Distributed Minimum Spanning Trees

Swaroop Indra Ramaswamy & Rohit Patki

June 3, 2015

Abstract

Minimum spanning trees are one of the most important primitives used in graph algorithms. They find applications in numerous fields ranging from taxonomy to image processing to computer networks. In this report, we present 3 algorithms to compute the minimum spanning tree (MST) or minimum spanning forest (MSF) for large graphs which do not fit in the memory of a single machine. We analyze the theoretical processing time, communication cost and communication time for these algorithms under certain assumptions. Finally, we compare the performance of the 3 algorithms on real-world data using Apache Spark [1]. The code for the project can be found here. `https://github.com/s-ramaswamy/CME-323-project`

1 Introduction

The spanning tree of a connected, undirected graph is a subgraph of the graph, that is a tree that connects all the vertices. The minimum spanning tree is the spanning tree with least sum of edge weights. The minimum spanning forest is a generalization of the minimum spanning tree for unconnected graphs. A minimum spanning forests consists of minimum spanning trees on each of the connected components of the graph.

2 Applications

Minimum spanning trees find applications in a range of fields. A few of them are listed below.

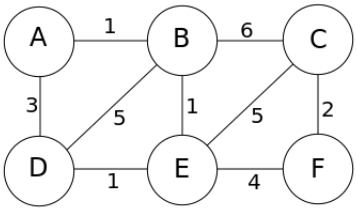
- They are an important part of many approximation algorithms for NP-hard and NP-complete problems. For example, the first step in most approximation algorithms for the Steiner tree problem requires computing the MST. It is also the first step in the Christofede's algorithm for the traveling salesman problem.
- They find numerous applications in image processing. For example, if you have an image of cells on a slide, then you could use the minimum spanning tree of the graph formed by the nuclei to describe the arrangement of these cells.
- They are the basis for single-linkage clustering. Single-linkage clustering is a hierarchical clustering method. Each element is in its own cluster at the beginning. The clusters are then sequentially combined into larger clusters, until all elements end up being in the same cluster. At each step, the two clusters separated by the shortest distance are combined. We will later see that this process basically mimics the Kruskal's algorithm for constructing the MST.
- An obvious application of MST is in construction of road or telephone networks. We would like to connect places/houses with the minimum length of road/wire possible. This is exactly the same as computing the minimum spanning tree.

3 Properties

3.1 Cut property

Theorem 3.1 *For any cut C in the graph, if the weight of an edge e of C is strictly smaller than the weights of all other edges of C, then this edge belongs to all MSTs of the graph.*

Proof Assume the contrary, i.e., in the figure at the bottom, make edge BC (weight 6) part of the MST T instead of edge e (weight 4). Adding e to T will produce a cycle, while replacing BC with e would produce MST of smaller weight. Thus, a tree containing BC is not a MST, a contradiction that violates our assumption.



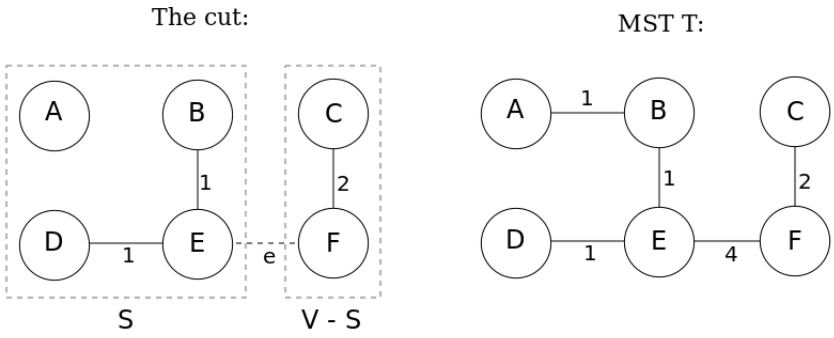


Figure 1: This figure shows the cut property of MST. T is the only MST of the given graph. If $S = \{A, B, D, E\}$, thus $V - S = \{C, F\}$, then there are 3 possibilities of the edge across the cut $(S, V - S)$, they are edges BC, EC, EF of the original graph. Then, e is one of the minimum-weight-edge for the cut, therefore $S \cup \{e\}$ is part of the MST T.

