MSBD5003 Project Final Report
Implementation of MST algorithm using PySpark

GU Chenghao 20527420
GAO Han 20565848
CHEN Weili 20528175
WANG Ruolan 20551328

Abstract

Minimum spanning tree is commonly used in multiple areas. There are many algorithms which can generate it from original graph. In this report, we will introduce the implementation details of two related algorithms, Brouvka's algorithm and Prim's algorithm, which are parallel and sequential respectively. Then we prepare different sizes of graph datasets to compare their performance in one single machine and clusters. The results show that Brouvka's algorithm using Pyspark API outperforms Prim's algorithm both in saving memory and time.

1 INTRODUCTION

Here is an actual problem. Assuming that there are air routes data among all Chinese cities and corresponding price, what is the cheapest combination of flights from Beijing to Hong Kong? Transfer flight is allowed.

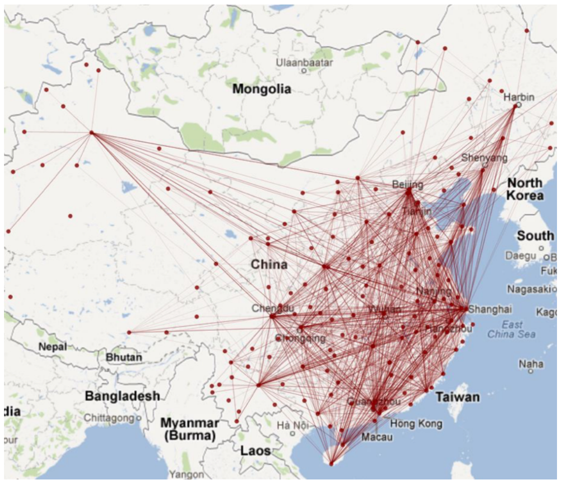


Figure 1: An example of air routes in China

Figure 1 shows that there are a lot of nodes representing cities, while red lines mean direct flight between two cities. Price can be viewed as weight of edges. One possible solution is the combination of MST and shortest path algorithm.

A minimum spanning tree(MST) is a subset of the edges of a connected, edge-weighted undirected graph that connects all the vertices together, without any cycles and with the minimum possible total edge weight.

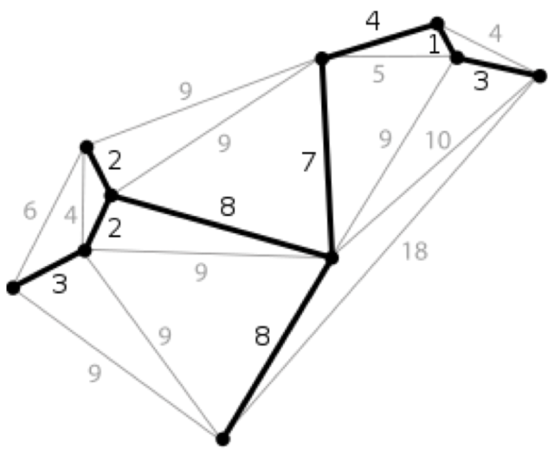


Figure 2: An example of minimum spanning tree generated from original graph

Table 1: Similarities and differences between MST and its graph

Properties	MST	Original graph
Directionality	Undirected	Undirected
Connectivity	Connected	Connected
Weight	minimum total weighted edges	All weighted edges
Circulation	No circles	Possibly having circles

There are several algorithms to generate MST. Some are sequential. Some can be implemented concurrently. So we choose two most significant algorithms called Boruvka's algorithm and Prim's algorithm, which are parallel and sequential algorithm respectively to implement. Therefore, the main tasks of our project are:

- Using Pyspark API to implement Boruvka's algorithm, which is a parallel algorithm of generating Minimum Spanning Tree.
- Using Pyspark API to implement Prim's algorithm, which is a classical sequential algorithm to generate MST.
- Utilizing different scales of weighted graph datasets to compare the performance of two algorithms above running in one machine and cluster.

The pipeline of our report is in Section 2, it introduces explanation, analysis and pyspark implementation of Brouvka's algorithm. Then in Section 3, the report will introduce corresponding explanation, analysis and pyspark implementation of Prim's algorithm. In Section 4, the report shows numerical experiment configuration, results and explanation. At last in Section 5, we conclude our work and provide possible expansions.

2 BORUVKA's ALGORITHM

Boruvka's algorithm, also called Sollin's algorithm, is a greedy algorithm for finding a minimum spanning tree in a graph for which all edge weights are distinct. It can run in parallel, which improves the efficiency of finding the lightest connected way, especially for big graph with large quantity of nodes and edges.[1]

2.1 Algorithm Analysis

According to paper and Wikipedia[1], we have compiled the pseudo code.Our target is to accomplish the algorithm in python spark and decrease the overhead both on single computer and cloud cluster. The main step of Boruvka's algorithm contains two parts:

Algorithm 1 Boruvka's Algorithm

```
Input: A graph G whose edges have distinct weights
Output: Minimum spanning tree F, Algorithm running time.
Initialize a tree F to be a set of one-vertex trees, one for each vertex of the graph.
while F has more than one component: do
    Find the connected components of F and label each vertex of G by its component.
    Initialize the cheapest edge for each component to None.
    for each edge uv of G: do
        if u and v have different component labels: then
            if uv is cheaper than the cheapest edge for the component of u: then
                Set uv as the cheapest edge for the component of u.
            else
                Set uv as the cheapest edge for the component of v.
            end
        end
    end
    foreach component whose cheapest edge is not None do
        Add its cheapest edge to F.
    end
end
```

1.Finding the minimum weight edge: For the first iteration, we find the minimum edge of each vertex in parallel. After the first round, the whole number of vertices decrease to a half. It is possible that some edges may be chosen twice.

