

Dynamic Algorithms

Spring 2008

Project Report: Dynamic Minimum Spanning Tree

Pooya Davoodi

May 23, 2008

Contents

1	Introduction	1
2	Offline Method	1
3	Online Method	3
3.1	First Approach	4
3.1.1	Init Operation	5
3.1.2	Change Operation	6
3.2	Second Approach	8
3.2.1	Init Operation	8
3.2.2	Change Operation	10
4	Implementation	10
4.1	Efficiency	10
5	Conclusion	11

1 Introduction

Finding a minimum-spanning-tree in a given graph with n vertices and m edges is one of the popular problems in computer science and it can be solved in $O(m \log n)$ by some algorithms like Kruskal that are called offline algorithms. But the problem that is noticed in this project is to find a minimum-spanning-tree dynamically. Means that if the weights of the edges of the graph change consecutively how the minimum-spanning-tree can be found after each change without seeking for it from scratch. A data structure should be created for storing some useful properties of the previous graph and its minimum-spanning-tree and then that data structure should be used for making the new graph and its minimum-spanning-tree. This algorithm is called online algorithm. The exact definition of the problem is:

- **Input:** A sequence of $\binom{n}{2}$ triples (u, v, w) that specifies a complete graph G with n vertices $0, \dots, n - 1$ and all its weighted edges. Also some other triples like (i, j, w) that specify changes in weight of the edge (i, j) to w .
- **Output:** The weight of a minimum-spanning-tree of the G after each change operation or after a block of change operation.

The goal of this project is to design two different methods, offline and online algorithms to solve this problem, analyse and implement both of them and then compare their running time. Each of the two methods have three functions as follows:

- **Init(n, edges).** Storing the properties of G in data structure and seeking for its minimum-spanning-tree.
- **Change(i, j, w).** Changing the weight of the edge (i, j) from its previous weight to w and then updating the whole data structure if it's necessary.
- **Query().** Returning the total weight of a minimum-spanning-tree of G .

Obviously it is expected that the running time of the online algorithm is better than offline one. Because the offline method seek for the minimum-spanning-tree from scratch based on a popular algorithm but the online one uses the previous states of the graph and its minimum-spanning-tree and doesn't start the search from scratch.

2 Offline Method

The Kruskal algorithm is used in this method to find the minimum-spanning-tree. An array of triples namely *Edge* is used for storing the edges of the graph. Each *Edge* has three parts that are integers for storing two vertices and weight of the edge. The Kruskal algorithm that has been used here, utilizes a priority queue to get the minimum weight edge of the graph in each iteration. The data structure that has been used for priority

queue is an array of integers to store the complete binary tree or MinHeap. Extracting (finding and deleting) the minimum value from the heap and adding a new value to it, each one take $O(\log m)$ time. But the required time for making a heap from scratch for m values, is $O(m \log m)$ [?].

Also to make sure that the selected edge in each iteration doesn't result a cycle, a type of set representation has been used that it takes $O(1)$ for union of two sets and takes $O(\alpha(n))$ to find out the set of a member if there are n members in that set [?]. The pseudocode of the three functions of the offline algorithm is as follows:

```

Init(n,edges){
    w_mst=Kruskal(n,edges)
}

Change(i,j,w){
    edge(i,j).w=w
    Kruskal(n,edges)
}

Query(){
    return w_mst
}

Kruskal(n,edges){
1   store all the edges in MinHeap
2   for i=0 to n-1
3       store i in a new set in SETS
4   w_mst=0
5   for i=0 to n-1{
6       E=MinHeap.DeleteMin()
7       if(SETS.CollapsingFind(E.u) not equal SETS.CollapsingFind(E.v)){
8           SETS.WeightedUnion(E.u,E.v)
9           w_mst=w_mst+E.w
10          i=i+1
11      }
12  }
}

```

Analysis The running time of Init and Change functions are as same as Kruskal function and the Query takes $O(1)$ time. Line 1 in Kruskal function takes $O(m \log m)$ time for making a heap with m nodes. The for loop in lines 2-3 makes n sets that the i th set has only the number i and it takes $O(n)$. The for loop in lines 5-11 finds the minimum edge of the graph and checks if it doesn't result a cycle, its weight is added to the total weight of minimum-spanning-tree and then the two sets of two vertices of that edge, are

joined to make union. The time of line 6 is $O(\log m)$ and the time of line 7 for doing the CollapsingFind is $\alpha(n)$ [?] and line 8 is done in constant time. Therefore the total amount of time of Kruskal function is $O(m \log m) = O(m \log n^2) = O(m \log n) = O(n^2 \log n)$. Since that the Init and Change operation can be carried out in $O(n^2 \log n)$.