3.2 Cycle property

Theorem 3.2 *For any cycle C in the graph, if the weight of an edge e of C is larger than the individual weights of all other edges of C, then this edge cannot belong to a MST.*

Proof Assume the contrary, i.e. that e belongs to an MST T_1. Then deleting e will break T_1 into two subtrees with the two ends of e in different subtrees. The remainder of C reconnects the subtrees, hence there is an edge f of C with ends in different subtrees, i.e., it reconnects the subtrees into a tree T_2 with weight less than that of T_1, because the weight of f is less than the weight of e.

4 Single-machine algorithms

4.1 Classical algorithms

The two most popular algorithms for minimum spanning trees on a single machine are Kruskal's algorithm and Prim's algorithm [2]. Both of them have the same computational complexity and are relatively simple to implement.

4.1.1 Kruskal's algorithm

Algorithm 1 Kruskal's algorithm

Require: All the edges in E are sorted in increasing order by weight

function KRUSKAL($G(V, E)$)
  $A = \Phi$
  for $i \in V$ **do**
    MAKE-SET(i)
  end for
  for (u, v) in E **do**
    if FIND-SET(u)$\neq$FIND-SET(v) **then**
      $A = A \cup (u, v)$
      UNION(u, v)
    end if
  end for
  return A
end function

Basically, Kruskal's algorithm begins with a sorted list of edges and adds each edge to the tree if it does not form a cycle. By the cycle property, we know that if the weight of an edge is greater than all the other edges in a cycle, then this edge cannot be part of the MST. Therefore, we only add edges which are part of the MST.

Using a simple implementation of the disjoint-set data structure we can achieve a run-time of $O(m \log n)$. If we use a more sophisticated disjoint-set data structure, then we can achieve a run-time of $O(m\alpha(m, n))$ where α is the extremely slowly growing inverse of the Ackermann function. However, to achieve this bound, we need to assume that the edges are already sorted or can be sorted in linear time.

4.1.2 Prim's algorithm

Algorithm 2 Prim's algorithm

function PRIM($G(V, E)$)
  $A = V_0$
  $B = V$
  $T = \{\}$
  while $B \neq \Phi$ **do**
    Find smallest $(u, v) \in E$ such that $u \in A$ and $v \in B$
    $T = T + (u, v)$
    $A = A + v$
    $B = B - v$
  end while
  return T
end function

Prim's algorithm finds the minimum weight edge leaving the current tree and appends this edge to the tree. By the cut property, we know that the minimum weight edge leaving any cut is in the MST. Therefore,

we add only the edges that belong to the MST.

If we use a simple binary heap and a priority queue, the complexity of the algorithm is $O(m \log n)$. However, if we use the more sophisticated Fibonacci heap, it can be shown that the run-time is $O(m + n \log n)$ which is asymptotically better for dense graphs.

4.2 Faster algorithms

The fastest non-randomized algorithm with known complexity is by Bernard Chazelle that runs in $O(m\alpha(m, n))$ which for all practical purposes is linear time. However, this uses a soft heap and an approximate priority queue and is difficult to implement. Karger, Klein & Tarjan found a linear time randomized algorithm which uses only comparison of edge weights to find the MST. For integer edge weights, the current fastest algorithm, developed by Fredman and Willard takes $O(m + n)$ time.

For the purposes of this project, we will only consider the classical algorithms owing to their ease of implementation vis-à-vis the newer, faster algorithms.

5 Distributed Algorithms

We will look at three distributed algorithms. Two of them are based on Kruskal's algorithm - *edge partitioning* [3] and *vertex partitioning* [4], and the third, *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.

