THE UNIVERSITY OF CHICAGO

LOWER BOUNDS FOR PARALLEL ALGORITHMS

A DISSERTATION SUBMITTED TO
THE FACULTY OF THE DIVISION OF THE PHYSICAL SCIENCES
IN CANDIDACY FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

BY
PRADYUT SHAH

CHICAGO, ILLINOIS
AUGUST 2001

Copyright © 2001 by Pradyut Shah
All rights reserved.

To my family

ACKNOWLEDGMENTS

There are many people who deserve my gratitude for all their help and support through the years.

Firstly, my advisor Ketan Mulmuley for all his advice and help. The faculty have been most helpful, and they deserve my heartful thanks – Laci Babai, Janos Simon, Lance Fortnow and Stuart Kurtz. I have learnt a lot from them.

On a personal level, I'd like to thank all the graduate students and friends with whom I've worked – Behfar, Daniel, Dieter, Ivona, Murali, Rahul, Ravi, Sandy, Soren, Steph, and Tom. Special thanks to Gina, Iris and Varsha for all their help.

Finally, I'd like to thank my parents and my family for their support.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS

$\mathcal{RL}$	languages computable in logarithmic space using randomization with one-sided error	22
$\mathcal{RNC}^k$	languages computable by a uniform family of randomized circuits of polynomial size and $O(\log^k n)$ depth	24
$\mathcal{RNC}$	$\bigcup_k \mathcal{RNC}^k$	24
$\mathcal{RP}$	languages computable in polynomial time using randomization with one-sided error	22
$\leq$	reduction between computational problems	26
$\leq^{\mathcal{L}}$	reduction between problems, computable in $\mathcal{L}$	28
$\leq^{\mathcal{NC}^1}$	reduction between problems, computable in $\mathcal{NC}^1$	28
$\leq^{\mathcal{P}}$	reduction between problems, computable in $\mathcal{P}$	28
$\leq_m$	many-one reduction between problems	27
$\leq_T$	Turing reduction between problems	28
$\leq_{f(n)-T}$	Turing reduction with at most $f(n)$ queries	28
$\rho(n,\beta)$	parametric complexity for input size n and bit-length β	33
$\Theta(g)$	functions with growth rate g	12
$\mathcal{ZPP}$	languages computable in polynomial time using randomization with zero-sided error	23

ABSTRACT

Proving the optimality of algorithms for important combinatorial problems and determining their intrinsic hardness properties requires finding strong lower bounds for them.

The model of computation we consider is the PRAM without bit operations. The model eliminates those operations that allow bit-extraction or updates of the bits of the individual registers, but provides the usual arithmetic, indirect referencing, conditional and unconditional branch operations at unit cost. We consider here an unbounded fan-in model, in which the operations $\{+, \min, \max\}$ can have unbounded fan-in at unit cost.

We show that computing the shortest path between two specified vertices in a weighted graph on n vertices cannot be solved in $o(\log n)$ time on an unbounded fan-in PRAM without bit operations using $n^{\Omega(1)}$ processors, even when the bit-lengths of the weights on the edges are restricted to be of size $O(\log^3 n)$. This shows that the matrix repeated-squaring algorithm for the SHORTEST PATH PROBLEM is optimal in the unbounded fan-in PRAM model without bit operations.

We also show that computing the maximum blocking flow (or equivalently, the maximum flow in a directed acyclic graph) cannot be computed in time $o(n^{1/4})$ using $2^{\Omega(n^{1/4})}$ processors in the same model, even when the inputs are restricted to be of length $O(n^2)$.

The lower bounds also hold (with slightly weaker parameters) in the case of randomized algorithms with two-sided error, and even when the running time of the algorithm is allowed to depend on the total bit-length.

The thesis also provides lower bounds for other restricted cases of these problems which are useful in practice. We also obtain as a corollary a weak lower bound on the problem of computing weighted matchings in bipartite graphs which is not known to have a fast parallel algorithm.

CHAPTER 1

INTRODUCTION

1.1 The Question of Lower Bounds

The question of lower bounds in computation is essentially a pessimistic one. It attempts to show what ***cannot*** be done computationally if one restricts certain computational resources.

Attempting to say what is impossible or cannot be done in science is extremely hazardous. The failures of the imagination are the stories of legend. August Comte, a philosopher of science, attempted to give an example of something science could never achieve. He declared in his *Cours de Philosophie Positif (1830-42)* that it would be impossible to determine the chemical composition of the distant stars. Within 20 years, the discovery of spectroscopy by Robert Bunsen and Gustav Kirchhoff (published in the *Annalen der Physik und der Chemie (1860)*) proved Comte's prediction wrong.

Mathematics by comparison (not being a science), can get a handle on impossibility by giving a proof thereof. Famous impossibility results include the impossibility of *"squaring the circle"*, or finding a general solution in radicals to a quintic equation.

The nature of computation lies somewhere in between these two extremes. On the one hand, we construct models of computation that allow us to solve computational problems using mathematical tools. On the other, however, we have to ensure that our models are *"realistic"*, *i.e.*, they should be physically realizable.

The design and analysis of algorithms for combinatorial problems that arise in practice count among the major contributions of Theoretical Computer Sci-

ence to the discipline of computing. The models developed for sequential computing resemble reality to a very close degree.

Unfortunately, there are exceedingly few lower bounds known for important computational problems on these standard models. Most of them have been obtained by simplifying the model of computation substantially in order to make the problem mathematically tractable. However, these models are largely unrealistic, not in the sense that they cannot be physically realized, but that no one would use such simple models to compute when more powerful ones are readily available.

The advent of parallel computing has had a significant impact on both the theory and praxis of computing. Once again, there are numerous models that try to capture the essence of parallelism without worrying about the tiny details that arise in practice. The success of such an approach in the sequential setting suggests that the same would be true in parallel.

Once again, many fast parallel algorithms have been designed for many practical problems. However, there are algorithms that can be computed fast sequentially that we are unable to parallelize efficiently.

The technique developed by Mulmuley [35] (discussed in Chapter 3) is the first one that gives strong lower bounds on the parallel running time of certain combinatorial problems. The model is a very slightly restricted version of the standard PRAM model.

1.2 Contributions of This Dissertation

In this dissertation, we use Mulmuley's technique to give strong lower bounds on the parallel complexity of certain combinatorial problems.

In Chapter 4, we give a strong lower bound on computing the shortest path between two vertices in a graph. Essentially, we prove that the *repeated-squaring* algorithm used to compute the answer to the problem fast in parallel is optimal. The result holds under fairly stringent restrictions on the input, and can be

extended to randomized algorithms, and also to the case where the input graph is sparse.

The proof is a substantial simplification and careful reworking of a proof of Carstensen [7, 8] which takes into account the size of the coefficients that arise.

The lower bound also extends to a weak lower bound on the problem of computing the maximum-weight perfect matching in a bipartite graph. This was the problem that we had originally set out to study because it has no fast parallel algorithm, yet is not known to be $\mathcal{P}$-complete (which would explain the lack of a fast parallel algorithm assuming that $\mathcal{P} \neq \mathcal{NC}$).

In Chapter 5, we give a lower bound on the parallel running time of the maximum blocking flow problem (or equivalently, the maximum flow problem in directed acyclic graphs).

Our work leaves a whole spectrum of questions unanswered. A detailed discussion of these can be found in Chapter 6.

CHAPTER 2

PRELIMINARIES

2.1 Computational Problems

Definition 2.1.1 A *computational problem* is a relation $R \subseteq I \times O$ between a set of inputs (or instances) I, and a set of outputs (or solutions) O.

The answer to an *instance of the problem* x is a solution y such that (x, y) belongs to the relation R. Since the problem is a relation, there may be many different answers corresponding to the same input. Usually in this case, we expect the *algorithm* that solves the problem to output any one of these solutions. At other times, we may be interested in counting the number of such solutions.

Definition 2.1.2 An *undirected graph* $G = (V, E)$ consists of a collection V of *vertices*, and a collection E of *unordered* pairs of vertices called *edges*.

A graph is represented diagrammatically by drawing a point for each vertex, and a line connecting the two vertices of each edge of the graph. An example is shown in Fig 2.1. A *path* between two pairs of vertices x and y is a sequence of vertices $(v_0 = x, v_1, v_2, \ldots, v_{k-1}, v_k = y)$ such that the graph contains edges between adjacent pairs of vertices. Pictorially, it is a way to get from vertex x to vertex y by following edges in the graph.

Example 2.1.3 (Graph Connectivity) The GRAPH CONNECTIVITY PROBLEM is the following: given an undirected graph G, and two vertices s and t, determine whether there is a path in G connecting s and t.

For the instance of the problem in Figure 2.1, the answer is "yes" if the two specified vertices are s and t, but "no" if we are looking for a path between the vertices s and u.

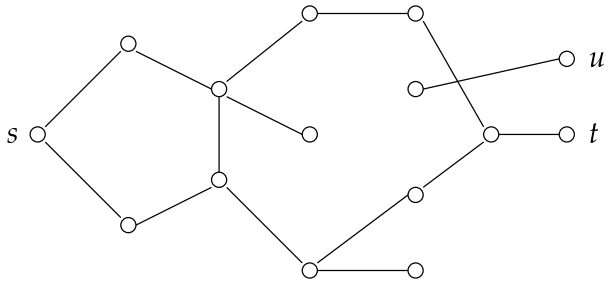

Figure 2.1: Instance of a Graph Connectivity Problem

Definition 2.1.4 A *directed graph* (or *digraph*) $G = (V, A)$ consists of a collection V of vertices, and a set of collection A of *ordered* pairs of vertices called edges or arcs.

A directed graph is represented similarly to an undirected one. The only difference in this case is that if there is an edge (x, y) between two vertices x and y in the graph, a directed arrow is drawn from the vertex x to the vertex y. An example of such a representation is shown in Figure 2.2.

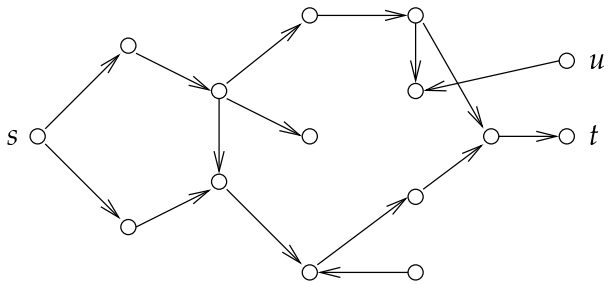

Figure 2.2: Instance of a Directed Graph Connectivity Problem

For the instance of the DIRECTED GRAPH CONNECTIVITY PROBLEM in Figure 2.2, once again the answer is "yes" if the specified vertices are s and t but "no" when the specified vertices are s and u, because there is no directed path between them.

A digraph is *acyclic* if it contains no directed cycle, *i.e.* a sequence of arcs forms a path from the vertex to itself. A *directed acyclic graph* is also referred to as a *DAG*.

2.1.1 Representation

Since we seek to solve problems on a computer, each instance of the problem must be represented in some way inside the computer. We will assume that every instance is coded by strings over some alphabet of letters.

An instance of a problem is described as a finite sequence of symbols over some finite *alphabet* Σ. Similarly, the solution to the instance of the problem will be represented in a similar format.

A finite sequence of symbols over an alphabet Σ is called a *string* over Σ. The set of strings of length exactly k is denoted by Σ^k. The set of all strings is denoted by $\Sigma^* = \bigcup_{k \geq 0} \Sigma^k$. The *length* of a string x is denoted by $|x|$. The symbol λ denotes the unique string of length 0. When the alphabet is not mentioned explicitly, it is conventional to assume that $\Sigma = \{0, 1\}$.

The *input size* of an instance of a problem is the size of the string encoding the problem. The input size depends on the choice of encoding chosen. However, the measures of complexity that we discuss are very robust under changes of encoding. We will not dwell excessively on the choice of encoding because our results hold for any reasonable encoding scheme.

Since we deal extensively with graphs in this thesis, it is worthwhile mentioning that there are two standard representations for graphs — the *adjacency matrix* representation which is a 0-1 matrix in which the $(i, j)^{\text{th}}$ entry is a 1 if there is an edge between vertex i and j; and the *adjacency list* representation in which a list of neighbors is maintained for each vertex.

2.1.2 Decision Problems

Decision problems, problems which have have a *"yes/no"* answer, are an important class of computational problems. Solving these problems is equivalent to deciding membership in the corresponding subset of Σ^* that consists of the encodings of the positive instances, *i.e.*, the strings for which the answer is *"yes"*.

A subset of Σ^* is called a *language*. The complement of a language L is the language $\Sigma^* - L$ and is denoted by $\bar{L}$. We are most often interested in a class

of languages C. The class of complements of the languages in C is denoted by co-C.

For example, the language corresponding to the GRAPH CONNECTIVITY PROBLEM would be the following subset of Σ^*:

$$L = \{ (G, s, t) : \exists \text{ a path between } s \text{ and } t \text{ in } G \}.$$

Another interesting language is that of bipartite graphs that have a perfect matching. An undirected graph is called *bipartite* if its vertex set can be divided into two disjoint sets A and B, such that all edges join some vertex of A to a vertex of B. An example is shown in Figure 2.3.

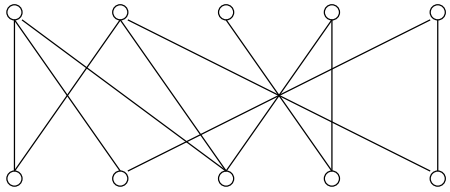

Figure 2.3: A Bipartite Graph

A *matching* is a maximal set of vertex-disjoint edges. A *perfect matching* is one in which every vertex is present in some edge of the matching. The edges that form a perfect matching in the above graph are displayed in Figure 2.4.

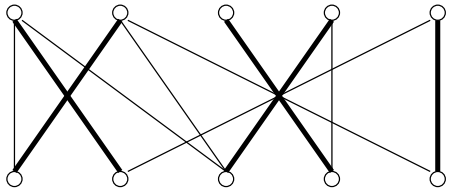

Figure 2.4: Perfect Matching in a Bipartite Graph

The notions of a matching and a perfect matching extend to general undirected graphs as well.

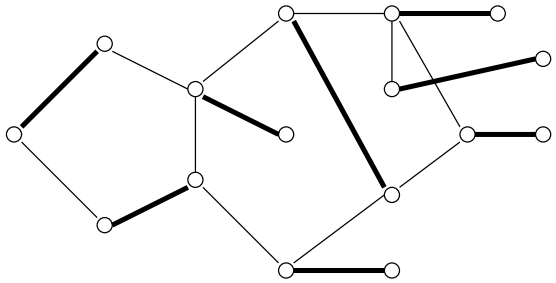

Figure 2.5: Perfect Matching in a General Graph

2.1.3 Weighted Optimization Problems

Another important class of problems is that of *weighted optimization problems*. Here the input consists of two parts, the non-numeric part and the numeric part. For every instance of the problem I, we have to find the optimum solution depending on some specified constraint.

An *objective function* is some function that depends only on the numeric part of the input.

For example, we can consider the SHORTEST PATH PROBLEM in which the edges are labeled by weights (which can be thought of as lengths or distances). The objective in the SHORTEST PATH PROBLEM is to find the length of the shortest path between two designated vertices s and t (Figure 2.6).

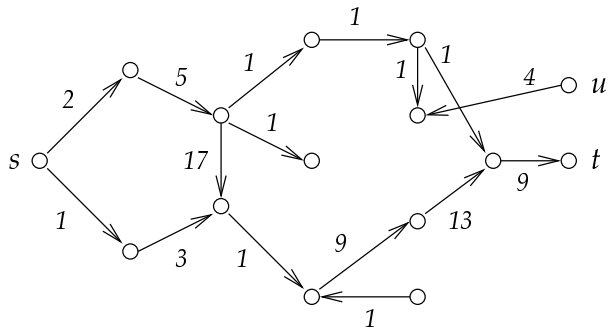

Figure 2.6: Instance of a Shortest Path Problem

The most frequently encountered optimization problems are *maximization* (or *minimization*) problems, in which the algorithm is expected to compute the

maximum (or minimum) of some objective function. The SHORTEST PATH PROBLEM is an example of a minimization problem.

For example, the length of the shortest path between s and t in Figure 2.6 is 19. If instead of just the value of the optimum, we ask for the path itself, the answer is the sequence of vertices that form the shortest path. The optimum path is shown in Figure 2.7.

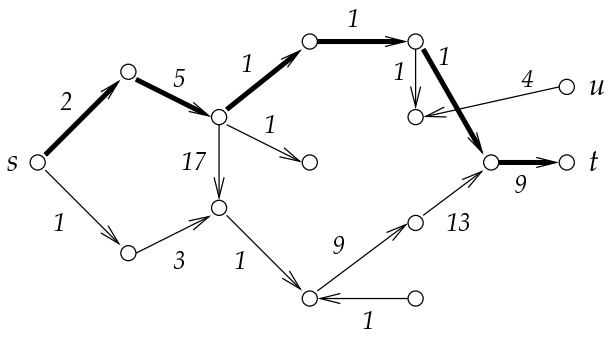

Figure 2.7: Solution to the Instance of a Shortest Path Problem

Another example of a weighted optimization problem that has practical importance is the minimum WEIGHTED BIPARTITE GRAPH MATCHING PROBLEM. The problem consists of computing the minimum weight among all perfect matchings in a given bipartite graph. If we don't restrict ourselves to bipartite graphs, we get the more general minimum WEIGHTED GRAPH MATCHING PROBLEM.

A weighted optimization problem is said to be *homogeneous* if scaling all the weights by $k \in \mathbb{R}$ scales the optimum value of the objective function by k as well.

Any weighted maximization (resp. minimization) problem can easily be transformed into a decision problem by adding an extra input parameter. The problem then consists of deciding whether the optimum value is larger (resp. smaller) than this additional parameter.

2.1.4 Resources

In order to compare the efficiency of algorithms, we need to talk about how much of some *resource* the algorithm consumes.

The resources used by a solution to a computational problem depend on the model of computation that we adopt. Typical resources that we are interested in include the running time of a program, or the space (memory) utilized by the program. In the case of parallel computation, we will also be interested in the number of processors that our algorithm uses.

Our emphasis in measuring the amount of resources used is on *asymptotic worst-case complexity*. This measures how the worst-case resources used by a problem grow as a function of the input size.

The *size of an instance* x is the length $|x|$ of the string that encodes it. The amount of resources needed for a given input size n is the maximum amount of resources needed over all instances x of size n (worst-case behavior of the program on input n). A *resource bound* is a monotone non-decreasing function $f : \mathbb{N} \to \mathbb{R}^+$.

The resource bounds that we consider are very robust with respect to the details of the particular encoding. Hence, we will often we able to substitute more natural problem-specific measures for the input size. For example, for problems on graphs, we will use the number of vertices as a measure of its input size. When we do that, and we also need to refer to the total bit-length, we use the symbol N for the latter.

The following notation allows us to capture the asymptotic growth of resource bounds while suppressing details like constant factors. Let $f, g : \mathbb{N} \to \mathbb{R}^+$ be functions.

Definition 2.1.5 $f \in O(g)$ if there exists a constant $c > 0$ such that $f(n) \leq c \cdot g(n)$ for all sufficiently large $n \in \mathbb{N}$. This means that f grows no faster than g.

Definition 2.1.6 $f \in \Omega(g)$ if there exists a constant $c > 0$ such that $f(n) \geq c \cdot g(n)$ for all sufficiently large $n \in \mathbb{N}$. Hence, f grows at least as fast as g.

Definition 2.1.7 $f \in \Theta(g)$ if $f \in O(g)$ and $f \in \Omega(g)$, *i.e.*, f and g have the same growth rate.

Definition 2.1.8 $f \in o(g)$ if for any constant $c > 0$, we have $f(n) \leq c \cdot g(n)$ for all sufficiently large $n \in \mathbb{N}$ (depending on c). This is equivalent to:

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0.$$

Hence, f grows strictly slower than g.

Definition 2.1.9 $f \in \omega(g)$ if for any constant $c > 0$, we have $f(n) \geq c \cdot g(n)$ for all sufficiently large $n \in \mathbb{N}$ (depending on c). This is the same as:

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = \infty.$$

This means that f grows strictly faster than g.

The most frequently used bounds and their names are listed in Table 2.1.

RESOURCE BOUND	NAME
$O(1)$	constant
$O(\log n)$	logarithmic
$(\log n)^{O(1)}$	polylogarithmic
$n^{O(1)}$	polynomial
$2^{n^{O(1)}}$	exponential

Table 2.1: Commonly Used Resource Bounds

As is conventional, we write $f = O(g)$, etc., instead of $f \in O(g)$. This abuse of notation is fine as long as we remember that the right-hand term contains less information than the left-hand side.

