Algorithms for String and Graph Layout

by

Alantha Newman

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2004

Author .....
Department of Electrical Engineering and Computer Science
September 10, 2004

Certified by .....
Santosh S. Vempala
Associate Professor of Applied Mathematics
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Algorithms for String and Graph Layout

by

Alantha Newman

Submitted to the Department of Electrical Engineering and Computer Science
on August 31, 2004, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

Many graph optimization problems can be viewed as graph layout problems. A layout of a graph is a geometric arrangement of the vertices subject to given constraints. For example, the vertices of a graph can be arranged on a line or a circle, on a two- or three-dimensional lattice, etc. The goal is usually to place all the vertices so as to optimize some specified objective function.

We develop combinatorial methods as well as models based on linear and semidefinite programming for graph layout problems. We apply these techniques to some well-known optimization problems. In particular, we give improved approximation algorithms for the string folding problem on the two- and three-dimensional square lattices. This combinatorial graph problem is motivated by the protein folding problem, which is central in computational biology. We then present a new semidefinite programming formulation for the linear ordering problem (also known as the maximum acyclic subgraph problem) and show that it provides an improved bound on the value of an optimal solution for random graphs. This is the first relaxation that improves on the trivial "all edges" bound for random graphs.

Thesis Supervisor: Santosh S. Vempala
Title: Associate Professor of Applied Mathematics

Acknowledgments

Most of all, I would like to thank Santosh Vempala for being my advisor for the past five years. Santosh spent countless hours with me early in my graduate career when I was starting to work on research problems. He has always given me excellent problems to work on and he has always been very generous with his insights and intuition with respect to these problems. Santosh has a special ability to explain your own ideas back to you much more clearly than you explained them to him. My favorite class in graduate school was Santosh's undergraduate combinatorial optimization course, which I took during my second year at MIT. He is great lecturer and this class made it clear to me what area of computer science I was most interested in.

I would like to thank Madhu Sudan for all of his support and advice during my time at MIT and for being on my thesis committee. I would like to thank Michel Goemans for running his weekly group meetings, for all of his help and advice, for solving the maxcut problem, and for being on my thesis committee. I would like to thank David Karger for being on my thesis committee and for making such useful suggestions about how to write this thesis.

I would like to thank Charles Leiserson, who was my graduate counselor and has helped me a great deal.

I thank all the students in the theory group for their support. I especially thank Anna Lysyanskaya for all of her sincere advice, support and friendship during the past six years. I thank Prahladh Harsha, my officemate of six years, for all of the geography and history lessons on the Far East and for being an encyclopedia when it comes to answering technical questions. I thank Nitin Thaper who was my officemate for three years for the everyday conversations we had about everyday life. I would also like to thank Matthias Ruhl for being a good friend and for collaborating with me on the results in Chapters 2 and 5 of this thesis. I would like to thank Mike Rosenblum for all of his friendly support during these last months of thesis writing. I would also like to thank John Dunagan for his patience in the many technical discussions that I had with him early on in graduate school.

I would like to especially thank my big sister Edith Newman, for being such a good friend, and for drawing Figures 5-1, 5-5, 5-3 and 5-4.

I would like to thank Dr. Bill Hart and Dr. Bob Carr for giving me a summer job at Sandia and for collaborating on the results in Section 4.4 of this thesis.

I would like to thank Mona Singh and Martin Farach-Colton for encouraging me to attend grad school and MIT in particular.

I thank Christina Olsson for being a good friend and roommate and making the past three years so much fun. I thank Maja Razlog and Alice Wang and Anna for letting me be their fourth roommate at 25 Willow Ave.

I thank my mother—who went back to school after thirty years and raising three daughters to complete her own Ph.D in Early Childhood Education during ten years

of part-time study—for all of the time and the sacrifices she made for me and my sisters. I thank my father for his unique insights on every topic I raise and for his special sense of humor and for always supporting me in his own way.

Finally, I would like to thank Be Blackburn, Joanne Talbot-Hanley, Peggy Carney and Marilyn Pierce for all of their help.

Contents

Chapter 1

Introduction

Graph layout problems involve arranging the vertices and edges of a given graph subject to specified constraints. For example, by definition a planar graph can be drawn in a two-dimensional plane such that no edges cross. There are several algorithms for finding such a layout of a planar graph; a linear-time algorithm was given by Booth and Lueker [BL76]. Another well-studied graph layout problem is finding a layout of a *non-planar* graph that *minimizes* the number of edge crossings.

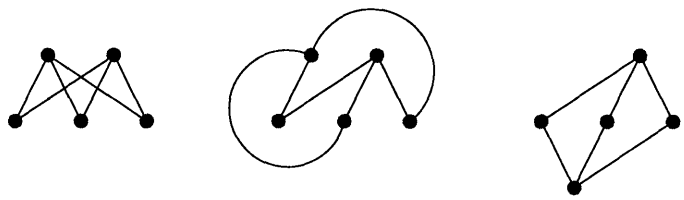

Figure 1-1: Planar graph drawing is the problem of laying out a planar graph so that no edges cross.

In this thesis, we focus on layouts of graphs defined as geometric arrangements of the vertices on a line, lattice, circle, etc. The goal is usually to optimize a specified objective function. Two examples of such vertex layout problems are the *maximum linear arrangement* problem and the *minimum bandwidth* problem. In each problem, the input is an undirected graph $G = (V, E)$. Each vertex $i \in V$ is assigned a unique label $\ell(i)$ from the set of integers $\{1, 2, \ldots, n\}$. The goal of the maximum linear arrangement problem is to assign the labels to the vertices so as to maximize the sum $\sum_{ij \in E} |\ell(i) - \ell(j)|$, i.e. maximize the sum of the lengths of the edges when arranged on a line according to their labels. The goal of the minimum bandwidth problem is to

assign the labels to the vertices so as to minimize the maximum value of $|\ell(i) - \ell(j)|$ over all edges $(i, j) \in E$, i.e. minimize the length of the maximum length edge.