First, we will look at the algorithms based on Kruskal's algorithm. There is a common theme involved in both algorithms. The following lemma helps in establishing the correctness of the algorithms in a distributed setting.

Lemma 5.1 *Any edge that does not belong to MST of a subgraph containing the edge does not belong to the MST of the original graph.*

Proof The fact that the edge does not belong to the MST of subgraph implies that there exists another edge with less weight in the cut that separates the two vertices of the original edge. This holds true for the original graph as well since we are just considering a subgraph. Hence, we edge will not feature in the MST of the original graph.

In the following algorithms we find MSTs on different subgraphs and in the process, keep discarding edges that don't belong to the global MST. Once we reach a stage where we can store the remaining edges on one machine, we use the single machine version of Kruskal's algorithm to obtain the final MST.

5.1 Edge Partitioning

The edges do not fit in a memory of single machine and the edges on each machine form a subgraph. Local MST is computed using the single machine version of Kruskal's algorithm. Next, these remaining edges, which are part of local MSTs are randomly split across the machines. Again, local MSTs are computed and this process goes on until we can fit all the remaining edges on one machine. Once this is possible, we gather them on one machine and use Kruskal's algorithm to compute the final MST.

Algorithm 3 Edge partitioning algorithm

function EDGEPARTITION($G(V, E)$)
  η = Memory of each machine
  $e = E$
  while $|e| > \eta$ **do**
    $l = \Theta\left(\frac{|E|}{\eta}\right)$
    Split e into e_1, e_2, e_3 ... e_l using a universal hash function
    Compute T_i^* = KRUSKAL($G(V, e_i)$) ▷ In parallel
    $e = \cup_i T_i^*$
  end while
  A = KRUSKAL($G(V, e)$)
  return A
end function

5.1.1 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, $l = n^{c-\epsilon}$

Lemma 5.2 *The algorithm terminates after $\lceil \frac{c}{\epsilon} \rceil$ iterations and returns the Minimum Spanning Tree.*

Proof By lemma 5.1 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 number of edges on each machine is η. After one iteration, $|\cup_i T_i| \leq l(n-1) = O(n^{1+c-e})$. 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 $\lceil \frac{c}{\epsilon} \rceil - 1$ iterations the input is small enough to fit onto a single machine, and the overall algorithm terminates after $\lceil \frac{c}{\epsilon} \rceil$ 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 5.3 *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:

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

To find optimal values for l_i, we differentiate the above expression with respect to each l_i and set it to zero.

We thus obtain:

$$\begin{aligned} l_1^2 &= l_2 \frac{m}{n} \\ l_2^2 &= l_1 l_3 \\ l_3^2 &= l_2 l_4 \\ &\vdots \\ l_i^2 &= l_{i-1} l_{i+1} \\ &\vdots \\ l_{t-2} &= l_{t-3} l_{t-1} \\ l_{t-1}^2 &= l_{t-2} \end{aligned}$$

Solving further and using $\epsilon = \frac{c}{t}$:

$$\begin{aligned} l_1 &= \left(\frac{m}{n}\right)^{\frac{t-1}{t}} = n^{c-\epsilon} \\ l_2 &= \left(\frac{m}{n}\right)^{\frac{t-2}{t}} = n^{c-2\epsilon} \\ l_3 &= \left(\frac{m}{n}\right)^{\frac{t-3}{t}} = n^{c-3\epsilon} \\ &\vdots \\ l_i &= \left(\frac{m}{n}\right)^{\frac{t-i}{t}} = n^{c-i\epsilon} \\ &\vdots \\ l_{t-1} &= \left(\frac{m}{n}\right)^{\frac{1}{t}} = n^{c-(t-1)\epsilon} \\ \therefore \frac{|e_i|}{l_i} &= \frac{n^{1+c-(i-1)\epsilon}}{l_i} = n^{1+\epsilon} = \text{constant} \end{aligned}$$

5.1.2 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:

$$\lceil \frac{c}{\epsilon} \rceil \left(\underbrace{O\left(\frac{m}{l} \log n\right)}_{Kruskal's\ on\ each\ machine} + \underbrace{O\left(\frac{m}{l}\right)}_{random\ partioning\ of\ edges} \right)$$

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:

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

5.1.3 Implementation

The random splitting of the edges is achieved by a **map** operation using scala random function. Then, using **groupByKey** the edges with the same key are grouped on a single machine. Next, **flatMap** is employed with Kruskal's to obtain an RDD containing only the edges belonging to local MSTs.