2.2 Models of Computation

No mathematical model of computation can reflect reality with perfect accuracy. Mathematical models are abstractions; as such, they have limitations. However, these simplified abstractions allow us to analyze computational problems with greater ease.

In the following sections, we introduce a number of models for sequential and parallel computation. All of the models are robust, in the sense that each of them can simulate the other with at most a polynomial slow down.

2.2.1 Turing Machines

The standard model of sequential computation is the *Turing machine*. It consists of a *finite state control*, a read-only *input tape*, several read-write *work tapes*, and a write-only *output tape*. Each tape has a read/write head which can sense the character written on a tape cell and change it. At any given instant the finite state control is in one of a finite number of states S, and at each step in time, it can make a state transition based on the symbols under the heads of the various tapes, change these symbols, and move any of the heads left or right by one position.

At the beginning of the computation, the finite control of the machine is in some designated *start* state $s_0 \in S$. During the computation, the finite state control keeps making transition depending on the symbols on the various tapes until it reaches the final state $s_f \in S$. The contents of the output tape define the result of the computation.

An *instantaneous description* of the Turing machine contains all the information about its current configuration (except for the input). In particular, it contains the state of the finite control, the contents of the work and output tapes, and the positions of the heads on all the various tapes. The sequence of instantaneous descriptions starting from the initial configuration forms a transcript of the computation on a given input.

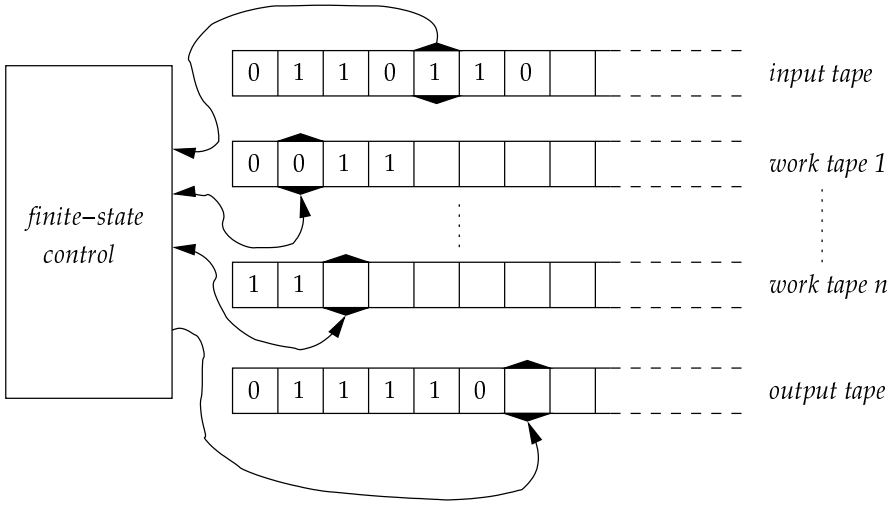

Figure 2.8: Turing Machine

2.2.1.1 Resources

The two important resources of a Turing machine are its time and space requirements. The *time* taken to compute some answer is the number of state transitions the machine makes before it reaches the final state s_f. The *space* needed is the number of different cells of the work tapes that are accessed in the course of the computation. The cells of the input tape do not contribute to the space requirements.

2.2.1.2 Complexity Classes

In order to bound the time or space usage, we use the notion of a *constructible function*. A function $t : \mathbb{N} \to \mathbb{R}^+$ is *time-constructible* if there exists a Turing machine that runs in time exactly $t(n)$ on every input of size n. Analogously, we call $s : \mathbb{N} \to \mathbb{R}^+$ *space-constructible* if there exists a Turing machine that runs in exactly space $s(n)$ on every input of size n. All the resource bounds used in this thesis are space-constructible, and all super-linear bounds are time-constructible.

$\mathcal{DTIME}[t(n)]$ is defined to be the class of languages that can be decided in time $O(t(n))$ on a Turing machine. The class $\mathcal{P}$ denotes the class of languages solvable in polynomial time. It is considered to be the class of languages efficiently solvable on a sequential computer [15].

$$\mathcal{P} = \bigcup_{c>0} \mathcal{DTIME}[n^c]$$

$\mathcal{DSPACE}[s(n)]$ is defined to be the class of languages that can be decided in space $O(s(n))$ on a Turing machine. The class $\mathcal{L}$ denotes the class of languages solvable in logarithmic space.

$$\mathcal{L} = \mathcal{DSPACE}[\log n]$$

Whenever we have a machine that runs in time $t(n)$, it cannot use more than $t(n)$ memory locations, and hence, we have the following corollary:

$$\mathcal{DTIME}[t(n)] \subseteq \mathcal{DSPACE}[t(n)].$$

Similarly, whenever we have a machine that uses space $s(n)$, the total number of instantaneous descriptions is $2^{c \cdot s(n)}$, where c depends on the size of the alphabet Σ. There is a natural directed graph associated with this set of instantaneous descriptions, in which an edge is drawn between two descriptions if the Turing machine makes a transition between them in one step. The task of deciding whether the input is in the language or not, is equivalent to finding whether there exists a directed path in this graph between the starting instantaneous description and the final one. This can be achieved in time proportional to the size of the graph [10, 30]. Hence, we obtain the following inclusion:

$$\mathcal{DSPACE}[s(n)] \subseteq \bigcup_{c>0} \mathcal{DTIME}[2^{c \cdot s(n)}].$$

In particular, we have the following inclusion which is conjectured to be strict.

$$\mathcal{L} \subseteq \mathcal{P}.$$

2.2.2 The Random Access Machine

The Random Access Machine (RAM for short) is a computational model that is much closer to the conventional notion of a computer. It consists of a processor which has access to *local memory*. The memory locations are numbered sequentially. Each memory location can store one integer which can be interpreted either as a number, or as the address of another memory location.

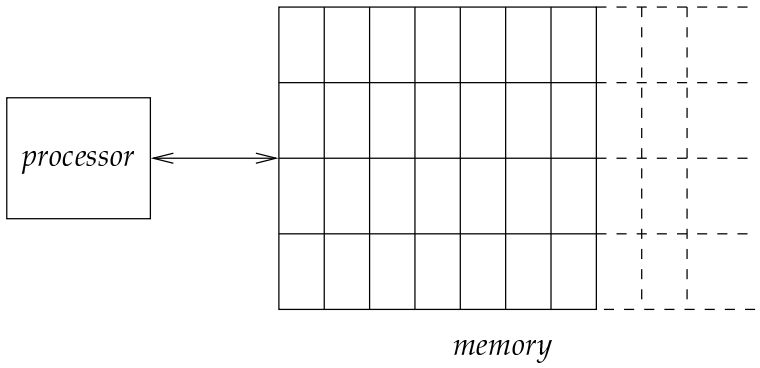

Figure 2.9: Random Access Machine (RAM)

The computation is represented by a sequence of instructions which can be grouped into different categories as follows:

Arithmetic These instructions are used to perform addition, subtraction, multiplication or integer division on the integers stored in two memory locations and store the result in a third.

Boolean These instructions are used to perform logical operations like AND, OR and NOT on two operands.

Branch Operations These instructions alter the *"flow"* of the program either unconditionally, or depending on the contents of some register.

Indirect Referencing These instructions allow the RAM to use a memory location as a *pointer* to another location.

Bit Operations These operations allow the RAM to modify individual bits of the integers stored in memory.

2.3 Parallel Models of Computation

Families of boolean circuits are the standard model of parallel computation just as the Turing machine is the standard model of sequential computation. The idea behind parallel computation is that several processors can work simultaneously on different parts of a problem, and collectively try to solve the problem faster than a single processor would.

2.3.1 Uniform Families of Boolean Circuits

A *boolean circuit* is a combination of AND-, OR-, and NOT-gates built on top of the inputs in the form of a DAG. Each input gate is labeled with a variable x_i, for $i \in N$. For each truth assignment $\sigma : x_i \to \{\text{TRUE}, \text{FALSE}\}$, we can inductively define a truth-value for each of the gates of the circuit. In order to compute a function from $\{0,1\}^n$ to $\{0,1\}$, we can consider FALSE to be represented as 0, and TRUE as 1.

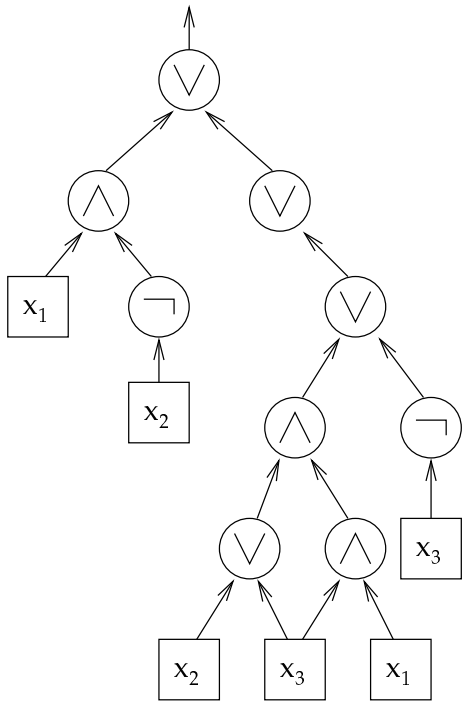

Figure 2.10: Boolean Circuit

Since any given boolean circuit only works for a fixed input length, if we wish to compute the solution for any instance of an infinite problem, we need to consider an infinite family of circuits. In order to decide a language L, we need a family of circuits $\{C_n\}$, where the circuit C_n computes the language L restricted to $\{0,1\}^n$. Since we are allowed to use a different *"program"* for each input length, the infinitely many programs may be very different from each other. Such a model of computation is referred to as a *non-uniform* model.

For a family of circuits to be considered an *"algorithm"* for this problem, we need to somehow ensure that we can generate the n^{th} program *"easily"*. The technical requirement is that there is some program-generating Turing machine that on input 1^n outputs a description of the n^{th} circuit C_n. Such a family of circuits is called *uniform* if the Turing machine needed to generate the circuit runs in logarithmic space.

2.3.1.1 Resources

The two most important resource parameters of a boolean circuit are its *depth* and its *size*. The depth of a circuit is the longest path from the input nodes to the output gate. It corresponds to the parallel running time it takes to compute the answer. The size of the circuit is the number of edges in the circuit. This is at most a factor of 2 of the total number of gates, and hence, is a measure of the number of processors used in the computation.

2.3.1.2 Complexity Classes

For any $k \in \mathbb{N}$, $\mathcal{NC}^k$ denotes the class of languages that can be decided by a uniform family of circuits $\{C_n\}$ of polynomial size and $O(\log^k n)$ depth. The union of the classes $\mathcal{NC}^k$ forms "Nick's Class" (after Nicholas Pippenger). The class $\mathcal{NC}$ is considered to be the class of languages that are efficiently computable on a parallel computer.

There is a strong connection between parallel computation and computation on a sequential machine that utilizes space efficiently.

Theorem 2.3.1 (Savitch [42])

$$\mathcal{NC}^1 \subseteq \mathcal{L} \subseteq \mathcal{NC}^2$$

For both inclusions, strictness is conjectured.

So far we have concerned ourselves with circuit gates with fixed *fan-in* (number of inputs). We can also allow the OR- and AND-gates to have unbounded fan-in, *i.e.*, they can compute the logical AND, or the logical OR of any number of inputs in one step. Of course, when we deal with circuits of polynomial size, this also means that the number of inputs to any gate is also bounded by a polynomial.

For any $k \in \mathbb{N}$, $\mathcal{AC}^k$ denotes the class of languages that can be decided by a uniform family of circuits $\{C_n\}$, with unbounded fan-in gates, of polynomial size and $O(\log^k n)$ depth. Unlike the bounded fan-in case, the class $\mathcal{AC}^0$ is meaningful.

Any $\mathcal{NC}^i$ circuit is already an $\mathcal{AC}^i$ circuit. Conversely, we can compute the logical AND, or the logical OR of a polynomial number of inputs by a circuit of depth $O(\log n)$. Hence, converting an $\mathcal{AC}$ circuit into an $\mathcal{NC}$ circuit increases the depth by a factor of $O(\log n)$. Thus, the two hierarchies alternate under inclusion as follows:

$$\mathcal{AC}^0 \subseteq \mathcal{NC}^1 \subseteq \mathcal{AC}^1 \subseteq \mathcal{NC}^2 \subseteq \cdots \subseteq \mathcal{AC}^i \subseteq \mathcal{NC}^{i+1} \subseteq \mathcal{AC}^{i+i} \subseteq \cdots$$

In particular, $\mathcal{AC} = \bigcup_k \mathcal{AC}^k = \bigcup_k \mathcal{NC}^k = \mathcal{NC}$.

We can simulate a parallel circuit in polynomial time by doing a *"breadth-first search"* from the inputs, computing the output of each gate and storing it. This shows that

$$\mathcal{NC} \subseteq \mathcal{P}.$$

It is widely conjectured that the inclusion is strict. The conjecture $\mathcal{P} \neq \mathcal{NC}$ is one of the major open problems in Theoretical Computer Science.

2.3.2 The Parallel Random Access Machine

The Parallel Random Access Machine (PRAM for short) is a model of a distributed computing environment, in which many of the underlying details of the hardware have been abstracted away. It consists of a set of processors, all of which have access to *shared memory*. Each processor is a Random Access Machine and has local memory as discussed in Section 2.2.2. Each processor can access either its own local memory or the common shared memory at unit cost. The processors can execute the same set of instructions as a RAM.

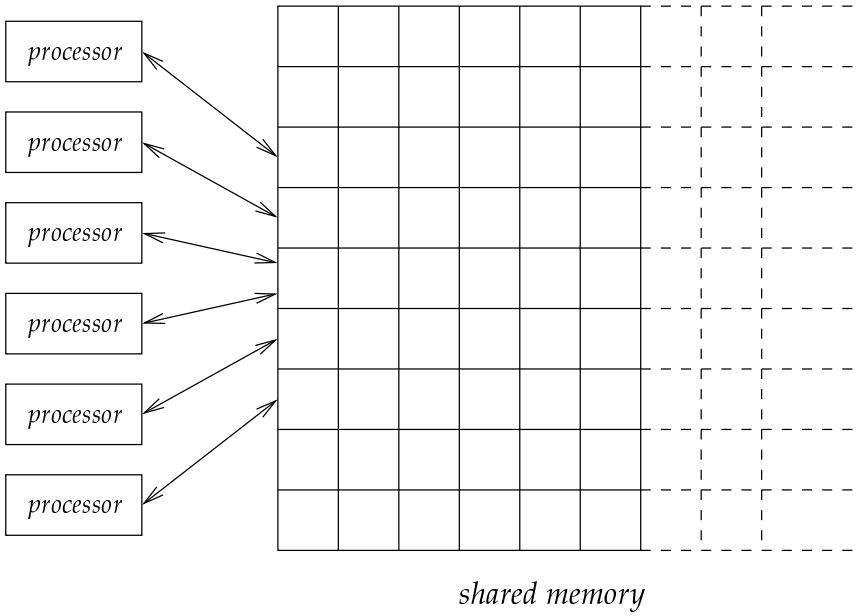

Figure 2.11: Parallel Random Access Machine (PRAM)

Each processor can access the shared memory at unit cost. Depending on how different processors access the same memory location simultaneously, we can define different models of the PRAM.

The most restrictive model is the exclusive-read, exclusive-write PRAM (EREW PRAM), in which exactly one processor is allowed to read or write any given memory location at a given time.

The most general model is the CRCW PRAM (concurrent-read, concurrent-write). Any number of processors are allowed to read a memory location at a

given time. Similarly, any number of processors are allowed to write a given memory location. Since different processors can write different values to the given memory location, there are several ways in which we can resolve this conflict. We can either insist that all the processors write the same value (or if not, the PRAM halts with an error), or we can allocate unique priorities to the various processors, and the write value of the processor with highest priority succeeds.

In between these two extremes is the CREW PRAM (concurrent-read, exclusive-write). Many processors can read the same location, but only one of them can write to any particular location at a given time.

The difference between the various models do not turn out to be significant for theoretical purposes, and hence, we can pick any of the above models. For the purposes of lower bounds, we pick the strongest of the above models because that enables us to prove the strongest possible lower bound for the problem.

2.4 Randomness

One of the most important questions in Theoretical Computer Science has been the role of randomization in computation. Does the introduction of random coin-flips introduce any efficiency into the algorithmic process? Researchers are currently divided on the subject. There are a number of results which suggest both possibilities — places where randomization seems to be indispensable, as well as other results which suggest that it might be possible to do away with randomization completely (with at most a polynomial loss of efficiency).

Randomized algorithms are similar to the usual algorithms except they possess the ability to flip coins. They are also allowed to have a small probability of error, and depending on the type of error they are allowed to make, we can distinguish between several types of algorithms:

Two-sided error These are the most general type of algorithms which are allowed to make small errors (with probability at most $1/3$), both on inputs in the language, as well as those not in it.

One-sided error These are algorithms that can err only on one side. If the input does not belong to the language, the algorithm always says so. However, if the input does belong to the language, the algorithm is allowed to give the wrong answer with a small probability (at most $1/2$).

Zero-sided error These algorithms never give the wrong answer. However, with a small probability (at most $1/2$), they are allowed to say *"I don't know"*. Whenever, the algorithm gives an answer, it is always correct.

2.4.1 Randomized Turing Machines

We can allow a Turing machine the ability to flip coins by giving it access to a sequence of *"random bits"*. Thus, a *randomized Turing machine* is a Turing machine that has an additional input tape (called the *random tape*) that contains the outcomes of a sequence of independent, unbiased coin tosses. The tape is read-only and the tape head can only move to the right. Reading a bit from this tape is thus equivalent to flipping a fair coin.

2.4.1.1 Resources

The definitions of the time and space usage of a randomized Turing machine are identical to that of a conventional Turing machine.

There is one more additional resource of randomized Turing machines that is frequently considered, namely the number of random bits used in the process of the computation. However, we will not be concerned with this in this thesis.

2.4.1.2 Complexity Classes

$\mathcal{RP}$ denotes the class of languages that can be decided with one-sided error by a randomized Turing machine that runs in polynomial time. Similarly, $\mathcal{RL}$ denotes the class of languages that can be decided with one-sided error by a randomized Turing machine running in logarithmic space.

The problem of deciding the GRAPH CONNECTIVITY PROBLEM can be solved in $\mathcal{RL}$ by taking a random walk on the graph starting from the vertex s. At each step, we pick one of the neighbors of the current vertex uniformly at random and move to it. If s and t are not connected, we will never reach t. If they are connected, then with high probability, we will surely hit t in $O(n^3)$ steps [1]. All we have to do is keep track of the current vertex, and the number of steps taken so far, both of which can be accomplished in logarithmic space.

$\mathcal{BPP}$ (resp. $\mathcal{ZPP}$) denotes the class of languages that can be decided with two-sided (zero-sided) error by a randomized Turing machine that runs in polynomial time. Also we have the following relationship:

$$\mathcal{ZPP} = \mathcal{RP} \cap \text{co-}\mathcal{RP}$$

2.4.2 Randomized Circuits

A circuit can be given the ability to flip coins by giving it additional input gates that are fed the outcomes of independent fair coin tosses. Thus, a *randomized circuit* has two types of input gates: those corresponding to the actual input bits, and those corresponding to the random bits.

2.4.2.1 Resources

We can define the size and depth of these randomized circuits in the same fashion as that of ordinary circuits. Since we are interested in circuits of polynomial size, we may assume that the number of input random bits is polynomial in the length of the input.

2.4.2.2 Complexity Classes

For any $k \in \mathbb{N}$, we define $\mathcal{RNC}^k$ as the class of languages for which there exists a uniform family of randomized Boolean circuits $\{C_n\}$ of polynomial size and depth $O(\log^k n)$ that decide the language with one-sided error.

For example, the problem of deciding whether a graph has a perfect matching or not can be decided in $\mathcal{RNC}^2$ [36].

2.5 Non-determinism

A *non-deterministic Turing machine* does not represent a physically realizable model of computation. In this model, the machine has the ability to make *"guesses"* about possible choices in the computation. Even though the model is physically unrealizable, it is so useful in capturing the computational complexity of many important problems that it is conventionally included in the standard models of computation.

It is widely believed that non-determinism plays a very powerful role in computation. There seems to be a wide array of problems that can be solved fast using non-determinism but seem to require exponential time to compute deterministically.

We will not be considering non-deterministic computation in any detail in this thesis but we state the definitions and some basic results here for the sake of completeness.

2.5.1 Non-deterministic Turing Machines

The model for a non-deterministic Turing machine is the same as that of a deterministic one except that there may be several possible transitions from the current configuration of the Turing machine. Each time that this happens, the machine makes a *"choice"* on what configuration to go to next.