3 Online Method

For this method a technique namely Sparsification has been used. In this technique, a balanced binary tree is used for maintaining some information that is required for calculating the minimum-spanning-tree after each change operation. Indeed every node of this tree maintains some of the edges of the graph and doesn't have any intersection in edges with its siblings. Every node gets its edges from its two children and make a minimum-spanning-forest by those edges and take the other edges away and at the end the root of the tree has the minimum-spanning-tree.

I tried to make this tree in a way that it is a complete binary tree always. We know that a complete binary tree has many advantages such that it is balanced, low cost in memory and also very easy to seek for the nodes from their children or parents. It is selected at first, how many leaves are needed for the tree and then a tree with that property is made.

Lemma 1. *There is at least one complete binary tree with m leaves and no $1 - degree$ nodes.*

Proof. It can be proved by induction. It is true for the tree with one node obviously. We suppose that the lemma is true for the tree T with $m - 1$ leaves. Means that we have a complete binary tree with $m - 1$ leaves and no $1 - degree$ nodes. We should prove that we can make a complete binary tree with m leaves and no $1 - degree$ nodes from T . If we add a new node to T , it will be inserted at the last position of T to maintain it complete. T did not have any $1 - degree$ node, therefore the new node will be the child of a node that was leaf before and since that the number of leaves will not increase but there is one $1 - degree$ node now. If we add another new node to T , it will be inserted beside the previous inserted node as its sibling. Therefore the node that was $1 - degree$ is a $2 - degree$ node and the number of leaves is m .

Although it can be proved that there is exactly one such complete binary tree, but we do not need it here. \square

Also the number of $2 - degree$ nodes in the tree will be one less than the number of leaves. Because when 2 leaves are added to the tree, one of the previous leaves converted to a $2 - degree$ node. Therefore for a tree with m leaves the total number of nodes will be $2m - 1$.

So, the complete binary tree that will be made for sparsification has no $1 - degree$ node. Two approaches have been used in this project for making and using this tree that the second one is better in running time. In the first one every edge of the graph, stored in one leaf of the tree and therefore the number of leaves will be $\frac{n(n-1)}{2}$ and its total number

of nodes will be $n(n-1) - 1 = O(n^2)$. But in the second approach, q edges (more than one) are placed in each leaf of the tree. Indeed if the number of nodes n is even, $k = \frac{n}{2}$ and $m = \frac{n(n-1)}{2} = q(n-1)$ and k edges are placed in each of the $n-1$ leaves. And if n is odd, then $n-1$ is even and $q = \frac{n-1}{2}$, and then $m = \frac{n(n-1)}{2} = qn$ and q edges are placed in each of the n leaves. In the second approach the total number of nodes in the tree is $2 * q - 1 = O(n)$.

The data structure that is used for the tree is an array that every element of this array is a pointer to a node namely SparseNode. Each node that should maintain a minimum-spanning-forest has three parts. One part is for number of edges that are maintained in that node and one part is for the total weight of the minimum-spanning-forest of that node. Also the last part of the node is an array of integers that every two of its elements are for one of the edges of the node. Indeed the minimum-spanning-forest of each node is stored in this array in sorted order into the weight of the edges. For example if a node has three edges (1, 2, 3), (1, 3, 7) and (0, 4, 2), total weight of the minimum-spanning-forest will be 12 and the array of its minimum-spanning-forest will have six elements in this order: (0, 4, 1, 2, 1, 3).

```
SparseNode{
    m : number of edges
    w_mst : weight of the minimum-spanning-forest
    E : list of the edges of the minimum-spanning-forest (array of 2*m integers)
}
```

In Init operation the tree is made bottom-up, means that the leaves of the tree are made at first, and then the above nodes get the required information from their children and store the needed values in their data structure up to the root. Also in each Change operation, the change in weight of an edge and in minimum-spanning-forests are started from one of the leaves that maintains that changed edge and then it continues in the path from that leaf to the root. After both of the Init and Change operation, the weight that is stored in the root of the tree, is the correct weight of the minimum-spanning-tree of the graph and the Query operation can be report this weight in constant time as follows:

```
Query(){
    return weight from the root
}
```

Both of the approaches that have been used, get the adjacency matrix as input for the edges of the graph, because the time to access the weight of an edge from its two vertices is the best by using this matrix.

