IMPLEMENTATION OF DISTRIBUTED MINIMUM SPANNING TREE ALGORITHM BASED ON PYSPARK

LAU Kwanyuen
Student ID: 20647191

YANG Rongfeng
Student ID: 20644943

TANG Huimin
Student ID: 20636635

YIN Dongxin
Student ID: 20658592

ABSTRACT

In this project, we implement three minimum spanning tree algorithms using Spark Python API. Compared with traditional algorithm, these distributed minimum spanning trees can handle with large graphs which do not fit in the memory of a single machine. To evaluate the performance of the distributed minimum spanning trees, we analysis the theoretical processing time, computation cost and computation time for these algorithms under certain assumption. Finally, we apply the 3 algorithms to real-word data to further analysis the algorithm performance and scalability.

1 MINIMUM SPANNING TREE ALGORITHM

$Minimum\ Spanning\ Tree$ (MST) algorithm is one of the most important and fundamental algorithms in graph problems. Given a connected, weighted and undirected graph, a spanning tree is the subset of the graph, that is a tree and connects all the vertices together (Patki, 2015). The minimum spanning tree is the spanning tree with the least sum of edge weights. Each minimum spanning tree includes $(V-1)$ edges, where V is the number of vertices in the original graph. MST finds diverse applications ranging from network design to image segmentation. Generally, there are two classical algorithms for finding a minimum spanning tree: $Kruskals\ algorithm$ and $Prims\ algorithm$.

1.1 KRUSKAL'S ALGORITHM

Kruskal's algorithm is a greedy method to build a minimum spanning tree by adding the edges one by one until all vertices are connected in one subgraph. Specifically, this algorithm begins with a set of sorted weighted edges. In each iteration, pick the edge with the smallest weight and check whether or not it forms a cycle with the former edges. If a cycle is not formed, take this edge as the partition of the minimum spanning tree. Else just discard this edge and continue. Repeat the last step until there are $(V-1)$ edges included in the constructed minimum spanning tree. One example for Kruskal's algorithm is shown in figure 1.

The time complexity for Kruskal's algorithm is $O(ElogE)$ or equally, $O(ElogV)$, where E is the number of edges and V is number of vertices. Sorting the edge weights takes $O(ElogE)$ time. A disjoint-set data structure, keeping track of a set of elements partitioned into a number of disjoint subsets, performs $O(logV)$ operations to detect cycle. So the overall time complexity is $O(ElogE+logV)$. The value of E can be at most V^2 and $log(V^2) = 2 \times logV$ is $O(logV)$. Therefore, the total time for Kruskal's algorithm is $O(ElogE)$ or $O(ElogV)$ (Algo).

1.2 PRIM'S ALGORITHM

Prim's algorithm is another greedy method to construct a minimum spanning tree. Compared with Kruskal's algorithm, Prim's algorithm adds vertices instead of edges to the growing spanning tree. The idea is to two disjoint set of vertices. One contains the vertices included in the minimum spanning tree, the other contains the vertices not yet included. Next, consider all edges connecting the two set and pick the edge with the least weight from these edges. Then remove the vertices