2.Vertices labelling: It is an essential step to recognize whether the vertex in the same component. When all the vertices are in the same label, MST can be found. Labelling is also regarded as graph contacting, in other words, vertices with the same label shrink to the same point and are connected to components of different labels. There are two different techniques to approaching this algorithm when contracting the graph: edge contraction and star contraction.

- **edge contraction:** it involves finding the min edge incident to each vertex in the first iteration and each component thereafter. Then, we add these edges to our MST and perform the necessary contractions in the contraction stage. By edge contraction, we acquire some components so that we are able to label remain vertices easily.

- **star contraction:** it involves flipping a coin for each component and finding the heads and tails. The basic idea of star contraction is to identify and contract non-overlapping stars, which includes two types of methods, small-star and large-star operations.[3] Both operations are given a node and a subset of its neighborhood, they replace for every the edge with the edge whose neighbor node has the minimum label. Small-star connects 1 to all nodes labels no larger than the given node. Large-star connects 1 to all nodes with label strictly larger than the given node.[4] More details can be found in the paper.

After labelling, vertices of all subgraphs will be recombined into new graphs for the next iteration. Several times later, the algorithm will cease when there is only one label left.

2.2 Key Data Structure

In order to implement this algorithm in PySpark, we apply some key data structures to cope with most tough problem. Following are our three data structures.

1.Graph DataFrame: We construct the undirected graph with an underlying representation of a directed graph (so a directed edge is listed in the graph twice for the two directions).

2.Union-find: It is a **disjoint-set data structure** that tracks a set of elements partitioned into a number of disjoint subsets. It provides near-constant-time operations (bounded by the anti-Ackermann function) to add new sets, to merge existing sets, and to determine whether elements are in the same set. The Avg time complexity is $O(\alpha(n))$ and worst case is $O(n)$. Therefore, we implement a simple union-find-set structure called **'QuickFind'**, in which the **find function** is relatively simple and the

union(p,q) function needs to consider that all the node ids connected to p should be set to the current id of q, so that the group where p is located and the group where q is combined are combined into one group. more implementation details can be find in **2.4.3**.

3.Connected Component: a **connected component** (or just **component**) of an undirected graph is a subgraph in which any two vertices are connected to each other by paths, and which is connected to no additional vertices in the supergraph.[5] In Boruvka's algorithm, given a connected, undirected graph G with N vertices and M weighted edges, a minimum spanning tree can be found. In the beginning of the procedure each vertex is considered as a separate component. In each step the algorithm chooses the cheapest outgoing edge for each component, and connects (merges) the newly derived component. Actually, we try to apply the **ConnectedComponents API** provided by GraphFrame, but this API function have quite worse performances in both small data set and large data set. So, after searching some papers and source codes avaliable on the Internet, we implement a parallel connected component labeling algorithm called **'Two phase algorithm'** whose time complexity is $O(logn * logn)$, n is number of nodes. more details of this algorithm can be found in the paper of Kiveris and etc.el[2].

2.3 Ideas in Coding

2.3.1 Initial Ideas

It is common to use an adjacency matrix for high vertex counts in a undirected graph, and rows and columns represent the source and destination respectively, the inner numbers mean the weight of each edge. Additionally, we need to join the vertices with new label after each iteration, which cause unnecessary computation. Our original idea is to delete the vertices which have been labeled in former iteration.

2.3.2 Final Ideas

Some of our initial ideas were not deliberate. As we started working with this idea and implemented a sequential version of Boruvka's with it, we realized that the linked list would not parallelise well and it would also require a lot of overhead in ensuring correctness if we continued with it for the parallel version.

So, we employed the undirected graph with an underlying representation of a directed graph so that we can use GraphFrame to coding. Vertices were numbered from 0 to n-1 where n is the number of vertices. Edges included three parts, one for source vertices, one for destination vertices, and one for weights. Every edge can be distinguished because of its unique weight. Considering that the deletion will affect the connectivity of the entire graph, we decided to use edge contract to simplify the remaining edges of graph, and iterate through the results step by step.

2.4 Implement in pySpark

There is no relative source code about Boruvka's algorithm using pyspark, scala or java in github and websites. So, we refer to the paper as well as follow pseudo codes to program. In order to make full use of spark's advantages, we use parallelized method to implement boruvka's algorithm and each executor perform parallel run on the program, in which case we can improve the performance by adding executor. As for optimization, we also try to prevent shuffle and repartition of data, like the usage of collect action. In addition, we try to apply cache and some other function API to improve performance.