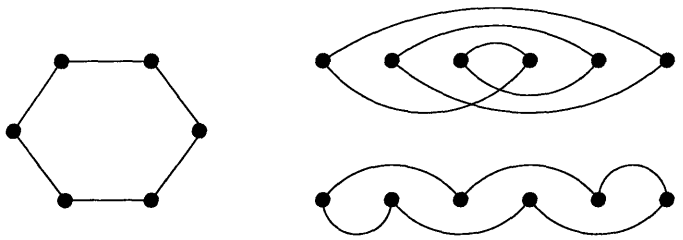

Figure 1-2: A 6-cycle with optimal vertex orderings for the maximum linear arrangement and minimum bandwidth problems.

Many graph optimization problems can be described as finding a maximum/minimum weight subset of edges (subgraph) with a particular property. Alternatively, many of these graph optimization problems can be described as a vertex layout problems, in which the placement of each vertex is chosen from a discrete set of possible positions. The goal is to place or lay out the vertices so as to optimize some specified objective function.

One of the most fundamental graph optimization problems is the maximum cut (maxcut) problem. Suppose we are given an undirected, weighted graph $G = (V, E)$. Two possible statements of the maxcut problem are:

(i) Find the maximum weight bipartite subgraph.

(ii) Partition the vertices into two disjoint sets $(S, \overline{S})$ so as to maximize the weight of the edges crossing the cut.

Figure 1-3: Two ways of defining the maxcut problem.

These are equivalent optimization problems. Each suggests a different way of communicating a solution. For example, suppose we want to record a known solution for the maxcut of a given graph. Two possible ways to do this are:

(i) Give a vector $\mathbf{x} \in \{0,1\}^{|E|}$, where x_{e_i} is a 1 if the edge e_i crosses the cut and 0 otherwise.

(ii) Give a vector $\mathbf{x} \in \{0,1\}^{|V|}$, where x_{v_i} is a 1 if the vertex v_i is in S and 0 if it is in $\overline{S}$.

Linear and semidefinite programming methods are frequently used to approximate the optimal solution for a combinatorial optimization problem by defining a polytope that closely approximates the convex hull of integer solutions. So the question is, what set of integer solutions should we approximate? Which way should we view the problem?

The problems and methods discussed in this thesis are motivated by viewing graph optimization problems as graph layout problems. In some cases, the most natural problem statement is in terms of graph layout. In other cases, this alternative viewpoint provides new insights into the problem.

1.1 Problems and Results

In this thesis, we will focus primarily on two combinatorial graph layout problems. The first problem is known as the *string folding* problem. The second is known as the *linear ordering* problem. The goal of each problem is to arrange the vertices of an input graph subject to specified constraints so as to maximize a given objective function. In this section, we will precisely define these problems and provide background and motivation. Additionally, we outline our new results. In Section 1.2, we explain the layout of this thesis.

1.1.1 String Folding

The first problem we address is an optimization problem called the *string folding* problem. The input graph can be viewed as a string; it is an undirected graph in which each vertex except for two end vertices has degree exactly two. Each end vertex has degree exactly 1. Each vertex in this input graph is labeled as either a '0' or a '1'. Additionally, we are given a lattice. For example, suppose we are given a two-dimensional square lattice in which one lattice point is arbitrarily assigned to be the origin with coordinates (0,0) and the rest of the lattice points are labeled accordingly. We say a vertex from the input graph is *placed* on a lattice point (x, y) if that vertex is assigned to lattice point (x, y). A *folding* of such an input graph corresponds to placing the vertices of the graph on lattice points subject to the following three constraints:

(i) Each lattice point can have at most one vertex placed on it.

(ii) Each vertex must be placed on *some* lattice point.

(iii) Adjacent vertices in the string must be placed on adjacent lattice points.

For example, suppose vertex i and $i+1$ are adjacent in the input graph. On a 2D square lattice, if vertex i is placed on lattice point (x, y), then vertex $i+1$ must be placed on one of four possible lattice points: $(x \pm 1, y)$ or $(x, y \pm 1)$. Thus, in a valid folding of a string, the string is laid out on the lattice so that it does not cross itself. Such a folding is commonly referred to as a *self-avoiding walk*. When the problem is defined for a particular lattice, part of the problem definition is to define which pairs of lattice points are "adjacent". For example, on the 2D square lattice, we will say each lattice point has four neighbors, but it is possible to define the problem such that lattice points diagonally across from each other, i.e. (x, y) and $(x+1, y+1)$, are neighbors.

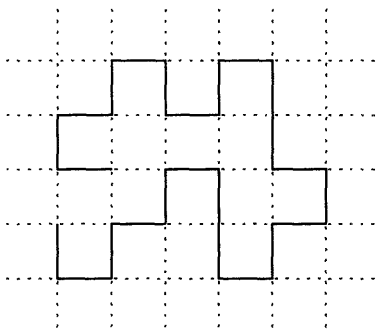

Figure 1-4: A so-called self-avoiding walk—a string forms a pattern that does not cross itself.

The goal of the string folding problem is to find a valid folding of a given input graph/string that maximizes the number of pairs of vertices both labeled 1 that occupy adjacent lattice points. Such pairs of vertices, i.e. pairs of vertices both labeled 1 that occupy adjacent lattice points, are called *contacts*. By the definition of a valid folding, two vertices labeled 1 that are adjacent in the input graph must occupy adjacent lattice points in any valid folding. Such pairs are not considered to be contacts.

For example, suppose the input graph corresponds to the string 101010101001010101. Then the folding shown in Figure 1-5 results in eight pairs of vertices labeled 1 that occupy adjacent lattice points. This folding yields the maximum possible number of contacts for this string over all foldings on the 2D square lattice. The vertices labeled 1 are denoted by black dots and the vertices labeled 0 are denoted by white or unfilled dots.

