[thm]

Parallel algorithms for the construction of special subgraphs

zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften

der Fakultaet fuer Informatik

der Universitaet Fridericiana zu Karlsruhe (TH)

genehmigte

Dissertation

von

Mahmoud Ibrahim Moussa

aus Dakhlia, Aegypten

Tag der muendlichen Pruefung: 13. Juli 2005

Erster Gutachter: Prof.Dr. Roland Vollmar

Zweiter Gutachter:Prof.Dr. Dorothea Wagner

Zusammenfassung meiner Dissertation

In dieser Arbeit werden wir zwei Probleme aus der Graphentheorie studieren und zwei parallele Algorithmen fuer das Loesen dieser Probleme beschreiben. Das Problem einen, minimalen spannenden Baum in einem zusammenhaengenden Graphen $G_0 = (V, E)$ mit ganzzahlig gewichteten Kanten zu finden, ist das erste Problem, das wir studieren werden. Wir geben einen neuen deterministischen parallelen Algorithmus fuer das Berechnen des minimalen spannenden Baums eines gegebenen Graphen. Dieser Algorithmus benoetigt fuer einen Graphen mit n Knoten und m Kanten eine Zeit von $O(\log n)$ bei der Benutzung von $O(n+m)$ Prozessoren.

Das zweite Problem ist das abgeleitete Teilgraphproblem. Wir geben einen parallelen Algorithmus an, welcher die Zahl der abgeleiteten Teilgraphen $n_d(\mathfrak{G}_o)$ berechnet, und die Menge aller abgeleiteten Teilgraphen durch Partitionierung der Menge $V(\mathfrak{G}_o)$ findet. Wir erklaeren die wissenschaftlichen und kommerziellen Anwendungen sowohl fuer das abgeleitete Teilgraphenproblem als auch fuer das Residuum-Kanten Problem in einem gegebenen Graphen.

Sei $\mathfrak{G}_o(n, m)$ ein Graph mit n Knoten und m Kanten. Wir benutzen ein verteiltes Speichermodell, in dem die Prozessoren ein pyramidale Verbindungsnetz bilden. In diesem Fall ist die parallele Laufzeit $O(n \log^2 n)$ unter Benutzung von $p = (4n^2/\log^2 n - 1)/3$ Prozessoren. Wir werden diese erste parallele Version des abgeleiteten Algorithmus mit anderen in Beziehung stehenden Resultaten praesentieren.

Abstract of my thesis

In this work we are going to study two problems in graph theory and describe two parallel algorithms for solving them. The problem of finding a minimum spanning tree (MST) in a connected graph $G = (V, E)$ with integer-valued edge weights, is the first problem we are going to study. We give a new deterministic parallel algorithm for computing the minimum spanning tree of a given graph. This algorithm runs on graph with n vertices and m edges in $O(\log n)$ time using $O(n + m)$ processors on an EREW PRAM.

The second problem examined is the derived subgraph problem. We give a parallel algorithm which calculates the number of derived subgraphs denoted $n_d(G)$ and finds the set of all derived subgraphs by making a partition of the vertex set $V(G)$. We explain scientific, commercial applications for the derived subgraphs problem and the residual and the non-residual edges in a given graph.

If the parallel algorithm runs on graph $\mathfrak{G}_{\circ}(n, m)$ with n vertices and m edges. We use a distributed memory model, in which processors form a pyramid-connected computer network. The parallel running time is $O(n \log^2 n)$ using $p = (4n^2/\log^2 n - 1)/3$ processors.

Acknowledgements

My deepest gratitude goes to my supervisor, Professor *Roland Vollmar*, for his guidance, support, motivation and encouragement throughout the period this work was carried out. His readiness for consultation at all times, his educative comments, his concern and assistance even with my life things.

Special thanks are due to Prof.Dr. Dorothea Wagner for an excellent and thorough review of this work.

I am grateful to my parents, and my wife *Dr. Hanaa* for their moral support, encouragement and for their pray for me.

I direct special thanks to my friend *Dr.eng. M. Helal*, thank you for your support during writing this work.

I am also indebted to all staff members of Institut fuer Algorithmen und Kognitive Systeme, Faculty of Computer Science, Karlsruhe University, Germany, for their support during the preparation of this thesis.

The List of Figures

Contents

Chapter 1

Why Parallel Computation ?

1.1 Introduction

Parallel computation is defined as the practice of using a large number of co-operating processors, communicating among themselves to solve large problems fast and it is quickly becoming an important area in computer science. It is possible that in the next years this area will have grown so wide and strong that most of the research conducted in the fields of design and analysis of algorithms, computer languages, computer applications and computer architectures will be within the context of parallel computation.

In this chapter we would like to talk about the reasons for the need of ever greater computing power. Parallel computers are used primarily to speed up computations.

1.1.1 The Need for Parallel Computation

We give the following example to illustrate the need for more computational power.

Suppose we wish to predict the weather over Europe for the next two days. Also suppose that we want to model the atmosphere from sea level to an altitude of 20 kilometers, and we need to make a prediction of the weather at each hour for the next days.

A standard approach to this type of problem is to cover the region of interest with a grid and then predict the weather at each vertex of the grid. So suppose we use a cubical grid, with each cube measuring 0.1 kilometer on each side. Since the area of Europe is about 11 million square kilometers, we need at least

$$11 \times 10^6 \times 20 \times 10^3 = 22 \times 10^{10} \text{ grid points.} \tag{1.1}$$

If it takes 100 calculations to determine the weather at a typical grid point, then in order to predict the weather one hour from now, we'll need to make about 22×10^{12} calculations. Since we want to predict the weather at each hour for 48 hours, we need to make a total of about

$$22 \times 10^{12} \text{ calculations} \times 48 \text{ hours} \approx 10^{15} \text{ calculations} \tag{1.2}$$

If our computer can execute 10^9 calculations per second, it will take about

$$10^{15} \text{ calculations} / 10^9 \text{ calculations per second} = 10^6 \text{ second} \approx 11 \text{ days !} \tag{1.3}$$

In other words, the calculation is hopeless if we can only carry out 10^9 operations per second. If, on the other hand, we can carry out 10^{12} calculations per second, it will take us about 16 minutes to carry out the computations. So we shall actually be able to make a complete prediction of the weather over each of the next 48 hours. It is not difficult to imagine simple modifications to this problem so that 10^9 operations per second will not be sufficient. For example, we might replace Europe with the entire earth. Then the area would go from 11×10^6 to about 5×10^8 square kilometers. So the required computation time would increase from 16 minutes to about 12 hours, and our first 11 predictions would be useless. Furthermore, it is not difficult to find completely different problems requiring vastly greater computational power than we currently possess. For example, detailed atomic-level simulations of biomolecules and numerous types of simulations that would expedite the design and manufacture of integrated circuits all require vastly greater computational power than we currently possess.

It is obvious that the one way around this problem is to use parallelism. The idea here is that if several operations are performed simultaneously, then the time taken by a computation can be significantly reduced.

1.1.2 Organization of the Thesis

The thesis consists of six chapters and is organized as follows. Each chapter begins with an introduction, in which some informations and ideas on the contents of this chapter are given. In the next chapter we review the existing models of

parallel computation. In chapter three, we summarize definitions of graphs and their subgraphs.

In chapter four we review some of the previous work concerning some famous sequential algorithms for the *Minimum Spanning Tree* problem. In chapter five, we introduce a new deterministic parallel algorithm for computing the Minimum Spanning Tree of a given Graph. In chapter six, we summarize the results concerning derived subgraph and residual edges. We present a parallel version for the derived subgraph algorithms, to find and calculate the number of derived subgraphs for a given graph.

Chapter 2

Models of Parallel Computation

2.1 Introduction

The main goal of writing a parallel program is to get a shorter computation time compared with the serial version. With this in mind, there are several issues that we need to consider when designing our parallel code to obtain the best performance possible within the constraints of the problem being solved. These issues are:

i) **Load balancing**

Load balancing is the task of equally dividing work among the available processes. This can be easily done when the same operations are performed by all the processes (on different pieces of data). It is not trivial when the processing time depends upon the data values being worked on. When there are large variations in processing time, we may be required to adopt a different

method for solving the problem.

ii) **Minimizing Communication**

The total execution time is a major concern in parallel programming because it is an essential component for comparing and improving all programs. Three components make up execution time:

1- **Computation Time** is the time spent performing computations on the data. Ideally, we should expect that if we had n processors working on a problem, we should be able to finish the job in $1/n$-th the time of the serial job. This would be the case if all the processor's time was spent in computation.

2- **Idle Time** is the time a process spends waiting for data from other processors. During this time, the processors do no useful work.

3- **Communication Time** is the time it takes for processes to send and receive messages. The cost of communication in the execution time can be measured in terms of latency and bandwidth. Latency is the time it takes to set up the envelope for communication, where bandwidth is the actual speed of transmission, or bits per unit time. Serial programs do not use interprocess communication. Therefore, we have to minimize this use of time to get the best performance improvements.

iii) **Overlapping Communication and Computation**

There are several ways to minimize idle time within processes, and one example is overlapping communication and computation. This involves occupying a process with one or more new tasks while it waits for communication to finish so it can proceed on another task. Careful use of nonblocking communication and data unspecific computation make this possible. It is very difficult in practice to interleave communication with computation.

iv) **Running Time**

Since speeding up computations appears to be the main reason behind our interest in building parallel computers, the most important measure in evaluating a parallel algorithm is therefore its running time. This is defined as the time taken by the moment the algorithm starts to the moment it terminates. If the various processors do not all begin and end their computation simultaneously, then the running time is equal to the time elapsed between the moment the first processor on the parallel computer to begin operating on the input starts and the moment the last processor to end producing the output terminates.

The running time of a parallel algorithm is usually obtained by counting two kinds of steps; computational steps and routing steps; each of these steps requires a constant number of time units: A computational step is an arithmetic or logic operation performed on a datum within a processor. In a

routing step, a datum travels from one processor to another via the shared memory or through the communication network.

For a problem of size n the parallel worst-case running time of an algorithm, a function of n, will be denoted $t_p(n)$.

v) **Speedup S_p.**

In evaluating a parallel algorithm for a given problem, it is quite natural to do it in terms of the best available sequential algorithm for that problem.Thus a good indication of the quality of a parallel algorithm is the speedup it produces. This is defined as

$$S_p = \ t_{seq}(n) \ / \ t_p(n)$$

where $t_{seq}(n)$ denotes the worst-case running time of the fastest known sequential algorithm for the given problem. Clearly, the larger the speedup, the better the parallel algorithm.

vi) **Number of Processors**

One of the most important criteria in evaluating a parallel algorithm is the number of processors it requires to solve a problem. It costs money to purchase, maintain, and run computers. Therefore, the larger the number of processors an algorithm uses to solve a problem, the more expensive the solution becomes to obtain. We denote the number of processors required by

an algorithm to solve a problem of size n by $p(n)$. Sometimes the number of processors is a constant independent of n.

vii) **The Cost $C(n)$.**

The cost $C(n)$ of a parallel algorithm is defined as the product of the parallel running time and the number of processors, hence

$$C(n) = t_p(n) \times p(n)$$

The cost of a parallel algorithm for a given problem is said to be cost optimal, if the cost of the parallel algorithm matches a lower bound on the number of sequential operations required in the worst case to solve the problem. A parallel algorithm is not cost optimal if a sequential algorithm exists whose running time is smaller than the parallel algorithm's cost.

viii) **The Efficiency $E(n)$.**

The efficiency of a parallel algorithm running on p processors is the speedup divided by p. Let us give the following example to illustrates the meaning of efficient parallel algorithm. If the best known sequential algorithm executes in 12 seconds (on one processor), while a parallel algorithm solving the same problem executes in 3 seconds when 5 processors are used, then we say that the parallel algorithm exhibits a speedup $S_5 = 4$ with five processors. A parallel algorithm that exhibits a speedup of 4 with five processors has an efficiency of 0.8 with five processors. The efficiency $E(n)$ of a parallel

algorithm for solving a problem is defined as follows:

$$E(n) = t_{seq}(n)/C(n).$$

2.2 Classification of Parallel Computers

Any computer, whether sequential or parallel, operates by executing instructions on data. A stream of instructions (the algorithm) tells the computer what to do at each step. A stream of data (the input to the algorithm) is affected by these instructions. In 1972 Michael Flynn [12] classified computers by the number of instruction- and data-streams, respectively.

In terms of memory-processor organization parallel computers have three basic architectures. These are:

2.2.1 Shared Memory Architectures

The main property of shared memory architectures is, multiple processor units share access to a global memory via a high-speed memory bus. This class is known in the literature as the Parallel Random-Access Machine (PRAM). This global memory allows the processors to efficiently exchange or share access to data. At the same time each processor is able to perform the usual computation of a sequential RAM using a finite amount of local memory (see Fig. 2.1). Typically, the number of processors used in shared memory architectures is limited to only a handful $(2-16)$ of processors. This is because the amount of data that

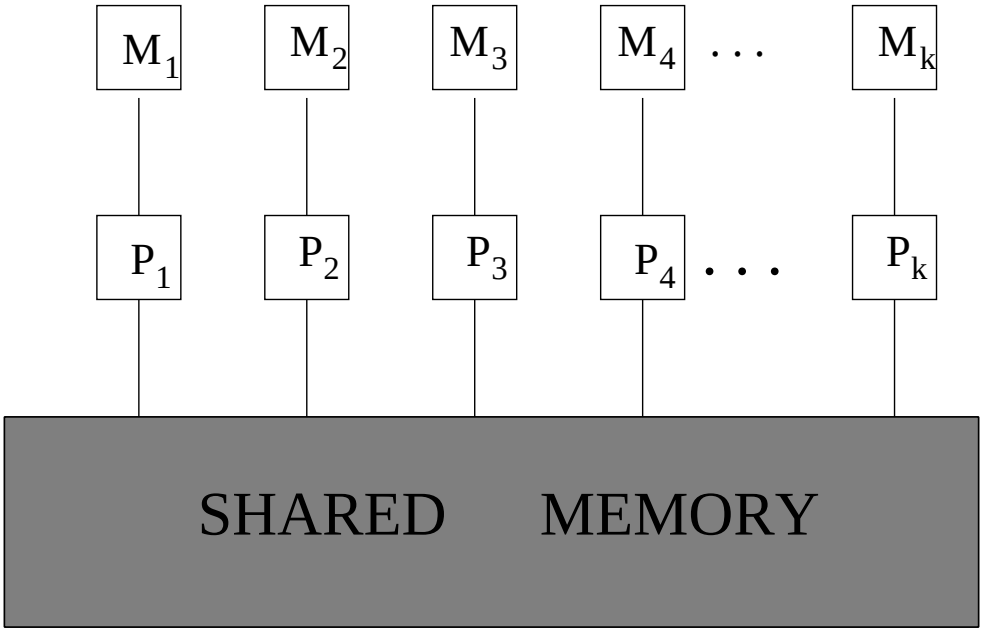


Figure 2.1: The PRAM model.

can be processed is limited by the bandwidth of the memory bus connecting the processors. Because the Shared Memory Parallel Computer models is the one that we will use, we give a detailed description of it.

If processor P_i wants to communicate with processor P_j, it can do so by writing to some memory location from which P_j will read. So, the model makes the assumption that communication between processors takes unit time. The basic model allows all processors to gain access to the shared memory simultaneously if the memory locations they are trying to read from or write into are different. However, the class of shared-memory computers can be further divided into four subclasses, according to whether two or more processors can gain access to the same memory location simultaneously:

I- **Exclusive-Read, Exclusive-Write (EREW).**

In the less powerful but perhaps most realistic EREW-PRAM model, no conflicts are permitted for either reading or writing, because no two processors are allowed simultaneously to read from or write into the same memory location.

II- **Concurrent-Read, Exclusive-Write (CREW).**

In CREW-PRAM model multiple processors are allowed to read from the same memory location but the right to write is still exclusive: No two processors are allowed to write into the same location simultaneously.

III- **Exclusive-Read, Concurrent-Write (ERCW).**

In ERCW-PRAM model multiple processors are allowed to write into the same memory location but the right to read is still exclusive: No two processors are allowed to read from the same location simultaneously.

IV- **Concurrent-Read, Concurrent-Write (CRCW).**

Finally, the CRCW-PARM, the strongest of these models, permits simultaneous accesses for both reading and writing.

2.2.2 Distributed Memory Parallel Computers

In case of a distributed memory computer a collection of serial computers (nodes) works together to solve a problem. Each node has rapid access to its own local

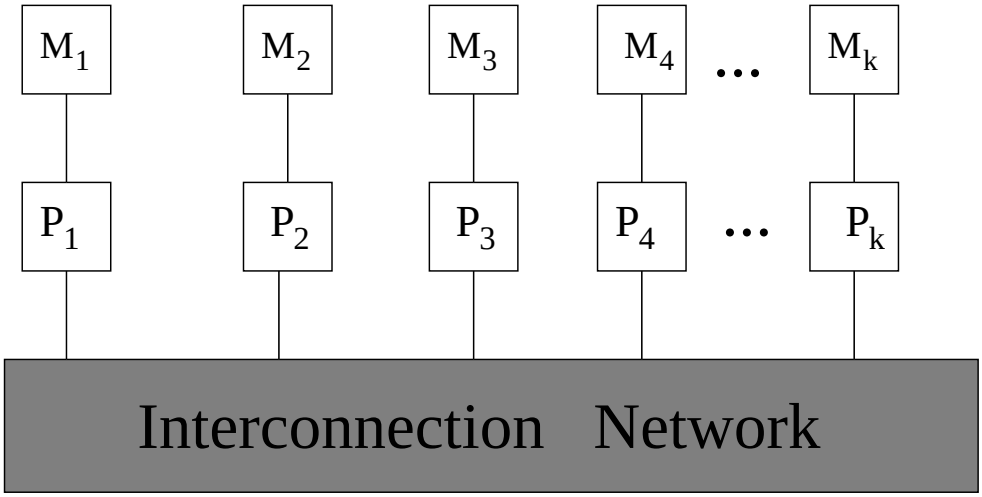


Figure 2.2: Distributed Memory Architecture

memory and access to the memory of other nodes via some sort of communications network, usually a proprietary high-speed communications network. Data are exchanged between nodes as messages over the network (see Fig. 2.2). Two processors directly connected by a link are said to be neighbors. Two processors connected by a link can exchange data simultaneously. In other words, the link between the processor P_i and the processor P_j represents two links, one from P_i to P_j and one from P_j to P_i. Examples of these models are IBM SP-2 and clusters built up of independent workstations. Such models are usually programmed using an explicit message passing library, e.g. Message Passing Interface (MPI) or Parallel Virtual Machine (PVM).

We now turn to a description of two various type of processor organization used in this thesis.

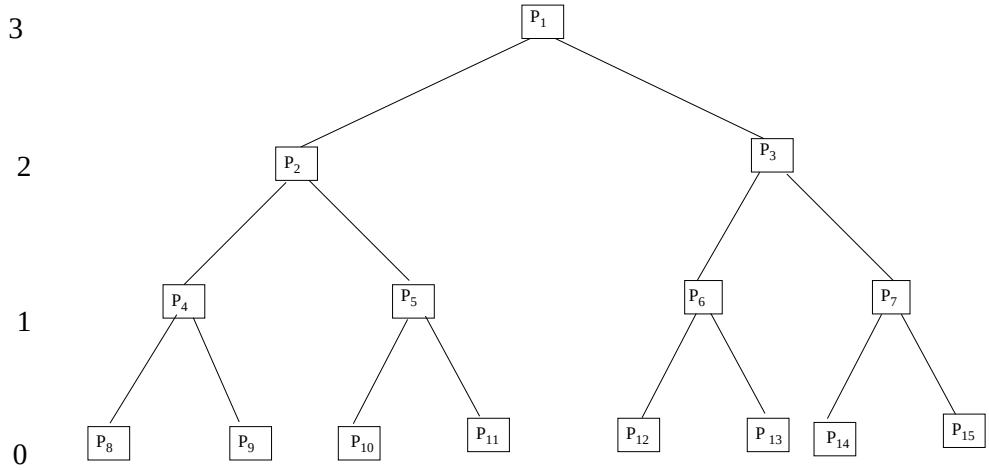


Figure 2.3: A tree interconnection network

2.2.3 The Binary Tree Networks

In a complete binary tree network of depth $k-1$, the number of nodes is $2^k - 1$ (Fig. 2.3). Every node has at most three links. Every node other than the root is connected to one parent, and every interior node is connected to its two children. The binary tree has a low diameter, $2(k-1)$, but it has only bisection width of one.

2.2.4 The Pyramid Networks

A two-dimensional pyramid computer consists of $(4^{d+1}-1)/3$ processors distributed among $d+1$ levels (Fig. 2.4). All processors at the same level are connected to form a two-dimensional mesh (and the nodes are arranged into a two-dimensional lattice). At level 0 (also called base), there are 4^d processors arranged in a $2^d \times 2^d$ mesh. There is only one processor at level d (called the apex). In general, a processor at level i, in addition to being connected to its

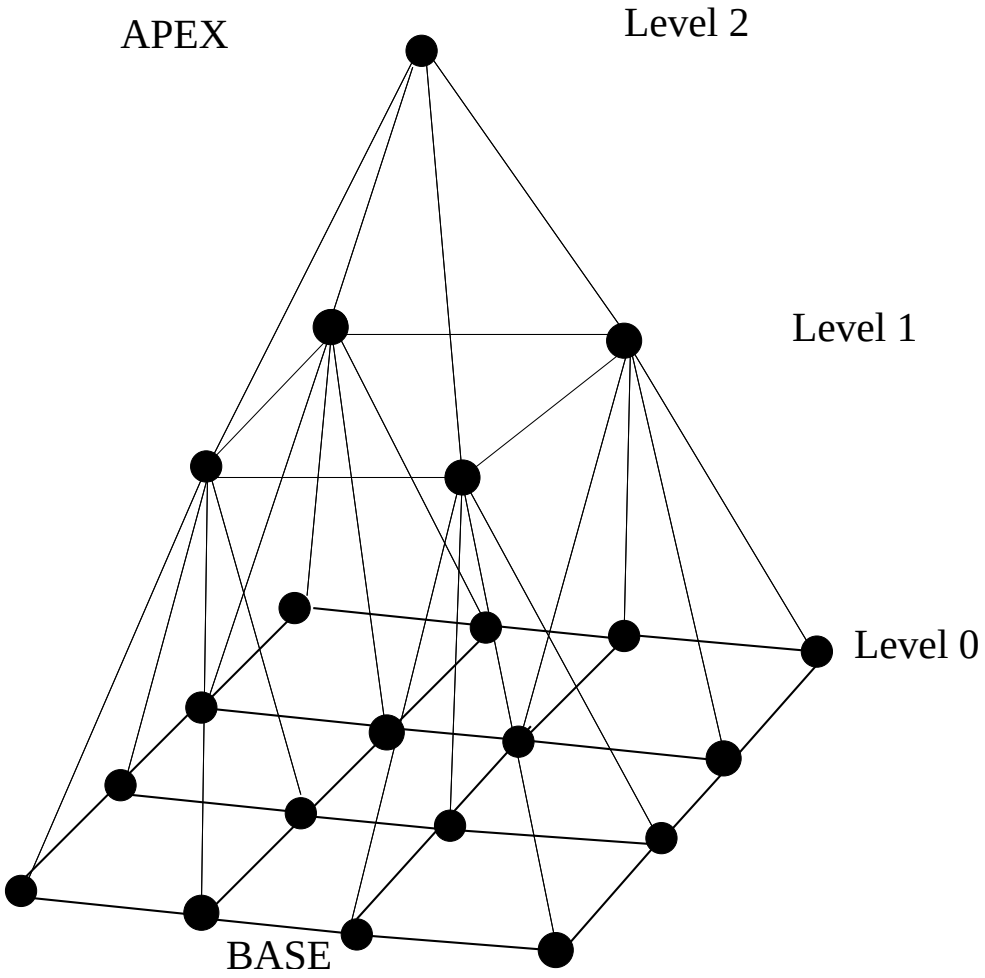


Figure 2.4: A pyramid interconnection network

four neighbors at the same level, has connections to four children at level $(i-1)$, provided that $i \geq 1$ and is connected to one parent at level $(i+1)$ provided that $i \leq d-1$. The maximum number of links per node is not greater than nine, regardless of the size of the network.

2.2.5 Distributed Shared Memory Architectures

The latest technology of parallel computers is based on a mixed shared-distributed memory architecture. On one hand, it eases programming, and on the other hand it has a high scalability, contrary to the distributed memory architecture. Each processor has its own local memory, all memory modules form one common address space, i.e. each memory cell has a system-wide unique address. Each node consists of a group of 2 to 16 processors connected via a local shared memory, and the multiprocessor nodes are connected via a high-speed communication.

2.3 NC Class and the Parallel Complexity Theory

A problem is said to belong to the class NC (Nick's Class) if it can be solved in time polylogarithmic in the size of the problem using at most a polynomial number of processors. Examples of problems in NC include sorting, finding minimum-cost spanning trees, and find convex hulls. To be more precise we present some definitions.

Definition 2.1.

The expression $T(n)^{O(1)}$ denotes polynomial functions of $T(n)$.

