I/O Efficient Algorithms for Computing Minimum Spanning Trees

A B. Tech Project Report Submitted
in Partial Fulfillment of the Requirements
for the Degree of

Bachelor of Technology

by

Abhinandan Nath
(09010101)

under the guidance of

Dr. Sajith Gopalan

to the

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY GUWAHATI
GUWAHATI - 781039, ASSAM
CERTIFICATE

This is to certify that the work contained in this thesis entitled “I/O Efficient Algorithms for Computing Minimum Spanning Trees” is a bonafide work of Abhinandan Nath (Roll No. 09010101), carried out in the Department of Computer Science and Engineering, Indian Institute of Technology Guwahati under my supervision and that it has not been submitted elsewhere for a degree.

Supervisor: Dr. Sajith Gopalan

Associate Professor,

May, 2013
Department of Computer Science & Engineering,

Guwahati. Indian Institute of Technology Guwahati, Assam.
I would like to take this opportunity to thank Professor Sajith Gopalan for guiding me in my B.Tech project, which in turn helped me get motivated about research. I would also like to thank Professor R. Inkulu for arousing my interest in the area of algorithms.
Contents

List of Figures vii
List of Tables/Algorithms ix

1 Introduction 1
  1.1 Massive Graphs .................................................. 2
  1.2 A Few Results on the External Memory Model .................. 3
  1.3 Organization of the Report ...................................... 3

2 Review of Prior Work 5
  2.1 The MST Problem : A Brief History ............................ 5
  2.2 The Boruvka Phase ............................................... 6
  2.3 The Current State of the Problem .............................. 7

3 A Friend, and Blind Alleys 9
  3.1 The Modified Prim’s Algorithm .................................. 9
  3.2 Failed Attempts .................................................. 10
    3.2.1 Attempt 1 .................................................... 10
    3.2.2 Attempt 2 .................................................... 11
    3.2.3 Attempts 3, 4 and 5 ........................................ 11
    3.2.4 Attempt 6 .................................................... 12
    3.2.5 Attempt 7 .................................................... 12
# Our Algorithm

## 4.1 The *Reduce Vertices* Procedure

### 4.1.1 Proof of correctness

### 4.1.2 Analysis

## 4.2 The Algorithm

### 4.2.1 Comparison with other Algorithms

# Conclusion and Future Work

# References
List of Figures

4.1 Relation between $\text{sort}(E)$ and number of calls to $\text{ReduceVertices}$ . . . . . 17
List of Tables/Algorithms

4.1.1 Reduce Vertices ($V, E, S$) .......................................................... 14
4.2.1 Compute MST ($V, E$) ................................................................. 18
Chapter 1

Introduction

A graph is a pair \((V, E)\) where \(V\) is a set of vertices and \(E\) is a set of edges. An edge \((u, v)\) is said to be incident on the vertices \(u\) and \(v\), and \(u\) and \(v\) are said to be neighbours of or adjacent to each other.

In an undirected graph, the edges are bidirectional. In a weighted graph, the edges have weights associated with them. A graph is connected if there exists a path between any 2 vertices. A tree is a minimal connected graph. There exists a unique path between any pair of vertices in a tree, thus precluding the presence of cycles. A subgraph of a graph \(G(V, E)\) is a graph \(G'(V', E')\) such that \(V' \subseteq V\) and \(E' \subseteq E\). A spanning subgraph is one that includes all the vertices of the original graph.

A minimum spanning tree (MST) is the lightest connected subgraph of a given graph, where weight of a graph is defined to be the sum of the weights of all the edges in the graph. Observe that it has to be a tree, and that it has to be one whose weight is the minimum among all possible spanning trees. Algorithms for computing MSTs have been studied for quite some time now. They have widespread applications. To motivate about the topic,
a simple example should suffice. Consider a network having multiple nodes, and they are connected by links. This setting can be modelled as a graph, in which nodes are vertices and links are edges. Now suppose we have to reinforce the network by laying reliable (and expensive) connections between existing links. We only want that the whole network is connected. In such a case, computing the MST of the network and using that to lay new connections is a plausible idea.