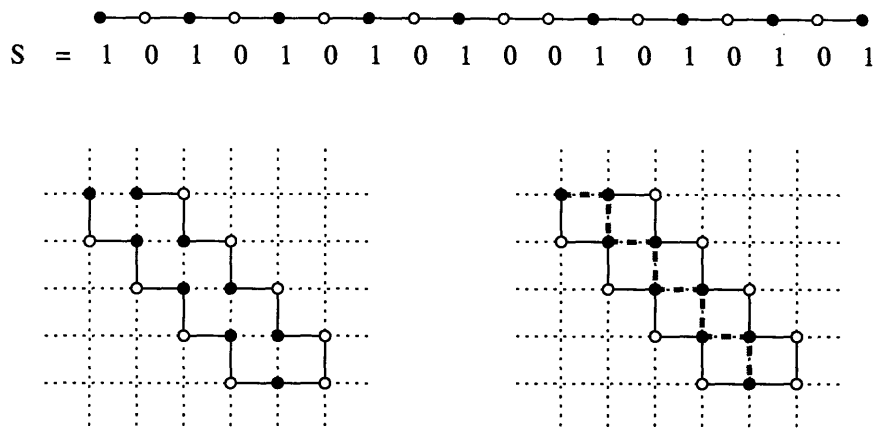

Figure 1-5: An optimal folding for the string $S = 101010101001010101$. The eight contacts are marked by thick (red) dashed lines.

Motivation

The string folding problem is motivated by the protein folding problem, which is a central problem in computational biology. A protein is a sequence of amino acid residues ranging in length from hundreds to thousands of residues. Shorter amino acid chains are called peptides. There are about 20 types of amino acids. The three-dimensional shape of a protein or peptide determines its function.

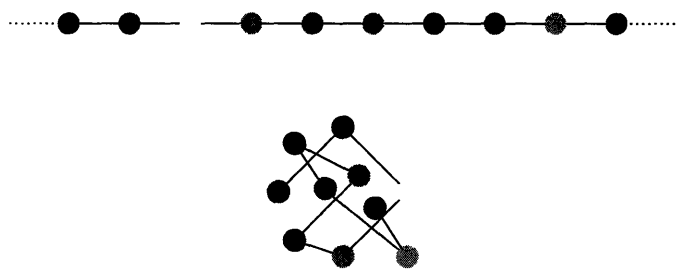

Figure 1-6: A protein is composed of a one-dimensional amino acid sequence and folds to a three-dimensional shape that determines its function.

In 1985, Ken Dill [Dil85, Dil90] introduced a simplified model of protein folding called the Hydrophobic-Hydrophilic (HP) model. This model abstracts the dominant force in protein folding: the hydrophobic interaction. The hydrophobicity of an amino acid is its propensity to avoid water. It is known that proteins contain tightly clustered cores of hydrophobic amino acids that avoid being close to the surface, which comes into contact with water. In the HP model, each amino acid is classified as an H (hydrophobic) or a P (hydrophilic or polar).

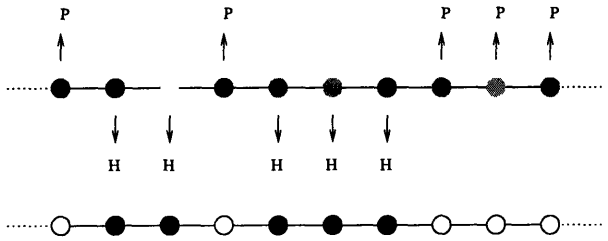

Figure 1-7: Each amino acid is classified as either an H or a P depending on its degree of hydrophobicity.

The problem is further simplified by restricting the foldings to a two-dimensional (2D) or three-dimensional (3D) square lattice rather than three-dimensional space. The goal of the protein folding problem in the HP model is to find a folding of an input string of H's and P's that maximizes the number of pairs of adjacent H's, i.e. H-H contacts. This is exactly the combinatorial problem that we called the string folding problem.

Figure 1-8: Two-dimensional and three-dimensional HP models.

Background

The HP lattice model is a significant simplification of the protein folding problem but nevertheless computationally difficult. In 1995, Hart and Istrail presented approximation algorithms for the string folding problem on the two-dimensional and three-dimensional square lattices [HI96]. If an optimization problem is shown to be NP-hard, then a typical approach is to give an *approximation algorithm* since it is commonly believed that the existence of efficient algorithms for NP-hard optimization problems is unlikely. A ρ-approximation algorithm is a polynomial-time algorithm that produces a solution of value at least ρ times the optimal value. Hart and Istrail presented the string folding problem to the theoretical computer science community and gave approximation algorithms for the problem on the two-dimensional and three-dimensional square lattice before either version was known to be NP-hard. Their linear-time algorithms guaranteed foldings in which the number of contacts was $\frac{1}{4}$ and $\frac{3}{8}$ of the optimal number of contacts for the 2D and 3D problems, respectively.

It was a major open problem to show that the string folding problem on the 2D or 3D square lattices is NP-hard or give an efficient exact algorithm for it. In 1998, the 2D string folding problem was shown to be NP-hard by Crescenzi, Goldman, Papadimitriou, Piccolboni and Yannakakis [CGP+98] and the 3D string folding problem was shown to be NP-hard by Berger and Leighton [BL98]. In 1999, Mauri, Piccolboni, and Pavesi presented another factor $\frac{1}{4}$-approximation algorithm for the 2D problem based on dynamic programming [MPP99]. They claimed that their algorithm performed better than that of Hart and Istrail in practice. Additionally, Agarwala et al. gave approximation algorithms for the string folding problem on the 2D and 3D triangular lattice with approximation guarantees of slightly better than $\frac{1}{2}$ [ABD+97]. It is not known if the string folding problem on the 2D or 3D triangular lattice is NP-hard.