For example, the functions $(\log n)^2$ and $(\log n)^3$ are in the set $(\log n)^{O(1)}$, such the set $(\log n)^{O(1)}$ is called the set of **polylogarithmic** functions. The functions n^2 and n^3 are in the set $n^{O(1)}$, this set $n^{O(1)}$ is called the set of **polynomial** functions. A problem is in class NC if and only if it has an algorithm whose time $t_p(n)$ and processor numbers $p(n)$ complexity, respectively are:

$$t_p(n) \in (\log n)^{O(1)} \quad \text{and} \quad p(n) \in n^{O(1)}$$

Many NC algorithms are cost optimal. For example, they have $t_p(n) = O(\log n)$ with $p(n) = n/\log n$ if $t_{seq}(n) = O(n)$ (for example parallel prefix computation). Some cost optimal NC algorithms are super fast, they achieve even sub-logarithmic time. For example, there are string matching algorithms with $O(\log\log n)$ parallel time if $p(n) = n/(\log\log n)$ processors, where n is the length of the string. It is an open problem whether $NC = P$, but it seems unlikely that every problem in P is in NC [25].

One of the major goals in designing parallel algorithms is to minimize $t_p(n)$ as well as the cost $C(n)$. It has been demonstrated that it is easier to design algorithms for the more powerful CRCW PRAM model than for the other models. However, due to the existing simulations between the PRAM models, the notion of efficient parallel algorithms is robust, in the sense that any efficient algorithm for some model is still efficient in any weaker model. On the other hand, the notion of optimal algorithms is not robust in this sense.

Chapter 3

Graphs and Their Subgraphs

3.1 Introduction

In this chapter we begin by working through the basic definitions of graphs (their subgraphs) and the properties of graphs (their subgraphs). We give some basic concepts and definitions about graphs. These concepts are needed in the next chapters.

3.2 Basic Definitions for Graphs

Definition 3.1.

An undirected graph (simply a graph) G=(V, E) is a finite non-empty set V of elements, called vertices, together with a -possibly empty- set E of unordered pairs of distinct vertices of G called edges. The vertex set V of G is denoted $V(G)$, while the set of edges E is denoted $E(G)$.

If u and v are two vertices of a graph G, then an edge $e = (u, v)$, simply

denoted $e = uv$, is said to join u and v. If $e = uv$ is an edge of a graph G, then the two vertices u and v are said to be adjacent. The edge e is said to be incident with both u and v. Furthermore, two distinct edges e_1 and e_2 are adjacent if e_1 and e_2 are incident with a common vertex. The order of a graph G is the cardinality of its vertex set $V(G)$ and is denoted $n(G)$ or simply n, while the cardinality of its edge set is denoted m.

There are two standard ways to represent a graph G=(V, E): as a collection of adjacency lists or as an adjacency matrix. An adjacency-matrix representation of a graph is a $|V| \times |V|$ matrix $A = (a_{ij})$ of boolean values such that $a_{ij} = 1$ if $(v_i, v_j) \in E$, and $a_{ij} = 0$ otherwise. The adjacent-list representation of a $G = (V, E)$ consists of an array Adj of $|V|$ lists, one for each vertex in V. For each $u \in V$, the adjacency list $Adj[u]$ contains (pointers to) all the vertices v such that there is an edge $(u, v) \in E$. That is, $Adj[u]$ consists of all the vertices adjacent to u in G. The vertices in each adjacency list are typically stored in an arbitrary order. Let v be a vertex of a graph G. The number of edges incident with v is called the degree order of v and is denoted $deg(v)$. If all the vertices of G have degree r then G is called an r-regular graph.

Let $U \subseteq V(G)$ be a non-empty subset of the vertex set of a graph G. The subgraph $< U >$ induced by U is the graph having vertex set U and whose edge set consists of those edges of G incident with two elements of U.

Definition 3.2.

A ***path*** *in a graph is a sequence of vertices in which each successive vertex (after the first) is adjacent to its predecessor in the path. In a simple path, the vertices and edges are distinct. A path graph with n vertices is a graph consisting of a single path and is denoted* P_n.

Definition 3.3.

A ***cycle*** *is a path that is simple except that the initial vertex and the final vertex are the same. A cycle graph is a graph consisting of a single cycle, and denoted* C_n.

Definition 3.4.

A graph is a ***connected graph*** *if there is a path from every vertex to every other vertex in the graph. A graph that is not connected consists of a set of connected components, which are maximal connected subgraphs.*

Definition 3.5.

A ***tree*** *is a graph in which any two vertices are connected by a single path. A spanning tree of a graph, G, is a set of* $|V| - 1$ *edges that connect all vertices of the graph.*

There are certain classes of graphs that occur so often that they deserve special mention and in some cases special notation. We describe here the most prominent of these.

1. A graph G is complete -denoted K_n - if every two of its vertices are adjacent.

2. A graph is regular of degree r if for each vertex v of G, $deg(v) = r$, such graphs are called $r-$regular. Therefore a complete (n, m) graph is a regular graph of degree $n-1$ having $m = n(n-1)/2$ edges.

3. A graph G is $n-$partite, $n \geq 1$, if it is possible to partition $V(G)$ into n disjoint subsets $V_1, V_2, ..., V_n$ (called partite sets) such that every element of $E(G)$ joins a vertex of V_i to a vertex of V_j, $i \neq j$. For $n = 2$, such graphs are called bipartite graphs. A star graph is a bipartite graph with two partite sets V_1 and V_2 having the additional property that $|V_1| = 1$.

4. A complete $n-$partite graph G is an $n-$partite graph with partite sets $V_1, V_2, ..., V_n$ having the additional property that if $u \in V_i$ and $v \in V_j$, $i \neq j$, then $uv \in E$. If $n = 2$, such graphs are called complete bipartite graphs.

5. The line graph $L(G)$ of G is a graph whose vertices can be put one-to-one correspondence with the edges of G in such a way that two vertices of $L(G)$ are adjacent if and only if the corresponding edges of G are adjacent.

Graph models where we associate weights or costs with each edge are called for in many applications. In an electric circuit where edges represent wires, the weights might represent the length of the wire, its cost, or the time that it takes a signal to propagate through it. In an airline map where edges represent flight routes, these weights might represent distances or fares. In a job-scheduling

problem, weights might represent time or the cost of performing tasks or of waiting for tasks to be performed. In the following we will give a definition for an important kind of graph.

The problem of finding the minimum spanning tree of an arbitrary weighted undirected graph has numerous important applications, and algorithms to solve it have been known since at least 1920s. However the problem has been solved only in 1926 by Otakar Boruvka [22].

Definition 3.6 (Minimum Spanning Tree).

A minimum spanning tree (MST) of a weighted graph is a spanning tree whose weight (the sum of the weights of its edges) is not larger than the weight of any other spanning tree.

The following two properties are very closed to the minimum spanning tree problem:

* *Cycle property:* For any cycle C in a graph G, the heaviest edge in C does not appear in the minimum spanning tree.

* *Cut property:* For any proper nonempty subset X of the vertices, the lightest edge with exactly one endpoint in X belongs to the minimum spanning tree.

Part I

The Minimum Spanning Tree Problem

Chapter 4

Review of the Minimum Spanning Tree Algorithms

4.1 Introduction

Computing the minimum spanning tree of an undirected graph is one of the fundamental computational problems. In this chapter, we will present some of the sequential minimum spanning tree algorithms that help us to describe our parallel minimum spanning tree algorithm.

4.2 Why Minimum Spanning Tree ?

The minimum spanning tree problem is perhaps the simplest, and certainly one of the most central points in the field of combinatorial problems. It is useful in constructing networks, by describing the way to connect a set of sites using the smallest amount of communication lines. The minimum spanning tree problem

arises in a number of other applications, such as, computer networks, television cables, etc. It is the mother of all network design problems.

The minimum spanning trees prove important for several reasons:

i) They can be computed quickly and easily, and they create a sparse subgraph that reflects a lot about the original graph.

ii) They provide a way to identify clusters in sets of points. Deleting the long edges from a minimum spanning tree leaves connected components that define natural clusters in the data set.

iii) They can be used to give approximate solutions to hard problems such as Steiner tree and traveling salesman.

4.2.1 History

The algorithmic issues of the minimum spanning tree MST (or T_{min}) have a rich history. It has engaged researchers at least from the 1920 motivated by interest in the MST problem's theoretical structure as well as its practical relevance. We begin our short history of the MST problem by presenting below three classical serial algorithms that have played a central role in the history of the problem.

4.2.2 Three Classical Serial Algorithms

I- **Boruvka's algorithm**

Boruvka [22] is the first one who described the first fully realized minimum spanning tree algorithm. The same algorithm was rediscovered by a number of other authors, for example Gustave Choquet [14], G. Sollin[27]. We simply quote below one of the other formulations of the Boruvka's algorithm which was translated by Graham and Hell, and reported in 1950 by a research group based in Wroclaw (Poland).

"Here is our method of constructing a spanning tree with the points of a given set Z. Let us join, by a segment, each point to the point nearest to it; these segments will be called Connections of the First Order. They form one or more connected polygonal lines (subtrees) which are the connections of the points of the certain disjoint subsets of Z. These subsets will be called Groups of the First Order. Let us join each such group with the group nearest to it (by distance between groups one understands, of course, the smallest pairwise distance between their points), by a segment realizing their distance, which we shall call a Connection of the Second Order. We proceed this way, using connections of higher and higher order, until we obtain a connected polygonal line joining all the points of the set Z."

Given a weighted, undirected graph G, the algorithm builds the T_{min} by adding edges to a spreading forest of subtrees $T_{Boruvka}$, but it does so in stages, adding several T_{min} edges at each stage. At each stage the algorithm finds the shortest edge that connects each $T_{Boruvka}$ subtree with a different one, then add all such edges to the T_{min}. It constructs a spanning tree in iterations composed of the following steps.

Boruvka's algorithm

1: $T_{Boruvka} \longleftarrow \phi$
2: $L \longleftarrow \phi$
3: **for** each $v \in V(G)$ **do**
4: $T_v \longleftarrow Make-Tree(v)$ produces n trees of a single vertex
5: $T_v \longleftarrow$ unmark
6: $L \longleftarrow T_v$ make a list L of n unmarked trees
7: **end for**
8: **while** L has unmarked tree **do**
9: **for** each unmarked tree $T_v \in L$ **do**
10: **if** the number of edges with one endpoint in T_v is zero **then**
11: $T_v \longleftarrow$ mark
12: **else**
13: Select the lightest edge $e = (v, v')$
14: $T_{Boruvka} \longleftarrow e = (v, v')$
15: $T_{vv'} \longleftarrow$ Union($T_v, T_{v'}$)
16: $T_{vv'}$ shrinks into a single vertex.
17: Eliminate loops, and all parallel edges except the lightest-edges in $T_{vv'}$.
18: **end if**
19: **end for**
20: **end while**

Algorithm 1: Boruvka's algorithm

This implementation of Boruvka's MST algorithm initializes as shown in lines $1-5$ the empty set $T_{Boruvka}$ and creates n unmarked trees T_v, where each one contains exactly one vertex, and makes a list L of n unmarked trees in line 6. Each iteration of the while loop (line 8) checks all remaining edges, and chooses the lightest edge which connects current disjoint subtrees. At the end of each iteration, each component is united with its nearest neighbor (they form together a single vertex) and the nearest-neighbor edges added to the set $T_{Boruvka}$. Lines $10-11$: If the input graph has more than one component, then a minimum spanning forest consists of MSTs of each connected components of the original graph, where each marked tree in line

11 constructs a minimum spanning tree of the component that contains it. It is a generalization of the idea of MST. Line 15: The number of trees in the list L decreases by one in each iteration. Line 17: If there are edges connecting two vertices in the same tree they are discarded. Multiple edges are removed such that only the lightest edge remains between a pair of vertices. Boruvka's algorithm is guaranteed to work correctly only if all the edge costs are distinct. If edge weights are not distinct, we can make them distinct by numbering the edges and breaking weight-ties according to the numbers, so, assume for simplicity that all edge weights are distinct.

The algorithm seems to be the most efficient implementation. It can be implemented so that an iteration in which the graph has n vertices and m edges takes $O(n + m)$ sequential time. Furthermore, the number of vertices of the graph at the $(i + 1)$st iteration is at most half of the number of vertices at the ith iteration. Hence, the number of iterations is at most $\log n$, yielding a total running time of $O(m \log n)$. Finally, if the input graph has n isolated vertices, then the while loop would do only one iteration and in this while loop only lines $9 - 11$ are executed. With this kind of graphs the running time per an iteration is $O(n)$ and the total running time of the algorithm is $O(n)$.

Two classical algorithms efficiently find minimum spanning trees, namely Prim's and Kruskal's. Brief overviews of both algorithms are given below.

II- **Prim's algorithm:**

Prim's algorithm manages a set T_{prim} that is always a subset of some mini-

mum spanning tree. The subtree T_{prim} starts with an arbitrary root vertex r and grows it until a minimum spanning tree is obtained. At each step a vertex v not in the tree, connected by the smallest possible cost edge e to the subtree already built, is added; such an edge e is called a *safe edge* for T_{prim}.

The key to implement Prim's algorithm efficiently is to make it easy to select a new edge to be added to the tree formed by the edges in T_{prim}. The connected graph G and the root vertex r of T_{prim} are inputs to the algorithm. During execution of the algorithm, all vertices that are not in the tree reside in a priority queue Q based on a key field. For each vertex v, $key[v]$ is the minimum weight of any edge connecting v to a vertex in the tree; by convention $key[v] = \infty$ if there is no such edge. The field $\pi[v]$ gives the parent of v in the tree. During the algorithm, the subtree T_{prim} is kept implicitly as

$$T_{prim} = \{(v, \pi[v]) : \ v \ \in V - \{r\} - Q\}.$$

When the algorithm terminates, the priority queue Q is empty; the minimum spanning tree T_{min} for G is thus

$$T_{min} = \{(v, \pi[v]) : \ v \ \in V - \{r\}\}$$

.

Lines $1 - 4$ in the above implementation of Prim's algorithm initialize the priority queue Q to contain all the vertices and set the key of each vertex to ∞, except for the root r, which is set to zero. Line 5 initializes $\pi[r]$ to

Prim's algorithm

```
1:  Q ⟵ V(G)
2:  for each u ∈ Q do
3:     key[u] ⟵ ∞
4:     key[r] ⟵ 0;
5:     π[r] ⟵ NIL
6:  end for
7:  while Q ≠ φ do
8:     u ⟵ Extract − Min(Q)
9:     for v ∈ Adj[u] do
10:       if v ∈ Q and w(u, v) < key[ v ] then
11:          π[v] ⟵ u
12:          key [v] ⟵ w(u, v)
13:       end if
14:    end for
15: end while
```

Algorithm 2: Prim's Algorithm

NIL, since the root r has no parent. The while loop in lines $6-11$ grows the subtree T_{prim} by identifying the vertex $u \in Q$ (with minimum key) incident on a light edge crossing the cut between the vertices on T_{prim} and vertices not on T_{prim}. Removing u from the set Q adds it to the set $V - Q$ of the vertices in the subtree T_{prim}. At the end of the loop the algorithm updates the key and π fields of every vertex v adjacent to u but not in the tree. The updating maintains the invariants that $key[v] = w(v,\ \pi[v])$ and that $(v,\ \pi[v])$ is a light edge connecting v to some vertex in the subtree T_{prim}.

Prim's algorithm clearly creates a spanning tree, because no cycle can be introduced by adding edges between tree and non-tree vertices. *However, why should it be of minimum weight over all spanning trees.?*

Suppose that there existed a graph G for which the algorithm did not return a minimum spanning tree. Since we have built the tree incrementally, this

means that there must have been some particular instant where we went wrong. Before we inserted edge (x, y), T_{prim} consisted of a set of edges that was a subtree of a minimum spanning tree T_{min}, but choosing edge (x, y) took us away from a minimum spanning tree. But how could it happen.? There must be a path P from x to y in T_{min}, using an edge (v_1, v_2), where v_1 is in T_{prim} but v_2 is not. This edge (v_1, v_2) must have weight at least that of (x, y), or else the algorithm would have selected it instead of (x, y) when it had the chance. Inserting (x, y) and deleting (v_1, v_2) from T_{min} leaves a spanning tree not larger than before, meaning that the algorithm could not have made a fatal mistake in selecting edge (x, y). Therefore, by contradiction, Prim's algorithm has to construct a minimum spanning tree. The performance of the algorithm depends on how we implement the priority queue Q.

If Q is implemented as a Fibonacci-Heap (simply Fib-Heap) [28] the n elements can be organized into Fib-Heap in $O(1)$, and an Extract-Min operation in $O(logn)$ time and a Decrease-Key operation in $O(1)$ time. Therefore, if we use a Fib-Heap to implement the priority queue Q, the running time of Prim's algorithm will be $O(m + n \log n)$

III- **Kruskal's algorithm**

Kruskal's algorithm is an alternative approach to find minimum spanning trees that is more efficient on sparse graphs. Like Prim's, Kruskal's algorithm is greedy; unlike Prim's, it does not start with a particular vertex. This means it finds a subset of the edges that forms a tree that includes

every vertex, where the total weight of all the edges in the tree is minimized. If the graph is not connected, then it finds a minimum spanning forest (a minimum spanning tree for each connected component). The algorithm starts with a forest which consists of n trees. Each one tree, consists only of one node and nothing else. In every step of the algorithm, two different trees of this forest are connected to a bigger tree. Therefore, the algorithm keeps having smaller and bigger trees in the forest until it ends up in a tree which is the minimum spanning tree . In every step the algorithm chooses the edge with the minimal cost, which means that we are still under greedy policy. If the chosen edge connects nodes which belong to the same tree the edge is rejected, and not examined again because it could produce a circle which will destroy the tree. Either this edge or the next one in order of least cost will connect nodes of different trees, the algorithm inserts it to connect two small trees into a bigger one.

Kruskal's Algorithm

1: $T_{Kruskal} \longleftarrow \phi$
2: **for** each vertex $v \in V[G]$ **do**
3: Make-Set(v)
4: **end for**
5: $Q \longleftarrow E(G)$
6: **for** $i = 0;\ i < n-1;\ i{+}{+}$ **do**
7: $e = (u, v) \longleftarrow Extract - Min(Q)$
8: **if** $Find - Set(u) \neq Find - Set(v)$ **then**
9: $T_{Kruskal} \longleftarrow T_{Kruskal} \cup (u, v)$
10: Union (u, v)
11: **end if**
12: **end for**
13: return $T_{Kruskal}$

Algorithm 3: Kruskal's Algorithm

The algorithm initially creates (in lines $1-4$) an empty set $T_{Kruskal}$ and a forest of n trees, where each vertex in the graph is a separate tree. Line 5 creates a priority queue Q containing all the edges in the graph. At the end of the algorithm, while Q is nonempty, remove an edge with minimum weight from Q. If that edge connects two different trees, then add it to the forest, combining two trees into a single tree, otherwise discard that edge. At the termination of the algorithm, the forest $T_{Kruskal}$ has only one component and forms a minimum spanning tree T_{min} of the graph. But *why must this be a minimum spanning tree?* Suppose it was not. As with the correctness of Prim's algorithm, there must be some graph for which it fails, and in particular there must be a single edge (x, y) whose insertion first prevented the tree $T_{Kruskal}$ from being a minimum spanning tree T_{min}. Inserting edge (x, y) in T_{min} will create a cycle with the path from x to y. Since x and y were in different components at the time of inserting (x, y), at least one edge on this path (v_1, v_2) would have been considered by Kruskal's algorithm after (x, y) was. But this means that $w(v_1, v_2) > w(x, y)$, so exchanging the two edges yields a tree of weight at most T_{min} . Therefore, we could not have made a mistake in selecting (x, y), and the correctness follows.

The above algorithm can be shown to run in $O(m \log m)$ time, where m is the number of edges in the graph.

It should be noted that all three algorithms initialize the spanning forest (a spanning forest of a graph G consists of subtrees of G each of which called frag-

ment) to contain each vertex in $V(G)$ as a one-vertex tree. Selectively, they add edges to the forest until it becomes a spanning tree of G. All three algorithms differ in the criterion used to select the next edge or edges to be added in each iteration. All three algorithms compute the minimum spanning tree of any connected graph. If the original graph is not connected, Prim's algorithm will find a minimum spanning tree in the first component, then it will fail to add any more edges. Boruvka's and Kruskal's algorithm will find a minimum spanning tree for each component.

It should be noted also that Prim's algorithm is faster on dense graphs, while Boruvka's and Kruskal's are faster on sparse graphs.

Kruskal's algorithm is a greedy algorithm because it considers each edge $e \in E(G)$ in a non-decreasing order according to its weight and immediately adds an appropriate edge to $T_{Kruskal}$.

In greedy algorithms, we make the decision of what to do next by selecting the best local option from all available choices regardless of the global structure. Since, Prim's minimum spanning tree algorithm is greedy. It starts from one vertex and grows the rest of the tree one edge at a time.

Minimum spanning tree algorithms have an interpretation in terms of matroids. The matroid $M = (S, \beth)$ consists of a ground set S and a nonempty family set $\beth$ of subsets of S (called "independent sets") such that:

* The empty set is in $\beth$
* Every proper subset of every set in $\beth$ is also in $\beth$
* For every subset in $\beth$ we can replace one element with another selection from S and get a new subset which is also in $\beth$.

A matroid $M = (S, \beth)$ is weighted if there is an associated weight function w that assigns strictly positive weight $w(x)$ to each element $x \in S$. The weight function w extends to subsets of S. The weight of a subset A is the sum of the weights of the elements in A. The natural question arises on a weighted matroid is: *What is the maximum weight independent set?*

The graphic matroid defined in terms of a given undirected graph $G = (V, E)$ denoted $M_G = (S_G, \beth_G)$, for which the set S_G is defined to be $E(G)$. If A is a subset of $E(G)$, then $A \in \beth_G$ if and only if A is acyclic. The elements of the graphic matroid are the edges of the graph. The independent sets S_G are the forests. Consider an undirected weighted graph $G = (V, E)$ such that $w(e)$ is a positive weight of e. In order to view the minimum spanning tree as a problem of finding an optimal subset of a matroid, we consider the weighted graphic matroid $M_G = (S_G, \beth_G)$ with weight function w', where $w'(e) = w_0 - w(e)$ and w_0 is larger than the maximum weight of any edge. In this weighted matroid, all weights are positive and a subset that is independent and has maximum possible weight is an optimal minimum spanning tree in G.

Let us say that the greedy algorithm optimizes all linear cost function over a hereditary systems $(S, \beth)$ if and only if $(S, \beth)$ is a matroid [10]

4.2.3 Other Algorithms

We give now the full description of the sequential algorithm introduced by Michael L.Fredman and Robert E. Tarjan (FT-algorithm) because it may be useful as a presentation of my parallel minimum spanning tree algorithm. In the beginning the FT-algorithm initializes the forest to contain each of the n vertices of G as one-vertex unmarked tree. The idea is to grow a single tree only until its heap of neighboring vertices exceeds a certain critical size k. Then the algorithm starts from a new vertex and grows another tree, again stopping when the heap gets too large. It continues in this way until every vertex is in a tree. Then the algorithm condenses every tree into a single super-vertex and begins a new pass of the same kind over the condensed graph. After a sufficient number of passes, only one super-vertex will remain, and by expanding the super-vertices, a minimum spanning tree was extracted.

The next lines describe the basic mode of the operations in a single pass of the algorithm (see FT-Algorithm). The pass begins with a forest of previously grown trees, which are called *old trees*, defined by the edges so far added to the forest.

The time for the clean up (step 5) and other initialization is $O(m)$. If t is the number of old trees, the total time for the construction of new trees is

1: $T_{Fre} \longleftarrow \phi$
2: **for** each old tree T **do**
3: Number T_i consecutively from one
4: Assign for each vertex $v \in T_i$ the number of T_i.
5: Discard every edge connecting two vertices in the same old tree, and all except the *best-edge* connecting each pair of trees.
6: Construct a list for each old tree of edges with one endpoint in T.
7: $key(\ T_i\ ) \longleftarrow \infty$
8: $T_i \longleftarrow unmarked$
9: **end for**
10: Create an empty heap H_i.
11: **while** unmarked old tree **do**
12: Select unmarked old tree T_i
13: Insert it as an item into the heap H_i with $key(\ T_i\ ) = 0$.
14: **repeat**
15: Delete an old tree T_v with minimum key.
16: $key(\ T_v\ ) \longleftarrow -\infty$.
17: Add $e(T_v)$ $(\ T_v \neq T_i)$to the forest T_{Fre}
18: **if** T_v is marked. **then**
19: empty the heap
20: $key(T_v) \longleftarrow \infty$
21: **else**
22: $T_v \longleftarrow mark.$
23: **for** each edge (v, w) s.t. $v \in T_v$ **do**
24: **if** $c(v, w) < key(\ T_w\ )$ **then**
25: $e(\ T_w\ ) = (v, w)$.
26: **if** $key(\ T_w\ ) = \infty$ **then**
27: Insert T_w in the heap H_i
28: **else**
29: $Decrease - Key\{H_i, key(T_w), c(v, w)\}$.
30: **end if**
31: **end if**
32: **end for**
33: **end if**
34: **until** $|\ H_i| > k$ or $H_i = \phi$
35: **end while**

Algorithm 4: FT- Algorithm

$O(m + t \log k)$: the algorithm makes at most t *delete minimum* operations, each on a heap of size $\leq\ k$, and $O(m)$ other operations, none of which is a deletion. Let us choose $k = 2^{(2m/t)}$, where m is the original number of edges in the graph and t is the number of trees before the pass. The value of k increases from pass to pass as the number of trees decreases. With this choice of k, the running time per pass is $O(m)$. As Fredman and Tarjan proved in [28] the number of passes is at most $min\{i|\ \log^{(i)} n \leq\ 2m/n + 1\} = \beta(m, n) + O(1)$ where $\beta(m, n) = min\{i|\ \log^{(i)} n \leq\ m/n\}$. The FT- algorithm can be shown to run in $O(m\beta(n, m))$ time, where n is the number of vertices and m is the number of edges in the given graph.