5.2 Vertex partitioning

In this algorithm, instead of partitioning the edges, we partition the vertices. We fix a number k and randomly split the vertices into k equally sized partitions, $V = V_1 \cup V_2 \cup V_3 \cdots \cup V_k$ and $V_i \cap V_j = \emptyset$ for $i \neq j$. Therefore $|V_i| = \frac{n}{k}$. If we consider each of these partitions and compute the MSTs on the subgraphs induced by them we are ignoring the edges that go across the partitions. Instead, we consider pairs of these partitions and compute local MSTs. For each pair i, j, let $E_{i,j} \subseteq E$ be the set of edges induced by the vertex set $V_i \cup V_j$. That is, $E_{i,j} = \{(u, v) \in E | u, v \in V_i \cup V_j\}$. Let the resulting subgraph be denoted by $G_{i,j} = (V_i \cup V_j, E_{i,j})$. So there are $\binom{k}{2}$ such subgraphs. For each such $G_{i,j}$, compute unique MST $M_{i,j}$. Let H be the graph consisting of all the edges present in some $M_{i,j} : H = (V, \cup_{i,j} M_{i,j})$. Next, according to the assumption, we can fit H on memory of single machine. Compute M, the MST of H which is the MST of the original graph G.

Algorithm 4 Vertex partitioning algorithm

```
1: function VertexPartition(G(V, E))
2:     Set k
3:     Split V into V_1, V_2, V_3 ... V_k using a universal hash function
4:     E_{i,j} = {(u, v) ∈ E | u, v ∈ V_i ∪ V_j}
5:     G_{i,j} = G(V_i ∪ V_j, E_{i,j})
6:     for i, j in k × k do
7:         M_{i,j} = KRUSKAL(G_{i,j})                    ▷ In parallel
8:     end for
9:     H = G(V, ∪_{i,j} M_{i,j})
10:    M = KRUSKAL(H)
11:    return M
12: end function
```

5.2.1 Analysis

Theorem 5.4 *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 5.1**, 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 $m = n^{1+c}$. $\tilde{O}$ is the same as O, except we ignore the logarithmic terms.

Lemma 5.5 *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. $|E_{i,j}| \leq \sum_{v \in V_i} deg(v) + \sum_{v \in V_j} deg(v)$. Only for the purpose of proof, partition the vertices into groups by their degree. Let W_1 be the set of vertices of degree atmost 2, W_2, the set of vertices with degree 3 or 4, and generally $W_i = \left\{v \in V : 2^{i-1} < deg(v) \leq 2^i\right\}$. Thus there are $\log n$ total groups.

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

Let X = number of elements of W_i mapped into partition j. Then, using Chernoff bounds:

$$E[X] = O(\log n)$$
$$P(X > (1+\delta)\log n) \leq e^{\frac{-\delta^2 \log n}{3}}$$
$$P(X > (1+\delta)\log n) \leq \frac{1}{n} e^{\frac{\delta^2}{3}}$$
$$\text{as } \delta \to 0, P(X > \log n) \leq \frac{1}{n}$$
$$\text{Therefore }, P(X \leq \log n) \geq 1 - \frac{1}{n}$$

Thus, with probability at least $1 - \frac{\log n}{n}$:

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

Hence proved.