3.1 First Approach

In this approach as has been said before, number of leaves of the tree are $\frac{n(n-1)}{2}$ and number of nodes of the tree are $n(n-1) - 1$.

3.1.1 Init Operation

Init operation in this approach makes the sparsification tree based on the weights of edges of the graph that are in *Adj* as adjacency matrix of the graph. ST in this algorithm is the array that has been used for the sparsification tree and each of its elements is a pointer to a SparseNode. The first nested loop puts each edge in one leaf (ST[k]) from the last leaf to the first one and the second nested loop that has three while loops inside a for loop, considers all the $2 - degree$ nodes of the tree and for each node (ST[k]), merges the sorted array of edges from each of its two children into the array of ST[k]. It is a familiar merge algorithm for merging two sorted array in to a new one.

This function uses the set data structure for recognising the cycle in the graph. The technique that has been used is the SimpleFind to find out that what is the set No. of a vertex. If the set No. of two end vertices of an edge are not in a single set, means that there is not any path between them and that edge can be added to the minimum-spanning-forest. Also adding an edge in a minimum-spanning-forest of a node, as well as putting it at the end of the array, makes the union of two sets of vertices of the edge. This operation is done by WeightedUnion that makes the result union set smaller in depth by connecting the smaller set to the the root of larger set.

```

Init(n,Adj){
  m=n*(n-1)/2
  make array ST of size 2*m-1
  k=2*m-2
  for i from 0 to n-1{
    for j from i+1 to n-1{
      put Adj[i][j] in ST[k]
      k=k-1
    }
  }
  for k from m-2 down to 0{
    while there is at least one edge in ST[2*k+1] and one edge in ST[2*k+2]
      that not added yet And another edge can be put to ST[k]{
      e1=minimum edge in ST[2*k+1]
      e2=minimum edge in ST[2*k+2]
      e=lower weight edge between e1 and e2
      if putting e in ST[k] does not make a cycle
        put e at the end of ST[k]
      }
    while there is at least one edge in ST[2*k+1] that not added yet
      And another edge can be put to ST[k]{
      e=minimum edge in ST[2*k+1]
      if putting e in ST[k] does not make a cycle
        put e at the end of ST[k]
      }
  }
}

```

```

    }
    while there is at least one edge in ST[2*k+2] that not added yet
      And another edge can be put to ST[k]{
        e=minimum edge in ST[2*k+2]
        if putting e in ST[k] does not make a cycle
          put e at the end of ST[k]
      }
    }
  }
}

```

Analysis The first nested loops takes $O(n^2)$ time obviously. The outer for loop in second nested loop iterates $m = O(n^2)$ times and in each iteration it merges two arrays. Merging operation that is done in while loops has different number of iterations in various running. The number of iterations for these while loops equals the addition of length of two merged arrays in worst case and this one equals $O(n^*)$ when n^* is the number of vertices in minimum-spanning-forest of that node. Also determining the cycle takes $O(\log n)$ for running the SimpleFind function[?]. Putting the edge in the new array and making the union set by running the WeightedUnion function takes constant time[?]. Therefore the total running time of the Init operation is $O(n^2 \log^2 n)$ (Though the proof is not complete).

3.1.2 Change Operation

If the weight of an edge e changes, there will be four cases. Whether the new weight w is greater than the previous weight w' or not and whether e is in the minimum-spanning-forest msf or not (every minimum-spanning-forest in the sparsification tree). Based on these two states, there are four cases.

- If e is in msf and w is less than w' , then no change will be occurred in edges of msf and only the weight of msf decreased by $(w' - w)$.
- If e is in msf and w is greater than w' , it is possible that e replaced with another edge. Indeed if e is removed from the msf , then there will be at least two components in the remained msf and the edge with minimum weight that is not in current msf and its weight is between w and w' and does not result a cycle, should be selected to replace e . Checking out the existence of cycle is done by set representation like Init operation. Means that if the two vertices of an edge are in the same set, there is path between them and adding that edge results a cycle.
- If e is not in msf and w is less than w' , it is possible that an edge in msf replaced with e . If e is between vertices i and j , the reason that e had not been selected in previous msf was that it had caused a cycle. Indeed there is a path from i to j now in current msf . Now that w is less than w' , and it is possible that e selected sooner than another selected edge and this edge could not be selected anymore. If e is between vertices i and j , it will suffice to search and remove an edge that is in the

path from i to j and its weight is between w and w' and has the maximum weight among such edges and then place e instead of it in msf . This searching task can be done with a special DFS search.