A non-deterministic Turing machine decides a language L if on every input, there is some sequence of choices that allow the machine to get from the initial state to an accepting final state.

It is not too hard to see that we can move all the non-determinism to the beginning of the computation, *i.e.*, we can guess the sequence of choices first, and then the computation that follows has to verify that the choices were *"correct"*.

2.5.1.1 Resources

The definitions of the time and space usage of a non-deterministic Turing machine are identical to those of a conventional Turing machine.

2.5.1.2 Complexity Classes

$\mathcal{NTIME}[t(n)]$ is defined to be the class of languages that can be decided in time $O(t(n))$ on a non-deterministic Turing machine. The class $\mathcal{NP}$ denotes the set of languages that are computable on a non-deterministic Turing machine in polynomial time.

$$\mathcal{NP} = \bigcup_{c>0} \mathcal{NTIME}[n^c]$$

Since every deterministic Turing machine is also a non-deterministic one, the following inclusion is obvious.

$$\mathcal{P} \subseteq \mathcal{NP}$$

The famous $\mathcal{P} \neq \mathcal{NP}$ conjecture asserts that this inclusion is strict.

Similarly, we may define $\mathcal{NSPACE}[s(n)]$ to be the class of languages that can be decided in space $O(s(n))$ on a non-deterministic Turing machine. The class $\mathcal{NL}$ denotes the class of languages solvable on a non-deterministic Turing machine in logarithmic space.

$$\mathcal{L} \subseteq \mathcal{NL}$$

Once again, strictness is conjectured.

2.6 Reductions and Completeness

A *reduction* of a computational problem A to a computational problem B is a way of classifying the relative hardness of the two problems. If we somehow have a *"magic box"* that allows us to solve the problem B, then the reduction is a computational way of solving the problem A using this box as an *oracle*.

The precise difference in computational difficulty depends on the different kinds of reductions that we allow between the two problems. Various types of reductions can be distinguished based on the model of computation, the resource bounds, and the kind of oracle access allowed. Notationally, we write $A \leq B$, which indicates that A is computationally no harder than B.

2.6.1 Oracle Turing Machines

An *oracle Turing machine* makes precise the notion of querying a black-box for the computational problem B. An oracle Turing machine is a Turing machine with an additional read/write tape (called the *oracle tape*), and two distinguished states: the query state s_q, and the answer state s_a. The computation of the machine is the same as that of the Turing machine, except for when it is in the query state s_q. At that point, the content of the oracle tape is interpreted as an instance of B, and is replaced in one step by the solution to that instance with the finite state of the machine moving into state s_q. In effect, we obtain a solution to an instance of B in a single computational step.

The time and space requirements of an oracle Turing machine with a given oracle B are defined in the same way as that of an ordinary Turing machine. The size of the oracle tape is not taken into account when considering space bounds for technical reasons.

2.6.2 Oracle Circuits

The notion of an *oracle circuit* is analogous to that of an oracle Turing machine. We introduce a new type of gate (called the *oracle gate*), whose inputs will be instances of the problem B, and whose outputs are the solution to the problem.

The *size* is the number of edges in the circuit. For an oracle gate, it equals the size of the input and output wires. The *depth* is the size of the largest weighted path from the output to the input. Input gates have weight zero, and all other gates have a weight equal to the logarithm of the total number of edges.

2.6.3 Reductions

Reductions can be used in several different ways. If we have a reduction between two problems $A \leq B$, then an efficient algorithm for B coupled with the reduction turns into an algorithm for A. Depending on the efficiency of the reduction, this may or may not be an efficient algorithm for A.

Similarly, if we have a lower bound (hardness result) for A, then we get a corresponding hardness result for B. The efficiency of the reduction dictates the strength of the lower bound for B. Thus, in general we try to give the strongest reduction between two problems. The strongest reduction corresponds to using the weakest complexity class for the reduction, and the weakest type of oracle access.

Very frequently, reductions only require a very restricted kind of oracle access. Not only does this make the analysis of the reduction easy, but it also helps us in obtaining fast algorithms or strong lower bounds for the problems. We consider two types of oracle access: many-one (m), and Turing (T). The type of oracle access is indicated as a subscript to the $\leq$ symbol. The symbols $\leq_m$ and $\leq_T$ denote many-one and Turing reductions respectively.

A *many-one reduction* is a reduction that makes exactly one query to the oracle and outputs the answer to that oracle query. In the case of two languages, A and B, a many-one reduction can be thought of as a mapping $f : \Sigma^* \to \Sigma^*$ such that for all $x \in \Sigma^*$, $x \in A$ iff $f(x) \in B$.

A *Turing reduction* is the most general kind of reduction in which no restriction is placed on the oracle access. Frequently, the notation $\leq_{f(n)-T}$ is used to denote the fact that on inputs of size n, at most $f(n)$ queries are made to the oracle.

The resource bounds that it takes to compute the oracle queries can also be used to classify the reductions in a manner orthogonal to the above classifications. We consider reductions that are computable in polynomial time, logarithmic space, and in parallel logarithmic time. The model of computation and the resource bounds are indicated using a superscript to the $\leq$ symbol. For example, $\leq^{\mathcal{P}}$, $\leq^{\mathcal{L}}$, and $\leq^{\mathcal{NC}^1}$ denote reductions running in polynomial time, logarithmic space, and parallel logarithmic time respectively. It should be noted that because of the inclusion relations between complexity classes, an $\leq^{\mathcal{NC}^1}$-reduction is also an $\leq^{\mathcal{L}}$-reduction, which in turn is a $\leq^{\mathcal{P}}$-reduction.

For example, the SHORTEST PATH PROBLEM problem discussed earlier can be $\leq_m^{\mathcal{NC}^1}$-reduced to the WEIGHTED BIPARTITE GRAPH MATCHING PROBLEM [31, 33].

2.6.4 Completeness

A computational problem A is called *hard* for a complexity class C under reduction $\leq$ if every problem T in C $\leq$-reduces to A. If the problem A also belongs to the class C, then we call A *complete* for C under $\leq$.

Frequently, in the absence of context, the type of reduction is often implicitly understood. The term $\mathcal{P}$*-complete* means complete for $\mathcal{P}$ under $\leq_m^{\mathcal{L}}$-reductions, and the term $\mathcal{L}$*-complete* means complete for $\mathcal{L}$ under $\leq_m^{\mathcal{NC}^1}$-reductions.

The CIRCUIT VALUE PROBLEM is the language consisting of pairs (C, x) where C is the description of a Boolean circuit, say on n inputs, and x is a bit string of length n such that C accepts the input x. The CIRCUIT VALUE PROBLEM is complete for $\mathcal{P}$ under $\leq_m^{\mathcal{NC}^1}$-reductions, and hence is $\mathcal{P}$-complete.

An example of an $\mathcal{L}$-complete problem is the DIRECTED GRAPH CONNECTIVITY PROBLEM.

CHAPTER 3

TECHNIQUES FOR LOWER BOUNDS

3.1 Introduction

In this chapter, we briefly discuss the history of the techniques that have been used to prove lower bounds for combinatorial problems. Additionally, we outline Mulmuley's technique that is used in this thesis to obtain lower bounds for combinatorial problems.

3.1.1 Lower Bounds for Circuit Models

There are virtually no lower bounds known for general circuits trying to compute boolean functions. It can be shown using counting arguments that there are boolean functions on n inputs that require circuits of size $\Omega(2^n/n)$. However, no super-linear lower bounds are known for the computation of an explicit family of functions in $\mathcal{NP}$ by a boolean circuit.

All the super-linear lower bounds in the area are characterized by the following idea: instead of attempting to prove a lower bound for the general circuit model, researchers attempted to *"handicap"* the circuit in different ways, and then proved lower bounds in these weaker models of computation.

The first success story in the area was in the case of *monotone circuits* where the model of computation allows OR- and AND-gates, but negations using NOT-gates are forbidden. Razborov [40] managed to give a super-polynomial lower bound for the computation of an explicit function in $\mathcal{NP}$ by monotone boolean circuits. This was later improved by Alon and Boppana [2] to an exponential lower bound on a related function. The hope was that all monotone functions in $\mathcal{P}$ would have polynomial-sized (or at least, sub-exponential sized) circuits

which would prove that $\mathcal{P} \neq \mathcal{NP}$. Unfortunately, Tardos [45] managed to give a similar lower bound for a function that was computable in $\mathcal{P}$, which dashed any hopes of trying to separate $\mathcal{P}$ from $\mathcal{NP}$ using monotone circuit techniques.

The other line of attack was on lower bounds for constant-depth circuits. The initial results of Yao [50], and Furst, Saxe and Sipser [20] were improved by Håstad [26] to give optimal, exponential lower bounds for computing the parity function using constant-depth circuits.

In contrast to circuits of constant depth, there are no known lower bounds for circuits that have unbounded depth. This should be contrasted with the results of this thesis in which we give lower bounds for the computation of certain combinatorial problems on a PRAM whose running time is unbounded.

3.1.2 Lower Bounds for Algebraic Models

The computation of a problem on a RAM can be described by a sequence of operations of two types: arithmetic and comparisons. For the time being, we ignore pointer and bit operations. The computation of the RAM can be viewed as a path in a rooted tree, in which the branch operations act as nodes which determine the different directions in which the program can go. Accordingly, we may define a model of computation called the *algebraic decision tree* model, in which the inputs $x_1, x_2, \ldots, x_n$ are integers, and the program is a set of instructions $L_1, L_2, \ldots, L_p$ where each instruction is either an arithmetic operation, or a comparison operation. The leaves of this decision tree are all labeled with YES or NO depending on whether the input is to be accepted or rejected.

Ben-Or [5] and Yao [49] used classical techniques in real algebraic geometry to give a lower bound on the depth of an algebraic decision tree computing some function. The technique has recently been extended by Ben-Amram and Galil [3, 4].

Let the inputs to the problem $x_1, x_2, \ldots, x_n$ be viewed as a single point $x = (x_1, x_2, \ldots, x_n)$ in the n-dimensional Euclidean space $\mathbb{R}^n$. The decision problem identifies the set of YES-instances $W \subseteq \mathbb{R}^n$, such that an input is accepted

iff $(x_1, x_2, \ldots, x_n) \in W$. Suppose we know the number of disjoint connected components of the set W (denoted by $\#W$). The following theorem of Ben-Or and Yao relates this quantity to the running time of any algebraic decision tree that solves the problem on a RAM.

Theorem 3.1.1 (Ben-Or, Yao) Let $W \subseteq \mathbb{R}^n$, and let T be an algebraic decision tree that solves the membership problem in W. If h denotes the height of the tree T, and $\#W$ the number of disjoint connected components of the set W, then

$$h = \Omega(\log \#W - n) \tag{3.1}$$

A very elegant presentation of the complete proof can be found in [38]. The principal technique of the proof is the use of the theorem of Milnor and Thom from classical algebraic geometry. A variant of this theorem is used later when discussing further techniques for obtaining lower bounds.

Theorem 3.1.2 (Milnor-Thom) Let V be an *algebraic variety* in the n dimensional Cartesian space $\mathbb{R}^n$ defined by the common set of zeros of p polynomial equations:

$$\begin{aligned} g_1(x_1, x_2, \ldots, x_n) &= 0 \\ g_2(x_1, x_2, \ldots, x_n) &= 0 \\ &\vdots \\ g_p(x_1, x_2, \ldots, x_n) &= 0 \end{aligned}$$

If the degree of each polynomial g_i is at most d, then the number of distinct connected components of V is bounded by:

$$\#V \leq d(2d-1)^{n-1}. \tag{3.2}$$

Note that the bound on the number of common zeros of the polynomials is independent of the number of polynomials p.

The above theorem of Ben-Or and Yao has been used to give lower bounds for a number of important problems such as the ELEMENT DISTINCTNESS and the SET DISJOINTNESS problems in Yao [49].

3.2 Lower Bounds for Parallel Computation

Mulmuley's lower bounds for parallel computation are a major advancement of the techniques of Yao and Ben-Or. They give a general tool for proving lower bounds for combinatorial optimization problems in a slightly restricted version of the general PRAM model. The lower bound results in this thesis depend on the theorem of Mulmuley [35].

Mulmuley's technique allows him to give a strong lower bound on the parallel computation time for the MAX FLOW Problem in a slightly restricted (but natural) model of computation. Since the MAX FLOW Problem is $\mathcal{P}$*-complete*, this gives strong evidence to the $\mathcal{P} \neq \mathcal{NC}$ conjecture.

In Section 3.2.1, we first discuss the model. The crucial notion of parametric complexity is introduced in Section 3.2.2. The theorem of Mulmuley is presented in Section 3.3, which is followed by an outline of the proof.

3.2.1 Model of Computation

The model for the lower bound is a variant of the Parallel Random Access Machine. In this restricted model, first defined by Mulmuley [35], bit operations on the registers are not allowed. Mulmuley's model eliminates those operations that allow bit-extraction or updates of the bits of the individual registers, but provides the usual arithmetic, indirect referencing, conditional and unconditional branch operations at unit cost (independent of the bit-lengths of the operands). In addition, it is an unbounded fan-in model, in which the operations $\{+, \min, \max\}$ have unbounded fan-in at unit cost (independent of the bit-lengths of the operands). However, multiplication is restricted to have bounded fan-in.

Unlike earlier models used for proving lower bounds, such as the constant-depth [26] or monotone circuit model [40], the PRAM model without bit operations is natural. Virtually all known parallel algorithms for weighted optimization and algebraic problems fit inside the model. Examples include fast parallel algorithms for solving linear systems [11], minimum weight spanning trees [32], shortest paths [32], global min-cuts in weighted, undirected graphs [28], blocking flows and max-flows [22, 43, 47], approximate computation of roots of polynomials [6, 37], sorting algorithms [32] and several problems in computational geometry [41]. In contrast to boolean circuits where no lower bounds are known for unbounded depth circuits, our result gives a lower bound for a natural problem in a natural model of computation.

3.2.2 Parametric Complexity

The technique for the lower bounds in this thesis involves the crucial notion of *parametric complexity* first used by Mulmuley [35].

Let us assume that we are given a fixed instance of a weighted optimization problem, in which the weights are replaced by linear functions in a parameter λ. If we plot the optimal value of the minimization (resp. maximization) problem as a function of λ over some interval I, the resulting *optimal cost graph* is piecewise linear and concave (resp. convex). The *parametric complexity* of the fixed instance of the weighted optimization problem over the interval I is defined as the number of *breakpoints*, *i.e.* points at which the function changes slope.

Definition 3.2.1 The *parametric complexity* $\rho(n, \beta)$ of a weighted optimization problem for input size n and size parameter β is the largest parametric complexity achieved on instances of the problem of size n with weights that are linear functions of the form $a + b\lambda$, where the bit-lengths of a and b are at most β.

3.3 The Theorem of Mulmuley

The theorem that connects the notion of parametric complexity to lower bounds for parallel computation on a PRAM without bit operations is due to Mulmuley [35]. The theorem can be loosely stated as combinatorial problems with high parametric complexity must have high parallel running time.

Recall that a weighted optimization problem is said to be homogeneous if scaling the weights by $k \in \mathbb{R}$, scales the optimum value by k as well. Even though we are dealing with optimization problems, we can easily convert them into decision problems as outlined in Section 2.1.3.

Theorem 3.3.1 (Mulmuley) Let $\rho(n, \beta(n))$ be the parametric complexity of any homogeneous optimization problem where n denotes the input cardinality and $\beta(n)$ the bit-size of the parameters. Then, the decision version of this problem cannot be solved on a PRAM without bit operations in $o\left(\sqrt{\log \rho(n, \beta(n))}\right)$ time using $2^{\Omega\left(\sqrt{\log \rho(n,\beta(n))}\right)}$ processors, even if we restrict every numeric parameter in the input to size $O(\beta(n))$.

Mulmuley's theorem is stronger in that it also provides a lower bound when the running time of an algorithm depends on its total bit-length.

This theorem is the main technique that is used to prove lower bounds in this thesis. For various combinatorial optimization problems, we construct explicit families of combinatorial objects that have high parametric complexity. This enables us to prove lower bounds on their parallel running time.

The other technique used is that of an efficient reduction between combinatorial problems. If we have two problems A and B such that $A \leq_m^{\mathcal{NC}^1} B$, and we show a lower bound on A, then we can obtain a lower bound on the parallel running time of B.

3.3.1 Outline of the Proof

In this section, we provide an outline of the proof of Theorem 3.3.1, along with some small extensions to the technique (which allow us to prove the lower bounds in Chapters 4 and 5).

The inputs to the problem can be divided into two distinct categories: the description of the problem encoded in some fashion which we refer to as the *non-numeric* part, and the weights for the optimization problem which are integers and hence, referred to as the *numeric* part of the input.

Let there be n distinct numerical inputs to the problem $x_1, x_2, \ldots, x_n \in \mathbb{Z}$. Assume that $p(n)$ processors work cooperatively to compute the solution to the problem, and that they achieve this in $t(n)$ time. The computation of each processor may be thought of as a computational binary tree of depth $t(n)$ in which each node of the binary tree is labeled with the instruction (and the operands) that it operates on. Let us also assume that the optimization problem is stated as a decision problem so that each leaf of the computation tree of each processor is labeled with either a YES or a NO which corresponds to the answer to the problem.

Fix a processor among the $p(n)$ available processors. It is clear that if two inputs follow the same branches of the tree on this processor, then the processor gives the same answer on these two inputs. If the two inputs follow the same branches on all the processors, then they are indistinguishable from the point of view of this entire computational system, and hence, we might as well regard them as being the same input.

This observation allows us to define a natural equivalence class on the inputs. Two inputs x and x' are *equivalent* if they follow the same branches of the computation trees on all the processors for the entire length of the computation.

We can relax the definition to define a less restricted equivalence relation on the inputs which classifies two inputs x and x' as *t-equivalent* if the two inputs follow the same branches of the computation trees on all processors until time t. The preceding notion of equivalence then becomes the same as the notion of $t(n)$-equivalence.

Lemma 3.3.2 Let C be a fixed t-equivalence class. Then for each memory location r, there exists a polynomial $f_{r,C,t}(z)$ of degree 2^t that gives the contents of location r at time t for every $z \in C$.

PROOF: Once we fix the equivalence class, the path of the processor until time t is fixed. At time $t = 0$, each memory location is either a constant or equal to some x_i. Hence it is a polynomial of degree 1. At each step, the degree of the polynomial can double if we multiply the values in two locations. Hence, by induction, we can obtain the polynomial $f_{r,C,t}$ of degree at most 2^t.

□

The above lemma from Mulmuley can be extended in the following way so that the model permits unbounded fan-in $\{+, \min, \max, \text{sort}\}$ operations. Note that, adding an unbounded number of integer locations does not change the degree of the polynomial. Likewise, taking the minimum or maximum of an arbitrary number of integers does not change the degree. Similarly we can provide a *sort* operation in unit step without changing the degree of the polynomial. The only thing that happens is that the memory locations are permuted.

The first part of the proof proceeds by giving a bound on the number of $t(n)$-equivalence classes. Let $\phi(t)$ be the number of t-equivalence classes, and ϕ be the number of equivalence classes (which is the same as $\phi(t(n))$).

In general, it is quite hard to give a tight upper bound on $\phi(t)$. Mulmuley's proof uses the technique of parameterization to give an upper bound on the number of equivalence classes when the inputs are restricted to a suitable affine subspace of the space of all inputs $\mathbb{Z}^n$.

Specifically, let us imagine that each input can be obtained as a linear form in d variables, $z_1, z_2, \ldots, z_d$ where $d \ll n$. In other words, there are linear functions l_i such that $x_i = l_i(z_1, z_2, \ldots, z_d)$. One of the key ideas of the proof lies in the fact that we can give a suitable upper bound on the number of equivalence classes when the inputs are thus restricted.

We can extend the notion of t-equivalence in a natural fashion to inputs restricted to an affine subspace. The two parametrized inputs z and z' are t-

equivalent if the corresponding problem inputs $x(z)$ and $x'(z')$ are t-equivalent. Let $\sigma(t)$ denote the number of equivalence classes when the inputs are restricted to this d-dimensional affine subspace. Then, we can state the following lemma about $\sigma(t)$.

Lemma 3.3.3 Let $p(n)$ be the number of processors used in the computation. Then, for all values of $t > 0$, the number of equivalence classes $\sigma(t)$ is bounded by:

$$\sigma(t) \leq \left(2 + 2 \cdot 2^t p(n)\right)^{dt}$$

The technique of Mulmuley actually also takes into account algorithms whose running time may depend on the total bit-length of the input to the problem. In order to talk about this, we need to introduce some additional notation. Let N be the total bit-length of the input. Let the solution to the problem be computed in $t(n,N)$ time using $p(n,N)$ processors. We can extend the notion of equivalence to mean $t(n,N)$-equivalence.