5.2.2 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 $= n^{1+\frac{c}{2}}$

Number of machines required, $k = n^{\frac{c}{2}}$

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:

$$\underbrace{O\left(\frac{m}{k}\log\frac{n}{k}\right)}_{Kruskal's\ on\ each\ machine} + \underbrace{O\left(n^{1+\frac{c}{2}}\log n\right)}_{Final\ Kruskal's}$$

The communication cost involves one *one-to-all* **broadcast** where we broadcast the vertex partition to all $\binom{k}{2}$ 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:

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

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:

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

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.

5.2.3 Implementation

The vertices are split randomly using **map** operation with scala's random number generator. It is then broadcasted to all the machines. Next, we go through all the edges on a machine is figure out the pair of partition $E_{i,j}$ it belongs to. Using a **flatMap**, each edge is alloted the key corresponding to its pair of partition. A **groupByKey** operation collects all the edges belonging to one partition on a single machine. Then we apply Kruskal's algorithm locally. Since an edge can belong to multiple pair of partitions, after the first Kruskal's we need to use **distinct** function to remove the duplicates. One more Kruskal's on the remaining edges gives us the final MST.

5.3 Parallel Prim's

This algorithm is different from the two algorithm described above in that it works by building up an MST from scratch rather than eliminating edges that do not belong to the MST. Again, the assumption is that that the edges do not fit in the memory of a single machine but the vertices do. At the first step, we find the smallest edge leaving every vertex. We add these edges to the MST and then at each subsequent iteration, we find the smallest edges leaving each connected component and add them to the MST. The smallest edges leaving each connected component can be found by finding the smallest edge leaving each connected component on individual machines and then performing a reduce operation to get the smallest edge overall.

Algorithm 5 Parallel Prim's Algorithm

```
1:  function ParallelPrims(G(V, E))
2:      A = DISJOINTSET()
3:      for i in V do
4:          T = {}
5:          A.MAKE-SET(i)
6:      end for
7:      Broadcast A
8:      Ê = List of minimum edges leaving each disjoint set                ▷ In parallel
9:      while |Ê| > 0 do
10:         for e in Ê do
11:             A.UNION(u, v)
12:             T = T + e
13:         end for
14:         Broadcast A
15:         Ê = List of minimum edges leaving each disjoint set            ▷ In parallel
16:     end while
17:     return T
18: end function
```

5.3.1 Analysis

Lemma 5.6 *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 hand-shake lemma, the number of unique edges that we find at iteration i, is at least $\frac{1}{2} \times z_i$ 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, $\implies z_1 = n$. 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 1 connected component, we have found the MST. Hence, the total number of iterations taken by the algorithm is $\lceil \log_2 n \rceil$.

5.3.2 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,

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

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,

$$\underbrace{\log n}_{iterations} \times \underbrace{O\left(\frac{m}{k}\right)}_{per\ iteration} + \underbrace{O(nk)}_{Total\ cost\ of\ all\ the\ reduces}$$

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 $O(nk \log n)$. Since, we are using a BitTorrent-like broadcast, the communication time for this is $O(n \log k \log n)$.

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,

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

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

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

Therefore, the **total communication cost** is,

$$O(nk \log n) + O(nk)$$

The **total communication time** is,

$$O(n \log k \log n) + O(n)$$

5.3.3 Implementation

We used a custom disjoint set class implemented in scala. However, to save computation and to avoid writing custom serialization methods, we use a hashtable to store the disjoint set each vertex belongs to and broadcast this hashtable. We then perform a **map** operation to find the minimum edges in each machine and do a **reduce** operation to get the overall minimum edges. After this, we go through each edge in the list of edges returned by the reduce and perform the union.

6 Theoretical comparison

The dominant terms in the processing and communication time for each of the 3 distributed algorithms is listed below.