- If e is not in msf and w is greater than w' , it is obvious that e wont selected again in msf .

```

Change(i,j,w){
1  old_w=previous weight of (i,j)
2  k=index of the node that has edge (i,j)
3  change the weight of edge in ST[k]
4  k=(k-1)/2
5  while k>=0{
6      if (i,j) is in ST[k]{
7          p=position of (i,j) in ST[k]
8          if w is greater than old_w{
9              remove (i,j) tentatively from ST[k]
10             find edge (u,v) from two children of ST[k]
11                 that does not make a cycle in ST[k] and
12                 its weight is between w and old_w and
13                 (u,v) is minimum among all such edges
14             if (u,v) exists put it in ST[k] instead of (i,j)
15             else maintain the (i,j) in ST[k]
16             change the w_mst of ST[k]
17         }
18         else
19             reduce (old_w-w) from w_mst of ST[k]
20     }
21     else{
22         if w is less than old_w{
23             Find maximum weight edge (u,v) in ST[k] by DFS such that:
24                 -(u,v) is in the path from i to j in ST[k]
25                 -weight of (u,v) is between w and old_w
26             if (u,v) exists then replace it with (i,j) and
27                 reduce (weight of (u,v)-w) from w_mst of ST[k]
28         }
29     }
30     k=(k-1)/2
31 }
}

```

Analysis As was said before, the number of nodes in the sparsification tree in the first approach is $O(n^2)$ and because the tree is complete, its depth is $O(\log n)$. Then the while

loop in line 5, iterates $O(\log n)$ times. In line 6-7, existence of (i, j) in $ST[k]$ and finding its position can be determined in $O(n)$ to search through the array of edges in each node. We know that stored edges in each node are the edges that belong to minimum-spanning-forest of that node and equals to the number of vertices. Removing (i, j) from $ST[k]$ in line 9, is only a comment in the psuedocode and indeed it is done in line 14. The most time consuming lines of the algorithm is lines 10-13 that searches through the list of edges in two children of each node. The searching takes $O(n)$ time because the number of nodes in each of the children is less than the number of nodes in the parent. But in each iteration of searching of lines 10-13, it should check out the selected edge does not make a cycle and it takes $O(\log n)$. Therefore the running time of lines 10-13 is $O(n \log n)$. The replacing tasks 14 and 26-27 are done in $O(n)$ because they should search in the array of edges and shift some of its elements. DFS in line 23-25 takes $O(n)$ to search through all the edges of $ST[k]$. All the other lines until 23 are $O(1)$. Consequently running time of the Change operation is $O(n \log^2 n)$.

3.2 Second Approach

The goal of the second approach is to improve the running time of the first approach. For doing this, two things have been changed in this algorithm. First is in data structure of sparsification tree. In the previous approach, every edge of the graph was stored in a single leaf of the tree at first. In this approach q edges are placed in each leaf of the graph. Means that the tree will be smaller and the number of nodes of the tree depeneds on q . If the number of nodes n is even, $q = \frac{n}{2}$ and $m = \frac{n(n-1)}{2} = q(n-1)$ and q edges are placed in each of the $n-1$ leaves. And if n is odd, then $n-1$ is even and $q = \frac{n-1}{2}$, and then $m = \frac{n(n-1)}{2} = qn$ and q edges are placed in each of the n leaves. In this approach the total number of nodes in the tree is $2 * q - 1 = O(n)$ in comparison with $O(n^2)$ in the first approach. Thus initializing the nodes of the tree is better in running time.

The second improvement in this approach is making the set data structure and its SimpleFind function better. There is another technique namely CollapsingFind that collapses every set and makes it less in depth during searching for a member. The running time of SimpleFind that was considered in the first approach is $O(\log n)$ and the running time of CollapsingFind is $O(\alpha(n))$ that is the reverse function of Ackermans function and is very low growth and its value for practical purposes is 4 [?].