Since we have restricted the total size of the input problem to N, we only need to consider inputs x whose total bit-length is bounded by N. We call an input z from the affine subspace *permissible* if the input x obtained from it has total bit-length bounded by N. We restrict our attention only to permissible inputs z.

The above lemma can be extended to this more general case where it gives essentially the same bound.

Lemma 3.3.4 Let $p(n,N)$ be the number of processors used in the computation. Then, for all values of $t > 0$, the number of equivalence classes $\sigma(t)$ is bounded by:

$$\sigma(t) \leq \left(2 + 2 \cdot 2^t p(n,N)\right)^{dt}$$

Each node in the computation tree operates on a fixed number of operands. It can be seen that each operand can be determined by a fixed polynomial in the inputs x. Since the inputs x have now been restricted to permissible inputs depending on z, we can extend this to say that the operands at each node depend on a fixed polynomial in the inputs z.

An important observation about computation trees is that the only instructions that allow us to *branch* are the comparison operations. In other words, since we are comparing two operands using the operations $>$, $\geq$, or $=$, and that these operands are polynomials in the permissible input variables z, we can label the branching nodes with operations of the form $g(z) > 0$, $g(z) \geq 0$, or $g(z) = 0$, where $g(z)$ is a polynomial in z. We can restate this as: the direction of the branch only depends on the sign of the function $g(z)$, where the sign of the function at a point is defined as $+$, $-$, or 0 depending on whether the function is positive, negative, or zero at that point.

Therefore, two permissible inputs z and z' follow the same branches up to level t (*i.e.* be t-equivalent) if the various polynomials that they encounter at the nodes of the different processors have the same sign. Thus, what we need is some upper bound on the number of sign-invariant components into which the various g_i's at the t^{th}-level of the computation divide the space of permissible inputs $\mathbb{Z}^d$ (Figure 3.1). This depends on the degree of the polynomials g_i which we shall denote by $\deg(g_i)$.

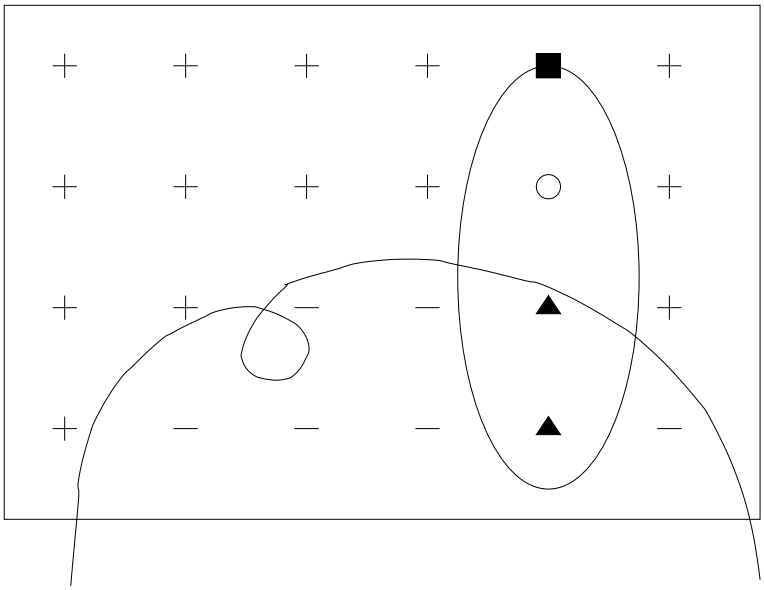

Figure 3.1: Sign-invariant components of polynomials

The bound on the number of sign-invariant components can be obtained by appealing to the classical bound in algebraic geometry of Milnor and Thom which gives an upper bound on the number of sign-invariant components of a set of polynomials.

Theorem 3.3.5 (Milnor-Thom) The number of non-empty sign-invariant components in the preceding stratification of $\mathbb{Z}^d$ is at most

$$\left(2 + 2\sum_i \deg(g_i)\right)^d. \tag{3.3}$$

The original theorem of Milnor and Thom gives a bound on the number of common zeros of a set of polynomials $\{g_i\}$ in $\mathbb{R}^n$ each of degree at most d. We can use the original theorem to give a bound on the number of sign-invariant computations by replacing each *hypersurface* $g(z) = 0$ by two hypersurfaces $g(z) = \pm\epsilon$ (as shown in Figure 3.2) where ϵ is an infinitesimal positive real. If we choose ϵ small enough then the new hypersurfaces do not contain any permissible integer point, and hence, the partition of the permissible integer points by the hypersurfaces into sign-invariant components remains the same. Now we can apply the original theorem to obtain the bound stated in Theorem 3.3.5.

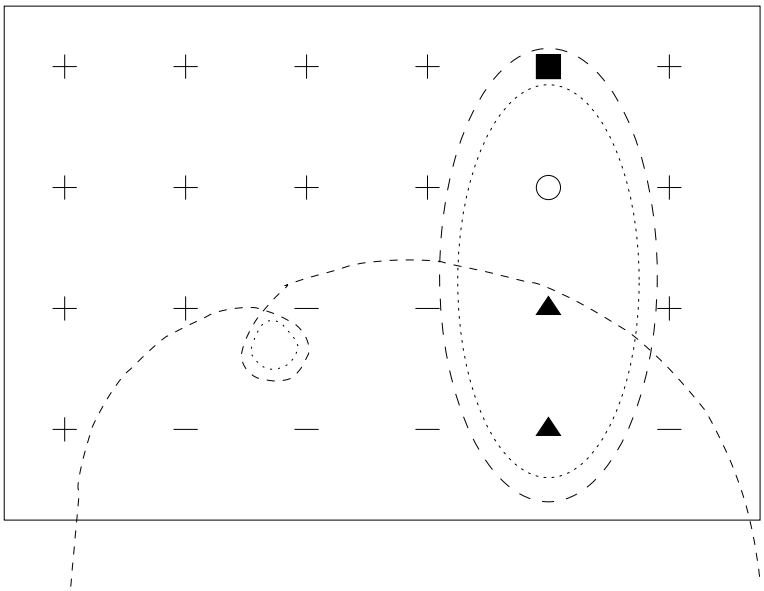

Figure 3.2: Perturbation of the polynomials

If we fix some equivalence class at step t of the computation, then at the next stage of the computation this class gets subdivided into a finer set of equivalence classes. Each equivalence class is at some node of the computation in the various processors, and hence, branches according to some polynomial g_i. The total number of such polynomials is clearly $p(n,N)$, and each has degree at most 2^t, which by using the above theorem of Milnor and Thom, gives us the following relation between $\sigma(t+1)$ and $\sigma(t)$.

$$\sigma(t+1) \leq \left(2 + 2 \cdot 2^t p(n,N)\right)^d \cdot \sigma(t)$$

The lemma on the number of equivalence classes follows by induction on t.

We can obtain an easy corollary from the statement of the lemma about the number of equivalence classes.

Lemma 3.3.6 For any t-equivalence class C, there are a set of $\Gamma(C,t)$ polynomial constraints (equalities or inequalities) in d variables such that $z \in C$ iff all the constraints are satisfied. Moreover, the degree of each polynomial is at most 2^t, and $|\Gamma(C,t)| \leq p(n,N)t$.

In other words, at the end of the computation, we can identify a set of polynomial constraints which number

$$\left(2 + 2 \cdot 2^{t(n,N)} p(n,N)\right)^{d \cdot t(n,N)} \cdot p(n,N)t(n,N) \tag{3.4}$$

whose sign-invariant components totally determine the fate of the computation. Each component is labeled YES or NO, and the inputs which are functions of the permissible inputs $z \in \mathbb{Z}^d$ are accepted if they are in a sign-invariant component labeled YES.

3.3.2 The Role of Parameterization

We have noted that restricting the inputs to a small affine subspace gives us a bound on the number of sign-invariant components of the computation. Mulmuley's proof now proceeds by showing that if we restrict the inputs to lie in a subspace spanned by a small number of parameters, then a partition of the inputs by the number of polynomials given in Equation 3.4 cannot exist.

Fix a parameterization $\mathcal{P}$ for cardinality n and bit-length $\beta(n)$ that gives $\rho(n)$ breakpoints for the graph of the optimization problem. For a given rational number λ, each numeric parameter in $\mathcal{P}(\lambda)$ is of the form $u\lambda + v$ where we can assume without loss of generality that $u, v \in \mathbb{Z}$ (because if not, we can scale the parameterization $\mathcal{P}$ by a large enough integer without changing the number of breakpoints).

Although the term $u\lambda + v$ might be rational, we expect all of our inputs to be integers. Since each rational number can be represented as a ratio of two integers, we counter this problem by replacing $\lambda = z_1/z_2$, and the original parameterization $\mathcal{P}$ by a homogeneous parameterization $\widetilde{\mathcal{P}}$ in two parameters z_1 and z_2 so that each original linear form $u\lambda + v$ is replaced by the corresponding integral form $uz_1 + vz_2$. The bit-length $\beta(n)$ and the complexity $\rho(n)$ of $\widetilde{\mathcal{P}}$ is the same as that of $\mathcal{P}$.

Since we are dealing with decision problems instead of the original optimization problems, we have an extra input z_3 which represents the *threshold* of the optimization problem. In other words, the algorithm has to decide whether the optimum value $F(I)$ of the input I is greater than or equal to the threshold z_3. The inputs I are restricted to be the inputs $\widetilde{\mathcal{P}}(z_1, z_2)$ are discussed above.

Lemma 3.3.7 For sufficiently large values of $\alpha > 0$, if we restrict the inputs to $(I, w) = \left(\widetilde{\mathcal{P}}(z_1, z_2), z_3\right)$, where z_1, z_2, z_3 all have bit-lengths $\leq a\beta(n)$, then no PRAM without bit operations can determine whether $F(I) \leq w$ correctly for all inputs I within $\sqrt{\log \rho(n, \beta(n))}/\kappa$ parallel time using $2^{\sqrt{\log \rho(n,\beta(n))}/\kappa}$ processors where κ is a constant independent of α.

By placing $d = 3$ in the statement of Equation 3.4, we can obtain the following statement:

Lemma 3.3.8 For any positive constant κ, if a PRAM works correctly on every input $I(z_1, z_2, z_3)$ for permissible values of z_i in time $t = \sqrt{\log \rho}/\kappa$ using 2^t processors, then $\mathbb{R}^3$ can be partitioned into at most 2^{20t^2} algebraic surfaces of degree at most 2^t and each sign-invariant component can be labeled YES or NO so that a permissible integer point lies in an sign-invariant component labeled YES iff $F(I) \leq z_3$.

The rest of the proof proceeds by showing that such a decomposition of $\mathbb{R}^3$ cannot be achieved for large enough constants α and κ.

3.3.3 A Problem on Lattices

Let $F(\lambda)$ be the optimum function associated by the homogeneous parameterization $\mathcal{P}$ and let G be its graph. An input $I(z_1, z_2, z_3)$ is *feasible* iff $\widetilde{\mathcal{P}}(z_1, z_2) \leq z_3$. In other words, it is feasible iff

$$F\left(\frac{z_2}{z_1}\right) \leq \frac{z_3}{z_1}.$$

There is a natural way of looking at the graph G in terms of projective coordinates. Imagine G to be sitting in the plane $z_1 = 1$ in $\mathbb{R}^3$ where $\mathbb{R}^3$ is viewed as a two-dimensional projective space. Each point in $\mathbb{R}^3$ can then be projected on to the affine plane by the ray that passes through the point and the origin. Hence, any point (z_1, z_2, z_3) is feasible iff it lies beneath the graph G in the projection. We can state the same criterion in the reverse fashion by projecting the shadow of the graph outward into $\mathbb{R}^3$ and calling it the fan(G). A point $(z_1, z_2, z_3) \in \mathbb{R}^3$ is feasible iff it lies below the fan. The situation is illustrated in Figure 3.3.

In the presentation that follows, we follow the standard convention of naming coordinates in $\mathbb{R}^3$ by renaming the coordinates z_1, z_2, and z_3 as z, x and y respectively. Note that this use of x should not be confused with the earlier use of x to denote a single instance of the input.

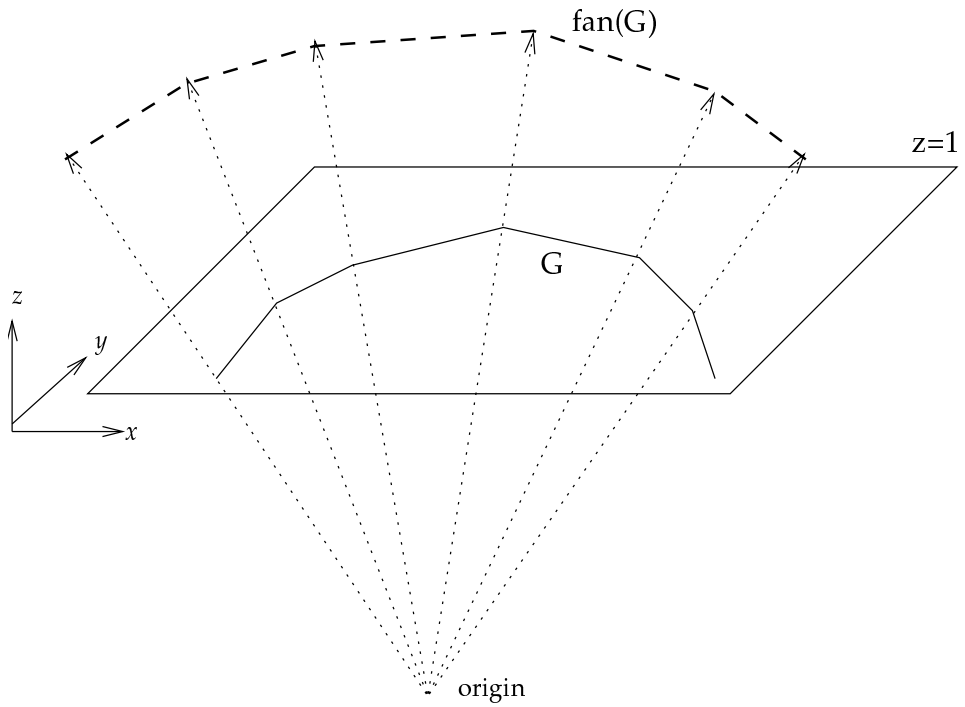

Figure 3.3: Graph G and its fan

The graph G is piecewise linear and has ρ segments. All its vertices have rational coordinates of bit-length at most β (or in equivalent words, size $\mu = 2^{\beta}$). In what follows, it will be much more convenient to work in terms of the size μ as opposed to the bit-length β.

Let us construct a bounding box defined by $|x| \leq 2\mu, |y| \leq 2\mu$ as shown in Figure 3.4. Clearly all the vertices of G lie within the bounding box.

Project the outline of the bounding box outward from the origin, and consider two bounding planes $z = \bar{\mu}$ and $z = 2\bar{\mu}$ as shown in Figure 3.5. The constant $\bar{\mu}$ will be chosen later in the course of the proof, such that $\log \bar{\mu}$ is a large constant multiple of $\log \mu$, *i.e.*, $\bar{\mu} = \mu^{O(1)}$. The slab shown in the figure is referred to as the *block* B. All of its coordinates have size at most $2\bar{\mu}$ (or equivalently bit-length at most $2\bar{\beta}$ where $\bar{\beta} = \log \bar{\mu}$).

The computation of the various processors defines sign-invariant components. In particular, we can talk about the components within the block B. The following lemma shows that if the number of breakpoints ρ is sufficiently large,

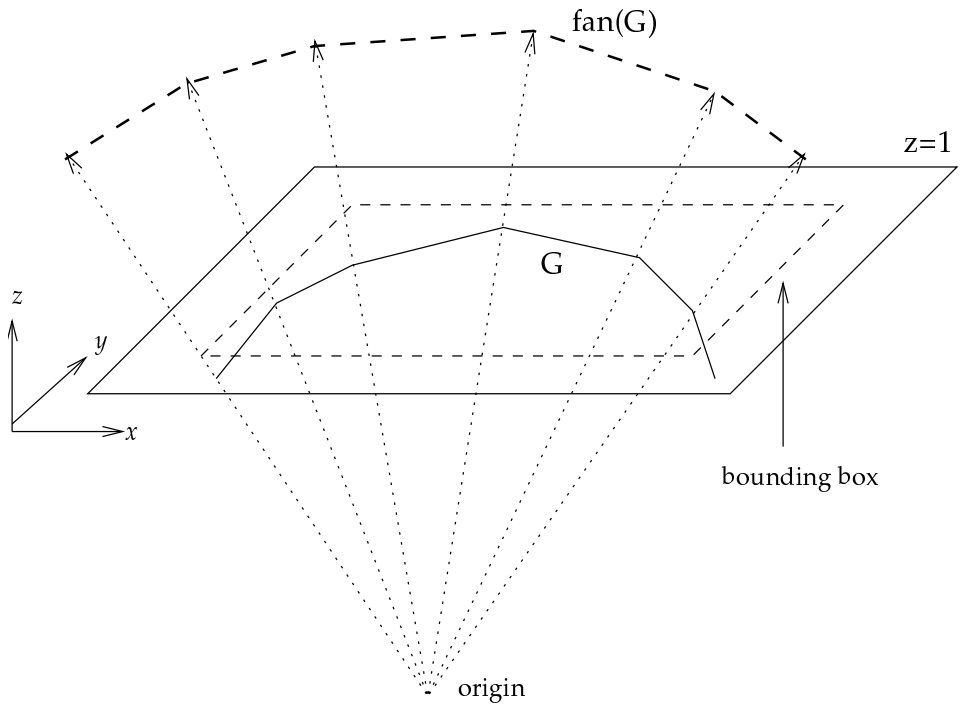

Figure 3.4: The Bounding Box

then a small number of surfaces cannot divide the region into sufficiently many parts to classify the points above and below the fan correctly. Hence, no such computation can exist.

Lemma 3.3.9 Let S be any set of surfaces in $\mathbb{R}^3$ of total degree D. Let $\bar{\beta} = \log \bar{\mu}$ be greater than a large enough multiple of $\beta = \log \mu$, and let $\log \rho$ be greater than a large enough multiple of $\log D$. Then, at least one sign-invariant component in the partition of the block B formed by the surfaces in S has an integer point lying below fan(G) and also an integer point lying above the fan.

Before we try to prove this theorem, let us observe how the Lemmas 3.3.8 and 3.3.9 combine to give us Theorem 3.3.1.

Assume that we have a set of machines that runs in time $t = \log \rho / \kappa$. Then by Lemma 3.3.8, we can obtain 2^{20t^2} surfaces of degree at most 2^t which partition the block B such that each sign-invariant portion can be labeled with the required answer.

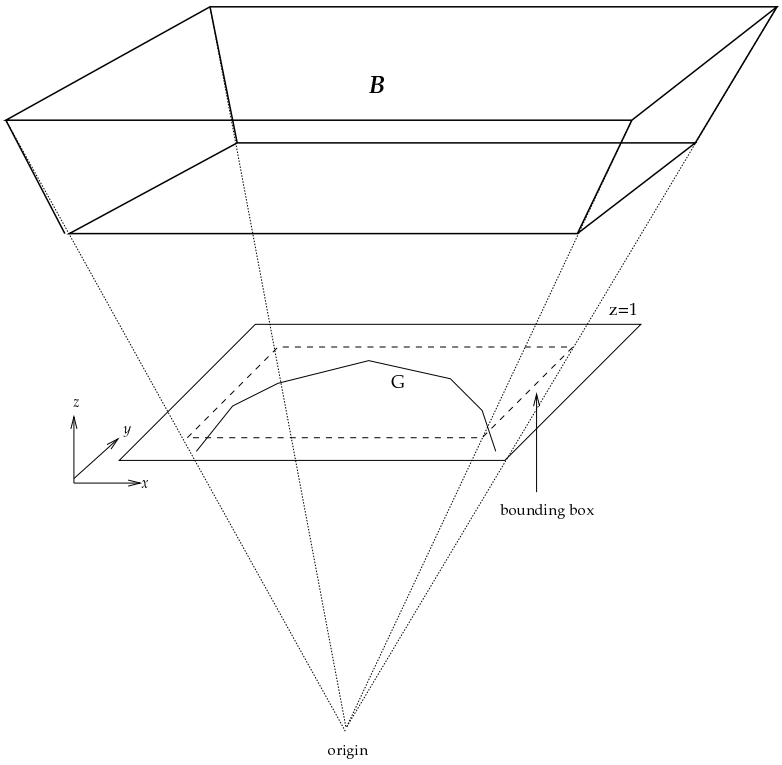

Figure 3.5: The Block B

The total degree of the surfaces from Lemma 3.3.8 is at most $2^t \cdot 2^{20t^2} < 2^{21t^2}$ where $t = \log \rho/\kappa$. Choose the constants a and κ large enough so that they satisfy the constraints in Lemma 3.3.9. This lemma tells us that there is a some sign-invariant component which contains two integer points, one from below the fan and one from above it, which means that the computation must have erred on these two inputs. This gives us the required contradiction.