A faster algorithm was found by H. N. Gabow, Z. Galil, andT. H. Spencer [13]. They described an algorithm similar to that one of Boruvka's algorithm. Their algorithm has a running time of $O(m \log \beta(m, n))$ on a graph of n vertices and m edges. B. Chazelle [3] gave a deterministic algorithm for computing a minimum spanning tree. Its complexity is $O(m\alpha \log \alpha)$, where $\alpha = \alpha(m, n)$ is a functional inverse of Ackermann's function. This time has been improved by B. Chazelle [4] to time complexity $O(m\ \alpha(m, n))$. Recently S. Pettie and V. Ramachandran [23] presented an optimal similar algorithm with the same running time, which gives an alternate exposition of the $O(m\alpha(m, n))$ result. As I knew, this is the smallest time bound for the sequential MST problem to date. On the other hand D. R. Karger, P. N. Klein and R. E. Tarjan [18] in 1995 presented a randomized linear-time algorithm. Their algorithm is recursive. The algorithm generates two subproblems, but with high probability the total size of

these subproblems is at most a constant fraction (less than one) of the size of the original problem. The algorithm relies on a random-sampling step to discard edges which can not be in the minimum spanning tree.

Chapter 5

A New Parallel Minimum Spanning Tree Algorithm

5.1 Introduction

The problem of determining a minimum spanning tree in parallel has been the focus of much research. Here is a brief summary of related results. In 1979 D. H. Chandra, and D. V. Sarwate [2] presented a parallel deterministic algorithm for graphs with n vertices and m edges, that runs in $O(\log^2 n)$ time using $n^2/\log n$ processors on the CREW model. In 1982, F. Chin, J. Lam, and I. Chen [20] gave a parallel deterministic algorithm, that runs in $O(\log^2 n)$ time using $n^2/\log^2 n$ processors Thus their algorithm achieves linear speed-up when the input graph is a complete graph. However, it is not very work-efficient for spare graphs. In 1982 Y. Shiloach and U. Vishkin [26] improved the result to $O(\log n)$ time and $O(m+n)$ processors on the CRCW model R. Cole and U. Vishkin [9] presented

the best deterministic CRCW parallel MST and connectivity algorithms that require $O(\log\ n)$ time and $O((m+n)\ \alpha(m,n)/\log\ n)$ processors. Recently in 1999 K. W. Chong, Yijie Han, and Tak W. Lam [5] presented a new approach for finding the minimum spanning trees that runs in $O(\log n)$ time using $n+m$ processors on EREW PRAM. Thus their algorithm as R. Cole and U. Vishkin algorithm all use super-linear work. There are somewhat simpler logarithmic-time linear expected work randomized minimum spanning tree algorithms, which have been successfully analyzed by R. Cole, P. N. Klein and R. E. Tarjan [8]. They improved the running time $O(2^{\log^* n}\ \log n)$ of their previous work [7] to $O(\log n)$. Their algorithms based on the sequential randomized linear-time algorithm to find MST which has been discovered by P. N. Klein, D. R. Karger and R. E. Tarjan [18]

5.2 Some Basic Techniques

We introduce here some techniques that will be used as a general basis for our parallel algorithm.

5.2.1 Prefix Sum

We discuss an optimal prefix sum algorithm on the EREW PRAM in this part (see [17] (p.44)). We begin by giving the definition of the prefix sum defined as follows:

Definition 5.1.

Given an array of numbers $A[1, ..., n]$, output a new array $S[1, ..., n]$ in which $S[i] = \sum_1^i A[i]$.

For example, given the operation + and the array of integers $A[1, ..., 5] = \{3, 2, 5, 1, 2\}$, the prefix sums of the array are $S[1, ..., 5] = \{3, 5, 10, 11, 13\}$. Note that we can define the "prefix sum" problem over any binary, associative operation, not only addition. For example, if the operation is minimum, every element in the output array will contain the minimum of all the elements to its left in the input array. In the numerical example above, the result of prefix sum with operation minimum would be $\{3, 2, 2, 1, 1\}$. There is a parallel algorithm for solving this problem in $O(\log n)$ time using n processors.

* **Input:** n elements $\{x_1, x_2, \ldots, x_n\}$ placed in memory cells $A[1..n]$, such that $A[i] = x_i$
* **Output:** The prefix sum s_i, for $1 \leq i \leq n$.

PREFIX(A, n)
1: **for** $i = 1$ *to* $\lceil n/2 \rceil$ in parallel **do**
2: Set $S[i] \longleftarrow A[2i-1] \oplus A[2i]$
3: **end for**
4: Call **PREFIX**$(S, \lceil n/2 \rceil)$, and store them in $Z[1], \ldots, Z[n/2]$.
5: **for** $1 \leq i \leq n$ in parallel **do**
6: $\{ i$ even : Set $s[i] = Z[i/2]$
$i = 1$: Set $s[1] = x_1$
i *odd* > 1 : Set $s[i] = Z[(i-1)/2] \oplus x_i\}$
7: **end for**

Algorithm 5: PREFIX-SUM Algorithm

The above algorithm works in $O(\log n)$ time using n EREW PRAM processors.

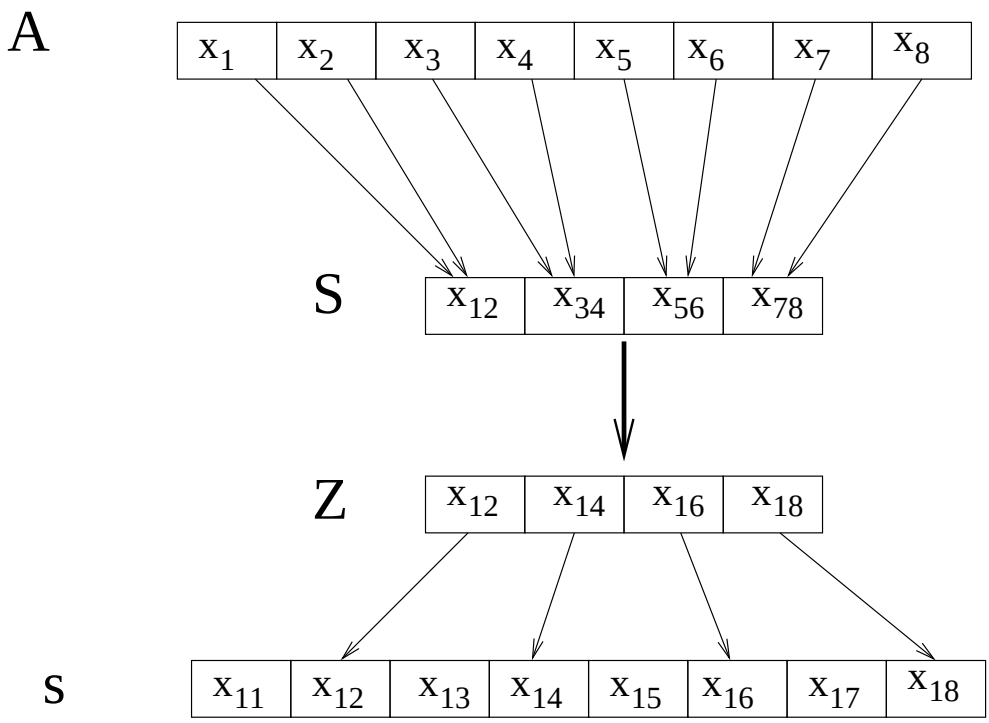


Figure 5.1: Prefix sum of eight elements. Element x_{ij} determines the sum $x_i \oplus \ldots \oplus x_j$.

5.2.2 Lexicographic Order

Let A be a finite alphabet and let $\prec$ be an ordering of A. Then the corresponding lexicographic ordering $<$ is defined on A^n, as follows. Let $u, v \in A^n$, where $u = a_1 \dots a_n,\ \ v = b_1 \dots b_n$, with $a_i,\ b_i \in A$. Then $u < v$ if and only if either:

1- $a_1 < b_1$, or

2- There exists $j \in \{1, \dots, n\}$ such that $a_i = b_i$ for $i < j$ but $a_j \prec b_j$.

A^n denotes the n-times Cartesian product of the set A. The elements of the set A^n are called words with fixed length n.

5.2.3 Parallel Radix Sort

We discuss how to design a parallel radix sort algorithm by solving problem 4.16 page 189 of "Parallel Computation: Models and Methods" [1].[1] We shall notice that it is in fact a parallel algorithm for sorting integers by bucketing, which essentially is a radix sort. To sort the sequence $A = \{\ a_1,\ a_2,\ \dots,\ a_n\ \}$ of b-bit integers, two arrays of n entries are created in the shared memory of an n-processor PRAM. These two arrays are called *bucket* 0 and *bucket* 1, respectively. The algorithm consists of b iterations. At the beginning of each iteration, all positions of *bucket* 0 and *bucket* 1 are set to 0 (this means *bucket* $0[n] =$*bucket* $1[n] = \{0\}$); this requires constant time (because b is constant). During iteration

[1] Its solution is given by S. Akl, Parallel Computation: Models and Methods, Solutions Manual, Prentice Hall, 1997.

i, $1 \leq i \leq b$, element a_j of A, where:

$$a_j = a_j(b)a_j(b-1)\ldots a_j(1),$$

is examined by processor P_j which places a 1 in position j of either *bucket* 0 or *bucket* 1 depending on whether $a_j(i) = 0$ or 1, respectively. This step also requires constant time. Now the values in *bucket* 0 followed by those in *bucket* 1 form a sequence of $0's$ and $1's$ of length $2n$. The prefix sums $\{s_1, s_2, \ldots, s_{2n}\}$ of this sequence are now computed. This requires $O(\log n)$ time. Finally, element a_j is placed by P_j in position s_j or s_{j+n} of A (depending on whether *bucket* 0 or *bucket* 1 contains a 1 in position j) in constant time, concluding this iteration. The algorithm thus runs in $O(b \log n)$ time and has a cost of $O(bn \log n)$. The cost can be reduced to $O(bn)$, which is optimal, by using $O(n/\log n)$ processors. The time of the prefix computation is $O(\log n)$. The time for the other tasks is $O(1)$. The overall running time of the parallel radix sort algorithm is $O(\log n)$ using $O(n/\log n)$ EREW-PRAM processors.

5.2.4 The Heap Data Structure

To implement our algorithm efficiently, we need a data structure that will store the vertices of V in a way that allows the vertex joined by the minimum cost edge incident to the tree to be selected quickly. A heap is a data structure consisting of a collection of items, each having a key. The basic operations on a heap are:

* **Make heap**: Return a new, empty heap H.

* **Insert**(i, k, H): Add item i to heap H using k as the key value.

* **Delete-Min**(H): Delete and return an item of minimum key from H.

* **Change-Key**(i, k, H): Change the key of item i in heap H to k.

* **Key**(i, H): Return the key value for item i.

We use an extension of binomial queues called *Fibonacci heaps* abbreviated **Fib-heaps**. Fib-heaps support **Delete-Min** in $O(\log n)$ time, and all the other heap operations, in particular **Change-Key**, in $O(1)$ time. For situations in which the number of deletions is small compared to the total number of operations, Fib-heaps are asymptotically faster than binomial queues[28].

5.2.5 The Model

We assume an EREW PRAM model employs $(n + m)$ processors where n is the number of vertices and m is the number of edges in the given graph G, each processor able to perform the usual computation of a sequential machine using some fixed amount of local memory. The processors communicate through a shared global memory to which all are connected. The essential assumption used in our analysis is that a processor can access data computed by another processor and stored in the shared memory in constant time.

5.3 The Proposed Parallel Algorithm

This section presents the proposed parallel algorithm implementation of the minimum spanning tree technique. The algorithm is similar to the sequential

algorithm of M. Fredman and R. Tarjan (FT-Algorithm), which has been discussed in section (4.2.3). However, at the beginning of the algorithm, we are using a small value of the parameter k, which is smaller than that one in the FT-Algorithm. The value of k does not depend on the number of trees in each pass of the algorithm. We also give another simple procedure to construct the new tree.

The algorithm is divided into $O(\log n/\log\log n)$ passes. In each pass, the algorithm reduces the number of trees t by the fraction $1/k$. In other words, consider the pass that begins with t trees and m' edges the number of trees t' remaining after the pass satisfies $t' \leq 2m'/k$, where $k = 2\lceil \log m \rceil$. We shall prove that, the total running time for each pass is bounded by $O(\log k) = O(\log\log m)$. This algorithm runs on EREW PRAM on a graph with n vertices and m edges in $O(\log m)$ time using $(n + m)$ processors.

5.3.1 Assumptions and Definitions

Given a graph G with n vertices and m edges. We assume that the input graph G is given in the form of adjacency lists, where every vertex v has a linked list $L(v)$ of incident edges (v, w). For instance, if $e = (u, v)$ is an edge in G, then e appears in the adjacency list of u and v. We call each copy of e as the mate of the other. In order to differentiate between them we use the notations (u, v) and (v, u) to indicate that the edge originates from u and v respectively. The weight of e, which can be any integer value, is denoted $w(e)$ or $W(u, v)$. The proposed algorithm can be implemented for a graph in which the weights of all edges are distinct, or there are some different edges that have the same weights.

We therefore say that the input graph G may have a unique minimum spanning tree, or more than one minimum spanning tree. The minimum spanning tree will be referred to as T_G throughout this chapter.

We also assume that G is an undirected connected graph and consists of only one component. If there is more than one component, then every connected component can be found by using any fast parallel algorithm for finding the connected components in the graph G. Then the proposed algorithm can run for every component in G. Hence we assume for simplicity and without loss of generality that the input graph has only one connected component.

Let $F = \{T_1, T_2, \ldots, T_t\}$ be an arbitrary set of subtrees of G. If a tree T_i contains no edge incidents on a vertex v, then v itself forms a tree. Consider any edge $e = (u, v) \in G$ and tree $T_i \in F$. If both vertex u and vertex v belong to T_i then e is called an internal edge of T_i; if only one vertex of $\{u, v\}$ belongs to T_i, then e is called an external edge. F is said to be a $k-$forest if each tree $T_i \in F$ has at least k vertices. A tree $T_j \neq T_i$ is adjacent to T_i if there is an edge $e = (u, v)$, $u \in T_i$, $v \in T_j$. If T_j is adjacent to T_i, then the best edge from T_i to T_j is the minimum cost edge $e = (u, v)$, $u \in T_i$, $v \in T_j$. For every tree $T_i \in F$ the linked list of T_i is the set of all best edges from T_i to its adjacent trees T_j, and is written by $L(T_i)$. For each tree T_i, if e is the minimum weight external edge connecting a vertex in T_i to a vertex in T_j, then, the edge e belongs to T_G. If $e = (u,)$ is an external edge from T_i to T_j that is not a minimal weight external edge, then e is never an edge in T_G.

5.3.2 Description of the Algorithm

The algorithm runs for a number of passes. In the beginning each pass assigns all single trees white. Each pass creates an empty Fib-Heap for each single tree and inserts its linked list (the set of all best edges from T to its adjacent trees) as items in the heap with keys equal to the weight $w(e)$ of the edge. It then chooses the edge with the minimum weight and begins from the other end point of that edge. The pass grows a single white tree only until its heap of incident edges exceeds a certain critical size and assigned it white. The algorithm continues in this way until there is no white tree remaining and then condenses every tree into a single super-vertex. The algorithm performs the condensing implicitly and then begins a new pass of the same kind over the condensed graph. After a sufficient number of passes, only one super-vertex will remain. By expanding the super-vertex back into trees then a minimum spanning tree is remaining as shown in Fig.(5.2).

Figure (5.2) depicts the methodology for running of the proposed algorithm. The figure is not realistic for considering the number of passes, because the value of the parameter $k = 6$ is smaller than $2\lceil \log m \rceil = 2\lceil 4.7 \rceil = 10$. If $k = 10$, only one pass is needed to induce the minimum spanning tree of the graph G_0, which reduces the benefit of the example.

For the graph G_0 shown in Fig.(5.2a) three passes are needed in order to identify the minimum spanning tree. In the first pass the old tree is only one vertex. Every vertex v_i is assigned a processor p_i. The processor p_i creates

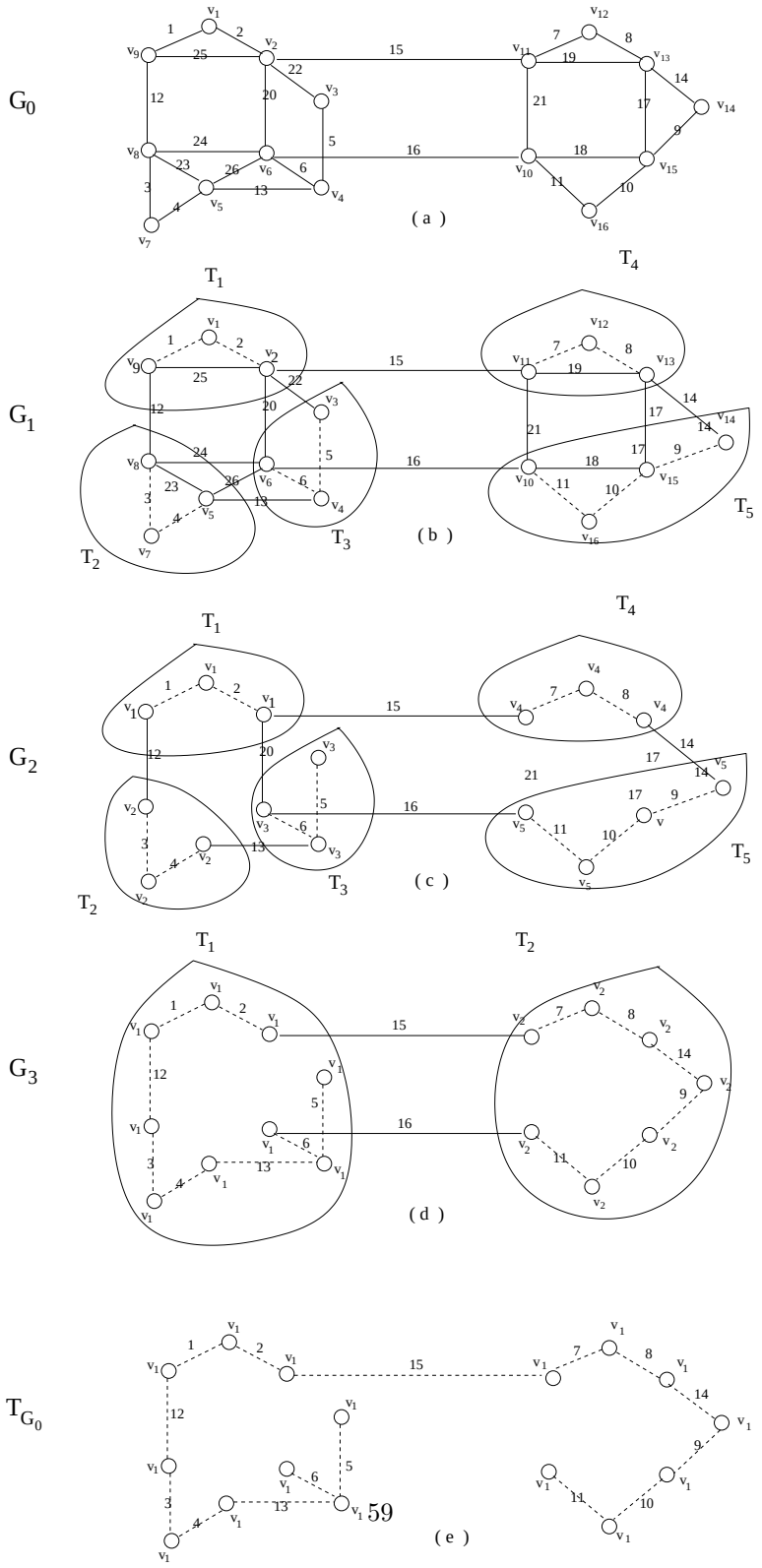


Figure 5.2: The execution of the proposed algorithm on a given graph G_0.(a) The situation just before the first pass of the algorithm. (b) The five trees $\{T_1, T_2, T_3, T_4, T_5\}$ are constructed after the first pass.(c) The five trees $\{T_1, T_2, T_3, T_4, T_5\}$ are constructed after the prune step.(d) The new growing trees are T_1 and T_2. (e) The situation after 3 successive passes.

the heap H_i and chooses the edge with the minimum weight to include it as a minimum spanning tree edge. From the other end point of this edge the processor repeats the same process until the heap size exceeds the parameter k, then the growth process finishes. The result of the first pass are the following five trees T_1, T_2, T_3, T_4, T_5 as shown in Fig.(5.2*b*), in which the internal edges are shown as dotted edges and their corresponding external edges are

$\{v_9v_8, v_2v_6, v_2v_3, v_2v_{11}\}$, $\{v_8v_9, v_8v_6, v_5v_6, v_5v_4\}$,

$\{v_4v_5, v_6v_5, v_6v_8, v_6v_2, v_3v_2, v_6v_{10}\}$,

$\{v_{11}v_2, v_{11}v_{10}, v_{13}v_{15}, v_{13}v_{14}\}$,

$\{v_{10}v_6, v_{10}v_{11}, v_{15}v_{13}, v_{14}v_{13}\}$

where the corresponding edge weights are

$\{12, 15, 20, 22\}$, $\{12, 13, 24, 26\}$,

$\{13, 16, 20, 22, 24, 26\}$,

$\{14, 15, 17, 21\}$, $\{14, 16, 17, 21\}$ respectively.

In the second pass the input trees are the five already computed trees $\{T_1, T_2, T_3, T_4, T_5\}$. We assign to each vertex in $V(G_0)$ the number of the tree that contains it. In the prune step of the pass removes un-dotted internal edges connecting two vertices have the same number and we identify the lightest edges of the weights in the corresponding linked lists, as shown in Fig.(5.2*c*). After the cleanup we construct the linked lists $L(T_1), L(T_2), L(T_3), L(T_4), L(T_5)$ where the edge weight for the edges in corresponding linked list are $\{12, 15, 20\}$, $\{12, 13\}$, $\{13, 16, 20\}$, $\{14, 15\}$, $\{14, 16\}$ respectively.

As it has happened in the first pass, we get the following new growing trees T_1 and T_2 with linked lists $L(T_1)$ and $L(T_2)$, which are adjacent by the multiple

edges having the weights 15 and 16 respectively. In the prune step of the last pass we identify the lightest edge of weight 15 as minimum spanning tree edge and delete the other heavy edge of weight 16. Finally, as shown in Fig.(5.2d) we get the minimum spanning tree T_{G_0}.

5.3.3 The Parallel MST Algorithm

The algorithm maintains a forest defined by the edges so far selected to be in the minimum spanning tree. It initializes the forest T such that it contains each of the n vertices of G as a one-vertex tree and maintains a key for measuring $w(e)$, which represents the tentative cost of incident edge e to T.

1: Form one trivial tree per each vertex v. Let n be the number of trees.
2: **for** each tree $v \in V$ **do**
3: Set $key(v) = \infty$.
4: Color each vertex v white.
5: **end for**

Algorithm 6: Procedure Initialization

The processor assignment for initialization procedure is to provide one processor to each vertex. A processor colors the vertex white and sets the key of the vertex to ∞, then this procedure takes $O(1)$ time and n processors. The main procedure of the MST algorithm is described as follows:

1: **for** $\log(m/\log\log m)$ times **do**
2: Call *Get-New-Tree*(T) procedure.
3: **end for**

Algorithm 7: MST Main procedure

In the first pass of the algorithm the input old tree will be considered as a single vertex. For each vertex v_i we assign one processor P_i and create the Fib-Heap H_i. We then insert the set of all edges incident with v_i in the heap with a key equal the weight of every edge. Since it is not expected that all vertex degrees will equal one, then we repeat the following step for at most k times:

Find a minimum cost edge with exactly one endpoint in the selected set of vertices (subtrees) and add it to the forest T_G; add its other endpoint to the selected set of vertices.

After the above process, we get the first set F of nontrivial subtrees of G with two non-empty sets of edges. The first of those are the internal edges (contain at least one edge), the second includes the external edges, which will be at most equal to ζ, where ζ refers to the number of end vertices in the non-trivial tree; it will be determined later. The end vertices may be incident to external or internal edges.

The forest $F = \{T_1, T_2, ..., T_{t_1}\}$ of subtrees of G are called the old trees. These old trees will be the input to the next pass of the algorithm in order to grow them to get other new trees which will be the old ones for the following pass.

The following is a description of a single pass (pass i) of the algorithm. The pass begins with a forest of previously grown trees (the old trees) defined by the edges so far added to the forest. The pass connects these old trees into new larger trees.

We start with the old trees by numbering it consecutively from one and assign to each vertex the number of the tree containing it. Each processor should keep its initial vertex. This allows us to refer to the trees by the numbers and directly access the old tree $T(v)$ that contains the vertex v.

Next we clean up the linked list of each old tree by removing every edge that connects any two vertices in the same old tree and all but a minimum-cost edge connecting each pair of old trees. A full description of the cleaning process using lexicographical sorting is given after the overview of the growing of a new tree process.