2.4.1 Read and Format Graph Data

The goal in this stage is to read CSV file by default hash partitioner , and then split is used to separate each line of data into three parts, respectively 'src', 'dst' and 'weight'. What's more, After this, we generate two RDDs: begin_edges and begin_vexs which are applied to construct the initial GraphFrame. In the initial graph, each vertex has different label.

```python
graph_data = sc.textFile(path_to_file, 8) # read CSV file

begin_edges = graph_data.map(lambda line: line.split()).map(lambda edge:(edge[0],int(edge[1]),
int(edge[2]))) # generate edges RDD
# generate vertices RDD
begin_vexs = edge_dataframe.map(lambda line: (line[0], line[0])).distinct()
```

Figure 3: Read and Format Graph Data

2.4.2 Find minimum weighted edges

In this step, we would like to find minimum edge of each vertex. First, we should maintain those edges in which the source node and destination node do not have the same label. Then, we aggregate edges and find each vertex's minimum weighted edge in parallel according to labels of vertices by using groupBy function. Next, we have to join with min_edges and inter_graph and gain a final_edges dataframe which have three columns: 'src', 'dst' and 'weight'.

```python
# Filter edges whose source node and destination node do not have same label
filter_df = g.find("(a)-[e]->(b)").filter("a.label != b.label").select("e.*")
inter_graph = GraphFrame(g.vertices, filter_df)
# groupby and parallelly find each vertice`s minimum weighted edge
min_edges = inter_graph.triplets.groupBy('src.label').agg({'edge.weight':
'min'}).withColumnRenamed('min(edge.weight AS `weight`)', 'min_weight')
final_edges = min_edges.join(inter_graph.triplets, (min_edges.label ==
inter_graph.triplets.src.label)&(min_edges.min_weight == inter_graph.triplets.edge.weight)) \
                           .select(col('src.id').alias('src'), col('dst.id').alias('dst'),
col('min_weight').alias('weight'))
```

Figure 4: Find minimum weighted edges

2.4.3 Vertices Labeling

In terms of labeling, we implement two approaches: connected component and union-find set. After labeling, we would gain a dataframe containing two columns(id and label), and define a new graph. Then, the new graph which contain new label data would be applied in next iteration.

- **Connected component:** Due to that Kiveris and etc[2] propose a connected component algorithm named 'Two-phase algorithm' to deal with labeling problem, we implement this algorithm in pyspark. In particular, we performing the large-star operation on all nodes in parallel preserves the connectivity of the graph, so that after one round of large-star the number of edges in the resulting graph is no larger than in the original graph. Then, the small-star operation preserves connectivity and does not increase the number of edges, repeat until convergence **(the time complexity of this algorithm is $O(log^2 n)$)**. We input a RDD containing edge pairs, and output a RDD in which each vertex own a label.more details can be found in paper.

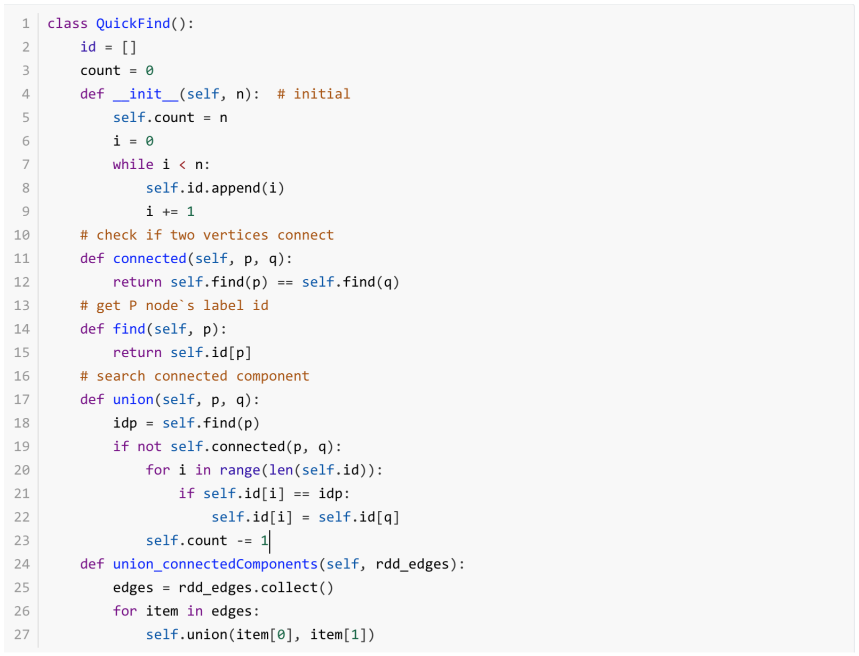
```python
def Connnected_Components(rdd):
    # Small-star
    while True:
        # Large-star
        while True:
            prev_rdd = rdd
            rdd = rdd.flatMap(largeStarMap).distinct().groupByKey().flatMap(largeStarReduce)
            if check_convergence(prev_rdd, rdd): break # Check if convergence
        prev_rdd = rdd
        rdd = rdd.map(smallStarMap).distinct().groupByKey().flatMap(smallStarReduce)
        if check_convergence(prev_rdd, rdd): break
    vals_rdd = rdd.values().distinct()
    rdd = rdd.union(vals_rdd.map(lambda k: (k, k)))
    # generate final (vertice, label) pairs
    final_rdd = rdd.map(outputEdges).map(read_data)
    return final_rdd
```