3.3.4 Collins' Decomposition

The crux of the proof lies in proving the statement of Lemma 3.3.9. For this we shall need several additional tools from geometry which we shall develop in this section.

We have already shown how to perturb the surfaces in the set S without changing the sign-invariant components, so that no integer point can lie on any of them (Figures 3.1 and 3.2).

Definition 3.3.10 Let Q denote the set of surfaces whose elements are:

1. the surfaces in S,
2. the planes bounding the slab B,
3. a set of $6D$ planes parallel to the affine plane $z = 1$ that divide the slab B into slabs of equal height.

Definition 3.3.11 Let s be a surface. A point p on the surface s is said to belong to its *silhouette* as seen from the origin if the straight line joining the origin to the point p is tangent to the surface s at the point p.

If the surface s is defined by a polynomial equation $f(x, y, z) = 0$, a point $p = (a, b, c)$ is on the silhouette if it satisfies $f(a, b, c) = 0$ and if the tangent vector along the surface at that point is orthogonal to the vector from the origin to the point. This can be restated as

$$a\frac{\partial f}{\partial x}\bigg|_p + b\frac{\partial f}{\partial x}\bigg|_p + c\frac{\partial f}{\partial x}\bigg|_p = 0.$$

The silhouette of a surface s is a smooth space curve if the surface s is in general position. However, we have no control over the set of surfaces in the set Q. In particular, they may be highly degenerate. However, by Sard's theorem, we can perturb all the surfaces by infinitesimal reals without disturbing the sign-invariant components. In other words, we can assume without loss of generality that the surfaces in Q (and hence in S) are in *general position* with respect to each other. This allows the proof to access transversality techniques which were hitherto not available because of the degeneracy in the surface intersections.

The following space curves are projected onto the affine plane:

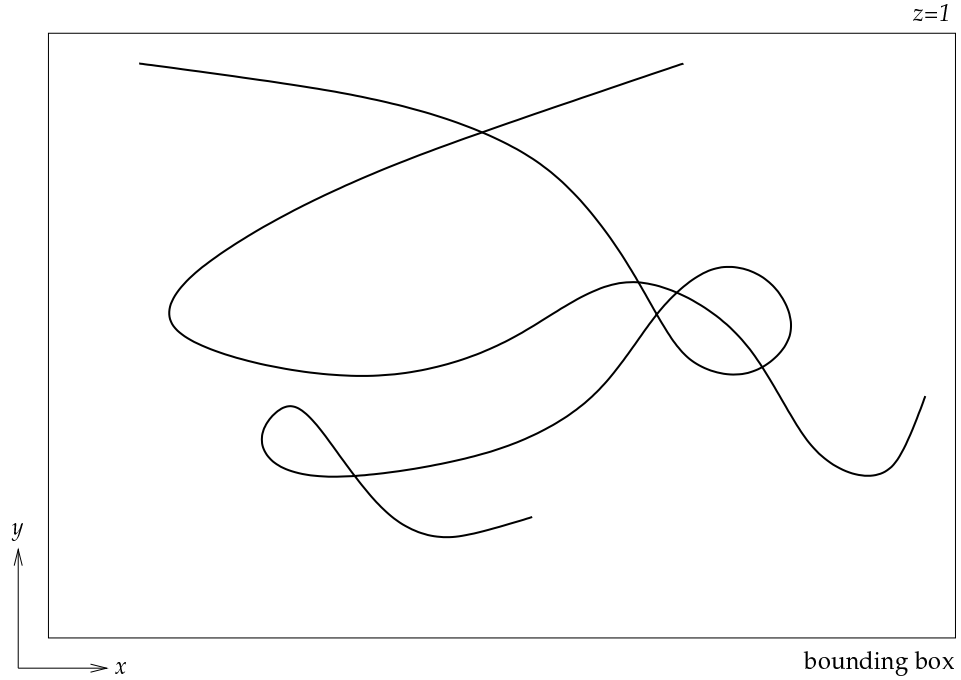

Figure 3.6: Projections of the Intersections (and the silhouettes)

1. the silhouette of every surface in S, and
2. the intersection between every two pair of surfaces in Q (because of our general position assumption, all of these are also smooth space curves).

An example of such a projection is shown in Figure 3.6.

The arrangement of curves thus obtained in the affine plane is further refined by passing lines parallel to the y-axis (vertical lines) through the following:

1. all points of intersection among the curves,
2. all *singular* points on the curves,
3. all *critical* points on the curves (where the tangents become parallel to the y-axis).

The resulting partition of the bounding box into regions is known as the two-dimensional *Collins' decomposition*, and is denoted as $A(Q)$. It has the property that the intersection of any connected region (also known as a *cell*) with a line

parallel to the y-axis is connected (assuming the intersection is non-empty). The Collins' decomposition of the curves in Figure 3.6 is shown in Figure 3.7.

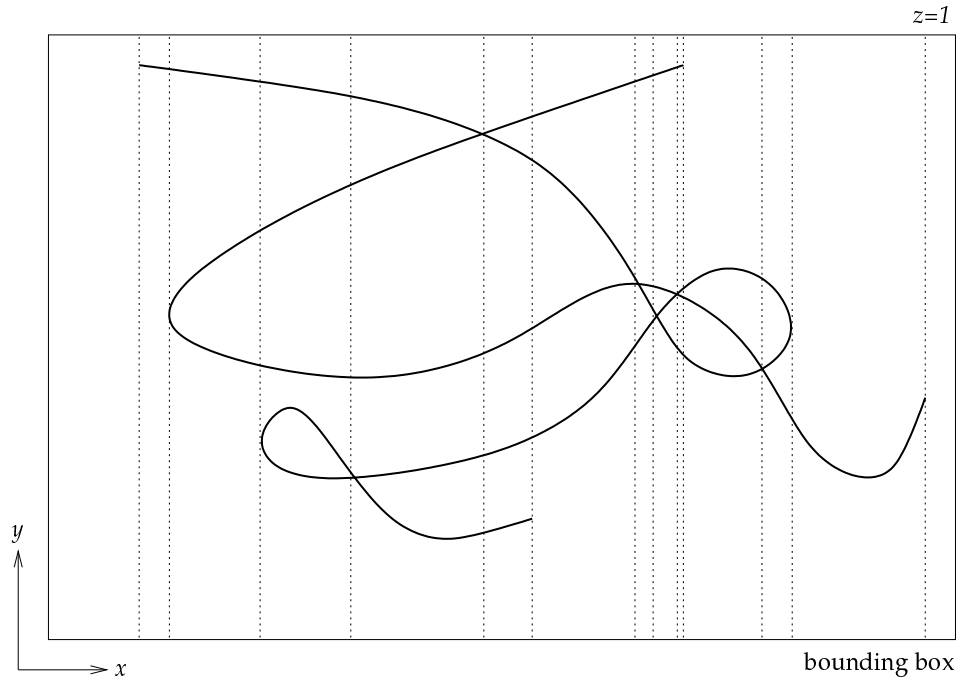

Figure 3.7: Decomposition $A(Q)$ in the Affine Plane

The partition $A(Q)$ can be lifted to the slab B in a natural fashion, by passing rays through all the projected curves, and the auxiliary vertical lines in the affine plane. This yields a decomposition of B into three-dimensional blocks denoted by $D(Q)$. The decomposition has the following properties:

1. Each region has at most six sides.
2. The intersection of each cell with a ray from the origin is connected (assuming that it is non-empty).
3. An intersecting ray intersects any given cell in exactly two surfaces (the nearer one being called the *floor*, and the further one the *roof*).
4. The remaining four sides are contained in fan surfaces.
5. The projection of each cell in $D(Q)$ is a cell in the affine partition $A(Q)$.

6. The total number of cells $d(Q)$ is $O\left(D^{O(1)}\right)$. This statement actually follows from the result of Milnor and Thom because the total degree of all the surfaces in Q along with the fan surfaces can be bounded by $O\left(D^{O(1)}\right)$.

3.3.5 The Choice of Sample Points

In order to derive the contradiction, we now choose sample points on all the edges of the graph G. Let C be a large enough integer constant to be chosen later. Divide each edge into equal spans along the x-axis by placing D^C points on it. Ignore the two outermost unbounded edges of G (if they exist). These points are referred to as the *sample points*. The x-coordinates of each of these differ by at least $1/\mu D^C$, because the coordinates of all vertices of G are rationals that can be expressed with absolute value at most μ. The total number of sample points is at least $(\rho - 1)D^C$. We denote the total number of cells by $d(Q)$.

Definition 3.3.12 Given a sample point p on an edge e of the graph G, we say that a particular cell $R \in D(Q)$ is *good* for p if its interior contains an integer lattice point on the projective ray through the point p.

The cell R is said to be *good* for an edge if it is good for $1/d(Q)^{\text{th}}$ fraction of the sample points on the edge e.

Lemma 3.3.13 The partition $D(Q)$ contains a cell that is good for $1/d(Q)^{\text{th}}$ fraction of all the edges in G.

PROOF: The number of cells in $D(Q)$ is $d(Q)$. Hence it suffices to show that for each edge $e \in G$, there is at least one good cell in $D(Q)$. An application of the pigeonhole principle then guarantees the existence of the cell needed by the lemma.

If we apply the pigeonhole principle once again, we only need to show that for each sample point p on the edge e, there is at least one good cell in $D(Q)$.

Fix a sample point p on an edge e of the graph. We will show that there is at least one good cell in $D(Q)$ for this point p.

The $6D$ dividing planes in slab B split any ray through the origin into $6D+1$ parts. In particular, it splits the ray through p into $6D+1$ parts. At most D of these intervals intersect surfaces in S. (This can be seen easily by applying Bezout's theorem to the surfaces in S and the ray passing through p).

Hence, there are a large number of intervals that are not intersected by any surface in S. Pick any one of these and call it e_p. The endpoints of this interval e_p lie on two adjacent horizontal dividing planes P_1 and P_2 and no part of e_p intersects a surface S. Such a cell whose roof and floor are horizontal planes is referred to as a *flat* cell. The vertical distance between the two points of intersection is therefore $\bar{\mu}/(6D+1)$.

CLAIM: The interior of the edge e_p contains a point on the integer lattice.

PROOF: Let the coordinates of the point p be $(u, v, 1)$. u and v are rational numbers whose numerator and denominator are at most μ.

Hence, the endpoints of e_p are (ru, rv, r) and (su, sv, s) for some $r, s \in \mathbb{R}$ such that $(s-r)\mu > \bar{\mu}/(6D+1)$. In other words, we have:

$$(s-r) > \frac{\bar{\mu}}{\mu(6D+1)}$$

Since we have assumed that $\log \bar{\mu}$ is a large enough multiple of $\log \mu$ and $\log D$, this means that there is an integer point m between r and s, such that the point (mu, mv, m) belongs to the integer lattice.

□

Thus, the flat cell containing e_p is good for the point p, and this proves the claim of the lemma.

□

The graph G has ρ distinct edges of different slopes. Hence, there is a flat cell $C \in D(Q)$ that is good for at least $\phi = \rho/d(Q)$ such edges. Let the ϕ edges be labeled $e_1, e_2, \ldots$ from left to right in the affine plane.

The projection of the cell C to the affine plane is a cell T in the affine partition $A(Q)$. The upper surface of the projection can only be a single surface in $A(Q)$

by the definition of the partition, and hence defines a smooth function $\theta(x)$ along the x-span of T.

The rest of the proof proceeds by giving upper and lower bounds on the extrema of the second derivative of this curve θ, *i.e.* the number of places where $\theta^{(3)}(x) = 0$. The two bounds are incompatible with each other for suitable choices of certain constants in the proof which yields the desired contradiction.

Lemma 3.3.14 The second derivative of the function $\theta(x)$ has at least $\phi - 1$ extrema along the x-span of T.

PROOF: The idea in the proof is to show that the curve θ gets very close to several sample points on each segment e_i. Since the segments are all within a small bounding box, and the slopes are all different, the curve θ must change direction several times.

Fix one of the segments e_i. Since the cell C is good for e_i, it is good for at least $1/d(Q)$ fraction of the sample points on e_i. Since the number of sample points on e_i is D^C, and $d(Q) = O\left(D^{O(1)}\right)$, we can choose C large enough so that there are at least eight sample points on e_i for which the cell C is good. Order these points from left to right. All of these points must lie within the affine cell T because the rays through all these points contain an integer point in the cell C. Fix one of these points $p = (a, b, 1)$.

CLAIM: $|\theta(a) - b| \leq 1/\bar{\mu}$.

PROOF: Consider the integer point on the ray through p in the cell C. Let this point be $\tilde{p} = (ka, kb, k)$ for some $k \in \mathbb{Z}$. Consider the point $\tilde{r} = (ka, kb + 1, k)$. This point lies within the slab B because we have chosen the bounding box to be sufficiently large. The projection of the point $r = (a, \tilde{b}, 1)$ onto the affine plane, must necessarily lie above the graph G. Hence, by the separation properties of our partition, the cell C cannot contain $\tilde{r}$. Because of the projection, we must have

$$\tilde{b} - b \leq \frac{1}{\bar{\mu}}.$$

Since $\tilde{r}$ lies outside the cell C, we must have that the curve θ separates p from r which proves the stated claim.

□

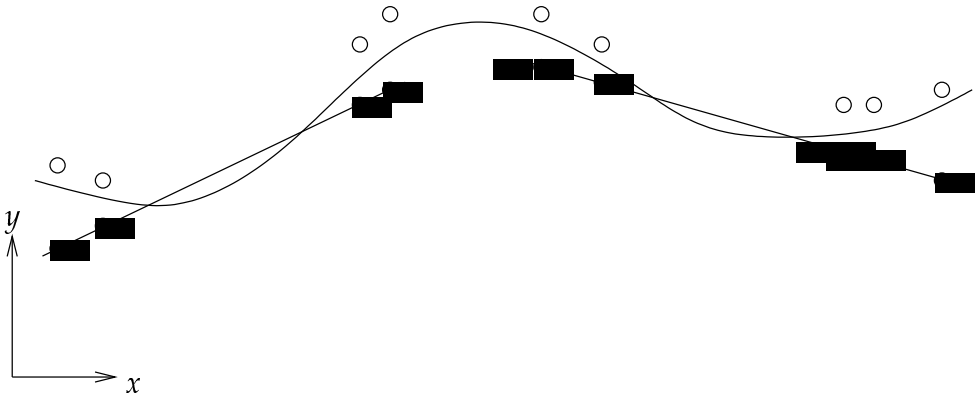

Figure 3.8: The curve θ and the sample points

An example of the sample points (in black) and the projections of the integer points above the curve (unshaded) onto the affine plane is shown in Figure 3.8.

Let $t_1, t_2, \ldots$ be the distinct slopes of the edges $e_1, e_2, \ldots$. The difference between any two slopes is at least $\Omega(1/\mu^2)$ because the coordinates of all vertices in G can be expressed as rationals with numerators and denominators of absolute value at most μ. By construction, the x-coordinates of any two sample points differ by at least $1/(\mu D^C)$.

The theorem follows from the repeated application of Rolle's mean-value theorem from calculus as follows:

1. The derivative $\theta'(x)$ attains a value very close to a_i between every two adjacent sample points on e_i. The error term is $O(\mu D^C/\bar{\mu})$.

2. This means that the second derivative θ'' is very close to zero between the x-coordinates of every third sample point.

3. The absolute value of the second derivative must change by at least $\Omega(|a_{i+1} - a_i|/\mu)$ between sample points on e_i and e_{i+1}, with error at most $O(\mu^2 D^{2C}/\bar{\mu})$.

Hence, each pair (e_i, e_{i+1}) contributes at least one distinct extremum of the second derivative because we have chosen $\log \bar{\mu}$ to be larger than a sufficiently large constant multiple of $\log \mu$ and $\log D$.

□

Lemma 3.3.15 The second derivative of the function $\theta(x)$ has at most $O\left(D^{O(1)}\right)$ extrema along the x-span of T.

PROOF: $\theta(x, y) = 0$ is the polynomial equation of the upper boundary of T. Its degree is at most $O\left(D^{O(1)}\right)$ because θ is either the projection of a silhouette or the intersection of two surfaces in S. Differentiate this equation implicitly thrice in a row to get the four implicit equations in x, y, and the three formal derivatives y', y'' and $y^{(3)}$.

$$\begin{aligned} \theta(x, y) &= 0 \\ \theta_1(x, y, y') &= 0 \\ \theta_2(x, y, y', y'') &= 0 \\ \theta_3(x, y, y', y'', y^{(3)}) &= 0 \end{aligned}$$

The extrema of the second derivative satisfy $y^{(3)} = 0$. By making that substitution for $y^(3)$, we get four equations in four unknowns. Because the surfaces S are in general position, these equations have at most $O\left(D^{O(1)}\right)$ solutions by the Milnor-Thom bound.

□

The two lemmas, Lemma 3.3.14 and Lemma 3.3.15 are in direct contradiction because $\phi = \rho / d(Q)$ and $d(Q) = O\left(D^{O(1)}\right)$, which tells us that $\rho = O\left(D^{O(1)}\right)$. Since, we have chosen $\log \rho$ to be larger than a sufficiently large constant multiple of $\log D$, we get the required contradiction. This completes the proof.

3.4 Lower Bounds for Randomized Algorithms

Mulmuley's theorem can be extended to produce similar lower bounds for parallel algorithms that flip bits with two-sided error. For details, we refer the reader to the original paper of Mulmuley [35]. We state the theorem here without proof.

Theorem 3.4.1 Let $\rho(n, \beta(n))$ be the parametric complexity of any homogeneous optimization problem where n denotes the input cardinality and $\beta(n)$ the bit-size of the parameters. Then the decision version of the problem cannot be solved in the randomized PRAM model without bit operations (with two-sided error) in $o\left(\sqrt{\log \rho(n, \beta(n))}\right)$ time using $2^{\Omega\left(\sqrt{\log \rho(n,\beta(n))}\right)}$ processors, even if we restrict every numeric parameter in the input to size $O(\beta(n))$.

CHAPTER 4

THE SHORTEST PATH PROBLEM

4.1 Introduction

The computation of the shortest path between two vertices in a graph (either directed or undirected) is one of the oldest problems in Computer Science. It has immense theoretical and practical importance. The problem has been studied extensively both in the general case, as well as in particular instances that arise frequently in practice.

The input to the SHORTEST PATH PROBLEM is a weighted, directed graph with two specified vertices s and t. The weight on an edge in the graph can be thought of as the length of the path between its two vertices. The objective is to compute the length of the shortest path between s and t.

If the graph has a negative directed cycle that intersects some path between s and t, it is clear that there can be no optimal solution, because by following the cycle around repeatedly, we can reduce the cost of the optimal path by an arbitrary amount. Thus, in the graphs that we consider we will assume *without loss of generality* that the graph has no negative cycles.

4.1.1 Algorithms

The SHORTEST PATH PROBLEM (Section 2.1.3) can be solved on a RAM by using a classical algorithm due to Dijkstra [13]. The algorithm works for graphs with non-negative weights, and runs in time $O(m + n \log n)$, where n denotes the number of vertices in the graph, and m refers to the number of the edges in the graph. The algorithm is a type of *greedy* algorithm, which constructs a set X

vertex by vertex, starting from the set $\{s\}$, and always choosing the next vertex as the one closest to vertices in X until all the vertices have been chosen.

There are other algorithms that work for graphs with negative weights. Additionally, they solve the problem of computing the shortest path between all pairs of vertices. The Floyd-Warshall algorithm [17, 48] solves the problem in $O(n^3)$ time, and Johnson's algorithm [27] runs in time $O(n^2 \log n + mn)$ (which is superior for sparse graphs).

In order to compute the length of the shortest path in parallel, let A be the *adjacency matrix* of the input graph. We define a multiplication operator $A^2 = A \otimes A$ where:

$$(A^2)_{ij} = \min_k \{a_{ik} + a_{kj}\}.$$

The entry $(A^2)_{ij}$ denotes the length of the shortest path with at most two edges connecting vertices i and j. The operator $\otimes$ is associative, and hence A^n is the matrix of weights of shortest paths between all vertices which have at most n edges. Since the graph has n vertices, this is the matrix of shortest paths between any two vertices in the graph.