1.1 Massive Graphs

Today, as the world is getting deluged by data, it is not astonishing to see huge graphs. e.g., Modelling the Web as a graph, with webpages as nodes and links as edges, we get a graph which has billions of vertices (see [1]). Another example that can be thought of is analyzing terrain data obtained from GIS. In such a scenario, the entire graph can in no way fit in main memory.

The solution to the above problem is: store the graph in secondary memory (hard disks or tapes), and bring them into main memory as and when required (the figure below shows this memory hierarchy). This approach brings with itself a few problems of its own. Traditional algorithms always dealt with optimizing running time and space usage. With secondary memory coming to the picture, the main bottleneck is the I/O complexity of the algorithm, i.e., the number of I/O transactions done. To put things into perspective, compare the access times of secondary memory and main memory (10 ms for disk vs 100 ns for RAM, see [2]).

A brief description of an I/O transaction is in order here. In one I/O transaction, data are read from and written into the disk in units of blocks. A block contains a fixed number of contiguous bytes of data. Also, reading two contiguous blocks of data takes lesser time than reading two blocks which are apart. On an average, half an I/O is wasted in the latter case.
We use the following convention: the block size is denoted by $B$, and the size of the main memory is denoted by $M$. We sometimes abuse our notation, and use $V$ and $E$ to refer both to the set and the cardinality of vertices and edges respectively. Which one is meant should be clear from the context.

1.2 A Few Results on the External Memory Model

Given $N$ items stored contiguously in disk, the number of I/Os required to scan all the items is denoted by $\text{scan}(N)$.

$$\text{scan}(N) = O(N/B)$$

The I/O complexity of sorting $N$ contiguous items stored in disk is denoted by $\text{sort}(N)$.

$$\text{sort}(N) = O\left(\frac{N}{B} \log_{M/B}\left(\frac{N}{B}\right)\right)$$

The above bound is easily obtained by a multi way merge algorithm for sorting.

For all intents and purposes, $\text{scan}(N) < \text{sort}(N) << N$.

1.3 Organization of the Report

The report is organized in the following manner. Chapter 2 gives a brief survey of the problem at hand. In chapter 3, we present a few existing algorithms which provided the motivation for devising a new algorithm, and also a few false starts. In chapter 4, we describe our new algorithm, prove its correctness and compare it with the existing state of the art. In chapter 5 we conclude and pave the way for future work.
Chapter 2

Review of Prior Work

In this chapter, we present a brief review of the MST problem. [Graham and Hell] provides a detailed and excellent survey.

2.1 The MST Problem: A Brief History

As mentioned earlier, the MST problem has been extensively studied. Two very important properties are worth keeping in the back of our minds:

1. **MST Cut Property**: A cut is a partition of the vertex set $V$ into 2 sets $(S, V - S)$, where both $S$ and $V - S$ are non-empty. An edge is said to cross a cut $(S, V - S)$ if its one end point lies in $S$ and the other in $V - S$. The MST Cut property states that an edge belongs to the MST if and only if it is the lightest edge crossing any cut of the graph. For a simple proof, refer to [3].

2. **MST Cycle Property**: An edge does not belong to the MST if and only if it is the heaviest edge in some cycle of the graph. Refer to [3] for proof.

Using these two fundamental properties, numerous algorithms have been proposed. The first among them was the Boruvka’s algorithm ([4]), which used the MST cut property in maintaining a minimum spanning forest, finally culminating in an MST. Kruskal’s and
Prim’s algorithm ([5]) also used the cut and cycle properties. While in Kruskal’s, each edge is considered and a decision is taken whether to include or discard it, Prim’s incrementally grows a tree from a single vertex. All these algorithms can be implemented in $O(|E| \log |V|)$ time. Yao gave an $O(|E| \log \log |V|)$ time algorithm ([6]), which is similar to Boruvka’s, but considers the edges incident to each vertex in a partially sorted order. The Fredman-Tarjan algorithm ([7]) uses Fibonacci heaps to engineer a $O(|E| \log^* |V|)$ time algorithm. The best known algorithm is by Chazelle, which uses a new data structure called soft heap to create a $O(|E| \alpha(|V|, |E|))$ time algorithm ([8]), where $\alpha$ is the inverse Ackermann function. The Karger-Klein-Tarjan algorithm ([9]) is a linear time randomized algorithm, that uses ideas from the Boruvka’s algorithm.

