proposed performance expectations.

TAKEAWAYS

From our successes and failures throughout this project, we have undoubtedly learned a plethora of lessons that can assist us in furthering this project and future related endeavors. Our most valuable takeaways include:

- Knowledge of real-world applications of popular MST algorithms
- Methods of refactoring and optimizing borrowed code
- Different implementations of graph generation

- Drawing conclusions from structural analysis
- Increased understanding of CUDA compilers, Valgrind, and Papii
- Enhanced C++ programming skills
- Effective teamwork practices

CONCLUSIONS

After completing our testing, we were able to confirm several of our original hypotheses and form new ones from what we learned. We proposed that Kruskal's algorithm would be more efficient than Boruvka's algorithm based on our benchmarks and the nature of the algorithm. Our own research shows that Kruskal's is indeed faster than Boruvka's especially on larger sized graphs. In terms of computer architecture, we

originally predicted that the CPU generation (i.e. i7 versus i5) would be a factor; however, we found that each machine's year of production was a leading factor in performance. We also observed that running Boruvka's algorithm performed much better on the GPUs than it did on the CPUs.

Although we ran into issues while trying to run our parallelized code, we met all of our major project goals. We learned more about each algorithm and improved our understanding of each. We answered our major question, namely "Which algorithm is most efficient?". In our testing, we made sure the input sizes used for each

algorithm were the same. Additionally, we made sure the evaluations of the algorithm were from the same platform. We challenged ourselves to learn how to use Valgrind and were successful in our attempt.

Moving forward, we want to implement Prim's algorithm with an adjacency matrix and observe the performance. We also hope to successfully run the parallelized versions of each algorithm on the EECIS GPUs to gain a better understanding of parallelized MST algorithms and compare our results to that of our benchmarks.

Appendix A

Presented here is the C++ code written for our **graph generation**:

```
Graph* generateRandGraphs(int NOV, int NOE){
    int max = 100;
    int i = 0;
    int edges[NOE][2];
    Edge *allEdges = new Edge[NOE];

    while(i < NOE){
        edges[i][0] = rand()%NOV;
        edges[i][1] = rand()%NOV;
        if(edges[i][0] == edges[i][1])
            continue;
        else{
            for(int j = 0; j < i; j++){
                if((edges[i][0] == edges[j][0] && edges[i][1] == edges[j][1]) ||
                    (edges[i][0] == edges[j][1] && edges[i][1] == edges[j][0]))
                    i--;
            }
        }
        i++;
    }
    for(int i = 0; i < NOE; i++){
        Edge e = createEdge(edges[i][0], edges[i][1], rand()(max));
        allEdges[i] = e;
        cout << i << " " << allEdges[i].src << " " << allEdges[i].dest << " " << allEdges[i].weight << endl;
    }
    Graph* g = createGraph(NOV, NOE, allEdges);
    return (g);
}
```

Presented here is the C++ code written for **serial Boruvka's Algorithm**:

```
// The main function for MST using Boruvka's algorithm
void boruvkaMST(Graph* graph)
{
    // Get data of given graph
    int V = graph->V, E = graph->E;
    Edge *edge = graph->edge;

    // Allocate memory for creating V subsets.
    subset *subsets = new subset[V];

    // An array to store index of the cheapest edge of
    // subset. The stored index for indexing array 'edge[]'
    int *cheapest = new int[V];

    // Create V subsets with single elements
    for (int v = 0; v < V; ++v)
    {
        subsets[v].parent = v;
        subsets[v].rank = 0;
    }

    // Initially there are V different trees.
    // Finally there will be one tree that will be MST
    int numTrees = V;
    int MSTweight = 0;

    // Keep combining components (or sets) until all
    // components are not combined into single MST.
    //while (numTrees > 1)
    for (int x = 0; x < E; x++)
    {
        for (int v = 0; v < V; ++v) {
            cheapest[v] = -1;
        }
    }
```

```

// Traverse through all edges and update
// cheapest of every component
for (int i=0; i<E; i++)
{
    // Find components (or sets) of two corners
    // of current edge
    int set1 = find(subsets, edge[i].src);
    int set2 = find(subsets, edge[i].dest);

    // If two corners of current edge belong to
    // same set, ignore current edge
    if (set1 == set2)
        continue;

    // Else check if current edge is closer to previous
    // cheapest edges of set1 and set2
    else
    {
        if (cheapest[set1] == -1 ||
            edge[cheapest[set1]].weight > edge[i].weight)
            cheapest[set1] = i;

        if (cheapest[set2] == -1 ||
            edge[cheapest[set2]].weight > edge[i].weight)
            cheapest[set2] = i;
    }
}

// Consider the above picked cheapest edges and add them
// to MST
for (int i=0; i<V; i++)
{
    // Check if cheapest for current set exists
    if (cheapest[i] != -1)
    {
        int set1 = find(subsets, edge[cheapest[i]].src);
        int set2 = find(subsets, edge[cheapest[i]].dest);

        if (set1 == set2)
            continue;
        MSTweight += edge[cheapest[i]].weight;
        printf("Edge %d-%d included in MST, weight %d\n",
            edge[cheapest[i]].src, edge[cheapest[i]].dest, edge[cheapest[i]].weight);

        // Do a union of set1 and set2 and decrease number
        // of trees
        Union(subsets, set1, set2);
        numTrees--;
    }
}
} if (numTrees!=1){
cout << "DISJOINT" << endl;
cout << "NO MST" << endl;
return;
}

printf("Weight of MST is %d\n", MSTweight);
return;
}