Figure 5: Connected_component

Algorithm 2 The two-phase Algorithm

```
Input: Edges (u,v) as a set of key-value pairs <u;v>. A unique label $l_v$ for every node v belongs to V
Output: A new label $l_v$ for every node v.
repeat
    repeat
        large-star
    until Convergence;
    small-star
until Convergence;
```

- **Union-find set:** According to materials and source code on the Internet, we implement a union-find set class, this data structure can find the connected components in nearly linear time(O((n))). In this case, **union_connectedComponents** function is to union all edges in rdd_edges, and we can gain all vertices‘ labels in id list.



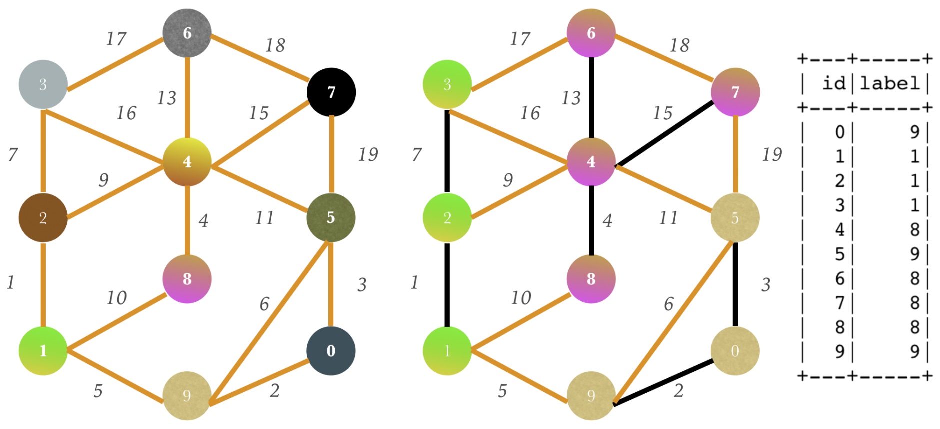
```
class QuickFind():
    id = []
    count = 0
    def __init__(self, n):  # initial
        self.count = n
        i = 0
        while i < n:
            self.id.append(i)
            i += 1
    # check if two vertices connect
    def connected(self, p, q):
        return self.find(p) == self.find(q)
    # get P node`s label id
    def find(self, p):
        return self.id[p]
    # search connected component
    def union(self, p, q):
        idp = self.find(p)
        if not self.connected(p, q):
            for i in range(len(self.id)):
                if self.id[i] == idp:
                    self.id[i] = self.id[q]
            self.count -= 1
    def union_connectedComponents(self, rdd_edges):
        edges = rdd_edges.collect()
        for item in edges:
            self.union(item[0], item[1])
```

Figure 6: QuickFind

2.5 Testing on different graph

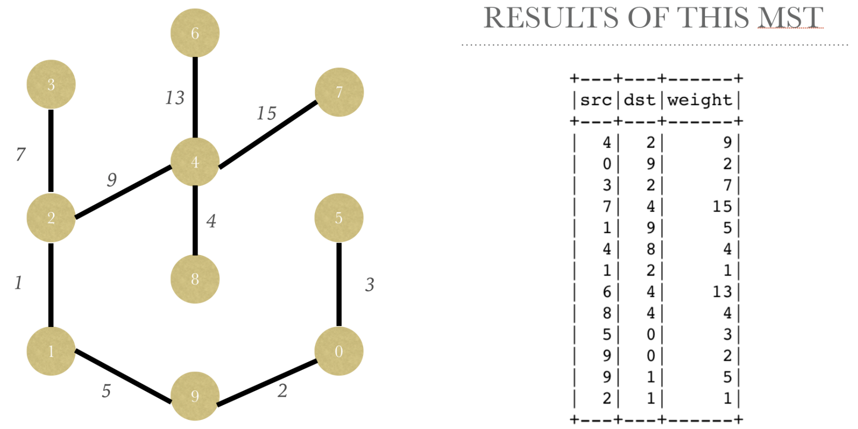
We started testing correctness and efficiency on a small graph with 10 nodes and 17 undirected edges (indeed 34 directed edges). In the first iteration, 10 vertices in 10 different component, and the minimum weight edge out of every component is added. After first round, three components remain. Then, in the second iteration, we use the same way to choose minimum edge. However, the edge (9,5) is shortest in this iteration but it cannot be considered because both endpoints are in the same component. Finally, all the vertices in this dataset are in the same component. And we compute the correct MST, the results are as following.(Edges (u,v) and (v,u) are regarded as the same one in the final result because of undirected graph.) We use different codes to test and the results are all right.