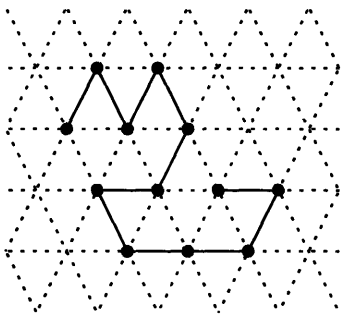

Figure 1-9: A valid folding on the 2D triangular lattice.

Each of the approximation algorithms referred to above for the string folding problem on the 2D or 3D square lattice use a simple combinatorial upper bound on the optimal number of contacts. Hart and Istrail [HI96] and Mauri et al. [MPP99] showed that their algorithm always achieves at least $\frac{1}{4}$ as many contacts as demonstrated by this combinatorial upper bound.

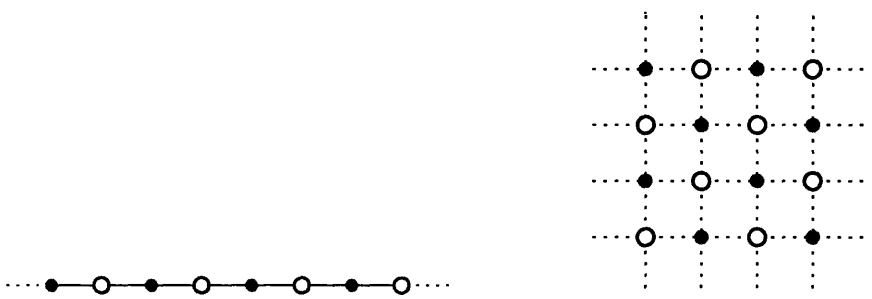

Figure 1-10: The square lattice is a bipartite graph.

Consider an input graph/string to the string folding problem. If we fix an arbitrary

endpoint to be the first vertex on the string, then each vertex has an odd index or an even index. The square lattice is a bipartite graph, i.e. the lattice points can be divided into two sets, each containing no edges. In a valid folding, each odd vertex is assigned to a lattice point from one of these sets and each even vertex is assigned to a lattice point in the other set. Thus, 1's with even indices (even-1's) in the string can only have contacts with 1's in odd indices (odd-1's) in the string. Moreover, each lattice point has four neighboring lattice points and each vertex—except for the two vertices with degree 1—can have at most two contacts in a valid folding since two of its neighboring lattice points will be occupied by adjacent vertices on the string and therefore cannot form contacts. Let S denote the given input string. Then $\mathcal{O}[S]$ is the number of 1's in S that have odd indices and $\mathcal{E}[S]$ is the number of 1's in S that have even indices. Let $M_2[S]$ be the maximum number of contacts for any folding over all possible foldings of the string S on the 2D square lattice. An upper bound on the maximum number of contacts is:

$$M_2[S] \leq 2 \cdot \min\{\mathcal{O}[S], \mathcal{E}[S]\} + 2. \tag{1.1}$$

Hart and Istrail prove that their approximation algorithm for the 2D string folding problem achieves at least $\min\{\mathcal{O}[S], \mathcal{E}[S]\}/2$ contacts, resulting in a factor $\frac{1}{4}$-approximation algorithm. As in the 2D case, the 3D square lattice is also bipartite. Each lattice point has six neighbors. If a vertex (that is not an endpoint) is placed on a particular lattice point, then two out of six neighboring lattice points will be occupied by neighboring vertices from the string. Thus, each 1 in the input string can have at most four contacts. Let $M_3[S]$ be the maximum number of contacts for any folding over all possible foldings of the string S on the 3D square lattice. The upper bound for the 3D string folding problem is therefore:

$$M_3[S] \leq 4 \cdot \min\{\mathcal{O}[S], \mathcal{E}[S]\} + 2. \tag{1.2}$$

Agarwala et al. argue that the triangular lattice is a more realistic model of protein folding because it does not have this "parity problem", i.e. vertices in odd positions need not exclusively form contacts with vertices in even positions. However, the square lattice model seems potentially simpler since once a vertex is placed on the lattice, there are fewer possible positions for each neighboring vertex and has been very well studied.

New Results

Improving the approximation guarantees of $\frac{1}{4}$ and $\frac{3}{8}$ given by Hart and Istrail for the 2D and 3D string folding problems, respectively, have been open problems for many years. In this thesis, we give a new combinatorial factor $\frac{1}{3}$-approximation algorithm

for the 2D string folding problem [New02]. Our algorithm runs in linear time and outputs a folding that yields $\frac{1}{3}$ as many contacts as the combinatorial upper bound given in Equation (1.1).

We also examine the combinatorial upper bound for the 2D string folding problem specified in Equation (1.1). We show that this bound cannot be used to obtain an approximation guarantee of more than $\frac{1}{2}$ [New02]. We show this by demonstrating a family of strings such that for any string S in the family, an optimal folding of S achieves at most $(1 + o(1)) \min\{\mathcal{O}[S], \mathcal{E}[S]\}$ contacts.

Additionally, we examine a simple linear programming formulation for the 2D string folding problem and analyze the bound it provides on the value of an optimal solution [CHN03]. We show that the upper bound it provides is no more than three times the value of an optimal solution, although we are not aware of a string for which the linear programming bound is actually this much larger than the value of an optimal folding. The best gap we can construct is 2: we give an example in which the bound provided by this linear program can be twice as large as optimal.

Next, we consider the 3D string folding problem. We give another $\frac{3}{8}$-approximation algorithm for the 3D folding problem based on new geometric ideas [NR04]. The $\frac{3}{8}$-approximation algorithm of Hart and Istrail [HI96] produces a folding with $\frac{3}{8}OPT - \Theta(\sqrt{\mathcal{O}[S]})$ contacts. Our algorithm produces a folding with $\frac{3}{8}OPT - c$ contacts, where c is a small positive integer. Thus, our algorithm improves on the absolute approximation guarantee of Hart and Istrail.

