MST

Ababacar and Pegah

May 2021

Contents

1 Introduction

2 Task 1

2.1 Prim algorithm

We suppose that we are given a weighted undirected graph.
Data structures we use are:

1. parent array which holds the **possible** parent of each node in the MST. Initially the parent of every node is set to -1.

2. weights array which indicates the weight of the optimal edge that connects the node to the MST.

3. Set array which is a bool array and is true in the ith place is the node is already in MST.

4. to storage the graph we use an adjacency matrix.

In this algorithm we start from a node, by default the node zero, and we add it to the MST tree ($Set[0] = 0$). Now we should update the weight array for all of the nodes connected to this first node, the node zero. This means that we change the value of $weight[u]$ if u is connected to the node zero.
In each of the following steps we add the cheapest node (that is not already in MST) to our MST by doing the following process:

1. initialize two parameters: min and min-index to INF(infinity as you can define in your program to be a great value) and -1

2. iterate over the array weight, for every node v that is not in the MST, if its corresponding weight is smaller than min, then replace min by $weight[v]$ and min-index by v.

3. after iterating over all of the nodes, we now have the optimal node to add to MST and we return the min-index.

4. we repeat this process until having all of the nodes in MST.

The pseudo code of this algorithm is:

Algorithm 1: Prim

```
 Input: a general graph
 Output : minimum spanning tree initialization;
for $i = 1, 2, \ldots, n$ do
    int u = minConnecter(n,weights,set);
    for v neighbor of u and v not in MST do
        update weights and parent of u;
    end
end
```

Where minConnector function finds the optimal node to add to MST.

Time complexity
$Time complexity of this algorithm is$O(n^2) where n is the number of nodes in the graph, Because we iterate n times (if we add the first node initially, $n-1$ times) and each iteration, as we must find the optimal node to add, it takes $O(n)$ to find such a node.

2.2 Boruvka's algorithm

We suppose in this algorithm that we are given a connected, undirected and weighted graph.
Data structures are:

1. subset is defined as a struct containing the integers parent and rank.

2. subsets is an array of type subset, it indicates the subset that each node belongs to and also its rank.

3. cheapest is an integer array of size of number of nodes. $cheapest[v]$ indicates the index of the optimal edge from this node to outside of the set it belongs to.

4. Edge is a struct containing three integers : start,end and weight.

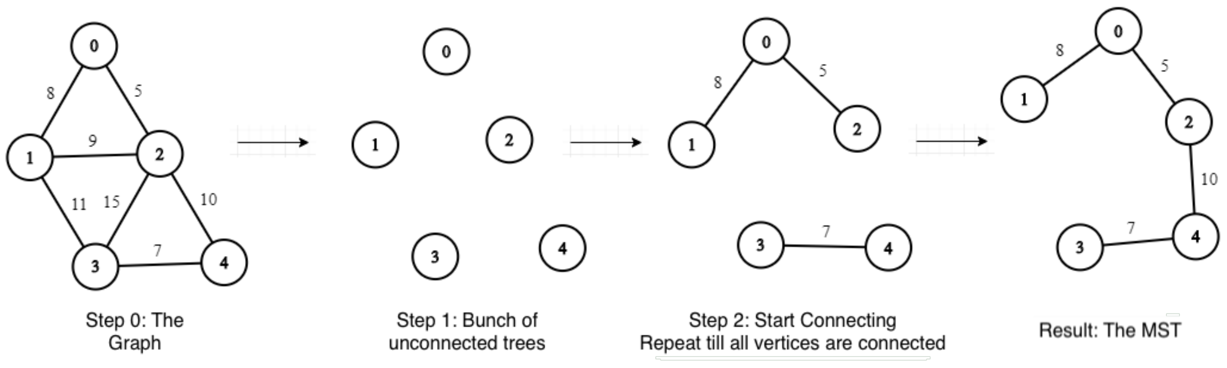
5. Edge is also an array of type Edge, which is itself a member of the struct Graph.

6. Graph is a struct containing an array of type Edge,number of nodes and number of edges.

In this algorithm we first initialize all the data structures. Then as long as the number of existing sets is more than one, we repeat the following process:

1. we set the cheapest corresponding to each node to -1.
2. we go through all of the edges to see if they are the optimal edge for one of the sets. If the edge e is optimal for a set, then we set the *cheapest* of the representer of this set to e.
3. now we should add these optimal edges to our graph. In order to do this we go through all of the nodes, for a node v there are two cases:
   (a) if the $cheapest[v]$ is -1, we don't further consider this node.
   (b) if the $cheapest[v]$ is not -1 and the edge does not connect two already connected sets, we add the $cheapest[v]$ to our MST and we connect two sets that this edge connects.

In the following schema you can see how this algorithm works step by step on a given graph.