After cleaning up we construct a new edge list for each old tree. However since every old tree and all vertex incidents with its internal edges have the same number and are sorted lexicographically according to their end point then we can in constant time merge the linked list of all vertices which are contained in the current grown tree. We can merge the linked list of the old trees into a single list efficiently and the time does not depend on the length of the list. We use a technique introduced by Tarjan and Vishkin [29] and Chong, Han and Lam [6]. In our algorithm every pass grows the new tree T by replacing some old trees $\{T_i, T_{i+1}, ..., T_j\}$ by their union, this means that, $T = T_i \cup T_{i+1} \cup ... \cup T_j$. We shall describe the technique to merge the linked list of these old trees in a single list. Suppose the two old trees T_i and T_k with two linked lists $L(T_i)$ and $L(T_k)$ contain an edge (u, v) and its mate (v, u) respectively. The two linked lists can be combined by having the edge (u, v) and its mate (v, u) exchange

their successors. If every edge of T_i or its mate of T_j exchange their successors in their linked list $L(T_i)$ and $L(T_j)$ respectively, we will get a new combined list for T in $O(1)$ time (see Fig.(5.3)). The merging process does not terminate before merging the linked lists of all above old trees in a single one. However, the algorithm guarantees that the merging process will not fail because all the edges of T_i (or its mate) are included in the corresponding linked list.

In order to finish the growth process empty the heap and set the keys of all old trees with key equals to infinity.

1: Number the old trees consecutively starting from one and assign to each vertex the number of the tree that contains it.
2: Prune the linked list of each old tree.
3: For each old tree construct a list of edges that have one endpoint in T.
4: Every processor P_i calls the Grow-Step(T) procedure.
5: Finish the growth step by emptying the heap and set $key(T) = \infty$.

Algorithm 8: Get-New-Tree(T)

Prune the linked list

Discard every edge that connects two vertices in the same old tree as follows.

When the subroutine prune (step 2) considers an edge with the same number for its both two endpoints, it assigns this edge an internal (dead). Afterward sort the edges (external edges) that connect different old trees lexicographically according to their endpoints. Sorting can be performed in parallel by using the Parallel Radix Sort algorithm as described earlier. The algorithm sorts n elements in $O(\log n)$ time using $n/(\log n)$ EREW PRAM processors. In the sorted

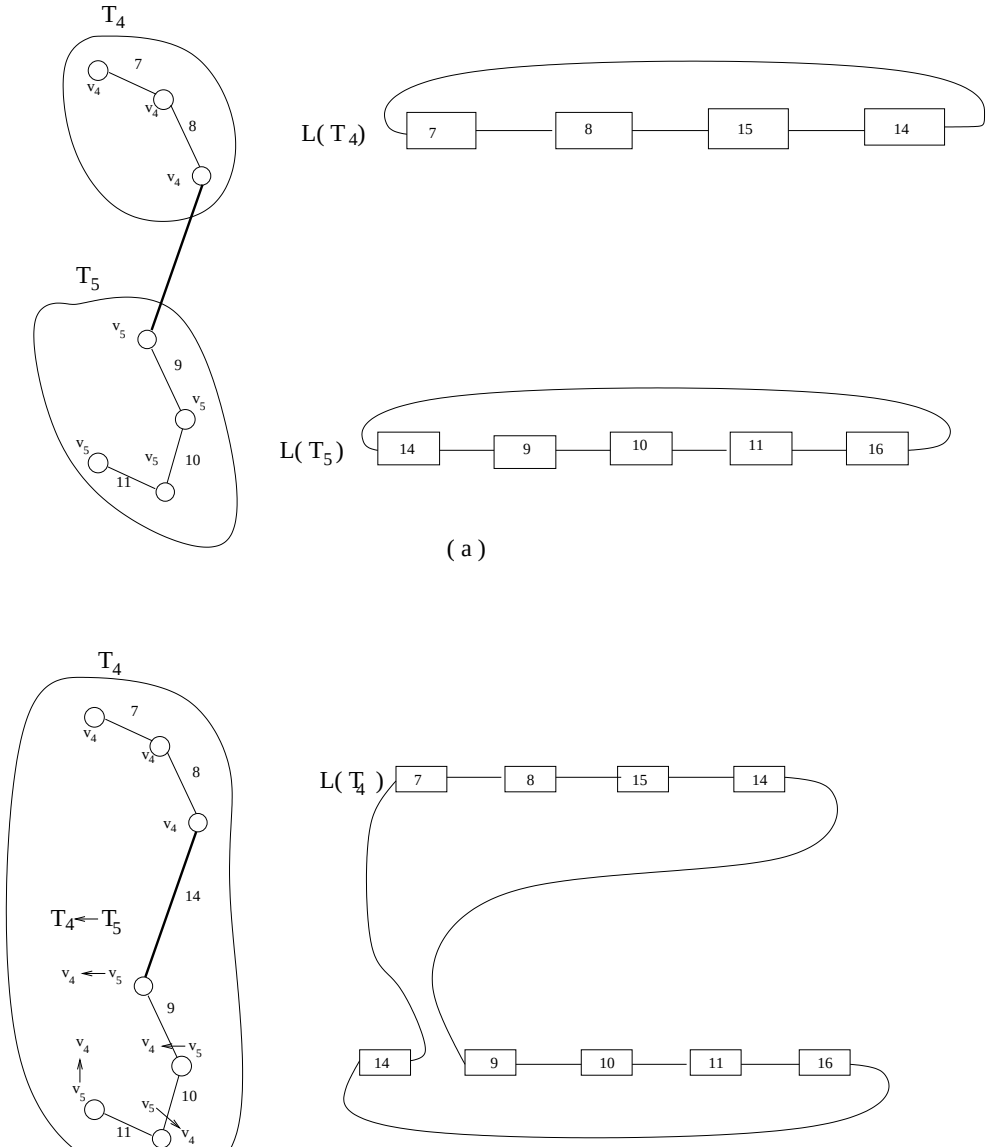


Figure 5.3: Merging a pair of linked lists $L(T_4)$ and $L(T_5)$ with respect to a common edge that has weight 14. This is the execution of the merging process on the two trees T_4 and T_5 from figure 5.2c. (a) The linked lists $L(T_4)$ and $L(T_5)$ before merging. (b) The final result of the join of $L(T_4)$ and $L(T_5)$ in a single bigger linked list named $L(T_4)$.

list, all multiple edges should end up in a sequence. Then, we save for each sequence of multiple (x, y) edges the minimum weight while the remaining multiple ones are deleted.

The running time of this step depends on the number of the external edges, which is greater than the size of the linked list of the tree. The following two lemmas help us to estimate the upper bound of this number.

Lemma 5.1.

The number of end vertices ζ in a tree T with $V_T = \{v_1, v_2, \ldots, v_n\}$ equals

$$2 + \sum_{deg(v) \geq 2} (deg(v) - 2)$$

.

Proof.

Suppose $v_1, ..., v_\zeta$ be the vertices which have degree equal one.

While $v_{\zeta+1}, ..., v_n$ have degree more than one,

$$deg(v_1) + ... + deg(v_\zeta) + deg(v_{\zeta+1}) + ... + deg(v_n) = 2n - 2$$

.

$$\zeta + \sum_{deg(v) \geq 2} deg(v) = 2n - 2$$

$$\zeta + \sum_{deg(v)\geq 2} (deg(v) - 2) = 2n - 2 - 2n + 2\zeta$$

.

$$\zeta = 2 + \sum_{deg(v)\geq 3} deg(v) - 2.$$

□

Note, that it is possible to internal edges to have incidents with some end vertices of T. Consequently the number of all external edges is at most equal to ζ . This can be explained by the next lemma.

Lemma 5.2.

The number of the external edges in a non-trivial tree T is

$$\zeta < 2(r-2)k/r.$$

Proof.

Suppose that tree T has the vertex set V_T and the edge set E_T and that the cardinality of its vertices is denoted n_T while the cardinality of its edge set is denoted m_T.

If T_0 is the first old tree among those making up T and it is placed in the heap then T_0 will keep growing until the heap reaches size k. At that time the

current tree T' that contains T_0 will have more than k incident edges. Other trees may later become connected to T' causing some of these incident edges to have their endpoints in the final tree T.

According to that, after the completion of the pass each tree T will have more than k edges with at least one endpoint in T. This implies that

$$\sum_{v \in V(T)} deg(v) \geq 2k.$$

If the degree of each vertex in T has an upper bound r then

$$rn_T > 2k,$$

and

$$\zeta < (r-2)(n_T - \zeta) \leq (r-2)n_T.$$

$$(r-2)n_T \geq \zeta$$

, and

$$rn_T > 2k,$$

From these last two inequalities and by divisioning them, we find the relation between the size of the edge list for T and k as follows:

$$\zeta/2k \leq (r-2)/r$$

. The number of external edges for each tree T is less than $\zeta = O(k)$.

□

Grow Step

In the Grow-Step procedure we maintain the set A of the vertices of the current tree T that contains an old tree T_i to be treated by processor P_i. The implementation assumes that graph G is represented by adjacency lists while the set of light edges $e(T)$, which are the edges that appear in the minimum spanning tree, is added consecutively to the forest F.

```
1: Create an empty Fib-Heap H.
2: Insert each T's edges into H with key(e) = w(e).
3: Let A = T
4: while |H| < k  do
5:    repeat
6:       Find and delete min-weight edge (u, v) from the heap H
7:    until T ' is not an element in A
8:    A ⟵ A ∪ {T '}
9:    Add e = (u, v) to the forest F.
10:   if  T ' is white then
11:      Empty the heap
12:      Set key of the current tree equal to infinity.
13:   else
14:      Insert each (T')'s edges into the heap H with key(e) = w(e).
15:   end if
16: end while
17: Mark the current tree in white.
```

Algorithm 9: Grow-Step(T)

The work and the running time for a pass

So we can sort all external edges in a lexicographic order (the linked list of

the tree) in parallel by using the Parallel Radix Sort algorithm as was described in the beginning of this chapter. The algorithm was used here to sort at most $\zeta = O(k)$ edges in $O(\log \zeta) = O(\log k)$ time using $m/\log\log m$ EREW PRAM processors.

Since the number of the external edges in a non-trivial tree T is $\zeta = O(k)$ according to Lemma(5.2), we can finish the pruning process (Step 2) of the linked list of each old tree in the condensed graph as the number of external edges is bounded by $O(k)$ to run in $O(\log k)$ time using at most m processors.

To analyze the running time of growing new tree, we need to determine the upper bound of the size of the edge list for each tree T after the pass. Lemma (5.2) is the key to the complexity analysis. It gives the upper bound of the adjacency list (and the linked list) of each tree T so as to minimize the running time of pruning the adjacency list of T. At the same time, the lemma guarantees that every pass creates a new big tree by replacing the old trees $\{T_i, T_{i+1}, \ldots, T_j\}$ by their union where $i < j$ is the smallest index such that the size of the associated heap H is less than or equal to critical size k.

The result of the above lemma implies that every pass grows a single tree T by absorbing the old trees one by one, so we can determine the running time required to grow a new tree. We need at most r delete-minimum operations, each on a heap of size k or smaller. Then the total time for Grow-Step procedure is $O(r \log k)$ time, using at most n EREW PRAM processors.

Consider the parallel running time of the pass. The key observation depends

on the size of the edge list for each tree T. The edge list of T is at most $1/r$ times the parameter k (as shown in lemma(5.2)), where r is the maximum degree in T. We wish to choose values of k for successive passes so as to minimize the total running time. For each pass let us choose the global parameter k equal to $2\lceil \log m \rceil$. The most time-consuming steps are steps 2 and 4. Step 2 (The Prune Step) takes $O(\log k) = O(\log \log m)$ running time using m processors while step 4 (Growth of New Tree) takes $O(r \log k) = O(\log \log m)$ running time using n processors. The remaining steps can be implemented to run in $O(1)$ time period using linear number of processors. The running time per pass is of $O(\log \log m)$ running time using $(n + m)$ processors.

Lemma 5.3.

The algorithm terminates after no more than $O(\log m / \log \log m)$ passes.

Proof.

Since each of the m edges has only two endpoints in the given graph G then the number of trees remaining after the first pass is at most $2\ m/\ k$. For a pass i, which begins with t trees and $m\ ' < m$ edges (some edges may have been discarded), after i passes the number of remaining trees is at most $2^i m/k^i$. Since the expected number of trees that are equal to one only occurs in the last pass then the number of passes is at most $O(\log m / \log \log m)$.

□

From the above analysis it follows that there are at most $O(\log m / \log \log m)$ passes and each pass takes $O(\log \log m)$ run time using $n + m$ processors. Our

parallel algorithm runs on graphs with n vertices and m edges in $O(\log n)$ time using $n + m$ EREW PRAM processors.

5.3.4 Conclusions

This chapter presented a new deterministic parallel algorithm on EREW PRAM based on the sequential algorithm of M. L. Fredman and R. E. Tarjan [28]. The proposed parallel algorithm is simple and has the same running time as the previous best deterministic parallel algorithms which were described by K. W. Chong, Y. Han and T. W. Lam [5], and Cole and Vishkin [9]. Those algorithms run in logarithmic-time yet they all use super linear work as it is in the proposed parallel algorithm which runs in logarithmic time as well.

Part II

The Derived Subgraph Problem

Chapter 6

Derived Subgraphs

In this chapter we summarize the results related to the derived subgraphs and a derived subgraph conjecture, the graphs which satisfy the derived subgraph conjecture and the number of derived subgraphs for some famous graphs. At this time and according to the review of the related literature there is no published work that describes scientific or commercial applications for the derived subgraphs problem or residual and non-residual edges in a given graph. However, we guess the cheminformatics and bioinformatics provide two domains to apply derived subgraphs analysis. Finally we present a parallel derived subgraph algorithm which finds the set of all derived subgraphs of a given graph.

6.1 Derived Subgraph and Derived Subgraph Conjecture

A *Union-Closed Family Set* F is defined as a non-empty finite collection of finite distinct sets, closed under union. The following conjecture is due to Peter

Frankl (1979) [24].

Conjecture 1. Let $\mathfrak{A} = \{A_1, A_2, ..., A_n\}$ be a union-closed family of distinct sets, then there exists an element which belongs to at least $n/2$ of the sets in $\mathfrak{A}$.

A graph theoretic version of the Union-Closed Family was introduced by M. El-Zahar [11].

6.1.1 Definitions and Examples

Definition 6.1 (The Derived Subgraph).

If S is a nonempty subset (with $|S| \geq 2$) of the vertex set $V(\mathfrak{G}_\circ)$ of a graph $\mathfrak{G}_\circ$, then the derived subgraph induced by S is the graph having vertex set S, whose edge set consists of those edges of $\mathfrak{G}_\circ$ incident with two elements of S and having no isolated vertices.

The set of all derived subgraphs of $\mathfrak{G}_\circ$ is denoted $D(\mathfrak{G}_\circ)$, while the cardinality of $D(\mathfrak{G}_\circ)$ is denoted $n_d(\mathfrak{G}_\circ)$, and the subgraph induced by the empty graph ϕ will be considered here and denoted Φ.

Definition 6.2 (The Residual Edge).

Let $\mathfrak{G}_\circ$ be an (n, m) graph (with n vertices and m edges). An edge e of $\mathfrak{G}_\circ$ is called a residual edge if it belongs to more than half of the derived subgraphs of $\mathfrak{G}_\circ$. Otherwise e is a non-residual edge.

Example 6.1.

Consider the graph $\mathfrak{G}_\circ$ *in Figure (*6.1*). The derived subgraphs of* $\mathfrak{G}_\circ$ *are* Φ, $\mathfrak{G}_\circ$ S_1, S_2, S_3,..., S_{30}. *In all* $n_d(\mathfrak{G}_\circ) = 32$ *and the edge* v_2v_4 *occurs in* 16 *derived subgraph, and the edges* v_1v_2, v_2v_3, v_4v_5, v_4v_6 *occur in* 12 *derived subgraphs, and the remaining edges* v_1v_3, v_5v_6 *occur in* 11 *derived subgraph, and therefore each edge of* $\mathfrak{G}_\circ$ *is non-residual.*

6.1.2 Derived Subgraph Conjecture

Conjecture 2. Every non-empty graph $\mathfrak{G}_\circ$ contains at least one non-residual edge.

The above Conjecture is weaker than the Union-Closed Set Conjecture. The derived subgraph conjecture supposes that every non-empty graph has a non-residual edge. El-Zahar is the first one who presented this idea and he proved that if a graph $\mathfrak{G}_\circ$ has a vertex v with a degree (simply $deg(v)$) one, then the edge incident with v is a non-residual edge; moreover, for any graph $\mathfrak{G}_\circ$ with a vertex v with $deg(v)$ equal two, then at least one of the two edges incident with v is non-residual. If the graph $\mathfrak{G}_\circ$ has two adjacent vertices $v_1,\ v_2$ with $deg(v_1) = deg(v_2) = 3$. Then one of $v_1,\ v_2$ is incident with a non-residual edge. More results are described below.

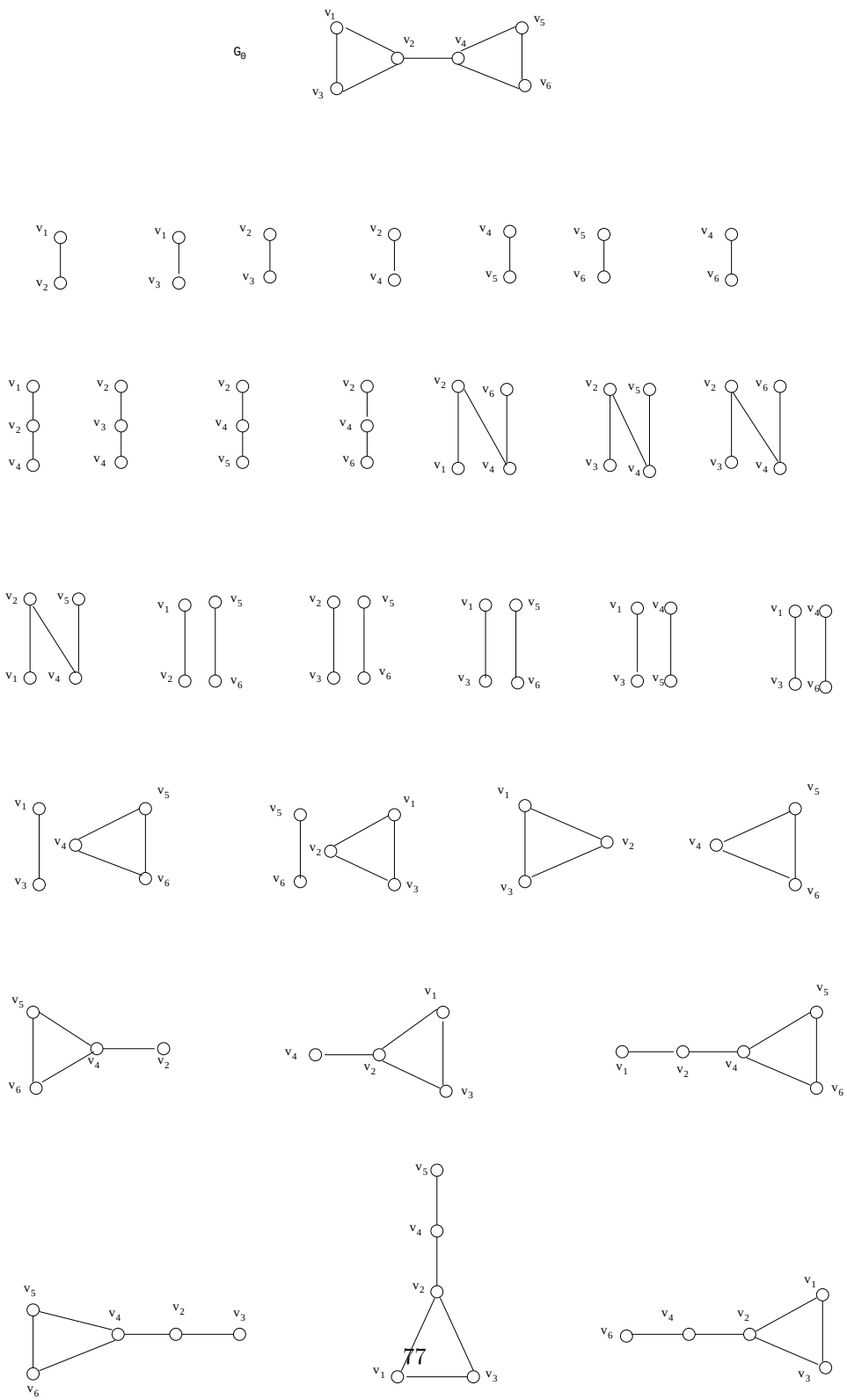


Figure 6.1: The set of all derived subgraphs of $\mathfrak{G}_\circ$

6.2 The Graphs Satisfying the Derived Subgraph Conjecture

In this section we shall restrict our attention to introduce a derived subgraphs formulas for some special graphs.

Lemma 6.1. *[11]*

Let $V = V_1 \cup V_2$ be a partition of the vertex set $V(\mathfrak{G}_o)$ of a graph $\mathfrak{G}_o$. Let the subgraphs induced by V_1 and V_2 be denoted $\mathfrak{G}_1$ and $\mathfrak{G}_2$ respectively. Then

$$n_d(\mathfrak{G}_o) \geq n_d(\mathfrak{G}_1) n_d(\mathfrak{G}_2) \tag{6.1}$$

Proof.

Consider two derived subgraphs S_1 and S_2 in $\mathfrak{G}_1$ and $\mathfrak{G}_2$ respectively. Let S denote the induced subgraph $< V(S_1) \cup V(S_2) >$. This means that, $V(S) = V(S_1) \cup V(S_1)$ and $E(S) = E(S_1) \cup E(S_1)$. Since there is no isolated vertex in S_1 or S_2, S contains no isolated vertices; therefore $S \in D(\mathfrak{G}_o)$.

□

Lemma 6.2. *[11]*

Suppose that the graph $\mathfrak{G}_o$ is the disjoint union of the two graphs $\mathfrak{G}_1$, $\mathfrak{G}_2$. Then $n_d(\mathfrak{G}_o) = n_d(\mathfrak{G}_1) n_d(\mathfrak{G}_2)$. Moreover an edge e of $\mathfrak{G}_1$ (resp. $\mathfrak{G}_2$) is residual in $\mathfrak{G}_o$ if and only if it is residual in $\mathfrak{G}_1$ (resp. $\mathfrak{G}_2$)

Proof.

Let $S \in D(\mathfrak{G}_\circ)$ and denote S_i its restriction to $\mathfrak{G}_\mathfrak{i}$ for $i = 1, 2$. Then S_i has no isolated vertices so that $S_i \in D(\mathfrak{G}_\circ), i = 1, 2$. Conversely, if $S_i \in D(\mathfrak{G}_\mathfrak{i})$, for $i = 1, 2$, then, as in the proof of Lemma(6.1), $S_1 \cup S_2 \in D(\mathfrak{G}_\circ)$. It follows that $D(\mathfrak{G}_\circ) = \{S_1 \cup S_2 : S_1 \in D(\mathfrak{G}_\mathfrak{i}), i = 1, 2\}$. The two assertions of the lemma are now obvious.

□

Lemma 6.3.

Let $K(m, n)$ be a complete bipartite graph with partite sets V_1 and V_2. Then $n_d(K(m, n)) = (2^m - 1)(2^n - 1) + 1$. In particular, if $m = 1$ then $n_d(K(1, n)) = (2 - 1)(2^n - 1) + 1 = 2^n$.

Proof.

A non-trivial derived subgraph of $K(m, n)$ is formed by taking a non-empty subset from V_1 together with a non-empty subset of V_2. There are $(2^m - 1)(2^n - 1)$ such subgraphs. Adding the empty graph, we have

$$n_d(K(m, n)) = (2^m - 1)(2^n - 1) + 1.$$

Now we put $m = 1$ in $n_d(K(m, n))$ equation then we get the number of derived subgraphs for the star graph $K(1, n)$ which is 2^n.

□

Let $v_1, v_2, ..., v_m$ be pairwise nonadjacent vertices in a graph $\mathfrak{G}_\circ$ and let $S \in D(\mathfrak{G}_\circ)$. The vertices $v_1, v_2, ..., v_m$ are said to be *free* for S whenever S contains none of these vertices and none of their neighbors. Note that, in this

case the vertices $v_1, v_2, ..., v_m$ are exactly the isolated vertices of the subgraph $< V(S) \cup \{v_1, v_2, ..., v_m\} >$.

Lemma 6.4. *[11]*

Let $\mathfrak{G}_0$ be a graph . Let $v_1, v_2, ..., v_m$ be pairwise nonadjacent vertices with degrees $d_1, d_2, ..., d_m$ such that no pair of these vertices has a common neighbor. For the number n_0 of derived subgraphs of $\mathfrak{G}_0$ for which these vertices are free holds

$$n_0 \leq n_d(\mathfrak{G}_0)2^{(d_1+d_2+...+d_m)} \tag{6.2}$$

Proof.

Let $\mathfrak{G}_1$ be the subgraph of $\mathfrak{G}_0$ spanned by $v_1, v_2, ..., v_m$ and their neighbors, and put $\mathfrak{G}_2 =< V(\mathfrak{G}_0) - V(\mathfrak{G}_1) >$. From Lemma(6.1), we have $n_d(\mathfrak{G}_0) \geq n_d(\mathfrak{G}_1)n_d(\mathfrak{G}_2)$. But $n_d(\mathfrak{G}_2) = n_0$, and $\mathfrak{G}_1$ is the union of disjoint stars so that

$$n_d(\mathfrak{G}_1) = 2^{d_1+d_2+...+d_m}$$

$$n_d(\ \mathfrak{G}_0) \geq n_0 2^{d_1+d_2+...+d_m}.$$

This proves the required result.