We show that modifying this new algorithm leads to an improved approximation guarantee of $\frac{3}{8} + \epsilon$ for the 3D string folding problem, where ϵ is a small positive constant [NR04]. These modifications yield two new approximation algorithms for the 3D folding problem. Both of these algorithms exploit properties of the string rather than (additional) new geometric ideas: Both have approximation guarantees expressed in terms of the number of transitions in the input string S from sequences of 1's in odd positions to sequences of 1's in even positions. We refer to the number of such transitions in a string S as $\delta(S)$. Our algorithms have approximation guarantees of $(.439 - \Theta(\delta(S)/|S|))$ and $(.375 + \Theta(\delta(S)/|S|))$.

Both of the factor $\frac{3}{8}$-approximation algorithms referred to previously divide the input string S into two substrings, one substring containing at least half of the 1's with even indices and one substring containing at least half of the 1's with odd indices. They produce a folding in which all of the even-1's from one of the substrings has at least three contacts and all of the odd 1's from the other substring has at least three contacts, resulting in a $\frac{3}{8}$-approximation algorithm.

In our improved algorithm, the resulting folding guarantees that there are contacts using *both* odd-1's and even-1's from each of the two substrings. However, in order to use odd-1's and even-1's from the same substring, it would be convenient if they form a predictable pattern. Thus, one of the main tools we use is a new theorem on binary strings. We call a binary string in $\{a, b\}^*$ *block-monotone* if every maximal

sequence of consecutive a's is immediately followed by a block of at least as many consecutive b's. Suppose a given binary string has the property that every suffix of the string has at least as many b's as a's. What is the longest block-monotone subsequence in the string? We obtain a non-trivial lower bound on the length of a block-monotone subsequence and we show a connection between this problem and the 3D string folding problem.

1.1.2 Linear Ordering

The second problem we address is a well-studied graph optimization problem called the *linear ordering* problem. Given a complete weighted directed graph, $G = (V, A)$, the goal of the linear ordering problem is to find an ordering of the vertices that maximizes the weight of the forward edges. A *vertex ordering* is defined as a mapping of each vertex $i \in V$ to a unique label $\ell(i)$, where $\ell(i)$ is an integer. An edge $(i, j) \in A$ is a *forward edge* with respect to an ordering if $\ell(i) < \ell(j)$. For the linear ordering problem, we can assume without loss of generality that the labels are integers chosen from the range $\{1, 2, \ldots, n\}$, where $n = |V|$. The linear ordering problem is also known as the *maximum acyclic subgraph* problem, which is defined as follows: Given a weighted, directed graph, find the subgraph of maximum weight that contains no directed cycles. The forward edges in any linear ordering comprise an acyclic subgraph and a topological sort of an acyclic subgraph yields a linear ordering of the vertices in which all edges in the acyclic subgraph are forward edges. Thus, these two problems are equivalent.

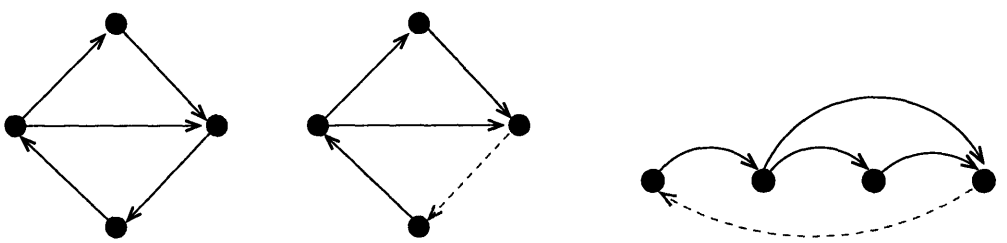

Figure 1-11: A maximum acyclic subgraph of a directed graph corresponds to a linear ordering of its vertices.

Although the problem is NP-hard [Kar72], it is easy to estimate the optimum to within a factor of $\frac{1}{2}$: In any ordering of the vertices, either the set of forward edges or the set of backward edges accounts for at least half of the total edge weight. It is not known whether the maximum can be estimated to a better factor using a polynomial-time algorithm. The outstanding open problem with respect to the linear ordering problem is finding a ρ-approximation algorithm for the problem where ρ is a constant greater than $\frac{1}{2}$. Approximating the problem to within better than $\frac{65}{66}$ is NP-hard [NV01].

Motivation

The linear ordering problem is a fundamental graph optimization problem that has applications in the fields of scheduling, economics, archaeology and psychology. For example, in archaeology, it can be applied to the *archaeological seriation* problem [GKK74, Tho95]. Archaeologists want to determine a relative time line for the artifacts they unearth. Sometimes they cannot ascertain an exact date for an artifact, but they can determine that a certain artifact came before another artifact. They can draw a graph with a directed edge from i to j if they guess that artifact i came before artifact j. Then they can find an ordering of the artifacts that is compatible with the most guesses in order to determine the most likely time line.

In economics, the linear ordering problem is known as the triangulation problem for input-output matrices. Economists use an input-output matrix to describe an economy. These matrices have the following graphical representation: In a given economy, an economic sector i has an edge with weight w_{ij} to sector j if a w_{ij} fraction of its output is used by sector j. An ordering of the economic sectors that maximizes the weight of the forward edges determines the direction of production in the economy [CW58, KO69].

In addition to its specific applications, the linear ordering problem is also interesting because it belongs to the family of vertex ordering problems. Vertex ordering problems comprise a fundamental class of combinatorial optimization problems that, on the whole, is not well understood. For the past thirty years, combinatorial methods and linear programming techniques have failed to yield improved approximation guarantees for many well-studied vertex ordering problems such as the linear ordering problem and the famous *traveling salesman* problem. Semidefinite programming has proved to be a powerful tool for solving a variety of cut problems, as first exhibited for the maxcut problem [GW95]. Cut problems are problems in which the objective is to partition the vertices into disjoint sets so as to optimize some stated objective function. Since then, semidefinite programming has been successfully applied to many other problems that can be categorized as cut problems such as coloring k-colorable graphs [KMS98], maximum-3-cut [GW04], maximum k-cut [FJ97], maximum bisection and maximum uncut [HZ01], and correlation clustering [CGW03], to name a few. In contrast, there is no such comparably general approach for approximating vertex ordering problems.

