Arithmetic Circuit Complexity and Motion Planning

Stephen R. Tate

Dissertation successfully defended March 26, 1991 to the committee:

John Reif
Donald Loveland
Donald Rose
Robert Wagner
Joseph Kitchen

Dissertation submitted in partial fulfillment of the
requirements for the degree of Doctor of Philosophy
in the Department of Computer Science
in the Graduate School of
Duke University

Abstract

This dissertation presents the results of my research in two areas: parallel algorithms/circuit complexity, and algorithmic motion planning. The chapters on circuit complexity examine the parallel complexity of several fundamental problems (such as integer division) in the model of small depth circuits. In the later chapters on motion planning, we turn to the computationally intensive problem of planning efficient trajectories for robots in both cooperative and non-cooperative environments.

Specifically, we first examine the complexity of integer division with remainder under the standard model of constant fanin boolean circuits. We restrict our attention to circuits which are logspace uniform, and present a novel algorithm that has better asymptotic complexity bounds than any previously known algorithm. In fact, it matches the best previously known depth bound and the best previously known size bound *simultaneously*.

Next, we examine circuits where each gate has arbitrary fanin, and can compute the MAJORITY function. Interestingly, while it is impossible to compute integer division with constant depth, unbounded fanin AND/OR circuits, we show that it is possible to compute it with only $O(n^{1+\epsilon})$ gates (for any constant $\epsilon > 0$) and constant depth when MAJORITY gates are allowed. Unfortunately, to get a constant depth circuit, we allow the circuit to be only P-uniform (rather than logspace uniform).

In the chapters on motion planning, we first give an approximation bound for optimal time motion planning, where the robot is given bounds on the L_2 norms of velocity and acceleration. This (and concurrent, independent work by Donald and Xavier) was the first such approximation algorithm for robots with dynamics bounded in the L_2 norm.

The second chapter on motion planning addresses the following problem: what if a second, non-cooperating (or even adversarial) robot is added to the environment. This problem is referred to as a pursuit game, and we must make a plan that avoids collisions with the second robot. We present both an exponential time lower bound and several polynomial time approximation algorithms for this problem. The lower bound is the first truly intractable lower bound for a robotics problem with perfect information. Despite this lower bound, we present a polynomial time algorithm that gives approximately optimal solutions to an important class of pursuit games — namely, those where it is possible for our robot to keep a certain "safety margin" between it and its adversary.

Contents

1 Introduction **1**

1.1 Thesis Overview . . . 1
1.2 The Study of Algorithms . . . 2
1.3 Circuit Complexity . . . 3
1.3.1 Basic Definitions . . . 3
1.3.2 Circuit Uniformity . . . 4
1.3.3 Division with Bounded Fanin Boolean Circuits . . . 5
1.3.4 Threshold Circuit Results . . . 5
1.4 Algorithmic Motion Planning . . . 6
1.4.1 Geometric Planning Problems . . . 6
1.4.2 Dynamics Constraints . . . 6
1.4.3 Approximation Algorithms . . . 7
1.4.4 Planning in a Cooperative Environment . . . 7
1.4.5 Planning in a Hostile Environment . . . 8

2 Newton Iteration and Division **11**

2.1 Introduction . . . 11
2.2 Newton Approximation . . . 12
2.3 Integer Powering . . . 16
2.3.1 Bit Reduction . . . 17
2.3.2 Power Reduction . . . 21
2.3.3 Putting the Pieces Together . . . 23
2.4 High Order Convergence with Newton Approximation . . . 25
2.5 An Efficient Parallel Reciprocal Circuit . . . 28
2.6 Chapter Summary . . . 31

3 Threshold Circuits **33**

3.1 Introduction . . . 33
3.2 Computing Arithmetic Using Threshold Circuits . . . 34
3.2.1 Logspace Uniform Circuits . . . 34
3.2.2 P-uniform Circuits . . . 37
3.3 Relation to Finite Field Circuits . . . 39
3.3.1 Simulating Finite Field Circuits . . . 40
3.3.2 Simulating Threshold Circuits . . . 41
3.3.3 Combined Results . . . 41
3.4 Chapter Summary . . . 42

4 Motion Planning in Cooperative Environments **43**
4.1 Introduction 43
4.2 Preliminaries 44
4.2.1 Definitions and Terminology 44
4.2.2 Outline of Algorithm and Proof 45
4.3 Constructing a Grid 46
4.4 Tracking in the Absence of Obstacles 49
4.5 Tracking with Obstacles 57
4.6 Chapter Summary 59

5 Motion Planning in Hostile Environments **61**
5.1 Introduction 61
5.2 Lower Bounds 62
5.2.1 Basic Geometry and the Encoding of a Configuration 62
5.2.2 Basic Form of the Proof 65
5.2.3 Traps 65
5.2.4 ATM Transitions 70
5.3 Approximation Algorithms 71
5.3.1 Bounded L_∞-norm velocity 72
5.3.2 Bounded L_∞-norm velocity and acceleration 74
5.3.3 Bounded L_2-norm velocity 75
5.3.4 Bounded L_2-norm velocity and acceleration 76
5.4 The Point-Robot Pursuit Game 76
5.5 Open Problems 77

6 Conclusion **79**

Chapter 1

Introduction

1.1 Thesis Overview

The ultimate goal of research in theory of computation is to fully understand why some functions are easy to compute, while others require considerable computing resources. Unfortunately, the present state of knowledge falls far short of this goal — there are very few functions for which we know exact complexity measures (i.e., matching upper and lower bounds).

This dissertation represents two areas of research interest, both falling under the general area of theoretical computer science; specifically, we look at problems in parallel algorithms (or circuit complexity) and problems from computational robotics. These areas are very active fields of current research, with great possibilities for future work. These two areas contain problems from two extremes of complexity— both very easy (computationally) and very difficult problems. A common thread between these areas is the algebraic nature of the problems. In the chapters on circuit complexity, finite algebras are repeatedly used as tools for developing very efficient algorithms. Computational robotics problems have a strong algebraic structure, and in fact, many problems in this area are solved by reduction to the first-order theory of the reals.

In the first part of this work, we examine several easily computable functions under different models of computation. Specifically, in chapter 2 we look at the parallel circuit complexity of integer division using boolean circuits (chapter 2). Integer division is a fundamental arithmetic problem, and has been studied with respect to both sequential and parallel models of computation. The best-known sequential algorithm for integer division was given in Cook's Ph.D. thesis [19], but this algorithm does not parallelize well. In chapter 2, we use a novel technique to give the best known parallel algorithm for integer division. In fact, this algorithm has *optimal* size, since the size complexity is asymptotically the same as for integer multiplication.

In chapter 3, we change the model to threshold circuits, a powerful model of computation that captures many features of neural nets. Under this model, division can be computed by constant-depth circuits, and the algorithm presented in chapter 3 significantly reduces the size required by previously-known constant-depth division circuits. In addition, we prove several results that relate the power of threshold circuits to the power of circuits over finite fields; these results give a strong algebraic structure to the class of functions computable by constant-depth threshold circuits.

The second half of this dissertation is concerned with problems from the emerging field of computational robotics. Clearly, as the use of robots becomes more widespread, it is vital to examine the associated computational problems. We examine the problem of planning a

trajectory for a robot moving through either a cooperating environment (chapter 4) or a hostile environment (chapter 5). Simple path-planning has been extensively studied, and polynomial-time algorithms are known [61]; however, these algorithms ignore such real-world issues as the fact that the amount of force used to move the robot is bounded, so the computed path may take an unacceptably long time for a real robot to traverse. In this dissertation, we include these real-world issues in our model; the increased realism of the model is unfortunately accompanied by a large increase in complexity. In addition, we consider further restrictions on the trajectory, such as finding a time-optimal trajectory. These planning problems are quite computationally intensive; in fact, one result in chapter 5 proves that motion planning in a hostile environment is EXPTIME-hard. In light of the lower bounds, we must consider approximation algorithms in order to obtain any practical results. In chapters 4 and 5 we show that approximation algorithms are indeed the answer to these planning problems, and we give polynomial time approximation algorithms for both types of planning problems. Our algorithms are the first polynomial-time algorithms to solve the planning problems giving solutions that are provably close to the optimal solution.

1.2 The Study of Algorithms

Inside the broad area of theory of computation, there are many subfields. The two most prevalent subfields are the design and analysis of algorithms and computational complexity theory. The study of algorithms began far before the invention of the computer. For many centuries, the notion of an algorithm has been used by mathematicians in the idea of a constructive proof. The first known reference to the study of algorithms is Euclid's work on calculating greatest common divisors (this was done in approximately 300 B.C.). Of course, Euclid was trying to reduce the amount of pencil-and-paper work required to compute GCD's, since the notion of a digital computer was quite inconceivable. It is interesting that approximately 2300 years later, Euclid's algorithm is the most efficient and widely used algorithm for computing GCD's on modern computers. It should be noted here that Euclid's algorithm simply performs repeated integer division, which is the main topic of chapters 2 and 3.

The study of algorithms is clearly the most fundamental area of computer science, since *everything* running on a computer, whether in the operating system, an artificial intelligence program, or a numerical analysis program, is an algorithm of some type. Algorithm designers look for ways of mapping problem statements to a set of instructions that a computer can understand and execute efficiently.

The analysis of algorithms is a skill that any computer scientist must have in order to function effectively. The goal of algorithm analysis is to predict the performance of an algorithm (or program) on large inputs. Practically every programmer has at some time written a program that, while working fine on small-sized sample inputs, slows down unacceptably when full production-sized inputs are supplied. If the algorithms are correctly examined first, this behavior can be predicted, and the programmer should not experience the surprise that usually accompanies such a slow-down.

The area of complexity theory is concerned with the fundamental problem: what makes some problems difficult to compute. In pursuing this question, many important concepts have been introduced to computer science, such as the concepts of NP-complete problems, alternation, and circuit complexity. In general, many people like to view the study of algorithms as producing upper bounds (efficient algorithms) and complexity theory as producing lower bounds (proofs that certain problems are actually very difficult to compute). Of course, there is a lot of overlap

between the two areas, and such a distinction is not always clear.

The remainder of this introduction gives background for the problems considered, basic definitions, and a summary of the results that are proved in the main text. It is assumed that the reader is familiar with the basic notation and terminology used in complexity theory and the analysis of algorithms (such as the complexity classes P, NP, PSPACE, and EXPTIME, as well as the asymptotic notations $O(f(n))$, $o(f(n))$, $\Omega(f(n))$, and $\Theta(f(n))$. Definitions for these terms can be found in the widely-used textbooks of theoretical computer science [2, 5, 21, 33]. It is noted here that $\log n$ represents the base 2 logarithm of n (i.e., $\log_2 n$), and $\log\log n$ represents a double logarithm (i.e., $\log(\log n)$).

1.3 Circuit Complexity

The most natural way to examine the inherent parallelism of an algorithm is to look at a graph of the data movement of the algorithm (this is referred to as the "data flow graph"). The circuit model of computation is a natural extension of the data flow graph that can be viewed as a model of computation. When the nodes of the graph are restricted to computing very basic functions (such as the boolean functions AND and OR), circuits are a very useful model for examining the parallel complexity of easy to compute functions such as the parity function of n input bits, or the basic arithmetic functions.

The reason for examining easy to compute functions is twofold. First, the problems examined (such as the basic arithmetic functions) are of a very fundamental nature, and are used repeatedly in more complex calculations. Thus, it is important to understand completely the complexity of these fundamental problems, and the most efficient ways of computing them. Secondly, the analysis of easy functions adds important new tools to the techniques available in proving lower bounds for more complex functions. In fact, many results in circuit complexity have also given results related to the polynomial time hierarchy, and vice-versa. This relationship with the polynomial time hierarchy was a primary reason for the initial work on lower bounds for boolean circuits computing parity [29], and led to a strong result about the simulation of boolean circuits by threshold circuits [3].

1.3.1 Basic Definitions

A *circuit* is a very general model of computation over any domain. Fix a value domain Σ. A *function basis* is a set F of functions that includes a set of k-adic functions $\{f : \Sigma^k \to \Sigma\}$, for each $k \geq 1$. A *circuit* C_n over function basis F is an oriented, acyclic digraph with a list of *input nodes* $v_1, ..., v_n$ each with fanin[1] 0, a list of *output nodes* $u_1, ..., u_m$, and a k-adic function in F labeling each noninput node with fanin $k \geq 1$. Given an input string $(x_1, ..., x_n) \in \Sigma^n$, we assign each input node v_i a value $val(v_i) = x_i$, for $i = 1, ..., n$. For each non-input node w with k predecessors $w_1, ..., w_k$, we recursively assign w a value $val(w) = f(val(w_1), ..., val(w_k)) \in \Sigma$, where $f \in F$ is the function that labels node w. C_n finally outputs the string $(val(u_1), ..., val(u_m)) \in \Sigma^m$ (where the output length m is fixed for the circuit C_n). Thus C_n computes a function from Σ^n to Σ^m.

The *size* of circuit C_n is the number of edges of the circuit. The *depth* of circuit C_n is the length of the longest path from any input node to an output node. A *circuit family* is an infinite list of circuits $\mathbf{C} = (C_1, C_2, ..., C_n, ...)$ where C_n has n inputs. As described above, $\mathbf{C}$ computes a family of functions $(f_1, f_2, ..., f_n, ...)$, where f_n is the function of n inputs computed by circuit

[1] The fanin of a node is also referred to as the indegree of the node.

C_n. Circuit family $\mathbf{C}$ is said to have *size complexity* $S(n)$ and simultaneous *depth complexity* $D(n)$ if, for all $n \geq 0$, circuit C_n has size $\leq S(n)$ and depth $\leq D(n)$.

The most common type of circuit uses the boolean value domain $\Sigma = \{0, 1\}$, and a function basis consisting of the basic boolean functions AND, OR, and NOT (this includes k-adic AND and OR for all $k \geq 2$). If these circuits are restricted to those that have a constant bound on the indegree of each node, then the resulting circuits are called *bounded fanin boolean circuits*. This type of circuit is the focus of chapter 2. It is easy to show that the exact bound on fanin does not affect the asymptotic complexity bounds obtained [46], so we assume, without loss of generality, a bound of 2.

Another common type of circuit over the boolean domain is the class of *threshold circuits*. A threshold function Th_k^n is defined by

$$Th_k^n(x_1, ..., x_n) = \begin{cases} 1 & \text{if } \sum_{i=1}^{n} x_i \geq k \\ 0 & \text{otherwise} \end{cases}$$

The boolean negation of the threshold function Th_k^n is denoted by $Th_{<k}^n$. Circuits using the set of all threshold functions and their negations as the function basis are called threshold circuits. Threshold circuits are the model of computation used in chapter 3.

The following complexity classes have been defined in the literature, with respect to the circuit model.

AC^i The class of functions computable by unbounded fanin boolean circuit families with polynomial size and $O(\log^i n)$ depth.

NC^i The class of functions computable by *bounded* fanin boolean circuit families with polynomial size and $O(\log^i n)$ depth.

TC^i The class of functions computable by threshold circuit families with polynomial size and $O(\log^i n)$ depth.

1.3.2 Circuit Uniformity

The circuit definition of the preceding section has a striking difference with other models of computation, such as the Turing machine, RAM, and PRAM. Specifically, since there is a separate circuit for each input length, by including a lookup table in each circuit of the family, there is a circuit family for *every possible function* — even for uncomputable functions! This is the same interesting concept that arises in Turing machines with advice (see, for example, [5], [38]). Furthermore, it would be nice for the two main models of parallel computation (PRAMs and circuits) to give similar complexity results.

The solution is to consider the Turing machine complexity of computing a description of the circuit family.

Definition 1.3.1 A circuit family $\mathbf{C} = \{C_1, C_2, ..., C_n, ...\}$ is logspace uniform (P-uniform) if there exists an $O(\log n)$ space bounded (polynomial time bounded, respectively) Turing machine that, given n as input in unary, computes a binary encoding of circuit C_n.

By requiring circuit families to be logspace uniform, many nice complexity theory results follow. For instance, the parallel computation thesis states that parallel time is polynomially related to sequential space [44]; while this is not true for general circuits, it does hold for logspace uniform circuit families [8]. On the other hand, P-uniform circuit families loosely correspond to practical, constructible circuits.

1.3.3 Division with Bounded Fanin Boolean Circuits

Of the basic arithmetic functions (addition, subtraction, multiplication, and division), division is the most computationally complex, and the least understood. The sequential complexity of division was examined and compared to the complexity of multiplication in Cook's Ph.D. thesis [19]. In fact, it is known that addition, subtraction, and multiplication of n-bit numbers can be computed by a logspace transducer, but it is unknown whether this is possible for division. From a result of Borodin, any function computable by a logspace-uniform bounded-fanin boolean circuit family with polynomial size and $O(\log n)$ depth can be computed by a Turing machine in $O(\log n)$ work-space [8]. Such circuits are known for addition, subtraction, and multiplication, but not for division.

An interesting result of Beame, Cook, and Hoover gives a circuit family for division with polynomial size and $O(\log n)$ depth, but the circuit family is P-uniform, and not known to be logspace uniform (the circuit family requires tables of prime numbers whose size grows polynomially with the input size) [7]. The first polynomial size logspace uniform circuit family with depth less than $\Omega(\log^2 n)$ was given by Reif, who gave a circuit family with depth $O(\log n \log\log n)$ [51]. This depth bound is still the best achieved by any logspace uniform circuit family, although the size has been decreased by later work of Shankar and Ramachandran [62] and the work in this dissertation.

The results presented in chapter 2 of this dissertation were originally presented at the Twenty First ACM Symposium on Theory of Computing [56], and later appeared in journal form [58]. Let $M(n)$ denote the size of the smallest $O(\log n)$ depth logspace uniform circuit family for multiplication. The best upper bound known for this problem is from the Schönhage-Strassen multiplication algorithm, giving $M(n) = O(n \log n \log\log n)$ [60]. The parallelization of the classic sequential algorithm for division gives a circuit family with size $O(M(n))$ and depth $O(\log^2 n)$ [19]. As stated above, the best depth bound known for division is $O(\log n \log\log n)$, and the best previously known corresponding size bound is $O(n^{1+\epsilon})$ for small, constant $\epsilon > 0$ [62]. The algorithm presented in chapter 2 of this dissertation gives a logspace uniform circuit family with $O(M(n))$ size and $O(\log n \log\log n)$ depth, so the best previously known size and depth bounds are met *simultaneously.*

1.3.4 Threshold Circuit Results

In chapter 3 the model of computation is the threshold circuit. Such circuits are allowed to have unbounded fanin, and the nodes can compute arbitrary threshold functions (notice that AND and OR are just special cases of threshold functions). This is a very powerful model of computation, and can compute many functions with polynomial size circuit families that have a constant bound on their depth. In particular, the parity function is computable by constant depth threshold circuits, but not by constant-depth unbounded-fanin boolean circuits (when the size of both is restricted to be polynomial) [29]. Threshold circuits are a model of interest to artificial intelligence researchers, as a neuron can be modeled by a threshold gate. In fact, constant-depth threshold circuits correspond to several proposed models of learning, such as the Connectionist models [26] and the Boltzmann machine [34, 1, 45]. Furthermore, recent results have shown that it is practical to use threshold gates in VLSI constructions [30].

In fact, we show in chapter 3 that many non-trivial functions can be computed by constant depth threshold circuits. Previously, it had been shown that multiplication and iterated sum can be computed by polynomial size, constant depth threshold circuits [46]. Furthermore, when we consider P-uniform circuit families, it is possible to compute iterated product and integer

division in constant depth [52]. In chapter 3, we show how to reduce the size of constant depth circuits for iterated product and division. Specifically, we construct a constant depth circuit family for integer division with size $O(n^{1+\epsilon})$ for any constant $\epsilon > 0$, and then show how to use these results to give simulations (using threshold gates) of circuits over finite fields. Furthermore, we show how finite field circuits can simulate threshold gates in constant depth, showing an equivalence between constant depth finite field circuits and constant depth threshold circuits. This result shows that the class of constant-depth threshold circuits (TC^0) has a concise algebraic structure. This could be an important first step in proving lower bounds for threshold circuits.

1.4 Algorithmic Motion Planning

Now we review the problems addressed in the second part of this dissertation. With the increasing use of industrial robots, the associated computational problems such as planning and control are receiving a lot of attention. In a typical industrial setting, we need to plan movements for a robot from some initial position to a given goal position. Furthermore, it is desirable to maximize the efficiency of the robot under some cost function; we examine the problem of finding a plan that takes as little time as possible. In the future, robots will take on more tasks in manufacturing and manual labor, and minimizing the time required to perform the robot's tasks is necessary for the most efficient use of high-cost robots.

We also address a more general problem — what if some obstacles are allowed to move? When the movement of the obstacles is easily predictable, the problem has been extensively studied [53, 66]. We look at a single moving obstacle with unpredictable motion. In the worst case, this can be viewed as an additional robot in the system that is a hostile adversary, and we need to devise a strategy for reaching the goal while avoiding this adversary.

The following sections present a brief history of work on algorithmic motion planning problems, followed by a summary of results presented in chapters 4 and 5.

1.4.1 Geometric Planning Problems

The most basic motion planning problem, and the first problem examined from an algorithmic standpoint, is simply to determine if a robot can travel through a set of fixed obstacles to some goal position. This problem has been extensively studied by many researchers, including Lozano-Perez and Wesley (under the name "FIND-PATH problem") [39], Reif (using the name "furniture mover's problem") [47], and Schwartz and Sharir (who use the name "piano mover's problem") [61]. In all these cases, the robot is described as a set of linked polyhedra, and the obstacles are polyhedra fixed in 3-space. The complexity of this problem depends heavily on the specifics of the input instance. For instance, if the robot is allowed to have n links, then the problem has been shown to be PSPACE-hard [47]; however, if the robot has only a constant number of degrees of freedom (but the obstacle complexity can grow with n), then the problem is solvable in polynomial time [61].

1.4.2 Dynamics Constraints

The geometric planning problems discussed above do not address the question of *how long* it takes the robot to reach its goal. We look at the problem of finding trajectories when the time to reach the goal is to be minimized. Clearly, some new parameters must be introduced into

the problem to state how fast the robot can move, how fast it can change speed, etc. — these parameters are called the dynamics constraints.

The simplest type of dynamics constraint is a bound on the velocity of the robot. Specifically, for some bound $v_{\max}$, the norm of the velocity vector can never exceed $v_{\max}$; the exact norm used can affect the complexity of the problem, and the two most commonly used for robotics problems are the Euclidean norms[2] L_2 and L_∞. If only the velocity is bounded (so arbitrarily sharp turns are allowed), then this problem becomes exactly the shortest path problem studied in computational geometry. Polynomial time algorithms are known for the shortest path problem in two dimensions [63], but it has been shown that in three dimensions (using the L_2 norm) the problem becomes NP-hard (the proof actually works for any L_p norm, where p is finite) [14].

In a real world situation, the force available to move the robot is also bounded. This can be directly represented by bounding the acceleration of the robot. As with the velocity bounds, the acceleration bounds are stated as a bound on the norm of the acceleration. When both velocity and acceleration are bounded, the motion planning problem is referred to as *kinodynamic motion planning* — a term that reflects the presence of both kinematic constraints (the obstacles) and dynamics constraints (the velocity and acceleration bounds). Exact solutions to kinodynamic motion planning problems are extremely difficult to produce — the only presently known solutions are for one dimension [41] and for two dimensions using the L_∞ norm [13]. The first algorithm runs in polynomial time, while the latter requires polynomial space.

1.4.3 Approximation Algorithms

In light of the computational difficulty of motion planning problems, any practical algorithms must be restricted to solving an approximate version of the original problem. For instance, as stated above, the 3-dimensional shortest path problem is NP-hard, but Papadimitriou has given a polynomial time algorithm that can find a path whose length is provably within a $(1+\epsilon)$ factor of the optimal path [43]. In other words, if the shortest path has length L, then for any $\epsilon > 0$, Papadimitriou's algorithm will find a path that has length at most $(1+\epsilon)L$ — the running time is polynomial in both the geometric complexity of the environment and in $\frac{1}{\epsilon}$. The running time for this approximation problem has been improved by Clarkson, as well as giving an $O(\frac{n}{\epsilon}\log n)$ time approximation algorithm for two dimensions [17].

Several algorithms for kinodynamic planning have been devised, but most do not give provable bounds on the goodness of the approximation. Examples of such early approximation algorithms include the work of Sahar and Hollerbach [59] and Shiller and Dubowsky [64]. The first polynomial time algorithm that produces a provably good approximation is due to Canny, Donald, Reif, and Xavier, for dynamics bounds stated in terms of the L_∞ norm [12]. Later work has given approximation algorithms using the L_2 norm — one such algorithm appears in chapter 4 of this dissertation (it originally appeared in [57]), and another algorithm was devised in concurrent, independent research by Donald and Xavier [24]. An approximation algorithm for open-chain manipulators is given by Jacobs, Heinzinger, Canny, and Paden [37].

1.4.4 Planning in a Cooperative Environment

In chapter 4, the following motion planning problem is examined: the environment consists of a set of polyhedral obstacles in d-dimensional space with bounded diameter. The robot is a single point, and the L_2 norms of both velocity and acceleration are bounded by values $v_{\max}$

[2] For a vector $\mathbf{x} = (x_1, x_2, ..., x_n)$, the L_p norm (for finite p) is defined by $\|\mathbf{x}\|_p = \left[\sum_{i=1}^n x_i^p\right]^{1/p}$, and the L_∞ norm is $\|\mathbf{x}\|_\infty = \max_{1\le i\le n} |x_i|$.

and $a_{\max}$, respectively. The robot must also respect an affine safety function $\delta(v) = c_1 v + c_0$ that maps velocity magnitudes to distances — at any time t, if the instantaneous velocity of the robot is v, then the robot must be at least $\delta(v)$ distance away from all obstacles.

We look at solving an ϵ-approximation of the exact minimum time motion planning problem described above. In particular, if there exists an exact $\delta(v)$-safe solution taking time T, then an ϵ-approximation algorithm guarantees that it will find a $(1-\epsilon)\delta(v)$-safe solution taking time (at most) $(1+\epsilon)T$. Such an approximation algorithm for dynamics bounds stated in terms of the L_2 norm is given in chapter 4, and the running time of the algorithm is $O\left(n^2 \log n + \left[\frac{v_{\max} a_{\max} D}{\epsilon^9}\right]^d\right)$, where D is the diameter of the workspace and n is the length of the encoding of the environment. In other words, for fixed d the algorithm is fully polynomial in the combinatorial and algebraic complexity of the environment, and pseudopolynomial[3] in the dynamics bounds.

Canny, Donald, Reif, and Xavier examined a similar problem, the key difference being that dynamics bounds for their problem were stated in terms of the L_∞ norm [12]. The L_2 norm is a more natural measure because it removes the dependence on the orientation of the coordinate axes. The key to both algorithms is the proof of a *tracking lemma*, which states that for any exact robot trajectory, there is a “close” approximating trajectory that travels only between discrete points. In fact, in all aspects other than the tracking lemma, the algorithms of Canny, Donald, Reif, and Xavier and the algorithm in chapter 4 are very similar. The tracking lemma in this dissertation is much more complex than that of Canny, Donald, Reif, and Xavier — this is necessary due to the increased complexity introduced by using the L_2 norm. In particular, with the L_∞ norm, each dimension has dynamics bounds that are independent of the other dimensions — thus the tracking lemma needs to be proved in only one dimension. The L_2 norm does not possess such a nice independence property, and the result is the increased complexity of the tracking lemma proof.

1.4.5 Planning in a Hostile Environment

In chapter 5, we examine a similar motion planning planning problem, but add a moving, unpredictable obstacle. Since the obstacle growing techniques [39] do not work in a situation with several moving obstacles, we allow both our robot and the moving obstacle to be polyhedral objects. Since the motion of this obstacle is unpredictable (but it will be restricted by dynamics constraints), any successful motion plan must be successful even in the worst case — against a hostile adversary. The motion plan for this problem is allowed to be *dynamic*; that is, it can change during execution depending on the current position of the moving obstacle. We call such a problem a *pursuit game*, and the goal is to devise a strategy for a robot (the evader) that successfully avoids the moving obstacle (the pursuer).

It turns out that a very simple version of this problem — when the pursuer and evader have bounds on just the L_2 norm of their velocity and the game takes place in 3-space — is very hard (EXPTIME-hard, to be exact). The proof of this lower bound does not even require that the discovered strategy be time-minimum, so the simple *reachability* problem is very hard. Recall that without the pursuer, the reachability problem can be solved in polynomial time [61]. This is the first provably intractable robotics problem with perfect information.

In addition, by extending the tracking lemma from chapter 4, we give provably good approximation algorithms to solve pursuit games with various types of dynamics constraints: either velocity or both velocity and acceleration may be bounded, and the bound may be in terms of

[3] This terminology comes from [42].

either the L_2 or the L_∞ norm. The approximation algorithm uses a notion of safety that is similar to that used in approximate kinodynamic planning, adding the constraint that the evader needs to keep a safe distance between itself and the pursuer, as well as the static obstacles.

Chapter 2

Newton Iteration and Division

2.1 Introduction

At the heart of all numerical computations are the basic operations of addition, subtraction, multiplication, and division (also, to a lesser extent, more complex operations such as powering, finding square roots, computing logarithms, etc.). It is vital to study these problems from the standpoint of parallel algorithms, because even in commonly used single processor machines the basic operations are done in parallel (by parallel paths through low-level logic circuits).

Early in school, a student learns how to perform the functions of addition, subtraction, multiplication, and division. In fact, these topics are usually presented in this order due to the increasing difficulty of the operations. Studies in parallel algorithms support the sense of difficulty assigned by our elementary school teachers — optimal algorithms exist for addition and subtraction, while good algorithms exist for multiplication (even the best of which is not known to be optimal), and division seems to be even harder. For more information on the operations of addition and subtraction, see the fantastic overview written by Pippenger [46]; for multiplication see either [2] or the original paper by Schönhage and Strassen [60].

In this chapter, the parallel complexity of division is compared with the complexity of the other elementary operations. The problem of integer division is defined to be a function that takes a pair of input values (y, x), and produces the pair of values (q, r) such that $y = qx + r$, where $0 \le r < x$ (i.e., a quotient and a remainder). Reduction of division to multiplication via Newton approximation has been shown to provide good sequential results [19], but these results do not translate well to parallel algorithms. Rather, a modified version of Newton approximation (called *high order* Newton approximation for reasons that will become clear) is used to obtain parallel results that are almost optimal. On the way to the results for division, it will be discovered that finding limited integer powers is vital for division, so ways of accomplishing this are discussed.

As is standard practice when comparing the complexity of algorithms, the focus of this chapter will be on reductions to other problems. It is sufficient to consider the problem of finding reciprocals in place of division. As we are considering only integer operations, the idea of a reciprocal is not clear — in general, the real reciprocal of an integer will *not* be an integer. Therefore, given an n-bit input integer x, the integer reciprocal is defined as the value

$$\left\lfloor \frac{2^{2n}}{x} \right\rfloor .$$

Notice that this is simply the shifted binary fixed point approximation to the real reciprocal;

it should be obvious how the reciprocal can be used with a constant number of multiplications to solve the division problem. This definition has been taken from [2].

The notation $\leq_{sd}$ denotes a constant size and depth reduction; in other words, if f and g are two functions, then $f \leq_{sd} g$ if, given any circuit family computing g in size $S(n)$ and depth $D(n)$, a circuit family can be constructed which computes f in size $O(S(n))$ and depth $O(D(n))$. Letting SQ denote the function that squares an n-bit integer and MULT denote the problem of multiplying two n-bit integers, it is easy to see that SQ $\leq_{\mathrm{sd}}$ MULT. It is also true, but not quite as obvious, that MULT $\leq_{sd}$ SQ since $xy = \frac{1}{2}[(x+y)^2 - x^2 - y^2]$ (addition is easily accomplished, and the multiplication by $\frac{1}{2}$ is simply a binary shift by one bit). The notation $\equiv_{sd}$ is used for two problems that are constant size-depth reducible to each other, so SQ $\equiv_{sd}$ MULT as just shown.