□

We will show that the graphs which have the minimum degree is more than or equal to $O(\log n)$ have no residual edge. The minimum degree of the vertices of a graph $\mathfrak{G}_0$ is denoted $\delta(\mathfrak{G}_0)$.

Theorem 6.1. *[11]*

Let $\mathfrak{G}_\circ$ be a graph with n vertices. If

$$\sum_{i=1}^{n} 2^{-deg(v_i)} \leq 1, \tag{6.3}$$

then every edge of $\mathfrak{G}_\circ$ is non-residual. In particular, if $\delta(\mathfrak{G}_\circ) \geq \log n$ then $\mathfrak{G}_\circ$ has no residual edges.

Proof.

The number of induced subgraphs of $\mathfrak{G}_\circ$ that contain $v \in V(\mathfrak{G}_\circ)$ as an isolated vertex is $2^{n-1-deg(v)}$. So

$$n_d(\mathfrak{G}_\circ) \geq 2^n - \sum_{i=1}^{n} 2^{n-1-deg(v_i)}.$$

On the other hand, every edge e of $\mathfrak{G}_\circ$ is contained in exactly 2^{n-2} induced subgraphs of $\mathfrak{G}_\circ$. Then e occurs in at most 2^{n-2} derived subgraphs of $\mathfrak{G}_\circ$. If $\mathfrak{G}_\circ$ has a residual edge, then $n_d(\mathfrak{G}_\circ) \leq 2^{n-1}$ and then

$$2^{n-1} > 2^n - \sum_{i=1}^{n} 2^{n-1-deg(v_i)}.$$

This proves the first part of the theorem. Now if $\delta(\mathfrak{G}_\circ) \geq \log_2 n$ then

$$\sum_{i=1}^{n} 2^{-deg(v_i)} \leq 1.$$

□

As it was proved in [11]; for any $\mathfrak{G}_\circ$ which has two adjacent vertices v_1, v_2

with $deg(v_1) = deg(v_2) = 3$, one of v_1, v_2 is incident with a non-residual edge. The derived subgraph conjecture has proved for any graph on order less than or equal to 10 vertices.

Theorem 6.2. *[11]*

Every graph with $n \leq 10$ vertices satisfies the derived subgraph conjecture.

Proof.

Let $\mathfrak{G}_\circ$ be a graph with n vertices, such that $\mathfrak{G}_\circ$ has no non-residual edge. Then $\delta(\mathfrak{G}_\circ) \geq 3$, and by theorem (1) $\delta(\mathfrak{G}_\circ) < \log_2 n$, this implies that $n \geq 9$. Suppose first that $n = 9$, by theorem(1) $\sum_{i=1}^{n} 2^{-deg(v_i)} > 1$, this implies that $\mathfrak{G}_\circ$ has at least 8 vertices of degree 3. This forces $\mathfrak{G}_\circ$ to have 3 vertices of degree at least 7. But $\sum_{i}^{n} 2^{-deg(v_i)} > 1$ cannot be satisfied. This contradiction completes the proof.

□

By using inclusion and exclusion principle, it is easy to get the following inequality to proof that derived subgraph conjecture satisfies for graphs on order $n = 11$ and 12 vertices. Let the number of vertices of degree 3 in $\mathfrak{G}_\circ$ be denoted P_3. By Theorem (3.2.5) in [21] the vertices of degree 3 must be pairwise non-adjacent vertices.

Theorem 6.3. *[21]*

The number of derived subgraphs $n_d(\mathfrak{G}_\circ)$ of $\mathfrak{G}_\circ$ satisfies

$$2^n - \sum_{i=1}^{n} 2^{n-1-deg(v_i)} + \sum_{i=2}^{P_3} (-1)^i \binom{P_3}{i} 2^{P_3-i} \leq n_d(\mathfrak{G}_o). \tag{6.4}$$

Proof.

Use the inclusion and exclusion principle to prove the theorem. Let $v \in V(\mathfrak{G}_o)$ then the number of induced subgraphs which contain v as isolated vertex is $2^{n-1-deg(v)}$. Then

$$n_d(\mathfrak{G}_o) \geq 2^n - \sum_{i=1}^{n} 2^{n-1-deg(v)}$$

.

Since P_3 denotes the number of vertices of degree 3 in $\mathfrak{G}_o$ which are non-adjacent then any subset S ($|S| \geq 2$) from these vertices has an isolated vertex.

Add the number $\binom{P_3}{2}2^{P_3-2}$ of induced subgraphs which contain two vertices from P_3, but the number $\binom{P_3}{3}2^{P_3-3}$ of induced subgraphs which contains three vertices from P_3 will be counted twice, so subtract this number, continue the above procedure until finally arriving at the set of all non-adjacent vertices of degree 3. This accounts for the number

$$\sum_{i=2}^{P_3} (-1)^i \binom{P_3}{i} 2^{P_3-i}$$

of induced subgraphs which contain isolated vertices of degree 3.

□

By direct use of the above inequality and Theorem(6.2), any graph on $n \leq 12$ vertices satisfies the derived subgraph conjecture. More results about the graphs

which satisfies the derived subgraph conjecture presented in [21].

6.3 The Number of Derived Subgraphs

For simplicity, denote $n_d(P_n)$ and $n_d(C_n)$ by a_n and b_n respectively.

Theorem 6.4. *[21]*

Let P_n be the path $v_1, v_2, ..., v_n$. Then the number of its derived subgraphs, a_n, is given by the relation

$$a_n = 2a_{n-1} - a_{n-2} + a_{n-3} \tag{6.5}$$

where

$$a_0 = a_1 = 1, \quad a_2 = 2.$$

Proof.

The number of derived subgraphs of P_n not containing the vertex v_1 is a_{n-1}, and let the number of derived subgraphs which contain v_1 be denoted c_n then $a_n = a_{n-1} + c_n$, and c_n can be determined as follows. The number of derived subgraphs which contain v_1 and v_2 and do not contain v_3 is a_{n-3}, and the number of derived subgraphs which contain v_1, v_2 and v_3 is $c_{n-1} = a_{n-1} - a_{n-2}$. Then

$$c_n = a_{n-3} + a_{n-1} - a_{n-2}$$

and

$$a_n = 2a_{n-1} - a_{n-2} + a_{n-3}.$$

□

There are two equivalent formulas to calculate the number b_n for a given cycle. The following theorem describes one of them.

Theorem 6.5. *[21]*

Let C_n be a cycle on $n \geq 4$ vertices. Then the number b_n of its derived subgraphs satisfies the relation

$$b_n = a_{n-1} + 2(n-1) + \sum_{i=2}^{n-3} i\ a_{n-i-2}. \tag{6.6}$$

Proof.

Let C_n be the cycle $v_1, v_2, ..., v_n, v_1$. Let x_1 denote the number of derived subgraphs of C_n not containing v_1; then $x_1 = a_{n-1}$. On the other hand, let x_2 denote the number of derived subgraphs which contain v_1. Such a derived subgraph contains a path P_i of length $(i-1)$ that contains v_1, and a derived subgraph of path of order $(n-i-2)$, where $2 \leq i \leq n-1$. Then for fixed i this number is ia_{n-i-2}. Thus

$$x_2 = \sum_{i=2}^{n-3} i\ a_{n-i-2} + (n-2) + (n-1).$$

Moreover C_n is a derived subgraph of itself. Therefore

$$b_n = x_1 + x_2 + 1$$

and

$$b_n = a_{n-1} + 2(n-1) + \sum_{i=2}^{n-3} i\ a_{n-i-2}.$$

□

The number of derived subgraphs for a given path P_n or cycle C_n which contain an arbitrary edge in $E(P_n)$ or $E(C_n)$ respectively are presented here.

Theorem 6.6. *[21]*

Let $P_n : v_1, e_1, ..., v_i, e_i, v_{i+1}, ..., e_{n-1}, v_n$ be a path. Then the number of derived subgraphs of P_n which contain the edge $e_i, i = 1, 2, ..., n-1$ is equal to

$$a_n - a_i a_{n-i-1} + a_{i-1}(a_{n-i-1} - a_{n-i}). \tag{6.7}$$

Proof.

Let x_i denote the number of derived subgraphs containing v_i and v_{i+1}, x_2 denote the number of derived graphs containing v_i but not containing v_{i+1} while x_3 denote the number of derived subgraphs containing v_{i+1} but do not contain v_i, and x_4 denote the number of derived subgraphs containing neither v_i nor v_{i+1} then

$$x_1 + x_2 + x_3 + x_4 = a_n,$$

$$x_4 = a_{i-1} a_{n-i-1},$$

$$x_2 + x_4 = a_i a_{n-i-1},$$

$$x_3 \ + \ x_4 \ = \ a_{i-1}a_{n-i},$$

$$x_1 \ = \ a_n \ - \ (x_2 \ + \ x_4) \ - \ (x_3 \ + \ x_4) \ + \ x_4,$$

$$\therefore \quad x_1 \ = \ a_n \ - \ a_i a_{n-i-1} \ + \ a_{i-1}(a_{n-i-1} \ - \ a_{n-i}).$$

□

Theorem 6.7. *[21]*

Let C_n denote the cycle $v_1, e_1, ..., e_{n-1}, v_n, e_n, v_1$. Then the number of derived subgraphs of C_n which contain an arbitrary edge $e \in E(C_n)$ is equal to

$$1 + 3(n-3) + \sum_{i=2}^{n-4}(i-1)a_{n-2-i}. \tag{6.8}$$

Proof.

Let y_1 be the number of derived subgraphs of C_n which contain an arbitrary edge, say, $e_n = v_1v_n$ and take the set $B' = \{v_2, ..., v_{n-1}\}$. The number of derived subgraphs which contain the edge e_n and result by removing one vertex from B' is $n-2$. If we remove two consecutive vertices from B', the resulting number is $(n-3)$, while the number is $(n-4)$ when remove three consecutive vertices. Moreover, C_n is a derived subgraph of itself.

Now suppose the edge v_1v_n belongs to a component of the derived subgraph which is a path of order i where $i = 2, 3, ..., n-4$. The number of such derived subgraphs, for a fixed i, is equal to $(i-1)a_{n-i-2}$. Then

$$y_1 = 1 + (n-4) + (n-3) + (n-2) + \sum_{i=2}^{n-4}(i-1)a_{n-2-i}.$$

□

The next consideration is a special case of a bipartite graph $G(n,n)$ on $2n$ vertices in which $deg(v) = n-1$ for each $v \in V(G(n,n))$.

Theorem 6.8. *[21]*

Let $G(n,n)$ be a bipartite graph with two partitioning sets V_1 and V_2, where $|V_1| = |V_2| = n$ and $deg(v) = n-1$ for each $v \in V(G(n,n))$. Then

$$n_d(G(n,n)) = 2^{2n} + n + 2 - n2^n - 2^{n+1}.$$

And each edge $uv \in E(G(n,n))$ is contained in exactly $2^{n-1}(2^{n-1}-1)$ derived subgraphs.

Proof.

Let $V_1 = v_1, v_2, ..., v_n$ and $V_2 = u_1, u_2, ..., u_n$ where $u_iv_i \notin E(G(n,n))$ for each $i = 1, 2, ..., n$.

To form a derived subgraph, we take $S_1 \cup S_2$ where $S_i \subset V_i$ for $i = 1, 2$. If $|S_1| \geq 2$ and $|S_2| \geq 2$ then we get a derived subgraph.

If $|S_1| = 1$, say $S_1 = v_i$ then $\phi \neq S_2 \cup V_2 \setminus \{u_i\}$. This shows that

$$n_d G(n,n) = (2^n - n - 1)^2 + 1 + 2n(2^n - 1).$$

Now we fix an $i = 2, 3, ..., n$. We count the number of derived subgraphs which contain the edge v_1u_i. Such derived subgraph will have the form $S_1 \cup S_2$ where $v_1 \in S_1 \subset V_2$.

Again if $|S_1| \geq 2$ and $|S_2| \geq 2$ then we have a derived subgraph.

If, say $S_1 = \{v_1\}$ then $u_1 \notin S_2$. This shows that the number of derived subgraph which contain v_iu_i is equal to

$$(2^{n-1} - 1)^2 + 2(2^{n-2}) - 1 = 2^{2n-2} - 2^{n-1}.$$

□

Many results related to the number of derived subgraphs and residual edges for the paths, cycles, complete graphs, star graphs and bipartite graphs are presented in [21].

6.4 The Serial Derived Subgraph Algorithm

Here we consider a serial algorithm to calculate the number of derived subgraphs for a given graph $\mathfrak{G}_\circ$. The algorithm also determines residual and non-residual edges. In addition this algorithm shows every derived subgraph of $\mathfrak{G}_\circ$. The parameters of the algorithm are:

i) The number *total* denotes the number of all derived subgraphs of $\mathfrak{G}_o$.

ii) The set $S \subset V(\mathfrak{G}_o)$ represented by an array $S[j]$, $j = 1, \ldots, n$. The initial subset is the empty set denoted S_0.

iii) The (i, j) entry of the matrix $E[i, j]$ is the number of derived subgraphs which contain the edge $v_i v_j$.

The graph $\mathfrak{G}_o$ has n vertices and m edges represented by the *Adjacency-Graph* class, where $a[i][j]$ is the entry element (i, j) in the adjacency matrix A. The algorithm finds all subsets of $V(\mathfrak{G}_o)$; then it checks if the current subset induces a derived subgraph or not. The algorithm finds the number of derived subgraphs that contain any edge $e \in E(\mathfrak{G}_o)$.

In the beginning, we assume that the initial subset S_0 is represented by an array $S[j] = 0$. The subgraph induced by S_0 is the empty derived subgraph. We outline below the initialize procedure which considers the empty subgraph as the first derived one.

1: Take the empty set to be the initial subset S_0.
2: Set the value of total = 1.
3: For every edge $e = (i, j)$ let $E[i, j] = 0$.
4: Done $\longleftarrow$ False.

Algorithm 10: Initialize-subset $(\mathfrak{G}_o, S_0)$

After initialization the algorithm finds all subsets of $V(\mathfrak{G}_o)$. The simplest approach to get a new subset is based on the observation that any subset S' of $V(\mathfrak{G}_o)$ is defined by which of the $n = |V|$ items are in S'. We can represent S' by a binary string of n bits, where bit i is 1 if and only if the ith element of S is in S'. This defines a bijection between the 2^n binary strings of length n, and the 2^n subsets of n items. For $n = 3$, binary counting generates subsets in the following order: {}, {3}, {2}, {2, 3}, {1}, {1, 3}, {1, 2}, {1, 2, 3}.

This alternative is known as *a binary counting* representation; it is the key to solving all subset generation problems. To generate all subsets in order, simply count from 0 to $2^n - 1$. For each integer, successively mask off each of the bits and compose a subset of exactly the items corresponding to 1 bits. To generate the next or previous subset, increment or decrement the integer by one. We give now the Next-Subset procedure.

```
1: j ⟵ n + 1
2: repeat
3:    j ⟵ j − 1
4: until ((S[j] = 0)or(j = 0))
5: if j ≠ 0 then
6:    S[j] ⟵ 1
7:    MAX ⟵ j
8:    for  i = MAX + 1 ⟶ n do
9:       S[i] = 0
10:   end for
11: else
12:   Done ⟵ True
13: end if
```

Algorithm 11: Next-Subset($\mathfrak{G}_o$, S)

The procedure Check-Subset($\mathfrak{G}_o, S$) verifies the current set S as a derived subgraph or not. A precise description of this process is the following.

```
1: DERIVED ⟵ False
2: count ⟵ 1
3: for  k = 1 ⟶ MAX  do
4:    if  S[k] = 1 then
5:       sum = 0
6:       for j = 1 ⟶ n do
7:          sum = sum + a[k][j] * S[j]
8:          if  sum ≠ 0 then
9:             sum ⟵ 1
10:            count ⟵ count * sum
11:         end if
12:      end for
13:   end if
14: end for
15: if count ≠ 0 then
16:    DERIVED ⟵ True
17: end if
```

Algorithm 12: Check-Subset($\mathfrak{G}_o$,S)

In the main procedure (denoted SDS) of the algorithm calls all subsets one by one and checks if the current subset is a derived subgraph or not. If it is, the algorithm adds one to the parameter *total* and it checks all edges if they are residual or not.

The SDS algorithm can be shown to run in $n2^n$ time, where n is the number of vertices in the given graph. There are 2^n subsets of $V(\mathfrak{G}_o)$. We check every one by calling Check-Subset(S). The Check-Subset(S) procedure requires time n. We need exactly $2^n - 1$ calls of Next-Subset procedure, each one runs in n time. Then the total running time of derived subgraph algorithm is $n2^n$ sequential time.

```
1: Initialize-subset(𝔊₀,S₀)
2: while Not Done do
3:    Get-Next-Subset (𝔊₀, S) of the vertex set V(𝔊₀).
4:    Check-Subset(𝔊₀, S)
5:    if DERIVED then
6:       total ⟵ total + 1 {Once a derived subgraph has been induced, the algorithm counts
         the frequency for each edge e = (v_i, v_j)}
7:       for each edge e = (v_i, v_j) do
8:          if S[i] = S[j] = 1 then
9:             E[i, j] ⟵ E[i, j] + 1
10:         end if
11:      end for
12:   end if
13: end while
14: Return total,
15: For each e = (v_i, v_j) if E[i, j] > total/2 the edge e is residual otherwise it is non-residual
```

Algorithm 13: SDS($\mathfrak{G}_\circ$, n, m)

6.4.1 Related Works

A derived subgraph is a graphic version of the Union-Closed Family sets. Up to my knowledge and according to the review of the related literature there is no published work that describes scientific or commercial applications for the derived subgraphs problem or residual and non-residual edges in a given graph. However, I believe the cheminformatics and bioinformatics provide two domains to apply derived subgraphs analysis. Some researchers have used *the frequent subgraph mining problem*, which is a similar problem and most closely related to the one considered in this study. They used that problem in various applications in the cheminformatics and bioinformatics fields (For instance, consider a problem of mining chemical compounds to find recurrent substructures). See for example [15], [16], [19], [30] and [31].

In the following paragraphs we describe the frequent subgraph mining problem and show how it is similar to the derived subgraph problem.

Definition 6.3 (The Frequent Subgraphs).

Given a set $D = \{G_1, ..., G_n\}$ *of labeled graphs* $G_i, i = 1, ..., n$. *D is referred to as a graph database. The support of an arbitrary graph g - denoted support*(g)*- is the number of graphs in D in which g is a subgraph. The graph g is frequent if support*$(g) \geq \sigma$*, where* $0 < \sigma < 1$ *is a minimum support threshold; a frequent subgraph is maximal if none of its super graphs are frequent. The problem of frequent subgraph mining is to find all connected frequent subgraphs from a graph database D.*

The Frequent Subgraphs Methodology

To find the frequent subgraphs, every graph in D is represented by an adjacency matrix M. Then define the code of M, denoted *code(M)*, as the sequence of lower triangular entries of M. A graph can be represented in many different codes, depending on the order of its edges or vertices. Given a graph G, its canonical form is the maximal code among all its possible codes. The adjacency matrix which produces the canonical form is denoted as G's canonical adjacency matrix (CAM). The methodology for solving the frequent problem is outlined below.

Step I- Enumerating all the frequent subgraphs: This methods might be classified into two categories. One is the join operation (see Fig 6.2): The sub-

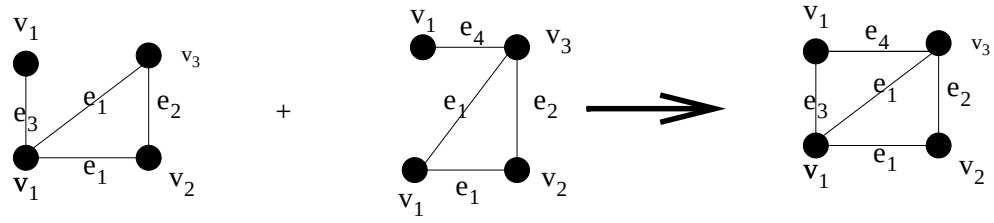


Figure 6.2: The subgraph of size 5 is generated by joining two frequent 4−subgraph.

graphs of size $(k+1)$ are generated by joining two frequent $k-$subgraph. In order for two frequent $k-$subgraphs to be eligible for joining they must contain the same $(k-1)-$subgraph. This joining of two subgraphs of size k can lead to multiple subgraphs of size $k+1$. The major challenge for the join operations is that every distinct subgraph is generated only once.

The other one is the extension operation: Starting from the subgraphs of size k, generates candidate subgraphs whose size is $(k+1)$ by adding an additional edge(while preserving connectivity) to the $k-$subgraphs.

Step II- Frequency Counting: Once the candidate subgraphs have been generated, the proposed methodology counts the frequency for each of these candidates, and prunes subgraphs that do not satisfy the σ threshold.

From the above description of the frequent subgraph mining problem and the work done by others for that problem, we can recognize the theoretic similarities between the frequent subgraph mining problem and the derived subgraph problem.

In the beginning, the derived subgraph methodology can be used to find and enumerate all connected subgraphs in Step I. This proceeds as follows: Initially induce the line graph $L(G)$ of G. Then apply the SDS$(L(G), m, m\,')$ algorithm to enumerate all derived subgraphs of $L(G)$. Once the derived subgraphs of $L(G)$ have been induced, consider only the connected ones, and prunes other derived subgraphs. Finally enumerate all the single node subgraphs of the graph $L(G)$ (corresponding to the single edge subgraphs in G). All the above generated subgraphs of $L(G)$ are corresponding to all the subgraphs in Step I.

On the one hand and returning to the definition of the frequent subgraph problem, we find that the graph dataset D and the set of all derived subgraph $D(\mathfrak{G}_\circ)$ are similar in having threshold. However we note that the threshold is not necessarily equal for both of them. The threshold σ (in the frequent problem) may have several values, but in the derived subgraph problem the threshold has a fixed value, 1/2. Another way of similarity between the solution of the two problems is that each finds the subgraphs that belong to more than the set defined by this threshold. The frequent methodology finds the subgraphs (for example g) that can belong to more than or equal to σ of the graphs in D, while the derived subgraphs methodology finds the edges in $D(\mathfrak{G}_\circ)$ that belong to more than 1/2 of the graphs in $D(\mathfrak{G}_\circ)$.

Let $\mathfrak{G}_\circ$ be a graph with n vertices and m edges and $g \subset \mathfrak{G}_\circ$ a subgraph with α edges. Since g is a subgraph of exactly $2^{m-\alpha}$ graphs in the power set $2^{E(\mathfrak{G}_\circ)}$,

then, based on conjecture 2, the subgraph g belongs to at most $1/2^{\alpha}$ of the derived subgraphs of $\mathfrak{G}_{\circ}$. Let us call a subgraph g of $\mathfrak{G}_{\circ}$ residual if it belongs to more than $1/2^{\alpha}$ of the derived subgraphs of $\mathfrak{G}_{\circ}$, otherwise g is non-residual. Therefore conjecture 2 is equivalent to the following.

Conjecture 3. Every non-empty graph $\mathfrak{G}_{\circ}$ contains at least one non-residual subgraph.

We arrive to the following conjecture which is implied by conjecture 3.

Conjecture 4. Let $D = \{G_1, ..., G_n\}$ be a graph database set. Then there exists an arbitrary graph g with α edges and $support(g) \leq 1/2^{\alpha}$.

On the other hand, I guess we could use the frequent subgraph methodology (see FFSM algorithm [15]) to check for residual edges, and we should feed it the set $D(\mathfrak{G}_{\circ})$, we set the support at 0.5 and count only the frequency of the single edge subgraphs in $\mathfrak{G}_{\circ}$.

Hence we believe that our problem is applicable in the same areas as the frequent subgraph mining problem.

6.4.2 The Modification of Serial Derived Subgraph Algorithm

Before we describe our parallel algorithm, we need to add new functions to the SDS procedure. For an input graph $\mathfrak{G}_{\circ}$ a proper non-empty subset $V^g(\mathfrak{G}_{\circ})$ is called the set of global vertices. For any subset $I_i \subseteq V^g(\mathfrak{G}_{\circ})$ we give below two

distinct kinds of graphs that contain I_i.