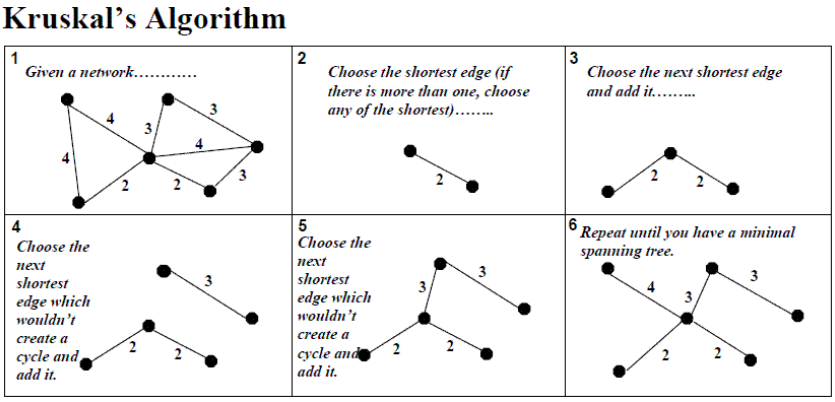

Figure 1: Example for Kruskal's Algorithm

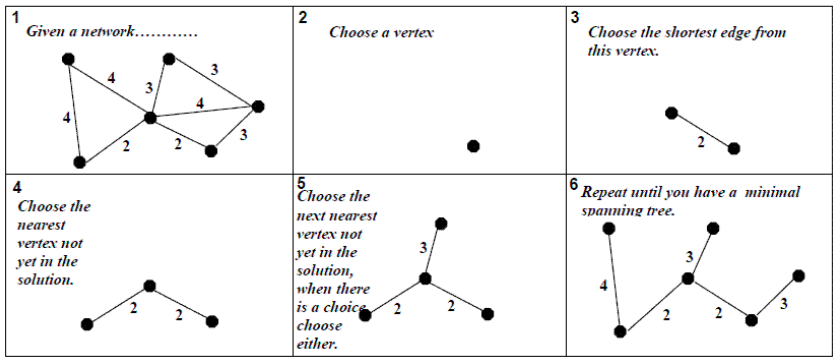

Figure 2: Example for Prim's Algorithm

connecting the chosen edge to the set containing MST. Repeat the above steps until all the vertices is included in the MST set. One example for Prim's algorithm is shown in figure 2.

Time complexity for Prim's algorithm is $O(v^2)$. While with the help of a binary heap and a priority queue, the time complexity can be reduced to $O(ElogV)$.

2 IMPLEMENTATION IN SPARK

How to handle with the large-scale problem is a very import issue. Many of the state-of-the-art approaches for such problems rely on numerical optimization, however, because of the scalability issues, usually one cannot directly apply classical optimization methods to solve large-scale problems (Hsieh et al., 2006). In this project, we apply $divide\ and\ conquer$ scheme to implement distributed minimum spanning tree algorithms in spark. In divide and conquer algorithm, problems should be easily decomposed into many independent subproblems and the solutions to the subproblems can be easily combined to give the solution to the original problems. The process of divide and conquer is shown in figure 3.

Particularly, in big data system, divide and conquer scheme begins with divide the problem into k partitions using $mapPartitions$ (one method in spark which maps function on each partition of RDD instead of each element of partition) where k is the number of executors in the system. Next, search for the minimum spanning tree on each partition. Additionally, we might apply $distinct$ (a spark function which returns a new RDD with the distinct elements of existing RDD) to remove duplicate solutions belonging to local MSTs. Until the local solutions from the subproblems are

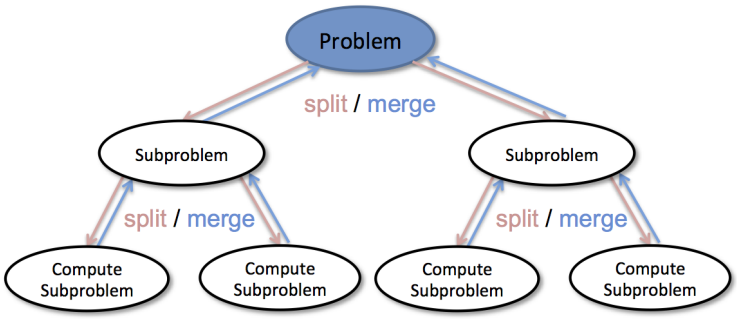

Figure 3: Divide and Conquer Scheme

small enough to fit on single machine, we use to initialize a global MST solver which converges quickly. Finally, we get global MST by computing MST algorithm on local MSTs and one machine.

3 IMPLEMENTATION

Next, we will look at three distributed algorithms. Among them, *edge partitioning* and *vertex partitioning* are based on Kruskal's algorithm. The remaining one, *parallel prim's* is based on Prim's algorithm. The basic assumption in all three algorithms is that the vertices of the graph fit in the memory of a single machine, but the edges don't.

Firstly, we should introduce the lemma above to help in establishing the correctness of the algorithms in a distributed setting.

Lemma 1 *Any edge that does not belong to MST of a subgraph containing the edge does not belong to the MST of the original graph.*

Based on this lemma, we are inspired to improve the speed of Kruskal's algorithm by first eliminating edges parallelly and performing Kruskal's algorithm on the remaining edges. Note that, implementations of disjoint set structure, Kruskal's algorithm and Prim's algorithm are required before the implementations of parallel algorithms. Although the implementations of those serial algorithms will not be discussed for the sake of simplicity, details could be found in the attachment.

3.1 EDGE PARTITIONING

3.1.1 ALGORITHM

Following the above scheme, we come up with a divide-and-conquer algorithm, which is to split edges randomly into disjoint sets and compute local MSTs parallelly first, and then to compute a global MST with edges belonging to local MSTs. Note that, all distributed machines share a set of vertices when computing local MSTs.