The addition of n numbers can be done on a PRAM in exactly $\log n$ steps using $n/2$ processors by arranging the computation in the form of a tree. The same procedure allows us to compute the product of two $n \times n$ matrices in $\log n$ time using n^3 processors because all the entries of the product matrix are independent of each other, and hence can be computed in parallel. Computing A^n can be achieved in $O(\log^2 n)$ using n^4 processors by repeatedly squaring the matrix [10, 30].

The matrix-based *repeated squaring* algorithm for SHORTEST PATH PROBLEM can be solved in $\varepsilon \log n$ steps for any $\varepsilon > 0$ with poly(n) processors on a PRAM that allows unbounded fan-in min operations (but only bounded fan-in additions), because multiplying k matrices (for any fixed constant k) can be done in 2 steps in this model using n^{k+1} processors.

The results of this chapter show that this algorithm is optimal, even when the edge weights are restricted to be fairly small, and the running time of the algorithm is allowed to depend on the total bit-length.

4.1.2 Overview of the Chapter

The rest of the chapter is laid out as follows. Section 4.2 discusses the main result of this work. The lower bound is proved using the notion of parametric complexity (Section 3.2.2), and the precise statement of the lower bound can be found in Section 4.3. The main technical lemma is stated in Section 4.3.2. The inductive construction can be found in Section 4.4. Proofs are relegated to the end of the chapter in Section 4.5.

The proof is based on a theorem of Carstensen [8, 7]. However, Carstensen's proof is very complex and does not take into account the issue of bit-lengths. It is not possible to obtain a lower bound that is sensitive to bit-lengths without obtaining good bounds on the bit-lengths of the coefficients of the edge weights. We give a simplified proof of her theorem (using a similar construction) which allows us to to take into account the issue of bit-lengths.

4.2 The Main Result

Theorem 4.2.1 The SHORTEST PATH PROBLEM cannot be computed in $o(\log n)$ steps on an unbounded fan-in PRAM without bit operations using $n^{\Omega(1)}$ processors, even if the weights on the edges are restricted to have bit-lengths $O(\log^3 n)$.

In fact, we can prove a slightly stronger theorem.

Theorem 4.2.2 The SHORTEST PATH PROBLEM cannot be computed in $o(\log N)$ steps on an unbounded fan-in PRAM without bit operations using $N^{\Omega(1)}$ processors.

Since the model for the lower bound assumes unit cost for all operations (including some with unbounded fan-in), the result shows that the above algorithm for the SHORTEST PATH PROBLEM is optimal in the unbounded fan-in PRAM model without bit operations.

4.3 Parametric Complexity

Our proof proceeds by giving a lower bound on the parametric complexity of the SHORTEST PATH PROBLEM.

Theorem 4.3.1 There is an explicit family of graphs G_n on n vertices with edge weights that are linear functions in a parameter λ, such that the optimal cost graph of the weight of the shortest path between s and t has $2^{\Omega(\log^2 n)}$ breakpoints. In addition, the bit-lengths of the coefficients of the cost functions have size $O(\log^3 n)$. Thus, the parametric complexity of the SHORTEST PATH PROBLEM for graph size n and bit-length $O(\log^3 n)$ is $2^{\Omega(\log^2 n)}$.

4.3.1 Preliminaries

A directed graph is said to be *layered* if its vertices can be arranged in columns so that all edges go between vertices in adjacent columns.

All the graphs used in this chapter are directed and layered. We imagine the graph to be embedded in a grid and label each vertex of the graph by its coordinate. The vertex in the r^{th} row and c^{th} column is labeled as (r, c). Occasionally, we omit the column number for the vertex if it is clear from the context.

Edges are denoted by $(r, c) \to (k, c+1)$ or simply $r \to k$ if the column number is unambiguous from the context. The weights of edges are labeled by $w_{r,k}$. If we wish to emphasize the fact that the weights are linear functions in the parameter λ, we denote the weight as $w_{r,k}(\lambda)$.

All the graphs in this chapter have two special vertices s and t between which we wish to compute the shortest path. Since all the graphs we use are layered, we may assume that the vertex s sits in the 0^{th} column and the vertex t in the last column. The graph obtained by eliminating the special vertices s and t is referred to as the *core* of the graph.

4.3.2 The Technical Lemma

Theorem 4.3.2 For any $m, n \in \mathbb{N}$, there exists a graph with core $G_{m,n}$ having the following properties:

(i) $G_{m,n}$ is a layered graph with at most $4^m \cdot n$ vertices, and has exactly n vertices in the first column.

(ii) The edges of $G_{m,n}$ are labeled by linear functions in a parameter λ such that the optimal cost graph of the weight of the shortest path between s and t (as a function of λ) has at least n^m breakpoints.

(iii) There is an edge from s to each of the n vertices in the first column with weights
$$w_{s,(i,1)}(\lambda) = \frac{i(i+1)}{2} - i\lambda \qquad (0 \le i < n).$$

(iv) All the vertices q in the last column of the core are connected to t with weight
$$w_{(q,3^m),t} = 0.$$

The substitution $m = \log n$ will yield a graph on n^3 vertices such that the optimal cost graph of the shortest path has at least $2^{\log^2 n}$ breakpoints. We can rephrase this to say that graphs G_n on n vertices can be constructed with $2^{\Omega(\log^2 n)}$ breakpoints on the optimal cost graph of the shortest path. The coefficients involved in this construction will be shown to have bit-lengths $O(\log^3 n)$.

The construction will use negative edge weights, but since the graphs are layered, we can always add a large positive weight to each edge without changing the structure of the optimal paths.

4.4 Construction

The graph $G_{m,n}$ is constructed inductively from $G_{m-1,n}$. The idea behind the proof is that each optimal path in $G_{m-1,n}$ yields n optimal paths in $G_{m,n}$ with varying slopes, thus increasing the number of breakpoints by a factor of n.

Given a layered graph and a particular shortest path over some fixed interval of the parameter λ, one can easily create n shortest paths by breaking up the interval into n pieces, appending a new layer of n vertices, and attaching them to the endpoint of the given path with suitable weights. However, the weights would depend on the interval. The goal of the construction is to create a gadget that behaves the same way but with a choice of weight functions that are independent of the interval. In order to do this, we need the following stronger inductive hypothesis.

Lemma 4.4.1 For any $D_1, D_2 \in \mathbb{R}$, $m, n \in \mathbb{N}$, $g : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$ such that $g(r, 0) = 0$, and $0 < \epsilon < 1$, there is a graph with core $G_{m,n}$ that possesses the following properties:

(i) There exist n^{m-1} disjoint intervals, $\mathcal{I}_{j,m} = [\alpha_{j,m} + \epsilon, \beta_{j,m} - \epsilon]$ where $0 \le j < n^{m-1}$ such that $\beta_{j,m} - \alpha_{j,m} > n$. The intervals $\mathcal{I}_{j,m}$ depend only on m and n, and are independent of the parameters D_1, D_2 and g.

(ii) For each interval, there exist n paths $P_{i,j}$ (from vertices in the first column of G to the last column of G) that are pairwise vertex-disjoint.

Notationally, $P_{i,j}$ denotes the path in the core of the graph starting from the vertex $(i, 1)$, and $\mathcal{P}_{i,j}$ denotes the s-t path that contains $P_{i,j}$,
i.e., $\mathcal{P}_{i,j} = (s \to i) \cup P_{i,j} \cup (r_{i,j} \to t)$ where $r_{i,j}$ is the last vertex of $P_{i,j}$.

(iii) $P_{i,j}$ is the optimal path starting from vertex $(i, 1)$ in the interval $\mathcal{I}_{j,m}$, where $0 \le i < n$.

(iv) For $0 \le i < n$, let $j = nd + r$ where $0 \le r < n$. Then

$$C(P_{i,j})(\lambda) = C(P_{0,j})(\lambda) + iD_1\alpha_{d,m-1} + iD_2\lambda + g(r, i). \tag{4.1}$$

(v) The difference in cost between $P_{i,j}$ and any other non-optimal path starting at vertex $(i, 1)$ is at least ϵ.

4.4.1 Construction of the Intervals

Fix $N > mn^3$. Let $j = nd + r$ where $0 \leq r < n$. Define $\alpha_{j,m}$ and $\beta_{j,m}$ as follows:

$$\begin{aligned}
\alpha_{0,1} &= 0 \\
\beta_{0,1} &= N^2 \\
\alpha_{j,m} &= \alpha_{nd+r,m} = N\alpha_{d,m-1} + rN^2 \\
\beta_{j,m} &= \beta_{nd+r,m} = N\beta_{d,m-1} + (r+1)N^2
\end{aligned}$$

Intuitively, at each stage we stretch the intervals by a factor of N and divide it into n parts. Hence, $\beta_{j,m} - \alpha_{j,m} = N^2 \gg n$, and this satisfies condition (i) of the inductive hypothesis.

4.4.2 Construction of the Graph

The graph $G_{m,n}$ is constructed by induction on the parameter m.

4.4.2.1 The Base Case

The graph $G_{1,n}$ has 3 columns with n, n and $2n-1$ vertices respectively as shown in Figure 4.1. Each of the n vertices in the first column is connected to the corresponding vertex in the second column, and each vertex $(i, 2)$ in the second column is connected by n edges to the vertices $(i + j, 3)$ where $0 \leq j \leq n - 1$.

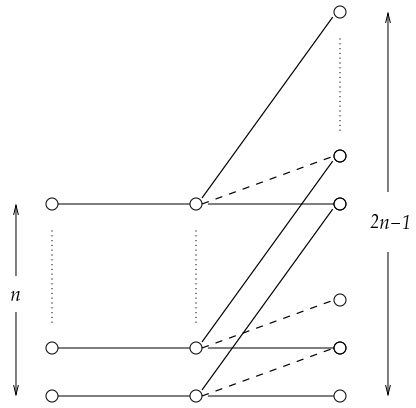

Figure 4.1: Construction of the Graph $G_{1,n}$

4.4.2.2 The Inductive Case

$G_{m,n}$ is constructed recursively from two copies of $G_{m-1,n}$ and a third copy of $G_{m-1,2n-1}$ (which are referred to as G^L, G^M, and G^R respectively) as shown in Figure 4.2.

The first two copies of $G_{m-1,n}$ are connected *back-to-back*. G^M is a reflection of G^L with the edges reversed as well. G^M has exactly n vertices in the last column (because it is a mirror image of G^L). G^R (which is a copy of $G_{m-1,2n-1}$) has $2n-1$ vertices in the first column. We connect the i^{th} vertex in the last column of G^M to the $(i+j)^{\text{th}}$ vertex in the first column of G^R where $0 \le j \le n-1$ (similar to the construction in the base case).

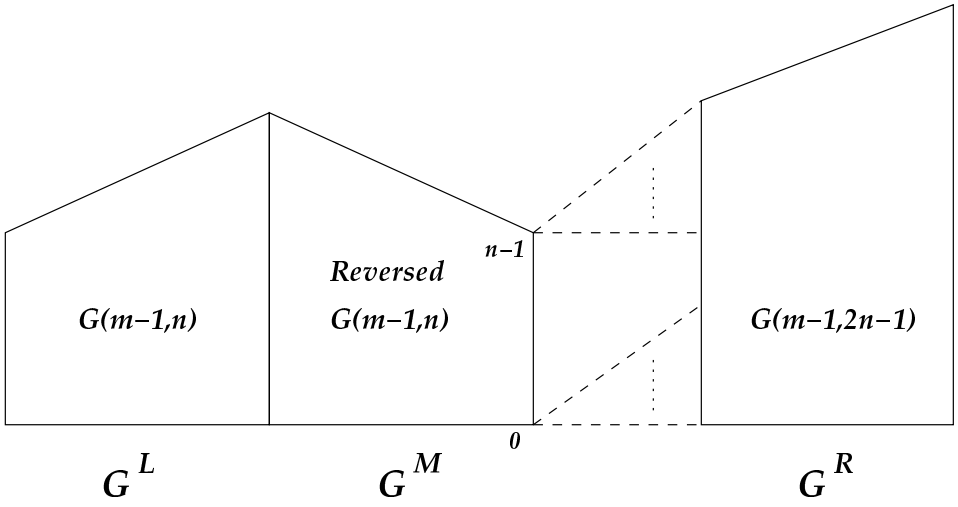

Figure 4.2: Construction of the Graph $G_{m,n}$

4.4.3 Construction of the Weight Functions

4.4.3.1 The Base Case

Fix the parameters D_1, D_2 and the function $g : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$. The parameter K in the definition of the weights is a constant whose value is fixed later. Define the weights on the edges as follows:

$$
\begin{aligned}
w_{(k,1),(k,2)} &= 0 \\
w_{(k,2),(k+r,3)} &= \begin{cases} K\left[\frac{r(r+1)}{2}N^2 - r\lambda\right] & k = 0, \\ & 0 \le r < n; \\ w_{(0,2),(r,3)} + kD_1\alpha_{0,1} + kD_2\lambda + g(r,k) & 1 \le k < n, \\ & 0 \le r < n. \end{cases}
\end{aligned}
$$

4.4.3.2 The Inductive Case

Let the parameters to the construction be F_1, F_2, and $h : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$. The symbols K_1, K_2, and K_3 stand for constants whose values will be fixed later.

(We use the symbols F_1, F_2, and h as the parameters to the construction instead of the familiar D_1, D_2, and g because we will need to set appropriate parameters for each of the three graphs used in the inductive construction, and we want to avoid cluttering the notation with superfluous superscripts.)

G^L and G^M are chosen with parameters:

$$
\begin{aligned}
D_1 &= \frac{N}{2K_3}\left(F_1 - \frac{K_2}{K_1}\right) \\
D_2 &= 0 \\
g(r,i) &= Nri D_1.
\end{aligned}
$$

The graph G^R (which is a copy of $G_{m-1,2n-1}$) is chosen with parameters:

$$
D_1 = \frac{N}{K_1}
$$

$$\begin{aligned} D_2 &= -\frac{1}{K_1} \\ g(r,i) &= \frac{N}{K_1}\frac{i(i+1)}{2} + NriD_1. \end{aligned}$$

Since G^M has exactly n vertices in the last column and G^R has $2n-1$ vertices in the first column, we define the edges analogously to the base case but with the following weights:

$$w_{i,i+r}(\lambda) = h(r,i) - N\frac{K_2}{K_1}\left\{ir + \frac{i(i+1)}{2}\right\} + i\left(F_2 + \frac{K_2}{NK_1}\right)\lambda \qquad (0 \le r < n).$$

The cost functions on the edges of $G_{m,n}$ are defined by shifting the cost functions in G^L and G^M and then scaling them by a factor of K_3, by shifting the cost function in G^R, and then scaling it by a factor of K_2 as follows:

$$w_e(\lambda) = \begin{cases} K_3 \cdot w_e^L\left(\frac{\lambda}{N}\right) & e \in G^L \\ K_3 \cdot w_e^M\left(\frac{\lambda}{N}\right) & e \in G^M \\ K_2 \cdot w_e^R\left(\frac{\lambda}{N}\right) & e \in G^R. \end{cases}$$

4.5 Proofs

4.5.1 Proof of the Technical Lemma

4.5.1.1 Proof of the Base Case

PROOF: [BASE CASE]

Define $P_{i,j}$ to be the path $((i,1) \to (i,2)) \cup ((i,2) \to (i+j,3))$. These paths $P_{i,j}$ are vertex-disjoint. All of the conditions of the inductive hypothesis can be easily verified provided:

$$K \gg \frac{\max_{r,i} |g(r,i)|}{\epsilon}.$$

□

4.5.1.2 *Proof of the Inductive Case*

PROOF: [INDUCTIVE CASE]

Fix j and $\lambda \in \mathcal{I}_{j,m}$. Let $j = nd + r$ where $0 \le r < n$. Then $\lambda/N \in \mathcal{I}_{d,m-1}$, and hence the path $P^L_{i,d}$ is optimal in G^L starting from vertex $(i,1)$.

Define $P_{i,j}$ to be $P^L_{i,d} \cup P^M_{i,d} \cup (i \to i+r) \cup P^R_{i+r,d}$. The following two lemmas provide the proof that $P_{i,j}$ is an optimal path starting at vertex $(i,1)$. These paths are vertex-disjoint (as required by the inductive hypothesis).

Lemma 4.5.1 shows that $P^M_{i,d}$, which is the mirror image of $P^L_{i,d}$, is optimal in G^M. This is not at all clear *a priori* since there is no reason to believe that optimal paths will remain optimal when the edges are reversed. In particular, we would like to have the optimal path in G^M end at $(i, 2 \cdot 3^{m-1})$. Lemma 4.5.2 finishes up the proof by showing that the path $P^R_{i+r,d}$ is indeed optimal in G^R in the interval $\mathcal{I}_{j,m}$.

Lemma 4.5.1 Fix $\lambda \in \bigcup_{\substack{k=nd+r \\ 0 \le r < n}} \mathcal{I}_{k,m}$. Then for sufficiently large values of K_3, the optimal path in G^L and G^M starting at node $(i,1)$ is $P^L_{i,d} \cup P^M_{i,d}$.

PROOF: Assume that Q is the optimal path in this interval, and that Q is not symmetric in G^L and G^M. Further, assume without loss of generality that $Q^L = P^L_{i,j}$ and $Q^M \neq P^M_{i,j}$. Let Q^M end at vertex k where $k \neq i$.

The idea of the proof is that the difference between the costs of $P^M_{i,j}$ and $P^M_{k,j}$ is small but the difference in costs between Q^M and $P^M_{k,j}$ is at least ϵ before scaling, and hence at least $K_3\epsilon$ after scaling.

Consider the situation in Figure 4.3. Before scaling, in the graph $G_{m-1,n}$, the inductive hypothesis guarantees that

$$C(Q^M)(\lambda) - C(P^M_{k,j})(\lambda) > \epsilon.$$

Therefore, after scaling we have that

$$C(Q^M)(\lambda) - C(P^M_{k,j})(\lambda) > K_3\epsilon.$$

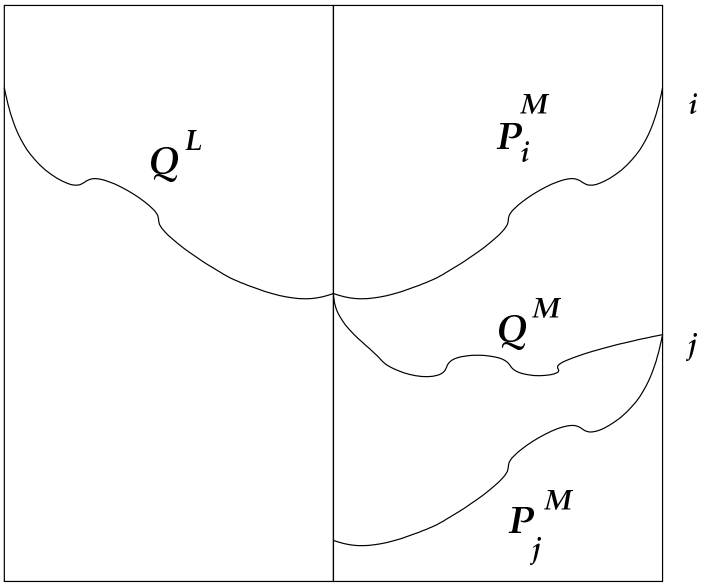

Figure 4.3: Proof of Lemma 4.5.1

The difference in costs between the *parallel* paths $P_{i,j}$ and $P_{k,j}$ is small by the inductive hypothesis:

$$\begin{aligned} \left| C(P^M_{i,j})(\lambda) - C(P^M_{k,j})(\lambda) \right| &\leq nN\left(F_1 - \frac{K_2}{K_1}\right)N^{2+m} \\ &\leq n^{O(m)}\left(F_1 + \frac{K_2}{K_1}\right). \end{aligned}$$

We have that $C(Q)(\lambda) = C(Q^L)(\lambda) + C(Q^M)(\lambda) + C(Q^R)(\lambda)$. Again, without loss of generality Q^R is optimal. It is possible that the path Q gains some advantage in the links between G^M and G^R and also in G^R. If we take the quantity $K_3\epsilon$ to be greater than all these gains and the quantity from above, this would contradict the assumption that Q is the optimal path in this interval.

The maximum gain in the intermediate links from Equation (4.2) is

$$\max_{r,i} h(r,i) + 4n^2N\frac{K_2}{K_1} + n(F_2 + \frac{K_2}{NK_1})N^{1+m}.$$

The maximum gain that can be achieved in G^R is $nN\frac{K_2}{K_1}N^{2+m}$. Clearly the quantity $n^{O(m)}\left(F_1 + F_2 + \max|h| + \frac{K_2}{K_1}\right)$ dominates all of the above terms. Thus,

choosing K_3 to be

$$K_3 \gg \frac{n^{O(m)}\left(F_1 + F_2 + \max|h| + \frac{K_2}{K_1}\right)}{\epsilon} \tag{4.2}$$

gives us a contradiction, and thus proves our lemma.