1- The set $D_0(I_i)$ gives all derived subgraphs which contain all the global vertices of I_i

$$D_0(I_i) = \{s|\ s \in D(\mathfrak{G}_\circ),\ \text{ s.t. for each global vertex } v \in s \Longrightarrow v \in I_i\}. \quad (6.9)$$

2- The set $\Re_0(I_i)$ contains all subgraphs of $\mathfrak{G}_\circ$ which contain all vertices in I_i as the only pairwise nonadjacent vertices.

$$\Re_0(I_i) = \{s|\ s \subset \mathfrak{G}_\circ,\ \text{ s.t.}\forall\ v \in s \quad deg(v) = 0 \text{ if and only if } v \in I_i\}. \quad (6.10)$$

In the modifying process we write the main procedure denoted MSDS which was used only in the last phase of our parallel proposed algorithm. The MSDS procedure uses two arrays X and Y to store the numbers of the sets $D_0(I_i)$ and $\Re_0(I_i)$ respectively. The $i-$th position of the arrays X and Y represents the numbers $|D_0(I_i)|$ and $|\Re_0(I_i)|$ respectively. The MSDS procedure (as SDS procedure) calls all subsets of $V(\mathfrak{G}_\circ)$ one by one and checks if the current subset induces a derived subgraph or not. If it is, the MSDS procedure (different from SDS) calls SUBPROCEDURE-1 which returns the array X. If the current subset does not induce a derived subgraph, the MSDS procedure (different from SDS) calls SUBPROCEDURE-2 which returns the array Y. The MSDS procedure (as SDS procedure) finds the number of derived subgraphs and determines the residual and non-residual edges. Below, we present a high-level outline of the MSDS procedure which returns the following:

1: Create the empty arrays X_1,Y_1 and X_2 Y_2
2: Initialize-subset($\mathfrak{G}_\circ$, S_0)
3: **while** Not Done **do**
4: Next-Subset ($\mathfrak{G}_\circ, S$)
5: Check-Subset($\mathfrak{G}_\circ, S$)
6: **if** S induces a derived subgraph **then**
7: $n_d(\mathfrak{G}_\circ) \longleftarrow n_d(\mathfrak{G}_\circ) + 1$
{If the current subset induces a derived subgraph, the algorithm counts all derived subgraphs which contain an arbitrary subset of global vertices.}
8: SUBPROCEDURE-1
{Once a derived subgraph has been induced, the algorithm counts the frequency for each edge $e = (v_i, v_j)$}
9: **for** each edge $e = (v_i, v_j)$ **do**
10: **if** $S[i] = S[j] = 1$ **then**
11: $E[i, j] \longleftarrow E[i, j] + 1$
12: **end if**
13: **end for**
{If the current subset does not induce a derived subgraph, the algorithm counts all subgraphs which contain an arbitrary subset of global vertices as the only pairwise non-adjacent vertices.}
14: **else**
15: SUBPROCEDURE-2
16: **end if**
17: **end while**

Algorithm 14: MSDS($\mathfrak{G}_\circ$, n, m)

1- The number of derived subgraph $n_d(\mathfrak{G}_\circ)$ for the given graph

2- The number $|D_0(I_i)|$ that is represented in the $i-$th position of the array X, for any arbitrary global set I_i

3- The number $|\Re_0(I_i)|$ that is represented in the $i-$th position of the array Y, for any arbitrary global set I_i

4- The residual and non-residual edges

The MSDS procedure considers two disjoint sets with a constant number of global vertices, the first denoted V_1^g and the second denoted V_2^g. Then the MSDS

procedure returns the arrays (X_1, Y_1) and (X_2, Y_2) corresponding to V_1^g and V_2^g respectively.

Implementation Note 1

In step 8 of procedure MSDS, we use the **SUBPROCEDURE-1** which returns the two arrays X_1 and X_2. The sub-procedure initially creates (in lines 1-2) two empty sets I and J and two integers k and z. The for loop in lines $3-11$ looks for the global vertices of V_1^g (or V_2^g) and stores those vertices in the set I (or in the set J). In line 12 we create an index for each subset I of global vertices. We let this index equals i. In line 13 we increase the $i-$th position of the array X_1 by one. This means that the number of derived subgraphs which contains the subset $I_i \in V_1^g$ is increased by one. In line 14 we create an index for the subset J of global vertices. We let this index equals j. In line 15 we increase the $j-$th position of the array X_2 by one. This means that the number of derived subgraphs which contains the subset $J_j \in V_2^g$ is increased by one.

How fast is **SUBPROCEDURE-1** ? Because the length of the current subset S equals n, the for loop in lines $3-11$ is executed $O(n)$ times, which in total takes $O(n)$ time. Other steps in the sub-procedure take time $O(1)$. Therefore, the total running time is $O(n)$.

Implementation Note 2

In step 15 of procedure MSDS, we use the **SUBPROCEDURE-2** which returns the two arrays Y_1 and Y_2. In line 1 we induce the subgraph $\mathfrak{S}$ (is

1: $I, J \longleftarrow \phi$
2: $k, z \longleftarrow 0$
3: **for** each $v_i \in S$ **do**
4: **if** $v_i \in V_1^g$ **then**
5: $I \longleftarrow I \cup v_i$
6: $k \longleftarrow k + 1$
7: **else if** $v_i \in V_2^g$ **then**
8: $J \longleftarrow J \cup v_i$
9: $z \longleftarrow z + 1$
10: **end if**
11: **end for**
12: $i \longleftarrow index.subsets(I, k, |V_1^g|)$
13: $x_1[i] \longleftarrow x_1[i] + 1$
14: $j \longleftarrow index.subsets(J, z, |V_2^g|)$
15: $x_2[j] \longleftarrow x_2[j] + 1$

Algorithm 15: SUBPROCEDURE-1: Compute X_1 and X_2

induced on the subset S). In lines $(2-4)$ we create two empty sets I and J and some integer numbers k, z, and d.

The for loop in lines $5-21$ looks for the global vertices in V_1^g (or V_2^g) and stores those vertices in the set I (or in the set J). In lines 6 and 9 we consider here only the global vertices which are pairwise non-adjacent vertices in the induced subgraph $\mathfrak{S}$. In line 13 the value d_i refers to the degree of the local (not global) vertex $v_i \in V(\mathfrak{S})$. In line 15 if there is an isolated local vertex in the induced subgraph $\mathfrak{S}$ we delete the subgraph $\mathfrak{S}$ and terminate the sub-procedure. In line 16 we create an index for each subset I of global vertices. We let this index equal i. In line 17 we increase the $i-$th position of the array Y_1 by one. This means that the number of subgraphs which contains the subset $I_i \in V_1^g$ (as the only pairwise non-adjacent vertices in $\mathfrak{S}$) is increased by one. In line 18 we create an index for the subset J of global vertices. We let this index equal j. In line 19 we increase the $j-$th position of the array Y_2 by one. This means

that the number of subgraphs which contains the subset $J_j \in V_2^g$ (as the only pairwise non-adjacent vertices in $\mathfrak{S}$) is increased by one. Finally, in line 22 we delete the induced subgraph $\mathfrak{S}$.

How fast is **SUBPROCEDURE-2** ? Because the length of the current subset S equals n, the for loop in lines $5-21$ is executed $O(n)$ times, which in total take $O(n)$ time. Other steps in the sub-procedure take time $O(1)$. Therefore, the total running time is $O(n)$.

1: $\mathfrak{S}$ is the induced subgraph on S
2: $I, J \longleftarrow \phi$
3: $k, z \longleftarrow 0$
4: $d \longleftarrow 1$
5: **for** each $v_i \in S$ **do**
6: **if** $v_i \in V_1^g$ and $deg(v_i) = 0$ **then**
7: $I \longleftarrow I \cup v_i$
8: $k \longleftarrow k + 1$
9: **else if** $v_i \in V_2^g$ and $deg(v_i) = 0$ **then**
10: $J \longleftarrow J \cup v_i$
11: $z \longleftarrow z + 1$
12: **else**
13: $d \longleftarrow d \times d_i,$
14: **end if**
15: **if** $d \neq 0$ **then**
16: $i \longleftarrow index.subsets(I, k, |V_1^g|)$
17: $y_1[i] \longleftarrow y_1[i] + 1$
18: $j \longleftarrow index.subsets(J, z, |V_2^g|)$
19: $y_2[j] \longleftarrow y_2[j] + 1$
20: **end if**
21: **end for**
22: Delete the induced subgraph $\mathfrak{S}$

Algorithm 16: SUBPROCEDURE-2: Compute Y_1 and Y_2

The MSDS algorithm uses all the sub-procedures; Initialize, Next-Subset, and Check-Subset as does the SDS algorithm. There are two additional subrou-

tines **SUBPROCEDURE-1** and **SUBPROCEDURE-2** every one of them running in $O(n)$ time. Therefore, the modified procedure MSDS has the same running time as the SDS algorithm. It runs in $O(n2^n)$ time, where n is the number of vertices in the given graph.

The proposed PDS algorithm is a parallel application of the serial derived subgraph algorithm. The graph $\mathfrak{G}_o$ will be divided into a number of subgraphs in order to apply the proposed method to each of them. The following section describes the assumptions needed to do this.

6.5 Parallel Derived Subgraph Algorithm

In this section, we outline the parallel method used to find the set of all derived subgraphs, and to recognize the residual and non-residual edge of a given undirected graph $\mathfrak{G}_o$.

6.5.1 Assumptions and Definitions

The partitioning of the graph $\mathfrak{G}_o$ into a number of subgraphs is the key idea in the parallel derived subgraph algorithm (denoted PDS). Let the vertices of the graph $\mathfrak{G}_o = (V, E)$ are partitioned into the two sets V_1 and V_2. We define the set of bridge (shared) edges to be the edge subset $H(\mathfrak{G}_o) \subset E(\mathfrak{G}_o)$ where an edge $(v, w) \in H(\mathfrak{G}_o)$, if and only if $v \in V_1$ and $w \in V_2$. A vertex is considered in the global set of vertices $V^g(\mathfrak{G}_o)$, if and only if the vertex is an endpoint for some edges in $H(\mathfrak{G}_o)$. We use the set of local vertices, denoted V^l as the set $V - V^g$,

and we use V_i^g to denote the vertex set $V^g \cap V_i$ where $i = 1, 2$.

To simplify the presentation and make our proposed parallel algorithm clear, we first apply the algorithm on the special class of graphs that have a non-empty proper subset W of the vertex set V of the input graph such that the number of edges joining W and $V - W$ is one. The tree graphs and the graph shown in Fig.(6.3a) belong to this class.

Let $\mathfrak{G}_1$ and $\mathfrak{G}_2$ be the two subgraphs induced on V_1 and V_2 respectively such that there is only one edge $e = (v, u)$ connecting $\mathfrak{G}_1$ and $\mathfrak{G}_2$, where v and u are the global vertices of $\mathfrak{G}_1$ and $\mathfrak{G}_2$ respectively. We will use the term *missing set* to describe the set of derived subgraphs of $\mathfrak{G}_0$ that contain the bridge edge $e = (v, u)$, such that, $deg(v) = 1$ or $deg(u) = 1$, or $deg(v) = deg(u) = 1$. Let this missing set be denoted $D_{out}^{(e)}(\mathfrak{G}_1, \mathfrak{G}_2)$, and its cardinality be denoted $n_{out}^{(e)}(\mathfrak{G}_1, \mathfrak{G}_2)$. The method will then find all the derived subgraphs that belong to $D_{out}^{(e)}(\mathfrak{G}_1, \mathfrak{G}_2)$.

The number of derived subgraph $n_d(\mathfrak{G}_0)$ satisfies the following equality (see Lemma (6.5)):

$$n_d(\mathfrak{G}_0) = n_d(\mathfrak{G}_1) n_d(\mathfrak{G}_2) + n_{out}^{e}(\mathfrak{G}_1, \mathfrak{G}_2). \tag{6.11}$$

To calculate $n_{out}^{e}(\mathfrak{G}_1, \mathfrak{G}_2)$ let $\Re_1(v)$ ($\Re_2(u)$) denotes the set of all induced subgraphs of $\mathfrak{G}_1$ ($\mathfrak{G}_2$) that contain the global vertices v (u) as the only isolated vertex. The set of derived subgraphs of $\mathfrak{G}_1$ ($\mathfrak{G}_2$) that contain v (u) is denoted $D_1(v)$ ($D_2(u)$). Based on that the following three derived subgraph sets can be

constructed:

The Set S_1: The set of all non-empty derived subgraphs induced by all subgraphs $\Re_1(v)$ and $\Re_2(u)$ together, $S_1 =< \Re_1(v) \cup \Re_2(u) >$.

The Set S_2: The set of derived subgraphs induced by $D_1(v)$, and $\Re(u)$ together, $S_2 =< D_1(v) \cup \Re_2(u) >$.

The Set S_3: The set of all derived subgraphs induced by $D_2(u)$, and $\Re_1(v)$ together, $S_3 =< D_2(u) \cup \Re_1(v) >$.

Note that all derived subgraphs resulting from the above sets keep track of the global vertices and the bridge edges. We give the following example to illustrate how the PDS-Algorithm works.

Example 6.2.

This example is a straightforward implementation of the following lemma (6.5) *using the graph* $\mathfrak{G}_\circ$ *shown in Fig.* (6.3*a*). *The graph* $\mathfrak{G}_\circ$ *has been divided into two equal subgraphs* $\mathfrak{G}_1$ *and* $\mathfrak{G}_2$ *each is a triangle, then*

$$n_d(\mathfrak{G}_1)n_d(\mathfrak{G}_2) = 5 \times 5 = 25.$$

The global vertices of $\mathfrak{G}_1$ and $\mathfrak{G}_2$ are v_2 and v_4 respectively (shown black in Fig. (6.3a)). As

$|\Re_1(v_2)| = |\Re_2(v_4)| = 1$, $|D_1(v_2)| = |D_2(v_4)| = 3$. (see Fig. 6.3), then

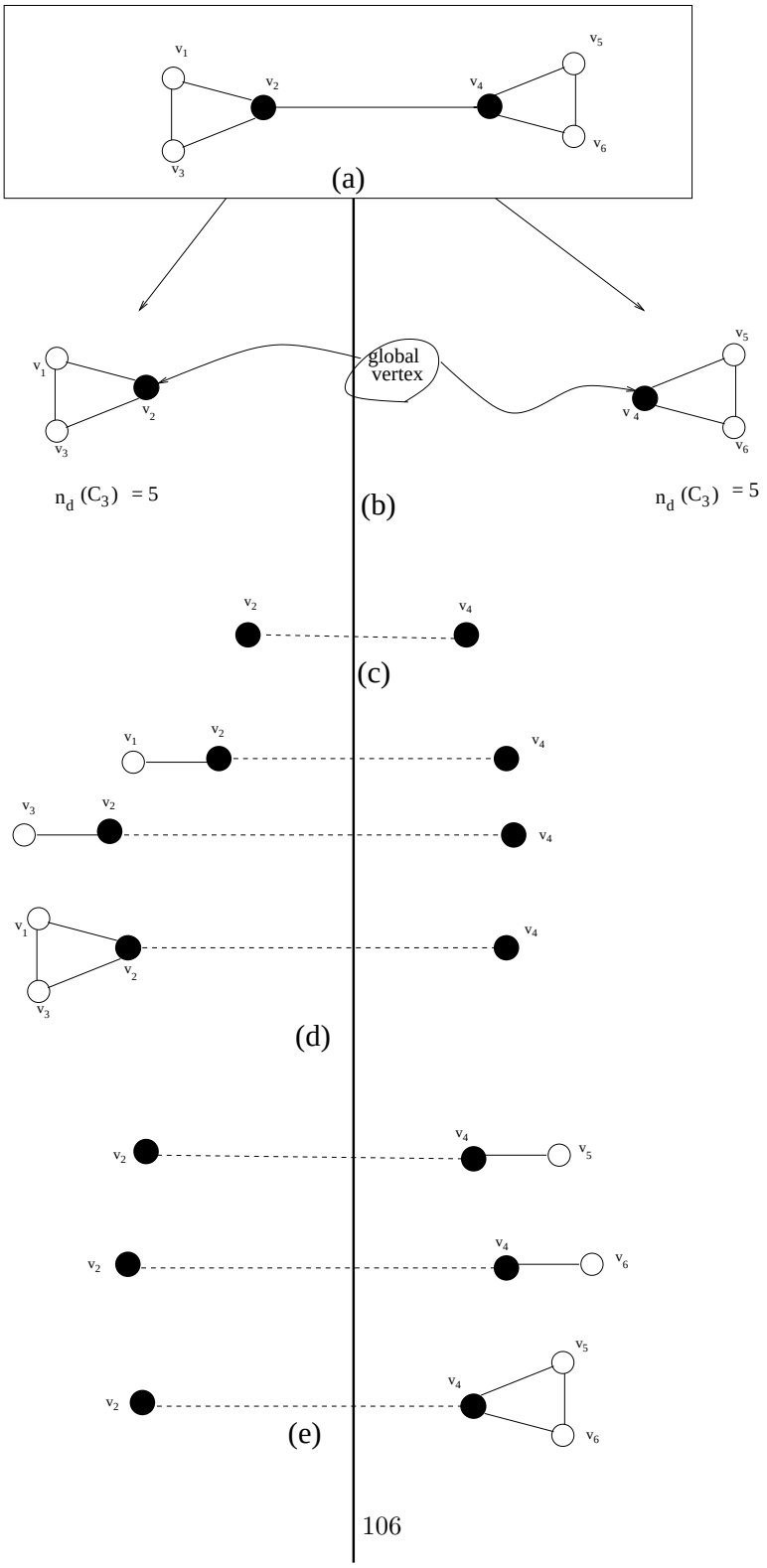


Figure 6.3: (a) A graph $\mathfrak{G}_o$ that contains v_2 and v_4 as the global vertices. (b) The two subgraphs $\mathfrak{G}_1$, $\mathfrak{G}_2$. (c) The derived subgraphs induced by $\Re_1(v_2)$ and $\Re_2(v_4)$, (d) The derived subgraphs induced by $\Re_2(v_4)$ and $D_1(v_2)$. (e) The derived subgraphs induced by $\Re_1(v_2)$ and $D_2(v_4)$.

$$n^{e}_{out}(\mathfrak{G}_1, \mathfrak{G}_2) = |\Re_1(v_2)||\Re_2(v_4)| + |D_1(v_2)||\Re_2(v_4)| + |\Re_1(v_2)||D_2(v_4)|.$$

$$n^{e=(v_2,v_4)}_{out}(\mathfrak{G}_1, \mathfrak{G}_2) = 1 \times 1 + 3 \times 1 + 1 \ \times 3 = 7.$$

Thus, by substituting in equation (6.11) we get:

$$n_d(\mathfrak{G}_o) = 25 + 7 = 32.$$

Lemma 6.5.

Let $V = V_1 \cup V_2$ be a partition of the vertex set $V(\mathfrak{G}_o)$ of a graph $\mathfrak{G}_o$. Let the subgraphs induced by V_1 and V_2 be denoted $\mathfrak{G}_1$ and $\mathfrak{G}_2$ respectively. There is only one bridge edge $e = (v, u)$ such that $v \in V(\mathfrak{G}_1)$ and $u \in V(\mathfrak{G}_2)$. Then

$$n_d(\mathfrak{G}_o) = n_d(\mathfrak{G}_1)n_d(\mathfrak{G}_2) + n^{e}_{out}(\mathfrak{G}_1, \mathfrak{G}_2) \tag{6.12}$$

And

$$n^{e}_{out}(\mathfrak{G}_1, \mathfrak{G}_2) = |\Re_2(u)||\Re_1(v)| + |D_1(v)||\Re_2(u)| + |\Re_1(v)||D_2(u)|. \tag{6.13}$$

Proof.

Consider an arbitrary derived subgraph $S \in D(\mathfrak{G}_o)$. If S does not contain the bridge edge e then $S \in D(\mathfrak{G}_1)$ or $S \in D(\mathfrak{G}_2)$ and hence S is counted in the number $n_d(\mathfrak{G}_1)n_d(\mathfrak{G}_2)$.

Let S contains the bridge edge $e = (v, u)$ which means that S contains the global vertices v and u. Therefore the edge e connects two subsets $s_1 \in V(\mathfrak{G}_1)$ and $s_2 \in V(\mathfrak{G}_2)$. Let us consider here all possible kinds of s_1 and s_2 :

Case 1: s_1 and s_2 are derived subgraphs, then S is counted in $n_d(\mathfrak{G}_1)n_d(\mathfrak{G}_2)$.

Case 2: s_1 and s_2 have the global vertices as the only isolated vertices, then $S \in S_1$.

Case 3: Let $s_1 \in D(\mathfrak{G}_1)$ and s_2 has an isolated vertex. If this isolated vertex is u then $S \in S_2$. Otherwise the subgraph S contains an isolated vertex. So that $S \notin D(\mathfrak{G}_o)$ and this contradicts our assumption.

Case 4: The last case is similar to Case 3 only we exchange the conditions of s_1 with the conditions of s_2. Let $s_2 \in D(\mathfrak{G}_2)$ and s_1 has an isolated vertex. If this isolated vertex is u then $S \in S_3$. Otherwise the subgraph S contains an isolated vertex. So that $S \notin D(\mathfrak{G}_o)$ and this contradicts our assumption.

Then the derived subgraph $S \in < D(\mathfrak{G}_1) \cup D(\mathfrak{G}_2) > \cup S_1 \cup S_2 \cup S_3$. This proves the first direction.

Conversely, if $S \in D(\mathfrak{G}_i)$, for any subgraph $\mathfrak{G}_i \subset \mathfrak{G}_o$, then as in the proof of Lemma(6.1) $S \in D(\mathfrak{G}_o)$. It follows that

$$D(\mathfrak{G}_{\mathfrak{o}}) = \{S : S \in < D(\mathfrak{G}_{\mathfrak{1}}) \cup D(\mathfrak{G}_{\mathfrak{2}}) >, \text{ or } S \in (S_1 \cup S_2 \cup S_3)\}$$

and

$$n_d(\mathfrak{G}_{\mathfrak{o}}) = n_d(\mathfrak{G}_{\mathfrak{1}}) n_d(\mathfrak{G}_{\mathfrak{2}}) + |\Re_2(u)||\Re_1(v) + |D_2(u)||\Re_1(v)| + |\Re_2(u)||D_1(v)|.$$

Since $n^e_{out}(\mathfrak{G}_{\mathfrak{1}}, \mathfrak{G}_{\mathfrak{2}})$ denotes the cardinality of the missing derived subgraph set, then

$$n^e_{out}(\mathfrak{G}_{\mathfrak{1}}, \mathfrak{G}_{\mathfrak{2}}) = |\Re_2(u)||\Re_1(v)| + |D_2(u)||\Re_1(v)| + |\Re_2(u)||D_1(v)|.$$

□

In summary, this subsection showed how to use the proposed PDS algorithm to find the value of $n^e_{out}(\mathfrak{G}_{\mathfrak{1}}, \mathfrak{G}_{\mathfrak{2}})$ in a parallel approach that divides the graph $\mathfrak{G}_{\mathfrak{o}}$ into two subgraphs. We showed a simple case in which the graph $\mathfrak{G}_{\mathfrak{o}}$ has been divided into two subgraphs $\mathfrak{G}_{\mathfrak{1}}$ and $\mathfrak{G}_{\mathfrak{2}}$ that are connected by only one bridge edge. In the following subsections more divisions will be considered and the algorithm will be compared to the serial method with respect to the running time.

6.5.2 Desired Divide Step

A better graph partition criterion seeks a small shared edge that partitions the vertices into roughly equal-sized pieces. If p subgraphs are required the partition

method is called a general $p-$way partition. An instance of graph partitioning that deserves special attention is the graph bisection problem. This is simply a variation on graph partitioning in which $\mathfrak{G}_\circ$ must be divided into two subsets. The most commonly used $p-$way partitioning method is recursive bisection. Unfortunately, although bisection seems considerably easier than general $p-$way partitioning, it is still NP-hard. Fortunately, certain special graphs always have small separators, which partition the vertices (in polynomial time) into balanced pieces. For any tree[1], there always exists a single vertex whose deletion partitions the tree so that no component contains more than $n/2$ of the original n vertices. Similarly, every necklace graph (Figure 6.3a and 6.7a) has a constant number of vertices whose deletion leaves two components with roughly equal size. Every planar graph has a set of vertices whose deletion leaves no component with more than $2n/3$ vertices. The bounded degree graphs [2] have a set of vertices whose deletion leaves two components with roughly equal size. Graphs embeddable in interval graphs have a small set of vertices whose deletion leaves two components with roughly equal-size. Our Divide Step runs on the above graphs and the graphs with n vertices and $m = O(n)$ edges. So that our parallel derived subgraph algorithm runs on the above graphs. The proposed algorithm uses a recursive bisection algorithm. In each phase the proposed algorithm uses a simple optimal bisection algorithm working in logarithmic time (see Fig. 6.4).

We use a very simple deterministic strategy to divide the input graph into

[1]The star grah $K(1, n)$ is not included here because there is a formula which already computes the number $n_d(K(1, n))$ see Lemma 6.3.

[2]A graph is bounded degree if the maximum degree of its vertices is bounded.

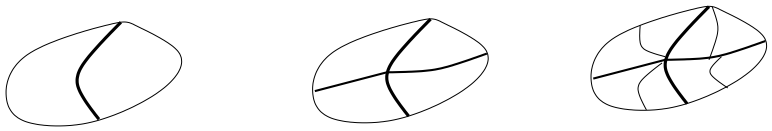


Figure 6.4: An example demonstrating the use of recursive bisection which is used in our proposed algorithm.

two roughly equal-sized graphs. Let the input be a graph $G = (V, E)$ with n verteices and m edges. Let V^g be the set of global vertices. For all $S \subset V(G)$, define the neighborhood of S to be $N(S) = \{i \in V : \exists\ j \in S,\ (i, j) \in E(G)\}$. The output consists of two vertex subsets S and $V - S$ with a roughly equal-size.

1: $S \longleftarrow V^g$
2: $I \longleftarrow N(S)$
3: **while** $|S| < n/2$ **do**
4: $\quad S \longleftarrow S \cup I$
5: $\quad I \longleftarrow N(I)$
6: **end while**

Algorithm 17: BISECTION($V(G), V^g$)

The first two steps can be executed in constant. The body of the while loop is executed at most $(\log n)$ times, where each execution takes constant time. Therefore the total expected running time is $O(\log n)$.