3.1.2 ANALYSIS

We use the following notation for the analysis:

Number of vertices: n

Number of edges: $m = n^{1+c}$

Memory of each machine: $\eta = n^{1+\epsilon}$

Number of machines required $l = \frac{m}{\eta} = n^{c-\epsilon}$

Lemma 2 *The algorithm terminates after* $\lceil \frac{c}{\epsilon} \rceil$ *iterations and returns the Minimum Spanning Tree.*

Proof By lemma 1 we know that any edge that is not part of the MST on a subgraph of G is also not part of the MST of G by the cycle property. Since the partition of edges is random, expected

Algorithm 1 Edge Partitioning Algorithm

1: **function** EDGEPARTITION($G(V, E)$)
2: η =Memory of each machine
3: $e = E$
4: **while** $|e| > \eta$ **do**
5: $l = \Theta(\frac{|E|}{\eta})$
6: Split e into $e_1, e_2, \ldots, e_l$ using a universal hash function
7: Compute T_i^* =KRUSKAL($G(V, e_i)$) ▷ In parallel
8: $e = \cup_i T_i^*$
9: **end while**
10: A =KRUSKAL($G(V, e_i)$)
11: **return** A
12: **end function**

number of edges on each machine is η. After one iteration, $|\cup_i T_i| \leq l \times (n-1) = n^{c-\epsilon} \cdot (n-1) = n^{1+c-\epsilon} + n^{c-\epsilon} = O\left(n^{1+c-\epsilon}\right)$. Thus the number of edges reduce by a factor of n^{ϵ}.

For the next iteration, if you choose l such that it is just enough to fit all the remaining edges on l machines, we can prove that the edges reduce by factor of n^{ϵ} again. Thus if we continue this process, after $\left\lceil \frac{c}{\epsilon} \right\rceil - 1$ iterations the input is small enough to fit onto a single machine, and the overall algorithm terminates after $\left\lceil \frac{c}{\epsilon} \right\rceil$ iterations.

One question that arises is the choice of the number of machines to use at each iteration. We will prove that it is in fact better to use only as many machines as required to fit all the edges in memory as opposed to using all the available machines.

Lemma 3 *The optimal number of machines that should be used decreases for each iteration such that $\frac{|e_i|}{l_i}$ is constant and is equal to $\frac{m}{l}$. Here, e_i are the number of edges left after $(i-1)^{th}$ iteration and l_i is the machines used at i^{th} iteration.*

Proof Let us assume that the algorithm terminates after t iterations. We know after t - 1 iterations, the edges can fit in memory of single machine. Therefore, the total processing time can be written as:

$$\frac{m}{l_1} logn + \frac{l_1 n}{l_2} logn + \frac{l_2 n}{l_3} logn + \ldots + \frac{l_{t-2} n}{l_{t-1}} logn + l_{t-1} \text{nlogn}$$

In algorithm 3 line 7, we distribute remaining edges to l_i machines after $i-1$ iterations. The l_i machines compute parallelly so the computation complexity in each machine is the same. Therefore, we only need to consider the processing time of a single machine.

The time complexity of KRUSKAL is $O(mlogn)$. m represents the number of edges, n represents the number of vertices. In the proof of Lemma 2, we get $|\cup_{i-1} T_{i-1}| \leq l_{i-1} \times (n-1) = O(n^{1+c-\epsilon}) = O(l_{i-1} \times n)$,$|\cup_{i-1} T_{i-1}|$ which is the number of remaining edges after $i-1$ iterations. Thus, at the i iteration, these edges need to be distributed to l_i machines and the processing time of each machine at this iteration is $\frac{|\cup_{i-1} T_{i-1}|}{l_i} logn = \frac{l_{i-1} \cdot n}{l_i} \text{logn}$.

The optimal l_i can be solved by doing partial derivative on $l_1, l_2, \ldots, l_{t-1}$ respectively to T and setting to zero.

$$T = \frac{m}{l_1} logn + \frac{l_1 n}{l_2} logn + \frac{l_2 n}{l_3} logn + \ldots + \frac{l_{t-2} n}{l_{t-1}} logn + l_{t-1} \text{nlogn}$$