All the above algorithms assume that the graph can fit in main memory. Directly using them for massive graphs results in very poor performance. e.g., directly using the Prim’s algorithm results in $\Omega(|V|)$ I/Os. Specialized schemes were developed for handling massive graphs. They are discussed later. For the moment, we mention a few prominent algorithms. A modified Prim’s algorithm was proposed in [10] that uses $O(|V| + \text{sort}(|E|))$ I/Os. Munagala ([11]) proved a lower bound of $\Omega(|E|/\text{sort}(|V|))$ I/Os. The currently best known algorithm is by Arge et al ([10]) and requires $O(\text{sort}(|E|) \log \log(|V|B/|E|))$ I/Os. For planar graphs too, a $O(\text{sort}(|E|))$ I/Os algorithm exists ([12]).

### 2.2 The Boruvka Phase

A popular and simple technique used to reduce the size of the input graph is the Boruvka phase. In one such phase, the lightest edge incident to each vertex is selected. This process is called Hooking. By the MST cut property, this edge belongs to the MST.

Once the edges are selected, the connected components induced by these edges are contracted into a single supervertex. Multiple edges between two supernodes are replaced
by the lightest among them. Internal edges are also removed. This gives us a new graph with atmost half the number of vertices in the previous graph.

Implementing this is simple. We assume that the graph is given in the form of an edge list. Each element is of the form \((u, v)\). An edge \((u, v)\) is thus present as \((v, u)\) as well. We call them the dual or twins of each other. We sort the edges, taking the key of an edge \((u, v)\) to be the ordered pair \((u, w(u, v))\), where \(w\) is the weight of the edge. Thus all the incident edges of a given vertex are together. Selecting the lightest edge incident to each vertex then becomes a matter of scanning the whole edge list, taking \(\text{scan}(|E|)\) I/Os. These selected edges are then stored contiguously. A representative vertex is now selected for each connected component induced by the selected edges. This can be done in \(\text{sort}(|V|)\) I/Os using the Euler tour technique (see [13]). Now, each edge \((u, v)\) in the original edge list is replaced by \((r_u, r_v)\), where \(r_u\) and \(r_v\) are the representatives of \(u\) and \(v\) respectively. Once done, internal edges of the form \((u, u)\) are removed. Also, multiple \((u, v)\) edges are replaced by the lightest among them. All these operations take \(O(\text{sort}(|E|))\) I/Os.

Since the number of vertices reduces by atleast half after each phase, \(\log \left( \frac{|V|}{M} \right)\) phases render the graph small enough to fit in main memory. Thus, we get a naive \(O(\text{sort}(|E|) \log \left( \frac{|V|}{M} \right))\) I/Os algorithm.

### 2.3 The Current State of the Problem

As mentioned earlier, the \(O(|V| + \text{sort}(|E|))\) I/Os modified Prim’s algorithm matches the lower bound of \(\text{sort}(|E|)\) when \(|V| < \text{sort}(|E|)\). Thus for such ‘dense’ graphs, the algorithm is optimal and quite simple. Even for planar graphs, the lower bound has been achieved. So the only thorn in the flesh is non-planar sparse graphs.
Planarity testing itself is difficult, so we have to tackle sparse graphs, disregarding their planarity. Many algorithms geared towards this end have been proposed. Typically, such an algorithm consists of an initial preprocessing step that reduces the number of vertices to less than $\text{sort}(|E|)$, then runs the modified Prim’s algorithm on the reduced graph. The current best algorithm by Arge that uses $O(\text{sort}(|E|) \log \log(v | B / | E |))$ I/Os, does so by first reducing the no. of vertices to $|E|/B$, which is less than $\text{sort}(|E|)$ in $\log(v | B / | E |)$ phases. It does this by grouping these phases into $\log \log(v | B / | E |)$ superphases, each requiring $\text{sort}(|E|)$ I/Os. A naive way would require $\text{sort}(|E|) \log(v | B / | E |)$ I/Os.
Chapter 3