	Edge Partitioning	Vertex Partitioning	Parallel Prim's
Processing Time	$O\left(\frac{m}{\epsilon k}\log n\right)$	$O\left(\left(\frac{m}{n^{\frac{c}{2}}}+n^{1+\frac{c}{2}}\right)\log n\right)$	$O\left(\frac{m}{k}\log n+nk\right)$
Communication Time	$O\left(m\frac{1-n^{-c}}{1-n^{-\epsilon}}\right)$	$O\left(m+cn\log n\right)$	$O\left(n\log k\log n\right)$

Recall that $m = n^{1+c}$ and the memory per machine is $n^{1+\epsilon}$.

We plot the dominant terms to study how they grow with respect to change in c. Here, for the purposes for the plots, we set $\epsilon = \frac{c}{5}$.

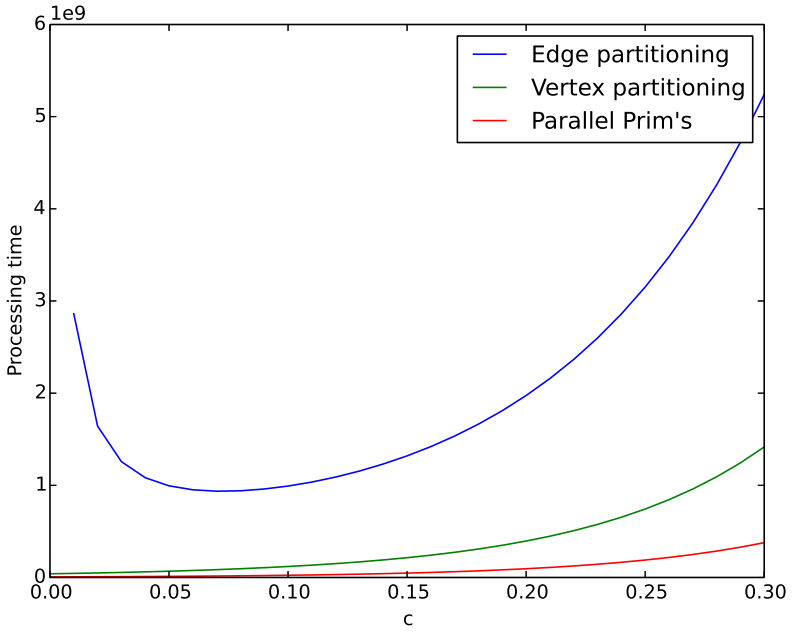


Figure 2: Processing time vs c, for n = 1,000,000

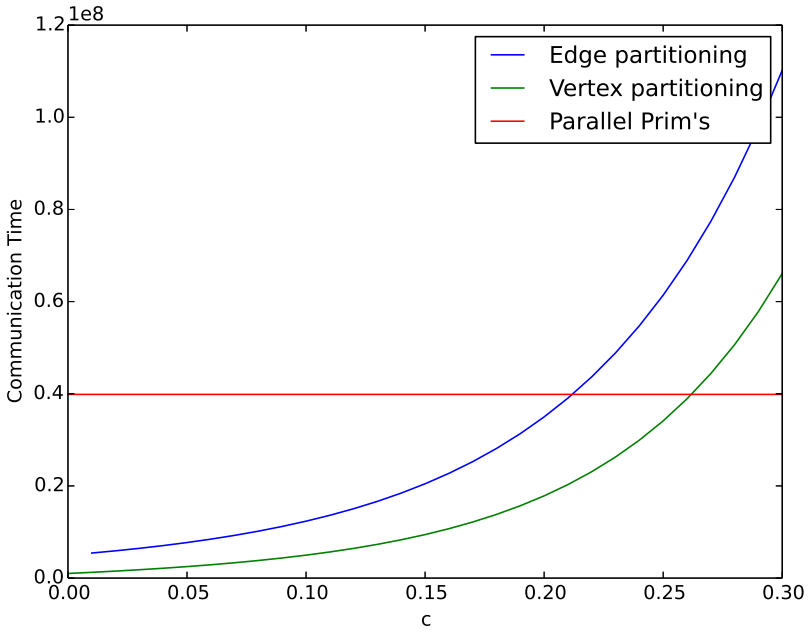


Figure 3: Commmunication time vs c, for n = 1,000,000

We see that the processing time for the parallel Prim's algorithm is smaller and grows much slower than the processing time for edge partitioning and vertex partitioning. The communication time for parallel Prim's might be larger than the communication time for edge partitioning or vertex partitioning for sparse graphs but as the density of the graph increases, parallel Prim's does better in terms of both processing time and communication time since the communication time is independent of the number of edges.