Then we use networkx package to generate some graphs which help us test efficiency of our codes. Specific time consuming will be demonstrated in the evaluation section.



```
+---+-----+
| id|label|
+---+-----+
|  0|    9|
|  1|    1|
|  2|    1|
|  3|    1|
|  4|    8|
|  5|    9|
|  6|    8|
|  7|    8|
|  8|    8|
|  9|    9|
+---+-----+
```

Figure 7: Original graph and first round graph



RESULTS OF THIS MST

```
+---+---+------+
|src|dst|weight|
+---+---+------+
|  4|  2|     9|
|  0|  9|     2|
|  3|  2|     7|
|  7|  4|    15|
|  1|  9|     5|
|  4|  8|     4|
|  1|  2|     1|
|  6|  4|    13|
|  8|  4|     4|
|  5|  0|     3|
|  9|  0|     2|
|  9|  1|     5|
|  2|  1|     1|
+---+---+------+
```

Figure 8: Result of Boruvka's algorithm

3 Prim's Algorithm

3.1 Explanation Analysis

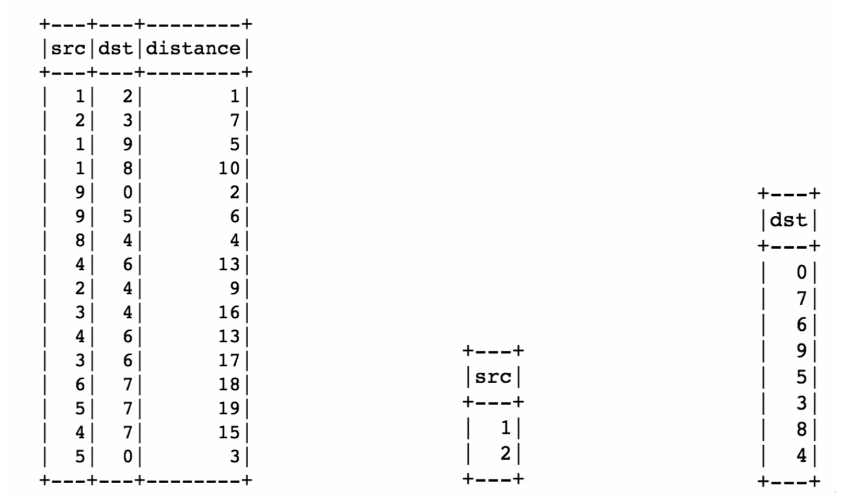
Using dataframe in the pyspark api to store the data structure of the initial graph, the dataframe will be the input for the prim algorithm. Then, we randomly select one initial vertex and put it into source dataframe, and the left vertexes will be out into destination dataframe. By joining source, destination and initial graph dataframe, we filter the edges which one vertex is from the source and the other is from the dest. Then we change the dataframe to rdd and use reduce method to find the minimum distance. The output of MST is also represented by dataframe.[6][7]

Algorithm 3 Prim's Algorithm

```
Input: A graph G whose edges have distinct weights
Output: Minimum spanning tree, Algorithm running time.
repeat
    1. Initialize the minimum spanning tree (MST) with a vertex chosen randomly.
    2. Find the minimum edge that one vertex is from the MST and the other is from the left vertexes.
    3. Add the new vertex into MST and subtract the vertex from the left vertexes.
until all vertices are added into MST;
```

3.2 Key Data Structure

When implementing prime algorithm, we use dataframe to represent the initial graph. The newly added edge to the MST will be printed step by step. The source and destination dataframe are both used for searching the minimum distance from the vertexes in MST and the left vertexes. Source includes vertexes which have been added into MST and destination includes left vertexes. The structure of the three dataframes are as follows.The Prime algorithm has a sequential pattern and we should find minimum edge step by step.



```
+---+---+--------+
|src|dst|distance|
+---+---+--------+
|  1|  2|       1|
|  2|  3|       7|
|  1|  9|       5|
|  1|  8|      10|
|  9|  0|       2|
|  9|  5|       6|
|  8|  4|       4|
|  4|  6|      13|
|  2|  4|       9|
|  3|  4|      16|
|  4|  6|      13|
|  3|  6|      17|
|  6|  7|      18|
|  5|  7|      19|
|  4|  7|      15|
|  5|  0|       3|
+---+---+--------+
```

```
+---+
|src|
+---+
|  1|
|  2|
+---+
```

```
+---+
|dst|
+---+
|  0|
|  7|
|  6|
|  9|
|  5|
|  3|
|  8|
|  4|
+---+
```

Figure 9: structure of the three dataframes

3.3 Implementation

There are mainly four steps to implement the algorithm, and I use while loop to update states:

Step1: Initialize the source, destination dataframe.

```
# Randomly select vertex 1 and put it into source dataframe.
r=g.edges.filter("src = 1").sort(g.edges.distance).take(1)
source=sc.parallelize(((r[0][0],),(r[0][1],))).toDF().withColumnRenamed('_1', 'src')
temp1=g.edges.select('dst')
temp2=g.edges.select('src')
# Other vertexes will be put into dest dataframe.
dest=temp1.union(temp2).distinct()
dest=dest.sbtract(source)
```

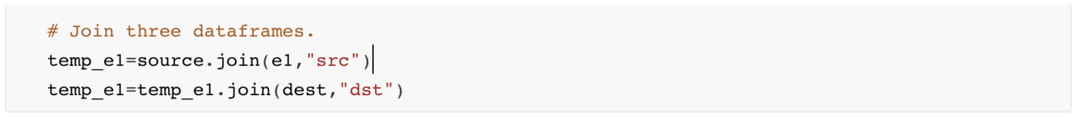
Figure 10: Codes of step1

Step2: Filter the edges which one vertex is from MST and the other is from left vertexes.

```
# Found the minmimum weight.
r=temp_e1.rdd.reduce(lambda a,b: a if (a[2]<b[2]) else b)
```

Figure 11: Codes of step2

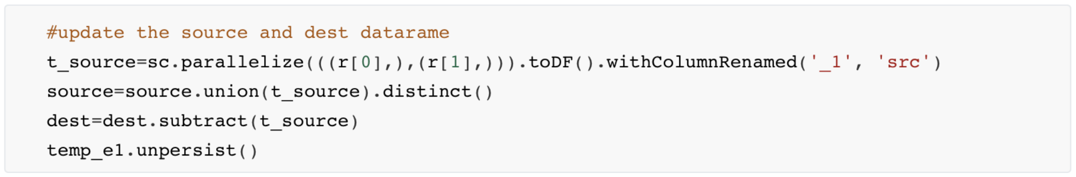
Step3: Convert dataframe to rdd so as to obtain the edge which has minimum distance.



```
# Join three dataframes.
temp_e1=source.join(e1,"src")
temp_e1=temp_e1.join(dest,"dst")
```

Figure 12: Codes of step3

Step4: Update the source and dest DataFrame.



```
#update the source and dest datarame
t_source=sc.parallelize(((r[0],),(r[1],))).toDF().withColumnRenamed('_1', 'src')
source=source.union(t_source).distinct()
dest=dest.subtract(t_source)
temp_e1.unpersist()
```

Figure 13: Codes of step4

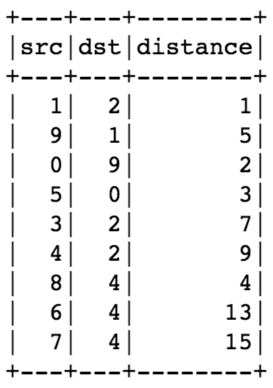
3.4 Challenges of Prim's Algorithm

Challenge1: Join the Dataframes efficiently. When implementing the algorithm, we should join three Dataframes of initial graphframe, destination and source Dataframe in every sequential step. Thus, joining these three dataframes correctly can help to improve the efficiency of the algorithm. In our implementation, source and destination Dataframe is much smaller than the initial graphframe. Thus, we use small dataframe to join large dataframe to improve the efficiency.

Challenge2: Searching the minimum distance.In every step, the algorithm should search for the minimum distance between vertexes from MST and the left vertexes. When implementing the algorithm, I convert the dataframe to rdd and use reduce method to find the minimum distance. Compared with directly operating on the dataframe, this method saves a lot of time.

3.5 Testing on different graph

The initial graph is as same as figure 7(left). The minimum spanning tree can be connected by the following edges(figure 14).



```
+---+---+--------+
|src|dst|distance|
+---+---+--------+
|  1|  2|       1|
|  9|  1|       5|
|  0|  9|       2|
|  5|  0|       3|
|  3|  2|       7|
|  4|  2|       9|
|  8|  4|       4|
|  6|  4|      13|
|  7|  4|      15|
+---+---+--------+
```

Figure 14: Result of Prim's algorithm

4 Evaluation

Based on the implementation of two algorithms above, the report utilizes different scales of datasets to compare their performance in single machine and cluster.

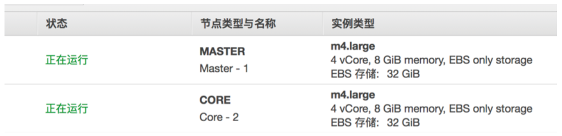
4.1 Experimental Environment

The configuration of our single machine and cluster is shown in Table 2. The configuration of cluster

Table 2: Machine configuration

Single machine	Cluster
Processor: 2.3 GHz Intel Core i5 Memory: 8GB 2133 MHz	Structure: HDFS file system(1 master and 2 slaves) Processor: 4 vCore Memory: 8 GiB memory

constructed on Amazon AWS is shown in Figure 15. Then based on requirements of comparison and evaluation, we generate different scale of datasets. They comes from different sources. All information of these datasets is shown in Table 3. In Table 3, 2elts are generated by ourselves