0
1
2
3
4
8
5
9
11
15
10
7
Step 0: The Graph
Step 1: Bunch of unconnected trees
Step 2: Start Connecting Repeat till all vertices are connected
Result: The MST

The pseudo code of this algorithm is:

```
Algorithm 2: Boruvka's
Result: Write here the result
int numberOfSets = numberOfNodes;
int weightMst = 0;
while numberOfSets >1 do
    for $i = 1, 2, \ldots m$ in Edge array do
        int first = Edge[i].start;
        int second = Edge[i].end;
        int set1 = find(first,subsets);
        int set2 = find(second,subsets);
        if set1 == set2 then
            continue;
        else
            cheapest[set1] == -1 → cheapest[set1] = i;
            Edge[cheapest[set1]].weight > Edge[i].weight →
              cheapest[set1] = i;
            cheapest[set2] == -1 → cheapest[set2] = i;
            Edge[cheapest[set2]].weight > Edge[i].weight →
              cheapest[set2] = i;
        end
    end
    for $i = 1, 2, \ldots n$ in the Nodes do
        if cheapest[i] != -1 then
            int set1 = find(subsets,Edge[cheapest[i]].start);
            int set2 = find(subsets,Edge[cheapest[i]].end);
            if set1 ==set2 then
                continue;
            else
                Union(subsets,set1,set2);
                weightMst += Edge[cheapest[i]].weight;
                numberOfNodes–;
            end
        end
    end
end
```

Time complexity

Time complexity of this algorithm is $O(m * log(n))$: each contraction takes O(m) work. Each round at least halves the number of existing sets, because we an edge for each set and each edge connects at most two sets. So, there are $O(log(n))$contractions, and each round has $O(m)$ work, with $O(m * log(n))$ work in total.

Comparing two algorithms:
[testing two programs on a complete graph of size n and the same weight for each edge]
Time is in micro seconds.

n	Prim	Boruvka's
3	4	28
4	6	42

3 Task 2

3.1 Kruskal

In this task we implement the Kruskal algorithm. Important data structures are:

1. parents : an integer array of size n (number of nodes), which holds the parent of each node in the MST. parent of each node is initially itself.
2. Edge is a struct containing three integers corresponding to : start, end and the weight of an edge.
3. Edge is also an array of edges, of size m (number of edges).
4. Graph is a struct containing an array of type Edge,number of nodes and number of edges.

In this algorithm we first store all of the edges in an array of type Edge, then ,by defining a comparing function, we sort these edges by their weights.
Having the sorted array of edges, we start to iterate over this array from the beginning, and we add an edge to MST if and only if it does not make a cycle.

1. Sort the edges of the graph by their weights.
2. iterate over the sorted array and add an edge to MST if it does not make a cycle in MST.

The pseudo code of this algorithm is:

Algorithm 3: Kruskal

```
Result: Minimum spanning tree of a general graph
sort(edges);
int numberOfSets = numberOfNodes;
int iterate = 0;
int weightOfMST = 0;
while numberOfSets > 1 do
    int start = edges[iterate].start;
    int end = edges[iterate].end;
    int set1 = find(start,parents);
    int set2 = find(end,parents);
    if set1 == set2 then
        continue;
    else
        Union(set1,set2,parents);
        numberOfSets–;
        weightOfMST += edges[iterate].weight;
    end
    iterate++;
end
```

Where the functions find and Union are:

Algorithm 4: find

```
 Input: a node v
 Output : representer of the set v belongs to
if parents[v] == v then
    Return v;
else
    Return (parents[v] = find(parents[v]));
end
```

Algorithm 5: Union

```
 Input: v and u as the representers of two sets
parents[u] = v;
```

Time complexity

The most time consuming operation in Kruskal algorithm is Sort and it takes $O(m * log(n))$.

Performance

Time is measured in micro seconds, on a complete graph of size n and the same weight for each edge.

n	Prim	Boruvka's	Kruskal
3	4	28	43
4	6	42	46

4 Task 3

4.1 MPI

In this task we are supposed to implement parallel versions of Prim and boruvka's algorithms by using MPI. MPI is a directory of C++ programs which illustrate the use of the Message Passing Interface for parallel programming. We first start by Prim parallel algorithm.
Here you can find a list of main MPI commands that we have used in the implementation of these two parallel algorithms, however this is just a summary of descriptions in documentation :
MPI_Init: must be called before most other MPI routines are called,accepts the C/C++ argc and argv arguments to main, but neither modifies, interprets, nor distributes them.
MPI_Comm_size:This function indicates the number of processes involved in a communicator.
MPI_Comm_rank:Determines the rank of the calling process in the communicator, this will be used to convert the number of a local node to global and inverse.
MPI_Type_contiguousMPI_Type_commit: These two functions are used to define a new type in MPI. Why do we need to define a new type and we can not use the existing types? Because we further we will need to communicate this type between different processes and hence this need to be defined in the MPI structure.
MPI_Op_createMPI_Reduce: The first function binds a user-defined global operation to an op handle that can be used for example in MPI_Reduce. MPI_Reduce combines the elements provided in the input buffer of each process in the group, using the operation op, and returns the combined value in the output buffer of the process with rank root.
MPI_Bcast:Broadcasts a message from the process with rank root to all other processes of the group.