Re-examining our rather arbitrary hierarchy of difficulty for arithmetic problems, a good candidate for a reduction of division would be multiplication. In fact, letting REC denote the integer reciprocal problem and using the new notation, it can be shown that SQ $\leq_{sd}$ REC by

$$x^2 = \frac{1}{\frac{1}{x} - \frac{1}{x+1}} - x.^{1} \tag{2.1}$$

Noting that the $\leq_{sd}$ relation is transitive, this also means that MULT $\leq_{sd}$ REC, so finding reciprocals is at least as hard (in the sense of constant size-depth reductions) as multiplication. This verifies the fact that multiplication is a good candidate when trying to reduce division.

Throughout this chapter, the notation $M(n)$ will be used to represent the smallest size required by any circuit family that multiplies two n-bit numbers in $O(\log n)$ depth. As there are no known optimal algorithms for multiplication at this time, the exact value of $M(n)$ is unknown; however, the value is easily lower-bounded by $M(n) = \Omega(n)$ and upper-bounded by $M(n) = O(n \log n \log\log n)$ (the upper bound is due to an algorithm by Schönhage and Strassen [60]). It is assumed that $M(n)$ satisfies the equation

$$M(cn) \leq cM(n) \tag{2.2}$$

for all positive $c \leq 1$. Almost all complexity measures that are $\Omega(n)$ satisfy this bound, so the assumption is not too great.

In the text that follows, the notation $\mathsf{RECIPROCAL}(x, n)$ refers to the *function* of integer reciprocal, without reference to a particular algorithm; the arguments x and n denote the input value and the size of the input, respectively. When referring to specific algorithms that compute the reciprocal function, the notation used will be $\mathsf{RECIP1}(x, n)$, $\mathsf{RECIP2}(x, n)$, etc.

2.2 Newton Approximation

Newton approximation is a tool commonly used by numerical analysts to find the zeros of a function. In numerical analysis terms, Newton approximation (in general) has quadratic convergence — what this means to the division problem will become clear shortly.

Consider a differentiable function $f(x)$ that has first derivative $f'(x)$ and has a zero at x_0 (so $f(x_0) = 0$). Assuming that $f'(x)$ is non-zero in a reasonable neighborhood of x_0, we can make an initial guess for x_0 (call the initial guess y_1) and use the slope $f'(y_1)$ to estimate how

[1]This is actually an abuse of notation, since equation (2.1) uses real reciprocals instead of the defined integer reciprocal; however, it is not hard to see how the integer reciprocal can be used to approximate real reciprocals in the calculation of equation (2.1).

far y_1 is from the zero. This produces a new estimate for x_0 (call it y_2) and the process can be repeated producing a sequence of estimates $y_1, y_2, y_3, \ldots$ that converges to x_0 for all well-behaved functions and good initial approximations. In mathematical terms, this becomes

$$y_{i+1} = y_i - \frac{f(y_i)}{f'(y_i)}. \tag{2.3}$$

The convergence rate for the general case is beyond the scope of interest of this dissertation — the interested reader can consult any introductory numerical analysis text (for example, see [10]).

Consider the function $f(y) = 1 - \frac{1}{xy}$. Obviously, $\frac{1}{x}$ is a zero of f, and the derivative $f'(y) = \frac{1}{xy^2}$ is non-zero for all $y \neq 0$. Using this function f, equation (2.3) gives a sequence defined by

$$y_{i+1} = 2y_i - xy_i^2. \tag{2.4}$$

This equation will take a good initial estimate and converge to $\frac{1}{x}$. A word of warning is appropriate here — notice how easily we slipped into solving the problem of real reciprocals instead of integer reciprocals. Fortunately, the problem is not too great — as was noted before, the integer reciprocal is simply a scaled representation of the fixed point binary approximation to the real reciprocal. Re-writing the above equation with this scaling in mind, the following equation generates a sequence that converges to the integer reciprocal using only integer operations.

$$y_{i+1} = \left\lfloor \frac{2^{2n+1}y_i - xy_i^2}{2^{2n}} \right\rfloor \tag{2.5}$$

This formula works quite well, and direct implementation yields a circuit that computes integer reciprocals in size $O(M(n)\log n)$ and depth $O(\log^2 n)$. The $\log n$ multiplier in the size comes from the fact that $\Theta(\log n)$ iterations of equation (2.5) are needed, each of which requires a multiplication of n bit values.

Noticing that the approximation y_i is very inaccurate in the early stages, it seems pointless to do calculations with all the erroneous bits of y_i. In fact, this observation produces a new algorithm which removes the $\log n$ multiplier from the size bound above; the algorithm that accomplishes this is shown in figure 2.1, and proofs of correctness and complexity are given in theorem 2.2.1. The approximation formula used in figure 2.1 looks different from that in equation (2.5), but the only difference is due to the new scaling required by having only $\frac{n}{2}$ bits for y_i.[2] The "for loop" in algorithm RECIP1 is also a new addition; it is present to overcome errors induced by using fixed point approximation to real numbers. The usefulness of this adjustment stage will become apparent from the proof of theorem 2.2.1. Algorithm RECIP1 is originally due to Cook [19], and can be found in [2]. The parallel analysis found here is original, and is given as a basis upon which we build our more efficient algorithm.

For the remainder of this section, as well as in sections 4 and 5, the n-bit input x is assumed to satisfy $2^{n-1} \leq x < 2^n$ (i.e., the high-order bit is set). The algorithms may be modified so they do not require this assumption by simply shifting x (by bits) into the appropriate range, performing the algorithms found in this chapter, and shifting the results back into the proper range. The complexity of the shifting stages is negligible compared to the complexity of the algorithms discussed. Similarly, it is assumed that n is a power of 2.

---

[2] One way to view this is that now the precision of the fixed point representation is changed at each stage; at the smallest stage, the fractional precision is only $\pm\frac{1}{2}$, but at the next stage the precision is $\pm\frac{1}{4}$, and then $\pm\frac{1}{16}, \pm\frac{1}{256}, \ldots$

```
Algorithm RECIP1(x, n);
    if n = 1
      then begin
        y ← 4;
        end;
      else begin
        t ← RECIP1(⌊x / 2^{n/2}⌋, n/2);
        y ← ⌊(2^{(3/2)n+1} t − x t^2) / 2^n⌋;
        for i ← 3 downto 0 do
            if (x(y + 2^i) ≤ 2^{2n})
              then begin
                y ← y + 2^i;
                end;
        end;
    return (y);
end.
```

Figure 2.1: Algorithm RECIP1

Theorem 2.2.1 Algorithm RECIP1 in figure 2.1 correctly computes the integer reciprocal of x, and is realized with a circuit family of size $O(M(n))$ and depth $O(\log^2 n)$.

Proof: The following proof of the correctness is rather tedious; this comes from the fact that fixed point approximations to real numbers are used, so small errors (from rounding or truncating) are introduced at various points. A very simple way to get a feeling for why this method works is to examine how the error of an approximation is affected by equation (2.4); while this is not a proof that algorithm RECIP1 is correct, it does provide insight that is useful if the following proof is found to be confusing.

To simplify notation, let r represent the value returned by RECIP1(x, n). To prove the correctness of RECIP1, it is necessary to show that $r = \left\lfloor \frac{2^{2n}}{x} \right\rfloor$; in other words, $xr = 2^{2n} - s$ where $0 \leq s < x$. The proof is by induction on n; the correct value for $n = 1$ is stated explicitly in the algorithm.

Assume that the algorithm returns a correct value for inputs of size $\frac{n}{2}$. Let t be the value of RECIP1$(\left\lfloor \frac{x}{2^{n/2}} \right\rfloor, \frac{n}{2})$ as in figure 2.1, and let $d = 2^{\frac{3}{2}n+1}t - xt^2$. Also, denote the most significant $\frac{n}{2}$ bits by x_1 and the least significant $\frac{n}{2}$ bits by x_0, so $x = x_1 2^{n/2} + x_0$. The value d can now be written as

$$d = 2^{\frac{3}{2}n+1}t - t^2(x_1 2^{n/2} + x_0).$$

The value of interest in this proof is xr, so first we will find xd and then bound the difference between this and xr.

$$xd = 2^{2n+1}x_1 t + 2^{\frac{3}{2}n+1}x_0 t - t^2(x_1 2^{n/2} + x_0)^2$$

Using the induction hypothesis (that $x_1 t = 2^n - s'$, where $0 \leq s' < x_1$), this can be simplified to

$$xd = 2^{3n} - (2^{n/2}s' - tx_0)^2.$$

Dividing by 2^n, the result is

$$\frac{xd}{2^n} = 2^{2n} - (s' - \frac{tx_0}{2^{n/2}})^2.$$

Noting that s' and $\frac{tx_0}{2^{n/2}}$ are both positive and that the difference of these two is squared, it is possible to bound

$$\left(s' - \frac{tx_0}{2^{n/2}}\right)^2 \leq \max\left\{(s')^2, \left(\frac{tx_0}{2^{n/2}}\right)^2\right\}.$$

By the induction hypothesis, $s' < x_1 < 2^{n/2}$, so $(s')^2 < 2^{n/2}x_1 \leq x$. Furthermore,

$$\left(\frac{tx_0}{2^{n/2}}\right)^2 \leq \left(\frac{2^{n/2}x_0}{x_1}\right)^2 < \left(2^{n/2+1}\right)^2 = 2^{n+2} \leq 8x,$$

so $(s' - \frac{tx_0}{2^{n/2}})^2 < 8x$. In other words,

$$\frac{xd}{2^n} > 2^{2n} - 8x.$$

Now, considering the value y calculated by the Newton approximation equation,

$$xy = x\left\lfloor\frac{d}{2^n}\right\rfloor > x(\frac{d}{2^n} - 1) = \frac{xd}{2^n} - x > 2^{2n} - 9x$$

The adjustment stage of $\mathsf{RECIP1}$ will adjust the least significant four bits of y to the correct value, as long as $\mathsf{RECIPROCAL}(x,n) - y \leq 15$ entering the adjustment stage. It has just been shown that, in fact, $\mathsf{RECIPROCAL}(x,n) - y \leq 9$, so $\mathsf{RECIP1}$ correctly returns the integer reciprocal.

The complexity of the circuit is very straight-forward to calculate. To calculate the size, notice that $\mathsf{RECIP1}$ performs only a constant number of multiplications and simpler operations on $O(n)$ bit numbers in addition to the recursive call. In other words, the recurrence

$$S(n) \leq S(\frac{n}{2}) + cM(n) \tag{2.6}$$

$$S(1) = 1$$

describes the size of the circuit for $\mathsf{RECIP1}$. The solution to equation (2.6) is given by

$$S(n) \leq c\sum_{i=0}^{\log n} M(\frac{n}{2^i}).$$

From the assumption of equation (2.2), we know that $M(\frac{n}{2^i}) \leq \frac{1}{2^i}M(n)$, so the resulting size is $S(n) = O(M(n))$.

The depth of each level of recursion is bounded by $O(\log n)$, and since there are $\log n$ stages, the total depth is bounded by $O(\log^2 n)$. This is a rather simplistic depth analysis, but closer examination shows that this is the tightest upper bound possible. ∎

```
Algorithm REPEATSQ(x, m);
    {Consider m in its binary representation:
                 m = m_{⌊log m⌋} 2^{⌊log m⌋} + ··· + m_2 2^2 + m_1 2^1 + m_0 2^0}
    i ← 0;
    p ← x;
    y ← 1;
    while i ≤ log n do begin
        if m_i = 1
          then begin
            y ← yp;
            end;
        p ← p^2;
    end;
end.
```

Figure 2.2: Repeated squaring method of taking powers

2.3 Integer Powering

The seemingly unrelated problems of integer reciprocal and integer powering are actually very closely related. In fact, it has been shown by Beame, Cook, and Hoover [7] that the two problems are equivalent with respect to constant depth reductions.[3] A survey of the research on integer division shows that all known efficient parallel reciprocal algorithms use powering as an integral part (see, for example, [7], [62], and [56]).

As an introduction to powering, consider a simple powering algorithm; the problem is to raise an n-bit number x to the m-th power, where $m \leq n$. Now write m in its binary notation, so $m = m_{\lfloor \log m \rfloor} 2^{\lfloor \log m \rfloor} + \cdots + m_2 2^2 + m_1 2^1 + m_0 2^0$. An algorithm (called repeated squaring) that takes advantage of the binary representation of m is shown in figure 2.2.

The complexity analysis of this algorithm is particularly easy, resulting in a circuit family with size $O(M(nm))$ and depth $O(\log n \log m)$. Note that this is considerably better than simply multiplying x by itself m times which takes size $O(mM(nm))$ and depth $\Theta(m \log n)$.

With this algorithm in mind, consider the reciprocal algorithm of the previous section; at first glance, RECIP1 doesn't seem to take any powers greater than squaring y_i. However, if a more global view is invoked, this squared term is again squared in the next stage, and repeatedly squared until the end of the algorithm. In other words, the algorithm actually takes large powers using the repeated squaring algorithm! An observant reader would have noticed that the depth of the algorithm RECIP1 is the same as the depth of the algorithm REPEATSQ (with $m = n$). Now it can be seen that this is no coincidence — RECIP1 was actually performing operations almost identical to REPEATSQ.

An interesting question now arises: Can reciprocals be computed in depth smaller than $\Omega(\log^2 n)$ if there were an algorithm for computing powers in small depth? Indeed, this is the case; in fact, with respect to constant depth reductions that preserve polynomial size (note the difference between this and constant size and depth reduction), it has been shown that division and powering are equivalent [7]. Unfortunately, finding small depth circuits for powering seems

[3] A constant depth reduction is similar to the constant size and depth reduction mentioned earlier in this chapter, except that the size can increase by a polynomial amount.

to be as hard as looking at the reciprocal problem directly. What follows is a description of a powering algorithm that only requires $O(\log n \log\log n)$ depth (for $m = n$). The algorithm is rather confusing to people who haven't seen anything like it before; a good "warm-up" exercise would be to read and understand the multiplication algorithm of Schönhage and Strassen (a good description can be found in [2]). The algorithm presented here consists of two parts: reducing the size of the input number x, and reducing the power m.

2.3.1 Bit Reduction

Again, we wish to raise an n-bit number x to a power m, where $m \leq n$. The number x has at most $d = \left\lfloor \frac{n}{\log b} \right\rfloor + 1$ digits in base b notation and can be written as

$$x = x_{d-1}b^{d-1} + \cdots + x_2 b^2 + x_1 b + x_0. \tag{2.7}$$

If an indeterminate z is substituted for the occurrences of b that are raised to a power, then x can be represented by a polynomial $p(z) = x_{d-1}z^{d-1} + \cdots + x_2 z^2 + x_1 z^1 + x_0$, where $p(b) = x$.

Operations with such polynomials mirror the same operations performed on the numbers themselves, so, for example, if x is represented by $p(z)$ and y is represented by $q(z)$, then the product of the two polynomials has the property that $p(z)q(z)|_{z=b} = p(b)q(b) = xy$.[4] The current interest is in powering, and it can be noticed that if x is represented by $p(z)$ and m is an integer, then $[p(z)]^m|_{z=b} = [p(b)]^m = x^m$. Efficient polynomial arithmetic is made possible by a domain change through Fourier transforms; we now see how this is done.

Returning to the original problem, let x be an n-bit number, where n is a power of 2 — say $n = 2^p$. The input x can be broken into $k = 2^r$ blocks of $l = 2^{p-r}$ bits each, so letting $b = 2^l$, equation (2.7) becomes

$$x = x_{k-1}2^{l(k-1)} + \cdots + x_2 2^{2l} + x_1 2^l + x_0.$$

The polynomial representation of x (as described above) is therefore $p(z) = x_{k-1}z^{k-1} + \cdots + x_2 z^2 + x_1 z + x_0$; notice that $p(2^l) = x$.

To raise x to the mth power, simply find the polynomial $[p(z)]^m$ and evaluate at $z = 2^l$. Unfortunately, the polynomial $[p(z)]^m$ has degree $m(k-1)$, which is too large for an efficient powering algorithm (it is an interesting exercise to follow the development of the powering algorithm using all terms of $[p(z)]^m$ to see exactly where things go amiss).

Consider calculating $[p(z)]^m \pmod{z^k - 1}$. When the value $z = 2^l$ is inserted, the result is $x^m \pmod{2^n - 1}$; by padding the input with zeros and increasing n to insure that $2^n - 1 > x^m$, this method produces the exact answer. Furthermore, polynomials modulo $z^k - 1$ never have degree greater than $k - 1$, so the problem of growing polynomial degrees has disappeared. With this in mind, the subject of most of the remainder of this section will be the problem of modular powering.

Let $b(z) = b_{m(k-1)}z^{m(k-1)} + \cdots + b_2 z^2 + b_1 z + b_0$ be the exact value of $[p(z)]^m$, and let $d(z) = d_{k-1}z^{k-1} + \cdots + d_1 z + d_0$ be the reduction of $b(z)$ modulo $z^k - 1$ so that $d(z)$ has degree less than k. Since $z^k \equiv 1 \pmod{z^k - 1}$, it is easy to see that for $i = 0, 1, ..., k-1$,

$$d_i = \sum_{j=0}^{m-1} b_{jk+i}.$$

[4]The notation $p(x)|_{x=a}$ means the polynomial $p(x)$ evaluated at $x = a$; in other words, $p(a)$. Similarly, $p(z)q(z)|_{z=b}$ means to multiply the polynomials $p(z)$ and $q(z)$, and evaluate the resulting polynomial at $z = b$.

All b_i with $i > m(k-1)$ are assumed to be zero. Let $D = (d_0, d_1, ..., d_{k-1})$ and $X = (x_0, x_1, ..., x_{k-1})$ denote vectors of the coefficients of $d(z)$ and $p(z)$, respectively. The following lemma demonstrates an efficient way of computing the modular power polynomial $d(x)$ using Discrete Fourier Transforms (DFTs).[5] This is a standard method (introduced for powering in [51]) called *positive wrapped convolution*.

Lemma 2.3.1 Let X and D be defined as above, and let $\mathsf{DFT}_k(X) = (t_0, t_1, ..., t_{k-1})$. Then $\mathsf{DFT}_k^{-1}((t_0^m, t_1^m, ..., t_{k-1}^m)) = D$.

Proof: Let ω be a principal kth root of unity. By the definition of the DFT,

$$t_i = \sum_{j=0}^{k-1} x_j \omega^{ij} = p(\omega^i)$$

for all $i = 0, 1, ..., k-1$, where ω is a principal kth root of unity. Raising each t_i to the mth power gives

$$t_i^m = [p(\omega^i)]^m = [p(z)]^m|_{z=\omega^i} = \sum_{j=0}^{m(k-1)} b_j \omega^{ij} = \sum_{p=0}^{m-1} \sum_{q=0}^{k-1} b_{pk+q} \omega^{i(pk+q)}.$$

But $\omega^{i(pk+q)} = (\omega^k)^{ip} \omega^{iq} = \omega^{iq}$ since ω is a kth root of unity, so

$$t_i^m = \sum_{p=0}^{m-1} \sum_{q=0}^{k-1} b_{pk+q} \omega^{iq} = \sum_{q=0}^{k-1} \left(\sum_{p=0}^{m-1} b_{pk+q} \right) \omega^{iq} = \sum_{q=0}^{k-1} d_q \omega^{iq}.$$

By the definition of the DFT, this is simply the ith term of $\mathsf{DFT}_k(D)$. As this holds for all $i = 0, 1, ..., k-1$, it must be true that $\mathsf{DFT}_k^{-1}((t_0^m, t_1^m, ..., t_{k-1}^m)) = D$. ∎

The Fourier transform of a k-vector (representing a degree $k-1$ polynomial) requires a principal k-th root of unity ω. The polynomials that represent integers have integer coefficients, and to avoid doing computations over the complex field, it is possible to use finite rings as the basis of our computation. The ring of integers modulo $2^k - 1$ has a principal k-th root of unity of $\omega = 2$, giving this ring the further nice property that multiplication by powers of ω is easily accomplished by bit shifts. Since computations on each element of X are now done modulo $2^k - 1$ it is clear how the original problem (powering an n-bit number modulo $2^n - 1$) is reduced to smaller subproblems (powering k-bit numbers modulo $2^k - 1$). This reduction can be repeated until the size of the subproblems is trivial. Furthermore, $k^{-1} \pmod{2^k - 1}$ exists by insuring that k is a power of 2, so the inverse DFT is possible.

A problem arises from the fact that the previous discussion of powering assumes that the *exact* values for the coefficients of $[p(z)]^m \pmod{z^k - 1}$ are known, and the previous paragraph refers to only finding the coefficients modulo $2^k - 1$. The following lemma addresses this problem by showing how large to make k to insure that the coefficients are unambiguously represented in this ring (i.e., the coefficients are less than $2^k - 1$). A similar lemma was proved in [51], but the following version is more exact.

Lemma 2.3.2 The coefficients of $[p(z)]^m \pmod{z^k - 1}$ are less than $2^k - 1$ if

$$2^r - r(m-1) - lm > 0 \tag{2.8}$$

(where r, l, and m are defined in the preceding text).

[5] If the reader is unfamiliar with the Fourier transform or the Fast Fourier Transform algorithm, an introductory level discussion can be found in [2].

Proof: First, it is proved by induction that the coefficients of the polynomial $[p(z)]^m \pmod{z^k - 1}$ (calculated with coefficients from $\mathbf{Z}$) are less than or equal to $k^{m-1}(2^l - 1)^m$ for $m = 1, 2, \ldots$. The basis of the induction is easy; simply let $m = 1$ and the claimed bound becomes $2^l - 1$. The coefficients of $p(z)$ are all less than or equal to $2^l - 1$ since each coefficient is l bits long.

Now assume the claim is true for $m-1$ (that is, all of the coefficients of $[p(z)]^{m-1} \pmod{z^k - 1}$ are less than or equal to $k^{m-2}(2^l - 1)^{m-1}$). Let the expansion of $p(z)$ and $[p(z)]^{m-1} \pmod{z^k - 1}$ be as follows:

$$p(z) = x_{k-1}z^{k-1} + x_{k-2}z^{k-2} + \cdots + x_2z^2 + x_1z + x_0$$

$$[p(z)]^{m-1} = y_{k-1}z^{k-1} + y_{k-2}z^{k-2} + \cdots + y_2z^2 + y_1z + y_0$$

Notice that since $z^i \equiv z^{i \,(\bmod k)} \pmod{z^k - 1}$,

$$[p(z)]^m = [p(z)][p(z)]^{m-1} \equiv \sum_{i=0}^{k-1} \left(\sum_{j=0}^{k-1} x_j y_{i-j \,(\bmod k)} \right) z^i \pmod{z^k - 1}.$$

Regardless of the particular values of i and j, it must be true that $x_j \le 2^l - 1$ and $y_{i-j \,(\bmod k)} \le k^{m-2}(2^l - 1)^{m-1}$ (by the induction hypothesis), so $x_j y_{i-j \,(\bmod k)} \le k^{m-2}(2^l - 1)^m$. Since there are k terms like this added together for each coefficient of $[p(z)]^m \pmod{z^k - 1}$, each coefficient must be less than or equal to $k^{m-1}(2^l - 1)^m$, and the proof by induction is finished.

Returning to the lemma, condition (2.8) states that $2^r > r(m-1) + lm$. In other words, taking each side as an exponent, $2^{(2^r)} > 2^{r(m-1)}2^{lm}$, and since $k = 2^r$ this implies that $k^{m-1}2^{lm} < 2^k$. Loosening the inequality slightly, this implies that (for $m \ge 1$)

$$k^{m-1}(2^l - 1)^m < 2^k - 1. \tag{2.9}$$

The previous inductive proof showed that each coefficient in the polynomial $[p(z)]^m \pmod{z^k - 1}$ must be less than or equal to the left hand side of inequality (2.9), so each coefficient must also be less than $2^k - 1$, completing the proof of the lemma. ∎

As an example of the reduction technique just described, consider a single stage of bit reduction as shown in figure 2.3. The value for k comes from calculations involving lemma 2.3.2; lemma 2.3.3 shows how this works. Notice the call on MODPOWER in REDUCE1 — this is a recursive call that is left unspecified for the moment. As it turns out, a second type of reduction will be needed for efficient powering, and the recursive call (named MODPOWER here) may be on a *different* type of reduction. Notice the new assumption that $m \le n^{\frac{3}{8}}$. This assumption simply makes the lemma easier to prove, and will not affect the final powering result at all (in fact, it will become apparent that this is the result of passing the assumption $m \le n$ down through several lemmas).

Lemma 2.3.3 Let m be an integer satisfying $m \ge 16$ and $m \le n^{\frac{3}{8}}$. Then assuming that MODPOWER(t_i, m, k) correctly returns $t_i^m \pmod{2^k - 1}$, the reduction REDUCE1 shown in figure 2.3 correctly returns $x^m \pmod{2^n - 1}$. Furthermore, if the call on MODPOWER requires size $S(m, k)$ and depth $D(m, k)$, then REDUCE1 requires total size $kS(m, k) + O(nm^{\frac{4}{3}} \log n)$ and total depth $D(m, k) + O(\log n)$.

Proof: The correctness of REDUCE1 follows directly from the previous discussion with the important points being lemma 2.3.1 and lemma 2.3.2. The only verification that needs to be

```
Algorithm REDUCE1(x, m, n);
    p ← log n;
    q ← ⌈log m⌉;
    r ← ⌈p/2 + 2q/3⌉;
    k ← 2^r;
    Divide x into k blocks of l = 2^(p−r) bits each as (x_0, x_1, ..., x_{k−1});
    (t_0, t_1, ..., t_{k−1}) ← DFT_k(x_0, x_1, ..., x_{k−1});
    for all i = 0, 1, ..., k − 1 pardo begin
        u_i ← MODPOWER(t_i, m, k);
    end;
    (y_0, y_1, ..., y_{k−1}) ← DFT_k^{−1}(u_0, u_1, ..., u_{k−1});
    y ← y_0 + y_1 2^l + y_2 2^{2l} + ··· + y_{k−1} 2^{(k−1)l}   (mod 2^n − 1);
    return (y);
end.
```

Figure 2.3: Powering reduction style 1

done is that the condition (2.8) of lemma 2.3.2 holds; that is, that $2^r - r(m-1) - lm > 0$. What follows is basically an exercise in minimizing the function on the left hand side.

From figure 2.3, let $r = \left\lceil \frac{p}{2} + \frac{2q}{3} \right\rceil$. To avoid the ceiling function write r as $\frac{p}{2} + \frac{2q}{3} + \epsilon$, where ϵ is some value satisfying $0 \leq \epsilon < 1$. Obviously, if condition (2.8) holds for all ϵ in this interval, then the condition must also hold with the ceiling. Substituting this value for r and letting $m = 2^q$ and $l = 2^{p-r}$, the left hand side of condition (2.8) becomes

$$f(p,q,\epsilon) = 2^{\frac{p}{2}+\frac{2q}{3}+\epsilon} - \left(\frac{p}{2} + \frac{2q}{3} + \epsilon\right)(2^q - 1) - 2^{\frac{p}{2}+\frac{q}{3}-\epsilon}.$$

This formula is quite messy, but can be simplified greatly just by taking the partial derivative with respect to ϵ (which will reveal a lot of useful information).

$$\frac{\partial f}{\partial \epsilon}(p,q,\epsilon) = 2^{\frac{p}{2}+\frac{q}{3}} \ln 2 \left(2^{\epsilon+\frac{q}{3}} + 2^{-\epsilon}\right) - (2^q - 1)$$

This function is easily minimized for a given p and q (for an easy trick, substitute $t = 2^\epsilon$ and minimize with respect to t) when $\epsilon = -\frac{q}{6}$. Substituting this value into $\frac{\partial f}{\partial \epsilon}$, for any p and q the minimum value of the partial derivative with respect to ϵ is

$$2^{\frac{p}{2}+\frac{q}{2}+1} \ln 2 - (2^q - 1). \tag{2.10}$$

In terms of p and q, the assumption that $m \leq n^{\frac{3}{8}}$ translates to $q \leq \frac{3p}{8}$ (or equivalently, that $p \geq \frac{8q}{3}$). We wish to show that equation (2.10) is greater than zero for all valid p and q. For any given q, equation (2.10) is minimum when p is at its minimum — in other words, when $p = \frac{8q}{3}$. Making this substitution, equation (2.10) becomes

$$2^{\frac{11q}{6}+1} \ln 2 - (2^q - 1),$$

which is easy to show greater than zero for all $q \geq 0$.

The past few paragraphs have shown that for all valid p, q, and ϵ, the derivative $\frac{\partial f}{\partial \epsilon}(p,q,\epsilon) > 0$. In other words, for all valid p and q, $f(p,q,\epsilon)$ is increasing in ϵ; therefore, for all valid p, q, and ϵ,

$$f(p,q,\epsilon) \geq f(p,q,0) = 2^{\frac{p}{2}+\frac{2q}{3}} - 2^{\frac{p}{2}+\frac{q}{3}} - \left(\frac{p}{2}+\frac{2q}{3}\right)(2^q - 1).$$

Differentiating the right hand side with respect to p gives

$$\frac{\partial f(p,q,0)}{\partial p} = 2^{\frac{p}{2}}\frac{\ln 2}{2}\left(2^{\frac{2q}{3}} - 2^{\frac{q}{3}}\right) - \frac{2^q - 1}{2}.$$

This is obviously increasing in p, so is minimized when p is minimum; after making the substitution $p = \frac{8q}{3}$ and doing some rearranging, the above becomes

$$\frac{1}{2}\left[\left(2^{2q} - 2^{\frac{5q}{3}}\right)\ln 2 - (2^q - 1)\right],$$

which is easily shown to be greater than zero for all $q \geq 4$. In other words, for a given $q \geq 4$, $f(p,q,0)$ is increasing in p, so to minimize $f(p,q,0)$, again set p to $\frac{8q}{3}$. Therefore,

$$f(p,q,0) \geq f(\frac{8q}{3},q,0) = 2^{2q} - 2^{\frac{5q}{3}} - 2q\,(2^q - 1),$$

which is greater than zero for all $q \geq 4$.

Summarizing, it has been shown that for all valid p, q, and ϵ,

$$f(p,q,\epsilon) \geq f(p,q,0) \geq f(\frac{8q}{3},q,0) \geq 0,$$

so condition (2.8) must hold, and reduction REDUCE1 gives the correct answer by lemma 2.3.2, lemma 2.3.1, and the properties of polynomials discussed in the text before lemma 2.3.1.

The complexity of REDUCE1 relies on two results beyond the scope of this chapter: namely, that DFT_k and DFT_k^{-1} can be computed in size $O(k^2 \log k)$ and depth $O(\log n)$ by using the Cooley-Tukey algorithm [20], and the evaluation of $[p(z)]^m|_{z=2^l}$ can be done in size $O(k^2)$ and depth $O(\log n)$. In other words, all steps except the call on MODPOWER can be done in size $O(k^2 \log k)$ and depth $O(\log n)$. Furthermore, since

$$k = 2^{\lceil \frac{p}{2}+\frac{2q}{3} \rceil} \leq 2^{\frac{p}{2}+\frac{2q}{3}+1} = 2n^{\frac{1}{2}}m^{\frac{2}{3}},$$

the above size can be written as $O(nm^{\frac{4}{3}}\log n)$. Including the size for the k calls on MODPOWER, the resulting size is $kS(m,k) + O(nm^{\frac{4}{3}}\log n)$. All recursive calls are done in parallel, so the total depth is $D(m,k) + O(\log n)$. ∎

The problem with repeatedly applying REDUCE1 is that the requirements of lemma 2.3.3 make reduction to a trivial problem size impossible (since n must be at least $m^{\frac{8}{3}}$); however, it is possible to reduce the power as well as the number of bits.