A Friend, and Blind Alleys

In this chapter, we discuss, in brief, the modified Prim’s algorithm which we use later as a subroutine in our algorithm. We also document a few failed attempts on the way. Failed, not because they were wrong, but because they did not perform up to expectations.

3.1 The Modified Prim’s Algorithm

The classical Prim’s algorithm is well known. [10] suggests a slight modification to the original one for massive graphs. We require an external memory priority queue, which supports \( N \) insert and delete operations in \( \text{sort}(N) \) I/Os. The algorithm is different because it stores edges instead of vertices in the priority queue, and uses a novel way to detect internal edges.

The algorithm assumes that the edge weights are distinct, but it is not a problem. Even non-distinct edge weights can be made distinct by using the weight of \((u, v)\) as \((w(u, v), min(u, v))\), and weights are compared with first component, then with the second component, assuming that the vertices are ordered in some fashion (e.g., we can number them).

The queue is initialized to contain all edges incident to the source vertex. The algo-
algorithm works as follows: The minimum weight edge \((u, v)\) is repeatedly extracted from the priority queue. If \(v\) is already in the MST the edge is discarded. Otherwise \(v\) is included in the MST and all edges incident to \(v\), except \((v, u)\), are inserted in the priority queue.

The correctness of the algorithm follows directly from the correctness of Prims algorithm. The key to its I/O-efficiency is that we have a simple way of determining if \(v\) is already included in the MST if both \(u\) and \(v\) are in the MST when processing an edge \(e = (u, v)\), the edge \(e\) must have been inserted in the priority queue twice. Thus we can determine if \(v\) is already included in the MST by simply checking if the next minimal weight edge in the priority queue is identical to \(e\).

The algorithm performs at least one I/O per vertex, and one scan of the edge list to read the adjacency list of each vertex. Thus it takes \(O(|V| + \frac{|E|}{B})\) I/Os. Also the heap operations take \(O(sort(|E|))\) I/Os. So, total I/Os is \(O(|V| + sort(|E|))\).

### 3.2 Failed Attempts

In these subsections, we document a few attempts that we made in attacking the MST problem.

#### 3.2.1 Attempt 1

We tried to bring in ideas from the Yao’s algorithm ([6]). It is similar to the Boruvka’s algorithm, except that it considers the edges in a partially ordered manner. In the original Boruvka’s algorithm, the edges are unsorted, and we needed to go through the whole adjacency list of a vertex to find its lightest edge. In Yao’s approach, the incident edges of each vertex are partially sorted into \(log|V|\) groups of equal number of members each, where any edge of one group is lighter than any edge of the subsequent groups each. Then, while finding the lightest edge, only one group is considered, thus reducing the time complexity and resulting in an \(O(|E| \log \log |V|)\) time algorithm. In the external version, we can anyway
sort the edges once in the beginning, and the edges are always considered in sorted order. So, selecting the lightest edge is not the main issue. We could not get any more insights from Yao’s approach.

### 3.2.2 Attempt 2

The Fredman-Tarjan ([7]) algorithm is an efficient one for in-core MST computation, and is almost linear, taking time $O(|E| \log^* |V|)$. We tried to externalize it. It proceeds in stages. In each stage, a tree is grown (similar to Prim’s) from each vertex till one of two things happen - the number of neighbors of the tree increase beyond a specified quantity, or the tree hits another tree formed in the same stage. There are $O(\log^* |V|)$ such stages, so if we could perform each stage in $O(sort(|E|))$ I/Os, we get a near optimal algorithm.