	状态	节点类型与名称	实例类型
	正在运行	**MASTER** Master - 1	**m4.large** 4 vCore, 8 GiB memory, EBS only storage EBS 存储: 32 GiB
	正在运行	**CORE** Core - 2	**m4.large** 4 vCore, 8 GiB memory, EBS only storage EBS 存储: 32 GiB

Figure 15: A screen shot from Amazon AWS Virture Machine

Table 3: Prepared datasets

Name	Nodes	Edges	Generation method
2elts	10	17×2	Manually
edges_generate_4	3900	56550×2	connected caveman graph
edges_generate_3	6000	117000×2	
edges_generate_2	10000	245000×2	
4elts	15606	45878×2	web downloaded
edges_generate_5	83	2208×2	extracted from connected caveman graph
edges_generate_6	144	1999×2	

manually, which has been shown just now. 4elts is an example downloaded from web. All others are generated according to the mechanism of connected caveman graph. For testing large scale graph dataset, we use python package networkx to generate connected caveman graph. It has following characteristics:

- Complete connected graph
- Comprised with cliques. Each clique has a number of nodes.
- More edges are in internal clique, while between cliques there is only one edge.

The structure is just like social groups. It has more communication within a group but less among different groups. Then we assigned distinct weight on edges randomly. Figure 16 is an example of Connected Caveman graph. It has 5 cliques.

4.2 Comparison

Based on implementations of two algorithms with multiple approaches above, the report makes 3 main comparisons with different sizes of datasets:

- Compare the performance of 2 vertices labeling approaches in Boruvka's algorithm.

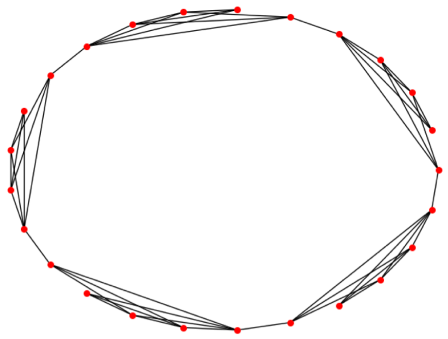


Figure 16: An example of Connected Caveman graph

- Compare the performance of Boruvka's algorithm with connected components in single machine and cluster.
- Compare the performance of Boruvka's algorithm and Prim's algorithm in cluster.

First of all, in Section 2.4.3, the report introduces 2 ways to do vertices labeling, which are unionfind and connectedcomponent. Here we test these 2 approaches in single machine along with different scale of datasets. Figure 17 shows the result.

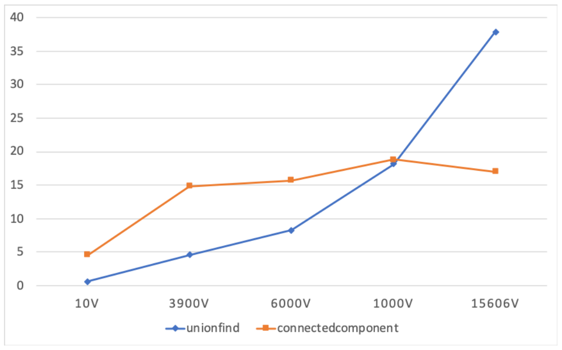


Figure 17: Testing in single machine with 2 ways to do vertices labeling

According to Figure 15, the datasets in Figure 17, from left to right, gradually have more vertices. It means larger scale of network. Figure 17 shows that along with the increase of network scale, unionfind outperforms connectedcomponent at the beginning but fails later. In Section 2.2, the report shows that unionfind has near-constant-time operations to add new edges, which is a pretty good performance when network scale is not too large though it is sequential. At the same time, although connectedcomponent is parallel, it costs at the part of communication between different partitions. So, when network scale is small, it runs slower than unionfind. Then when network scale goes up, obviously labeling vertices in parallel can save a lot of time. Secondly, we utilize the same datasets to test the performance of Boruvka's algorithm both in single machine and cluster. The labeling vertices way is connectedcomponent. The result is shown in Figure 18.

Figure 18 shows that when network scale is not large, the performance of single machine and cluster is similar but cluster is slightly faster. When network scale becomes larger, the gap has a trend to be wider. Obviously cluster can better take advantage of the parallelism of Boruvka's algorithm. And along with the growth of network scale, such a advantage will become more obvious. At last, the report compares the performance of Boruvka's algorithm and Prim's algorithm running in cluster. Table 4 shows the result.