Background

The goal of most approaches to an NP-hard maximization problem is to find a good upper bound on the value of an optimal solution. For an input graph $G = (V, A)$, a straightforward upper bound on the size of an optimal solution for the linear ordering problem is the total edge weight. In any ordering, the set of forward edges or the set of backward edges contains half the total edge weight. Thus, the "all edges" bound can be no more than twice as large as optimal. The major open problem is to find a

bound that is strictly less than twice the value of an optimal solution.

Suppose $G = (V, A)$ is a complete directed graph in which every edge has weight 1, $n = |V|$, and $|A| = n(n-1)$. Since the graph contains $\binom{n}{2}$ 2-cycles, the maximum acyclic subgraph of G contains exactly half the edges in A (in a 2-cycle, exactly one edge if forward and one edge is backward in any vertex ordering). For an unweighted input graph $G = (V, A)$ that contains no 2-cycles, Berger and Shor gave an algorithm that always produces an acyclic subgraph of size $(\frac{1}{2} + \Omega(\frac{1}{\sqrt{d_{max}}}))|A|$, where d_{max} denotes that maximum degree of G [BS97]. When G does contain 2-cycles, their algorithm produces an acyclic subgraph $(\frac{1}{2} + \Omega(\frac{1}{\sqrt{d_{max}}}))$ times the number of arcs in an optimal solution. Their algorithm has running time $O(|A||V|)$. Rubinstein and Hassin gave algorithms with the same guarantee but with running time $O(|A| + d_{max}^3)$, which is better than $O(|A||V|)$ in certain cases [HR94]. These bounds are tight in terms of $|A|$ since since the existence of a class of graphs without 2-cycles for which the maximum acyclic subgraph has size at most $(\frac{1}{2} + O(\frac{1}{\sqrt{d_{max}}}))|A|$ follows from a result of Spencer [Spe87] and de la Vega [dlV83]. Thus, an approximation guarantee of $\frac{1}{2}$ is the best constant factor that can be achieved using the "all edges" upper bound.

A typical approach for finding improved upper bound for an NP-hard maximization problem is to compute an optimal solution for a linear programming relaxation of a corresponding integer program. An integer program for an NP-hard maximization problem is a set of constraints whose integer solutions correspond to solutions for the optimization problem. For example, solutions for the following integer program correspond to acyclic subgraphs:

$$\begin{aligned} & \max \sum_{ij \in A} w_{ij} x_{ij} \\ \sum_{ij \in C} x_{ij} & \leq |C| - 1 \quad \forall \text{ cycles } C \in A \\ x_{ij} & \in \{0, 1\}. \end{aligned}$$

In a solution to the above integer program, at least one edge (i, j) in any cycle C has value $x_{ij} = 0$. Thus, if we consider the subset of edges that have value $x_{ij} = 1$, they form an acyclic subgraph. In general, it is NP-hard to solve an integer program. However, if the constraints are linear in the variables and we relax the requirement that x_{ij} are integers and allow fractional solutions, then we can efficiently solve the respective linear programming relaxation via the ellipsoid algorithm [YN76, GLS81].

Recently it was shown that several widely-studied polyhedral relaxations for the linear ordering problem each have an integrality gap of 2, showing that it is unlikely these relaxations can be used to approximate the problem to within a factor greater than $\frac{1}{2}$ [NV01, New00]. The graphs used to demonstrate these integrality gaps are random graphs with uniform edge probability of approximately $\frac{2^{\sqrt{\log n}}}{n}$, where n is the number of vertices. For sufficiently large n, such a random graph has a maximum

acyclic subgraph close to half the edges with high probability. However, each of the polyhedral relaxations studied provide an upper bound for these graphs that is asymptotically close to all the edges, which is off from the optimal by a factor of 2. Thus, in the worst case, the upper bound provided by these polyhedral relaxations is no better than the "all edges" bound. The main question with respect to the linear ordering problem is to find an efficiently computable upper bound that is better than the "all edges" bound for all graphs that have maximum acyclic subgraph close to half the edges. In particular, is there such an efficiently computable bound that that beats the "all edges" bound for random graphs with uniform edge probability, i.e. the graphs used to demonstrate the poor performance of the linear programming relaxations?

Semidefinite programming has proved to be a very useful tool for computing improved upper bounds on a variety of cut problems such as the maxcut problem. A semidefinite program is the problem of optimizing a linear function of a symmetric matrix subject to linear equality constraints and the constraint that the matrix is positive semidefinite. (Inequality constraints can be modeled with equality constraints by using additional variables.) For any $\epsilon > 0$, semidefinite programs can be solved with an additive error of ϵ in polynomial time (ϵ is part of the input, so the running time dependence on ϵ is polynomial in $\log \frac{1}{\epsilon}$) using the ellipsoid algorithm [GLS88]. Semidefinite programming relaxations of quadratic integer programs have been used in efficient algorithms for optimization problems. (A more thorough discussion of semidefinite programming and its applications to optimization problems is given in Chapter 3.)

Semidefinite programming techniques have also been applied to some vertex ordering problems such as the *betweenness* problem [CS98] as well as the *bandwidth* problem [BKRV00]. The input to the betweenness problem is a set of elements $\{x_1, \ldots x_n\}$ and a set of constraints with the following form: x_j should go between x_j and x_k. The goal is to find an ordering of the elements so as to maximize the number of satisfied constraints. Note that a constraint of the stated form is satisfied if the relative order of elements x_i, x_j, x_k in the ordering is $x_i < x_j < x_k$ or $x_k < x_j < x_i$. Chor and Sudan showed how to round a semidefinite programming relaxation to find an ordering satisfying half of the constraints provided the original constraint set is satisfiable. The minimum bandwidth problem was defined in the beginning of this introduction. Blum, Konjevod, Ravi and Vempala gave an $O(\sqrt{\frac{n}{b}} \log n)$-approximation algorithm for an n-node graph with bandwidth b. They gave the first approximation algorithm with an approximation guarantee better than the trivially achievable factor of n and introduced new tools such as spreading metrics that have proven useful in applications to other problems.