Then

$$\frac{\partial T}{\partial l_1} = 0$$

$$\frac{\partial T}{\partial l_2} = 0$$

$$\vdots$$

$$\frac{\partial T}{\partial l_i} = 0$$

$$\vdots$$

$$\frac{\partial T}{\partial l_{t-2}} = 0$$

$$\frac{\partial T}{\partial l_{t-1}} = 0$$

Simplify them and we get

$$l_1^2 = l_2 \frac{m}{n}$$

$$l_2^2 = l_1 l_3$$

$$l_3^2 = l_2 l_4$$

$$\vdots$$

$$l_i^2 = l_{i-1} l_{i+1}$$

$$\vdots$$

$$l_{t-2} = l_{t-3} l_{t-1}$$

$$l_{t-1}^2 = l_{t-2}$$

According to Lemma 2, make the iteration $t = \frac{c}{\epsilon}$. The initial condition is $l_1 = \frac{m}{\eta} = n^{c-\epsilon}$. substitute l_1 into the above equtions,

$$l_1 = n^{c-\epsilon} = \left(\frac{m}{n}\right)^{\frac{t-1}{t}}$$

$$l_2 = n^{c-2\epsilon} = \left(\frac{m}{n}\right)^{\frac{t-2}{t}}$$

$$l_3 = n^{c-3\epsilon} = \left(\frac{m}{n}\right)^{\frac{t-3}{t}}$$

$$\vdots$$

$$l_i = n^{c-i\epsilon} = \left(\frac{m}{n}\right)^{\frac{t-i}{t}}$$

$$\vdots$$

$$l_{t-1} = n^{c-(t-1)\epsilon} = \left(\frac{m}{n}\right)^{\frac{1}{t}}$$

Therefore

$$\frac{|e_i|}{l_i} = \frac{n^{1+c-(i-1)\epsilon}}{l_i} = n^{1+\epsilon} = constant$$

3.1.3 PROCESSING TIMES AND COMMUNICATION COSTS

The processing time per iteration is sum of time taken to perform Kruskal's on one machine and time taken to partition the remaining edges for the next iteration. The total processing time can be written as:

$$\left\lceil \frac{c}{\epsilon} \right\rceil \times \left(O\left(\frac{m}{l}\text{logn}\right) + O\left(\frac{m}{l}\right)\right)$$

The communication costs involve one all-to-all communication which is due to the shuffle performed at the end of each iteration. As the number of edges decrease with subsequent iterations, the communication cost also decreases. The total communication cost becomes a geometric progression:

$$m + \frac{m}{n^{\epsilon}} + \frac{m}{n^{2\epsilon}} + \frac{m}{n^{3\epsilon}} + \ldots + \frac{m}{n^{(t-1)\epsilon}} = \frac{n(n^c - 1)}{1 - n^{-\epsilon}}$$

3.1.4 IMPLEMENTATION

The random splitting of edges is conducted before the parallelization with the built-in sort function and random function. Then, $mapPartitions$ is employed with Kruskal's to obtain an RDD containing edges belonging to local MSTs. We finally compute the global MST with edges belonging to local MSTs.

```
# parallelize
edges = sc.parallelize(shuffle(edges), num_partition)

# define function for calculating local MSTs
def local_kruskal(iterator):
    for edge in kruskal(nodes=nodes, edges=iterator):
        yield edge
# calculate local MSTs
subtrees = edges.mapPartitions(local_kruskal).collect()
```

Figure 4: Edge Partitioning Implementation on Spark

3.2 VERTEX PARTITIONING

3.2.1 ALGORITHM

In the vertex partitioning algorithm, instead of partitioning edges, we partition vertices. Like the edge partitioning algorithm, we first split vertices randomly into disjoint sets with equal size. After that rather than compute local MSTs for each partition, we consider pairs of these partitions, which means we compute local MSTs for each pair of partitions, to cover edges that go across partitions. Finally, we compute a global MST with edges belonging to local MSTs.

Algorithm 2 Vertex Partitioning Algorithm

function VERTEXPARTITION($G(V, E)$)
  Set k
  Split V into $V_!, V_2, \ldots, V_k$ using a universal hash function
  $E_{i,j} = \{(u, v) \in E | u, v \in V_i \cup V_j\}$
  $G_{i,j} = G(V_i \cup V_j, E_{i,j})$
  for i, j in $k \times k$ **do**
    $M_{i,j}$ =KRUSKAL($G_{i,j}$) ▷ In parallel
  end for
  $H = G(V, \cup_{i,j} M_{i,j})$
  M =KRUSKAL(H)
  return M
end function

3.2.2 ANALYSIS

Theorem 4 *The tree M computed by algorithm is the minimum spanning tree of G.*

Proof The algorithm works by sparsifying the graph and then computing MST of the resulting subgraph. By Lemma 1, it is clear that we are not removing the edges which are part of MST of G. Hence the resulting MST M is indeed the MST of G.

For this algorithm, we assume that we can fit $\tilde{O}(n^{1+\frac{c}{2}})$ on one machine, where $m = n^{1+c}$. $\tilde{O}$ and O is the same as O, except we ignore the logarithmic terms.