2.3.2 Power Reduction

Consider raising a number x to the mth power. If m is a perfect square with $w = \sqrt{m}$, it is easy to see that $x^m = (x^w)^w$; unfortunately, m is usually not a perfect square. To handle the more common case, let $v = \lfloor\sqrt{m}\rfloor$ and calculate $(x^v)^v$. Of course, this is not the desired answer, but notice that if $e = m - v^2$ is the error in the exponent of this approximation, e can be easily bounded by

$$e = m - v^2 \leq ((v+1)^2 - 1) - v^2 = 2v = 2\lfloor\sqrt{m}\rfloor.$$

```
Algorithm REDUCE2(x, m, n);
    p ← ⌊√m⌋;
    In Parallel do part1, part2
        part1: begin
                t ← MODPOWER(x, p, n);
                u ← MODPOWER(t, p, n);
               end;
        part2: begin
                e ← m − p^2;
                e' ← ⌊e/2⌋;
                v ← MODPOWER(x, e', n);
                if (2e' = e)
                  then begin
                    w ← v^2  (mod 2^n − 1);
                    end;
                  else begin
                    w ← xv^2  (mod 2^n − 1);
                   end;
               end;
    y ← uw  (mod 2^n − 1);
    return (y);
end.
```

Figure 2.4: Powering reduction style 2

Letting $e' = \lfloor \frac{e}{2} \rfloor$, $x^{e'}$ can be computed, squared, and multiplied by x (if e is odd) to achieve x^e. Notice that this computation of x^e can be done in parallel with the computation of $(x^v)^v$, so the original problem has been reduced to 3 smaller powerings (each of which raises a number to a power less than or equal to $\sqrt{m}$) and a constant number of multiplications. This reduction is called REDUCE2 and is shown in figure 2.4.

The correctness of REDUCE2 follows easily from the above discussion, and the complexity analysis is simple, so the following lemma is stated without proof.

Lemma 2.3.4 Assuming MODPOWER(t, m, n) correctly returns the value $t^m \pmod{2^n - 1}$ for all t, m, and n, the reduction REDUCE2 shown in figure 2.4 correctly returns $x^m \pmod{2^n - 1}$. Furthermore, if the call on MODPOWER(t, m, n) requires size $S(m, n)$ and depth $D(m, n)$, then modular powering via REDUCE2 requires total size $3S(\sqrt{m}, n) + O(M(n))$ and total depth $2D(\sqrt{m}, n) + O(\log n)$.

Again, there is a problem with using just the reduction REDUCE2 — while the correct answer is returned, the number of subproblems grows too rapidly, and the depth of the powering circuit using just REDUCE2 is $\Theta(\log n \log m)$. Fortunately, in the design of REDUCE1 and REDUCE2 there were some subtle adjustments made (such as the choice for r in REDUCE1) that allow the two reductions to work very well together. Combining the two reductions is addressed in the following section.

2.3.3 Putting the Pieces Together

The final modular power algorithm consists of an initial reduction using REDUCE2 followed by a test to see if the power has been reduced to smaller than 16. If the power is less than 16, then the result can be computed using the REPEATSQ algorithm presented at the beginning of this section (taking size $O(M(n))$ and depth $O(\log n)$); otherwise, the subproblems are further reduced by two applications of REDUCE1. All three of these reductions can be viewed together as a single "composite reduction" that produces subproblems with reduced size (i.e., number of bits) and reduced power. A proof of the correctness of this algorithm, along with the complexity analysis, is given in the following theorem.

Theorem 2.3.5 Let x be an n-bit integer, and m be an integer such that $m^2 \leq n$. The algorithm just described correctly computes $x^m \pmod{2^n - 1}$ in size $O(nm^4 \log n \log\log n)$ and depth $O(\log n + \log m \log\log m)$.

Proof: The correctness of the above algorithm is proved by induction on the number of complete composite reductions required before the power is reduced below 16. If no reductions are required, the result is correct by the correctness of algorithm REPEATSQ. Assume that $R \geq 1$ reductions are required — by the condition of the theorem, $m \leq n^{\frac{1}{2}}$, so after the first reduction using REDUCE2, each subproblem of raising an n-bit number to the m'th power is such that $m' \leq n^{\frac{1}{4}}$. (Note that this means the condition for lemma 2.3.3 is satisfied.)

After the first reduction via REDUCE1, each subproblem has $k \geq n^{\frac{1}{2}}(m')^{\frac{2}{3}}$ bits. (Notice that

$$m' = (m')^{\frac{3}{4}}(m')^{\frac{1}{4}} \leq n^{\frac{3}{16}}(m')^{\frac{1}{4}} = \left(n^{\frac{1}{2}}(m')^{\frac{2}{3}}\right)^{\frac{3}{8}} \leq k^{\frac{3}{8}},$$

so the condition for lemma 2.3.3 is again satisfied.)

Following the second reduction via REDUCE1, each subproblem has $k' \geq k^{\frac{1}{2}}(m')^{\frac{2}{3}}$ bits; using the previous bounds for k, $(m')^2$ can be bounded as

$$(m')^2 = (m')(m') \leq n^{\frac{1}{4}}(m') = \left(n^{\frac{1}{2}}(m')^{\frac{2}{3}}\right)^{\frac{1}{2}}(m')^{\frac{2}{3}} \leq k^{\frac{1}{2}}(m')^{\frac{2}{3}} \leq k'.$$

In other words, after one composite reduction each subproblem of raising a k'-bit number to the m'th power satisfies $(m')^2 \leq k'$. Only $R-1$ composite reductions are required for these subproblems (since R reductions were required for the original problem), and since $(m')^2 \leq k'$, the induction hypothesis applies to say that all these subproblems are correctly solved.

Going backwards through each individual reduction in the composite reduction, it has been noted that the conditions for lemmas 2.3.3 and 2.3.4 have been satisfied, so the correctness of the algorithm follows directly from these lemmas.

Now examine the size required for this algorithm. Let $S(m,n)$ denote the size of raising an n-bit number to the mth power modulo $2^n - 1$. The result of applying the size of REDUCE2 (from lemma 2.3.4) to the size of REDUCE1 (from lemma 2.3.3) which is again applied to itself gives the size for one composite reduction. The result is (using k, k', and m' as defined above)

$$S(m,n) = 3kk'S(m',k') + O(k^2(m')^{\frac{4}{3}}\log k) + O(n(m')^{\frac{4}{3}}\log n) + O(M(n)).$$

Using the bounds $k \leq 2n^{\frac{1}{2}}(m')^{\frac{2}{3}}$ (see the proof of lemma 2.3.3) and $m' \leq m^{\frac{1}{2}}$, in addition to the new bound $k' \leq 2k^{\frac{1}{2}}(m')^{\frac{2}{3}} = 2^{\frac{3}{2}}n^{\frac{1}{4}}m'$, gives a size of

$$S(m,n) = 3kk'S(m',k') + O(nm^{\frac{4}{3}}\log n) + O(nm^{\frac{2}{3}}\log n) + O(M(n)).$$

Using the result of Schönhage and Strassen that $M(n) = O(n \log n \log\log n)$, this can be simplified greatly to

$$S(m,n) = 3kk'S(m',k') + O(nm^{\frac{4}{3}} \log n \log\log n).$$

Removing the big-O notation, the above size bound can be expressed (for some constant c) as

$$S(m,n) \le 3kk'S(m',k') + cnm^{\frac{4}{3}} \log n \log\log n.$$

Notice that this size only applies if a complete composite reduction is performed (i.e., $m' \ge 16$ or $m \ge 256$). For $m < 256$, only a constant number of multiplications are required, so $S(m,n) = O(M(n))$.

The claim is that $S(m,n) \le c'nm^4 \log n \log\log n$ for some c', and is proved by induction on m. For $m < 256$ and the appropriate c' and c'',

$$S(m,n) \le c''M(n) \le c'nm^4 \log n \log\log n,$$

so this serves as a basis for the induction. Now assume $m \ge 256$, and the induction hypothesis states that

$$S(m',k') \le c'k'(m')^4 \log k' \log\log k'$$

for $m' < m$. Using the bound $k' \le 2^{\frac{3}{2}} n^{\frac{1}{4}} m^{\frac{1}{2}}$ and noticing that $2^{\frac{3}{2}} m^{\frac{1}{2}} \le m^{\frac{11}{16}}$ for $m \ge 256$, k' can now be bounded as $k' \le n^{\frac{1}{4}} m^{\frac{11}{16}} \le n^{\frac{19}{32}}$. This means that $\log k' \le \frac{19}{32} \log n$, so using all the upper bounds,

$$\begin{aligned}
3(kk')S(m',k') &\le 3\left(2^{\frac{5}{2}} n^{\frac{3}{4}} m^{\frac{5}{6}}\right)\left(c'2^{\frac{3}{2}} n^{\frac{1}{4}} m^{\frac{1}{2}} m^2 \frac{19}{32} \log n \log\log n\right) \\
&= \frac{57}{2} c'nm^{\frac{10}{3}} \log n \log\log n,
\end{aligned}$$

so

$$\begin{aligned}
S(m,n) &\le \frac{57}{2} c'nm^{\frac{10}{3}} \log n \log\log n + cnm^{\frac{4}{3}} \log n \log\log n \\
&\le \left(\frac{57}{2} c'm^{-\frac{2}{3}} + cm^{-\frac{8}{3}}\right) nm^4 \log n \log\log n.
\end{aligned}$$

Since $m \ge 256$, this can be loosely upper bounded by

$$S(m,n) \le \left(\frac{3}{4}c' + c\right) nm^4 \log n \log\log n,$$

and for $c' \ge 4c$ this becomes

$$S(m,n) \le c'nm^4 \log n \log\log n,$$

proving the claimed size bound.

Turning to the depth, let $D(m,n)$ represent the depth of raising an n-bit number to the mth power modulo $2^n - 1$, and the depth of a composite reduction can be expressed as

$$D(m,n) = 2D(m',k') + O(\log n)$$

for $m \ge 256$ (i.e., $m' \ge 16$), and $D(m,n) = O(\log n)$ for $m < 256$. A depth bound of $D(m,n) \le c'(\log n + \log m \log\log m)$ can be proved by induction; the basis follows easily for $m < 256$.

For $m \geq 256$, the induction hypothesis states that

$$D(m', k') \leq c'(\log k' + \log m' \log\log m').$$

Since $m' \leq m^{\frac{1}{2}}$, we can bound $\log m' \log\log m' \leq \frac{1}{2}\log m(\log\log m - 1)$, so

$$\begin{aligned} D(m', k') &\leq c'(\frac{3}{2} + \frac{1}{4}\log n + \frac{1}{2}\log m + \frac{1}{2}\log m(\log\log m - 1) \\ &= c'(\frac{3}{2} + \frac{1}{4}\log n + \frac{1}{2}\log m \log\log m). \end{aligned}$$

In other words, for some constant c,

$$D(m,n) \leq 2D(m',k') + c\log n \leq \left(\frac{3c'}{\log n} + \frac{c'}{2} + c\right)\log n + c'\log m \log\log m.$$

Since $\frac{3c'}{\log n} \leq \frac{3c'}{2\log m} \leq \frac{3c'}{16}$ for $m \geq 256$,

$$D(m,n) \leq \left(\frac{11}{16}c' + c\right)\log n + c'\log m \log\log m.$$

For $c' \geq \frac{16}{5}c$, this can be simplified to

$$D(m,n) \leq c'(\log n + \log m \log\log m),$$

proving the claimed depth bound. ▮

Returning to the original (exact) powering problem, the following easy corollary completes the study of integer powering.

Corollary 2.3.6 If x is an n-bit integer and m is an integer satisfying $m \leq n$, then x^m can be computed by a circuit of size $O(nm^5 \log n \log\log n)$ and depth $O(\log n + \log m \log\log m)$.

Proof: Let $N = nm$. Since $m \leq n$, multiplying both sides of the inequality by m shows that $m^2 \leq nm = N$. By theorem 2.3.5, after padding x with zeros in the most significant $n(m-1)$ places, $x^m \pmod{2^N - 1}$ can be computed in size $O(Nm^4 \log N \log\log N) = O(nm^5 \log n \log\log n)$ and depth $O(\log N + \log m \log\log m) = O(\log n + \log m \log\log m)$. Since x^m must be less than $2^{nm} - 1$, the modular computation actually gives the exact value of x^m. ▮

2.4 High Order Convergence with Newton Approximation

Given that repeated application of the Newton approximation formula given in section 2.2 computes powers in a depth-inefficient way, it is worthwhile to examine how efficient powering methods can be incorporated to reduce the complexity of finding reciprocals.

Recall the approximation formula for finding *real* reciprocals given in equation (2.4). The initial ideas here are presented in terms of real reciprocals, and then the simple changes to the integer reciprocal problem are examined. Some algebraic manipulation shows that applying the approximation formula twice, the approximation refinement becomes

$$y_{i+2} = y_i(1 + (1 - xy_i) + (1 - xy_i)^2 + (1 - xy_i)^3).$$

In fact, the original equation can be re-written as

$$y_{i+1} = y_i(1 + (1 - xy_i)),$$

with the basic pattern emerging of

$$y_{i+m} = y_i \sum_{j=0}^{2^m-1} (1 - xy_i)^j. \tag{2.11}$$

(Of course, we haven't *proven* that this is the general form of repeated application of equation (2.4) — this is left to the interested reader. A proof that this equation, after scaling, gives the correct answer will be given in theorem 2.4.1.)

A nice property of equation (2.11) is that the upper limit of the sum does not necessarily have to be of the form $2^m - 1$ in order to work correctly. We wish to view an application of equation (2.11) as a single approximation step, so the new approximation formula can be written as

$$y_{i+1} = y_i \sum_{j=0}^{k-1} (1 - xy_i)^j. \tag{2.12}$$

This equation is called the k-th order Newton approximation formula; the name comes from the fact that convergence is of order k. Desirable convergence properties can be proven for equation (2.12), but as we are interested in integer reciprocals, the scaled version should be examined first. Performing fixed point scaling exactly as was done for the second order formula of section 2.2 gives a fixed-point equation; however, as before, only a small number of bits of y_i need to be considered in the calculation of y_{i+1}. If we let $y_i = \mathsf{RECIPROCAL}(\left\lfloor \frac{x}{2^{n-d}} \right\rfloor, d)$ (i.e., the integer reciprocal of the d most significant bits of x), and $x' = \left\lfloor \frac{x}{2^{n-dk}} \right\rfloor$ (the dk most significant bits of x) then the resulting equation is

$$y_{i+1} = \left\lfloor \frac{y_i \sum_{j=0}^{2k-1} 2^{d(k+1)(2k-j-1)} (2^{d(k+1)} - x'y_i)^j}{2^{2dk^2}} \right\rfloor \tag{2.13}$$

Notice that here the upper limit on the sum is $2k-1$ instead of $k-1$ — the upper limit has been raised to overcome the same type of problem that required the adjustment stage of RECIP1; however, equation (2.13) is still referred to as the kth order Newton approximation formula.

To construct an algorithm using equation (2.13), the exact order of each approximation step must be considered; this schedule of approximations depends on complexity considerations and will be addressed in the next section. The following lemma shows how equation (2.13) affects an approximation.

Lemma 2.4.1 If $d \geq 2$ and $y_i = \mathsf{RECIPROCAL}(\left\lfloor \frac{x}{2^{n-d}} \right\rfloor, d)$, then applying equation (2.13) gives y_{i+1} that satisfies

$$0 \leq \mathsf{RECIPROCAL}(\left\lfloor \frac{x}{2^{n-dk}} \right\rfloor, dk) - y_{i+1} \leq 2.$$

Furthermore, equation (2.13) can be evaluated by a logspace uniform circuit family with size $O(dk^7 \log dk \log\log dk)$ and depth $O(\log dk + \log k \log\log k)$.

Proof: This proof closely parallels the proof of theorem 2.2.1. Writing x' in two parts as $x' = x_1 2^{d(k-1)} + x_0$, the assumption on y_i states that $y_i = \mathsf{RECIPROCAL}(x_1, d)$, or that $x_1 y_i = 2^{2d} - s$, where $0 \le s < x_1$. This implies that $x' y_i = (x_1 2^{d(k-1)} + x_0) y_i = 2^{d(k+1)} - (2^{d(k-1)} s - x_0 y_i)$. To simplify notation, let $w = 2^{d(k+1)}$ and $z = (2^{d(k-1)} s - x_0 y_i)$, so $x' y_i = w - z$.

Let

$$d = y_i \sum_{j=0}^{2k-1} 2^{d(k+1)(2k-j-1)} (2^{d(k+1)} - x' y_i)^j = y_i \sum_{j=0}^{2k-1} w^{2k-j-1} z^j.$$

The quantity of interest is $x' y_{i+1}$, so first compute $x'd$ as

$$x'd = (w-z) \sum_{j=0}^{2k-1} w^{2k-j-1} z^j = \sum_{j=0}^{2k-1} w^{2k-j} z^j - \sum_{j=0}^{2k-1} w^{2k-j-1} z^{j+1} = w^{2k} - z^{2k}$$

$$= 2^{2dk(k+1)} - (2^{d(k-1)} s - x_0 y_i)^{2k}.$$

Dividing by 2^{2dk^2} gives

$$\frac{x'd}{2^{2dk^2}} = 2^{2dk} - \left[\frac{s}{2^d} - \frac{x_0 y_i}{2^{dk}} \right]^{2k}.$$

Since $\frac{s}{2^d}$ and $\frac{x_0 y_i}{2^{dk}}$ are both positive, we can bound

$$\left| \frac{s}{2^d} - \frac{x_0 y_i}{2^{dk}} \right| \le \max \left\{ \frac{s}{2^d}, \frac{x_0 y_i}{2^{dk}} \right\}. \tag{2.14}$$

The first of these terms is easy to bound: $\frac{s}{2^d} < 1$ since $s < x_1 < 2^d$. To bound the second term, notice that $y_i = \left\lfloor \frac{2^{2d}}{x_1} \right\rfloor \le \frac{2^{2d}}{x_1}$, so

$$\frac{x_0 y_i}{2^{dk}} \le \frac{x_0}{2^{d(k-2)} x_1} < \frac{2^{d(k-1)}}{2^{d(k-2)} 2^{d-1}} = 2.$$

Therefore, using equation (2.14),

$$\left(\frac{s}{2^d} - \frac{x_0 y_i}{2^{dk}} \right)^{2k} = \left(\left| \frac{s}{2^d} - \frac{x_0 y_i}{2^{dk}} \right| \right)^{2k} < 2^{2k}. \tag{2.15}$$

Since $d \ge 2$, this can be further bounded as

$$2^{2k} \le 2^{dk} < 2 \cdot 2^{dk-1} \le 2x'.$$

Notice that since the power $2k$ on the left hand side of equation (2.15) is even, the error term in equation (2.15) must be positive; in other words, $\frac{x'd}{2^{2dk^2}} \le 2^{2dk}$. It follows that $2^{2dk} - 2x' < \frac{x'd}{2^{2dk^2}} \le 2^{2dk}$.

The formula in equation (2.13) actually uses $\left\lfloor \frac{d}{2^{2dk^2}} \right\rfloor$, so

$$x' y_{i+1} = x' \left\lfloor \frac{d}{2^{2dk^2}} \right\rfloor > x' \left(\frac{d}{2^{2dk^2}} - 1 \right) = \frac{x'd}{2^{2dk^2}} - x' > 2^{2dk} - 3x'$$

If $\mathsf{RECIPROCAL}(x', dk) - y_{i+1} \ge 3$, then $x' y_{i+1} \le 2^{2dk} - 3x'$. As just shown, this is impossible, so $\mathsf{RECIPROCAL}(x', dk) - y_{i+1} \le 2$.

To evaluate equation (2.13), a circuit has to compute the jth power of $d(k+1)$ bit numbers, for $0 \le j < 2k$. Noticing that for each j the size of this powering is $O(dkj^5 \log dk \log\log dk)$

from corollary 2.3.6, the total size required to take all the powers necessary is asymptotically upper-bounded by

$$\sum_{j=0}^{2k-1} cdkj^5 \log dk \log\log dk = cdk \log dk \log\log dk \sum_{j=0}^{2k-1} j^5$$

$$< cdk^7 \log dk \log\log dk.$$

As the reader can easily verify, the cost of adding these powers, multiplying by y_i, and scaling back down are all negligible compared the cost of powering, so the size of the circuit to evaluate equation (2.13) is $O(dk^7 \log dk \log\log dk)$.

All the powers are done in parallel, each having depth at most $O(\log dk + \log k \log\log k)$, and every other operation (the large sum and the re-scaling) in the evaluation of equation (2.13) can be shown to have depth $O(\log dk)$; therefore, the total depth of evaluating equation (2.13) is $O(\log dk + \log k \log\log k)$. ∎

2.5 An Efficient Parallel Reciprocal Circuit

The results of the previous section can be used to design a parallel algorithm for finding reciprocals in depth $O(\log n \log\log n)$. In essence, lemma 2.4.1 says that an approximation to the reciprocal that is accurate to d bits can be extended to an accuracy of dk bits in $O(dk^7 \log dk \log\log dk)$ size and $O(\log dk + \log k \log\log k)$ depth.

To design a reciprocal algorithm, we need to come up with a sequence of approximation accuracies $d_1, d_2, d_3, ...$ such that after doing i approximation refinements, the result is accurate to d_i bits; eventually, all n bits should be known. In searching for criteria to design such a sequence, a desirable feature of parallel algorithms is that the work is spread out evenly across time. Looking at the form of the size bound from lemma 2.4.1, a good candidate is to set the size of each stage to $O(n \log n \log\log n)$. Setting $d_1 = 2$ (so two bit are known initially), the schedule then works out as

$$d_i k_i^7 \log d_i k_i \log\log d_i k_i \leq n \log n \log\log n$$

$$\Longrightarrow d_i \left(\frac{d_{i+1}}{d_i}\right)^7 \log d_{i+1} \log\log d_{i+1} \leq n \log n \log\log n$$

$$\Longrightarrow d_{i+1}^7 \log d_{i+1} \log\log d_{i+1} \leq n d_i^6 \log n \log\log n$$

Noticing that $d_{i+1} \leq n$ at all times (otherwise, the whole answer would be known!), the above inequality is satisfied with

$$d_{i+1} = n^{\frac{1}{7}} d_i^{\frac{6}{7}}.$$

Solving this recurrence (with the initial condition $d_1 = 2$) reveals that the sequence of accuracies is

$$d_i = 2n^{1-\left(\frac{6}{7}\right)^{i-1}}.$$

Unfortunately, this schedule does not produce just integers for accuracies (in fact, not necessarily even rational numbers!), so instead, let $m = \log n$ (recall that n is a power of 2 by assumption) and define the function

$$f(i) = \left\lfloor m \left(1 - \left(\frac{6}{7}\right)^{i-1}\right)\right\rfloor. \tag{2.16}$$

Algorithm RECIP2(x, n);
  $m \leftarrow \log n$;
  $d_1 \leftarrow 2$;
  $i \leftarrow 2$;
  if $(x \geq 3 \cdot 2^{n-2})$
    then begin
      $y_1 \leftarrow 5$;
      end;
    else begin
      $y_1 \leftarrow 8$;
      end;
  while $i \leq \left\lceil \frac{\log\log n}{\log \frac{7}{6}} \right\rceil$ **do begin**
    $t \leftarrow \left\lfloor m\left(1 - \left(\frac{6}{7}\right)^{i-1}\right)\right\rfloor$;
    $d_i \leftarrow 2^t$;
    $k_i \leftarrow \frac{d_i}{d_{i-1}}$;
    $x' \leftarrow \left\lfloor \frac{x}{2^{n-d_i}} \right\rfloor$;
    $y_{i+1} \leftarrow \left\lfloor \frac{y_i \sum_{j=0}^{2k_i-1} 2^{d_{i-1}(k_i+1)(k_i-j-1)}(2^{d_{i-1}(k_i+1)} - x'y_i)^j}{2^{2d_{i-1}k_i^2}} \right\rfloor$;
    for $j \leftarrow 1$ **downto** 0 **do**
      if $(x'(y_{i+1} + 2^j)) \leq 2^{2d_i}$
        then begin
          $y_{i+1} \leftarrow y_{i+1} + 2^j$;
          end;
    $i \leftarrow i + 1$;
  end;
  return (y_i);
end.

Figure 2.5: Algorithm RECIP2

Then the schedule can be defined by

$$d_i = 2^{f(i)}. \tag{2.17}$$

The result is the algorithm shown in figure 2.5.

Lemma 2.5.1 Algorithm RECIP2 shown in figure 2.5 correctly computes the reciprocal of an n-bit number, and can be realized with a circuit family of size $O(n \log n (\log\log n)^2)$ and depth $O(\log n \log\log n)$.

Proof: The fact that algorithm RECIP2 meets the schedule of equation (2.17) is a very simple proof by induction. The basis of the induction is trivial — the integer reciprocals of the two possible two-bit numbers are hard-wired into the algorithm. The induction step is proved by lemma 2.4.1 (notice the adjustment step in figure 2.5 that takes up the slack in possible error

from lemma 2.4.1). The final answer after $p = \left\lceil \frac{\log\log n}{\log \frac{7}{6}} \right\rceil + 1$ steps is d_p bits. Computing $f(p)$ (where f is defined in equation (2.16)) shows that $f(p) = m$; in other words, $d_p = 2^m = n$.

For the complexity, the size of stage i is $O(d_{i-1}k_i^7 \log d_{i-1}k_i \log\log d_{i-1}k_i)$ from lemma 2.4.1. Examining k_i, the order of approximation at stage i is $k_i = 2^{f(i)-f(i-1)}$. Focusing on the exponent,

$$f(i) - f(i-1) = \left\lceil m\left(\frac{6}{7}\right)^{i-2}\right\rceil - \left\lceil m\left(\frac{6}{7}\right)^{i-1}\right\rceil \leq \left\lceil m\left(\frac{6}{7}\right)^{i-2}\right\rceil - \frac{6}{7}\left\lceil m\left(\frac{6}{7}\right)^{i-2}\right\rceil + 1$$

$$= \frac{1}{7}\left\lceil m\left(\frac{6}{7}\right)^{i-2}\right\rceil + 1.$$

This means that

$$d_{i-1}k_i^7 \leq 2^{m-\left\lceil m\left(\frac{6}{7}\right)^{i-2}\right\rceil + \left\lceil m\left(\frac{6}{7}\right)^{i-2}\right\rceil + 7} = O(n).$$

Furthermore, since $d_{i-1}k_i < n$, the total size of stage i (regardless of i) is $O(n \log n \log\log n)$. Over all $O(\log\log n)$ stages, the total size of the circuit becomes $O(n \log n (\log\log n)^2)$.

The depth of stage i is $O(\log d_{i-1}k_i + \log k_i \log\log k_i)$ from lemma 2.4.1. Examining each term separately, the first term is $O(\log n)$ for all i, giving a total depth of $O(\log n \log\log n)$ over all stages. In the second term, $\log\log k_i$ can be bounded by $\log\log n$ to obtain a depth over all stages of

$$\sum_{i=2}^{p} \log k_i \log\log k_i \leq \log\log n \sum_{i=2}^{p} \log k_i = \log\log n \sum_{i=2}^{p} [f(i) - f(i-1)]$$

$$= \log\log n[f(p) - f(1)] = O(\log n \log\log n).$$

Combining both terms, the total depth can be seen to be $O(\log n \log\log n)$. ▮

The algorithm RECIP2 just described is certainly an efficient reciprocal algorithm (in terms of both size and depth), but it does not clearly specify a relationship between the complexity of multiplication and that of division. (The similarity of the size bound with the size of the Schönhage-Strassen multiplication algorithm is mere coincidence.) In this sense, algorithm RECIP1 was better, since the size was closely tied to the size of multiplication (in fact, the size was $O(M(n))$). Can the good qualities of both algorithms (the size bound of RECIP1 and the small depth of RECIP2) be combined? Fortunately, the answer to this question is yes.

The new algorithm is RECIP3 shown in figure 2.6; the value N is the number of bits of the *original* problem (before any reductions). The basic idea behind algorithm RECIP3 is to use RECIP2 to find a sufficiently accurate initial estimate of the integer reciprocal so that only $O(\log\log n)$ stages of second order approximations are needed.

Theorem 2.5.2 Algorithm RECIP3 in figure 2.6 correctly computes the reciprocal of an n-bit number, and can be realized with a circuit family of size $O(M(n))$ and depth $O(\log n \log\log n)$.

Proof: Algorithm RECIP3 is a hybrid of RECIP1 and RECIP2, and the correctness follows directly from the correctness of those algorithms (see theorem 2.2.1 and lemma 2.5.1).

After i steps of recursion in RECIP3, $n = \frac{N}{2^i}$, so it only takes $\log(\log^2 N) = O(\log\log N)$ steps of second order reduction before $n \leq \frac{N}{\log^2 N}$. The complexity analysis of the second order stages is identical to theorem 2.2.1, but with only $O(\log\log N)$ stages. In other words, the size

```
Algorithm RECIP3(x, n);
    if n ≤ N / log^2 N
      then begin
        y ← RECIP2(x, n);                {N is the size of the original problem.}
        end;
      else begin
        t ← RECIP3(⌊x / 2^{n/2}⌋, n/2);
        y ← ⌊(2^{(3/2)n+1} t − x t^2) / 2^n⌋;
        for i ← 3 downto 0 do
            if (x(y + 2^i) ≤ 2^{2n})
              then begin
                y ← y + 2^i;
                end;
        end;
    return (y);
end.
```

Figure 2.6: Algorithm RECIP3

of the second order approximations (not counting the call on RECIP2) is $O(M(N))$ and the depth is $O(\log N \log\log N)$.

The size of the call on RECIP2 is easily computed from lemma 2.5.1 to be

$$O(\frac{N}{\log^2 N} \log \frac{N}{\log^2 N} \log\log \frac{N}{\log^2 N}) = O(N),$$

and the depth is $O(\log N \log\log N)$.

Combining the complexity of the second order stages with the complexity of the call on RECIP2, the final result is that RECIP3 has size $O(M(N))$ and depth $O(\log N \log\log N)$. ∎

2.6 Chapter Summary

This chapter examined the most complex of the basic arithmetic problems — division. While it can be shown that division is at least as hard as the other arithmetic problems (addition, subtraction, and multiplication), it is unknown whether division is strictly *harder* than the other operations. In comparison with multiplication (the second hardest problem), the results of theorem 2.5.2 show that while it is still be possible that division is harder than multiplication, the difference is not all that great (in terms of asymptotic growth).

Chapter 3

Threshold Circuits

3.1 Introduction

The model of computation in the preceding chapter was the bounded fanin boolean circuit. When we allow gates to have an arbitrary number of inputs, we have the general class of boolean circuits. Obviously, any boolean gate with n inputs can be simulated by a circuit of bounded fanin boolean gates with $O(n)$ size and $O(\log n)$ depth, so we know that $AC^0 \subseteq NC^1$ (the definitions of the classes AC^0 and NC^1 are in section 1.3.1 of the introduction).

Much of the important lower bounds obtained in the past decade have been with the model of unbounded fanin boolean circuits, including several proofs that the parity function of n bits (denoted $PARITY^n$) cannot be computed by any boolean circuit family with polynomial size and constant depth (i.e., $PARITY^n$ is not in AC^0) [29, 68]. However, it is easy to see that $PARITY^n$ can be computed with a bounded fanin circuit family with $O(n)$ size and $O(\log n)$ depth, so this lower bound separates the classes AC^0 and NC^1 (i.e., AC^0 is a proper subset of NC^1).

In subsequent work, researchers have tried to increase the separation between AC^0 and NC^1 by looking for complexity classes that lie between AC^0 and NC^1; possible candidates for this class are ACC and TC^0 (it is known that $AC^0 \subset ACC \subseteq TC^0 \subseteq NC^1$). In this chapter, we are concerned with the constant depth threshold circuit model (as defined in section 1.3.1), which corresponds to the class TC^0.

For simplicity of presentation, we often assume that we can use $EXACT_k^n$ gates in threshold circuits, which are defined by

$$EXACT_k^n(x_1, ..., x_n) = \begin{cases} 1 & \text{if } \sum_{i=1}^{n} x_i = k \\ 0 & \text{otherwise} \end{cases}$$

The addition of these gates does not change the class of functions computable by constant-depth threshold circuits, as

$$EXACT_k^n(x_1, ..., x_n) = Th_k^n(x_1, ..., x_n) \ AND \ Th_{<k+1}^n(x_1, ..., x_n)$$

(recall that ANDs and ORs are just special cases of threshold gates, so can be easily computed). Using $EXACT_k^n$ gates, it is easy to see that $PARITY \in TC^0$:

$$PARITY^n = EXACT_1^n \ OR \ EXACT_3^n \ OR ... OR \ EXACT_{2\lceil n/2\rceil -1}^n$$

($2\lceil n/2 \rceil - 1$ is the greatest odd integer $\leq n$). This fact (in addition to the lower bound showing $PARITY \notin AC^0$ and the observation that $AC^0 \subseteq TC^0$) shows that $AC^0 \subset TC^0$ (meaning that AC^0 is a *proper* subset of TC^0).