Even though both these problems are vertex ordering problems, the semidefinite programming formulations used for these two problems cannot immediately be extended to obtain a formulation for the linear ordering problem. This is because it is not clear how to use these techniques to model objective functions for directed

graphs in which the contribution of edge (i, j) and edge (j, i) to the objective function may differ. In other words, in a solution to the linear ordering problem, an edge (i, j) could be a forward edge and contribute to the objective function, while edge (j, i) is a backward edge and does not contribute to the objective function. Thus, to use semidefinite programming, we need to find a formulation in which $f(i, j)$ is the contribution of edge (i, j) to the forward value and $f(i, j)$ is not equal to $f(j, i)$.

New Results

In this thesis, we present a new semidefinite programming relaxation for the linear ordering problem. A vertex ordering for a graph with n vertices can be fully described by a series of $n - 1$ cuts. We use this simple observation to relate cuts and orderings. This observation also leads to a semidefinite program for the linear ordering problem that is related to the semidefinite program used in the Goemans-Williamson algorithm to approximate the maxcut problem [GW95]. Besides the linear ordering problem, this semidefinite program can be used to obtain formulations for many other vertex ordering problems, since the feasible region over which we are optimizing is a relaxation of a quadratic integer program whose solutions correspond to vertex orderings problems.

We would like to show that this new semidefinite programming relaxation provides an upper bound that is better than the "all edges" bound for all graphs that have a maximum acyclic subgraph close to half the total edge weight. This problem remains open. However, we can show that our relaxation provides an upper bound strictly better than the "all edges" bound for the class of random graphs with uniform edge probability, which with high probability have a maximum acyclic subgraph close to half the edges. This is the first relaxation known to provide a good bound on this large class of graphs. Graphs from this class were used to demonstrate that several widely-studied polyhedral relaxations provide poor upper bounds, i.e. bounds twice as large as an optimal solution, in the worst case [NV01].

Specifically, we show that for sufficiently large n, if we choose a random directed graph on n vertices with uniform edge probability $p = \frac{d}{n}$ (i.e. every edge in the complete directed graph on n vertices is chosen with probability p), where $d = \omega(1)$, our semidefinite relaxation will have an integrality gap of no more than 1.64 with high probability. The main idea is that our semidefinite relaxation provides a "good" bound on the value of an optimal linear ordering for a graph if it has no small roughly balanced bisection. With high probability, a random graph with uniform edge probability contains no such small balanced bisection. These results also appear in [New04].

1.2 Layout of This Thesis

The results in this thesis are based on combinatorial as well as linear and semidefinite programming methods. The chapters in this thesis fall into two categories: methods and applications. Chapters 2 and 3 focus on methods and Chapters 4, 5, and 6 focus on results obtained by applying these methods.

In Chapter 2, we discuss some combinatorial theorems about binary strings that are used in our algorithms for the string folding problem. These combinatorial theorems can be stated independently of the string folding problems and may have other applications. Therefore, they have been placed in their own chapter. In Chapter 3, we discuss linear and semidefinite programming and how these methods can be applied to graph layout problems. These ideas are applied to both linear programs for string folding (Section 4.4) and to semidefinite programs for the linear ordering problem (Chapter 6).

In Chapter 4, we present algorithms for 2D string folding. First, we present Hart and Istrail's factor $\frac{1}{4}$-approximation algorithm for the 2D string folding problem and then we present their factor $\frac{3}{8}$-approximation algorithm for the 3D string folding problem, since it uses the 2D algorithm as a subroutine [HI96]. Next, we present our improved factor $\frac{1}{3}$-approximation algorithm. Our algorithm uses a theorem from Chapter 2—the chapter containing combinatorial methods. We then discuss the quality of the combinatorial upper bound used in the analyses of both our algorithm and that of Hart and Istrail. We present a family of strings such that the number of contacts in an optimal folding of any string from this family is only half of the number of contacts represented in this upper bound. We also use methods from Chapter 3 to obtain and analyze a linear programming relaxation for the string folding problem. The algorithm and the analysis of the combinatorial upper bound have been previously published [New02]. The linear programming results appear as a technical report [CHN03].

In Chapter 5, we discuss the 3D string folding problem. We present a new $\frac{3}{8}$-approximation algorithm and the modifications we can make to this algorithm so as to obtain a slightly improved approximation guarantee. Our algorithms use theorems from Chapter 2. Understanding the proofs of these theorems is not required to understand the applications of the theorems to our algorithms. These results have been previously published [NR04].

Finally, in Chapter 6, we use methods from Chapter 3 to formulate a new semidefinite program for the linear ordering problem. We prove that our relaxation provides a good bound on the optimal value of a linear ordering for random graphs with uniform edge probability. These results have also been previously published [New04].

Chapter 2

Methods I: Combinatorial

In this chapter, we present some combinatorial methods that are used in our algorithms for string layout. In the string folding problem, odd-1's (1's with odd indices) in the string can only have contacts with even-1's (1's with even indices) and vice versa. Therefore, proving properties about the patterns and occurrences of odd-1's and even-1's can be useful when trying to find folding rules that guarantee many contacts, which is the goal of the string folding problem. In this chapter, we will use strings in $\{a, b\}^*$ to represent our binary strings, rather than strings in $\{0, 1\}^*$. We use the latter representation of binary strings to represent input to the string folding problem in Chapters 4 and 5. The theorems in this chapter will be used in the string folding algorithms in Chapters 4 and 5 by mapping subsequences of odd-1's and even-1's to strings of a's and b's, applying the lemmas we prove in this chapter to the strings of a's and b's, and subsequently obtaining lemmas about patterns of odd-1's and even-1's in the input strings to the folding problem. Thus, throughout this chapter, we note that we could prove theorems about strings in $\{0, 1\}^*$. However, since we would not be mapping 0's and 1's in these strings directly to 0's and 1's in the input strings to the string folding problem, we use strings in $\{a, b\}^*$ to avoid confusion.