The growing trees part was possible, using external memory heaps. However, the main problem came with how to check if the current tree has hit another tree. For this, we needed to maintain a label for each vertex indicating which tree it belonged to. This would require one I/O per vertex for checking its current tree, thus requiring $\Omega(|V|)$ I/Os per stage, which is too costly. We then searched for an external memory union-find data structure to solve the problem. But it turned out to be an open problem. We, however, did find out about a union-find data structure that took $sort(N)$ I/Os for $N$ union and find operations ([14]), but only if all the union and find operations were already given at the beginning, i.e., the queries were offline. But in this case, the queries would have come in an online manner.

### 3.2.3 Attempts 3, 4 and 5

We also contemplated devising efficient disjoint set data structures for massive data, but later decided against it because we felt it would be easier to tackle the MST problem from an algorithmic aspect rather than from a data structure point of view.
There exists an optimal algorithm for planar graphs. We thought about converting the input graph into a planar one by removing additional edges. It turns out that planarity testing is itself difficult. So we abandoned this line of attack.

We then tried to reduce the graph so that the graph can fit in main memory. This would require $O(\log |V|/M)$ Boruvka phases. We tried to group these phases into $\log^*|V|$ superphases, where there is a logarithmic reduction in the number of vertices. A simple analysis, and use of lemma 1 (discussed later in chapter 4) leads to a $O(\log^*|V| \cdot \text{sort}(\frac{|V|}{\log |V|}) \log \left( \frac{|V|}{\log |V|} \right))$ I/Os algorithm.

3.2.4 Attempt 6

We also tried to modify the modified Prim’s algorithm. The modified Prim’s algorithm’s Achilles’ heel was the $O(V)$ I/Os it took while reading the adjacency list of each vertex. This occurs because when a new MST edge $(u, v)$ is extracted from the heap, the vertex $v$ can be any arbitrary vertex, so we have to bring the read-write head of the hard disk to a new position. This on an average leads to half an I/O. Other than that, the modified Prim’s is a very simple and elegant algorithm. We did try to store the pointer to the starting of the adjacency list of each vertex with each edge, but it did not result in any improvement.

3.2.5 Attempt 7

Another approach we tried was reducing the number of vertices to $\text{sort}(E)$ instead of $\frac{|E|}{B}$, as most current algorithms try. We got a naive $O(\text{sort}(|E|) \log \left( \frac{|V|}{\text{sort}(E)} \right))$ I/Os algorithm. This simple idea finally led to us to devise the $\text{ReduceVertices}$ procedure discussed in the next section, which finally led to our improved algorithm.
Chapter 4

Our Algorithm

In this chapter, we present a new and simple algorithm for computing an MST. The algorithm uses a procedure ReduceVertices to reduce the number of vertices in an input graph using Boruvka phases.

4.1 The ReduceVertices Procedure

We present below the procedure. Its proof of correctness and analysis follow.

4.1.1 Proof of correctness

Lemma: 1 To perform $i$ Boruvka phases, $2^i$ lightest edges incident to each vertex are sufficient.

Proof: All we require is that after $i$ phases, each supervertex should have at least $2^i$ vertices. We prove the claim by induction.
For $i = 1$, only 1 edge incident to each vertex is used up. So, the lemma holds.
Suppose it holds for $i \leq n - 1$. Now, after performing $n - 1$ Boruvka phases and while performing the $nth$ phase, if a supervertex $v$ has an external edge left from the initial edge list, then we use that for hooking. So, the lemma holds. But if $v$ does not have any external edges, then it means there is a vertex $u$ in $v$ for which all the $2^i$ incident edges have become
Procedure 4.1.1 ReduceVertices\((V, E, S)\)

**Input:** An undirected graph \((V, E)\) and a number \(S\)

**Output:** An undirected graph \((V', E')\) having at most \(S\) vertices

{It performs at most \(O(\log \frac{V}{S})\) Boruvka phases}

1. \(i \leftarrow 0; V_0 \leftarrow V; E_0 \leftarrow E; T \leftarrow []; F \leftarrow []\)

2. while \(|V_i| > S\) do

3. Form a partial edge list \(E'\) from \(E_i\) containing \(\left\lceil \frac{|V_i|}{S} \right\rceil\) lightest edges incident to each vertex \(v \in V_i\).