In this chapter, we show that in fact, the threshold circuit is a very powerful model of computation. In the following section, we describe threshold circuit families with small size and constant depth for the following functions: iterated sum, discrete Fourier transform, integer multiplication, Chinese Remaindering, iterated product, integer powering, and integer division. For the last three of these (iterated product, integer powering, and integer division) we allow the circuits to be P-uniform — the others satisfy the stronger constraint of logspace uniformity.

3.2 Computing Arithmetic Using Threshold Circuits

All of the circuits presented in this section (with the exception of the integer powering circuit) have small size. By "small" we mean that for any constant $\epsilon > 0$, the circuit families have size $O(n^{1+\epsilon})$ for inputs of size n. This fact is not entirely evident from the lemma statements in this section, and confusion may arise from the fact that n does not always represent the total input length. The notation used is from commonly accepted conventions in the literature. For instance, in lemma 3.2.1, we describe a circuit for adding m numbers, each of n bits. In other words, the input length is actually nm bits, and the size complexity is therefore $O((nm)^{1+\epsilon})$.[1]

The circuit families described in the next subsection are all logspace uniform. The proof of uniformity is not included here — it is a straight-forward matter to show that these circuit families can be computed in $O(\log n)$ space.

3.2.1 Logspace Uniform Circuits

The problem of finding the sum of a set of integers is called the *iterated sum problem.* Pippenger has given a constant depth threshold circuit for multiplication, and the method used is the straight-forward reduction to iterated sum (i.e., the "grade-school method" of multiplication) [46]. Looking at just the iterated sum circuit, we see that Pippenger's circuit for adding m values, each of n bits, has size $O(nm^2)$ and depth $O(1)$. In the following lemma, we show how to produce a constant depth circuit for iterated sum with a smaller size.

Note: All of the following algorithms use a parallel version of divide-and-conquer. The most commonly used method of divide-and-conquer is to divide a size n problem into 2 problems of size $n/2$, resulting in a computation tree of depth $\Omega(\log n)$. In our circuits, we divide a size n problem into n^ϵ problems, each of size $n^{1-\epsilon}$; therefore, the resulting computation tree has depth $O(\frac{1}{\epsilon})$.

Lemma 3.2.1 Given any constant ϵ satisfying $0 < \epsilon \leq 1$, there exists a circuit for computing the iterated sum of m numbers, each of n bits (with $m \leq n^{O(1)}$), that has size $O(nm^{1+\epsilon})$ and depth $O(\frac{1}{\epsilon})$.

Proof: Since $m \leq n^{O(1)}$, it is trivial to show that the result of the iterated sum will have less than cn bits for some constant c.

To calculate the iterated sum, we build a computation tree where each node has $\lfloor m^\epsilon \rfloor$ children, and there are a total of m leaves. Placing the m input values at the leaves, computation proceeds toward the root of the tree with each internal node computing the sum of its children.

[1] The size is actually slightly better than this — but remember that the big-oh notation gives an *upper bound.*

After all computations, the root contains the sum of all m input values. It is easy to see that the desired tree has $O(m^{1-\epsilon})$ internal nodes, and a height of $O(\frac{1}{\epsilon})$. We use Pippenger's circuit at each internal node for a node size of $O(nm^{2\epsilon})$, so the total circuit size is $O(nm^{1+\epsilon})$. Since the depth of each node in the tree is constant, the total depth of the circuit is the same as the height of the tree, or $O(\frac{1}{\epsilon})$. ∎

Using this result, we can also construct small size circuits for discrete Fourier transform. Let DFT_M denote the discrete Fourier transform of an M-vector.

Lemma 3.2.2 Given any constant ϵ satisfying $0 < \epsilon \leq 1$, we can construct a circuit that computes $\mathrm{DFT}_M(a_0, a_1, ..., a_{M-1}) \bmod 2^N + 1$ (where M and N are both powers of 2 and $M \leq N$) that has size $O(\frac{1}{\epsilon}MN^{1+\epsilon})$ and depth $O(\frac{1}{\epsilon^2})$.

Proof: Since N and M are powers of 2, let $N = 2^n$ and $M = 2^m$. We will first show DFT_M exists in the ring Z_{2^N+1}. If we let $\omega = 2^{2N/M}$, then by taking $\omega^{M/2} = 2^N \equiv -1 \pmod{2^N+1}$ it is easy to see that ω is a principle Mth root of unity in Z_{2^N+1}. Also, since M is a power of 2, we know that M and 2^N+1 are relatively prime; therefore, M^{-1} exists in the ring. By these facts, the ring Z_{2^N+1} supports DFTs on M-vectors.

We introduce a new constant $\delta = \frac{\sqrt{1+4\epsilon}-1}{2}$. We will construct a computation tree as we did in lemma 3.2.1, but the fanout in this case will be $f = 2^{\lfloor m\delta \rfloor}$. Let $v_0, v_1, ..., v_{f-1}$ be the children of the root, and assume each child computes the $\frac{M}{f}$-vector $val(v_i) = (x_{i,0}, x_{i,1}, ..., x_{i,\frac{M}{f}-1}) = \mathrm{DFT}_{\frac{M}{f}}(a_i, a_{f+i}, ..., a_{M-f+i})$. Note that these vectors exist since ω^f is a principle $\frac{M}{f}$th root of unity, and $\left(\frac{M}{f}\right)^{-1}$ exists in Z_{2^N+1}. From these vectors we can produce the final result vector $(y_0, y_1, ..., y_{M-1}) = \mathrm{DFT}_M(a_0, a_1, ..., a_{M-1})$ by calculating

$$y_i = \sum_{j=0}^{f-1} \omega^j x_{j,i} \bmod 2^N + 1. \tag{3.1}$$

The proof of correctness for equation (3.1) is straight-forward, and is not included in this dissertation. Equation (3.1) is a simple modular iterated sum, since multiplication by powers of ω is just a bit shift (which costs nothing in the circuit model). This problem sub-division process is repeated down the tree until there are less than f values in each node. In general, if we label the root as level 0, we are calculating DFT_{M/f^i} at each node of level i from its f children. By using the iterated sum circuit of lemma 3.2.1 (the reduction mod 2^N+1 can be done after a non-modular iterated sum with a single subtraction), we can do this in size $O(\frac{M}{f^i}Nf^{1+\delta})$ for each node on level i. Since there are f^i nodes on level i, the total size for all nodes of that level is $O(MNf^{1+\delta})$. There are $O(\frac{1}{\delta})$ levels, so the total size of the circuit is $O(\frac{1}{\delta}MNf^{1+\delta})$. Since f is $O(N^\delta)$, the size can be written as $O(\frac{1}{\delta}MN^{1+\delta+\delta^2}) = O(\frac{1}{\epsilon}MN^{1+\epsilon})$. The depth of each level is $O(\frac{1}{\delta})$, so the total depth is $O(\frac{1}{\delta^2}) = O(\frac{1}{\epsilon^2})$. ∎

Using this circuit for discrete Fourier transform we can construct a constant depth multiplication circuit.

Lemma 3.2.3 Given any constant ϵ satisfying $0 < \epsilon \leq 1$, we can construct a circuit for multiplication of two N bit numbers that has size $O(\frac{1}{\epsilon}N^{1+\epsilon})$ and depth $O(\frac{1}{\epsilon^3})$.

Proof: The circuits that we construct are actually for multiplying two N-bit numbers modulo 2^N+1, where N is a power of 2. For exact (non-modular) multiplication of N' bit numbers, we

use the same circuit with $N = 2^{\lceil \log N' \rceil + 1}$. It is easy to show that this will produce the exact answer, and the size is only a constant factor larger than the modular circuit for N' bits.

We will denote the two input numbers by a and b, and their product by c. Since N is a power of 2, let $N = 2^n$, where n is an integer. Letting $m = 2^{\lfloor \epsilon n \rfloor}$, we can write any N-bit number a as an m-vector of blocks of $s = \frac{N}{m}$ bits, $a = (a_0, a_1, ..., a_{m-1})$; a_0 is the block of least significant bits. We can view this vector as a vector of polynomial coefficients, and define the polynomial $A(x) = \sum_{i=0}^{m-1} a_i x^i$. Notice that $A(2^s) = a$. Defining a polynomial for b in a similar way, the product polynomial $C(x) = A(x)B(x)$ will be such that $C(2^s) = c$.[2]

We use discrete Fourier transforms for the polynomial multiplication, and since the product polynomial will have degree $2m - 2$, we must calculate the transform of $2m$-vectors. (We could actually use wrapped convolutions on m-vectors, but nothing is gained over our asymptotic bounds.) Looking at the straight-forward method of polynomial multiplication, it is easy to bound $\max_{0 \le i < 2m} \{c_i\} < m2^{2s} < m(2^{2s} + 1)$. Since m and $2^{2s} + 1$ must be relatively prime, we can calculate the coefficients of $C(x)$ modulo both m and $2^{2s} + 1$, and combine these results for the final answer modulo $m(2^{2s} + 1)$. This ring includes as a subset the range of all possible results, so the result of these modular calculations is also the exact (non-modular) answer. The calculations modulo m can be done using lemma 3.2.1 (with a new constant $\frac{\epsilon}{3}$) and "grade-school multiplication", with a total size of $O(N^{1+\epsilon})$. We will now concentrate on the cost of the calculations modulo $2^{2s} + 1$.

We will again use a divide and conquer tree with the root labeled as level 0. The fanout of the tree is $2m$, and it should be obvious that on level i we are computing products of $s_i = N\left(\frac{2}{m}\right)^i$ bit numbers. The DFT_{2m} mod $(2^{2s_{i+1}} + 1)$ required at this level can be done in size $O(\frac{1}{\epsilon}2m(2s_{i+1})^{1+\epsilon})$ by lemma 3.2.2. On level i, there are $(2m)^i$ such DFTs to calculate, for a total size of $O(\frac{1}{\epsilon}4^{i+1}\left(\frac{2}{m}\right)^{(i+1)\epsilon}(2N)^{1+\epsilon})$. For sufficiently large N (and therefore m) we have $\left(\frac{m}{2}\right)^\epsilon > 8$, so the size of level i can be simplified to $O(\frac{1}{\epsilon}\left(\frac{1}{2}\right)^i N^{1+\epsilon})$. Summing over all levels we have a total size of $O(\frac{1}{\epsilon}N^{1+\epsilon})$.

The depth of each level in the tree is $O(\frac{1}{\epsilon^2})$ by lemma 3.2.2, so the total depth of our multiplication circuit is $O(\frac{1}{\epsilon^3})$. ▮

The problem of Chinese Remaindering is defined as follows: given m primes $p_1, p_2, ..., p_m$ (actually, they only have to be pairwise relatively prime) and an n bit number a, calculate the residue of a mod p_i for all $1 \le i \le m$. Conversely, given the residues modulo each of the primes $r_1, r_2, ..., r_m$, we would like to calculate the least positive a such that $a \equiv r_i \pmod{p_i}$ for all $1 \le i \le m$. We will only be interested in the case where $m \ge n$, and this fact simplifies the analysis.

Lemma 3.2.4 Given any constant ϵ satisfying $0 < \epsilon \le 1$, we can construct a circuit for Chinese Remaindering (in both directions) with size $O(\frac{1}{\epsilon^2}m^{1+\epsilon})$ and depth $O(\frac{1}{\epsilon^4})$.

Proof: When the multiplication circuit of lemma 3.2.3 is used in the Chinese Remaindering circuit of Hastad and Leighton (who designed the circuit for the bounded fanin boolean circuit model) [32], the result is exactly as stated in the lemma. The proof of the size and depth of the circuit is also analogous to that found in [32], and is not included here. ▮

[2] This is the same method that was used for integer powering in section 2.3.1.

3.2.2 P-uniform Circuits

In this section, we relax our restrictions on the uniformity of the circuit families; namely, we allow the circuit families to be P-uniform rather than logspace uniform. The following circuits all use tables of prime numbers, and we don't know how to compute these tables in $O(\log n)$ space.

The next problem we will look at is that of iterated product over a finite field. An iterated product of m values $a_1, a_2, ..., a_m$ over the field Z_p is defined to be $\prod_{i=1}^{m} a_i \bmod p$.

Lemma 3.2.5 Given any constant ϵ satisfying $0 < \epsilon \leq 1$, we can construct a circuit for iterated product of m numbers over the field Z_p with size $O(\frac{1}{\epsilon^2}(m \log p)^{1+\epsilon})$ and depth $O(\frac{1}{\epsilon^5})$.

Proof: Define a new constant $\delta = \frac{\epsilon}{5}$. We will perform the iterated product in a tree similar to the tree used for iterated sum. The tree will have fanout m^δ, and will perform an iterated product of m^δ values in Z_p at each node. The iterated product at each node is computed by performing a Chinese Remainder step, followed by calculating the iterated product over each of the smaller fields (using discrete logs, iterated sum, and powering), and finally a Chinese Remainering step to recover the full result. This produces the exact iterated product, and by multiplying by an $m^\delta \log p$ bit approximation to $1/p$, we can find the residue modulo p.

Let $p_1, p_2, ..., p_s$ be the prime moduli (in increasing order) used in the Chinese Remainder step. To insure that there is no loss of information, we must be sure that the product of the moduli is greater than the maximum possible result. Specifically, we must insure that $\prod_{i=1}^{s} p_i > p^{m^\delta}$. By basic number theoretic results, we can achieve this with $s \leq p_s = \Theta(m^\delta \log p)$ [31]. Obviously, $s > \log p$, so the condition of lemma 3.2.4 is satisfied, and we may construct the required Chinese Remaindering circuit with size $O(\frac{1}{\delta^2} m^{2\delta} (\log p)^{1+\delta})$ and depth $O(\frac{1}{\delta^4})$.

After performing the initial Chinese Remaindering step, we must perform an iterated product over each of the p_i. Since for all prime p_i, Z_{p_i} under multiplication is a cyclic group, there is a generator (not necessarily unique), call it g_i, that generates the entire group. Let $f_i(x) = g_i^x$; due to the fact that g_i is a generator, this function is one-to-one and onto for all $x \in Z_{p_i}^*$. We make tables for $f_i(x)$ and $f_i^{-1}(x)$, each of size $O(p_i \log p_i)$. Within a particular field, there must be tables for all m^δ input values, so the total size taken up by tables for p_i is $O(m^\delta p_i \log p_i)$.

The iterated product is calculated by taking the discrete logarithm of all input values ($f_i^{-1}(x)$), performing the iterated sum of these values modulo $p_i - 1$, then raising the generator to the resulting power in Z_{p_i} (this is just $f_i(x)$, above). This is a fairly common method of performing iterated product (see, for example, [7]). The only part we haven't examined here is the iterated sum. By lemma 3.2.1, we can calculate the exact iterated sum of m^δ numbers, each of $\log p_i$ bits, in size $O(m^{\delta+\delta^2} \log p_i)$ and depth $O(\frac{1}{\delta})$. With an $m^\delta \log p_i$ bit approximation to $(1/(p_i - 1))$, we can reduce this exact result to the result modulo $p_i - 1$ with a single multiplication. By lemma 3.2.3, this takes size $O(\frac{1}{\delta} m^{\delta+\delta^2} (\log p_i)^{1+\delta})$ and depth $O(\frac{1}{\delta^3})$. Therefore the total complexity of calculating the iterated product of m^δ numbers modulo p_i is $O(\frac{1}{\delta^2} m^{2\delta} p_i (\log p_i)^{1+\delta})$ size and $O(\frac{1}{\delta^4})$ depth.

Since this must be done for all s prime fields, the total size complexity of iterated product of m^δ numbers is s times the above value, plus the cost of Chinese Remaindering. Using the upper bounds for s and p_i, the total size is $O(\frac{1}{\delta} m^{5\delta} (\log p)^{1+2\delta})$, and the total depth is $O(\frac{1}{\delta^4})$. With an $m^\delta \log p$ bit approximation to $(1/p)$, we can reduce this result (the exact iterated product) modulo p. The complexity of this multiplication is negligible compared to the rest of the circuit.

All the above results are for one node of the tree. Summing over all nodes and rewriting in terms of ϵ, we get a total size of $O(\frac{1}{\delta^2}m^{1+5\delta}(\log p)^{1+2\delta}) = O(\frac{1}{\epsilon^2}(m\log p)^{1+\epsilon})$, and a total depth of $O(\frac{1}{\epsilon^5})$. ∎

The preceding lemma is used in relating threshold circuits to finite field circuits (section 3.3). The following lemma addresses the problem of integer powering — as in the preceding chapter, this is an important part of integer division.

Lemma 3.2.6 Given any constant ϵ satisfying $0 < \epsilon \leq 1$, we can construct a threshold circuit family that computes the mth power of an n-bit number (where $m \leq n$) with $O(\frac{1}{\epsilon^2}n^{1+\epsilon}m^{2+\epsilon})$ size and $O(\frac{1}{\epsilon^5})$ depth.

Proof: Let x represent the n-bit input; the desired output of the circuit is x^m. Notice that since $x < 2^n$, we can bound the output by $x^m < 2^{nm}$. We use a sequence of primes $p_1 < p_2 < \cdots < p_s$ as in the preceding proof, and we want to guarantee that $\prod_{i=1}^{s} p_i > 2^{nm}$. This is accomplished with $s < p_s = \Theta(nm)$ [31], and by lemma 3.2.4 we can compute $x \bmod p_i$ for all s primes in $O(\frac{1}{\epsilon^2}(nm)^{1+\epsilon})$ size and $O(\frac{1}{\epsilon^4})$ depth.

After the Chinese Remaindering is performed, we want to compute $x^m \bmod p_i$ for each of the s primes. By lemma 3.2.5, this can be done for prime p_i in $O(\frac{1}{\epsilon^2}(m\log p_i)^{1+\epsilon})$ size and $O(\frac{1}{\epsilon^5})$ depth. Since $\log p_i = O(\log n)$ for all p_i, the total size for computing $x^m \bmod p_i$ for all s primes is

$$O(\frac{1}{\epsilon^2}n(\log n)^{1+\epsilon}m^{2+\epsilon}) = O(\frac{1}{\epsilon^2}n^{1+\epsilon}m^{2+\epsilon}).$$

Now simply using another Chinese Remainder circuit gives the completed powering circuit. Clearly, both the size and depth are dominated by the middle section of the circuit (i.e., computing $x^m \bmod p_i$ for $p_1, p_2, ..., p_s$). This gives the complexity bounds stated in the lemma. ∎

We are now ready to describe a small-size threshold circuit family that computes integer reciprocals. Of course, the circuit family is only P-uniform, since it relies on the P-uniform circuit for powering.

Theorem 3.2.7 Given any ϵ satisfying $0 < \epsilon \leq 1$, we can construct a threshold circuit family that computes the integer reciprocal of n-bit numbers with $O(\frac{1}{\epsilon^3}n^{1+\epsilon})$ size and $O(\frac{1}{\epsilon^5})$ depth.

Proof: The circuit described in this proof manipulates fixed-point binary approximations to real numbers. The first step of the circuit is to shift the input so that it represents a fixed-point number in the range $[\frac{1}{2}, 1)$. This is essentially the same circuit that was used in chapter 2 to shift the input so that the most significant bit is set; using unbounded fanin gates, this circuit has $O(n)$ size and $O(1)$ depth.

Let x represent this shifted number, and let $u = 1 - x$ (so $0 < u \leq \frac{1}{2}$). We also define a new constant $\delta = \sqrt{4+\epsilon} - 2$. We use the reciprocal equation from Melhorn and Preparata [40]; namely, we use the fact that the following formula gives an n-bit approximation to the reciprocal of x (with $r = \lceil n^\delta \rceil$ and $k = \lceil \frac{1}{\delta} \rceil$):

$$\begin{aligned}(1+u+u^2+\cdots+u^{r-1})(1+u^r+u^{2r}+\cdots+u^{(r-1)r})\cdots \\ \cdots(1+u^{r^{k-1}}+u^{2r^{k-1}}+\cdots+u^{(r-1)r^{k-1}}).\end{aligned} \tag{3.2}$$

We compute each factor in sequence. Specifically, first compute the set $\{u, u^2, u^3, ..., u^r\}$ from u. Then from u^r, we can compute $\{u^r, u^{2r}, u^{3r}, ..., u^{r^2}\}$. In general, we use u^{r^i} to compute

$\{u^{r^i}, u^{2r^i}, ..., u^{r^{i+1}}\}$. There are a total of k such stages, so if we let $S(i,r)$ and $D(i,r)$ denote the size and depth of the ith stage, then the total size of computing all powers of u required in equation (3.2) is

$$\sum_{i=1}^{k} S(i,r),$$

and the total depth is

$$\sum_{i=1}^{k} D(i,r).$$

In [40] it was shown that for the reciprocal problem, we can use an $n+\log(12k)$ bit approximation to each u^{r^i} in the above computations, and the result will still be an n-bit approximation to $1/x$. In other words, we only need to take r powers of $n+\log(12k) = O(n)$ bit numbers at each stage. By lemma 3.2.6 we can now bound

$$S(i,r) = O(\frac{1}{\delta^2} n^{1+\delta} r^{3+\delta}) = O(\frac{1}{\delta^2} n^{1+4\delta+\delta^2}),$$

so the total size of computing all powers of u needed in equation (3.2) is $O(\frac{1}{\delta^3} n^{1+4\delta+\delta^2})$; in addition, it is easy to see that the total depth is $O(\frac{1}{\delta^5})$.

Computing each factor of equation (3.2) is now a simple iterated sum. By lemma 3.2.1 this can be done for all factors with a total size of $O(knr^{1+\delta}) = O(\frac{1}{\delta} n^{1+\delta+\delta^2})$, and a total depth of $O(\frac{1}{\delta})$. Clearly, the complexity of this stage is insignificant when compared to the complexity of the first stage (above).

Finally, we need to multiply all k factors together. Even if we do all k multiplications sequentially[3], the total size (see lemma 3.2.3) is $O(\frac{1}{\delta^2} n^{1+\delta})$, and the total depth is $O(\frac{1}{\delta^4})$. Once again this complexity is dominated by the first stage. Notice how, as in chapter 2, the complexity of division is most strongly affected by the complexity of powering.

Using the definition of δ, we see that the total size of our integer reciprocal circuit is

$$O(\frac{1}{\delta^3} n^{1+4\delta+\delta^2}) = O(\frac{1}{\epsilon^3} n^{1+\epsilon}),$$

and the total depth is

$$O(\frac{1}{\delta^5}) = O(\frac{1}{\epsilon^5}). \quad \blacksquare$$

3.3 Relation to Finite Field Circuits

In this section we relate the power of threshold circuits to the class of arithmetic circuits over a finite field. Finite field circuits differ from the previous models used in this dissertation in one major way — this is the first model that does not use the boolean value domain. In fact, the value domain changes depending on the size of the input to the circuit. Let $p(n)$ be a function that maps input lengths to prime numbers — the value domain for the circuit with input size n is the field $\mathbf{Z}_{p(n)}$. The function basis (i.e., the gates of the circuit) is the set of iterated sum and iterated product functions over $\mathbf{Z}_{p(n)}$.

We relate the power of these finite field circuits to the power of threshold circuits by giving simulations of finite field circuits with threshold circuits, and vice-versa. These simulations are important because they give some algebraic structure to the class of functions computable by

[3]A more elegant solution would be to use a depth $\log k$ tree of multiplications.

threshold circuits — this could be an important first step to proving lower bounds for threshold circuits.

To simplify the statement of our results, we introduce some new notation. In particular, we use $\mathsf{Threshold}(S(n), D(n))$ to refer to the class of functions computable by threshold circuits with size $O(S(n))$ and depth $O(D(n))$. Similarly, let $\mathsf{FiniteField}(p(n), S(n), D(n))$ be the class of functions computable by finite field circuits with size $O(S(n))$ and depth $O(D(n))$, where $p(n)$ is the function mapping input lengths to prime moduli.

3.3.1 Simulating Finite Field Circuits

The results of this section have been essentially proved in section 3.2. The simulation is summed up in the following theorem. Note that the size of the binary input to the simulating threshold circuit is $n \log p(n)$, so there is very little size increase in the simulation.

Theorem 3.3.1 For any constant ϵ satisfying $0 < \epsilon \leq 1$,

$$\mathsf{FiniteField}(p(n), S(n), D(n)) \subseteq \mathsf{Threshold}((S(n) \log p(n))^{1+\epsilon}, D(n)).$$

Proof: This theorem is essentially proved in lemmas 3.2.1 and 3.2.5. In this simulation, we use binary representations, each with $O(\log p(n))$ bits, to represent elements of $\mathbf{Z}_{p(n)}$. When constructing a threshold circuit that simulates an n input finite field circuit (with modulus function $p(n)$ and size $S(n)$), we first compute a $\log S(n) + \log p(n)$ bit approximation to $1/p(n)$. This value is hard-wired into the threshold circuit for easy access by different parts of the threshold circuit.

Now consider a single gate of the finite field circuit, where the fanin of the gate is f. If the gate is an iterated sum gate, then we first perform an exact addition (as in lemma 3.2.1) with a threshold circuit of size $O(f^{1+\epsilon} \log p(n))$ and depth $O(\frac{1}{\epsilon})$. This exact sum is at most $fp(n)$, so with a $\log fp(n)$ bit approximation to $1/p(n)$, we can perform the reduction modulo $p(n)$. This approximation is available from the pre-computed value of $1/p(n)$, described above (note that $\log S(n) + \log p(n) \geq \log fp(n)$). The reduction requires two multiplications and an addition, with a total size of $O(\frac{1}{\epsilon}(\log fp(n))^{1+\epsilon})$ and a total depth of $O(\frac{1}{\epsilon^3})$ (see lemma 3.2.3). If the gate is an iterated product gate, then by lemma 3.2.5 we can simulate it with a threshold circuit of size $O(\frac{1}{\epsilon^2}(f \log p(n))^{1+\epsilon})$ and depth $O(\frac{1}{\epsilon^5})$. In either case (iterated sum or product), we can simulate any fanin f gate of the finite field circuit by a threshold circuit of size $O(\frac{1}{\epsilon^2}(f \log p(n))^{1+\epsilon})$ and depth $O(\frac{1}{\epsilon^5})$.

Now consider the entire finite field circuit, and let f_i denote the fanin of gate i. The size of the finite field circuit can be written as

$$S(n) = \sum_i f_i.$$

Therefore, the size of the complete threshold circuit that simulates each gate of the finite field circuit is at most (for some constant c)

$$\sum_i (c\frac{1}{\epsilon^2}(f_i \log p(n))^{1+\epsilon}) = c\frac{1}{\epsilon^2}(\log p(n))^{1+\epsilon} \sum_i f_i^{1+\epsilon} \leq c\frac{1}{\epsilon^2}(\log p(n))^{1+\epsilon} S(n)^{1+\epsilon}.$$

Ignoring the $\frac{1}{\epsilon^2}$ factor (since it is a constant), the size of the resulting threshold circuit is $O((S(n) \log p(n))^{1+\epsilon})$. Furthermore, it is easy to see that the total depth is $O(\frac{1}{\epsilon^5} D(n)) = O(D(n))$. ∎

3.3.2 Simulating Threshold Circuits

In this section, we examine the reverse simulation: given a threshold circuit, how can we simulate the function using a finite field circuit? The simulation uses interpolating polynomials over $\mathbf{Z}_{p(n)}$, as described in the following theorem.

Theorem 3.3.2 For any function $p(n)$ that maps only to prime numbers and satisfies $S(n) \leq p(n)$ for all n,

$$\mathsf{Threshold}(S(n), D(n)) \subseteq \mathsf{FiniteField}(p(n), S(n)\log S(n), D(n)).$$

Proof: We use the following property of the field $\mathbf{Z}_{p(n)}$:

> Given any ordered list of $k+1$ values $v_0, v_1, ..., v_k$, there exists a degree k interpolating polynomial $g(x)$ over $\mathbf{Z}_{p(n)}$ such that $g(i) \equiv v_i \pmod{p(n)}$ for $i = 0, 1, ..., k$.

For such a polynomial, let $g(x) = \sum_{i=0}^{k} c_i x^i$. Notice that with a finite field circuit, we can compute (for input x) the values x^{2^i} for $i = 0, 1, ..., \lfloor \log k \rfloor$ with a circuit of size $O(k)$ and depth 1. Furthermore, using these values we can construct a circuit that computes x^i for all $i = 0, 1, ..., k$ which has size $O(k \log k)$ and depth 1 (we simply multiply together the appropriate powers x^{2^i}). Now it is a simple matter to evaluate the polynomial $g(x)$ — the total size of the circuit is $O(k \log k)$, and the total depth is 3.

Now consider a single threshold gate of the threshold circuit we are simulating (assume it is a $Th_k^f(x_1, ..., x_f)$ gate). We use the identity element of $\mathbf{Z}_{p(n)}$ under addition (1) to represent boolean 1, and the identity element under multiplication (0) to represent boolean 0.[4] Since $p(n) \geq S(n)$ (by a condition of the theorem), it is also true that $p(n) \geq f$. In other words, $\sum_{i=1}^{f} x_i$ has an unambiguous representation in $\mathbf{Z}_{p(n)}$. To simulate the threshold gate, we first compute the sum of the inputs, and then evaluate the interpolating polynomial for the threshold function. This takes total size $O(f \log f)$ and depth $O(1)$.

Doing this for every threshold gate in the circuit gives the results stated in the theorem — a finite field circuit with $O(S(n) \log S(n))$ size and $O(D(n))$ depth. ▮

3.3.3 Combined Results

In this section, we combine the results of the two preceding sections to show an equivalence between threshold circuits and finite field circuits. First, we define a more general class of functions computable with finite field circuits by removing the dependence on a specific function $p(n)$.

Definition 3.3.3 The class of functions $\mathsf{FiniteField}(S(n), D(n))$ (notice that $p(n)$ is not specified) is defined as

$$\mathsf{FiniteField}(S(n), D(n)) = \bigcup_{\substack{\text{Valid } p(n) \\ \text{with } p(n) \leq 2^{S(n)}}} \mathsf{FiniteField}(p(n), S(n), D(n)).$$

By "valid $p(n)$" we mean any function $p(n)$ whose range is contained in the set of prime numbers.

[4] Actually, any two elements of $\mathbf{Z}_{p(n)}$ can be used to represent the boolean values. The proof above directly applies by first shifting the representatives to 0 or 1.

This definition simply means that for a function $f(x)$ to be a member of the complexity class $\mathsf{FiniteField}(S(n), D(n))$, there must be *some* relatively small modulus function $p(n)$ such that $f(x) \in \mathsf{FiniteField}(p(n), S(n), D(n))$. Now we can relate the power of threshold circuits and finite field circuits.

Theorem 3.3.4 For any size function $S(n)$ and depth function $D(n)$,

$$\bigcup_{k\geq 1} \mathsf{Threshold}(S(n)^k, D(n)) = \bigcup_{k\geq 1} \mathsf{FiniteField}(S(n)^k, D(n)).$$

Proof: By elementary number theory, for every n there exists a prime number $p(n)$ such that $S(n) \leq p(n) \leq 2S(n)$. This defines the modulus function for theorem 3.3.2 which, combined with theorem 3.3.1, proves the theorem. ∎

The most important consequence of this theorem is the following characterization of TC^0.

Corollary 3.3.5

$$TC^0 = \bigcup_{k\geq 1} \mathsf{FiniteField}(n^k, 1)$$

In other words, the class of all functions computable by polynomial-size constant-depth threshold circuits is exactly the same as the class of functions computable by polynomial-size constant-depth finite field circuits. This shows that the class TC^0 has a surprisingly strong algebraic characterization, which might be useful in proving a separation (or equivalence) between TC^0 and NC^1.

It should be noted that NC^1 also has an algebraic description, where the power of NC^1 is characterized by the problem of iterated product over the alternating group A_5 [6]. Thus, comparing the powers of TC^0 and NC^1 might be possible by looking at only these algebraic characterizations. The exact relationship between TC^0 and NC^1 remains an open problem at the time of this dissertation.