Lemma 5 *Let $k = n^{\frac{c}{2}}$, then with high probability the size of every $E_{i,j}$ is $\tilde{O}(n^{1+\frac{c}{2}})$*

Proof We bound the total number of edges in $E_{i,j}$ by bounding the total degrees of the vertices. Obviously, the number of edges of $G_{i,j}$ is smaller than the sum of degrees of vertices i.e. $|E_{i,j}| \leq \sum_{v \in V_i} \deg(v) + \sum_{v \in V_j} deg(v)$. Only for the purpose of proof, partition the vertices into groups by their degree. W_1 represents the vertices with 1 or 2 degree, W_2 represents the vertices with 3 or 4 degree. When $i \geq 2$, $W_i = \{v \in V : 2^{i-1} < \deg(v) \leq 2^i\}$. Assume G has n vertices so the vertices can be divided into logn groups.

Consider the number of vertices from W_i mapped to V_j. If the group has a small number of elements, that is, $|W_i| < 2n^{\frac{c}{2}}\text{logn}$, then $\sum_{v \in V_i} \deg(v) \leq 2n^{1+\frac{c}{2}} logn = \tilde{O}(n^{1+\frac{c}{2}})$. If the W_i is large, that is, $|W_i| > 2n^{\frac{c}{2}}\text{logn}$, following application of Chernoff bound says that the number of elements of W_i mapped into V_j is $O(logn)$ with probability at least $1 - \frac{1}{n}$. i.e.

$$P[|W_i \cap V_j| = O(logn)] \geq 1 - \frac{1}{n}$$

Let $X = |W_i \cap V_j|$, using Chernoff bounds

$$E(X) = O(logn)$$

$$P\left(X > (1+\delta)\,\text{logn}\right) \leq e^{-\frac{\delta^2 \text{logn}}{3}}$$

$$P\left(X > (1+\delta)\log n\right) \leq e^{-\frac{\delta^2 \log n}{3}}$$

$$P\left(X > (1+\delta)\log n\right) \leq \frac{1}{n} e^{\frac{\delta^2}{3}}$$

When $\delta \to 0$

$$P\left(X > logn\right) \leq \frac{1}{n}$$

$$P\left(X \leq logn\right) \geq 1 - \frac{1}{n}$$

Therefore with at least the probability of $1 - \frac{\log n}{n}$

$$\sum_{v \in V_j} \deg(v) \leq \sum_i \sum_{v \in V_j \cap W_i} deg(v) \leq \sum_i 2n^{1+\frac{c}{2}} \ n \leq \tilde{O}(n^{1+\frac{c}{2}})$$

Hence proved.

3.2.3 Processing times and communication costs

We use the following notation for the analysis:

Number of vertices: n
Number of edges: $m = n^{1+c}$
Memory of each machine: $\eta = n^{1+\frac{c}{2}}$
Number of machines required: $k = \frac{m}{\eta} = n^{\frac{c}{2}}$

We showed that with high probability, number of edges on one machine is $O(n^{1+\frac{c}{2}})$ which is $O(\frac{m}{k})$. The number of vertices on each machine is $O(\frac{n}{k})$. Also, we know that after one iteration we can fit remaining $O(n^{1+\frac{c}{2}})$ edges on single machine. We perform Kruskal's on this to obtain MST. Thus, the total processing time is:

$$O\left(\frac{m}{k} \cdot \frac{\log n}{k}\right) + O(n^{1+\frac{c}{2}} \cdot logn)$$

The communication cost involves one *one-to-all broadcast* where we broadcast the vertex partition to all C_k^2 machines. There is one *all-to-all* $groupByKey$ because we have to gather edges with same key on one machine. Lastly, there is one *all-to-one* communication for the last step where we collect all the remaining edges to perform single machine kruskal's. Thus, the total communication cost becomes:

$$O\left(nk^2\right) + O\left(m\right) + O(n^{1+\frac{c}{2}})$$

For the communication time we assume the broadcast is done by the BitTorrent model. For the final *all-to-one* collect, each machine sends $n^{1-\frac{c}{2}}$ data in expectation. Therefore, the total communication time can be written as:

$$O\left(n \log k\right) + O\left(m\right) + O(n^{1-\frac{c}{2}})$$

Note that we have assumed $k = n^{\frac{c}{2}}$ for the purposes of analysis. If we have fewer than $n^{\frac{c}{2}}$ machines, say l machines, then the processing time just gets multiplied by the factor $\frac{l}{k}$ and the communication time can be rewritten trivially in terms of l.

```
# define function for calculating combinations of different vertex partitions
def combine(iterator):
    for i in iterator:
        if i[0] < i[1]:
            yield i[0] + i[1]
vertices = sc.parallelize(shuffle(nodes), num_partition).glom()
vertices = vertices.cartesian(vertices) \
                   .mapPartitions(combine, preservesPartitioning=False)

# parallelize
vertices = sc.parallelize(vertices.collect(), num_partition)

# define function for calculating local MSTs
def local_kruskal(iterator):
    for subset in iterator:
        for edge in kruskal(nodes=set(subset), edges=edges):
            yield edge
# calculate local MSTs
subtrees = vertices.mapPartitions(local_kruskal).distinct().collect()
```