7 Experimental comparison

	No.of Vertices	No.of Edges	Edge Partitioning	Vertex Partitioning	Parallel Prim's
`web-Stanford`	281903	2312497	791 s	316 s	92 s
`web-Google`	875713	5105039	7384 s	3733 s	229 s
`web-BerkStan`	685230	7600595	3670 s	1569 s	313 s
`as-Skitter`	1696415	11095298	$>$ 7200 s	$>$ 3600 s	335 s
`roadnet-PA`	1088092	1541898	$>$ 7200 s	$>$ 3600 s	394 s

Table 1: Graphs obtained from [5]

The experimental results are in line with the theoretical predictions. One anomaly we notice is that even though the graph `as-Skitter` has more vertices and edges than `roadnet-PA`, the Parallel Prim's runtime for `as-Skitter` is smaller than the runtime for `roadnet-PA`. This is because the number of iterations Parallel Prim's takes to run depends on the structure of the graph. The analysis assumes the worst case of $\lceil \log_2 n \rceil$ iterations but in practice it takes fewer steps depending on the structure of the graph.

8 Conclusions

- Three distributed algorithms for computing MST of a given graph were presented and compared based on processing time and communication time.
- The parallel Prim's algorithm's communication time is independent of the number of edges.
- For sparse graphs, the communication time for vertex partitioning algorithm is less than that of parallel Prim's while the processing time is better for Prim's algorithm. Hence in some cases, vertex partioning algorithm might perform better and it is worth consideration.
- For dense graphs, parallel Prim's algorithms wins in both, communication time and processing time.
- Both vertex partitioning and parallel Prim's outperform the edge partitioning algorithm in all cases.

9 Future work

- Since all the above algorithms were tested on 4 cores at the most, we did not study how well the algorithms scale in a practical setting. Depending on how fast the CPUs are, and how much network bandwidth is available, there could be variations in the performance of the algorithms.
- All the runtimes presented above were in expectation or sometimes, worst case. In reality, they depend on the structure of the graph and there can be large variations depending on the structure of the graph as we saw with `as-Skitter` and `roadnet-PA`. It would be worthwhile to study the performance of the algorithms on different types of graphs.
- There might be ways to further improve the design of the algorithms. For example, one might be able to think of a clever partitioning scheme for edge partitioning and vertex partitioning. Or, for parallel Prim's, one might be able to think of a way of caching the smallest edges leaving each connected component eliminating some duplication in work done.

10 Acknowledgements

We thank Reza Zadeh and Dietrich Lawson for helping us in both theoretical and computational aspects of this project.

References

[1] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, pages 10–10, 2010.

[2] Reza Zadeh. Discrete mathematics and algorithms. `http://web.stanford.edu/class/cme305/Notes/2.pdf`.

[3] Silvio Lattanzi, Benjamin Moseley, Siddharth Suri, and Sergei Vassilvitskii. Filtering: a method for solving graph problems in mapreduce. In *Proceedings of the twenty-third annual ACM symposium on Parallelism in algorithms and architectures*, pages 85–94. ACM, 2011.

[4] Howard Karloff, Siddharth Suri, and Sergei Vassilvitskii. A model of computation for mapreduce. In *Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 938–948. Society for Industrial and Applied Mathematics, 2010.

[5] Jure Leskovec and Rok Sosič. SNAP: A general purpose network analysis and graph mining library in C++. `http://snap.stanford.edu/snap`, June 2014.