3.4 Chapter Summary

In this chapter, we have shown that the threshold circuit is a very powerful model of computation, with many arithmetic functions computable in constant-depth and sub-quadratic size (lemmas 3.2.1 through 3.2.6). In fact, integer division is a problem which seems quite difficult for bounded fanin boolean circuits, but has a P-uniform threshold circuit family with size $O(n^{1+\epsilon})$ and constant depth (theorem 3.2.7).

It was also shown that the power of threshold circuits has a concise algebraic description. For instance, TC^0 is exactly the class of functions computable by constant-depth polynomial-size finite field circuits (corollary 3.3.5). This permits us to use the great wealth of knowledge about finite fields when reasoning about threshold circuits. However, it remains an important open question as to whether or not TC^0 is a proper subset of NC^1.

Chapter 4

Motion Planning in Cooperative Environments

4.1 Introduction

In this chapter, we begin presenting results for algorithmic motion planning problems. As stated in the introduction, the basic foundation for motion planning comes from geometric problems, such as finding a path for an object (robot) which avoids a set of obstacles (known as the "Piano Movers' Problem") [61]. Even for the case of the robot being a simple point, finding the *shortest* path through a set of objects can be very difficult in three dimensions, but a fully polynomial approximation algorithm was given by Papadimitriou [43]. Unfortunately, these problems do not take into account the physical limitations of a real robot (for instance, the shortest path between two points will usually involve an instantaneous change in the direction of motion); furthermore, it is much more important to consider a path that takes the shortest *time* rather than covering the shortest distance. With this in mind, the problem of kinodynamic motion planning addresses these real-world issues.

Kinodynamic planning extends kinematic planning (avoiding a set of static obstacles) by including dynamics (or dynamical) constraints, such as dynamics laws (e.g. $\mathbf{f} = m\mathbf{a}$) and dynamics bounds (a maximum allowable acceleration $a_{\max}$ and velocity $v_{\max}$). In addition to simply finding a trajectory between a start state and a goal state (a state consists of both a position and a velocity), it is desirable to find the *optimal* trajectory, i.e., the trajectory that takes the least amount of time. Dynamics bounds are given by bounding the norm of the vectors that represent velocity and acceleration. As finding optimal trajectories is computationally intensive, practical algorithms must focus on *approximately optimal* trajectories; specifically, an approximation algorithm will find a trajectory connecting the start state and goal state that requires time only slightly greater than the time required by the optimal trajectory. Previously, an approximation algorithm was known when the dynamics bounds are stated in terms of the L_∞ norm [12]; however, while such a case is easier to show (due to the independence of the dimensions), it relies on somewhat artificially imposed properties, such as the orientation of the coordinate axes.

In this chapter, we present an approximation algorithm that uses the L_2 norm for dynamics bounds; our results parallel those of Canny, Donald, Reif, and Xavier [12], but the proof techniques are very different. In independent work concurrent with the research presented in this chapter, Donald and Xavier have also developed an approximation algorithm with dynamics bounds stated in terms of L_2 norms [24].

Optimal kinodynamic planning seems to be very hard in practical situations; the only exact solutions to the optimal kinodynamic planning problem are for one or two dimensions. In fact, in three dimensions (or more) finding a minimum distance path has been shown to be NP-hard [14], and this proof can be used to show that finding the exact solution for kinodynamic planning in ≥ 3 dimensions is NP-hard. However, as with many NP-hard problems, it is possible to find an approximately optimal solution in polynomial time; as we show here, the goodness of the approximation can be bounded by a proven scalar multiple. In other words, if the optimal solution is a robot trajectory that takes time T, then for any given $\epsilon > 0$ we can find a solution that takes time at most $(1+\epsilon)T$ by a search algorithm whose running time is polynomial both in the complexity of the environment and in $\frac{1}{\epsilon}$.

In real life there are additional problems to address (such as external forces) that we do not address in this chapter. One additional real-world property that we *do* address is the inability of real robots to navigate accurately at high speeds. To this end, we use the notion of "safe" and "also-safe" trajectories introduced in [12]; basically, this concept uses an affine mapping from speed (i.e., magnitude of velocity) to distance that bounds how close the robot may be to an obstacle. Exact definitions of "safe" and "also-safe" trajectories can be found in section 4.5. The robot model that we use is simply a point robot with unit mass; non-point robots can be handled easily by "growing" the obstacles to reflect the shape of the robot. It should be noted that the approximation algorithm we present is extremely simple; the complex equations found in this chapter are used exclusively for proving the correctness of the algorithm.

4.2 Preliminaries

4.2.1 Definitions and Terminology

Before starting the technical material, we will present the definitions and terminology that are used in this chapter and the next. All vector variables will be typeset in boldface, to separate them from scalars which are typeset in standard math italics. For example, $\mathbf{v}$ is a vector (of reals), and t is a scalar real. First and second derivatives are denoted by superscripted dots as in standard control theory literature. For example, if $\mathbf{p}(t)$ is a (twice differentiable) function, then $\dot{\mathbf{p}}(t)$ is its first derivative, and $\ddot{\mathbf{p}}(t)$ is its second derivative.

Consider a point traveling through d-dimensional Euclidean space. By a trajectory Γ, we mean both the velocity and position of the path that the point takes. By a point on a trajectory, we mean both the position and velocity at a particular time; for example, the endpoints can be given by $(\mathbf{p_0}, \mathbf{v_0})$ and $(\mathbf{p_1}, \mathbf{v_1})$, where $\mathbf{p_0}$ and $\mathbf{p_1}$ are the starting and ending positions, respectively, and $\mathbf{v_0}$ and $\mathbf{v_1}$ are the starting and ending velocities. If trajectory Γ takes time T, we say that Γ is a time T trajectory. For a subscripted trajectory Γ_r, we denote the position at time t by $\mathbf{p}_r(t)$, the velocity by $\dot{\mathbf{p}}_r(t)$, and the acceleration by $\ddot{\mathbf{p}}_r(t)$. The change (from time 0) in any of these functions is represented by a delta prefix; for example, the change in position is $\Delta\mathbf{p}_r(t) = \mathbf{p}_r(t) - \mathbf{p}_r(0)$. Similar definitions hold for $\Delta\dot{\mathbf{p}}_r(t)$ and $\Delta\ddot{\mathbf{p}}_r(t)$. The environment is a set of polyhedral obstacles in d-dimensional space, where d is considered to be a small constant.

The 2-norm of a vector $\mathbf{v}$ is written as $\|\mathbf{v}\|_2$, and the infinity norm is $\|\mathbf{v}\|_\infty$. Hereafter, if we write simply $\|\mathbf{v}\|$ without a subscript, the 2-norm should be understood.

The set of obstacles in the environment is represented by $\mathcal{E}$. All obstacles are polyhedral and require a total of n bits to encode. Furthermore, it is assumed that the space in which the robot may move is bounded by a ball of diameter D.

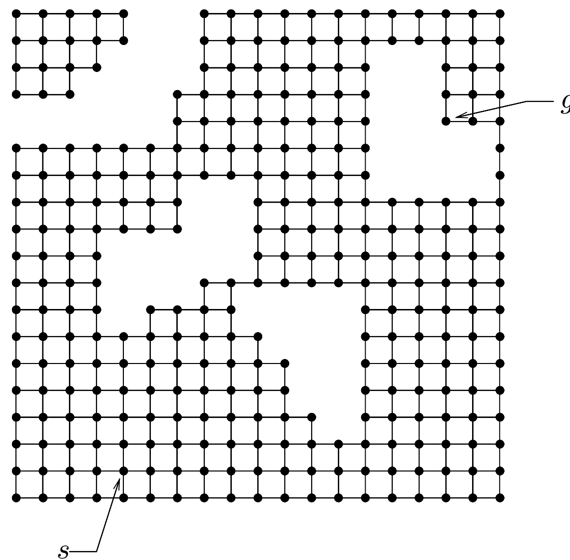


Figure 4.1: An example grid search problem

4.2.2 Outline of Algorithm and Proof

Consider the following discrete search problem: we are given a subgraph of a d-dimensional grid-graph; in other words, a grid-graph with some vertices missing. There are two distinguished vertices s and g, and we want to know if there is a path from s to g (an example in two dimensions is shown if figure 4.1). This problem is easy to solve using depth-first search on the graph; a minimum distance path from s to g can be found (if a path exists) by using breadth-first search.

The problems we are interested in for this chapter are similar, but involve searching a continuous space. By a *discretization* of the environment with grid-length g, we are referring to a graph constructed from the environment as follows. First, construct a graph with nodes for each point $(i_1 g, i_2 g, ..., i_d g)$ in the environment, where each i_j is an integer; since the environment is bounded by a ball of diameter D, the graph is finite. Edges are added between neighboring vertices to form a grid-graph. Finally, the vertices that lie inside any obstacle are removed from the graph.

The graph of figure 4.1 is such a graph — the missing parts of the grid correspond to obstacles. Simple reachability problems can be answered using this graph: by making g small enough we can guarantee that there exists a continuous path in the environment if and only if there exists a path on the constructed grid-graph, and a breadth-first search on the grid-graph gives an approximately minimum distance path in the continuous environment. Unfortunately, even this simple reachability problem requires a grid whose size grows exponentially with the algebraic complexity of the environment. We use a variant of this strategy that requires only a polynomial size graph (described fully in section 4.3) to solve approximate kinodynamic planning.

The proof of the correctness of our algorithm is based on a *tracking theorem* (theorem 4.4.8). This theorem states that for *any* continuous trajectory Γ_e, there exists a trajectory Γ_a that travels only between grid-points of our discretization and is always close (in both position

and velocity) to the continuous trajectory Γ_e. Thus, the *minimum time* continuous trajectory has a corresponding approximating trajectory in the constructed grid, and this approximating trajectory can be found by simple breadth-first search. Since any discovered trajectory between grid-points is also a valid continuous trajectory, we never find an invalid trajectory, and the correctness of the approximation algorithm follows.

The proof of the tracking theorem is rather involved, so we outline it here. First we show that any continuous trajectory can be stretched in time so that it takes slightly longer, but the new trajectory meets a smaller acceleration bound (lemma 4.4.3). Thus, when approximating the slowed-down continuous trajectory, the additional acceleration available to the approximating trajectory can be used to reach a grid-point that is close to the continuous trajectory. Unfortunately, there may still be some position error build-up while approximating the continuous trajectory, so we alternate phases of approximating with phases of error correction. A slightly modified continuous trajectory that doesn't change velocity during the error correction phase is shown to exist (lemma 4.4.4), and this trajectory is used in the approximating phases instead of the original one. By making the approximating and error correcting phases short enough, we show that the constructed trajectory is still a good approximation of the original continuous trajectory, which completes the proof of the tracking theorem.

4.3 Constructing a Grid

For our kinodynamic planning approximation, we build a grid of points in state space, rather than just in the position as outlined above. The approximation proceeds in time steps of length τ as follows: At all times $i\tau$ (i an integer), the velocity that is desired at time $(i+1)\tau$ is chosen from the neighbors of the current state, and the trajectory in the time interval $(i\tau, (i+1)\tau)$ is a linear transition to the desired next velocity (i.e., constant acceleration). Notice that the position at time $i\tau$ and the selected velocity transition completely determine the position at time $(i+1)\tau$. For such a discrete step method, we must show that it is possible to stay reasonably close to an exact path by this method of moving between neighboring grid-points. Note that while we still refer to our discretization as a grid, it is *not* a regular grid-graph in position space — the actual structure is a grid-graph in *velocity space*, along with the positions that correspond to moves on this velocity grid.

Since we want to define a finite grid, at any time step there must be finitely many choices for the change in velocity over the next time interval. If we let $\mathbf{v}_1, \mathbf{v}_2, ..., \mathbf{v}_k$ be these vectors (called *choice vectors*), then for each vector $\mathbf{v}_i$ we can determine θ_i, the smallest angle between $\mathbf{v}_i$ and any other choice vector. Remember that these vectors are actually *change* in velocity vectors, so the velocity at time $(i+1)\tau$ is $\dot{\mathbf{p}}(i\tau) + \mathbf{v}_j$ for the chosen vector $\mathbf{v}_j$. We always include the zero vector ($\mathbf{0}$) in a set of choice vectors to denote that it is possible to stay at the current velocity during a time interval; thus the set of choice vectors referred to above is $V = \{\mathbf{0}, \mathbf{v}_1, \mathbf{v}_2, ..., \mathbf{v}_k\}$. We now argue that θ_i must vary with ϵ if we bound the 2-norm of the acceleration; this implies that the number of choice vectors must grow as ϵ decreases.

Assume that the angles do not vary with ϵ, and pick a particular non-zero θ_i. Let $\mathbf{v}_m$ be a choice vector that makes angle θ_i with $\mathbf{v}_i$. Consider a continuous path with maximum acceleration at an angle that exactly bisects the angle made by $\mathbf{v}_i$ and $\mathbf{v}_m$; it should be obvious that by making ϵ sufficiently small, the exact path taking time T simply outruns any path made up of choice vectors taking time $(1+\epsilon)T$. In other words, any approximating path will fall farther and farther behind the exact path. In particular, in two (or more) dimensions we can show that there needs to be $\Omega(\frac{1}{\epsilon})$ choice vectors to approximate within an ϵ factor of optimal.

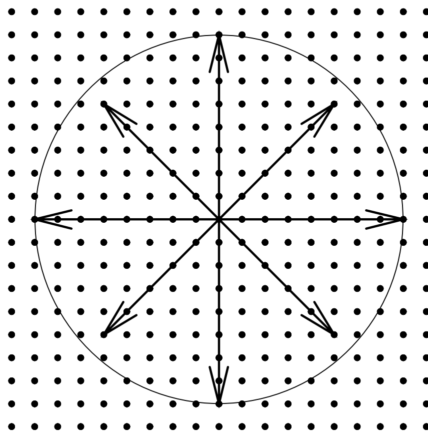


Figure 4.2: Possible choice vectors in two dimensions for $\epsilon = \frac{1}{2}$.

Now we examine how to vary the angle between choice vectors with ϵ. The first method that comes to mind is to simply use maximal acceleration vectors at angles that are evenly spaced (and varying with ϵ); unfortunately, this gives rise to a "grid" that grows exponentially with the number of time steps, and in fact does not even form a finite graph. The method we actually use is to superimpose a square grid on top of this set of choices, and then using parts of this grid with a new neighbor relationship, we have a grid that grows polynomially with the number of time steps. For a small enough square grid, we can track velocities closely; a more formal presentation of this follows.

Definition 4.3.1 A set of choice vectors $\{\mathbf{0}, \mathbf{v}_1, \mathbf{v}_2, ..., \mathbf{v}_k\}$ is called δ-dense $(0 < \delta < 1)$ if for *any* non-zero vector $\mathbf{v}$ there exists a non-zero choice vector $\mathbf{v}_i$ such that

$$\frac{\mathbf{v}_i \cdot \mathbf{v}}{\|\mathbf{v}_i\|\|\mathbf{v}\|} \geq \delta.$$

What this means geometrically is that given any vector $\mathbf{v}$, you can always find a choice vector $\mathbf{v}_i$ such that the angle between $\mathbf{v}$ and $\mathbf{v}_i$ is small (less than or equal to $\arccos\delta$).

The easiest way to obtain a δ-dense set of vectors is to space unit vectors evenly with respect to angles. As mentioned above, this is not good enough for our application, so we consider a square grid with small grid length. A set of "almost unit length" (i.e., within one grid length of unit length, but never more than unit length) choice vectors can be constructed using these grid-points while assuring that the set is δ-dense. A set of $(1 - \frac{\epsilon}{4(1+\epsilon)})$-dense choice vectors on a square grid with grid-length $\frac{\epsilon}{4}$ (exactly the conditions required by the following theorem) is illustrated in figure 4.2 for the specific case of two dimensions and $\epsilon = \frac{1}{2}$. The dots represent the points of the square grid, and the circle is a unit radius circle drawn for reference.

Theorem 4.3.2 For $0 < \epsilon \leq 1$, let $V = \{\mathbf{0}, \mathbf{v}_1, \mathbf{v}_2, ..., \mathbf{v}_k\}$ be a set of $(1 - \frac{\epsilon}{4(1+\epsilon)})$-dense choice vectors that are "almost unit length" (as defined above) on a square grid with grid-length $\frac{\epsilon}{4}$. Then for any vector $\mathbf{v}$ with $\|\mathbf{v}\| \leq 1 + \frac{1}{1+\epsilon}$, there is a choice vector $\mathbf{v}_c$ with $\|\mathbf{v} - \mathbf{v}_c\| \leq 1$.

Proof: Let $\mathbf{v}$ be any vector with $\|\mathbf{v}\| \leq 1 + \frac{1}{1+\epsilon}$. Since $V = \{\mathbf{0}, \mathbf{v}_1, \mathbf{v}_2, ..., \mathbf{v}_k\}$ is a set of $\left(1 - \frac{\epsilon}{4(1+\epsilon)}\right)$-dense choice vectors, there exists a $\mathbf{v}_c \in V$ such that

$$\mathbf{v} \cdot \mathbf{v}_c \geq \left(1 - \frac{\epsilon}{4(1+\epsilon)}\right) \|\mathbf{v}\|\|\mathbf{v}_c\|. \tag{4.1}$$

We are interested in finding $\|\mathbf{v} - \mathbf{v}_c\|$. A simple geometric identity states that

$$\|\mathbf{v} - \mathbf{v}_c\|^2 = \|\mathbf{v}\|^2 + \|\mathbf{v}_c\|^2 - 2\mathbf{v} \cdot \mathbf{v}_c = \|\mathbf{v}\|^2 + \|\mathbf{v}_c\|^2 - 2\|\mathbf{v}\|\|\mathbf{v}_c\| \cos\theta,$$

where θ is the angle between $\mathbf{v}$ and $\mathbf{v}_c$. Fixing $\|\mathbf{v}_c\|$ and θ and viewing the above equation as a polynomial in $\|\mathbf{v}\|$, differentiating with respect to $\|\mathbf{v}\|$ shows that the *minimum* value of $\|\mathbf{v} - \mathbf{v}_c\|^2$ occurs when $\|\mathbf{v}\| = \|\mathbf{v}_c\| \cos\theta$. For all $\|\mathbf{v}\| < \|\mathbf{v}_c\| \cos\theta$, the maximum value for $\|\mathbf{v} - \mathbf{v}_c\|^2$ occurs at the smallest possible value for $\|\mathbf{v}\|$; i.e., at $\|\mathbf{v}\| = 0$. When $\|\mathbf{v}\| = 0$, it is obvious that $\|\mathbf{v} - \mathbf{v}_c\| = \|\mathbf{v}_c\| \leq 1$.

It is also seen that for all $\|\mathbf{v}\| > \|\mathbf{v}_c\| \cos\theta$, the quantity $\|\mathbf{v}-\mathbf{v}_c\|^2$ is monotonically increasing, so the maximum value occurs at the largest allowable value for $\|\mathbf{v}\|$; in other words, when $\|\mathbf{v}\| = 1 + \frac{1}{1+\epsilon}$. Similar arguments show that $\|\mathbf{v} - \mathbf{v}_c\|^2$ is maximized when $\|\mathbf{v}_c\| = 1 - \frac{\epsilon}{4}$ and $\cos\theta = 1 - \frac{\epsilon}{4(1+\epsilon)}$. In other words, for all $\mathbf{v}$ such that $\|\mathbf{v}\| \leq 1 + \frac{1}{1+\epsilon}$, there exists a choice vector $\mathbf{v}_c$ such that

$$\|\mathbf{v} - \mathbf{v}_c\|^2 \leq \left(1 + \frac{1}{1+\epsilon}\right)^2 + \left(1 - \frac{\epsilon}{4}\right)^2 - 2\left(1 + \frac{1}{1+\epsilon}\right)\left(1 - \frac{\epsilon}{4}\right)\left(1 - \frac{\epsilon}{4(1+\epsilon)}\right).$$

Algebraic manipulation reveals that the right side of the above inequality is equivalent to

$$1 - \frac{\epsilon(8 + 3\epsilon - \epsilon^3)}{16(1+\epsilon)^2}.$$

In this form, it is obvious that for all valid ϵ (i.e., all ϵ satisfying $0 < \epsilon \leq 1$), $\|\mathbf{v} - \mathbf{v}_c\|^2 \leq 1$. This completes the proof of the theorem. ∎

This theorem is used to show that with a certain finite set of choice vectors for the change in velocity, any exact trajectory can be closely tracked using only velocity changes from the set of choice vectors; the direct application of this theorem can be found in the text following lemma 4.4.4.

To see how trajectories are constructed from a set of choice vectors, let τ denote the length of one discrete time interval. Consider a trajectory with an acceleration bound of a. The most that the velocity can change during one time interval is $a\tau$, so we consider this to be one "unit length"; it is obvious that theorem 4.3.2 applies using this as one unit, and this fact is made explicit in the following corollary.

Corollary 4.3.3 For $0 < \epsilon \leq 1$, let $V = \{\mathbf{0}, \mathbf{v}_1, \mathbf{v}_2, ..., \mathbf{v}_k\}$ be a set of $(1 - \frac{\epsilon}{4(1+\epsilon)})$-dense choice vectors that are "almost $a\tau$ length" on a square grid with grid length $\frac{\epsilon}{4}a\tau$. Then for any vector $\mathbf{v}$ with $\|\mathbf{v}\| \leq \left(1 + \frac{1}{1+\epsilon}\right) a\tau$, there is a choice vector $\mathbf{v}_c$ with $\|\mathbf{v} - \mathbf{v}_c\| \leq a\tau$.

Now consider a trajectory made up of N time intervals. Let $i : \{0, 1, ..., N-1\} \to \mathbf{Z}^+$ be an indexing function such that at the beginning of time interval t, we decide to use choice vector $\mathbf{v}_{i(t)}$. First, a preliminary lemma shows how the position component of a trajectory is affected by the schedule of choice vectors taken. The proof of the lemma is omitted, but is trivial; simply integrating over the velocity function defined by the indexing function gives the formula in the lemma. Notice that the velocity at any time $k\tau$ is given by $\dot{\mathbf{p}}(0) + \sum_{t=0}^{k-1} \mathbf{v}_{i(t)}$.

Lemma 4.3.4 If i is an indexing function as above, then the total change in position is given by

$$\Delta \mathbf{p}_a = \dot{\mathbf{p}}(0)N\tau + (N - \frac{1}{2})\tau \sum_{k=0}^{N-1} \mathbf{v}_{i(k)} - \tau \sum_{k=0}^{N-1} k\mathbf{v}_{i(k)}. \tag{4.2}$$

4.4 Tracking in the Absence of Obstacles

Before talking about trajectories that avoid obstacles, we must first show how paths can be constructed on our grid. To simplify this, arbitrary trajectories are shown to be easily approximated by a series of moves on the grid, with no obstacles in the environment.

The following lemma is stated in general terms, and will be used in several ways. Applications will be discussed after the proof of the lemma.

Lemma 4.4.1 Let $f : [0, T] \to \mathbf{R}$ be a continuous real-valued function on the closed interval $[0, T]$. If we know that $f(0) = f_0$, $f(T) = f_0 + \Delta f$, and that $|\frac{df(t)}{dt}| \le a$ for all $t \in [0, T]$, the following inequalities must hold:

$$f_0 T + \frac{\Delta f T}{2} + \frac{(\Delta f)^2}{4a} - \frac{aT^2}{4} \le \int_0^T f(t)dt \le f_0 T + \frac{\Delta f T}{2} - \frac{(\Delta f)^2}{4a} + \frac{aT^2}{4}$$

Proof: First we argue that for any function $f(t)$ satisfying the end-point and derivative constraints of the lemma, the following inequalities must hold for all times t in the interval $[0, T]$.

$$f(t) \le f_0 + at \tag{4.3}$$

$$f(t) \le f_0 + \Delta f + a(T - t) \tag{4.4}$$

Consider equation (4.3). If the inequality does not hold, then there exists a time t_1 such that $f(t_1) > f_0 + at_1$, and by the mean value theorem of derivatives there must be some time t_2 in the interval $[0, t_1]$ such that $f'(t_2) = \frac{f(t_1) - f_0}{t_1} > a$. This contradicts our bound on the derivative as stated in the lemma, so cannot be true; therefore, equation (4.3) must hold. The argument for equation (4.4) is similar.

Since any function that satisfies the constraints of the lemma must satisfy both upper bounds of equations (4.3) and (4.4), it must satisfy the least of the two at any particular time. Let $g_1(t) = f_0 + at$ and $g_2(t) = f_0 + \Delta f + a(T - t)$, and define $g(t) = \min\{g_1(t), g_2(t)\}$. A simple check of $g(t)$ shows that it satisfies the constraints of the lemma, and by the above argument must be the point-wise maximum of all valid functions.

Since $g(t)$ is the point-wise maximum of all valid functions, the definite integral of $g(t)$ over the interval $[0, T]$ must also be greater than that of any other valid function. Actually calculating this integral gives the upper bound stated in the lemma. The proof of the lower bound is similar. ▮

The most immediate and obvious result is stated in the following corollary.

Corollary 4.4.2 If we let Γ be a one dimensional time T trajectory from starting state $(p(0), \dot{p}(0))$ to goal state $(p(T), \dot{p}(T))$ that obeys acceleration bound a, then we can say that

$$p(T) \le p(0) + \dot{p}(0)T + \frac{\Delta\dot{p}(T)T}{2} - \frac{(\Delta\dot{p}(T))^2}{4a} + \frac{aT^2}{4}$$

and

$$p(T) \ge p(0) + \dot{p}(0)T + \frac{\Delta\dot{p}(T)T}{2} + \frac{(\Delta\dot{p}(T))^2}{4a} - \frac{aT^2}{4}.$$

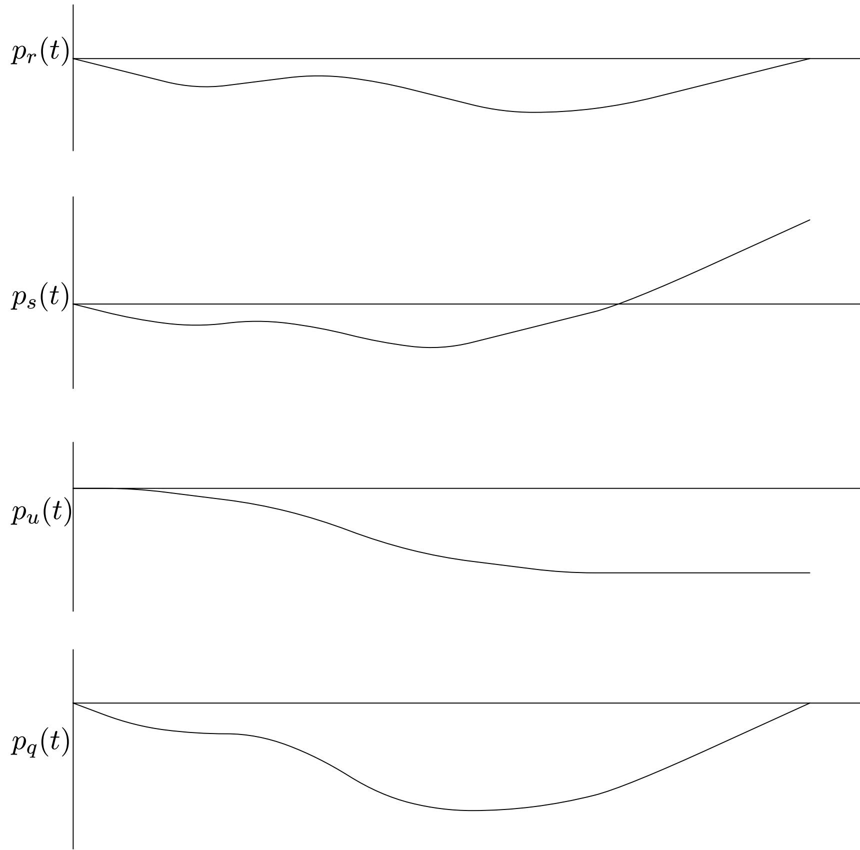


Figure 4.3: Position graphs for trajectories in lemma 4.4.4.

Further uses of lemma 4.4.1 will occur when we bound the norm of the integral of vector functions.

The following lemma explains how we can reduce the acceleration bound of a trajectory and still meet the same endpoints. This occurs with a corresponding increase in the time required by the trajectory. Henceforth, assume that whenever ϵ is mentioned, it satisfies $0 < \epsilon \leq 1$.

Lemma 4.4.3 Given a time T trajectory Γ_r from $(\mathbf{p}_r(0), \mathbf{0})$ to $(\mathbf{p}_r(T), \mathbf{0})$ with acceleration bound a, then there exists a trajectory Γ_q with acceleration bound $\frac{a}{(1+\epsilon)^2}$ and the same endpoints, but takes time $(1+\epsilon)T$.

Proof: Simply let $\ddot{\mathbf{p}}_q(t) = \ddot{\mathbf{p}}_r(\frac{t}{1+\epsilon})/(1+\epsilon)^2$ with $\dot{\mathbf{p}}_q(0) = 0$ and $\mathbf{p}_q(0) = \mathbf{p}_r(0)$. The verification that the ending conditions are met is now a simple calculus problem, and the details are omitted. ∎

The problem we must now overcome is that given the endpoints of a trajectory, in general we know very little about what happens between the endpoints. The next lemma is designed to solve this problem. Example trajectories as constructed by the lemma are shown in figures 4.3 and 4.4. These examples are one dimensional trajectories, and the horizontal axis represents time.

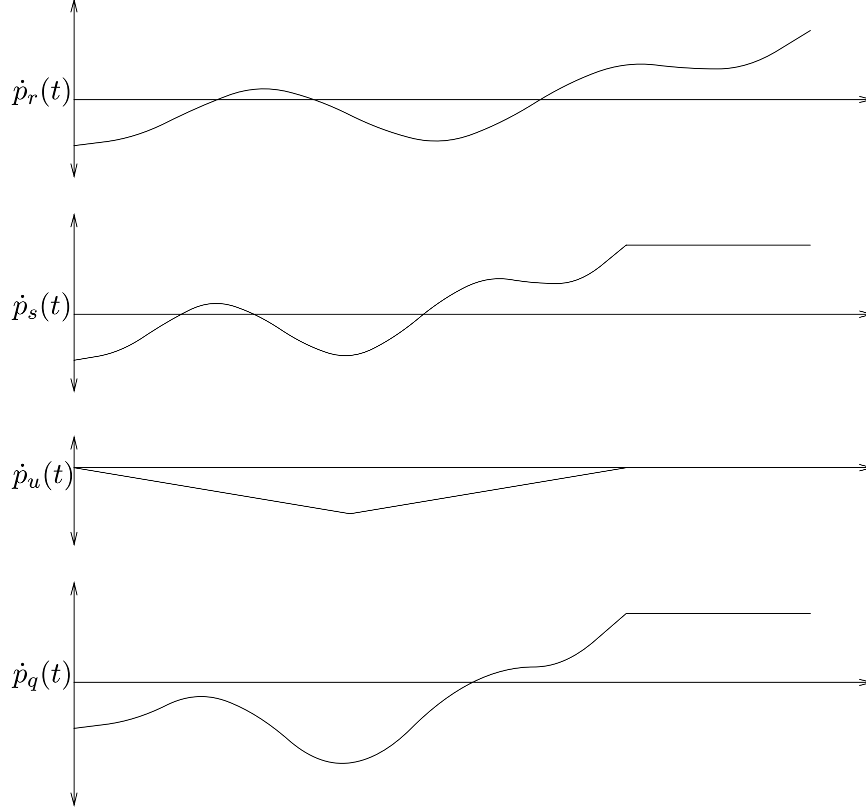


Figure 4.4: Velocity graphs for trajectories from lemma 4.4.4.

Lemma 4.4.4 If we let $c = \frac{\sqrt{9+8\epsilon}-1}{2(1+\epsilon)}$ (note that $c < 1$ for all valid ϵ, and $c \to 1$ as $\epsilon \to 0$), then given an arbitrary time T trajectory Γ_r with acceleration bound $\frac{a}{(1+\epsilon)^2}$, there exists a time T trajectory Γ_q which has the same endpoints but does not change velocity for the last time interval of length $(1-c)T$. Furthermore, Γ_q meets acceleration bound $\frac{a}{1+\epsilon}$.