Figure 5: Vertex Partitioning Implementation on Spark

3.2.4 IMPLEMENTATION

The vertices are split randomly using the built-in $sort$ function and $random$ function. Next, we obtain every distinct pair of partitions by a combination of $cartesian$ and $mapPartitions$. After that, another $mapPartitions$ is employed with Kruskal's to obtain an RDD containing edges belonging to local MSTs. Since an edge can belong to multiple pairs of partitions, we need to use $distinct$ function to remove duplicates among edges belonging to local MSTs.

3.3 PARALLEL PRIM'S

3.3.1 ALGORITHM

The parallel prim's differs from the two aforementioned algorithm in which it works by building an MST from scratch rather than eliminating edges that do not belong to the MST. At the first step, we find the smallest edges leaving every vertex. Then, we add these edges to the MST and at each subsequent iteration, we find the smallest edges leaving each connected component and add them to the MST. The global smallest edges leaving each connected component can be found by finding the sorted smallest edges leaving each connected component on individual machines.

3.3.2 ANALYSIS

Lemma 6 *The algorithm takes at most* $\lceil log_2 n \rceil$ *iterations to complete.*

Proof At each step, we find the smallest edge leaving each connected component. Therefore, by the handshake lemma, the number of unique edges that we find at iteration i, is at least $\frac{z_i}{2}$ where z_i is the number of connected components at the beginning of iteration i. Therefore, at the end of iteration i, there are at most $\frac{z_i}{2}$ connected components. If we had more than $\frac{z_i}{2}$ unique edges, the number of connected components at the end of the iteration will only be fewer.

At the beginning of the first iteration, each vertex is in its own connected component, i.e. $z_1 = n$. Therefore, the number of unique edges added at the first iteration is at least $\frac{n}{2}$. This means, that at the end of the end of the first iteration, there are at most $\frac{n}{2}$ connected components.

Therefore, by induction, the number of connected components at the edge of iteration i is $\frac{n}{2^i}$. When we are left with just one connected component, we have found the MST. Hence, the total number of iterations taken by the algorithm is $\lceil log_2 n \rceil$.

Algorithm 3 Parallel Prim's Algorithm

function PARALLELPRIMS($G(V, E)$)
  $A = DISJOINTEST()$
  for $i \in V$ **do**
    $T = \{\}$
    A.MAKE-SET(i)
  end for
  Broadcast A
  $\hat{E}$ =List of minimum edges leaving each disjoint set ▷ In parallel
  while $|\hat{E}| > 0$ **do**
    for e in $\hat{E}$ **do**
      A.UNION(u, v)
      $T = T + e$
    end for
    Broadcast A
    $\hat{E}$ =List of minimum edges leaving each disjoint set ▷ In parallel
  end while
  $H = G(V, \cup_{i,j} M_{i,j})$
  return T
end function

3.3.3 Processing times and communication costs

The processing time for each iteration consists of 3 parts,

- Time to find the smallest edge leaving each component component in each machine
- Time to perform the reduce operation to obtain the smallest edge
- Time to find the perform the union operations to merge connected components

If implemented efficiently using path-compression the cost of each operation in a disjoint-set data structure is $O(\alpha(n))$. Since this function is grows very slowly and is effectively a constant ¡ 5 for all practical purposes, we will assume that the disjoint set operations take a constant amount of time.

We have already shown that the number of connected components at step i is at most $\frac{n}{2^i}$. The total processing time of the reduce operations can be expressed as the sum of a geometric series,

$$\frac{\text{nk}}{2} + \frac{\text{nk}}{4} + \frac{\text{nk}}{8} + \ldots + \frac{\text{nk}}{2^i} = O(nk)$$

The overall processing time for the union operations is the same as the processing time for the reduce operations since we assume that disjoint set operations take a constant amount of time. Therefore, the total processing time can be written as,

$$logn \times O\left(\frac{m}{k}\right) + O(nk)$$

At each iteration, we have one one-to-all broadcast of the disjoint set data structure. This is done using a BitTorrent-like broadcast. The size of the disjoint set data structure is $O(n)$ at each iteration and it is broadcast to k machines. Therefore, the communication cost for this is $O(nk \cdot logn)$. Since, we are using a BitTorrent-like broadcast, the communication time for this is $O(n \cdot logk \cdot logn)$.