Table 4 shows the large gap of efficiency between Boruvka's algorithm and Prim's algorithm. Except 2elts, all other generated graphs have similar structure, which is Connected Caveman graph. However, for Prim's algorithm, the running time of a graph only with 83 vertices has already been much larger than the running time of a graph with 3900 vertices by Boruvka's algorithm. Prim's algorithm is a

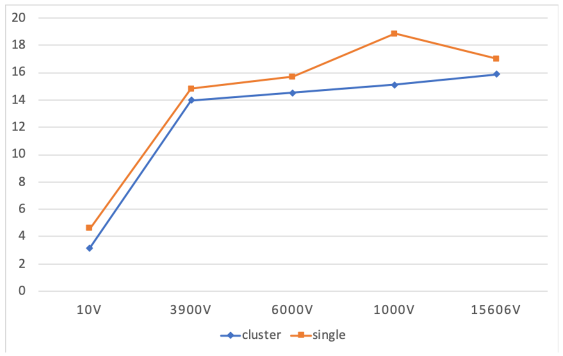


Figure 18: Time cost of Boruvka's algorithm with connectedcomponent in single machine and cluster

Table 4: Time cost of Boruvka's and Prim's algorithm with multiple datasets on cluster

Boruvka's		Prim's	
Dataset	Time	Dataset	Time
2elts(10V)	3.19s	2elts(10V,34E)	3.2s
edges_generate_4(3900V)	13.99s	edges_generate_5(83V)	49.51s
edges_generate_3(6000V)	14.53s	edges_generate_6(144V)	62.31s

sequential algorithm, so we cannot use parallel method to realize it. However, in every sequential step, we can parallelly execute it. In this algorithm, we should join three dataframes in every sequential step in order to find the minimum weight. This part becomes more difficult to execute when the three dataframes becomes larger. Thus, joining action actually consumes most time of the algorithm.[8] Meanwhile, Boruvka's algorithm is totally parallel. Also it cost of join dataframes will decrease sharply along with every iteration. That's why they have such a huge difference in efficiency.

4.3 Theoretical Analysis of the Result

When implementing the algorithm, we use adjacency table to represent the initial graph. The time complexity of the algorithm is as follows: $O((|V| + |E|)log|V|) = O(|E|log|V|)$ where E is the number of edges, and V is the number of vertices in G. As is shown in above, Prim's algorithm is more sensitive to the number of vertexes. If the number of vertexes is large, prim algorithm may not be functional.

Traditionally, Boruvka's algorithm can be shown to take $O(log|V|)$ iterations of the outer loop until it terminates, and therefore to run in time $O(|E|log|V|)$. According to the paper, for a graph G = (V, E), each step of star Contract using $O(|E| + |V|)$ work and $O(log|V|)$ span. Therefore, since there are $O(log|V|)$ rounds, the overall span of the algorithm is $O(log^2|V|)$. However, the Avg time complexity of QuickFind is $O(\alpha(n))$ and worst case is $O(n)$, where α means anti-Ackermann function.

5 Potential Improvements

In this project, we used pyspark to implement two different methods for finding the minimum spanning tree. Using the knowledge of the what we learned about GraphFrame and rdd, we started from the edge or vertex to find the minimum weight edge and get the correct result. And through the experiments on the cluster, we verified the reliability, scalability and efficiency of our parallel algorithms. For Boruvka's algorithm, we tried the two methods of labeling using UnionFind set and connectedComponent respectively, as well as analyzed the difference between the two approaches in terms of idea and time complexity.

For Boruvka's Algorithm, in order to improve the efciency of the algorithm, we can optimize the partition scheme of the graph strcuture to improve the parallel efciency. For instance, the zigzag Find operation involved in this paper[3] is a relatively efcient segmentation scheme. By appling ghost vertices mentioned by this paper, the Union-Find set can be implemented in parallel, which improves the scalability and efciency of the algorithm.

For Prim algorithm, joining three dataframes takes a lot of time to run. Thus, for improvement, if there is a good method for filtering edges which connect outside vertexes and MST edges, the efficiency of the algorithm will be improved. By referencing wikipedia[8], if we use complex Fibonacci heap, the time cost will become half of the current time cost which can highly improve the eifficiency of the algorithm.

References

[1] https://en.wikipedia.org/wiki/Bor%C5%AFvka%27s_algorithm

[2] Raimondas Kiveris, Silvio Lattanzi, Vahab Mirrokni Vibhor, Rastogi Sergei, Vassilvitskii. Connected Components in MapReduce and Beyond(2014). SoCC '14, 3-5 Nov. 2014, Seattle, Washington, USA. Copyright is held by the owner/author(s). ACM 978-1-4503-3252-1. http://dx.doi.org/10.1145/2670979.2670997

[3] Fredrik Manne, Md. Mostofa Ali Patwary. A Scalable Parallel Union-Find Algorithm for Distributed Memory Computers(2009). Department of Informatics, University of Bergen, N-5020 Bergen, Norway. R. Wyrzykowski et al. (Eds.): PPAM 2009, Part I, LNCS 6067, pp. 186–195, 2010.

[4] Guy Blelloch. Graph Contraction, Min Spanning Tree(2011). Parallel and Sequential Data Structures and Algorithms — Lecture 15.

[5] D.S. Hirschberg, A.K. Chandra, D.V. Sarwate. Computing Connected Components on Parallel Computers. Communications of the ACM.

[6] https://blog.csdn.net/liuxingwan/article/details/52758595

[7] https://www.itread01.com/articles/1487678227.html

[8] https://zh.wikipedia.org/wiki/%E6%99%AE%E6%9E%97%E5%A7%86%E7%AE%97%E6%B3%95