Distributed Systems Spring 2018: Project Report

Umesh Singla (201401204) **Aagam Shah (201464155)**

Approximate-MST

M. Khan, G. Pandurangan, A fast distributed approximation algorithm for minimum spanning trees. Distributed Computing 20(6): 391-402 (2008)

Contents:

Main Contribution/ Problem Statement

1. An efficient implementation of nearest neighbor tree scheme (NNT) in a general weighted graph

Idea

1. Construct NNT, in a distributed fashion:
    1. each node chooses a unique rank
    2. each node connects to the nearest node of higher rank (except the highest ranked node i.e the leader node)
2. NNT around each node is approximately a spanning subgraph

Assumptions

1. Weighted undirected connected graph
2. Nodes only know about their immediate neighbors
3. Maximum weight W <= n

<u>Algorithm steps</u>

1. Step 1: Rank Selection:
    1. there's a leader - process 0 in our case.
    2. leader picks a number p(leader) and sends this number and ID to all of its neighbors; neighbors choose a number p(v) from [p, p-1) and send these numbers and their IDs to their neighbors and so on. Clearly, multiple nodes can have same p's.
    3. if there are multiple messages from neighbors (there will be), then decide your rank according to the first one received and just store the rest ones.
    4. rank of a vertex r(v) comprises both of its ID(v) and this number p(v).

2. Step 2: Connection to a higher-rank node:
    1. Note: it is sufficient for a node to explore the nodes only in the neighborhood limited by maximum weighted edge attached to it
    2. A node checks for higher ranked node in its neighborhood defined by ρ (phase). ρ = 1 initially and keeps doubling till it finds a node to connect to. ρ needs to be increased to at most the largest weight adjacent to it.
    3. While ρ is fixed, there are multiple rounds λ (λ = 1 to begin with). So, the actual neighborhood is defined by both ρ and λ.
    4. A node sends explore messages <*explore*, v, *r*(v), ρ, λ, *pd*, *l*> to all its neighbors lying within distance ρ.
    5. At the end of each round, v counts the number of the nodes it has explored.
    6. If the number of nodes remain the same in two successive rounds of the same phase (that is, v already explored all nodes in ρ(v)), then v doubles ρ and starts the next phase ρ.
    7. When v finds a node of higher rank, it terminates its exploration with that node sending a found message back on the path towards v.

3. Step 3: Making Connection:
    1. After receiving a found message from any of the (higher-ranked) node, it sends a connect message along the same path found message was received.

<u>Types of messages used</u>

1. *explore*: visiting-for-the-first-time message

2. *count*: sent as a reply to explore message, used to determine the number of nodes explored by v in this round (individual gets added on the path)
3. *found*: reply to explore message - a node with a higher rank was found in the exploration
4. *connect*: After a node has found a higher-ranked node, this type of message is sent along that path
5. *done*: after a node has connected, it sends a done message to the leader (process 0 in this case)
6. *break*: leader process sends this message after receiving *done* messages from all the processes

Advantages

Each node individually finds the nearest node of higher rank to connect to, and hence no need of explicit coordination among the nodes

1. Graphs containing edges of lower weights converge faster since the number of phases for a node is limited by the highest weight of edge incident on it.

Algorithm challenges

1. Avoiding cycle formation
    - solved by using “incremental” neighborhood exploration

Implementation challenges

1. The algorithm assumes each node to be a processor and thus we need n MPI threads for a graph with n nodes (thus n>200 is tough to achieve even with two systems)
2. Congestion (since nodes explore their neighborhoods simultaneously and many nodes may have overlapping ρ-neighborhoods)
3. Message is sometimes received but dropped without processing

Time Complexity

O(D + L log n)

- D is the (unweighted) diameter of the graph, and
- L = max over all vertices(diameter around a vertex where diameter is only considered for the nodes which lie in its neighborhood) - local shortest path diameter

Message Complexity

O(|E| log L log n)

- |E| is the number of edges

How to run

```
Compile:
      $ (sudo) make

Generate input:
      $./gen_graph > input.txt
       N e

Run:
      $ mpirun -n N ./amst < input.txt,
      or
      $ mpirun -n N -hosts master ./amst < input.txt
      or
      $ mpirun -n N -hosts client,master ./amst < input.txt

      where N is the number of vertices.
```