We also have one all-to-one reduce operation to find the smallest edges leaving each connected component. Since the number of connected components at iteration i is at most $\frac{n}{2^i}$, the communication cost for this is,

$$\frac{\text{nk}}{2} + \frac{\text{nk}}{4} + \frac{\text{nk}}{8} + \ldots + \frac{\text{nk}}{2^i} = O(nk)$$

```
# edges of the minimum spanning tree
mst = []

# initialize a forest for all nodes
forest = DisjointSet(nodes)

# define function for generating graph
def generate_graph(iterator):
      for edge in iterator:
          for i in range(2):
              yield (edge[i], (edge[1-i], edge[2]))
# store the graph in an adjacency list
adjacent = sc.parallelize(edges, num_partition) \
             .mapPartitions(generate_graph, preservesPartitioning=True) \
             .groupByKey(numPartitions=num_partition) \
             .mapValues(lambda x: sorted(x, key=lambda y: y[1])) \
             .persist()
```

Figure 6: Parallel Prim's Implementation on Spark A

The communication for the reduce happens in parallel, therefore the communication time for this is,

$$\frac{n}{2} + \frac{n}{4} + \frac{n}{8} + \ldots + \frac{n}{2^i} = O(n)$$

Therefore, the total communication cost is,

$$O\left(nk \cdot logn\right) + O(nk)$$

The total communication time is,

$$O\left(n \cdot logk \cdot logn\right) + O(n)$$

3.3.4 Implementation

We use disjoint set structure to indicate connectedness of vertices and broadcast it to all machines. Before we use $mapPartitions$ to find the minimum edges in each machine, we first sort all edges by their weights, so that we find the local smallest edges more quickly.

Then, we user $distinct$t to eliminate duplicates among edges belonging to local MSTs. To find the global smallest edges, we simply sort all local smallest edges. Lastly, we update the connectedness of vertices and broadcast it to all machines again.

4 Experiments

4.1 Experiments setup

Our distributed algorithms experiments setup are shown in Table 1:

It's should be noticed that our test tasks were done in one machine and we divided our **RDD** into 4 parallel partitions to do computation.

4.2 Dataset and Results

The experiment was done on 8 different networks: *netscience*, *cora*, *dblp*, *web-NotreDame*, *web-Stanford*, *web-Google*, *web-BerkStan* and *roadNet-PA*. In table 2, we order these networks and their results from top to bottom according to the complexity. Nodes from 1,589 to 1,088,092 and edges from 2,742 to 1,541,898.

```
# candidate edges of the global MST
candidates = [None]
# loop until there is no candidate
while len(candidates) != 0:
    # broadcast the forest to each machine
    connection = sc.broadcast(forest)

    # define function for finding minimum edges leaving each disjoint set
    def find_minimum(iterator):
        for group in iterator:
            src = group[0]
            for (dst, weight) in group[1]:
                if connection.value.find(src) != connection.value.find(dst):
                    yield (src, dst, weight) if src < dst else (dst, src, weight)
                    break
    # obtain the list of minimum edges leaving each disjoint set
    candidates = sorted(adjacent.mapPartitions(find_minimum).distinct().collect(), key=lambda
        x: x[2])

    # calculate the global MST
    for candidate in candidates:
        # find the parents of src and dst respectively
        if forest.unite(candidate[0], candidate[1]):
            # add the current edge into the minimum spanning tree if it doesn't make a circuit
            mst.append(candidate)
```

Figure 7: Parallel Prim's Implementation on Spark B

Table 1: Performance Experiment Setup

OS	Windows10
Framework	Spark 2.4.4 and Hadoop 2.7.7
RAM	24G RAM
SSD	500G SSD
CPU	Intel Core i7-7500 (2 cores / 4 threads)
Partition	4 Partitions

To have a better view of the experiment results, we visualize the result in this bar chart. The blue, pink and red represent edge partition, vertex partition and parallel prim respectively. The height of the bar shows the running time of an algorithm on a specific network.