□

Lemma 4.5.2 Fix $\lambda \in \bigcup_{\substack{k=nd+r \\ d}} \mathcal{I}_{k,m}$. Then if K_2/K_1 is sufficiently large, the optimal path in G^R starting from node $(i,1)$ is $P^R_{i+r,d}$.

PROOF: Fix a particular value for d. Since $\lambda \in \mathcal{I}_{nd+r,m}$, therefore $\frac{\lambda}{N} \in \mathcal{I}_{d,m-1}$.

From the previous lemma, it is clear that the optimal paths are symmetric in G^L and G^M and that the optimal path in G^R is $P^R_{k,d}$ for some k. We claim that $k = i + r$.

By adding up the costs, we get that

$$C(P_{i,j})(\lambda) - C(P_{i-1,j})(\lambda) \quad = \quad \frac{1}{N}\frac{K_2}{K_1}(\alpha_{j,m} - \lambda) + h(r,i) - h(r-1,i)$$

which means that if we impose the condition

$$\frac{K_2}{K_1} \gg \frac{N \max_{r,i}|h(r,i)|}{\epsilon}, \tag{4.3}$$

then the optimal path is as required.

□

We now continue the proof of the inductive step. The inductive hypothesis gives us the following equations about paths in G^L, G^M, and G^R (before scaling):

$$\begin{aligned}
C(P^L_{i,d})(\lambda) &= C(P^L_{0,d})(\lambda) + \frac{i}{2K_3}\left(F_1 - \frac{K_2}{K_1}\right)\alpha_{d,m-1} \\
C(P^M_{i,d})(\lambda) &= C(P^M_{0,d})(\lambda) + \frac{i}{2K_3}\left(F_1 - \frac{K_2}{K_1}\right)\alpha_{d,m-1} \\
C(P^R_{i,d})(\lambda) &= C(P^R_{0,d})(\lambda) + \frac{1}{K_1}\left\{\frac{Ni(i+1)}{2} + i\alpha_{d,m-1} - i\lambda\right\}.
\end{aligned}$$

After scaling, adding up the costs of the above paths along with the weights of the edges joining the end vertices of G^M to G^R and simplifying, we get:

$$\begin{aligned} C(P_{i,j})(\lambda) &= C(P^L_{i,d})(\lambda) + C(P^M_{i,d})(\lambda) + w_{i,i+r}(\lambda) + C(P^R_{i+r,d})(\lambda) \\ &= \left\{C(P^L_{0,d})(\lambda) + C(P^M_{0,d})(\lambda) + C(P^R_{r,d})(\lambda)\right\} \\ &\quad + iF_1\alpha_{d,m-1} + iF_2\lambda + h(r,i) \\ &= C(P_{0,j})(\lambda) + iF_1\alpha_{d,m-1} + iF_2\lambda + h(r,i). \end{aligned}$$

This yields the required relationships between the optimal paths in the interval $\mathcal{I}_{j,m}$. Condition (v) is easily satisfied by noting that if we have an optimal and a sub-optimal path starting from vertex $(i,1)$, then it must deviate from the optimal path in either G^L, G^M, G^R, or in the intermediate connecting links. Then the proof of Lemma 4.5.2 shows that the difference in optimal costs must be at least ϵ. This concludes the proof.

□

4.5.2 *Proof of the Main Theorem*

PROOF:

Let $G_{m,n}$ be the graph obtained by choosing the parameters $D_1 = N$, $D_2 = 0$ and $g(r,i) = N^2 ir$. Substituting the values into equation (4.1) and simplifying using the definition of the intervals above, we get the following equation for the optimal paths in the core of the graph:

$$C(P_{i,j})(\lambda) = C(P_{0,j})(\lambda) + i\alpha_{j,m}.$$

Therefore, for s-t paths we have that

$$C(\mathcal{P}_{i,j})(\lambda) = C(\mathcal{P}_{0,j})(\lambda) + i\alpha_{j,m} + \frac{i(i+1)}{2} - i\lambda.$$

Hence, we have that $\mathcal{P}_{i,j}$ is the optimal path for the interval

$$[\alpha_{j,m} + i, \alpha_{j,m} + i + 1] \cap \mathcal{I}_{j,m},$$

and since $\epsilon < 1$, it follows that each of these paths is optimal in a non-zero interval. Each path must have a different slope because its linear term depends on i. Hence we get n breakpoints in each of the n^{m-1} intervals, yielding n^m breakpoints in all.

□

4.5.3 Analysis of the Main Theorem

PROOF: [Proof of Theorem 4.3.1]

From Equations (4.2) and (4.3) in Section 4.5.1.2, and the fact that N is of size $O(n^3 \log n)$ and $\epsilon < 1$, we can rewrite the recurrences for the constants as

$$\begin{aligned} \frac{K_2}{K_1} &\gg n^{O(1)} \max_{r,i} |h(r,i)| \\ K_3 &\gg n^{O(m)} \left(F_1 + \frac{K_2}{K_1} \right). \end{aligned}$$

At the topmost level of the recurrence, we choose $F_1 = N$ and $h(r,i) = N^2 ir$, both of which are polynomial in n. However, the function h and parameter F_1 changes as we descend down the construction. In both G^L and G^M we choose $h(r,i) = N^2 ir$, and in G^R we have $|h(r,i)|$ dominated by poly$(n)\frac{K_2}{K_1}$. We can choose $a > 0$ large enough so that recurrence

$$\left(\frac{K_2}{K_1} \right)_{m-1} \gg \left(\frac{K_2}{K_1} \right)_m n^a$$

has the solution

$$\left(\frac{K_2}{K_1} \right)_r = n^{a(m-r)}.$$

Now in G^L and G^M, the quantity F_1 keeps decreasing by the current value of K_2/K_1, and hence its absolute value increases by at most K_2/K_1. Thus, we can

choose $c > 0$ sufficiently large so that

$$(K_3)_r \gg n^{cr} \sum_{r \le t \le m} \left(\frac{K_2}{K_1}\right)_t .$$

This yields the following solution:

$$(K_3)_r = n^{b(m-r)} \qquad \text{for some } b > a > 0.$$

The coefficients grow as the product of the individual multipliers:

$$\begin{aligned} \text{size of coefficients} &= O(n^{b(1+\cdots+(m-1)+m)}) \\ &= O(n^{b\binom{m}{2}}) \\ &= 2^{O(\log^3 n)} \qquad \text{since } m = O(\log n). \end{aligned}$$

Since the magnitude of the coefficients is $2^{O(\log^3 n)}$, it follows that their bit-lengths are $O(\log^3 n)$.

□

4.6 Corollaries

4.6.1 Shortest Paths in Sparse Graphs

In practice, it is often the case that the graphs for which we need to compute shortest paths are *sparse*, *i.e.*, the number of edges $m = O(n)$ as opposed to the general case where m can be as large as $\Omega(n^2)$. The construction that we give above gives rise to *dense* graphs where $m = \Omega(n^2)$. We can replace each edge in the graph by a path of length n. This yields a new graph which has $m(n-1)+n$ vertices. However, the total number of edges in the graph is mn. We can also distribute the weight of each edge evenly between the new edges. This new graph is now sparse, and a simple calculation shows that we obtain essentially

the same theorem for sparse graphs (the constant factor gets absorbed into the $\Omega(1)$ term in the exponent for the number of processors).

Theorem 4.6.1 The SHORTEST PATH PROBLEM cannot be computed in $o(\log N)$ steps on an unbounded fan-in PRAM without bit operations using $N^{\Omega(1)}$ processors, even if the input graph is sparse.

4.6.2 Lower Bounds for Matching Problems

The SHORTEST PATH PROBLEM is $\leq_m^{\mathcal{NC}^1}$-reducible to the WEIGHTED BIPARTITE GRAPH MATCHING PROBLEM [31, 33]. The size of the weights in the reduction is the same as the original size of the shortest path problem. Thus, we obtain a lower bound for the WEIGHTED BIPARTITE GRAPH MATCHING PROBLEM which is similar to the one obtained for the SHORTEST PATH PROBLEM.

Corollary 4.6.2 The WEIGHTED BIPARTITE GRAPH MATCHING PROBLEM cannot be solved in $o(\log N)$ steps on an unbounded fan-in PRAM without bit operations using $N^{\Omega(1)}$ processors.

4.6.3 Lower Bounds for Matroid Problems

A *matroid* is a combinatorial structure that can be viewed as a generalization of the theory of algebraic independence in vector spaces.

Definition 4.6.3 A *matroid* $M = (V, \mathcal{F})$ is a structure over a finite set V, and a family $\mathcal{F}$ of subsets of V, which satisfies the following axioms:

1. $\emptyset \in \mathcal{F}$,
2. if $A \in \mathcal{F}$, then all subsets of A are in $\mathcal{F}$,
3. if $A, B \in \mathcal{F}$, and $|B| > |A|$, then there is an $e \in B - A$, *s.t.* $A \cup \{e\} \in \mathcal{F}$.

The set V is called the *base set*, and the sets of the family $\mathcal{F}$ are referred to as *independent sets*. Note that if the elements of V were vectors in a vector space, then the set of all independent elements would satisfy the above axioms. Therefore, the matroid acts as a generalization of the notion of independence in vector spaces.

If we assign weights to the elements of the base set V, and define the weight of a set $A \in \mathcal{F}$ to be the sum of the weights of its constituent elements, a natural problem is to find an independent set of largest weight. A surprising property of matroids is that one can find a solution to this by choosing elements from the set V in a *"greedy"* fashion [31, 33].

Given two matroids M_1 and M_2 over the same weighted base set V, the MATROID INTERSECTION PROBLEM is to find the set of largest weight that is independent in both matroids.

A related problem is the MATROID PARITY PROBLEM. Here, we are given a matroid M as well as a family $\mathcal{P}$ of pairs of elements of the base set, *i.e.*, $\mathcal{P} \subseteq \binom{V}{2}$. The goal is to find an element $\mathcal{X} \in \mathcal{P}$ such that the set $\bigcup_{A \in \mathcal{X}} A$ is independent, and its weight is maximum. It would have been better to refer to the problem as the MATROID PAIRING PROBLEM but the inappropriate name is now firmly established in the literature.

Since the WEIGHTED MATCHING problem is $\leq_m^{\mathcal{NC}^1}$-reducible to both the MATROID INTERSECTION PROBLEM and the MATROID PARITY PROBLEM [31, 33], we obtain similar lower bounds for these problems.

Corollary 4.6.4 The MATROID INTERSECTION PROBLEM (resp. MATROID PARITY PROBLEM) cannot be solved in $o(\log N)$ steps on an unbounded fan-in PRAM without bit operations using $N^{\Omega(1)}$ processors.

4.6.4 Lower Bounds for Randomized Algorithms

Since the theorem of Mulmuley also applies to the randomized versions of the PRAM, all the lower bounds stated earlier are also true in the randomized setting (with two-sided error).

CHAPTER 5

THE MAXIMUM BLOCKING FLOW PROBLEM

5.1 Introduction

Network flow algorithms have a venerated history in the study of algorithms. They were among the first combinatorial problems for which fast algorithms were designed in the early days of computing. Once again, like the algorithms for computing shortest paths in graphs, they have substantial practical importance.

The key problem in this area is computing the maximum flow in an undirected network. The input to the problem is a weighted, undirected graph G and two designated vertices s and t (*source* and *sink* respectively). The edges can be thought of as pipes carrying water, and the weights as the capacities of the pipes. The objective is to compute the maximum amount of water that can be transported from the source s to the sink t.

5.1.1 Algorithms

The original algorithm proposed by Ford and Fulkerson for computing the MAX FLOW did not run in polynomial time. The first polynomial time algorithm was given by Edmonds and Karp [16]. In fact, they gave two algorithms, one of which runs in time $O(m \log |f^*|)$, in which f^* denotes the value of the optimal flow, and another that takes time $O(m^2 n)$,

The first major improvement in the running time of network flow algorithms was provided by Dinic [14] who discovered an algorithm that ran in time $O(mn^2)$. Dinic introduced the crucial notion of a blocking flow in the same paper, and used it to give a faster algorithm than the one given by Edmonds and Karp. Karzanov in 1974 [29], and later Malhotra, Pramodh-Kumar, and Maheshwari in

1978 [34] improved the time taken to compute a blocking flow from $O(mn)$ to $O(n^2)$ which improved the overall running time to $O(n^3)$.

A flow is said to be blocking if the elimination of saturated edges (edges in which the capacity has been reached) leaves the graph disconnected. Dinic showed that the MAX FLOW problem can be solved by solving a sequence of at most n blocking flow problems.

The MAX BLOCKING FLOW problem is the following: given a weighted, undirected graph G and two special vertices s and t, compute the value of the largest flow that *"blocks"*, *i.e.* every path between s and t has one edge that is *saturated*. Equivalently, we may say that the residual graph obtained by eliminating saturated edges, has no path between s and t. The problem is known to be equivalent to computing the maximum flow in a directed, acyclic graph.

Both the MAX FLOW and MAX BLOCKING FLOW problems are known to be $\mathcal{P}$-complete [39, 21]. Hence, under the assumption that $\mathcal{P} \neq \mathcal{NC}$, they cannot have fast parallel algorithms.

However, the design of parallel algorithms that are faster than known sequential algorithms has significant practical importance. The fastest parallel algorithm for computing a blocking flow in an acyclic network is due to Vishkin [47] who shows how to compute it in $O(n \log n)$ time using n processors.

5.1.2 Overview of the Chapter

The layout of the chapter is analogous to the layout of Chapter 4. The lower bound on the parametric complexity is stated in Section 5.3. The main technical lemma can be found in Section 5.3.2. The inductive construction of the graph is in Section 5.4, and the proofs for the theorems and lemmas are at the end of the chapter in Section 5.5.

A similar lower bound for the parametric complexity MAX FLOW problem was given by Carstensen [8] in a completely different context. Mulmuley [35] simplified the proof to get a strong lower bound on the MAX FLOW problem. Our lower bound for the MAX BLOCKING FLOW problem is based on the latter and follows its presentation rather closely.

5.2 The Main Result

Theorem 5.2.1 The MAX BLOCKING FLOW problem cannot be computed in time $o(n^{1/4})$ on an unbounded fan-in PRAM without bit operations using $2^{\Omega(n^{1/4})}$ processors, even if the bit-lengths of the capacities on the edges are restricted to be of size $O(n^2)$.

We can actually prove a stronger theorem that takes into account algorithms whose running time may depend on the total bit-length N.

Theorem 5.2.2 The MAX BLOCKING FLOW problem cannot be computed in time $o(N^{1/8})$ on an unbounded fan-in PRAM without bit operations using $2^{\Omega(N^{1/8})}$ processors.

5.3 Parametric Complexity

Theorem 5.3.1 The parametric complexity of the MAX BLOCKING FLOW problem on n vertices is $2^{\Omega(n)}$ for $\beta(n) = O(n^2)$.

5.3.1 Preliminaries

The computation of the maximum blocking flow is equivalent to computing the max-flow in a directed acyclic graph. All the graphs that we consider are DAGs.

An *s-t cut* in the graph is a set of edges such that removing these edges from the graph makes s and t disconnected, *i.e.*, there is no path from s to t in the resultant graph.

The presentation of the theorem can be simplified by appealing to the classic *"max-flow min-cut"* theorem [12] which states that the value of the max-flow in the graph is equal to the weight of the minimum s-t cut in the graph. It is convenient to use the word *capacity* when referring to the flow, but the term *weight* when we want to talk about the cut. In what follows, we use the two terms interchangeably.

5.3.2 The Technical Lemma

Theorem 5.3.2 For every $n \in \mathbb{N}$, there exists a weighted directed acyclic graph G_n whose capacities are linear functions in a parameter λ such that the following hold:

1. All the edge capacities are non-negative when λ is in some interval I.
2. The graph of the weight of the s-t min-cut as a function of a λ has $2^n - 1$ breakpoints in the interval I.
3. The coefficients of the weight functions have bit-lengths of size $O(n^2)$.

5.4 Construction

5.4.1 Construction of the Intervals

Fix $T = 2^{n+1}$. The interval I under consideration is $[-T, T]$. We define various sub-intervals of this interval I, which are in one-to-one correspondence with the nodes of a complete binary tree of depth n (Figure 5.1).

Consider a complete binary tree of depth n whose nodes are labeled with sub-intervals of $[-T, T]$. Associate the root node with I. For each of its two children, split the interval into two halves and associate the left half of the interval with the left child and the right half with the right child.

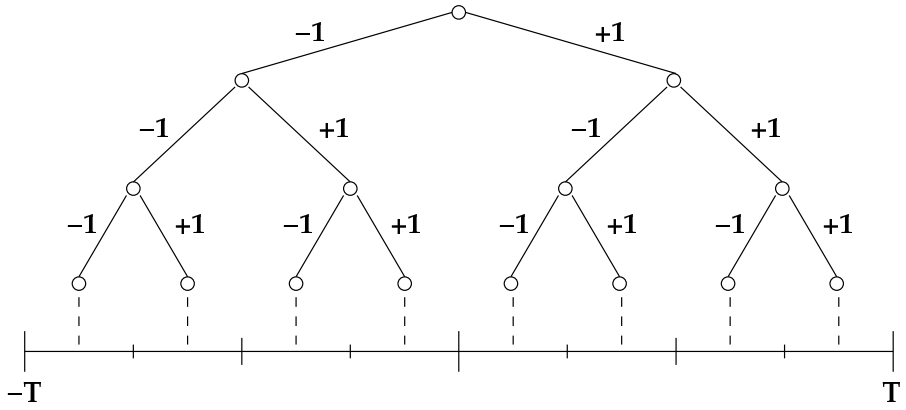

Figure 5.1: Tree of Intervals

Each node of the tree is also in one-to-one correspondence with a sequence of $+\mathbf{1}$'s and $-\mathbf{1}$'s. The root node corresponds to the empty string ε. Any such sequence σ of $+\mathbf{1}$'s and $-\mathbf{1}$'s of length i identifies a unique node at the i^{th} level in the tree, and hence, with a unique sub-interval of I. For any such sequence σ, let $\sigma(r)$ denote the r^{th} symbol of the sequence.

Definition 5.4.1 For any sequence σ, define $H(\sigma)$ to be the interval associated with the node corresponding to the sequence σ. Let $m(\sigma)$ be its midpoint.

Definition 5.4.2 For any sequence σ, define $I(\sigma)$ to be the interval created by removing intervals of length 1 from both ends of $H(\sigma)$.

Then, $m(\varepsilon) = 0$ and for $t_j = 2^{n-j+1}$, we have that

$$m(\sigma) \quad = \quad \sum_{j \le i} \sigma(j)\, t_j. \tag{5.1}$$

5.4.2 Construction of the Graph

The directed, acyclic graph G_n has $2n+2$ vertices as shown in Figure 5.2. Besides the designated vertices s and t (source and sink respectively), there are $2n$ other vertices labeled as $1, 2, \ldots, n$ and $\bar{1}, \bar{2}, \ldots, \bar{n}$ respectively.

There are edges connecting s to all of the vertices i and $\bar{\imath}$ ($1 \le i \le n$) and edges from all the vertices i and $\bar{\imath}$ ($1 \le i \le n$) to t. In addition, each i and $\bar{\imath}$ is connected by an edge to j and $\bar{\jmath}$ provided that $j > i$.