4. If \((u, v) \in E'\) but \((v, u) \notin E'\), then add \((v, u)\) to \(E'\).

5. Thus \((V_i, E')\) is a full fledged graph. Pick the lightest edge incident to each vertex. Add these edges to an empty list \(T'\).

6. The connected components of \(T'\) form pseudo trees. Remove the appropriate edge from each connected component to break the cycle. Then, the remaining edges in \(T'\) form trees, and are part of the MST of \((V, E)\). Add them to \(T\).

7. For each component in \(T'\), pick a representative vertex. Store each \((v, r_v)\) pair in \(F\), where \(r_v\) is the representative of \(v\). These representatives form \(V_{i+1}\). Contract the component, and remove internal and multiple edges from \(E'\) to get a new graph \((V_{i+1}, E_{i+1})\).

8. \(i \leftarrow i + 1\)

9. end while

10. \(T\) contains the MST edges computed till this stage. We now have a graph \((V_i, E_i)\) with \(|V_i| \leq S\). But the original edge list \(E\) is not clean and contains internal and multiple edges. We need to clean up \(E\).

11. The edges in \(T\) form an MSF of \((V, E)\). We now clean up \(E\). \(F\) contains all the clustering information. We can view it as a collection of star graphs. We find a final representative for each vertex using the information in \(F\).

12. Once we get a representative for each vertex, these form \(V'\) using which we clean up \(E\) to get \(E'\).

13. return \((V', E')\).

---

internal to \(v\). So, all \(2^i\) neighbours of \(u\) are in \(v\). Hence \(v\) has \(2^i\) vertices, and it has no need for hooking. Thus, lemma 1 holds in this case.

What lemma 1 means is this: To perform \(n\) phases, only \(2^n\) lightest edges incident to each vertex is sufficient. All the remaining edges are unnecessary and can be ignored.

Now, the loop runs at most \(\log \frac{|V|}{S}\) times, as in each iteration the number of vertices is reduced by at least half. Consider any \(i \leq \log \frac{|V|}{S}\). Suppose the loop runs successfully
for \((i - 1)\) times. When the loop runs for the \(ith\) time, it has input \((V_{i-1}, E_{i-1})\). Also, \(|V_{i-1}| > S\). So, at most \(\log \frac{|V_{i-1}|}{S}\) phases are required further. For this, \(\frac{|V_{i-1}|}{S}\) lightest edges are required for each vertex in \(V_{i-1}\). This partial edge list is formed in the first step of the loop body, and hence the loop runs successfully for the \(ith\) time.

Steps 3 onwards are self explanatory.

**4.1.2 Analysis**

We now analyse the I/O complexity of the above procedure. In one Boruvka phase, the number of vertices reduces by at least half. So, \(|V_i| \leq \frac{|V|}{2^i}\).

For any \(i\), \(|E'| \leq \frac{2|V|}{S}\).

Steps 1 and 2 of the loop take \(\text{sort}(|E_i|)\) I/Os, where \(E_0 = E\), and for \(i > 0\), \(|E_i| \leq \frac{2|V_i|^2}{S}\).

The remaining steps of the loop take \(\text{sort}(E') = O(\text{sort} \left( \frac{|V_i|^2}{S} \right))\) I/Os.

\[\therefore \text{Total I/Os required till we get out of the loop}\]

\[= \sum_{i=0}^{\log \frac{|V|}{2}} O(\text{sort} \left( \frac{|V|^2}{S} \right))\]

\[\leq \sum_{i=0}^{\infty} O(\text{sort} \left( \frac{|V|^2}{S} \right))\]

\[\leq c \sum_{i=0}^{\infty} \text{sort} \left( \frac{|V|^2}{S} \right), \text{ for some constant } c\]

\[= c \sum_{i=0}^{\infty} \frac{|V_i|^2}{BS} \log_{M/B} \left( \frac{|V|^2}{BS} \right)\]

\[\leq c \sum_{i=0}^{\infty} \frac{(|V|/2^i)^2}{BS} \log_{M/B} \left( \frac{|V|^2}{BS} \right)\]

\[\leq c \frac{|V|^2}{BS} \log_{M/B} \left( \frac{|V|^2}{BS} \right) \sum_{i=0}^{\infty} \frac{1}{2^i}\]

\[= O(\text{sort} \left( \frac{|V|^2}{S} \right))\]