4.2 Prim's parallel algorithm

We have already discussed the implementation of Prim's algorithm in the previous sections but what we are interested in here is to partition the global task into small units and make these units work at the same time, but they can also interact with each other.
Partitioning idea is not always possible for algorithm depending on how their different parts and tasks interact with each other but in this case we can think of the following parallelization:[have in mind the function of "weights" array in normal Prim's algorithm]

1. We first partition the adjacency matrix into d parts, d being the number of processors.
2. In each iteration, each processor finds its best candidate to be added to the MST (obviously this candidate node is not already in the global MST).
3. Then we perform a reduction which leads to choosing the best global candidate to be added to MST.
4. We add this global best node to MST and also we broadcast this node to every processor.
5. every processor updates its part of the "weights" array by knowing the added node to MST.

Or equivalently the following algorithm:

We have to add that this implementation is not really optimal in the sense that parallelization can not be implemented efficiently, Because two selections can not be performed independently, Since there is an update phase after each selection in each of the processors.

4.3 Boruvka's parallel algorithm

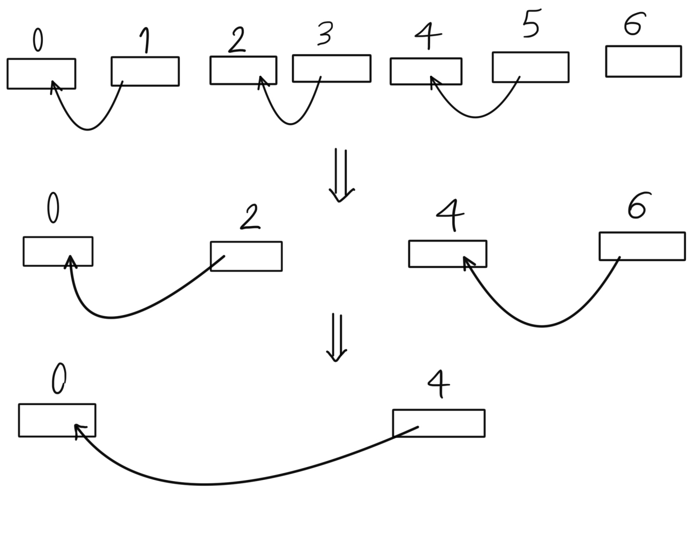
The important MPI functions we use in this implementation are MPI_Bcast ,MPI_send ,MPI_recv and MPI_Scatter.
We have already had MPI_Bcast ,MPI_send and MPI_recv in the previous tasks, but MPI_Scatter is sends data from one task to all tasks in a group.

Algorithm 6: Parallel Prim

```
Input: An undirected weighted connected graph
Output: MST
Partition adjacency matrix to d parts;
$V_i$ = nodes in the $i$th partition. IN EACH PROCESSOR i:
$v_i$ = optimal node to add to MST in $V_i$;
compare the optimality of $v_i$ to other $v_j$;
Receive the optimal node $v_k$;
if $v_k \in V_i$ then
    for $v$ neighbor of $v_k$ in $V_i$ do
        if weights[v] > adj[v][$v_k$] then
            weights[v] = adj[v][$v_k$] end
        end
    end
```

To have a closer look at its arguments : MPI_SCATTER(SENDBUF, SEND-COUNT, SENDTYPE, RECVBUF,RECVCOUNT,RECVTYPE, ROOT, COMM, IERROR)

In this algorithm we first scatter the edges between different (but we have all the vertices in each processor). Then at each step we merge the best choices of two processors. To have a quick illustration let's have a look at the following schema: In the first step we merge the information of odd nodes



with even nodes, then in the second step the information in $4k + 2$ nodes is merged with $4k$ nodes. This way the best set of edges would be gathered in

the root after $log(S)$ steps, where S is the number of processes.
Now to have a more formal look at the algorithm:

1. Scatter the edges in different processes.
2. for each process, iterate over all the edges associated with this process and perform the Boruvka's algorithm, so you would have the best edge to connect to each node in each process at the end of this step.
3. For each process if 2$*$step divides its rank, then receive the best edges of the process rank+step, compare them with the edges in the process, keep the optimal edges. And if 2$*$step does not divide the rank of the process but step divides the rank of the process, the send the best edges of this process to the process rank-step.
4. Continue this exchange of edges until all gathering all the edges of the MST in the root.

5 Task 4

6 Task 5

In this section we need to implement k clusters by using the kruskal algorithm and then removing the k edges with the largest weights. Now we know that this algorithm chooses the edges in ascending order, so this means stopping the algorithm as soon as we have |V| - k edges instead of |V| - 1.

Algorithm 7: MST-Kclustering

```
Input: An undirected connected graph G = (V, E), k the number of clusters
Output: An undirected graph G=(V, F) with k clusters
Sort E by increasing weight
Let $e_1, \ldots, e_m$ be the result
F = $\emptyset$
i := 1
while $F$ contains less than $|V| - k$ edges do
    if $(V, F \cup \{e_i\})$ contains no cycle then
        $F := F \cup \{e_i\}$
    end
    $i := i + 1$
end
```