3.2.1 Init Operation

As it can be seen in the following Init function, the only difference between Init of first approach and second approach is the number of leaves and placing the edges in leaves. There is a for loop that stores $\frac{m}{q}$ edges in a leaf in each iterations. But it stores them in increasing order, means that every $\frac{m}{q}$ edges should be sorted at first and then placed in the $ST[k]$.

```
Init(n,Adj){
```

```

if n is even q=n/2
else q=(n-1)/2
m=n*(n-1)/2
make array ST of size 2*q-1
k=2*q-2
for k from 2*q-2 down to q-1{
    put m/q number of edges from Adj to ST[k] in increasing order
}
for k from q-2 down to 0{
    while there is at least one edge in ST[2*k+1] and one edge in ST[2*k+2]
        that not added yet And another edge can be put to ST[k]{
            e1=minimum edge in ST[2*k+1]
            e2=minimum edge in ST[2*k+2]
            e=lower weight edge between e1 and e2
            if putting e in ST[k] does not make a cycle
                put e at the end of ST[k]
        }
    while there is at least one edge in ST[2*k+1] that not added yet
        And another edge can be put to ST[k]{
            e=minimum edge in ST[2*k+1]
            if putting e in ST[k] does not make a cycle
                put e at the end of ST[k]
        }
    while there is at least one edge in ST[2*k+2] that not added yet
        And another edge can be put to ST[k]{
            e=minimum edge in ST[2*k+2]
            if putting e in ST[k] does not make a cycle
                put e at the end of ST[k]
        }
    }
}
}

```

Analysis The first for loop has $O(n)$ iterations and in each iteration it sorts $\frac{m}{q} = O(n)$ edges in $O(n \log n)$. Thus it takes $O(n^2 \log n)$. The second for that is a nested loop is the same as Init operation in the first approach with two differences. First is that here the CollapsingFind function is used instead of SimpleFind and its time is $O(\alpha(n))$ instead of $O(\log n)$. The second difference is the number of iterations of for loop that here is $q = O(n)$. Therefor the running time of the second for is $O(n^2 \alpha(n))$ but because $O(n^2 \log n) > O(n^2 \alpha(n))$ the total running time of the Init operation is $O(n^2 \log n)$.

3.2.2 Change Operation

The Change operation of second approach is the same as first approach with two differences. The first difference is in line 3 of that function. It should place the changed edge in correct position as well as changing the weight. It is done by a shifting method in array and takes $O(\frac{m}{q} = O(n))$. The second difference is the using of CollapsingFind instead of SimpleFind again. Therefore the total running time of the algorithm is $n \log n \alpha(n)$.

4 Implementation

I implemented this algorithm with g++ in Fedora and tested it with a complete graph with 1000 vertices that is available in <http://www.daimi.au.dk/~gudmund/dynamicF08/mst/>. Also I experimented with 1000000 change operations that is available in the same url. The method was to perform the experiments for the first 10^i changes only, where i grew by one $i = 0, 1, 2, \dots, 6$. After each 10^i changes a query is called to check the weight of minimum-spanning-tree after all changes. The running time of the Init operation in offline method and both approaches of online method was good as can be seen in fig. But the running time of change operation in offline method was very excessive and I stopped the experiments for $i > 4$ and it takes for $i = 4$ almost 9 hours. The running time of the first approach of online method for $i = 6$ was nearly 70 minutes that is so long. The comparison between the algorithms is shown in fig.

4.1 Efficiency

In this section some of the improvements in efficiency that are achieved in this project are explained.

- The length of array that has been used for set representation equals the maximum value in the set rather than the number of values. The running time of Find and Union operations was improved however space were lossed.
- Also array of numbers was used for storing the edges in each SparseNode. There are many other ways to store the edges but this one is the best because of the merging of two sorted lists that is used in Init and Change operations.
- In Init operation of online-second approach, that more than one edge are placed in each leaf, edges could be sorted at first and the all of them placed in leaves part by part. But here, each part was sorted separately and placed in leaf. This way is better because sorting many times in a very smaller list is better than one sorting in a large list.
- In DFS that has been used in Change operation of online method to find the edges that are in the path between u and v , a stack was used to push any edges that visited in DFS routing and popped when DFS comes back from a path and if the DFS that

starts from v , encounters u every where, it stops and reports the edges in the stack. It seems that it is also a very efficient way to find such edges.

- An adjacency matrix is stored in the program and all the edges that are maintained in sparsification tree, point to the matrix to find out the weights. This is an improvement in memory consumption.

5 Conclusion

The results of this project shows that there is a huge difference between running time of offline method and online method. If the the input data is masive and also changes dynamically, the best approach is to design new algorithms specifically for dynamic purposes. It was observed that the difference between $O(n^2 \log n)$ and $O(n \log^2 n)$ that were for offline method and online method consecutively, is very much.