Step 11 takes \(\text{sort}(|T|)\) I/Os, using a technique discussed in [13]. Steps 12 and 13 take
\[ \text{sort}(|E|) \text{ I/Os. Since } |T| \leq |E|, \text{ we require a total of } O(\text{sort}(|E|)) \text{ I/Os.} \]

\[
\therefore \text{Total I/Os required} = O(\text{sort} \left( \frac{|V|^2}{|E|} \right) + \text{sort}(|E|)) \text{ I/Os.}
\]

4.2 The Algorithm

The ReduceVertices procedure gives us an efficient way to reduce the number of vertices. This procedure can be called before the modified Prim’s algorithm is run on the reduced graph. Modified Prim’s requires that \(|V| < \text{sort}(|E|)\) for optimal performance. So, we put \(S = \text{sort}(|E|)\) in the above procedure. For the above procedure to work in \(\text{sort}(|E|)\) I/Os, we require

\[
\frac{|V|^2}{\text{sort}(|E|)} \leq |E| \Rightarrow \frac{|V|^2}{|E|} \leq \text{sort}(|E|)
\]

But what if \(\frac{|V|^2}{|E|} > \text{sort}(|E|)\), or \(|V| > (|E|\text{sort}(|E|))^{1/2}\)? We can call ReduceVertices on \((V, E)\) with \(S = (|E|\text{sort}(|E|))^{1/2}\) to reduce the number of vertices. For this call to require \(\text{sort}(|E|)\) I/Os, we need

\[
\frac{|V|^4}{(|E|^2\text{sort}(|E|))^{1/2}} \leq |E| \Rightarrow \frac{|V|^4}{|E|^3} \leq \text{sort}(|E|)
\]

Again, what if \(\frac{|V|^4}{|E|^3} > \text{sort}(|E|)\), or \(|V| > (|E|^3\text{sort}(|E|))^{1/4}\)? We call ReduceVertices on \((V, E)\) with \(S = (|E|^3\text{sort}(|E|))^{1/4}\), then again with \(S = (|E|\text{sort}(|E|))^{1/2}\), for a total of two calls.

Since \(\frac{|V|}{|E|} < 1\), so, \(\frac{|V|^{i+1}}{|E|^{i}}\) decreases as \(i\) increases. Now, the question arises - how many times do we need to call ReduceVertices, with each call taking \(\text{sort}(|E|)\) I/Os? Well, as long as we do not get a graph \(G'(V', E')\) with \(|V'| \leq \text{sort}(|E|)\). If we closely observe the
above pattern, we see that the call has to be made \( n \) times where

\[
\left( \frac{|V|}{|E|} \right)^{2^n} \leq \frac{\text{sort}(|E|)}{|E|} < \left( \frac{|V|}{|E|} \right)^{2^{n-1}}
\]

\[
\Rightarrow \left( \frac{|E|}{|V|} \right)^{2^{n-1}} \leq \frac{|E|}{\text{sort}(|E|)} < \left( \frac{|E|}{|V|} \right)^{2^n}
\]

\[
\Rightarrow n - 1 \leq \log \log \frac{|E|}{|V|} \left( \frac{|E|}{\text{sort}(|E|)} \right) < n
\]

This can be better illustrated using a diagram. Consider the number line shown in figure 4.1. The number of calls to \textit{ReduceVertices} required depends upon the region in which \( \text{sort}(|E|) \) lies in the number line. With each call, we \textit{hop} from one region to its immediate left one (as the number of vertices decreases with each call w.r.t. to \( \text{sort}(|E|) \)), till we get to the lefmost unbounded region, after which we apply the modified Prim’s algorithm.