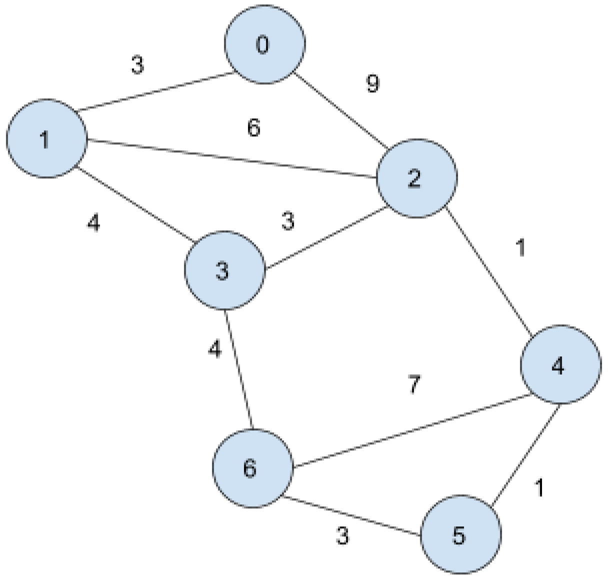
client and *master* need to be specified in /etc/hosts and nfs-server (on master) and nfs-common (on clients) services should be installed.
A common directory should be configured for use by all the machines, which the clients are mounted to.

```
sudo mount -t master:/home/dsproject/cloud ~/cloud
```

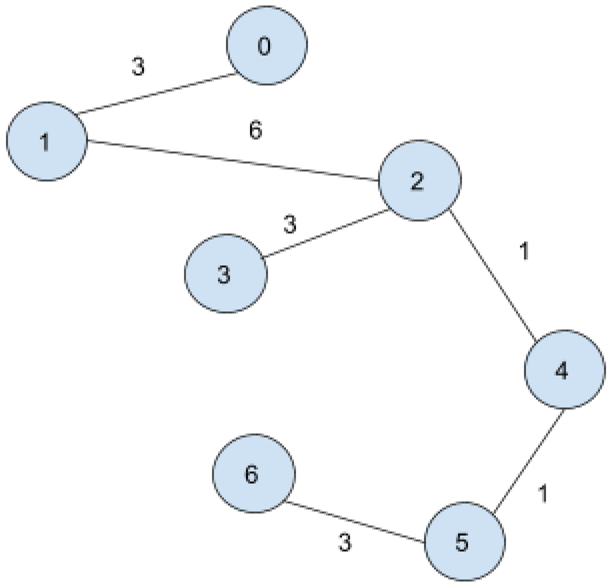
We used the accounts on each machine with same username *dsproject* for simplicity and *cloud* is the common folder at every machine's */home/dsproject/*.

Results

Sample Graph:

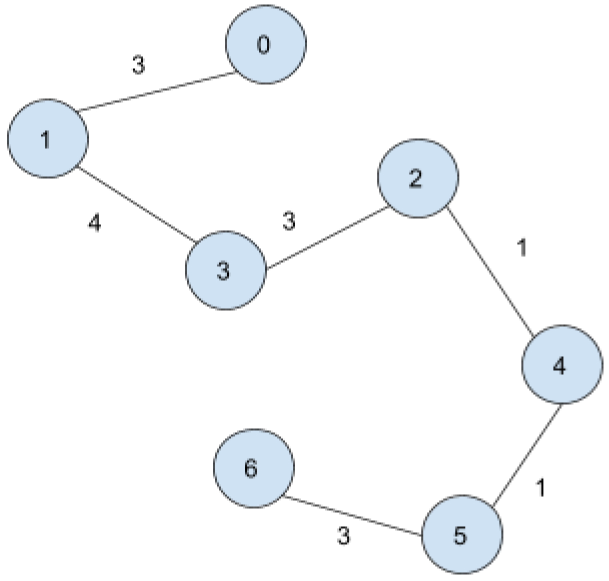


Approximate MST



Total weight = 17

Exact MST



Total weight = 15

Comparisons between approximate MST and exact MST

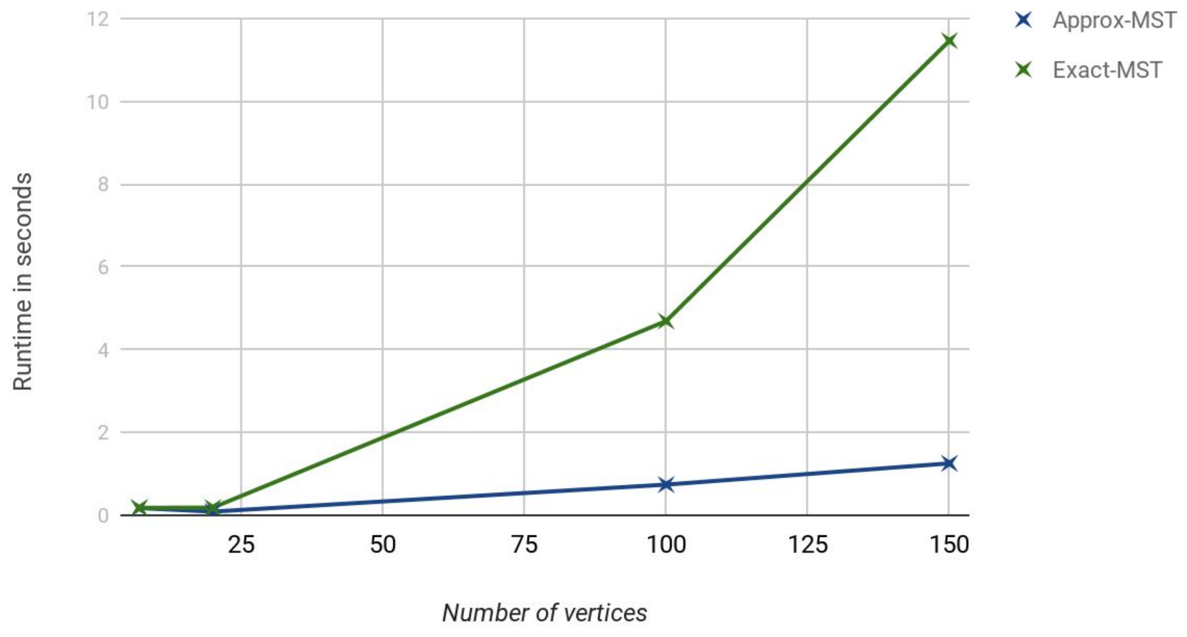
Table 1: Using one CPU

(V, E)	Approx-MST (run time)	Exact-MST (run time)	Approx-MST (total weight)	Exact-MST (total weight)
(7, 10)	0.175 s	0.181 s	17	15
(20, 100)	0.089 s	0.185 s	34	30
(100, 500)	0.741 s	4.692 s	194	169
(150, 800)	1.254 s	11.463 s	275	241
(200, 7500)	mpi stdin too slow	mpi stdin too slow	--	--
(300, 5000)	pipe error	pipe error	--	--

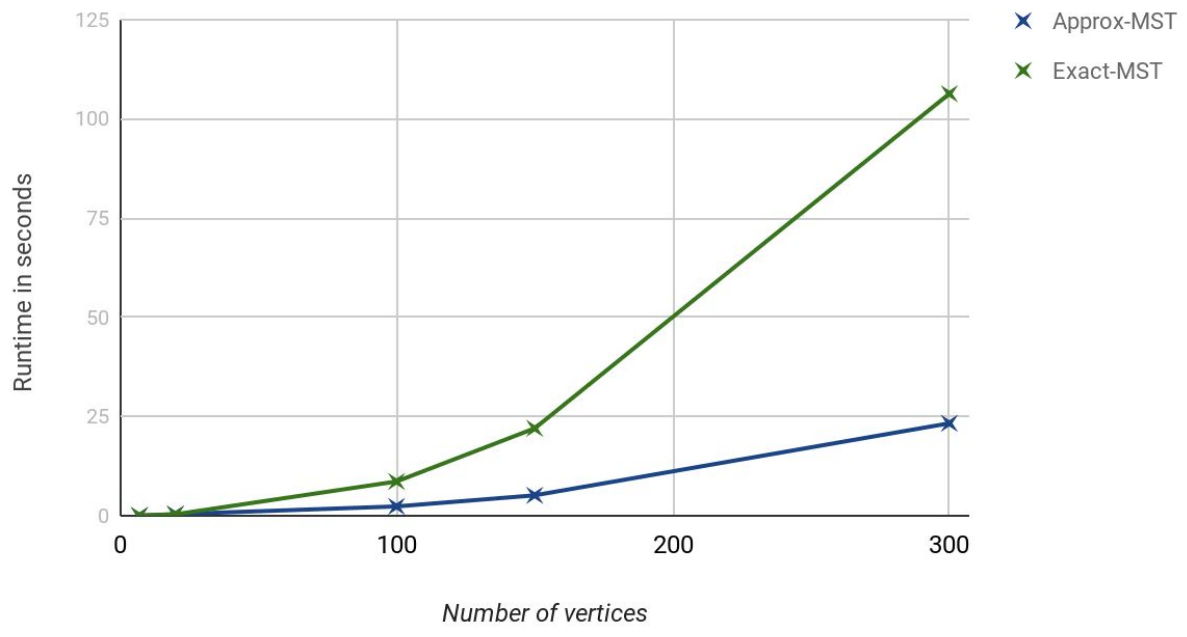
Table 2: Using two CPUs

(V, E)	Approx-MST (run time)	Exact-MST (run time)	Approx-MST (total weight)	Exact-MST (total weight)
(7, 10)	0.034 s	0.041 s	17	15
(20, 100)	0.398 s	0.363 s	37	30
(100, 500)	2.347 s	8.624 s	195	169
(150, 800)	5.142 s	22.020 s	270	241
(200, 7500)	mpi stdin too slow	mpi stdin too slow	--	--
(300, 5000)	23.292 s	106.323 s	344	309

Approx vs Exact MST using 1 CPU



Approx vs Exact MST using 2 CPUs


Approx-MST
Exact-MST
Runtime in seconds
Number of vertices