Proof: We define a temporary trajectory Γ_s by specifying that $\mathbf{p}_s(0) = \mathbf{p}_r(0)$, and then defining the velocity to be a "time-compressed" version of $\dot{\mathbf{p}}_r(t)$. More specifically,

$$\dot{\mathbf{p}}_s(t) = \begin{cases} \dot{\mathbf{p}}_r(\frac{t}{c}) & \text{, for } 0 \le t \le cT \\ \dot{\mathbf{p}}_r(T) & \text{, for } cT < t \le T \end{cases}$$

It is easy to see that $\mathbf{p}_s(T) = (1-c)\mathbf{p}_r(0) + c\mathbf{p}_r(T) + (1-c)T\dot{\mathbf{p}}_r(T)$, and that the velocity at both endpoints of Γ_s is the same as the corresponding velocities of Γ_r. Now we define another auxiliary trajectory Γ_u by setting the initial position to zero and letting

$$\dot{\mathbf{p}}_u(t) = \begin{cases} \mathbf{k}t & \text{for } 0 \le t \le \frac{cT}{2} \\ \mathbf{k}(cT - t) & \text{for } \frac{cT}{2} < t \le cT \\ 0 & \text{for } cT < t \le T \end{cases}$$

where $\mathbf{k}$ is the constant vector $\frac{4(1-c)}{(cT)^2}\left[\Delta\mathbf{p}_r(T) - \dot{\mathbf{p}}_r(T)T\right]$. In other words, Γ_u is a bang-bang

trajectory, used for correction of Γ_s. We can bound $\|\mathbf{k}\|$:

$$\|\mathbf{k}\| = \frac{4(1-c)}{(cT)^2}\left[\|\Delta\mathbf{p}_r(T) - \dot{\mathbf{p}}_r(T)T\|\right] = \frac{4(1-c)}{(cT)^2}\|\int_0^T [\Delta\dot{\mathbf{p}}_r(t) - \Delta\dot{\mathbf{p}}_r(T)]dt\|$$

$$\leq \frac{4(1-c)}{(cT)^2}\int_0^T \|\Delta\dot{\mathbf{p}}_r(t) - \Delta\dot{\mathbf{p}}_r(T)\|dt$$

Since $\frac{\mathrm{d}\|\Delta\dot{\mathbf{p}}_r(t)-\Delta\dot{\mathbf{p}}_r(T)\|}{\mathrm{d}t} \leq \frac{a}{(1+\epsilon)^2}$, we can apply lemma 4.4.1 to get

$$\|\mathbf{k}\| \leq \frac{4(1-c)}{(cT)^2}\left[\frac{\|\Delta\dot{\mathbf{p}}_r(T)\|T}{2} - \frac{\|\Delta\dot{\mathbf{p}}_r(T)\|^2(1+\epsilon)^2}{4a} + \frac{aT^2}{4(1+\epsilon)^2}\right]$$

Maximizing the part in brackets (and noticing that $\|\Delta\dot{\mathbf{p}}_r(T)\| \leq \frac{aT}{(1+\epsilon)^2}$), we get

$$\|\mathbf{k}\| \leq \frac{2(1-c)a}{c^2(1+\epsilon)^2}.$$

Now we can define the trajectory Γ_q by $\dot{\mathbf{p}}_q(t) = \dot{\mathbf{p}}_s(t) + \dot{\mathbf{p}}_u(t)$, and $\mathbf{p}_q(0) = \mathbf{p}_r(0)$. Notice that by the above definitions, $\dot{\mathbf{p}}_q(0) = \dot{\mathbf{p}}_r(0)$ and $\dot{\mathbf{p}}_q(T) = \dot{\mathbf{p}}_r(T)$. To verify that the ending position of Γ_q is the same as the ending position of Γ_r, notice that $\mathbf{p}_u(T) = (1-c)\left[\Delta\mathbf{p}_r(T) - T\dot{\mathbf{p}}_r(T)\right]$, and adding this to $\mathbf{p}_s(T)$ shown above, the resulting simplified expression shows that indeed, $\mathbf{p}_q(T) = \mathbf{p}_r(T)$.

To calculate the acceleration bound of Γ_q, notice that

$$\|\ddot{\mathbf{p}}_q(t)\| = \|\ddot{\mathbf{p}}_s(t) + \ddot{\mathbf{p}}_u(t)\| \leq \|\ddot{\mathbf{p}}_s(t)\| + \|\ddot{\mathbf{p}}_u(t)\| \leq \|\ddot{\mathbf{p}}_s(t)\| + \|\mathbf{k}\|.$$

Using the previously calculated bound for $\|\mathbf{k}\|$ and noticing that $\|\ddot{\mathbf{p}}_s(t)\| = \frac{\|\ddot{\mathbf{p}}_r(t/c)\|}{c} \leq \frac{a}{c(1+\epsilon)^2}$, we see that $\|\ddot{\mathbf{p}}_q(t)\| \leq \frac{2-c}{c^2}\frac{a}{(1+\epsilon)^2}$. Substituting $c = \frac{\sqrt{9+8\epsilon}-1}{2(1+\epsilon)}$, we find that $\frac{2-c}{c^2} = 1+\epsilon$, so $\|\ddot{\mathbf{p}}_q(t)\| \leq \frac{a}{1+\epsilon}$. ∎

Now we examine how closely we can track a trajectory constructed as in lemma 4.4.4. First we consider tracking only the velocity; staying close to the desired velocity keeps the position within a tolerable error, and the last part of the interval (the last time interval of length $(1-c)T$ which is called the *adjustment interval*) is used to correct the position while causing no net change in velocity.

The first step is to divide the time T interval into a series of discrete intervals, each of length τ. For the current velocity, consider a set of choice vectors as described in corollary 4.3.2 with the unit distance being $a\tau$. Assuming that the approximation is within $a\tau$ of the desired velocity at the beginning of an interval, and since the desired trajectory obeys acceleration bound $\frac{a}{1+\epsilon}$, the exact velocity at the end of the interval will be no more than $(1+\frac{1}{1+\epsilon})a\tau$ away from the original approximation. Now using the result of corollary 4.3.2, we can pick a choice vector that results in a final approximation velocity within $a\tau$ of the desired velocity.

From the above argument, it should be obvious that if our approximation velocity initially starts within $a\tau$ of the desired velocity, then at every time step the approximation velocity can be kept within $a\tau$ of the desired velocity. This is what we mean by being able to closely track the velocity of the given trajectory; now we examine how much the position may be in error from blindly following only the velocity of the given trajectory.

First, a better estimate of how closely the velocity is tracked is needed. Theorem 4.3.2 says that at the times $i\tau$ (i an integer), the velocity of the approximating trajectory is within

$a\tau$ of the velocity of the given trajectory, but what happens between these time instances? A maximizing argument (very similar to that used in the proof of lemma 4.4.1) shows that at *all* time instances the error is no more than $\frac{3}{2}a\tau$.

Letting Γ_e and Γ_a denote the exact and approximating trajectories, respectively, the error in position displacement can be bounded by

$$\|\int_0^T \dot{\mathbf{p}}_e(t)dt - \int_0^T \dot{\mathbf{p}}_a(t)dt\| = \|\int_0^T [\dot{\mathbf{p}}_e(t) - \dot{\mathbf{p}}_a(t)]\, dt\| \leq \int_0^T \|\dot{\mathbf{p}}_e(t) - \dot{\mathbf{p}}_a(t)\|dt$$

$$\leq \int_0^T \frac{3}{2}a\tau dt = \frac{3}{2}a\tau T$$

Since the time T interval is divided into length τ time segments, let N be the number of such segments (so $T = N\tau$); therefore, over the entire time T interval, the error in displacement is no more than $\frac{3}{2}aN\tau^2$.

Since the given trajectory we are tracking is a trajectory constructed as in lemma 4.4.4, the velocity does not change for the last $(1-c)T$ time in the time T interval (the approximating velocity as constructed above stays constant in this last time also), this last time can be used to correct the error in position with no net change to the velocity. To show how this is done more explicitly, a few preliminary lemmas are needed.

The next lemma is a purely combinatorial fact, but needs to be established to see how much error can be corrected in the adjustment interval.

Lemma 4.4.5 If M is an even integer ≥ 2, we define the sets

$$S_M = \{(a_1, a_2, ..., a_M) | a_k \in \{-1, 0, 1\} \text{ for } 1 \leq k \leq M, \text{ and } \sum_{k=1}^{M} a_k = 0\}$$

$$T_M = \{\sum_{k=1}^{M} k a_k | (a_1, a_2, ..., a_M) \in S_M\}.$$

Then the set T_M is simply $\{-\left(\frac{M}{2}\right)^2, -\left(\frac{M}{2}\right)^2 + 1, ..., -1, 0, 1, ..., \left(\frac{M}{2}\right)^2 - 1, \left(\frac{M}{2}\right)^2\}$.

Proof: The proof is by induction. For the base case, we will enumerate S_2 and T_2. It should be obvious that $S_2 = \{(1,-1), (0,0), (-1,1)\}$, and from this it is easy to construct $T_2 = \{-1, 0, 1\}$. This agrees with the lemma, and the base case has been proved.

For the induction, assume that the lemma holds for $M-2$, and we will prove that this implies that the lemma is true for M. We will construct a set $S'_M = \{(a_1, a_2, ..., a_M) | (a_1, a_M) \in S_2$, and $(a_2, a_3, ..., a_{M-1}) \in S_{M-2}\}$ and a set $T'_M = \{\sum_{k=1}^{M} k a_k | (a_1, a_2, ..., a_M) \in S'_M\}$. Clearly, $S'_M \subseteq S_M$ and $T'_M \subseteq T_M$.

We make the following observation: for all $(a_1, a_2, ..., a_M) \in S'_M$,

$$\sum_{k=1}^{M} k a_k = a_1 + M a_M + \sum_{k=2}^{M-1} k a_k = a_1 + M a_M + \sum_{k=1}^{M-2} (k+1) a_{k+1}$$

$$= a_1 + M a_M + \sum_{k=1}^{M-2} k a_{k+1} + \sum_{k=1}^{M-2} a_{k+1}$$

Since $(a_2, a_3, ..., a_{M-1}) \in S_{M-2}$, we know that $\sum_{k=1}^{M-2} a_{k+1} = 0$. Furthermore, since the image of $\sum_{k=1}^{M-2} ka_{k+1}$ over S'_M is T_{M-2} (by the induction hypothesis), we can use the definition of T'_M and this observation to see that

$$T'_M = \{M - 1 + e|e \in T_{M-2}\} \bigcup \{e|e \in T_{M-2}\} \bigcup \{1 - M + e|e \in T_{M-2}\}$$

It is easy to see that if $M - 1 + \min\{T_{M-2}\} \leq \max\{T_{M-2}\} + 1$, then $\{n|n \in T'_M \text{ and } n \geq 0\} = \{0, 1, ..., M-1+\max\{T_{M-2}\}\}$. Using the inductive hypothesis for $\min\{T_{M-2}\}$ and $\max\{T_{M-2}\}$, we see that this is indeed true for all $M \geq 2$. A similar argument holds for the negative half of T'_M. Noticing that

$$M - 1 + \max\{T_{M-2}\} = M - 1 + \left(\frac{M-2}{2}\right)^2 = M - 1 + \left(\frac{M}{2}\right)^2 - M + 1 = \left(\frac{M}{2}\right)^2,$$

we see that $T'_M = \{-\left(\frac{M}{2}\right)^2, -\left(\frac{M}{2}\right)^2 + 1, ..., -1, 0, 1, ..., \left(\frac{M}{2}\right)^2 - 1, \left(\frac{M}{2}\right)^2\} \subseteq T_M$.

To see that the inclusion also goes the other way, observe that the maximum value of T_M occurs when $a_1 = a_2 = \cdots = a_{M/2} = -1$ and $a_{M/2+1} = a_{M/2+2} = \cdots = a_M = 1$, so $\max\{T_M\} = \left(\frac{M}{2}\right)^2$. Similarly, it can be shown that $\min\{T_M\} = -\left(\frac{M}{2}\right)^2$. The proof of the lemma is now complete. ▮

This lemma easily applies to give a result about the adjustment interval.

Lemma 4.4.6 Let M be a positive multiple of $2d$ (d a positive integer), and let $\mathbf{e}$ be any d-dimensional vector with $\|\mathbf{e}\| \leq \left(\frac{M}{2d}\right)^2$. Define the set

$$A = \{(a_1, a_2, ..., a_d)|a_i \in \{-1, 0, 1\} \text{ for some } 1 \leq i \leq d, \text{ and } a_j = 0 \text{ for all } j \neq i\},$$

so $\|\mathbf{a}\| = 1$ or $\|\mathbf{a}\| = 0$ for all $\mathbf{a} \in A$. Then there exists a sequence $\mathbf{v}_1, \mathbf{v}_2, ..., \mathbf{v}_M$ where each $\mathbf{v}_i \in A$ such that $\sum_{k=1}^{M} \mathbf{v}_k = \mathbf{0}$ and $\sum_{k=1}^{M} k\mathbf{v}_k = \mathbf{m}$, where $\|\mathbf{e} - \mathbf{m}\| \leq \frac{\sqrt{d}}{2}$.

Proof: Let $\mathbf{e} = (e_1, e_2, ..., e_d)$ and define

$$A_1 = \{(a_1, a_2, ..., a_d)|a_1 \in \{-1, 0, 1\}, \text{ and } a_i = 0 \text{ for all } 2 \leq i \leq d\},$$

and

$$S_1 = \{(\mathbf{v}_1, \mathbf{v}_2, ..., \mathbf{v}_{M/d})|\mathbf{v}_i \in A_1 \text{ for } i = 1, 2, ..., \frac{M}{d} \text{ and } \sum_{k=1}^{M/d} \mathbf{v}_k = \mathbf{0}\}.$$

For any real number r with $|r| \leq \left(\frac{M}{2d}\right)^2$, we can pick an integer m_1 such that $|r - m_1| \leq \frac{1}{2}$ and $-\left(\frac{M}{2d}\right)^2 \leq m_1 \leq \left(\frac{M}{2d}\right)^2$. By lemma 4.4.5, there exists a sequence $(\mathbf{v}_1, \mathbf{v}_2, ..., \mathbf{v}_{M/d}) \in S_1$ such that $\sum_{k=0}^{M/d} k\mathbf{v}_k = (m_1, 0, ..., 0)$ — there are $d - 1$ zeros following m_1.

Since $|e_i| \leq \left(\frac{M}{2d}\right)^2$ for $i = 1, 2, ..., d$, this error correction can be repeated for each dimension, so there exists a sequence of M vectors $\mathbf{v}_1, \mathbf{v}_2, ..., \mathbf{v}_M$ from A such that $\sum_{k=1}^{M} \mathbf{v}_k = \mathbf{0}$ and $\sum_{k=1}^{M} k\mathbf{v}_k = (m_1, m_2, ..., m_d) = \mathbf{m}$, where $|m_i - e_i| \leq \frac{1}{2}$ for $i = 1, 2, ..., d$. Therefore,

$$\|\mathbf{m} - \mathbf{e}\| \leq \sqrt{(m_1 - e_1)^2 + (m_2 - e_2)^2 + \cdots + (m_d - e_d)^2} \leq \sqrt{d\left(\frac{1}{2}\right)^2} \leq \frac{\sqrt{d}}{2}.$$

∎

We use a sequence of choice vectors constructed as in lemma 4.4.6 to correct the position during the adjustment interval. As in corollary 4.3.3 we use $a\tau$ as one "unit length", and set $M = (1-c)N$. From lemma 4.4.4 and lemma 4.4.6 it can be seen that making adjustments during the adjustment interval as in lemma 4.4.6 keeps the final velocity the same, and the first two terms of equation (4.2) remain the same, but the last term can be adjusted by $\pm\left(\frac{(1-c)N}{2d}\right)^2 a\tau^2$. Thus as long as this possible adjustment is greater than the possible error, we can adjust the final position to within $\frac{\sqrt{d}}{2}a\tau^2$ of the exact trajectory, while the final velocity is within $a\tau$ of the exact trajectory.[1] This is summed up in the following theorem.

Theorem 4.4.7 If we set $N = \left\lceil \frac{6d^2}{(1-c)^2} \right\rceil$ (where c is from lemma 4.4.4) and $\tau = \frac{T}{N}$, then given any time T trajectory Γ_e that meets acceleration bound $\frac{a}{(1+\epsilon)^2}$, there is a trajectory Γ_a that uses only the velocity choice vectors (meeting acceleration bound a) with

$$\|\mathbf{p}_e(T) - \mathbf{p}_a(T)\| \leq \frac{\sqrt{d}}{2}a\tau^2$$

$$\|\dot{\mathbf{p}}_e(T) - \dot{\mathbf{p}}_a(T)\| \leq a\tau$$

Furthermore, we then have $N = O(d^2\left(\frac{1}{\epsilon}\right)^2)$.

Proof: By lemma 4.4.4, we can construct a trajectory Γ_s with the same endpoints as Γ_e, such that it takes time T, meets acceleration bound $\frac{a}{1+\epsilon}$, and has constant velocity on the interval $[cT, T]$. As was remarked following the proof of lemma 4.4.4, trajectory Γ_s can be tracked on our grid (producing a grid trajectory Γ_t) such that the grid trajectory also takes time T, meets acceleration bound a, and has constant velocity on $[cT, T]$. Furthermore, it was shown that the error of this approximation can be bounded as

$$\|\mathbf{p}_e(T) - \mathbf{p}_t(T)\| \leq \frac{3}{2}Na\tau^2.$$

$$\|\dot{\mathbf{p}}_e(T) - \dot{\mathbf{p}}_t(T)\| \leq a\tau$$

The interval $[cT, T]$ is used to remove the error from the position (with no net change in velocity) — the relationship between lemma 4.4.6 and the displacement of a grid trajectory is

[1] We have implicitly assumed that positive and negative unit length choice vectors for each coordinate axis exist in our set of choice vectors. This assumption is not too great, as adding these vectors only increases the size of our set of choice vectors by $2d$. Furthermore, these vectors obviously exist on our superimposed square grid.

obvious from equation (4.2). By lemma 4.4.6, the error of at most $\frac{3}{2}Na\tau^2$ can be reduced to $\frac{\sqrt{d}}{2}a\tau^2$ in $(1-c)N$ steps as long as this error is less than the possible adjustment: $\left[\frac{(1-c)N}{2d}\right]^2 a\tau^2$. In other words, the error bounds in the theorem are met if

$$\frac{3}{2}N \leq \left[\frac{(1-c)N}{2d}\right]^2.$$

This condition is met for all $N \geq \frac{6d^2}{(1-c)^2}$, so in particular is met for $N = \left\lceil \frac{6d^2}{(1-c)^2} \right\rceil$, and the error bounds have been proved.

Due to the odd form of c, the asymptotic growth of N is not clear. Consider $\frac{1}{1-c}$; by definition this is simply (for $\epsilon \leq 1$)

$$\frac{1}{1-c} = \frac{2(1+\epsilon)}{3+2\epsilon-\sqrt{9+8\epsilon}} \leq \frac{4}{3+2\epsilon-\sqrt{9+8\epsilon}}.$$

The growth rate (as $\frac{1}{\epsilon} \to \infty$) can be compared with that of $\frac{1}{\epsilon}$ by taking the limit of the ratio

$$\lim_{\frac{1}{\epsilon}\to\infty} \frac{\frac{1}{3+2\epsilon-\sqrt{9+8\epsilon}}}{\frac{1}{\epsilon}} = \lim_{\epsilon\to 0} \frac{\epsilon}{3+2\epsilon-\sqrt{9+8\epsilon}}.$$

The numerator and denominator of this limit both go to 0, so using L'Hôpital's rule, the limit is equal to

$$\lim_{\epsilon\to 0} \frac{1}{2-4(9+8\epsilon)^{-1/2}} = \frac{1}{2-\frac{4}{3}} = \frac{3}{2}.$$

In other words, $\frac{1}{1-c} = \Theta(\frac{1}{\epsilon})$. It follows that

$$N = \left\lceil \frac{6d^2}{(1-c)^2} \right\rceil = O(d^2 \left(\frac{1}{\epsilon}\right)^2).$$

∎

Now we turn attention to tracking within a certain tolerance. By tracking within tolerance (η_x, η_v), we mean that given an exact trajectory Γ_e and an approximating trajectory Γ_a, at all times t, both of the following inequalities hold.

$$\|\mathbf{p}_e(t) - \mathbf{p}_a(t)\| \leq \eta_x \tag{4.5}$$

$$\|\dot{\mathbf{p}}_e(t) - \dot{\mathbf{p}}_a(t)\| \leq \eta_v \tag{4.6}$$

The way we satisfy this is to divide the entire trajectory into a number of intervals, each of which meet the endpoint conditions of theorem 4.4.7. By making the length of such intervals sufficiently small, we can insure that equations (4.5) and (4.6) are satisfied.

For any two time T trajectories Γ_e and Γ_a satisfying the endpoint constraints of theorem 4.4.7, it is easy to see that the approximating velocity can never be farther than $aT + a\tau = a\tau(N+1)$ from the exact velocity; therefore, to satisfy condition (4.6) we only need to insure that $a\tau(N+1) \leq \eta_v$, or $\tau \leq \frac{\eta_v}{a(N+1)}$.

Guaranteeing that the position tolerance is obtained is also easy. An easy proof using lemma 4.4.1 shows that at all times the position can never be farther off than $\frac{(N(N+2)+\sqrt{d})a\tau^2}{2}$, so

to satisfy condition (4.5) we need to insure that $\tau^2 \leq \frac{2\eta_x}{a(N(N+2)+\sqrt{d})}$. Both tolerance conditions can be satisfied if

$$\tau \leq \min\left(\sqrt{\frac{2\eta_x}{a(N(N+2)+\sqrt{d})}}, \frac{\eta_v}{a(N+1)}\right). \tag{4.7}$$

Using the bound for N and noting that we want to control the growth of $\frac{1}{\tau}$, it is interesting to note that the above formula guarantees that we can track within tolerance (η_x, η_v) with $\frac{1}{\tau} = O(\frac{ad^2}{\epsilon^2}\max(\sqrt{\frac{1}{\eta_x}}, \frac{1}{\eta_v}))$ (in other words, polynomial in a, $\frac{1}{\epsilon}$, d, $\frac{1}{\eta_x}$, and $\frac{1}{\eta_v}$).

The above discussion can be summed up in the following tracking theorem.

Theorem 4.4.8 Given any time T trajectory Γ_e from $(\mathbf{p}_e(0), \mathbf{0})$ to $(\mathbf{p}_e(T), \mathbf{0})$ that meets acceleration bound a, there exists a time $(1+\epsilon)T$ trajectory Γ_a on a grid constructed as described in corollary 4.3.3 that also meets acceleration bound a and satisfies

$$\|\mathbf{p}_e(t) - \mathbf{p}_a((1+\epsilon)t)\| \leq \eta_x$$

$$\|\dot{\mathbf{p}}_e(t) - \dot{\mathbf{p}}_a((1+\epsilon)t)\| \leq \eta_v,$$

for any given tolerance (η_x, η_v). Furthermore, the time spacing τ of the grid can be made to meet

$$\frac{1}{\tau} = O\left(ad^2\left(\frac{1}{\epsilon}\right)^2 \max\left(\sqrt{\frac{1}{\eta_x}}, \frac{1}{\eta_v}\right)\right).$$

Proof: Consider the trajectory Γ_e slowed down as by lemma 4.4.3. This new trajectory joins the same endpoints, takes time $(1+\epsilon)T$, and meets acceleration bound $\frac{a}{(1+\epsilon)^2}$. From the given ϵ and the number of dimensions d, we can calculate N as in theorem 4.4.7 and τ as in equation (4.7). Now consider the time required by the slowed down trajectory to be divided into segments, each of the form $[iN\tau, (i+1)N\tau]$. Each segment meets all of the requirements to be tracked as described in the text preceding this theorem, so the result is exactly as stated in the theorem. ∎

4.5 Tracking with Obstacles

As stated in the introduction, we are actually interested in finding paths that avoid a given set of obstacles. The concepts of "safe" and "also-safe" trajectories reflect the real-world physical property that robots cannot navigate accurately at high speeds; the terms were introduced in section 4.1, and are restated here in a more formal setting.

Definition 4.5.1 Let $\delta(c_1, c_0) : \mathbf{R} \rightarrow \mathbf{R}$ be an affine function that maps real numbers to real numbers by $\delta(c_1, c_0)(x) = c_1 x + c_0$ (it will map velocity magnitudes to distance magnitudes); when there is no ambiguity about the values of c_1 and c_0 or the particular values are unimportant, this function is written as simply δ. A trajectory Γ_r is considered $\delta(c_1, c_0)$-*safe* (or just *safe*) if at all times t during the trajectory, the norm of the distance vector to any object is at least $\delta(\|\dot{\mathbf{p}}_r(t)\|)$. An approximating trajectory Γ_q (approximating with accuracy ϵ) is called "also-safe" if at all times t during the trajectory, the norm of the distance vector to any object is at least $(1-\epsilon)\delta(\|\dot{\mathbf{p}}_q(t)\|)$.

The notion of safe and also-safe trajectories comes from [12], and a more general version of the following theorem can be found in their paper (as lemma 3.3). Note that in the following proof, the only property of the norm that we use is the triangle inequality, so the theorem is true for *all* norms, not just the L_2 norm.

Theorem 4.5.2 Let $\delta(c_1, c_0)$ be a safety function as described in definition 4.5.1. A trajectory Γ_a (found as described in theorem 4.4.8) that tracks a safe exact trajectory Γ_e with tolerances

$$\eta_x = \eta_v = \frac{\epsilon c_0}{(1-\epsilon)c_1 + 1}$$

will be also-safe.

Proof: For any time t, we define the "safe ball" about Γ_e to be the set of points within distance $\delta(\dot{\mathbf{p}}_e(t))$ of the point $\mathbf{p}_e(t)$. Similarly, the "also-safe ball" about Γ_a at time $(1+\epsilon)t$ is the set of points within distance $(1-\epsilon)\delta(\dot{\mathbf{p}}_a((1+\epsilon)t))$ of the point $\mathbf{p}_a((1+\epsilon)t)$. It is only necessary to show that the also-safe ball around Γ_a lies entirely within the safe ball about Γ_e at all times. After showing this, it is clear that the also-safe ball around Γ_a is free of obstacles (since the safe ball around Γ_e is free of obstacles); in other words, Γ_a is also-safe.

To show that the also-safe ball for Γ_a lies within the safe ball for Γ_e, consider any point $\mathbf{q}$ in the also-safe ball about $\mathbf{p}_a((1+\epsilon)t)$ — we wish to prove that $\mathbf{q}$ lies within the safe ball about $\mathbf{p}_e(t)$, which is true if and only if $\|\mathbf{q} - \mathbf{p}_e(t)\| \leq \delta(\|\dot{\mathbf{p}}_e(t)\|)$. Of course,

$$\|\mathbf{q} - \mathbf{p}_e(t)\| \leq \|\mathbf{q} - \mathbf{p}_a((1+\epsilon)t)\| + \|\mathbf{p}_a((1+\epsilon)t) - \mathbf{p}_e(t)\|. \tag{4.8}$$

We can bound the first term on the right hand side by using the fact that $\mathbf{q}$ is within the also-safe ball of $\mathbf{p}_a((1+\epsilon)t)$ (so $\|\mathbf{q} - \mathbf{p}_a((1+\epsilon)t)\| \leq (1-\epsilon)\delta(\|\dot{\mathbf{p}}_a((1+\epsilon)t)\|)$), and then write this in terms of $\dot{\mathbf{p}}_e(t)$ and η_v. The final result is that

$$\|\mathbf{q} - \mathbf{p}_a((1+\epsilon)t)\| \leq (1-\epsilon)\delta(\|\dot{\mathbf{p}}_e(t)\| + \eta_v).$$

The second term on the right hand side of equation (4.8) is easily upper bounded by η_x (by the very definition of η_x), so

$$\|\mathbf{q} - \mathbf{p}_e(t)\| \leq (1-\epsilon)\delta(\|\dot{\mathbf{p}}_e(t)\| + \eta_v) + \eta_x$$

Substituting the values of η_x and η_v found in the statement of the theorem, it is easily shown that

$$(1-\epsilon)\delta(\|\dot{\mathbf{p}}_e(t)\| + \eta_v) + \eta_x \leq \delta(\|\dot{\mathbf{p}}_e(t)\|),$$

so $\mathbf{q}$ must lie in the safe ball around $\mathbf{p}_e(t)$. Since this is true for all points $\mathbf{q}$ in the also-safe ball of Γ_a, the also-safe ball of Γ_a must lie entirely within the safe ball of Γ_e. ∎

Combining this with the other results gives the following corollary (our main result).

Corollary 4.5.3 Given acceleration bounds a, obstacles $\mathcal{E}$, and positive reals $\epsilon \leq 1$, c_0, and c_1, for any $\delta(c_1, c_0)$-safe trajectory taking time T, there exists a time spacing τ with

$$\frac{1}{\tau} = O\left(\frac{c_1}{c_0} a d^2 \left(\frac{1}{\epsilon}\right)^3\right),$$

a grid constructed from choice vectors (as described in section 4.3), and a $(1-\epsilon)\delta$-safe approximating trajectory Γ_a between grid-points that takes time at most $(1+\epsilon)T$. Furthermore, this results in an approximation algorithm that is fully polynomial in the combinatorial and algebraic complexity of the environment, and pseudopolynomial in the kinodynamic bounds.

Proof: The existence proof of the $(1-\epsilon)\delta$-safe approximating trajectory follows from the results and discussion above. From the derivation of the bound on τ, it follows that a rational grid size can be chosen where the grid length can be represented with a number of bits that is polynomial in the lengths of the input parameters. It follows that the results of the other simple intermediate calculations will also have polynomially many bits. As the grid is searched, it is reasonably simple to check if the current state (a point on the grid) violates safety margins with the obstacles — simply find the closest obstacle boundary point to the point being tested, then check to see if that distance violates the safety function at the current velocity (the state gives the velocity at the point). Verifying that safety constraints are not violated *between* grid-points is a simple extension [12]. This operation is fully polynomial in the geometric complexity of the obstacles $\mathcal{E}$.

The size of the search space is exactly the number of possible states. Considering how fast the grid of section 4.3 grows, it is clear that the number of possible velocity vectors in the search space is bounded by $\left(\frac{4v_{\max}}{\epsilon a_{\max}\tau}\right)^d$. From the diameter D of the space and equation (4.2), it should be clear that the number of possible positions is bounded by $\left(\frac{4D}{\epsilon a_{\max}\tau^2}\right)^d$. Combining these quantities, the number of states is $O\left(\left[\frac{v_{\max}D}{\epsilon^2 a_{\max}^2\tau^3}\right]^d\right)$; in other words, since $\frac{1}{\tau}$ is polynomial in the dynamics bounds, the total number of grid-points is polynomial in the dynamics bounds (but not in their *lengths* — hence the search algorithm is only pseudopolynomial).

Since the grid size is polynomial in the kinodynamic bounds, and the complexity of checking the validity of each grid-point is polynomial in the geometric complexity, the complexity results claimed in the theorem are verified. ∎

4.6 Chapter Summary

We have shown that while the (exact) optimal kinodynamic planning problem may be computationally difficult, it is possible to approximate the optimal path with our simple algorithm — simply construct a grid as explained in section 4.3 and perform a search on this grid to find a path from the start state to the goal state. The main result of this chapter is that if the grid is constructed within certain parameters (see corollary 4.5.3, equation (4.7), etc.), then for any safe optimal path there exists an also-safe grid path that is within a $(1+\epsilon)$ factor of optimal. The size of the grid is polynomial in the input size, in $\frac{1}{\epsilon}$, and in the dynamics bounds, so the result is a polynomial approximation algorithm for kinodynamic planning (where dynamics bounds are expressed in terms of maximum 2-norm for acceleration).

Chapter 5

Motion Planning in Hostile Environments

5.1 Introduction

In alternation (as presented by Chandra, Kozen, and Stockmeyer [16]), the state configurations are finite strings, and two players (an existential and universal player), beginning at an initial configuration, alternately make discrete moves chosen from a given finitely described next move relation. The players make these moves with perfect information of the current position; the associated decision problem is to determine whether there is a strategy for the existential player that always reaches a given final accepting configuration. Discrete alternation has proved to be a very useful notion, with applications in complexity theory, game theory, and parallel computing.