6.5.3 Description of the Parallel Derived Subgraphs Algorithm

In this subsection, we describe the first parallel efficient derived subgraph algorithm that finds and counts all derived subgraphs of a graph $\mathfrak{G}_\circ = (V, E)$ which has n vertices and m edges. The input graph $\mathfrak{G}_\circ$ will be divided into a number of subgraphs with at least one shared edge connecting every two of them(minimizing the shared edges). Let $\mathfrak{G}_\circ = (V, E)$ be a graph with

vertex set $V(\mathfrak{G}_o)$ on n vertices and an edge set $E(\mathfrak{G}_o)$ on m edges. Let the mapping $\Psi : V(\mathfrak{G}_o) \longrightarrow \{1, 2, ..., l\}$ represent the assignment of the vertex set $V(\mathfrak{G}_o)$ to the set $\{1, 2, ..., l\}$. The mapping Ψ returns the number l of the partition to which each vertex belongs. We define the set of local vertices by $V^l = \{v|\ \Psi(v) = \Psi(u)\ \ \forall\ (v, u) \in E(\mathfrak{G}_o)\}$ and the set of global vertices by $V^g = \{v|\ \exists\ (v, u) \in E$ with $\Psi(v) \neq \Psi(u)\}$. Given that, we consider the graph partitioning $V(\mathfrak{G}_o) = V_1 \cup V_2$ and $E(\mathfrak{G}_o) = E_1 \cup E_2$, where E_i refers to the set of unordered pairs of distinct vertices of V_i for $i = 1, 2$. Then we assume that the subsets V_1 and V_2 have nearly equal numbers of vertices while the shared edges between them is minimum.

As shown in Fig. (6.5) all edges such as $e_i = (v, u)$ where $v \in V_i^g$ and $u \in V_j^g$ are shared edges. The set of all shared edges H is given as follows:

$$H = \{e_i : e_i = (v, u), v \in V_i \text{ and } u \in V_j\ \ \forall\ i \neq j\}.$$

Applying the PDS algorithm on $\mathfrak{G}_o$ are obtained the following four non-identical subgraphs:

$\mathfrak{G}_1$ is the induced subgraph on V_1,

$\mathfrak{G}_2$ is the induced subgraph on V_2,

$\bar{\mathfrak{G}_1}$ is the induced subgraph on $V_1 \cup < H >$,

$\bar{\mathfrak{G}_2}$ is the induced subgraph on $V_2 \cup < H >$.

The two subgraphs $\mathfrak{G}_1$ and $\mathfrak{G}_2$ are called the partite of $\mathfrak{G}_o$, while $\bar{\mathfrak{G}_1}$ and $\bar{\mathfrak{G}_2}$

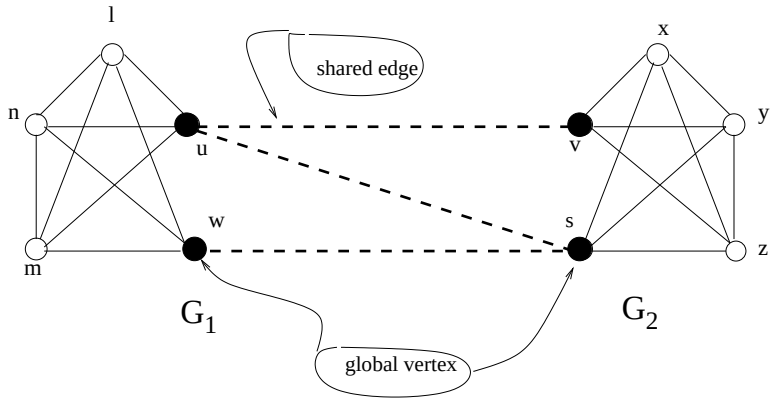


Figure 6.5: A graph $\mathfrak{G}_{\mathfrak{o}}$ has two subgraphs $\mathfrak{G}_{\mathfrak{1}}$ and $\mathfrak{G}_{\mathfrak{2}}$, and $V_1 = \{x, y, z, s, v\}$, $V_2 = \{l, m, n, w, u\}$. The shared edges are dotted

are called the assistant subgraphs. The set of global vertices of the partite subgraphs $\mathfrak{G}_{\mathfrak{1}}$ and $\mathfrak{G}_{\mathfrak{2}}$ were defined at the beginning of this subsection. The set of global vertices of the assistant subgraphs $\bar{\mathfrak{G}}_{\mathfrak{1}}$ and $\bar{\mathfrak{G}}_{\mathfrak{2}}$ are V_2^g and V_1^g respectively.

We denote the missing derived subgraphs by $D_{out}^{\mathfrak{h}}(\mathfrak{G}_{\mathfrak{1}}, \mathfrak{G}_{\mathfrak{2}})$ for all $\mathfrak{h} \subset H$, and its cardinality is $n_{out}^{\mathfrak{h}}(\mathfrak{G}_{\mathfrak{1}}, \mathfrak{G}_{\mathfrak{2}})$. The partite vertex set incident with the edge set $\mathfrak{h}$ consists of $s_i \subset V_1^g$ and $s_j \subset V_2^g$. Let We define the following graph sets:

1- The set $D_i(s_i)$ gives all derived subgraphs of $\mathfrak{G}_{\mathfrak{i}}$ which contain all the global vertices of s_i.

$D_i(s_i) = \{G |\ G \in D(\mathfrak{G}_{\mathfrak{i}}),\ \text{ s.t. for each global vertex } v \in G \Longrightarrow\ v \in s_i\}$.

2- The set $\Re_i(s_i)$ contains all subgraphs of $\mathfrak{G}_{\mathfrak{i}}$ which contain all vertices in s_i as the only pairwise nonadjacent vertices.

$\Re_i(s_i) = \{G |\ G \subset \mathfrak{G}_{\mathfrak{i}},\ \text{ s.t. } \forall\ v \in G\ \ deg(v) = 0\ \text{ if and only if }\ v \in s_i\}$.

3- The set $\bar{D}_i(s_i)$ gives all derived subgraphs of $\bar{\mathfrak{G}}_{\mathfrak{i}}$ which contain all the global vertices of s_i.

$\bar{D}_i(s_i) = \{G |\ G \in D(\bar{\mathfrak{G}}_{\mathfrak{i}}),$ s.t. for each global vertex $v \in G \Longrightarrow\ v \in s_i\}$.

4- The set $\bar{\Re}_i(s_i)$ contains all subgraphs of $\bar{\mathfrak{G}}_{\mathfrak{i}}$ which contain all vertices in s_i as the only pairwise nonadjacent vertices.

$\bar{\Re}_i(s_i) = \{G |\ G \subset \bar{\mathfrak{G}}_{\mathfrak{i}},$ s.t. $\forall\ v \in G\ \ deg(v) = 0$ if and only if $v \in s_i\}$.

The set $D^{\mathfrak{h}}_{out}(\mathfrak{G}_{\mathfrak{1}}, \mathfrak{G}_{\mathfrak{2}})$ will be one of the following three sets:

The Set S_1: The set of all non-empty derived subgraphs induced by all sets $\Re_1(s_i)$ and $\Re_2(s_j)$ together.

The Set S_2: The set of derived subgraphs induced by all derived subgraphs of $D_1(\mathfrak{s}_{\mathfrak{i}})$ that contain s_i, and all subgraphs $\Re_2(s_j)$ together.

The Set S_3: The set of derived subgraphs induced by all derived subgraphs of $D_2(\mathfrak{s}_{\mathfrak{j}})$ that contain s_j, and all subgraphs $\Re_1(s_i)$ together.

The analogue of Lemma (6.5) is the following Lemma (6.6). It proves the correctness of the steps which are executed by the proposed algorithm. So the Lemma does not present the number $n_d(\mathfrak{G}_{\mathfrak{o}})$ as a function in the number of vertices or the number of edges in the input graph, rather the way to obtain all

derived subgraph such that every distinct derived subgraph is generated only once.

Lemma 6.6.

Let $V = V_1 \cup V_2$ be a partition of the vertex set $V(\mathfrak{G}_o)$ of a graph $\mathfrak{G}_o$. Let the set of bridge edges is denoted H. The subgraphs induced by V_1, V_2, $V_1 \cup H$, and $V_2 \cup H$ are denoted $\mathfrak{G}_1$, $\mathfrak{G}_2$, $\bar{\mathfrak{G}}_1$, and $\bar{\mathfrak{G}}_2$ respectively. If $|V_1^g| \leq |V_2^g|$ and for any subset $s_i \in V_1^g$ the adjacent subset of V_2^g is s_j. Then

$$n_d(\mathfrak{G}_o) = n_d(\mathfrak{G}_1)n_d(\mathfrak{G}_2) + n_{out}^H(\mathfrak{G}_1, \mathfrak{G}_2) \tag{6.14}$$

where

$$n_{out}^H(\mathfrak{G}_1, \mathfrak{G}_2) = \sum_{\forall\ s_i \subseteq\ V_1^g} \Big(|\bar{D}_2(s_i)||\Re_1(s_i)| + \{|\bar{D}_1(s_j)| - |\Re_1(s_i)|\}|\Re_2(s_j)| \Big). \tag{6.15}$$

Proof.

The same proof as for Lemma (6.5), except we consider the subset $s_i \in V_1^g$ and $s_j \in V_2^g$ instead of v and u.

Consider an arbitrary derived subgraph $S \in D(\mathfrak{G}_o)$. If S does not contain a bridge edge e then $S \in D(\mathfrak{G}_1)$ or $S \in D(\mathfrak{G}_2)$ and hence S is counted in the number $n_d(\mathfrak{G}_1)n_d(\mathfrak{G}_2)$.

Let S contains a set of bridge edges $\mathfrak{h}$ which means that S contains two sets of global vertices $s_i \in V_1^g$ and $s_j \in V_2^g$. Therefore the edge set $\mathfrak{h}$ connects two

subgraphs $\mu_1 \in V(\mathfrak{G}_1)$ and $\mu_2 \in V(\mathfrak{G}_2)$. Let us consider here all possible types of μ_1 and μ_2.

Case 1: μ_1 and μ_2 are derived subgraphs, then S is counted in $n_d(\mathfrak{G}_1)n_d(\mathfrak{G}_2)$.

Case 2: μ_1 and μ_2 have the global vertices s_i and s_j respectively as the only isolated vertices, then $S \in S_1$. Then

$$|S_1| = |\Re_1(s_i)||\Re_2(s_j)| \tag{6.16}$$

Case 3: μ_1 induces a derived subgraph of $\mathfrak{G}_1$, and the subgraph induced on μ_2 has a set of isolated vertices, s_j . If s_j is a set of pairwise nonadjacent vertices in $\mathfrak{G}_2$ then $S \in S_2$. Otherwise the subgraph S contains isolated vertices and hence $S \notin D(\mathfrak{G}_0)$, which contradicts our assumptions. For case 3 we estimate the number of derived subgraphs as follows.

We introduce the assistant subgraph $\bar{\mathfrak{G}}_1$. The derived subgraphs induced by joining the subgraphs $\bar{D}_1(s_j)$ and $\Re_2(s_j)$ together belong to the graphs of types S_1 and S_2. The derived subgraphs belong to S_1 and S_2 can be induced from joining the subgraphs $\bar{D}_1(s_j)$ and $\Re_2(s_j)$ together. So that we can write

$$|\bar{D}_1(s_j)||\Re_2(s_j)| = |S_1| + |S_2|$$

The subgraph set S_1 is not included in case 3 then the number of derived

subgraphs that comply case 3 is

$$|S_2| = |\bar{D}_1(s_j)||\Re_2(s_j)| - |S_1| \tag{6.17}$$

Next we consider the subgraphs of the type S_3 in case 4 below.

Case 4: μ_2 induces a derived subgraph of $\mathfrak{G}_2$, and the subgraph induced on μ_1 has a set of isolated vertices, s_i . If s_j is a set of pairwise nonadjacent vertices in $\mathfrak{G}_1$ then $S \in S_3$. Otherwise the subgraph S contains isolated vertices and hence $S \notin D(\mathfrak{G}_o)$, which contradicts our assumptions. For case 4 we estimate the number of derived subgraphs as follows.

We introduce the assistant subgraph $\bar{\mathfrak{G}}_2$. The derived subgraphs induced by joining the subgraphs $\bar{D}_2(s_i)$ and $\Re_1(s_i)$ together belong to the graphs of types S_1 and S_3. The derived subgraphs belong to S_1 and S_3 can be induced from joining the subgraphs $\bar{D}_2(s_i)$ and $\Re_1(s_i)$ together. So that we can write

$$|\bar{D}_2(s_i)||\Re_1(s_i)| = |S_1| + |S_3|$$

The subgraph set S_1 is not included in case 4 then the number of derived subgraphs that comply case 4 is

$$|S_3| = |\bar{D}_2(s_i)||\Re_1(s_i)| - |S_1| \tag{6.18}$$

It follows from the last three cases (2, 3, 4) that all derived subgraphs $D^{\mathfrak{h}}_{out}(\mathfrak{G}_1, \mathfrak{G}_2)$

- that contain the edge set $\mathfrak{h}$ and the two subsets of global vertices s_i and s_j - belong to one of the sets S_1, S_2, or S_3. Then

$$n^{\mathfrak{h}}_{out}(\mathfrak{G}_1, \mathfrak{G}_2) = |S_1| + |S_2| + |S_3| \tag{6.19}$$

From equations (6.16), (6.17) and (6.18) it follows that

$$n^{\mathfrak{h}}_{out}(\mathfrak{G}_1, \mathfrak{G}_2) = |\bar{D}_1(s_j)||\Re_2(s_j)| + |\bar{D}_2(s_i)||\Re_1(s_i)| - |\Re_1(s_i)||\Re_2(s_j)| \tag{6.20}$$

Conversely, if $S \in D(\mathfrak{G}_i)$, for any subgraph $\mathfrak{G}_i \subset \mathfrak{G}_o$, then as in proof of Lemma (6.1) $S \in D(\mathfrak{G}_o)$. Then

$$D(\mathfrak{G}_o) = \{S : S \in D(< \mathfrak{G}_1 \cup \mathfrak{G}_2 >), \text{ or } S \in D^{\mathfrak{h}}_{out}(\mathfrak{G}_1, \mathfrak{G}_2) \ \forall\, s_i \text{ incident with } \mathfrak{h}\}.$$

And

$$n_d(\mathfrak{G}_o) = n_d(\mathfrak{G}_1) n_d(\mathfrak{G}_2) + n^H_{out}(\mathfrak{G}_1, \mathfrak{G}_2)$$

□

6.5.4 The Model of Computations

In our parallel algorithm to count the derived subgraphs and recognize residual and non-residual edges, we consider a pyramid with a base of size $\lceil n/\log n \rceil \times \lceil n/\log n \rceil = 2^{2\lceil \log n - \log\log n \rceil}$ that connects $p = \lceil (4n^2/\log^2 n - 1)/3 \rceil$ processors.

These processors respectively form $\lceil \log n - \log\log n \rceil + 1$ meshes of size

$$\lceil n/\log n \rceil \times \lceil n/\log n \rceil,\ \lceil n/(2\ \log n) \rceil \times \lceil n/(2\ \log n) \rceil, ...,\ 1 \times 1.$$

These meshes are stacked one on top of the other in decreasing order of size and are interconnected. Each processor has a unique index $P(l,i,j)$ where $0 \leq l \leq \lceil \log n - \log\log n \rceil$, and $0 \leq i,j \leq \lceil \log n \rceil - 1$. The following relationships can be defined for a pyramid:

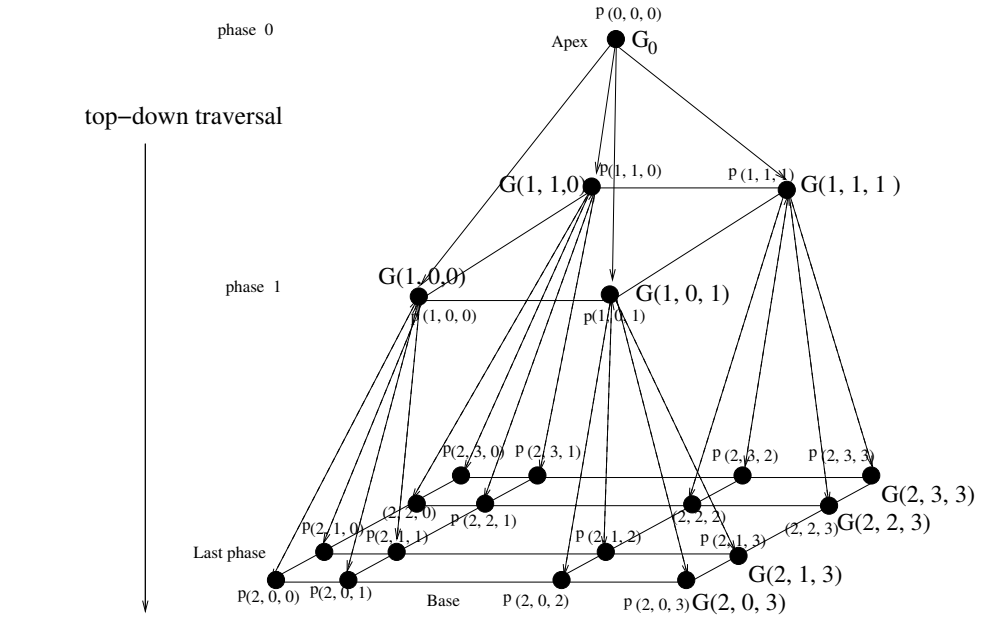
1- The father of the processors $P(l,i,j)$ is the processor $P(l-1,\ \lceil i/2\ \rceil,\ \lceil j/2\ \rceil)$.

2- The sons of the processor $P(l,i,j)$ are the processors $P(l+1, 2i+1, 2j+1)$, $P(l+1, 2i+1, 2j)$ and $\bar{P}(l+1, 2i+1, 2j+1)$, $\bar{P}(l+1, 2i+1, 2j)$.

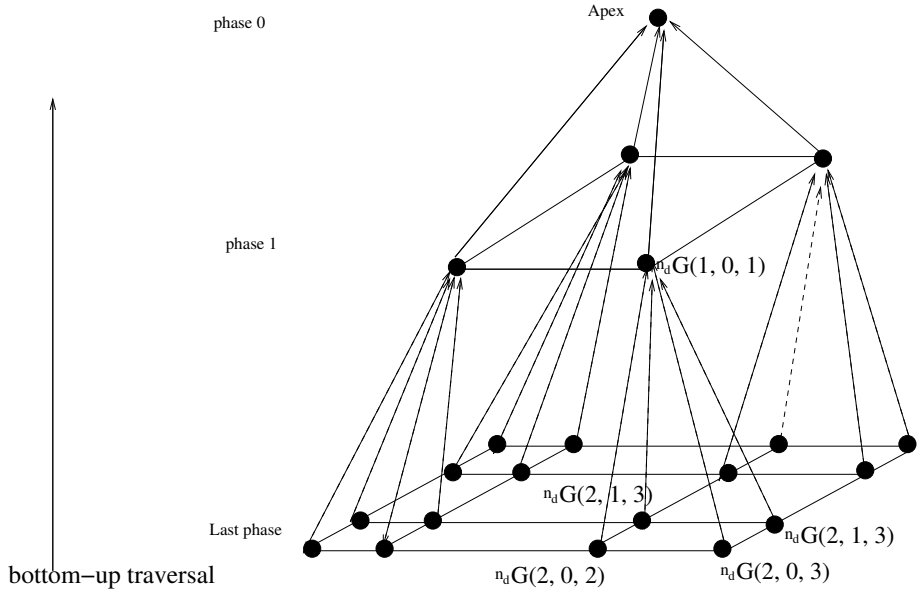
6.5.5 The Algorithm PDS($\mathfrak{G}_o, n, m$)

The parallel derived subgraph algorithm (denoted PDS($\mathfrak{G}_o, n, m$)) assumes that the input graph has $n > 3$ vertices and m edges. The input graph has only one component. If the graph $\mathfrak{G}_o$ has more than one component C_i, then we independently find in parallel the set $D(C_i)$ for all components, and hence $n_d(\mathfrak{G}_o) = \prod_i n_d(C_i)$. The algorithm represents $\mathfrak{G}_o$ by its adjacency matrix A which is fed to the apex processor $P(0,0,0)$.

The algorithm has two kinds of traversal. The first is the *top-down* traversal, as shown in Fig.(6.6a) in which the graph in phase l on n_l vertices is divided



(a)



(b)

Figure 6.6: (a) Pyramid-connected computer (top-down traversal) to find and count the set of all derived subgraphs of $\mathfrak{G}_o$, we assign processor $P(l,i,j)$ to the subgraph $G(l,i,j)$, (b) Pyramid-connected computer(bottom-up traversal).

into four subgraphs in phase $l+1$ on $n_l/2$ (or $n_l/2+\epsilon$) vertices. The second is the *bottom-up* traversal as shown in Fig.(6.6*b*), in which each internal node waits for the result of the work done by its children to begin finding the set of all derived subgraphs of its associated subgraph.

We shall describe a single phase of the algorithm. We begin with the partitioning described above, where the processor $P(0,0,0)$ at the apex level. Let $G(0,0,0)$ denote the input graph which is fed to the apex processor $P(0,0,0)$. The son processor $P(l,i,j)$ is assigned to the subgraph $G(l,i,j)$ which is obtained from $G(0,0,0)$ by executing the bisection subroutine l times.

To start the phase l, we use some processors from the available p processor for the algorithm. We assign processor $P(l,i,j)$ to the subgraph $G(l,i,j)$. The son processors $P(l+1,2i+1,2j+1)$, $P(l+1,2i+1,2j)$ are assigned to the two partite subgraphs $G(l+1,2i+1,2j+1)$ and $G(l+1,2i+1,2j)$. The son processors $\bar{P}(l+1,2i+1,2j+1)$ and $\bar{P}(l+1,2i+1,2j)$ are assigned to the two assistant subgraphs $\bar{G}(l+1,2i+1,2j+1)$ and $\bar{G}(l+1,2i+1,2j)$ respectively.

The processor $P(l,i,j)$ has the global set $V^g_{l,i,j}$. The processors $P(l+1,2i+1,2j+1)$ and $\bar{P}(l+1,2i+1,2j))$ have the global set $V^g_{l+1,2i+1,2j+1}$. The set of global vertices of the two processors $P(l+1,2i+1,2j)$, and $\bar{P}(l+1,2i+1,2j+1)$ is $V^g_{l+1,2i+1,2j}$. The processor $P(l,i,j)$ and its sons all together utilize the PDS algorithm to find the set of all derived subgraphs of the graph $G(l,i,j)$. This can be done as follows:

I- The two processors $P(l+1, 2i+1, 2j+1)$ and $P(l+1, 2i+1, 2j)$ recursively call PDS algorithm to do the following:

1− Find the set of all derived subgraphs and their numbers $n_d(G(l+1, 2i+1, 2j+1))$, $n_d(G(l+1, 2i+1, 2j))$.

2− Find the two arrays $Y_{l+1,2i+1,2j+1}$ and $Y_{l+1,2i+1,2j}$, where the $i-$th position contains the numbers $|\Re_{l+1,2i+1,2j+1}(s_i)|$ and $|\Re_{l+1,2i+1,2j}(s_i)|$, for all $s_i \subseteq V^g_{l+1,2i+1,2j+1}$ or $s_i \subseteq V^g_{l+1,2i+1,2j}$.

3− If the portion subgraph contains the global vertices $V^g_{l,i,j}$ of the phase l, the processor finds the corresponding arrays $Y_{l,i,j}$ and $X_{l,i,j}$, where the $k-$th position contains the numbers $|\Re_{l,i,j}(s_k)|$ and $|D_{l,i,j}(s_k)|$ respectively for all $s_k \subseteq V^g_{l,i,j}$.

4− Find the number of subgraphs $E_{l+1,2i+1,2j+1}(e)$ and $E_{l+1,2i+1,2j}(e)$ that contain an edge e.

5− Then every processor will send its local output to the father processor $P(l, i, j)$.

The recursive process continues until the number of vertices in the subgraphs is less than or equal to $\log n + \epsilon$. In this case, the processors will

call the MSDS-Procedure (see subsection 6.4.2) to compute the arrays $Y_{l+1,2i+1,2j+1}$, and $Y_{l+1,2i,2j}$, and the set of derived subgraphs of the associated subgraphs. The following code describes the above tasks of the processors:

1: **for** each processor in parallel **do**
2: Induce its subgraph $G(l+1,2i+1,2j+1)$ and $G(l+1,2i+1,2j)$
3: **if** $n_i > \log n + \epsilon$, for some constant $\epsilon > 0$ **then**
4: Recursively call PDS-Algorithm
5: **else**
6: call $MSDS(G(l+1,2i+1,2j+1), n_i, m_i)$ or $MSDS(G(l+1,2i+1,2j), n_i, m_i)$
7: **end if**
8: **end for**
9: Send to the father processor.
1- $n_d(G(l+1,2i+1,2j+1))$ and $n_d(G(l+1,2i+1,2j))$,
2- $Y_{l+1,2i+1,2j+1}$ and $Y_{l+1,2i+1,2j}$
3- $E_{l+1,2i+1,2j+1}$ and $E_{l+1,2i+1,2j}$
10: **if** the induced subgraphs have a subset $s_k \subseteq V^g_{l,i,j}$ **then**
11: Send the arrays $Y_{l,i,j}$ and $X_{l,i,j}$ to the father processor
12: **end if**

Algorithm 18: Processors $P(l+1,2i+1,2j+1)$, $P(l+1,2i+1,2j)$

II- The two processors $\bar{P}(l+1,2i+1,2j+1)$ and $\bar{P}(l+1,2i+1,2j)$ recursively call PDS to do the following:

1− Find the two arrays $\bar{X}_{l+1,2i+1,2j+1}$ and $\bar{X}_{l+1,2i+1,2j}$, where the $i-$th position contains the numbers $|\bar{D}_{l+1,2i+1,2j+1}(s_i)|$ and $|D_{l+1,2i+1,2j}(s_i)|$ respectively, for all $s_i \subseteq V^g_{l+1,2i+1,2j+1}$ or $s_i \subseteq V^g_{l+1,2i+1,2j}$.