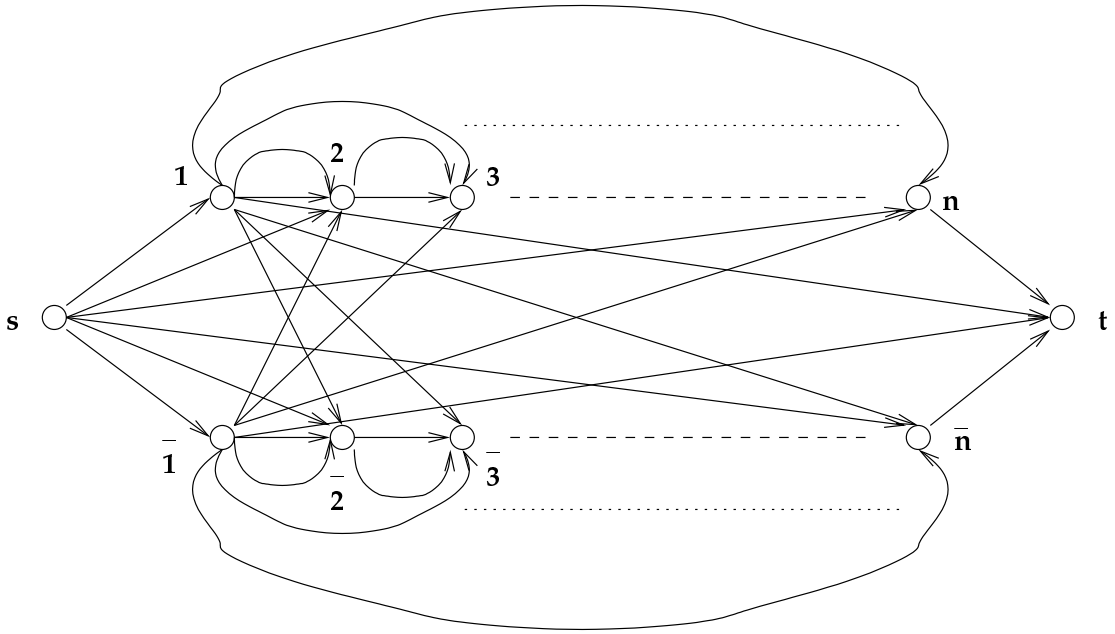

Figure 5.2: Construction of the Graph G_n

5.4.3 Construction of the Weight Functions

Define $w_n = 1$ and for $i < n$, define:

$$w_i > T \sum_{i<j\leq n} w_j. \tag{5.2}$$

The choice $w_i = 2 \cdot (T+1)^{n-i}$ suffices to satisfy the above equation. Recall that $t_j = 2^{n-j+1}$. Define the edge weights as follows:

$$\begin{aligned}
w(s,i) = w(s,\bar{i}) &= w_i T && 1 \leq i \leq n \\
w(i,t) &= w_i(T+\lambda) && 1 \leq i \leq n \\
w(\bar{i},t) &= w_i(T-\lambda) && 1 \leq i \leq n \\
w(i,j) = w(\bar{i},\bar{j}) &= \frac{w_j}{2}(T+t_i) && 1 \leq i < j \leq n \\
w(i,\bar{j}) = w(\bar{i},j) &= \frac{w_j}{2}(T-t_i) && 1 \leq i < j \leq n.
\end{aligned}$$

All the edge weights are positive in the interval I, and have bit-lengths $O(n^2)$.

5.5 Proofs

5.5.1 Proof of the Technical Lemma

Before we prove the main theorem, we need to prove a lemma which gives us the structure of the s-t min-cuts in G_n in the various intervals $I(\sigma)$.

Lemma 5.5.1 For any $i \leq n$, and a sequence σ of length i, fix $\lambda \in I(\sigma)$. Let $(U(\lambda), V(\lambda))$ be an s-t min-cut in G_n for this λ. Then for all $j \leq i$,

1. $U(\lambda)$ contains j iff $\sigma(j) = -\mathbf{1}$,
2. $U(\lambda)$ contains $\bar{j}$ iff $\sigma(j) = +\mathbf{1}$.

PROOF: Since the weighted graph G_n remains unchanged when each vertex i is exchanged with $\bar{\imath}$ and all terms with λ are replaced by $-\lambda$, it suffices to prove only the first statement. The proof is by induction on i.

BASE CASE: $i = 1$.

Let $\sigma(1) = +\mathbf{1}$. Suppose that $U(\lambda)$ contains the vertex 1. If we move the vertex 1 from $U(\lambda)$ to $V(\lambda)$ the value of the cut would decrease by:

$$\begin{aligned}
\Delta &= w(1,t) - w(s,1) \\
&\quad + \left\{ \sum_{\substack{j>1 \\ j \in V(\lambda)}} w(1,j) + \sum_{\substack{j>1 \\ \bar{j} \in V(\lambda)}} w(1,\bar{j}) \right\} + \left\{ \sum_{\substack{j>1 \\ j \in U(\lambda)}} w(1,j) + \sum_{\substack{j>1 \\ \bar{j} \in U(\lambda)}} w(1,\bar{j}) \right\} \\
&= w_1(T+\lambda) - w_1 T + \frac{1}{2} \sum_{j>1} (w_j(T+t_1) + w_j(T-t_1)) \\
&= w_1 \lambda + T \sum_{j>1} w_j.
\end{aligned}$$

By choice, the second term is smaller than w_1 and hence in the interval $I(+\mathbf{1}) = [1, T-1]$ the decrease in the min-cut is positive which contradicts the face that (U, V) is the min-cut. Therefore $U(\lambda)$ cannot contain the vertex 1.

Similarly, if $\sigma(1) = -\mathbf{1}$ and $V(\lambda)$ contained the vertex 1, then moving it from $V(\lambda)$ to $U(\lambda)$ would decrease the weight of the cut by

$$-w_1\lambda - T\sum_{j>1} w_j$$

which is positive if $\lambda \in I(-\mathbf{1}) = [-(T-1), -1]$ which contradicts the fact that $(U(\lambda), V(\lambda))$ was a min-cut. This proves that $U(\lambda)$ contains 1 iff $\sigma(1) = +\mathbf{1}$.

INDUCTIVE CASE: Fix a value of i where $i \leq n$.

Let $\hat{\sigma}$ be the string obtained by removing the last symbol from σ. We must have that $I(\sigma) \subset I(\hat{\sigma})$. Hence, by the inductive hypothesis, it follows that for all $j < i$, for all $\lambda \in I(\sigma)$, $U(\lambda)$ contains the vertex j iff $\sigma(j) = -\mathbf{1}$ and the vertex $\bar{j}$ iff $\sigma(j) = +\mathbf{1}$. We need to prove that $U(\lambda)$ contains the vertex i iff $\sigma(i) = -\mathbf{1}$.

Consider the case when $\sigma(i) = +\mathbf{1}$. Suppose to the contrary that $U(\lambda)$ contains the vertex i. By moving the vertex from $U(\lambda)$ to $V(\lambda)$, the weight of the cut would decrease by:

$$\begin{aligned}
\Delta &= w(i,t) - w(s,i) \\
&\quad + \left\{ \sum_{\substack{j>i \\ j\in U(\lambda)}} w(i,j) + \sum_{\substack{j>i \\ j\in V(\lambda)}} w(i,j) + \sum_{\substack{j>i \\ \bar{j}\in U(\lambda)}} w(i,\bar{j}) + \sum_{\substack{j>i \\ \bar{j}\in V(\lambda)}} w(i,\bar{j}) \right\} \\
&\quad + \left\{ \sum_{\substack{j<i \\ j\in U(\lambda)}} w(j,i) - \sum_{\substack{j<i \\ j\in V(\lambda)}} w(j,i) + \sum_{\substack{j<i \\ \bar{j}\in U(\lambda)}} w(\bar{j},i) - \sum_{\substack{j<i \\ \bar{j}\in V(\lambda)}} w(\bar{j},i) \right\} \\
&= w_i\lambda + \left\{ \sum_{j>i} w(i,j) + \sum_{j>i} w(i,\bar{j}) \right\} - \left\{ \sum_{j<i} \sigma(j) w(j,i) - \sum_{j<i} \sigma(j) w(\bar{j},i) \right\} \\
&= w_i\lambda - w_i \sum_{j<i} \sigma(j)\, t_j + T\sum_{j>i} w_j \\
&= w_i(\lambda - m(\hat{\sigma})) + T\sum_{j>i} w_j. \qquad \text{by (5.1)}
\end{aligned}$$

If $\lambda \in I(\sigma)$ and $\sigma(i) = +\mathbf{1}$ then $\lambda - m(\hat{\sigma}) \geq 1$. We know, by equation (5.2), that the absolute value of the last term is smaller than w_i. Hence, this decrease

is positive. Thus, moving the vertex i from $U(\lambda)$ to $V(\lambda)$ would strictly decrease the value of the cut which contradicts the fact that $(U(\lambda), V(\lambda))$ was a min-cut. Therefore, $U(\lambda)$ cannot contain i if $\sigma(i) = +\mathbf{1}$.

Similarly, if $\sigma(i) = -\mathbf{1}$ and $V(\lambda)$ contained the vertex i, then moving it to $V(\lambda)$ to $U(\lambda)$ would decrease the weight of the cut by

$$-w_i(\lambda - m(\hat{\sigma})) - T\sum_{j>i} w_j,$$

which is positive if $\lambda \in I(\sigma)$ which contradicts the fact that $(U(\lambda), V(\lambda))$ was a min-cut. This proves that $U(\lambda)$ contains i iff $\sigma(i) = +\mathbf{1}$.

□

5.5.2 *Proof of the Main Theorem*

We now continue the proof that the s-t min-cut cost function for G_n has at least $2^n - 1$ breakpoints in the interval $[-T, T]$.

PROOF: Let σ be any sequence of $+\mathbf{1}$'s and $-\mathbf{1}$'s of length n. Then, from the Lemma 5.5.1, we know that in each interval $I(\sigma)$, the min-cut remains the same. We also know that $U(\lambda)$ contains the vertex i iff $\sigma(i) = -\mathbf{1}$ and $\bar{\imath}$ iff $\sigma(i) = +\mathbf{1}$.

Hence, the weight of the cut at any $\lambda \in I(\sigma)$ is of the form $P(\sigma)\lambda + Q(\sigma)$ where $P(\sigma) = -\sum_i \sigma(i)\, w_i$. Thus, the slopes of the weights in distinct 2^n intervals are all different and hence, we must have $2^n - 1$ breakpoints for the graph of G_n as λ ranges over the interval $[-T, T]$.

□

5.6 Corollaries

5.6.1 Maximum Blocking Flow in Sparse Graphs

We can apply the same technique used in the SHORTEST PATH PROBLEM to convert the explicit family of dense graphs into an explicit family of sparse graphs. We place $n-1$ extra vertices on each edge of the graph, and divide the weight of the edge evenly among the new edges. This yields a graph with $m(n-1)+n$ vertices and mn edges. Thus, the new graphs G_n has n^3 vertices and $\Theta(n^3)$ edges. This yields a slightly weaker lower bound for sparse graphs which is stated below.

Theorem 5.6.1 The MAX BLOCKING FLOW problem cannot be computed in time $o(n^{1/12})$ using $2^{\Omega(n^{1/12})}$ processors, even if the bit-lengths of the capacities on the edges are restricted to be of size $O(n^2)$, and the input graph is restricted to be sparse.

Theorem 5.6.2 The MAX BLOCKING FLOW problem cannot be computed in time $o(N^{1/24})$ on an unbounded fan-in PRAM without bit operations using $2^{\Omega(N^{1/24})}$ processors, even if the input graph is sparse.

5.6.2 Lower Bounds for Randomized Algorithms

Both the lower bounds also hold for the randomized case. In both cases, the constants in the exponent are slightly smaller than the ones stated in the theorems above.

CHAPTER 6

CONCLUSION

The initial impetus for this thesis came from trying to determine the complexity of the WEIGHTED GRAPH MATCHING PROBLEM. The problem is known to be in $\mathcal{P}$ [33]. However, it has eluded all attempts at efficient parallelization, and is not even known to be $\mathcal{P}$-complete (which assuming $\mathcal{P} \neq \mathcal{NC}$, as is widely believed, would give a plausible reason not to expect a fast parallel algorithm).

Since the SHORTEST PATH PROBLEM $\leq^{\mathcal{NC}^1}$-reduces to the WEIGHTED BIPARTITE GRAPH MATCHING PROBLEM, we obtain a lower bound for the latter problem. However, this lower bound is very weak, and unfortunately, the original tantalizing question still remains open.

We conjecture that it should be possible to obtain super-polylogarithmic lower bounds using the same technique for the problem of computing WEIGHTED MATCHING in general graphs. Other such problems that have resisted parallelization and are known to be harder are the MATROID INTERSECTION PROBLEM and the MATROID PARITY PROBLEM. It would also be interesting to give similar lower bounds for these problems that are not known to be in $\mathcal{NC}$, nor known to be $\mathcal{P}$-complete.

In this work, we also give a lower bound for the complexity of the MAX BLOCKING FLOW problem. However, this technique does not extend to giving lower bounds on computing any blocking flow (other than the maximum one). Additionally, the technique does not work if we are only interested in solving the problem approximately; *i.e.*, we would like to compute the value of the optimum up to a small *multiplicative* factor. It should be noted that the technique of Mulmuley does extend to computing the optimum value approximately up to a small *additive* factor. The question of developing techniques for lower bounds for approximation problems is still wide open.

Mulmuley's technique does not carry over to providing lower bounds for sequential algorithms because the degrees of the polynomials are exponential in the running time. Thus, the problem of finding techniques that will allow us to prove strong lower bounds for sequential algorithms is also open.

REFERENCES

[1] Romas Aleliunas, Richard M. Karp, Richard J. Lipton, Laszlo Lovász, and Charles Rackoff. Random walks, universal traversal sequences, and the complexity of maze problems. In *Proceedings of the 20th Annual IEEE Symposium on Foundations of Computer Science*, pages 218–223, San Juan, Puerto Rico, 29–31 October 1979. IEEE.

[2] Noga Alon and Ravi Boppana. The monotone circuit complexity of boolean functions. *Combinatorica*, 7(1):1–22, 1987.

[3] Amir M. Ben-Amram and Zvi Galil. Topological lower bounds on algebraic random access machines. *SIAM Journal on Computing*. to appear.

[4] Amir M. Ben-Amram and Zvi Galil. Lower bounds on algebraic random access machines (extended abstract). In *Proceedings of the 22nd International Colloquium on Automata, Languages and Programming*, pages 360–371, 1995.

[5] Michael Ben-Or. Lower bounds for algebraic computation trees. In *Proceedings of the 15th Annual ACM Symposium on the Theory of Computing*, pages 80–86, 1983.

[6] Michael Ben-Or, Ephraim Feig, Dexter Kozen, and Prasoon Tiwari. A fast parallel algorithm for determining all roots of a polynomial with real roots. In *Proceedings of the 18th Annual ACM Symposium on the Theory of Computing*, pages 340–349, Berkeley, California, 28–30 May 1986.

[7] Patricia Carstensen. Complexity of some parametric integer and network programming problems. In *Mathematical Programming*, volume 26, pages 64–75, 1983.

[8] Patricia Carstensen. *The Complexity of Some Problems in Parametric Linear and Combinatorial Programming*. PhD thesis, Univ. of Michigan, 1983.

[9] Patricia Carstensen. Parametric cost shortest chain problem. published in ORSA/TIMS, 1987.

[10] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press, 6th edition, 1992.

[11] L. Csanky. Fast parallel matrix inversion algorithms. *SIAM Journal on Computing*, 5(4):618–623, December 1976.

[12] G B Dantzig and D R Fulkerson. On the max-flow min-cut theorem of networks. In H W Kuhn and A W Tucker, editors, *Linear Inequalities and Related Systems*, Annals of Mathematics Study, vol. 38, pages 215–221, Princeton Univ. Press, Princeton, NJ, 1956.

[13] Edsger W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[14] E. A. Dinic. Algorithm for solution of a problem of maximum flow in networks with power estimation. *Soviet Mathematics Doklady*, 11:1277–1280, 1970.

[15] Jack Edmonds. Paths, trees, and flowers. *Canadian Journal of Mathematics*, 17:449–467, 1965.

[16] Jack Edmonds and Richard M. Karp. Theoretical improvements in the algorithmic efficiency for network flow problems. *Journal of the ACM*, 19:248–264, 1972.

[17] Robert W. Floyd. Algorithm 97 (SHORTEST PATH). *Communications of the ACM*, 5(6):345, 1962.

[18] L. R. Ford and D. R. Fulkerson. *Flows in Networks*. Princeton Univ. Press, Princeton, NJ, 1973.

[19] A. Frank and Éva Tardos. An application of simultaneous Diophantine approximation in combinatorial optimization. *Combinatorica*, 7:49–65, 1987.

[20] Merrick Furst, James B. Saxe, and Michael Sipser. Parity, circuits, and the polynomial-time hierarchy. *Mathematical Systems Theory*, 17(1):13–27, April 1984.

[21] Michael R. Garey and David S. Johnson. *Computers and Intractability : A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, 1979.

[22] Andrew V. Goldberg and Robert E. Tarjan. A new approach to the maximum flow problem. In *Proceedings of the 18th Annual ACM Symposium on the Theory of Computing*, pages 136–146, Berkeley, California, 28–30 May 1986.

[23] Andrew V. Goldberg and Robert E. Tarjan. A parallel algorithm for finding a blocking flow in an acyclic network. *Information Processing Letters*, 31(5):265–271, June 1989.

[24] Raymond Greenlaw, H. James Hoover, and Walter L. Ruzzo. *Limits to Parallel Computation : P-Completeness Theory*. Oxford Univ. Press, 1995.

[25] Martin Grötschel, László Lovász, and Alexander Schrijver. *Geometric Algorithms and Combinatorial Optimization*. Springer, Berlin, 1988.

[26] Johan Håstad. Almost optimal lower bounds for small depth circuits. In *Proceedings of the 18th Annual ACM Symposium on the Theory of Computing*, pages 6–20, Berkeley, California, 28–30 May 1986.

[27] David B. Johnson. Efficient algorithms for shortest paths in sparse networks. *Journal of the ACM*, 24(1):1–13, January 1977.

[28] David Karger and Rajeev Motwani. An NC algorithm for minimum cuts. *SIAM Journal on Computing*, 26, 1997.

[29] A. V. Karzanov. Determining the maximal flow through a network by the method of preflows. *Soviet Mathematics Doklady*, 15:434–437, 1974.

[30] Dexter L. Kozen. *The Design and Analysis of Algorithms*. Springer, Berlin, 1992.

[31] Eugene L. Lawler. *Combinatorial Optimization: Networks and matroids*. Holt, Rinehart and Winston, New York, 1991.

[32] F. Thomson Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*. Morgan Kaufmann, San Mateo, 1992.

[33] László Lovász and M. D. Plummer. *Matching Theory*, volume 29 of *Annals of Discrete Mathematics*. North-Holland Math. Studies, 1986.

[34] V. M. Malhotra, M. Pramodh Kumar, and S. N. Maheshwari. An $O(|V|^3)$ algorithm for finding maximum flows in networks. *Information Processing Letters*, 7:277–278, 1978.

[35] Ketan Mulmuley. Lower bounds in a parallel model without bit operations. *SIAM Journal on Computing*, 28(4):1460–1509, August 1999.

[36] Ketan Mulmuley, Umesh V. Vazirani, and Vijay V. Vazirani. Matching is as easy as matrix inversion. *Combinatorica*, 7:105–113, 1987.

[37] C. Andrew Neff. Specified precision polynomial root isolation is in NC. *Journal of Computer and System Sciences*, 48(3):429–463, June 1994.

[38] Franco P. Preparata and Michael Ian Shamos. *Computational Geometry: An Introduction*. Texts and Monographs in Computer Science. Springer, Berlin, Germany, 1985.

[39] Vijaya Ramachandran. The complexity of minimum cut and maximum flow problems in an acyclic network. *Networks*, 17:387–392, 1987.

[40] Alexander Razborov. Lower bounds on the complexity of some boolean functions. *Dokl. Ak. Nauk.*, 1985.

[41] John H. Reif. *Synthesis of Parallel Algorithms*. Morgan Kaufmann, San Mateo, 1993.

[42] Walter Savitch. Relationships between nondeterministic and deterministic tape complexities. *Journal of Computer and System Sciences*, 4:177–192, 1970.

[43] Yossi Shiloach and Uzi Vishkin. An $O(n^2 \log n)$ parallel MAX-FLOW algorithm. *Journal of Algorithms*, 3(2):128–146, June 1982.

[44] Daniel Sleator and Robert E. Tarjan. An $O(nm \log n)$ algorithm for maximum network flow. Technical Report STAN-CS-80-831, Department of Computer Science , Stanford University, Stanford, CA, 1980.

[45] Éva Tardos. The gap between monotone and non-monotone circuit complexity is exponential. *Combinatorica*, 8, 1988.

[46] Robert E. Tarjan. *Data structures and network algorithms*, volume 44 of *CBMS-NSF Reg. Conf. Ser. Appl. Math.* SIAM, 1983.

[47] Uzi Vishkin. A parallel blocking flow algorithm for acyclic networks. *Journal of Algorithms*, 13(3):489–501, September 1992.

[48] Stephen Warshall. A theorem on Boolean matrices. *Journal of the ACM*, 9(1):11–12, January 1962.

[49] Andrew Yao. Lower bounds for algebraic computation trees with integer inputs. In *Proceedings of the 30th Annual IEEE Symposium on Foundations of Computer Science*, pages 222–227, 1977.

[50] Andrew Yao. Separating the polynomial-time hierarchy by oracles. In *Proceedings of the 26th Annual IEEE Symposium on Foundations of Computer Science*, pages 1–10, Los Angeles, Ca., USA, October 1985. IEEE Computer Society Press.

INDEX