In this chapter, we investigate an interesting variant of alternation, which we call continuous alternation. Each configuration is a point in a dense space, say $\Re^d$, and there are again two players: the existential and universal players. The configuration x is partitioned into two parts, where each part is controlled by a distinct player. Each player continuously makes moves satisfying differential constraints of the form $F(t, x, x', x'', ...)$, where F defines a set of semi-algebraic inequalities in the derivatives of x and their norms. We also specify distinguished initial and final configurations and their derivatives. Again, both players have perfect information, and the problem is to determine a continuous strategy for the existential player that is successful in reaching the final configuration with a continuous trajectory against any dynamic pursuer. In addition, there may also be a restriction on the time required to reach the final configuration. Interesting examples of continuous alternation games are the differential pursuit games considered in game theory (see, for example, [9]). A typical pursuit game has constraints $F(t, x, x', x'', ...)$, where F specifies the geometry of obstacles to be avoided by players, the shape of the players, as well as a restriction that the players not collide.[1] F can also specify a norm bound on the velocity or acceleration of each player. Such pursuit games are actually just robotic motion planning problems in the presence of an adversary that tries to stop our robot.

The complexity of continuous alternation depends very much on the form of the differential constraints. In this chapter, we provide the first upper and lower bounds on the complexity of a class of continuous alternation games. In particular, we consider pursuit games in 3

[1] A collision is defined as an intersection of the players that has non-zero volume. In other words, when avoiding collisions the boundaries are allowed to intersect, but no points internal to the players may overlap.

dimensions, where the obstacle sets are polyhedra with fixed rational position in $\Re^3$, and the L_2 norm of the velocity of each player is bounded. We show that the decision problem for these pursuit games is exponential time hard. The lower bound is quite surprising, since the degree of freedom of the players (the dimension d of the configuration space) is constant. This is the first provable intractability result for a robotics problem with perfect information. In fact, there had previously been no provable intractability results for any robotic problems with perfect information, even with n degrees of freedom.

We also give approximation algorithms for a wide class of pursuit games: the velocity and acceleration have either L_∞ or L_2 norm bounds, the obstacle sets are (possibly moving) polyhedra with fixed rational initial positions, and there are certain "safety" constraints on the closeness that the players can approach obstacles or each other. Our approximation algorithms allow the existential player to find a strategy that reaches the final configuration within an additive ϵ factor (for any $\epsilon > 0$) of the optimal deadline time. In the case of L_2-norm bounds, we have to exceed the dynamics bounds by a multiplicative factor of ϵ. The algorithm generalizes to any type of obstacle (not just polyhedra) that can be tested for collision in $O(\log n)$ space, such as obstacles having constant degree algebraic descriptions.

To emphasize the relationship with discrete alternation, both the lower and upper bounds are proved using the alternating Turing machine as the model of computation.

There are many variations on the pursuit game we have defined, and some minor changes can greatly affect the complexity of the problem. For instance, if each player is a single point, then it can be shown that whenever the pursuer is allowed a velocity bound at least as high as the evader, the game is easily decided by just calculating the minimum distance to the goal for each player. This result is proved in section 5.4.

5.2 Lower Bounds

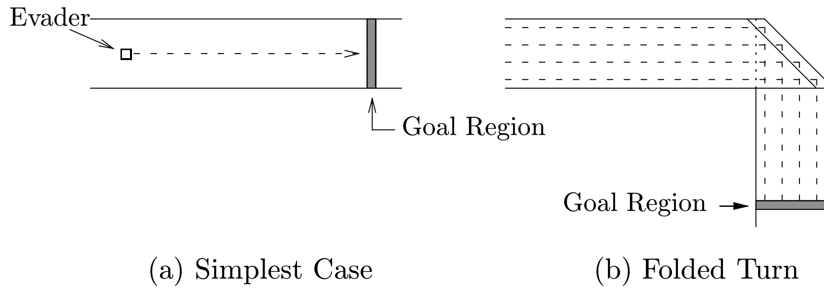
In this section we prove that the pursuit problem in three dimensions is hard for EXPTIME. We first prove the result for a system where only translations are allowed (no rotations), and then we show how a similar construction for arbitrary movement can be constructed, yielding the lower bound for the more general problem.

To prove lower bounds for the pursuit problem, we construct a problem that simulates a given polynomial space bounded alternating Turing machine (ATM) M. From the classic work on alternating Turing machines, we know that APSPACE=EXPTIME [16], so our lower bound follows. To simplify the explanation, we assume that the ATM M uses only n tape cells; the generalization to an arbitrary polynomial should be obvious.

One possible point of confusion in the proof that follows is that the existential player of the pursuit game simulates *all* moves of the ATM (both existential and universal); the universal player of our continuous game makes the transition *choices* in universal states, and forces the existential player to actually perform the appropriate transition. To avoid problems with terminology, in this section we will refer to the existential player (of the pursuit game) as the *evader*, and the universal player as the *pursuer*.

5.2.1 Basic Geometry and the Encoding of a Configuration

The evader in the construction will be a three-dimensional cube with each side having length 2^{-4n+1} (note that while many dimensions in our construction are exponentially small, only polynomially many bits are required to specify the boundary coordinates). Rectangular tunnels



(a) Simplest Case (b) Folded Turn

Figure 5.1: Some Basic Constructions

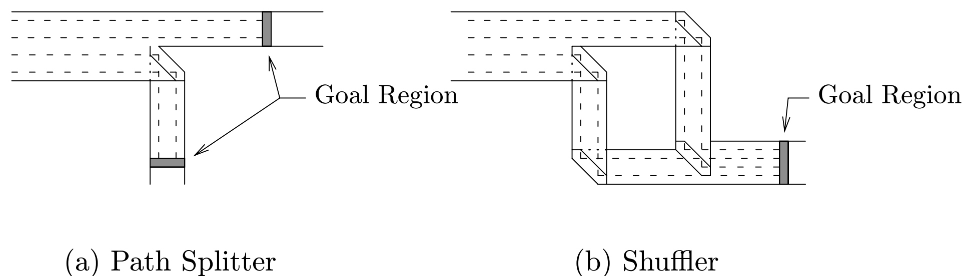
that are exactly 2^{-4n+1} units tall[2] and 4 units wide will be the areas in which the evader will travel. The position of the evader along the width of this passage will encode the contents of the ATM's tape. The general idea is to use the distance from a consistently chosen wall of the passage (called the zero wall) to represent the tape contents — if this distance is the binary number $0.a_1a_2...a_n$, then the tape contents are $a_1, a_2, ..., a_n$. Due to the size of the evader, there is clearly enough room to do this (the distance between valid configuration encodings would be 2^{-n}), but a slight modification must be made to include the position of the tape head. This will be discussed in further detail in section 5.2.4.

A similar encoding scheme was used by Canny and Reif [14] to represent the values of boolean variables in an instance of 3SAT. Their main result was an NP-hardness proof for 3-dimensional minimum path calculation, and we use several ideas from this earlier work. The most important concept in our lower bound proof is the idea of shortest path classes, modified slightly from the work of Canny and Reif [14]. Consider a polyhedral object (our robot) among a set of obstacles, and a given goal region (possibly disconnected). We wish to move our robot until it touches some point in the goal region. Obviously, if it is possible to reach the goal region, there is some minimum time T required to do so; the set of all paths to the goal region that take time T (there can easily be more than one) is the shortest path class for this problem instance.

In all of our problems, the robot is a small cube (the evader), and the goal region will be a set of cross-sections of the evader's passages. For example, consider the simplest case of a straight, unobstructed passage with the goal region being the end of the passage (see figure 5.1a for a top view). Obviously, the only path in the shortest path class is the straight-line path shown in the figure. When the evader follows this path, the distance to the walls of the passage is preserved.

We will be defining sets of obstacles (called traps) such that any winning strategy for the evader involves following a shortest path through the trap. The position of the evader in the cross-section of its passageway reflects the configuration of the ATM, and we have seen how this is preserved in shortest-path classes along a straight section of the evader's passage. What if we want the evader to change the direction of its motion? For this we use the "folded turn" shown in figure 5.1b. Imagine taking a piece of flat ribbon (representing the evader's passage) and folding it at a 45° angle — the top view should be like that shown in figure 5.1b. Several

[2] In the descriptions of this section, "tall" refers to an object's length in the z-coordinate direction, "wide" refers to the object's length in the x-coordinate direction, and "deep" refers to the object's length in the y-coordinate direction.



(a) Path Splitter (b) Shuffler

Figure 5.2: More Basic Constructions

shortest path classes are shown in the figure, representing various starting positions for the evader. The construction we use cannot be exactly as visualized by folding a flat ribbon, since the passages (and walls) must have some finite thickness. The actual construction consists of a 45° slot cut in the floor of the original passage, and a vertical drop to a second-level passage (below the first). This second passage is oriented at a right angle to the first.

We can also separate the path classes depending on the half of the passage in which the evader is traveling. By using a "folded turn" construction, but having the 45° slot extending only across the bottom half of the passage, and having the goal regions set up across the two resulting passages at an equal distance from the entrance, the path classes will be separated (see figure 5.2a). The resulting construction is called a "path splitter".

The shuffler is the most important construction, and can be viewed as a combination of the above constructions (see figure 5.2b). A path splitter divides the top and bottom path classes, and a folded turn on the top half of the passages leaves both halves on the same level. Both passages then take a folded turn to travel horizontally, with the top half of the passages dropping down to the bottom half with a slight offset from the bottom path classes. Given a discrete set of valid starting positions for the evader, this effectively interleaves (or shuffles) the path classes, as shown in the figure. There is a small technical problem with this construction; namely, we would like the paths for all starting positions to have the same length. Unfortunately, while all paths starting in the lower half of the passage have the same length, this length is slightly shorter than the paths starting in the upper half of the passage. To alleviate this problem, the lower-half path classes are lengthened by adding an additional vertical "jog" (a vertical drop followed by a vertical rise) after the path splitter. With this addition, it should be clear that the shuffler works as desired.

The effect of the shuffler is to change the distance of an evader from $0.a_1a_2...a_n$ (as measured from the zero wall and written in binary fixed point) to $0.0a_2a_3...a_na_1$. This function is vital for testing individual bits of the configuration. As in the paper of Canny and Reif [14], the shuffler halves the width of possible positions for our evader; this was a major problem that limited the time of the simulation in [14] to polynomial. However, in the current situation the presence of the pursuer allows us to overcome this problem.

The pursuer will be a rectangular box 5 units wide, 2^{-4n} units tall, and 2^{-4n} units deep. Since rotations are not allowed, we see that the pursuer cannot travel in the evader's passage (since the passage is only 4 units wide), and by making the pursuer's passages 2^{-4n} units deep, the evader will not be able to travel in the pursuer's passages (since the evader is 2^{-4n+1} units deep). In this way, we ensure that there are only a few spots of contention between the pursuer

and evader — namely, those places where the pursuer's passage intersects with the evader's passage.

The actual velocity bounds on the pursuer and evader (v_p and v_e, respectively) are not important, but the ratio of the two bounds will determine certain elements of the construction. For the sake of concreteness, we will let $v_p = 10v_e$.

5.2.2 Basic Form of the Proof

We describe a polyhedral environment that has a polynomial size binary encoding. This environment will be constructible in $O(\log n)$ space by a deterministic TM. We will show that the players will be forced to play essentially in a discrete manner that simulates the given ATM M, or else they will immediately lose the game. We will have a set of obstacles (called the state box) associated with each state of the ATM M. The initial positions for the pursuit game players are at the entrance of the state box associated with the initial state of M, and the position of the evader across the width of its passage encodes the input of the ATM (i.e., the initial tape contents). The goal state of the evader is in the box associated with the accepting state.

The proofs that follow show that for any winning strategy, the only valid paths through each state box correspond to valid transitions of the ATM. By induction, any winning strategy will reach the goal position (corresponding to the accepting state of the ATM) by a series of valid state transitions; therefore, there is a winning strategy if and only if the ATM accepts. The canonical strategy for any accepting ATM reflects the appropriate sequence of existential moves of the ATM.

5.2.3 Traps

The key component in the lower bound construction is the concept of a *trap*. A trap is a specific region where the evader will become trapped if the shortest path through the region is not taken. After being trapped, there will be no way for the evader to reach the goal position.

The basic trap is illustrated in figure 5.3. The "evader move box" is some type of basic construction, such as a shuffler. The pursuer's passage starts below the level of the evader's passage, and continues through the evader's passage (the horizontal strip in figure 5.3); when this passage reaches a certain height, it takes a 90 degree turn to stretch horizontally to a position after the evader's move box. At this point, the passage turns down and continues through the evader's passage. The length of the pursuer's passage is carefully chosen (and set by the height it rises over the evader's passage) so that the pursuer's shortest path from the entrance intersection to the exit intersection takes *exactly* the time of the evader's shortest acceptable path between the intersections. We call the time required by the pursuer to go between the two intersections with the evader's passage the *threshold time* of the trap.

A valid starting position for a trap is as follows. The pursuer is in its passage with its top flush with the floor of the evader's passage. The evader starts at a position following the entrance intersection, with its trailing edge flush with the edge of the intersection. The remaining degree of freedom for the evader's position is arbitrary (in fact, it will encode the ATM's tape contents). Note that in this position the pursuer and evader are actually touching at an edge; however, there is no collision as the volume of the intersection is zero.

Theorem 5.2.1 There exists a strategy for the pursuer such that from a valid starting position, the evader can leave a trap safely if and only if the time it takes to pass the exit intersection is no more than the threshold time of the trap.

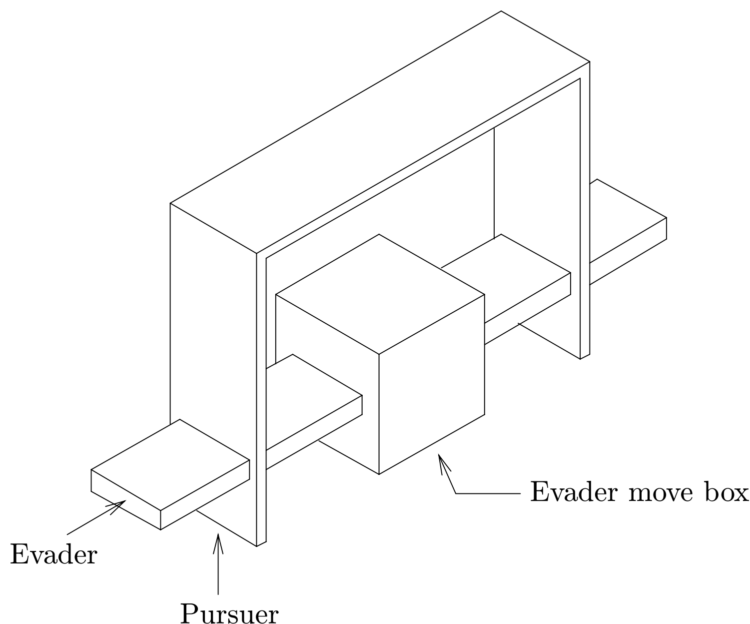


Figure 5.3: Basic Trap

Proof: The *state* of the pursuit game at any time can be completely specified by the positions of the players; to denote the state at time t, we write $\mathbf{s}_t$. Let S denote the set of all valid states. For a specific trap, we define the function $\Phi_e : S \to \Re$ that maps states to an amount of time. Specifically, for any state $\mathbf{s}$ (where the evader is in the trap), let t_m be the minimum time required by the evader to travel to the entrance intersection and first become flush with it. The function Φ_e is defined by $\Phi_e(\mathbf{s}) = t_m$ for all such position-time pairs — this may be a fairly difficult function to compute, depending on the complexity of the evader move box, but we are not concerned with the complexity of the pursuer's strategy. Notice that if $\mathbf{s}_0$ is a state with the evader in a starting position for the trap (as described above), then $\Phi_e(\mathbf{s}_0) = 0$. The function $\Phi_p : S \to \Re$ is defined similarly, but for the pursuer; specifically, $\Phi_p(\mathbf{s})$ is the minimum time required for the pursuer to travel back to its starting position in the trap.

Now we describe a strategy for the pursuer such that the evader will become caught in the trap if its path through the trap takes longer than the threshold time. Notice that if both players travel as fast as possible through the trap, then at all times, $\Phi_e(\mathbf{s}_t) = \Phi_p(\mathbf{s}_t) = t$. The idea behind the pursuer's strategy is as follows: if the evader strays from a minimum distance path, some additional time is available for the pursuer to cover more ground than the evader; in this way, the pursuer can reach the exit intersection before the evader leaves the trap. Obviously, we don't want the pursuer to get *too far* ahead of the evader — otherwise, the evader could exit the trap by reversing its course and leaving through the entrance. Specifically, the pursuer's strategy is to travel forward in its passage as fast as possible, as long as $\Phi_p(\mathbf{s}_t) \leq \Phi_e(\mathbf{s}_t) + \frac{2^{-4n}}{v_e}$. If such a time is ever reached where $\Phi_p(\mathbf{s}_t) = \Phi_e(\mathbf{s}_t) + \frac{2^{-4n}}{v_e}$, then the pursuer simply imitates the progress of the evader, and this equality is maintained.

We now show that the above strategy for the pursuer has the property that the evader can leave the trap only by taking a minimum time path to the exit. First, notice that the evader

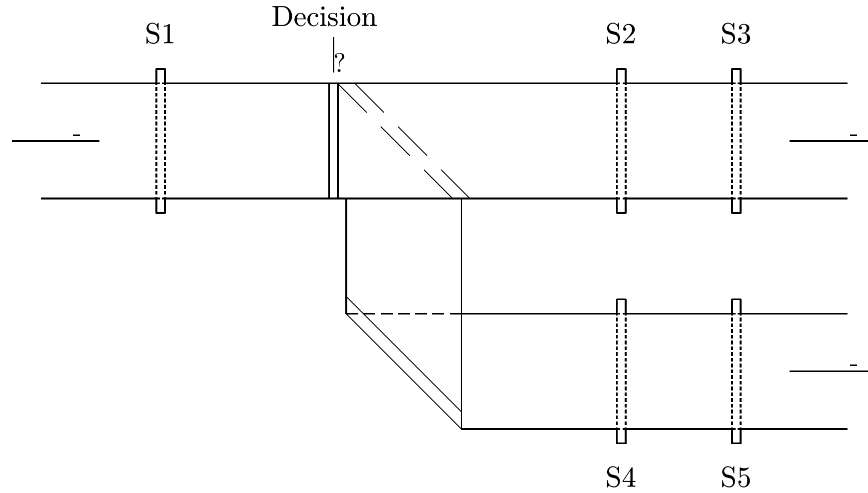


Figure 5.4: Forced choice trap

cannot leave the trap past the entrance — if the evader ever starts traveling backwards, then the pursuer will maintain the equality $\Phi_p(\mathbf{s}_t) = \Phi_e(\mathbf{s}_t) + \frac{2^{-4n}}{v_e}$. When the evader first becomes flush with the entrance intersection we have $\Phi_p(\mathbf{s}_t) = \frac{2^{-4n}}{v_e}$, so the pursuer is able to collide with the evader before the evader can leave the trap. This is because from a position with $\Phi_e(\mathbf{s}_t) = 0$, it takes the evader at least $3\frac{2^{-4n}}{v_e}$ time to travel completely past the entrance intersection and out of the trap, but it takes the pursuer at most $\frac{1}{3}$ of this time to cross the evader's passage (colliding with the evader). A similar argument shows that if the evader ever strays from a minimum distance path (so $\Phi_p(\mathbf{s}_t) > \Phi_e(\mathbf{s}_t)$), then the evader cannot leave the exit of the trap without colliding with the pursuer. ∎

Thus if the threshold time is equal to the shortest path time of the evader, then the evader *must* follow a shortest path. However, in the final construction, many traps will be connected together, so how can we be sure that the pursuer cannot improve its strategy by following a path backwards in its passage? In other words, with the above pursuit strategy, we can guarantee that the evader moves forward as quickly as possible — how can we guarantee that the pursuer will do the same? This is easily done by making the pursuer go forward in order to block a path to the goal for the evader. This idea is generalized in the notion of a "forced decision trap".

A top view of the forced decision trap is shown in figure 5.4 — only the evader's passage is shown in the figure, and the dotted boxes correspond to intersections with the pursuer's passage. The actual 3-dimensional construction is quite complex, and a rough drawing is shown in figure 5.5. This trap has a single entrance and two exit passages for the evader. Inside the trap (but not shown in figure 5.4), the pursuer's passage also splits into two separate paths, and the evader must pick which exit to take according to the pursuer's choice. The evader must choose the correct exit *and* take the shortest path, or else it will be caught in the trap.

The valid starting position is the same as for the basic trap, with respect to slot S1. The evader's passage is a straight passage with a slot cut in its floor (labeled "decision" in figure 5.4). The passage under the decision slot makes two "folded turns", and continues as the bottom exit of figure 5.4.

Shortly after the pursuer's passage passes through slot S1, it forks into two passages. One

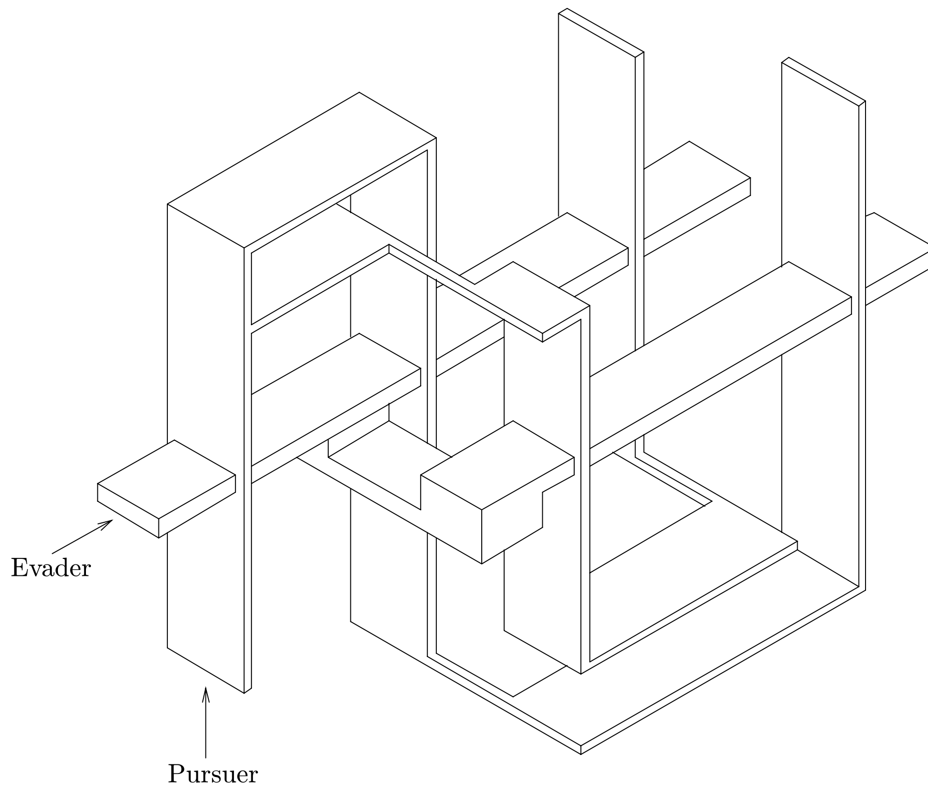


Figure 5.5: 3-D Rendering of Forced Decision Trap

path passes through S2 and then S5, and the other passage goes through S4 and S3. The requirement on the first passage is that the shortest time from slot S1 to S5 is exactly the time it takes the evader to travel from slot S1 to S5. Furthermore, the time required to travel from S1 to S2 must be slightly greater (see the proof of theorem 5.2.2 for the exact requirements) than the time for the evader to travel from S1 to the decision slot. The second passage has analogous requirements.

When the pursuer takes the passage to S5, the correct path for the evader is to travel to the lower exit along its shortest path. Similarly, the evader must take its shortest path to the upper exit if the pursuer takes the passage to S3.

Theorem 5.2.2 The evader can leave the trap if and only if it follows the canonical shortest path to the appropriate exit (as chosen by the pursuer).

Proof: In this proof we will assume that the pursuer wants to force the evader to take the bottom exit of figure 5.4. In such a case, the pursuer stays entirely in its passage between slots S1, S2, and S5. The case where the pursuer forces the evader to take the top exit is almost identical to the case presented below, so is not explicitly given here.

As in the proof of theorem 5.2.1, let $\Phi_p(\mathbf{s})$ denote the minimum time required for the pursuer to travel from its position in state $\mathbf{s}$ to its start position in the trap. It is important to remember that regardless of the moves made by the evader, the pursuer stays *only* in the passage from S1 to S2 and S5. We define $\Phi_e(\mathbf{s})$ a little differently than in theorem 5.2.1. In particular, let t_d be the time required for the evader to reach a position directly above the decision slot. Since the correct path for the evader is to the bottom exit, label all states with the evader to the right of the position over the decision slot and to the left of slot S2 as "bad states". Now define $\Phi_e(\mathbf{s})$ to be the same as in theorem 5.2.1 whenever $\mathbf{s}$ is not a bad state; in other words $\Phi_e(\mathbf{s})$ is the minimum time required for the evader to travel back to S1. On the other hand, if $\mathbf{s}$ is a bad state then set $\Phi_e(\mathbf{s}) = t_d$.

Now we describe the timing requirement for the pursuer's passage to slot S2: the minimum time required for the pursuer to travel from its start position to a position entirely blocking the evader's passage at S2 must be exactly $t_d + \frac{2^{-4n}}{v_e}$. This requirement is easily set by adjusting how high the pursuer's passage rises above slot S1.

The pursuer's strategy is exactly as in theorem 5.2.1: the pursuer travels forward in its passage as fast as possible while maintaining $\Phi_p(\mathbf{s_t}) \leq \Phi_e(\mathbf{s_t}) + \frac{2^{-4n}}{v_e}$. If the evader never enters a bad state, then the set of actions is exactly like a basic trap, so the evader must take its shortest path by theorem 5.2.1. If the evader ever enters a bad state, then the function $\Phi_e(\mathbf{s})$ remains constant for some amount of time. During this time, the pursuer can still travel, insuring that $\Phi_e(\mathbf{s}) > \Phi_p(\mathbf{s})$; as in theorem 5.2.1, once this inequality is achieved it can be maintained, and the evader cannot leave by either the entrance or the bottom exit. Clearly, when the evader reaches slot S2, then we have had time to reach the point where $\Phi_p(\mathbf{s}) = \Phi_e(\mathbf{s}) + \frac{2^{-4n}}{v_e}$; in other words, the pursuer will fully block the top exit before the evader reaches S2, so the evader is completely caught in the trap.

With the above strategy, the *only* way for the evader to leave the trap is for it to take its shortest path to the bottom exit. Clearly, the case for forcing the evader to take the top exit is almost identical, so is not presented here. ∎

By connecting the top exit directly with the goal position (through a tunnel that the pursuer cannot enter), we can ensure that the pursuer keeps moving forward on its path (to block the evader from this goal shortcut). The forced decision trap will also be used to simulate universal states of the ATM.

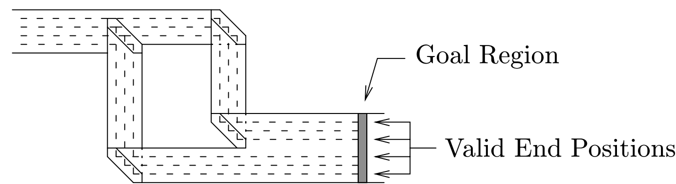


Figure 5.6: Bad Cases for Unshuffler

5.2.4 ATM Transitions

Consider the following method of representing the tape contents: the 0's and 1's on the tape correspond to a number in base four notation (not all base four numbers will correspond to valid tape configurations). In the position that is currently being scanned, the digit is changed from 0 or 1 to 2 or 3, respectively. The distance from a consistently chosen wall (the zero wall) to the evader encodes this representation. Let d denote this distance. Then the base four tape configuration representation r is related to d by $r = d2^{2n-2}$.

We can use the shuffler (figure 5.2b) to examine bits of the configuration, but only bits in even numbered positions will actually represent symbols on the tape (the odd positions mark the location of the tape head). The path splitter can be used to "peel off" the configuration when it is shuffled to the currently scanned tape position. Traveling through the shuffler backwards has the undesirable side-effect of doubling the number of shortest path classes (but also doubles the width of possible positions for the evader, which is good). In other words, for most input positions of the evader in the reversed shuffler there are two different shortest paths to the exit slot, only one of which is the correct unshuffling path (see figure 5.6). The incorrect path will always end at a position that is not a valid encoding of a base 4 number. This problem is impossible to overcome in the shortest path proof of [14]. Fortunately, the addition of a pursuer allows us to fix this problem by using a forced decision trap — one side will go to a verifier for the tape configuration, so any winning strategy must do the unshuffle correctly (or risk being trapped in the verifier).

Lemma 5.2.3 Assuming the distance d to the zero wall is an integer multiple of 2^{-4n+2}, there exists a trap such that only valid encodings of base four numbers can escape the trap (i.e., only when the evader distance is a multiple of 2^{-2n+2}).

Proof: The basic block is a modified version of the shuffler; it differs from the shuffler in that the final stage (combining the top and bottom halves of the path splitter) does *not* have the offset required by the shuffler. If the evader enters this box at distance $d = 2k + d'$ from the zero wall for $k \in \{0, 1\}$ and $0 \leq d' < 2$, then it leaves the box at distance d' from the zero wall. In other words, the valid positions from the upper half of the evader's passage are overlapped with valid positions in the lower half.

Repeating this construction $2n$ times (for geometrically decreasing passage width), the evader will leave the final exit at distance zero if and only if it entered the construction in a valid position. Since the evader enters at a multiple of 2^{-4n+2}, all *invalid* positions will be at a distance greater than 2^{-4n+2}. By placing a wall in front of all positions further than 2^{-4n+2} from the zero wall and enclosing this structure in a basic trap, all invalid positions will take too much time (since they have to go around the wall) and get caught in the trap. ▮

To perform a state transition, the tape representation is shuffled until the current tape cell is found (i.e., the evader passage is shuffled until the evader is in the upper half of the passage). This is accomplished with $2n$ shufflers in series, with a path splitter placed after every two shufflers. Even after the maximum number of shuffles ($2n$), there is still a 2^{-4n+2} spacing between valid positions for the evader, so it is still a simple matter to distinguish between different configurations. A transition is easily performed by simply shifting the evader's position in its passage (actually, the passage is shifted, and the evader maintains a straight-line path), followed by the correct number of unshufflers. Following all the unshufflers is a forced decision trap with one exit linked to a verifier as described in the lemma above (the output is a shortcut to the goal) — this ensures that all unshufflers work correctly. The total number of gadgets required is $2n$ shufflers, $2n$ unshufflers, n path splitters, a forced decision trap, and a verifier. The total number of bits required to encode these constructions is clearly polynomial in n.

A set of gadgets is built for every state in the ATM. If the state is existential and there are k possible transitions out of this state, then the incoming evader passage splits into k passages, each of which performs a transition as described above. The exit from each transition goes to the corresponding next state. If the state is universal with k next states, then construct a $\lceil \log k \rceil$ depth tree of forced decision traps with all but k exits linked directly to the goal. Each non-goal exit is followed by a construction that performs a transition out of the universal state.

Notice that in existential states, the evader has a free choice of which path to take, but for a winning strategy it must make a choice compatible with *all* possible future universal options (since the pursuer can arbitrarily force any choice in the universal states). We have proved the following theorem.

Theorem 5.2.4 For any polynomial space bounded ATM and n-bit input x, a pursuit game with no rotations can be constructed such that a winning strategy exists if and only if the ATM accepts x. The pursuit game has a polynomial length binary encoding, and can be computed by a Turing machine in $O(\log n)$ space.

To extend the proof to a lower bound when rotations are allowed, consider a pursuer as above, but with a groove cut in the top. The pursuer passages then have tracks that fit into the pursuer's groove and make rotations impossible. Re-examination of the above construction shows that allowing rotation of the evader does not affect the lower bound proof.

Corollary 5.2.5 For any polynomial space bounded ATM and n-bit input x, a general pursuit game can be constructed such that a winning strategy exists if and only if the ATM accepts x. The pursuit game has a polynomial length binary encoding, and can be computed by a Turing machine in $O(\log n)$ space.