We could see that the edge partition is the fastest algorithm among them. No matter how network becomes complicated, the edge partition could have stable performance. Moreover, when the edge number is small, vertex partition algorithm takes the longest time. After the *web-NotreDame*, parallel prim takes more time than vertex partition and it becomes slower and slower. As the complexity

Table 2: Dataset and Results

Networks	**Nodes**	**Edges**	**Edge Partition**	**Vertex Partition**	**Parallel Prim**
netscience	1,589	2,742	8.1s	28.1s	16.9s
cora	2,708	5,429	8.4s	28.8s	21.1s
dblp	317,080	1,049,866	21.0s	72.1s	63.5s
web-NotreDame	325,729	1,497,134	17.6s	72.6s	46.1s
web-Stanford	281,903	2,312,497	38.0s	95.8s	117.6s
web-Google	875,713	5,105,039	59.9s	180.9s	195.2s
web-BerkStan	685,230	7,600,595	69.4s	231.8s	334.7s
roadNet-PA	1,088,092	1,541,898	62.7s	185.8s	202.7s

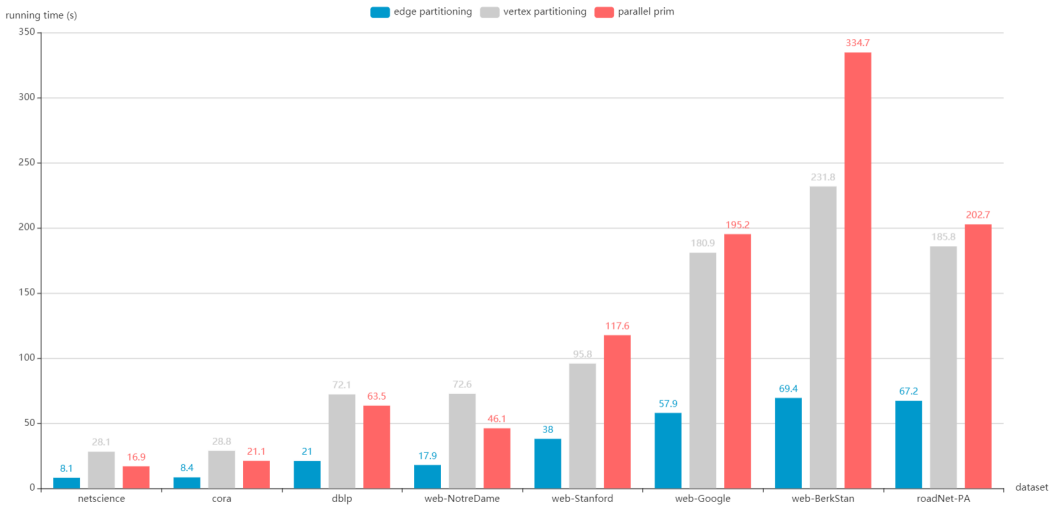

Figure 8: The Experiment Results About Three Distributed Algorithms Across Different Networks

of the network increases, the whole execution time of parallel prim rise from 16.9s to 202.7s. However, the edge partition algorithm only need half time of it in *roadNet-PA*.

Actually, the experiment results are somehow different from our hypothesis. Unfortunately, due to the lack of time, we are not able to conduct a rigorous analysis on it. Nevertheless, we are still able to propose some assumptions to explain the difference. The first possible reason is that our hypothesis is fully based on mathematics, not considering the efficiency of code. The second assumption is that our algorithms were tested on a single machine, resulting in an unexpected decay of performance.

5 Conclusion

- Based on pyspark, three distributed algorithms for computing the MST of a given graph are implemented and compared based on processing time.
- In all cases, the performance of Edge Partition is superior to Vertex Partition and Parallel Prim algorithms.

6 Future Work

- Since all of the above algorithms were tested on one machine, we did not study the scalability of the algorithms in a practical setting. The performance of the algorithm may vary depending on the speed of the CPU and the amount of available network bandwidth.
- Not all of the running time as mentioned above times are expected, and sometimes there is even a large gap from our expectations. In fact, the running time depends on the structure of the graph and may vary greatly depending on the structure of the graph. Therefore, it is significant to study the performance of algorithms on different types of graphs.
- There may be ways to improve the design of the algorithm further. For example, there may be a bright partitioning scheme for edge partitioning and vertex partitioning. Alternatively, for parallel Prim's, there may be a way to cache the smallest edges so that each connected component eliminates duplicate work.

References

Greedy Algo. Kruskal's minimum spanning tree algorithm. `https://www.geeksforgeeks.org/kruskals-minimum-spanning-tree-algorithm-greedy-algo-2/`.

Cho Jui Hsieh, Si Si, Donghyuk Shin, and Inderjit Dhillon. Divide conquer methods for big data analytics. 2006.

Rohit Patki. Distributed minimum spanning trees. *Swaroop Indra Ramaswamy*, 2015.