Proposal: Implementation of Minimum Spanning Tree Algorithm Using Spark Python API

YANG Rongfeng *20644943*   LAU Kwan Yuen *20647191*   TANG Huimin *20636635*   YIN Dongxin *20658592*

MSBD 5003, Big Data Technology, HKUST
{ryangag,klauak,htangak,dyinaa}@connect.ust.hk

1 Introduction

Our team decides to carry out a deep reseach on Minimum Spanning Tree Algorithms (MST), which is one of the most important primitives used in graph algorithms. To solve MST problems, there are two classical algorithms used widely, i.e Kruskal's algorithm and Prim's algorithm. In our project, we will implement three different parallel algorithms [1], which are adated from either Kruskal's algorithm or Prim's algorithm, to improve the speed of solving MST problems. For implementation, we plan to use the Apache Spark Python API. Finally, we will compare the performances of different methods.

2 Algorithm

2.1 Classical Algorithms

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

2.1.1 Kruskal's Algorithm

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 $O(m \cdot logn)$. 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.

Algorithm 1 Kruskal's algorithm

```
Require: All the edges in $E$ are sorted in increasing order by weight
 1: function KRUSKAL($G(V,E)$)
 2:     $A = \Phi$
 3:     for $i \in V$ do
 4:         MAKE-SET($i$)
 5:     end for
 6:     for $(u,v)$ in $E$ do
 7:         if FIND-SET($u$) $\neq$ FIND-SET($v$) then
 8:             $A = A \cup (u,v)$
 9:             UNION($u,v$)
10:         end if
11:     end for
12:     return $A$
13: end function
```

2.1.2 Prim's Algorithm

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 $O(mlogn)$. However, if we use the more sophisticated Fibonacci heap, it can be shown that the run-time is $O(m + nlogn)$ which is asymptotically better for dense graphs.

2.2 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, it 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

Algorithm 2 Prim's algorithm

```
1: function PRIM($G(V,E)$)
2:     $A = V_0$
3:     $B = V$
4:     $T = \{\}$
5:     while $B \neq \Phi$ do
6:         Find smallest $(u,v) \in E$ such that $u \in A$ and $v \in B$
7:         $T = T + (u,v)$
8:         $A = A + v$
9:         $B = B - v$
10:    end while
11:    return $T$
12: end function
```

weights to find the MST. For integer edge weights, the current fastest algorithm, developed by Fredman and Willard takes $O(m+n)$ time.

3 Distributed Algorithms and Implementation Plan in Spark

We will look at three distributed algorithms. Two of them are based on Kruskal's algorithm - edge partitioning and vertex partitioning, 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.

3.1 Edge Partitioning Algorithm

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

Following this scheme, we come up with a divide-and-conquer algorithm, i.e., to split edges randomly into disjoint sets and compute local MSTs parallelly first, and then compute a final MST with edges belonging to local MSTs. To notice that, all distributed machines share a same set of vertices when computing local MSTs.

The random splitting of edges is achieved by a **map** operation using random function. Then, with **groupByKey**, edges with 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.

Algorithm 3 Edge Partitioning Algorithm

```
1: function EDGEPARTITION($G(V,E)$)
2:     $\eta$ =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
```

3.2 Vertex Partitioning

In the Vertex Partitioning Algorithm, instead of partitioning edges, we partition vertices. Similar to 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 compute local MSTs for each pair of partitions, to cover edges that go across partitions. Finally, we compute a final MST with edges belonging to local MSTs.

The vertices are splited randomly using **map** operation with random number generator. It is then broadcasted to all the machines. Next, we go through all edges on a machine to figure out their corresponding pairs of partition. Using a **flatMap**, each edge is assigned a key corresponding to its pair of partition. Then, a **groupByKey** operation collects all edges belonging to a partition on a single machine. 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.

Algorithm 4 Vertex Partitioning Algorithm

```
1: function VERTEXPARTITION($G(V,E)$)
2:     Set $k$
3:     Split $V$ into $V_!, V_2, \ldots, V_k$ using a universal hash function
4:     $E_{i,j} = \{(u,v) \in E | u,v \in V_i \cup V_j\}$
5:     $G_{i,j} = G(V_i \cup V_j, E_{i,j})$
6:     for $i,j$ in $k \times k$ do
7:         $M_{i,j}$ =KRUSKAL($G_{i,j}$)    ▷ In parallel
8:     end for
9:     $H = G(V, \cup_{i,j} M_{i,j})$
10:    $M$ =KRUSKAL($H$)
11:    return $M$
12: end function
```

3.3 Parallel Prim's

This algorithm 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 smallest edges leaving each connected component on individual machines and then performing a reduce operation on the local smallest edges to get the global smallest edges.

We use disjoint set structure to indicate connectednesses of vertices and broadcast it to all machines. 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 that, we go through each edge in the list of edges returned by the **reduce** and perform a **union**.

Algorithm 5 Parallel Prim's Algorithm

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

4 Timetable

Oct. 19th Ensure the topic and submit a writeen proposal

Oct. 21st Search for references

Oct. 28th Implement the algorithms by Spark python API

Nov. 4th Write a final report

Nov. 11st Prepare slides for presentation

Nov. 18th Present our project

Dec. 2nd Submit a final report

References

[1] Swaroop Indra Ramaswamy and Rohit Patki. Distributed minimum spanning trees. *Stanford Education CME323 Projects*, 2015.