The following corollary follows from the fact that APSPACE=EXPTIME.

Corollary 5.2.6 Any algorithm that solves the decision problem for the polyhedral pursuit game (either with or without rotations) must take at least exponential time in the worst case.

5.3 Approximation Algorithms

In this section, we look at polyhedral pursuit games with various types of dynamics bounds, and develop approximation algorithms for these games. The closeness of the approximation (as defined below) is given by a parameter $\epsilon > 0$. We are also given a deadline time for the pursuit game that is bounded by a polynomial in $1/\epsilon$.

Given any rational number ϵ, we call a strategy ϵ-safe if the strategy will always keep distance ϵ between the evader and both the pursuer and all obstacles (notice the similarity between this notion of safeness and that used in chapter 4). In this section we give an algorithm which, when given a pursuit game and a safety margin ϵ, will always find a winning strategy if there exists an ϵ-safe strategy. If a winning strategy exists, but there is no ϵ-safe strategy, then the algorithm may or may not find a winning strategy (but will never give a bad strategy).

Such approximation algorithms exist for problems where either the velocity or both velocity and acceleration are bounded, and the bound can be on either the L_2 or L_∞ norm (although a slight concession must be made on the dynamics bounds in the L_2 case). For each variant of the problem, we have a different closeness lemma (this is lemma 5.3.1 below, for bounded L_∞ norm of velocity), but the relation to the continuous pursuit games is the same in every case. In the first section below, we present proofs for the simplest case: bounded L_∞ norm of velocity, with no bound on acceleration. The other cases are similar and involve "closeness" proofs (which we give in the following sections) that are very similar to the tracking lemmas in approximately optimal kinodynamic planning (see, for example, chapter 4 of this dissertation and [23]).

5.3.1 Bounded L_∞-norm velocity

The following discrete game will be used on a discretization of the geometry of the continuous game. Since the reachability sets for the players may be different (due to the different shapes of the players), we need a way of marking which players can follow which edges. This is the purpose of the labeling function below.

Consider the following discrete game. The input is a graph where each edge e has a label $l_e \subseteq \{0, 1\}$. Two players (player 0 and player 1) start at given vertices, and player p is allowed to traverse edge e if $p \in l_e$. The game consists of rounds where each player traverses a valid edge for that player, and we wish to know if there is a strategy for player 0 so that it can reach a given goal vertex while never coming "close" (within two steps on the graph) to player 1. It should be clear that this game can be decided in ASPACE($\log n$) for an n vertex graph.

We show that slight variants of this game can be constructed to solve the ϵ-safe approximation version of pursuit games. In d-dimensions, the graph will have $O(\left(\frac{1}{\epsilon}\right)^d)$ vertices for bounded L_∞ norm velocity, and slightly more (but still $\left(\frac{1}{\epsilon}\right)^{O(d)}$) for the more complex pursuit games. We now present the case for pursuit games with a bound on the L_∞-norm of the players' velocity.

If we are given the starting configuration for a pursuit game, and bounds on the L_∞ norm of the velocity for the evader and the pursuer (denote the bounds by v_e and v_p, respectively), we superimpose a regular grid G with grid-spacing g on the d-dimensional environment. We label each grid-point by a d-tuple of integers $(x_1, x_2, ..., x_d)$. A graph is constructed on the grid by connecting every point $(x_1, x_2, ..., x_d)$ to points $(y_1, y_2, ..., y_d)$ with $|x_i - y_i| \leq 1$ for all $i = 1, 2, ..., d$. (This is just a d-dimensional grid-graph with diagonal edges added.) An arbitrary point on each player is chosen; a player is at grid-point p when the chosen point of the player is at the grid-point p. The edges of the graph are labeled according to whether the path joining the two endpoints is free of obstacles for each player.

We will choose a sufficiently small grid-size so that the discrete game is a good approximation of the continuous game. First, we show that for sufficiently small grid-size, there is always a good strategy for the evader (traveling on the grid) against a *continuous* pursuer.

Lemma 5.3.1 Assume $g \leq \frac{2}{9}\epsilon$. Then if there is an ϵ-safe strategy for the continuous evader, there is a $3g$-safe strategy that travels only between grid-points.

Proof: Consider any path for the pursuer, and the corresponding ϵ-safe path (given by the ϵ-safe strategy) for the evader. We will approximate the continuous evader's path with a path traveling between grid-points. It takes the evader exactly $\tau = \frac{g}{v_e}$ time to traverse any edge of the grid-graph. By induction, it is easy to show that there is a grid-path that is no further than $v_e\tau = g$ away from the continuous path at all times $k\tau$ for k any integer. Furthermore, since the path is a close approximation at these discrete times, it is easy to show that the grid-path is no further than $v_e\tau + v_e\frac{\tau}{2} = \frac{3}{2}g$ at *all* times.

In particular, since the pursuer is at least ϵ away from the evader at all times, the distance from the pursuer to the grid-path evader must be at least $\epsilon - \frac{3}{2}g \geq \frac{9}{2}g - \frac{3}{2}g = 3g$. ∎

When the pursuer is restricted to traveling on the grid, there is a problem with the chosen grid-size being incompatible with the pursuer's velocity bound. For example, if the bounds are such that $v_p = \frac{3}{2}v_e$, then for each edge traversal of the evader, the pursuer can traverse one and a half edges. This does not fit into the simple discrete game defined earlier, so we make the following change. An additional game parameter s (a rational number called the scaling factor) is introduced, and the effect of this parameter is that the pursuer will make s moves for each move of the evader. For non-integer values of s, the meaning of this is unclear — if we are in round r of the game, we actually let the pursuer make $\lfloor rs \rfloor - \lfloor (r-1)s \rfloor$ moves. Using this scheme, the pursuer will always be within one grid-point of the place it would be if fractional moves along edges were allowed.

In our simulation, we let $s = \frac{v_p}{v_e}$. The following lemma ensures us that restricting the pursuer to a grid-path is not a great advantage for the evader.

Lemma 5.3.2 If both players are restricted to making movements between grid-points, then any winning strategy for the evader will also give a winning strategy against a continuous pursuer.

Proof: Consider any continuous path for the pursuer. If fractional edge traversals were allowed, then we could make an approximating path for the pursuer just as we did for the evader in lemma 5.3.1. This path is always within $\frac{3}{2}g$ of the continuous path, but due to the discretization of fractional moves, an additional error of g may be introduced. Thus at all times, the grid-path pursuer is within $\frac{5}{2}g$ of the continuous pursuer.

By the definition of the discrete game, we know that the distance between the discrete versions of the players is at least $3g$. Thus the continuous pursuer must be at least $\frac{1}{2}g$ away from the evader (i.e., the evader is not captured by the continuous pursuer). The strategy for the evader against a continuous adversary is therefore to make exactly the same moves as the discrete player would make against the discrete approximation of the continuous pursuer. This is clearly a winning strategy. ∎

The combination of the two preceding lemmas gives the proof of correctness for our approximation algorithm.

Theorem 5.3.3 If $g \leq \frac{2}{9}\epsilon$, then the discrete game will always find a winning strategy when there is an ϵ-safe strategy for the original game. Furthermore, any winning strategy found in the discrete game is also a winning strategy for the continuous game. The sequential time complexity of the approximation algorithm is $(n/\epsilon)^{O(1)}$.

Proof: By lemma 5.3.1, if there is an ϵ-safe strategy, then there is a $3g$-safe strategy that only uses grid moves. Restricting the pursuer to the grid means that at all times the pursuer is at least 3 edge traversals away from the evader — this is exactly the condition we need to satisfy for a winning strategy in the discrete game. The second claim in the theorem is exactly lemma 5.3.2.

The complexity of this algorithm is exactly that of the discrete game, with one minor modification. We only calculate grid-point adjacencies when a player is at the grid-point in question. To determine the adjacencies, we only need to do simple calculations on the obstacle descriptions — this can be done in $O(\log n)$ space, so the resulting complexity is $\mathrm{ASPACE}(d\log(1/\epsilon)+\log n)$, or $(n/\epsilon)^{O(d)}$ sequential time. For constant d, this is simply $(n/\epsilon)^{O(1)}$. ∎

Consider a generalization of this problem where the obstacles are allowed to move with constant velocity. The location of all obstacle coordinates can then be computed by a simple linear function of time, and it takes no more space than the original algorithm to compute vertex adjacencies.

Corollary 5.3.4 Given a pursuit game where obstacles are allowed to move with constant velocity, if $g \leq \frac{2}{9}\epsilon$, then we can approximately compute a winning strategy (in the sense of the last theorem) in sequential time $(n/\epsilon)^{O(1)}$.

In the following sections, we describe the discretization required for other forms of dynamics bounds, and prove closeness lemmas in each case.

5.3.2 Bounded L_∞-norm velocity and acceleration

Now we consider a pursuit game where the L_∞-norm of both velocity and acceleration are bounded. We use v_e and a_e to denote the velocity and acceleration bounds for the evader; v_p and a_p represent the velocity and acceleration bounds for the pursuer. The "grid" produced is not a regular grid in position space, as it was in the previous pursuit game. Instead, we construct a regular grid in velocity space, and the position grid consists of points corresponding to moves on the velocity grid. This is exactly the method used in kinodynamic planning, and we use the results of that work here; for more details on the exact method, see [12] and [23].

The grid construction (from [12]) is specified by the discrete time-step τ (along with the value of a_e). The following closeness lemma for this game is stated in terms of this time-step.

Lemma 5.3.5 Assume $\tau \leq \min(\frac{\epsilon}{20v_e}, \frac{v_e}{2a_e})$. Then if there is an ϵ-safe strategy for the continuous evader, there is a $\frac{3}{4}\epsilon$-safe strategy that travels only between grid-points.

Proof: The "Strong Tracking Lemma" of [23] states that if $\tau \leq \min(\frac{\eta_x}{5v_e}, \frac{v_e}{2a_e})$, then for any continuous trajectory meeting velocity and acceleration bounds v_e and a_e, respectively, there exists a grid trajectory that is always within distance η_x of the continuous trajectory. The lemma above follows by setting $\eta_x = \frac{\epsilon}{4}$. ∎

Now, of course, we must consider what happens when we discretize the continuous pursuer's trajectory. We can guarantee that the grid pursuer (with the scaling factor as before) stays within $\frac{\epsilon}{2}$ of the continuous pursuer's trajectory as long as $\tau \leq \min(\frac{\epsilon}{20v_p}, \frac{v_p}{2a_p})$. Notice that this means that when approximating both continuous players on the grid, there is always at least $\frac{\epsilon}{4}$ distance between the players, so the approximating trajectories correspond to a winning strategy for the evader. This fact, combined with the preceding lemma, gives the following theorem.

Theorem 5.3.6 Assume $\tau \leq \min(\frac{\epsilon}{20v_e}, \frac{v_e}{2a_e}, \frac{\epsilon}{20v_p}, \frac{v_p}{2a_p})$. Then the discrete game will always find a winning strategy when there is an ϵ-safe strategy for the continuous game. Furthermore, any winning strategy found in the discrete game is also a winning strategy for the continuous game. For a constant number of dimensions d, the sequential time complexity of the approximation algorithm is polynomial in $\frac{n}{\epsilon}$ and the parameters v_e, a_e, v_p, and a_p.

Proof: The size of the grid is given in [23], and it is upper bounded by

$$\left[\left(\frac{2v_{\max}}{a_{\max}\tau}+1\right)\left(\frac{D}{a_{\max}\tau^2}+1\right)\right]^d,$$

where $a_{\max} = \max(a_e, a_p)$, $v_{\max} = \max(v_e, v_p)$, and D is the diameter of the robot world. Since the time of our simulation is bounded by a polynomial in $(\frac{1}{\epsilon})^{O(1)}$, we can bound D by $\frac{v_{\max}}{\epsilon^{O(1)}}$. Furthermore, $\frac{1}{\tau}$ is polynomial in $\frac{1}{\epsilon}$, v_e, a_e, v_p, and a_p, so the result is an algorithm polynomial in these values, as stated in the theorem. ▮

5.3.3 Bounded L_2-norm velocity

As in kinodynamic planning, bounding the L_2-norm for dynamics bounds adds additional problems to approximation algorithms. Specifically, we need to be able to closely approximate the *direction* of motion (or acceleration, in the following section). In kinodynamic planning, we find an approximately optimal trajectory that takes time $(1+\epsilon)T_{\min}$, where $T_{\min}$ is the time required by the optimal trajectory. In the current pursuit game, we cannot allow this extra time because of the interaction with the pursuer — instead, we must allow the evader to exceed its dynamics bounds by a factor of ϵ. In other words, we allow the evader to have velocity as high as $(1+\epsilon)v_e$; the effect of this is identical to allowing the evader to take extra time, without the bad effects of allowing the evader more time.

In this section, we do not require the full power of the kinodynamic tracking lemma, but we use another theorem from chapter 4 to prove the following closeness lemma. We use a regular graph in position space with grid-spacing g, as in the case of bounded L_∞-norm velocity, but only parts of the grid are used. Refer back to chapter 4 for further details.

Lemma 5.3.7 Assume $g \leq \frac{\epsilon^2}{3(2+\epsilon)}$. Then if there is an ϵ-safe strategy for the continuous evader, there is a $3g$-safe strategy that travels only between grid-points.

Proof: By theorem 4.3.2, we can track the continuous trajectory of the evader such that at discrete times we stay within $\frac{4g}{\epsilon}$ of the continuous trajectory. Thus, at *all* times we can stay within $\frac{3}{2}\frac{4g}{\epsilon} = \frac{6g}{\epsilon}$ of the continuous trajectory. Since the continuous trajectory is ϵ-safe, this approximating trajectory is always at least $\epsilon - \frac{6g}{\epsilon}$ distance away from the pursuer and all obstacles. The proof is completed by noticing that

$$\epsilon - \frac{6g}{\epsilon} \geq \epsilon - \frac{2\epsilon}{2+\epsilon} = \frac{\epsilon^2}{2+\epsilon} \geq 3g. \quad ▮$$

The remainder of the proof of correctness for this case is almost identical to the bounded L_∞-norm velocity case, so is not spelled out here. The result is the following theorem.

Theorem 5.3.8 If $g \leq \frac{\epsilon^2}{3(2+\epsilon)}$, then the discrete game will always find a winning strategy when there is an ϵ-safe strategy for the original game. Furthermore, any winning strategy found in the discrete game is also a winning strategy for the continuous game. The sequential time complexity of the approximation algorithm is $(n/\epsilon)^{O(1)}$.

5.3.4 Bounded L_2-norm velocity and acceleration

When the L_2-norm of both velocity and acceleration are bounded, we use all of the ideas from the previous cases, in addition to the full L_2 tracking lemma from [25] (we could also use the tracking lemma from chapter 4, but the bounds of [25] are slightly better). As in the previous L_2-norm case, we must allow the evader to slightly exceed its dynamics bounds; specifically, we allow the approximating evader to have velocity $(1+\epsilon)v_e$ and acceleration $(1+\epsilon)^2 a_e$.

We state the following closeness lemma without proof — the proof is similar to the proof of lemma 5.3.5, but uses the kinodynamic tracking lemma from [25] (i.e., lemma 6.3 from [25]).

Lemma 5.3.9 Assume $\tau \leq \frac{\epsilon}{2\sqrt{a_e(48+2\epsilon)}}$. Then if there is an ϵ-safe strategy for the continuous evader, there is a $\frac{3}{4}\epsilon$-safe strategy that travels only between grid-points.

The final result for the bounded L_2-norm velocity and acceleration game is stated in the following theorem.

Theorem 5.3.10 Assume $\tau \leq \frac{\epsilon}{2\sqrt{a_{\max}(48+2\epsilon)}}$. Then the discrete game will always find a winning strategy when there is an ϵ-safe strategy for the continuous game. Furthermore, any winning strategy found in the discrete game is also a winning strategy for the continuous game. For a constant number of dimensions d, the sequential time complexity of the approximation algorithm is polynomial in $\frac{n}{\epsilon}$ and the parameters v_e, a_e, v_p, and a_p.

5.4 The Point-Robot Pursuit Game

In many other robotics problems, it can be assumed that the robot is a single point; more difficult problems are reduced to a point-robot problem by growing the obstacles according to the shape of the robot. In the pursuit game, there are two moving robots with possibly different shapes, so this obstacle-growing technique will not work.

In this section, we consider the pursuit game where each player is a single point, and show that this problem is computationally easier than the original pursuit game. We bound the velocity (but not acceleration) of each player, and further restrict the pursuer's velocity bound to be at least as high as the evader's velocity bound. Notice that this game is identical to the game used in the lower bound construction of section 5.2, except that the players are points. A key factor of the lower bound proof is that each player can only travel in its own passage, restricting the points of contention to several discrete locations. We cannot use this construction when the players are points, so the lower bound does not apply to this restriction. In fact, we can show that in this case, the problem is easily reducible to the shortest path problem.[3] We first prove the following theorem, showing the relationship between the point-robot pursuit game and the shortest path problem.

Theorem 5.4.1 In the point-robot pursuit game described above, there exists a winning strategy for the evader if and only if its fastest path to the goal is quicker than the pursuer's fastest path to the goal.

[3] The reduction used is a *Turing reduction*. In other words, we assume that there is an oracle for shortest path, and in particular, our reduction makes only a constant number of calls (two) on the oracle.

Proof: Let T_e (resp. T_p) be the minimum time required for the evader (resp., pursuer) to reach the goal. These can easily be calculated from the shortest *distance* path to the goal simply by dividing the distance by the maximum allowed velocity.

First assume that there is no winning strategy for the evader. Then, in particular, there is some trajectory for the pursuer that collides with the evader traveling along its fastest path to the goal. Let the time of collision be t_c. Ignoring the evader, the pursuer could follow its collision path until time t_c, and then follow the remaining segment of the evader's fastest path to the goal. This new path for the pursuer requires time

$$t_c + \frac{v_e}{v_p}(T_e - t_c) \leq t_c + T_e - t_c = T_e$$

(recall that the evader's velocity bound is no greater than the pursuer's bound, so $\frac{v_e}{v_p} \leq 1$). In other words, if the pursuer can always catch the evader, then the pursuer's fastest path to the goal takes no longer than the evader's fastest path to the goal.

Now assume that the purser's fastest path to the goal takes no longer than the evader's fastest path. Then a winning strategy for the pursuer is to simply move to the goal as fast as possible, effectively blocking the evader from the goal. In other words, there can be no winning strategy for the evader.

This proves both directions of the "if and only if" statement in the theorem. ▮

From the above theorem, the following corollary is obvious.

Corollary 5.4.2 The point-robot pursuit game described above is Turing reducible to the shortest path problem, and only two calls on the shortest path oracle are required.

5.5 Open Problems

There are many open problems in the area of pursuit games. One of the most interesting questions is to see what type of lower bound can be derived for pursuit games in which the L_∞ norm of velocity is bounded. Notice that in our lower bound construction, the position of the evader along the width of the passage acted as a "memory" of previous moves. With bounded L_∞ norm, the dimension representing the width of the passage is free to move with *no decrease* in the time it takes to travel through a basic trap. This allows the evader to "cheat" by performing invalid state transitions.

Another interesting open problem is to look at exact solutions for restricted games. For instance, many interesting problems have few (or no) obstacles. In addition, the lower bound is for three or more dimensions; are exact solutions possible in two dimensions?

Chapter 6

Conclusion

As promised in the introduction, we have proved significant results in two areas: circuit complexity (parallel algorithms) and computational robotics. These areas contain problems from both extremes of computational complexity. The problems studied in circuit complexity are typically very easy (computationally), and the problems from computational robotics are quite difficult, usually requiring approximation algorithms of the type we have given here.

Both areas provide many opportunities for future research. The problem of integer division using logspace-uniform boolean circuits (studied in chapter 2) has proved to be quite difficult. In particular, after many years of work, no one has been able to design a polynomial-size $O(\log n)$-depth circuit family for integer division. The iterative methods of chapter 2 work very well for reducing the size of the circuit family, but all known iterative methods seem to require $\Omega(\log n \log\log n)$ depth.

Examining the power of threshold circuits is also an intriguing and important question. In many models of computation, results about the power of the model are possible by first relating the model to a well-understood mathematical concept. The results of chapter 3 are hopefully a first step to understanding the exact power of constant-depth threshold circuits.

Computational robotics, the subject of chapters 4 and 5, is a relatively new field, and has many open problems. While many problems from robotics are computationally intensive, the approximation algorithms from chapters 4 and 5 show that even very difficult problems can be at least approximately solved. For many problems, approximation seems to be the only hope of handling the problem in practice — evidence of this comes from the EXPTIME-hard lower bound for pursuit games in chapter 5, and the corresponding approximation algorithm.

The preceding paragraphs suggest directions for future research in the areas of circuit complexity and computational robotics, as well as reflect some of the future research interests of the author. This is a very fast-paced time for computer science, and many significant advances can be expected in the coming decade.

Bibliography

[1] D. H. Ackley, G. E. Hinton, and T. J. Sejnowski. A Learning Algorithm for Boltzmann Machines. *Cognitive Sciences*, 9:147–169, 1985.

[2] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Anaslysis of Computer Algorithms*. Addison-Wesley Publishing Company, Reading, MA, 1974.

[3] E. Allender. A Note on the Power of Threshold Circuits. In *30th IEEE Symposium on Foundations of Computer Science*, pages 580–584, 1989.

[4] H. Alt. Comparing the Combinational Complexities of Arithmetic Functions. *JACM*, 35(2):447–460, April 1988.

[5] J. L. Balcazar, J. Diaz, and J. Gabarro. *Structural Complexity I*. Springer-Verlag, Berlin, 1988.

[6] D. A. Barrington. Bounded-Width Polynomial-Size Branching Programs Recognize Exactly Those Languages in NC^1. *JCSS*, 38:150–164, 1989.

[7] P. W. Beame, S. A. Cook, and H. J. Hoover. Log Depth Circuits For Division And Related Problems. *SIAM J. Comput.*, 15(4):994–1003, November 1986.

[8] A. Borodin. On Relating Time And Space To Size And Depth. *SIAM J. Comput.*, 6(4):733–744, December 1977.

[9] A. E. Bryson and Y.-C. Ho. *Applied Optimal Control*. Hemisphere Publishing Corporation, Washington, D. C., 1975.

[10] R. L. Burden and J. D. Faires. *Numerical Analysis*. Prindle, Weber, & Schmidt, Boston, MA, 1985.

[11] J. Canny. Some Algebraic and Geometric Computations in PSPACE. In *20th ACM Symposium on Theory of Computing*, pages 460–467, 1988.

[12] J. Canny, B. Donald, J. Reif, and P. Xavier. On The Complexity of Kinodynamic Planning. In *29th IEEE Symposium on Foundations of Computer Science*, pages 306–316, 1988.

[13] J. Canny, A. Rege, and J. Reif. An Exact Algorithm for Kinodynamic Planning in the Plane. In *6th Annual Symposium on Computational Geometry*, pages 271–280, 1990.

[14] J. Canny and J. Reif. New Lower Bound Techniques for Robot Motion Planning Problems. In *28th IEEE Symposium on Foundations of Computer Science*, pages 49–60, 1987.

[15] A. K. Chandra, S. J. Fortune, and R. J. Lipton. Unbounded Fan-In Circuits and Associative Functions. In *ACM Symposium on Theory of Computing*, pages 52–60, 1983.

[16] A. K. Chandra, D. C. Kozen, and L. J. Stockmeyer. Alternation. *JACM*, 28(1):114–133, January 1981.

[17] K. L. Clarkson. Approximation Algorithms for Shortest Path Motion Planning. In *19th ACM Symposium on Theory of Computing*, pages 56–65, 1987.

[18] G. Collins. Quantifier Elimination for Real Closed Fields by Cylindric Algebraic Decomposition. In *2nd GI Conference on Automata Theory and Formal Languages*, pages 134–183, 1975.

[19] S. A. Cook. *On The Minimum Computation Time of Functions.* PhD thesis, Harvard University, Cambridge, MA, 1966.

[20] J. M. Cooley and J. W. Tukey. An Algorithm for the Machine Calculation of Complex Fourier Series. *Math. Comput.*, 19:297–301, 1965.

[21] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* The MIT Press, Cambridge, MA, 1990.

[22] B. Donald. The Complexity of Planar Compliant Motion Planning Under Uncertainty. In *4th ACM Symp. on Computational Geometry*, pages 309–318, 1988.

[23] B. Donald and P. Xavier. A Provably Good Approximation Algorithm for Optimal-Time Trajectory Planning. In *IEEE Int. Conf. on Robotics and Automation*, pages 958–963, 1989.

[24] B. Donald and P. Xavier. Near-Optimal Kinodynamic Planning for Robots With Coupled Dynamics Bounds. In *IEEE Int. Symp. on Intelligent Controls*, 1989.

[25] B. Donald and P. Xavier. Provably Good Approximation Algorithms for Optimal Kinodynamic Planning for Cartesian Robots and Open Chain Manipulators. Technical Report TR-1095, Cornell University, February 1990.

[26] J. A. Feldman and D. H. Ballard. Connectionist Models and Their Properties. *Cognitive Science*, 6:205–254, 1982.

[27] S. Fortune and G. Wilfong. Planning Constrained Motion. In *20th ACM Symposium on Theory of Computing*, pages 445–459, 1988.

[28] J. B. Fraleigh. *A First Course in Abstract Algebra.* Addison-Wesley Publishing Company, Reading, MA, 1982.

[29] M. Furst, J. B. Saxe, and M. Sipser. Parity, Circuits, and the Polynomial-time Hierarchy. *Math. Sys. Theory*, 17:13–27, 1984.

[30] S. Goodwin-Johansson. Circuit to Perform Variable Threshold Logic, January 1990. U.S. Patent number 4896059.

[31] G. H. Hardy and E. M. Wright. *An Introduction to the Theory of Numbers.* Oxford at the Clarendon Press, Oxford, England, 1979.

[32] J. Hastad and T. Leighton. Division in $O(\log n)$ Depth Using $O(n^{1+\epsilon})$ Processors, 1986. Unpublished note.

[33] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addision-Wesley Publishing Company, Reading, MA, 1979.

[34] J. J. Hopfield. Neural Networks and Physical Systems with Emergent Collective Computational Abilities. *Proceedings of the National Academy of Sciences, USA*, 79:2554–2558, 1982.

[35] O. H. Ibarra and S. Moran. Probabilistic Algorithms for Deciding Equivalence of Straight-Line Programs. *JACM*, 30(1):217–228, January 1983.

[36] P. Jacobs and J. Canny. Planning Smooth Paths for Mobile Robots. In *IEEE Int. Conf. on Robotics and Automation*, pages 2–7, 1989.

[37] P. Jacobs, G. Heinzinger, J. Canny, and B. Paden. Planning Guaranteed Near Time-Optimal Trajectories for a Manipulator in a Cluttered Workspace. Technical Report ESRC 89-20/RAMP 89-15, Engineering Systems Research Center, University of California, Berkeley, 1989.

[38] R. M. Karp and R. J. Lipton. Some Connections Between Nonuniform and Uniform Complexity Classes. In *12th ACM Symposium on Theory of Computing*, pages 302–309, 1980.

[39] T. Lozano-Perez and M. A. Wesley. An Algorithm for Planning Collision-free Paths Among Polyhedral Obstacles. *CACM*, 22(10):560–570, October 1979.

[40] K. Melhorn and F. P. Preparata. Area-time Optimal Division for $T = \Omega((\log n)^{1+\epsilon})$. In *3rd Annual Symposium on Theoretical Aspects of Computer Science*, pages 341–352, 1986. in LNCS vol. 210.

[41] C. ÓDúnlaing. Motion Planning with Inertial Constraints. *Algorithmica*, 2(4):431–475, 1987.

[42] C. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Prentice Hall, Englewood Cliffs, NY, 1982.

[43] C. H. Papadimitriou. An Algorithm for Shortest-Path Motion in Three Dimensions. *Information Processing Letters*, 20(5):259–263, June 1985.

[44] I. Parberry. *Parallel Complexity Theory*. Pitman Publishing, London, 1987.

[45] I. Parberry and G. Schnitger. Parallel Computation with Threshold Functions. In *Proceedings of Conference on Structure in Complexity Theory*, pages 272–290, 1986. in LNCS 223.

[46] N. Pippenger. The Complexity of Computations by Networks. *IBM J. Res. Dev.*, 31(2):235–243, March 1987.

[47] J. H. Reif. Complexity of the Mover's Problem and Generalizations. In *20th IEEE Symposium on Foundations of Computer Science*, pages 421–427, 1979.

[48] J. H. Reif. Logarithmic Depth Circuits For Algebraic Functions. In *24th IEEE Symposium on Foundations of Computer Science*, pages 138–145, 1983.

[49] J. H. Reif. The Complexity of Two-Player Games of Incomplete Information. *JCSS*, 29(2):274–301, October 1984.

[50] J. H. Reif. A Survey on Advances in the Theory of Computational Robotics. In K. S. Narendra, editor, *Adaptive and Learning Systems: Theory and Applications*. Plenum Press, New York, NY, 1986.

[51] J. H. Reif. Logarithmic Depth Circuits For Algebraic Functions. *SIAM J. Comput.*, 15(1):231–242, February 1986.

[52] J. H. Reif. On Threshold Circuits and Polynomial Computation. In *2nd Conference on Structure in Complexity Theory*, pages 118–123, 1987.

[53] J. H. Reif and M. Sharir. Motion Planning in the Presence of Moving Obstacles. In *26th IEEE Symposium on Foundations of Computer Science*, pages 144–154, 1985.

[54] J. H. Reif and J. A. Storer. Minimizing Turns for Discrete Movement in the Interior of a Polygon. *IEEE Journal of Robotics and Automation*, RA-3(3):182–193, June 1987.

[55] J. H. Reif and J. A. Storer. 3-Dimensional Shortest Paths in the Presence of Polyhedral Obstacles. In *Symposium on Mathematical Foundations of Computer Science*, pages 85–92, 1988. in LNCS vol. 324.

[56] J. H. Reif and S. R. Tate. Optimal Size Integer Division Circuits. In *21st ACM Symposium on Theory of Computing*, pages 264–273, 1989.

[57] J. H. Reif and S. R. Tate. Approximate Kinodynamic Planning Using L_2-norm Dynamics Bounds. Technical Report CS-1990-13, Duke University Department of Computer Science, 1990.

[58] J. H. Reif and S. R. Tate. Optimal Size Integer Division Circuits. *SIAM J. Comput.*, 19(5):912–924, 1990.

[59] G. Sahar and J. M. Hollerbach. Planning of Minimum-Time Trajectories for Robot Arms. In *IEEE Int. Conf. on Robotics and Automation*, pages 751–758, 1985.

[60] A. Schönhage and V. Strassen. Schnelle Multiplikation großer Zahlen. *Computing*, 7:281–292, 1971.

[61] J. T. Schwartz and M. Sharir. On the Piano Movers' Problem: I. The Case of a Rigid Polygonal Body Moving Amidst Polygonal Barriers. *Comm. Pure and Appl. Math.*, 36:345–398, 1983.

[62] N. Shankar and V. Ramachandran. Efficient Parallel Circuits And Algorithms For Division. *Info. Proc. Letters*, 29(6):307–313, December 1988.

[63] M. Sharir and A. Schorr. On shortest paths in polyhedral spaces. In *16th ACM Symposium on Theory of Computing*, pages 144–153, 1984.

[64] Z. Shiller and S. Dubowsky. Global Time Optimal Motions of Robotic Manipulators in the Presence of Obstacles. In *IEEE Int. Conf. on Robotics and Automation*, pages 370–375, 1988.

[65] K. G. Shin and N. D. McKay. Selection of Near-Minimum Time Geometric Paths for Robotic Manipulators. In *Proc. Amer. Contr. Conf.*, pages 346–355, 1985.

[66] K. Sutner and W. Maas. Motion Planning Among Time-Dependent Obstacles, 1985. preprint.

[67] J. F. Traub. *Iterative Methods For The Solution of Equations.* Chelsea Publishing Co., New York, NY, 1964.

[68] A. Yao. Separating the Polynomial-Time Hierarchy by Oracles. In *26th IEEE Symposium on Foundations of Computer Science*, pages 1–10, 1985.