2− If the assistant subgraph contains the global vertices $V^g_{l,i,j}$ of the previous phase l, the processor finds the corresponding arrays $\bar{Y}_{l,i,j}$ and $\bar{X}_{l,i,j}$,

where the $k-$th position contains the numbers $|\bar{\Re}_{l,i,j}(s_k)|$ and $|\bar{D}_{l,i,j}(s_k)|$ for all $s_k \subseteq V^g_{l,i,j}$.

3− Find the number of subgraphs $\bar{E}_{l+1,2i+1,2j+1}(e)$ and $\bar{E}_{l+1,2i+1,2j}(e)$ that contain an edge e.

4− Then every processor will send its local output to the father processor $P(l,i,j)$.

The recursive process continues until the number of vertices in the subgraphs is less than or equal to $\log n + \epsilon$. In this case, the processors will call the MSDS-algorithm (see subsection 6.4.2) to return the arrays $\bar{X}_{l+1,2i+1,2j+1}$, and $\bar{X}_{l+1,2i+1,2j}$, and the set of derived subgraphs of presenting subgraphs.

1: **for** each processor in parallel **do**
2: Induce its subgraph $G_{l+1,2i+1,2j}$ or $G_{l+1,2i,2j}$
3: **if** $n_i > \log n + \epsilon$, for some constant $\epsilon > 0$ **then**
4: Recursively call PDS-Algorithm
5: **else**
6: call MSDS algorithm
7: **end if**
8: **end for**
9: Send $\bar{X}_{l+1,2i+1,2j}$, $\bar{X}_{l+1,2i,2j}$, $\bar{E}_{l+1,2i+1,2j+1}$ and $\bar{E}_{l+1,2i+1,2j}$ to the father processor.
10: **if** The induced subgraphs have a subset $s_j \subseteq V^g_{l,i,j}$ **then**
11: Send the arrays $Y_{l,i,j}$ and $X_{l,i,j}$ to the father processor
12: **end if**

Algorithm 19: Processors $\bar{P}(l+1,2i+1,2j)$, and $\bar{P}(l+1,2i,2j)$

III- The processor $P(l,i,j)$ receives $n_d(G(l+1,2i+1,2j+1))$, $n_d(G(l+1,2i+1,2j))$ and the arrays of kind X and Y that were computed by its sons.

The tasks executed by processor $P(l,i,j)$ are the following:

1− Processor $P(l,i,j)$ stores the product of $n_d(G(l+1,2i+1,2j+1))$, $n_d(G(l+1,2i+1,2j))$ in the register *total*.

2− The processor computes the number $n^H_{out}(G(l+1,2i+1,2j+1), G(l+1,2i+1,2j))$. The processor adds this number to the value in the register *total*.

3− The processor finds the graph sets $D_{(l,i,j)}(s_k)$ and $\Re_{l,i,j}(s_k)$ for any subset $s_k \subseteq V^g_{l,i,j}$ of its global vertices and stores their numbers in two arrays of kind X and Y respectively. The $k-$th position of X and Y represents the number $|D_{l,i,j}(s_k)|$ and $|\Re_{l,i,j}(s_k)|$ respectively. The sizes of those arrays are constant because the number of global vertices is constant, so that the computing of this two arrays executes constant time. The processor sends X and Y to its father processor in the phase $l-1$.

If the set $s_k \subseteq V^g_{l,i,j}$ and e are contained in the portion $G(l+1,2i+1,2j+1)$, we compute the set $\Re_{l,i,j}(s_i)$ as follows:

Since there is no common vertex between the global vertices in phase l and the global vertices in phase $l+1$, the subgraphs of kind $\Re_{l,i,j}(s_k)$ consist of two sets; the first set is the set of all graphs containing a derived subgraph of $D(G(l+1,2i+1,2j))$ and a subgraph of $\Re_{l+1,2i+1,2j+1}(s_k)$, the cardinality

of this set is equal to

$$|\Re_{l+1,2i+1,2j+1}(s_k)|\{n_d(G(l+1,2+1,2j))-1\}$$

. The second set is the set of graphs containing some shared edges but do not contain any derived subgraph of $D(G(l+1,2i+1,2j))$, these subgraphs are $\bar{\Re}_{l+1,2i+1,2j+1}(s_k)$. So we get the following relation

$$|\Re_{l,i,j}(s_k)| = |\bar{\Re}_{l+1,2i+1,2j+1}(s_k)|+|\Re_{l+1,2i+1,2j+1}(s_k)|\{n_d(G(l+1,2+1,2j))-1\}$$

In the same way we can say that

$$|D_{l,i,j}(s_k)| = |\bar{D}_{l+1,2i+1,2j+1}(s_k)|+|D_{l+1,2i+1,2j+1}(s_k)|\{n_d(G(l+1,2i+1,2j))-1\}$$

and

$$E_{l,i,j}(e) = \bar{E}_{l+1,2i+1,2j+1}(e) + E_{l+1,2i+1,2j+1}(e)\{n_d(G(l+1,2i+1,2j)-1)\}.$$

If $e=(v,u)$ is shared edge directly from lemma(6.7) and when $s_i = v$ and its adjacent global set $s_j = v$, then

$$E_{l,i,j}(e) = n^e_{out}(G(l+1,2i+1,2j+l), G(l+1,2i+1,2j))$$

The following code describes the above tasks of the processor $P(l,i,j)$:

1: Processor $P(l,i,j)$ do
2: Assign the processor $P(l,i,j)$ to the input graph $G(l,i,j)$
3: Apply BISECTION$(V(G(l,i,j)), Vl, i, j^g)$ to find two subset $V(G(l+1,2i+1,2j+1))$ and $V(G(l+1,2i+1,2j)$ of $V(G(l,i,j))$.
4: Assign the global vertices $V^g_{l+1,2i+1,2j+1}$ and $V^g_{l+1,2i+1,2j}$
5: Receive the data from its sons processors
6: **for** each $s_j \subset V^g_{l,i,j}$ and $e \in G(l,i,j)$ **do**
7: Compute $x_{l,i,j}[j]$
8: Compute $Y_{l,i,j}[j]$
9: Compute $E_{l,i,j}[e]$
10: **end for**
{ Compute $Sum = n^H_{out}(G(l+1,2i+1,2j+1), G(l+1,2i+1,2j))$. }
11: $s \longleftarrow \phi$
12: **for** each vertex $v \in V^g_{l+1,2i+1,2j+1}$ **do**
13: $s \longleftarrow s \cup v$
14: $i \longleftarrow index.subset(s, |s|, |V^g_{l+1,2i+1,2j+1}|)$
15: **if** s_j is the global set adjacent with s_i by the edge set η **then**
16: $$Sum \longleftarrow Sum + n^{\eta}_{out}\Big(G(l+1,2i+1,2j+1), G(l+1,2i+1,2j)\Big)$$
17: **end if**
18: **end for**
19: $n_d(G(l,i,j)) \longleftarrow n_d(G(l+1,2i+1,2j+1))n_d(G(l+1,2i+1,2j)) + Sum$
20: If $E_{l,i,j}(e) > n_d(G(l,i,j))/2$ then e is a residual edge otherwise, e is a non-residual edge.

Algorithm 20: The PDS-Algorithm

With the above definitions and under the described assumptions, the following example shows how the parallel algorithm can be applied in the general case where the input graph is partitioned such that more than one shared edge is considered.

Example 6.3.

Consider the triangle necklace *graph, that is a cycle of length n with every vertex adjacent to one vertex of triangle. Figure* (6.7*a*) *shows a cycle of length* 3 *with every vertex adjacent to one vertex of a triangle. The graph* $\mathfrak{G}_\circ$ *has the partitions* G_1 *and* G_2 *in the first phase, where* G_1 *has* 4 *vertices, while a graph* G_2 *has* 8 *vertices (see Figure* (6.7*b*)*). The graph* G_2 *itself has the partitions* G_5 *and* G_6 *in the second phase. The number of derived subgraphs for each subgraph* G_1*,* G_5 *and* G_6 *is equal to* 9*.*

The bottom-up Phase:

We have to find the set of all derived subgraphs of the subgraph G_2*. This can be done as follows*

$$n_d(G_5)n_d(G_6) = 9 \times 9 = 81 \tag{6.21}$$

where G_5 *and* G_6 *are two subgraph of* G_2 *and the global vertices of them are*

$V_5^g = \{1\}$ *and* $V_6^g = \{2\}$

$|\Re_5(1)| = |\Re_6(2)| = 2$ *and* $|D_5(1)| = |D_6(2)| = 4.$

Based on that the number of the three subgraph sets S_1, S_2, and S_3 are:

$S_1 = |\Re_5(1)| \times |\Re_6(2)| = 4$,

$S_2 = |\Re_5(1)| \times |D_6(2)| = 8$,

$S_3 = |\Re_6(2)| \times |D_5(1)| = 8$.

$$n_{out}^{e=(1,2)}(G_5, G_6) = 4 + 8 + 8 = 20 \tag{6.22}$$

*From the above two equations (*6.21*) and (*6.22*), the number of derived subgraphs of G_2 is equal*

$$n_d(G_2) = 81 + 20 = 101 \tag{6.23}$$

The top-down Phase:

We have to find the set of all derived subgraphs of the subgraph $\mathfrak{G}_\circ$. This can be done as follows

$$n_d(G_1)n_d(G_2) = 9 \times 101 = 909 \tag{6.24}$$

The set of global vertex of the partition G_2 is $V_2^g = \{1, 2\}$, where $|\Re_2(1)| = |\Re_2(2)| = 10$, $|\Re_2(\{1, 2\})| = 0$

The set of global vertex of the partition G_1 is

$V_1^g = \{3\}$, *where* $|\Re_1(3)| = 2$.

The set of shared edges is $H = \{e_1, e_2, \}$ (see Fig. 6.7*b), where*

$$e_1 = (1, 3), \quad e_2 = (2, 3)$$

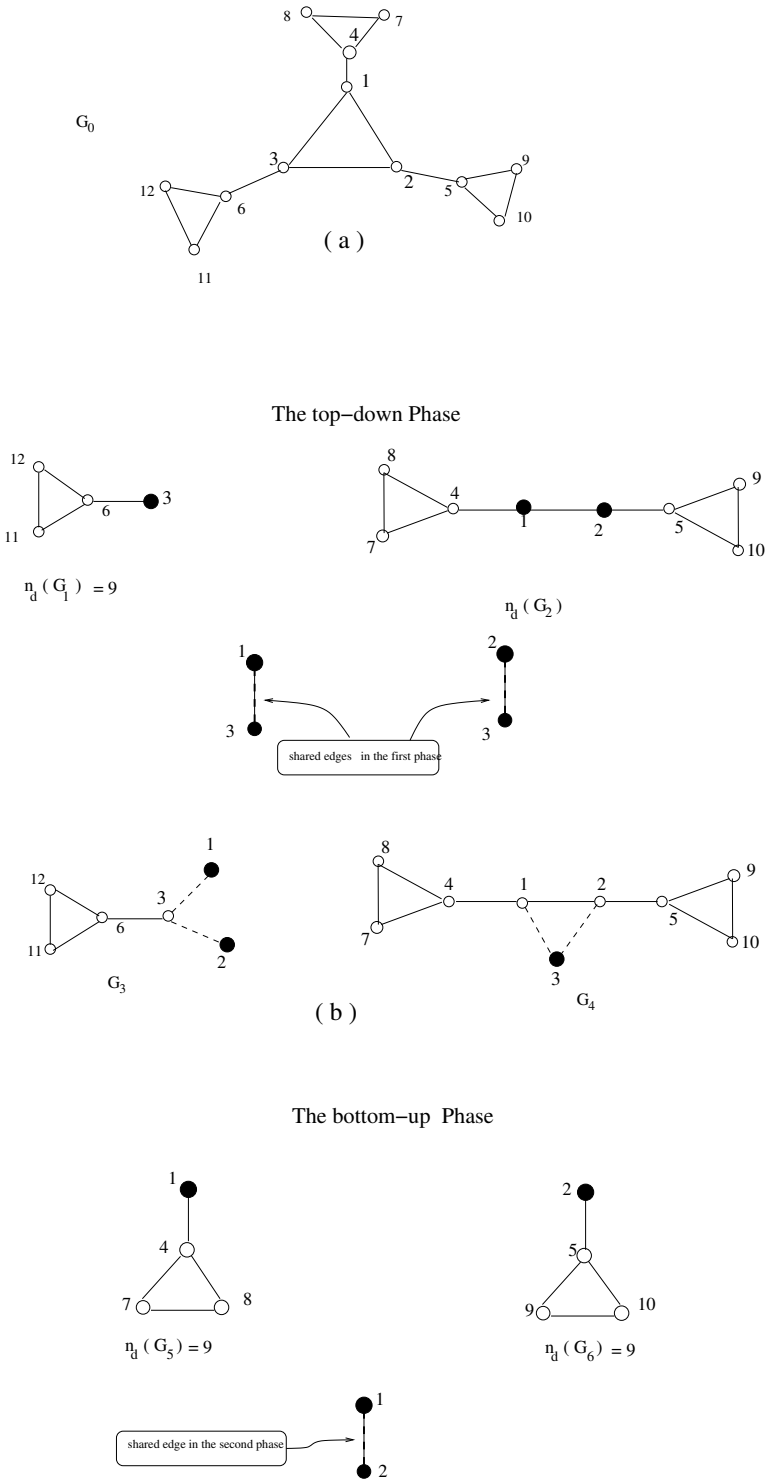


Figure 6.7: (a) The triangle necklace graph $\mathfrak{G}_o$, (b) the first phase of the algorithm in which the input graph $\mathfrak{G}_o$ has the partitions G_1 and G_2, the global vertices are marked black, the shared edges are dotted. The two assistant subgraphs G_3 and G_4 of this phase are shown, (c) the second phase in which the subgraph G_2 hasthe partitions G_5 and G_6, and the shared is dotted.

In order to calculate the three sets S_1, S_2, and S_3 we induced the two subgraph $G_3 = G_1 \cup H$ and $G_4 = G_2 \cup H$ (see Fig. 6.7b)

In the subgraphs G_3 the set of global vertices is $\{1, 2\}$ and the set of global vertices in G_4 is $\{3\}$. We found the following:

$|D_4(3)| = 96$,

$|D_3(1)| = |D_3(2)| = 6$, *and* $|D_3(\{1, 2\})| = 6$

Based on that, for each edge set $\{e_1\}$, $\{e_2\}$ $\{e_1, e_2\}$ the subgraph sets S_1 and S_2 are:

$$|D_4(3)| \times |\Re_1(3)| = 96 \times 2 = 192 \tag{6.25}$$

Note that

$$|D_4(3)| \times |\Re_1(3)| = \{|\Re_2(1)| + |\Re_2(2)| + |\Re_2(1,2)| + |D_2(1)| + |D_2(2)| + |D_2\{1,2\}|\} \times |\Re_1(3)|.$$

The set S_1 induced by the union of the subgraphs of G_1 which contain the global vertex 3 as the only isolated one and the subgraphs of G_1 which contain the vertices 1 or 2 or $\{, 2\}$ as the only isolated vertices. The set S_2 induced by the union of the derived subgraphs of G_2 which contain the global vertex 1 or 2 or $\{1, 2\}$ and the subgraphs of G_1 which contain the vertex 3 as the only isolated vertex.

But the subgraph set S_3 for each edge set $\{e_1\}$, $\{e_2\}$ $\{e_1, e_2\}$ is:

$$\{|D_3(1)| - |\Re_1(3)|\} \times |\Re_2(1)| = 4 \times 2 = 8 \tag{6.26}$$

$$\{|D_3(2)| - |\Re_1(3)|\} \times |\Re_2(2)| = 4 \times 2 = 8 \tag{6.27}$$

$$|D_3(\{1,2\})| - \{|\Re_1(3)|\} \times |\Re_2(\{1,2\})| = 4 \times 0 = 0 \tag{6.28}$$

The set S_3 induced by the union of the derived subgraphs of G_1 which contain the global vertex 3 and the subgraphs of G_1 which contain the vertices 1 or 2 or $\{,2\}$ as the only isolated vertices.

From the last four equations (6.25), (6.26), (6.27) and (6.28)

$$n^H_{out}(G_1, G_2) = 192 + 8 + 8 + 0 = 208 \tag{6.29}$$

From equation (6.24) and equation (6.29)

$$n_d(\mathfrak{G}_\circ) = n_d(G_1)n_d(G_2) + n^H_{out}(G_1, G_2) = 909 + 208 = 1117 \tag{6.30}$$

The example illustrates how we can get the set of all derived subgraphs of an undirected given graph when there is more than one shared edge. The example explains the method step by step which is used in the PDS algorithm.

6.5.6 The Work and The Running Time

In the following we analyze the work and time bounds for each phase of the algorithm.

The Number of Phases

The central difficulty in obtaining a fast parallel derived subgraph algorithm lies in the recursive structure of the algorithm. Since every phase l divides the input graph into two subgraphs each one has number of vertices equals to 1/2 the number of the vertices in the previous phase; in other words if the number of vertices of the input graph in phase l is equal to n_l, then the number of vertices of input graph in phase $l+1$ is equal to $n_l/2$. Since the initial number of vertices is n and the number of vertices in the last phase is $\log n$, the number of phases d is at most equal to $\lceil \log n - log \log n \rceil$. Then the total number of phases is $O(\log n)$ phase.

The Running Time per Phase

We can analyze the running time per phase l as follows: In the last phase the processors call the MSDS-Procedure. This calling requires $O(n_i 2^{n_i})$ time to return $n_d(G(l,i,j))$ and $X_{l,i,j}$, $Y_{l,i,j}$. The partitioning process continues until $n_i = \log n$. This means that the running time of the last phase is $O(n \log n)$. There are a constant number of global vertices in each phase and there are at most $O(\log n)$ phases. So that the total number of global vertices in the algorithm is $O(\log n)$. Then the phases of the algorithm except the last phase required at most $O(\log n)$ time. The expected running time of the bisection partitions is $O(\log n)$ time. Then the expected total running time of the phase is equal $O(n \log n)$.

From the above analysis it follows that there are at most $O(\log n)$ phases and each phase takes $O(n \log n)$ run time. Then the proposed parallel algorithm

for find all derived subgraphs of a given undirected graph and recognize the residual and non-residual edges in that graph has a total parallel running time $O(n \log^2 n)$.

The Number of Processors

The PDS algorithm uses a two dimensional pyramid-connected SIMD distributed memory computer which is described in section(6.5.3). The number of connecting processors in this model is equal to $p = \lceil (4n^2/\log^2 n - 1)/3 \rceil$ processor.

As given above the time bound for the computation of the set of all derived subgraphs and determining a residual and non-residual edges is $O(n \log^2 n)$, using a pyramid network of size $O(4n^2/\log^2 n - 1)/3$.

6.6 Conclusions

In this chapter we described the proposed PDS algorithm and showed how it can be used to find all derived subgraphs of a given graph and to determine the residual and non-residual edges in the given graph. We also estimated the running time for this parallel algorithm. The algorithm runs in $O(n \log^2 n)$ using a pyramid-connected SIMD distributed memory computer of size $O(4n^2/\log^2 n - 1)/3$ and the cost is $O(n^3)$.

A parallel algorithm is cost optimal when its cost matches the run time of the best known sequential algorithm for the same problem. The sequential run

time of (comparison based) derived subgraphs algorithm SDS is known to be $(n2^n)$ (see section (6.4)). The proposed parallel derived subgraphs algorithm PDS used $O(n^2/\log^2 n)$ processors for $O(n \log n)$ time. Our parallel derived subgraphs algorithm is cost optimal. The efficiency $E(n)$ of the parallel derived subgraphs algorithm is $O(2^n/n^2)$.

We explained scientific, commercial applications for the derived subgraphs problem and the residual and the non-residual edges in a given graph.

Bibliography

[1] S. G. Akl. *Parallel Computation: Models and Methods*, chapter 1 and 4, pages 7, 189. Alan Apt, 1997.

[2] D. H. Chandra and D. V. Sarwate. Computing connected components on parallel computers. *Communications of ACM*, 22:461–464, 1979.

[3] B. Chazelle. A faster deterministic algorithm for minimum spannig trees. *In FOCS*, pages 22–31, 1997.

[4] B. Chazelle. A minimum spanning tree algorithm with inverse-ackermann type complexity. *Journal of the ACM*, 47(6):1028–1047, November 2000.

[5] K. W. Chong, T. W. Lam, and Y. Han. On the parallel time complexity of undirected connectivity and minimum spanning trees. *SODA: ACM-SIAM Symposium on Discrete Algorithms(A Conference on Theoretical and Experimental Analysis of Discrete Algorithms)*, pages 243–250, 1999.

[6] K. W. Chong, T. W. Lam, and Y. Han. Concurrent threads and optimal parallel minimum spanning trees algorithm. *Journal of the ACM*, 48(2):297–323, March 2001.

[7] R. Cole, R. E. Tarjan, and P. N. Klein. A linear-work parallel algorithm for finding minimum spanning trees. *6th Annaul ACM Symposium on Parallel Algorithms and Architectures*, pages 11–15, 1994.

[8] R. Cole, R. E. Tarjan, and P. N. Klein. Finding minimum spanning forests in logarithmic time and linear work using random sampling. *In Proc. SPAA '96'*, pages 243–250, 1996.

[9] R. Cole and U. Vishkin. Approximate parallel scheduling: Applications to logarithmic-time optimal parallel graph algorithms. *Information and Computation*, 92(1):1–47, 1991.

[10] J. Edmonds. Matroids and the greedy algorithm. *Math. Programming*, 1:127–136, 1971.

[11] M. El-Zahar. A graph-theoretic version of the union-closed sets conjecture. *Graph Theory*, 26:155–163, 1997.

[12] M. J. Flynn. Some computer organizations and their effectiveness. *IEEE Trans. on Computers*, 24(9):948–960, Sept 1972.

[13] H. N. Gabow, Z. Galil, and T. H. Spencer. Efficient implementation of graph algorithms using contraction. *Journal of the ACM*, 36(3):540–572, July 1989.

[14] R. L. Graham and Pavol Hell. On the history of the minimum spanning tree problem. *Annals of the History of Computing*, 7(1):43–57, January 1985.

[15] J. Huan, W. Wang, and J. Prins. Efficient mining of frequent subgraphs in the presence of isomorphism. *University of North Carolina, Computer science Technique Report TR03-021*, 2003.

[16] J. Huan, W. Wang, and J. Prins. Spin: Mining maximal frequent subgraphs from graph databases. *KDD'04, August 22-25, Seattle, Washington, USA*, pages 581–586, 2004.

[17] J. Joseph. *An Introduction to Parallel Algorithms*. Addison-Wesley Publishing Company, 1992.

[18] D. R. Karger, R. E. Tarjan, and P. N. Klein. A randomized linear-time algorithm to find minimum spanning trees. *Journal of the ACM*, 42:321–328, 1995.

[19] M. Kuramochi and G. Karypis. Frequent subgraph discovery. *In Proc. International Conference on Data Mining (ICDM'01)*, pages 313–326, 2001.

[20] J. Lam, F. Chin, and I. Chen. Efficient parallel algorithms for some graph problems. *Communications of the ACM*, 25(9):659–665, 1982.

[21] M. Moussa. On the theory of graphs and their subgraphs. Master's thesis, Benha University, Benha, Egypt, April 1999.

[22] J. Nesetril, E. Milkov, and H. Nesetrilov. Otakar boruvka on minimum spanning tree problem (translation of both the 1926 papers, comments, history). *Discrete Math.*, 233(3-36), 2001.

[23] S. Pettie and V. Ramachandran. An optimal minimum spanning tree algorithm. *Journal of the ACM*, 49(1):16–34, January 2002.

[24] B. Poonen. Union-closed families. *Journal of Combin. Theory*, A59:253–268, 1992.

[25] M. J. Quinn. *Parallel Computing Theory and Practice*. McGraw-Hill Series in Computer Science, second edition, 1994.

[26] Y. Shloach and U. Vishkin. An $o(\log n)$ parallel connectivity algorithm. *J.of Algorithms*, 3:57–67, 1982.

[27] R. E. Tarjan. *Data Structures and Network Algorithms*. Society for Industrial and Applied Mathematics., 1983.

[28] R. E. Tarjan and M. L. Fredman. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34(3):596–615, July 1987.

[29] R. E. Tarjan and Uzi Vishkin. An efficient parallel biconnectivity algorithm. *SIAM Journal of COMPUT*, 14(4):863–874, 1985.

[30] X. Yan and J. Han. gspan: Graph-based substructure pattern mining. *In Proc.2002 international Conf. on Data Mining(ICDM'02)*, pages 721–724, 2002.

[31] Q. Zou, W. Chu, D. Johnson, and H. Chiu. A pattern decomposition(pd) algorithm for finding all frequent patterns in large datasets. *IEEE international Conference on Data Mining ICDM'01*, pages 673–674, 2001.