```

Presented here is the C++ code written for **serial Kruskal's Algorithm**:

```
// The main function to construct MST using Kruskal's algorithm
void KruskalMST(Graph* graph)
{
    int V = graph->V, E = graph->E;
    Edge* result = graph->edge; // This will store the resultant MST
    int e = 0; // An index variable, used for result[]
    int i = 0; // An index variable, used for sorted edges

    // Step 1: Sort all the edges in non-decreasing
    // order of their weight. If we are not allowed to
    // change the given graph, we can create a copy of
    // array of edges

    //print statements to check sorting
    /*
    cout << "BEFORE QSORT" << endl;
    for (int i = 0; i < E; i++){
        cout << result[i].src << " " << result[i].dest << " " << result[i].weight << endl;
    }
    */
    qsort(result, E, sizeof(result[0]), myComp);
    result = Kruskal_sort(result, E);
    /*
    cout << "AFTER QSORT" << endl;
    for (int i = 0; i < E; i++){
        cout << result[i].src << " " << result[i].dest << " " << result[i].weight << endl;
    }
    */

    // Allocate memory for creating V subsets
    subset *subsets =
        (subset*) malloc( V * sizeof(subset) );

    // Create V subsets with single elements
    for (int v = 0; v < V; ++v)
    {
        subsets[v].parent = v;
        subsets[v].rank = 0;
    }

    int numTrees = V;
    // Number of edges to be taken is equal to V-1
    //while (e < E)
    for (i = 0; i < E; i)
    {
        // Step 2: Pick the smallest edge. And increment
        // the index for next iteration
        Edge next_edge = graph->edge[i++];

        int x = find(subsets, next_edge.src);
        int y = find(subsets, next_edge.dest);

        // If including this edge doesn't cause cycle,
        // include it in result and increment the index
        // of result for next edge
        if (x != y)
        {
            result[e++] = next_edge;
            Union(subsets, x, y);
            numTrees --;
        }
        // Else discard the next_edge
    }
    if (numTrees > 1) {
        cout << "DISJOINT" << endl;
        cout << "NO MST" << endl;
        return;
    }
}
```

WORKS CITED

- Bazlamaçci, Cüneyt F., and Khalil S. Hindi. "Minimum-Weight Spanning Tree Algorithms A Survey and Empirical Study." *Computers & Operations Research*, vol. 28, no. 8, 2001, pp. 767–785., doi:10.1016/s0305-0548(00)00007-1.
- Chu, Ding. "BSP Implementation of Boruvka's Minimum Spanning Tree Algorithm." May 2015, etd.ohiolink.edu/!etd.send_file?accession=kent1424483867&disposition=inline.
- Lončar, Vladimir, et al. "Parallelization of Minimum Spanning Tree Algorithms Using Distributed Memory Architectures." www.scl.rs/papers/Loncar-TET-Springer.pdf.
- Mariano, Artur, et al. "Hardware and Software Implementations of Prim's Algorithm for Efficient Minimum Spanning Tree Computation." *IFIP Advances in Information and Communication Technology Embedded Systems: Design, Analysis and Verification*, 2013, pp. 151–158., doi:10.1007/978-3-642-38853-8_14.
- Morgado, Aleksander. "Understanding Valgrind Memory Leak Reports." <https://Aleksander.es/Data/Valgrind-Memcheck.pdf>, 4 Feb. 2010, www.bing.com/cr?IG=33CD3940A22A4E508F8A7EED78B7E2BC&CID=1A0C41B62AA36A5327BB4AEF2B0C6BE3&rd=1&h=5SaJndKnciYJbAcuRqqCFbyOM74rH4SKhpmFtppdUJ8&v=1&r=https%3a%2f%2faleksander.es%2fdata%2fvalgrind-memcheck.pdf&p=DevEx,5067.1.
- "NVIDIA® Tesla™ K40C GPU Computing Accelerator - 12GB GDDR5 - Active Cooler." *Thinkmate High Performance Computing and Storage Systems*, www.thinkmate.com/product/nvidia/900-22081-2250-000.

Sousa, Cristiano Da Silva, et al. "A Generic and Highly Efficient Parallel Variant of Boruvka's Algorithm." *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, 2015, doi:10.1109/pdp.2015.72.

Wang, Wei, et al. "Design and Implementation of GPU-Based Prim's Algorithm ." *I.J.Modern Education and Computer Science*, Modern Education and Computer Science, July 2011, www.mecs-press.org/ijmecs/ijmecs-v3-n4/IJMECS-V3-N4-8.pdf.