![Diagram](image)

\textbf{Fig. 4.1} Relation between \( \text{sort}(E) \) and number of calls to \textit{ReduceVertices}.

Since each call takes \( \text{sort}(|E|) \) I/Os, the I/O complexity of the algorithm is \( O \left( \text{sort}(|E|) \log \log \frac{|E|}{|V|} \left( \frac{|E|}{\text{sort}(|E|)} \right) \right) \). The complete algorithm is listed below.
**Algorithm 4.2.1 ComputeMST(V, E)**

**Input:** An undirected graph \((V, E)\)

**Output:** A minimum spanning tree \((V', E')\)

1: Find the smallest number \(n\) such that \(\frac{|E|}{\text{sort}(|E|)} < \left(\frac{|E|}{|V|}\right)^{2^n}\)
2: if \(n = 0\) then
3: Apply modified Prim’s on \((V, E)\) to get \((V', E')\)
4: return \((V', E')\)
5: end if
6: \(V_{n+1} \leftarrow V; E_{n+1} \leftarrow E;\)
7: for \(i = n\) down to 1 do
8: \(S_i \leftarrow \left(\frac{|E|^{2^i-1}\text{sort}(|E|)}{2^i}\right)^{\frac{1}{2^i}}\)
9: \((V_i, E_i) \leftarrow \text{ReduceVertices}(V_{i+1}, E_{i+1}, S_i)\)
10: end for
11: Apply modified Prim’s on \((V_1, E_1)\) to get \((V', E')\)
12: return \((V', E')\)

### 4.2.1 Comparison with other Algorithms

We have an \(O(\text{sort}(|E|) \log \log \frac{|E|}{|V|} B)\) I/Os algorithm given in [13]. For our algorithm to outperform, we must have

\[
\log \log \frac{|E|}{|V|} \left(\frac{|E|}{\text{sort}(|E|)}\right) < \log \log \frac{|E|}{|V|} B
\]

\[\Rightarrow \log_{M/B} \left(\frac{|E|}{B}\right) > 1\]

\[\Rightarrow \log \left(\frac{|E|}{B}\right) > \log(M/B)\]

\[\Rightarrow \log |E| > \log B ,\] which is anyway true

Thus our algorithm is better. It is also simpler to implement, and has no large, hidden constant factors, unlike in [13].

The best algorithm known ([10]) has complexity \(O(\text{sort}(|E|) \log \log(VB/E))\). The algorithm in [13] performs better than ([10]) for practically all values of \(V, E\) and \(B\), \(B \gg 16\), and \(B^{1-\sqrt{\frac{1}{2} - \frac{\log B}{2}}} \leq |E|/|V| \leq B^{\frac{1+\sqrt{1-\frac{1}{2} - \frac{\log B}{2}}}{2}}\). It matches the lower bound when \(|E|/|V| \geq B^e\)
for a constant $\epsilon > 0$. Specifically, when $|E|/|V| = B^\epsilon$, for a constant $0 < \epsilon < 1$, it performs faster than [10] by a factor of $\log \log B$. And our algorithm comprehensively trumps [13].
Chapter 5

Conclusion and Future Work

In this report we have documented the attempts we made for finding a better algorithm for the MST problem. We explored various lines of attack, like externalising Fredman-Tarjan, logarithmic reduction of vertices per superphase, modifying the modified Prim's algorithm, converting non-planar graphs to planar ones, reducing $|V|$ to $\text{sort}(|E|)$ rather than $|E|/B$, externalising Yao’s approach and externalising disjoint-set data structures. We finally presented an algorithm that used a very simple lemma and cleaned up only the required edges at each step, delaying the clean up of the entire edge list till the very last. It used nothing but sorting the edges. It triumphs over the current best algorithm for almost all practical situations, and even reaches the optimum in many cases.

Implementing this algorithm is very simple, and can be taken up at a later stage. The quest for a $O(\text{sort}(|E|))$ I/Os algorithm still continues.
References

[1] www.worldwidewebsize.com