We define a *loop* to be a binary string in $\{a, b\}^*$ whose endpoints are joined together. Our first theorem shows that given any loop in $\{a, b\}^*$ containing an equal number of a's and b's, we can find a point, i.e. element, in the loop such that if we begin at that point and move in the clockwise direction, we encounter at least as many a's as b's and if we begin at that point and move in the counter-clockwise direction, we encounter at least as many b's as a's. This theorem is a simple combinatorial exercise to prove, but proves to be very useful.

Our second theorem addresses a new combinatorial problem on binary strings. We

call a binary string in $\{a, b\}^*$ *block-monotone* if every maximal sequence of consecutive b's is immediately followed by a sequence of at least as many consecutive a's. Suppose we are given a binary string with the following property: every suffix of the string (i.e. every sequence of consecutive elements that ends with the last element of the string) contains at least as many a's as b's. What is the longest block-monotone subsequence of the string? The subsequence of all the a's is a block-monotone subsequence with length at least half the length of the string. Can we do better? In Section 2.2, we show that there always is a block-monotone subsequence containing at least a $(2 - \sqrt{2}) \approx .5857$ fraction of the string's elements. In contrast, we are aware of strings for which every suffix contains at least as many a's as b's and for which the largest block-monotone subsequence has length equal to a .7115 fraction of the string.

2.1 A Combinatorial Lemma for Strings

Suppose we are given a binary string $S \in \{a, b\}^*$ with an equal number of a's and b's. We join the endpoints of the string S together to form a loop L. We want to determine if there exists an element $s_i \in L$ such that if we move clockwise away from this element, we always encounter at least as many a's as b's, and if we move counter-clockwise away from s_i we always encounter at least as many b's as a's. Lemma 2 gives an affirmative answer to this question.

Definition 1. *Let $n_a(S)$ and $n_b(S)$ denote the number of a's and b's, respectively, in a string $S \in \{a, b\}^*$.*

Lemma 2. *Let $L \in \{a, b\}^*$ be a loop that contains an equal number of a's and b's. There is an element $s_i \in L$ such that if we go around L in one direction (i.e. clockwise or counter-clockwise) starting at s_i to any element $s_j \in L$, then the substring $s_i s_{i+1} \dots s_j$ that we have traversed contains at least as many a's as b's and for any element $s_k \in L$ the substring $s_{i-1} s_{i-2} \dots s_k$ has at least as many b's as a's.*

Proof. Given a loop $L \in \{a, b\}^*$, let $S = s_1 \dots s_n$ be a binary string in $\{a, b\}^*$ that results when the loop L is cut between elements s_1 and s_n to form a string, i.e. joining the endpoints of the string S should result in the loop L. Let $f(j) = n_a(s_1 s_2 \dots s_j) - n_b(s_1 s_2 \dots s_j)$. In other words, $f(j)$ is the number of a's minus the number of b's present in the substring $s_1 s_2 \dots s_j$. Then let j^* be a value of j that minimizes $f(j)$. For example, in Figure 2-1, the function $f(j)$ is shown for the string $ababbaaabb$ and $j^* = 5$ for this string. Note that s_{j^*} must be a b. (If s_{j^*} were an a, then $f(j^* - 1) < f(j^*)$.) Furthermore, s_{j^*+1} must be an a.

Now consider the string $S' = s'_1 \dots s'_n$ such that $s'_1 = s_{j^*+1}$ and $s'_2 = s_{j^*+2}$, etc. The function $f(j)$ for this new string S' is always positive for all j ranging from 1 to n. Thus, $n_a(s'_1 s'_2 \dots s'_j) \geq n_b(s'_1 s'_2 \dots s'_j)$ for any s'_j in the string S'. If we consider the reverse string $s'_n \dots s'_1$, then it is always the case that $n_b(s'_n s'_{n-1} \dots s'_j) \geq n_a(s'_n s'_{n-1} \dots s'_j)$ for any point s'_j. Thus, the theorem is true when $s_i = s'_1$. □

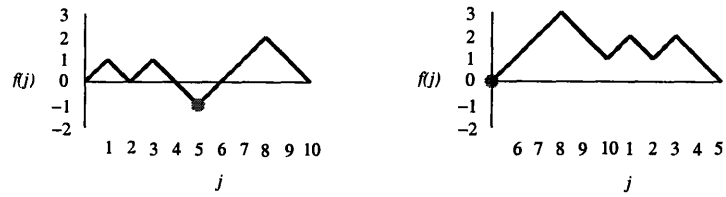

Figure 2-1: The graph of the function $f(j)$ for the string $S = ababbaaabb$ and the string $S' = aaabbbabab$.

2.2 Block-Monotone Subsequences

Consider a binary string in $\{a, b\}^*$. We define a *block* to be a maximal sequence of consecutive a's or a maximal sequence of consecutive b's in the binary string. For example, the string *bbbbaaabb* has two blocks of b's (of length four and two) and one block of a's (of length three). We say a binary string in $\{a, b\}^*$ is *block-monotone* if, as we go from one endpoint to the other—without loss of generality, from left to right—each block of consecutive b's is followed by a block of a's of at least the same length. Some examples of block-monotone strings are:

bbbbbaaaaa,

bababa,

aaaabbbbaaaabbbaaaa.

Some examples of *non* block-monotone strings are:

aaaabbbb,

aaabbbbaa,

bababab.

Given a binary string S in $\{a, b\}^*$, we address the problem of finding a long block-monotone subsequence. If the string S contains only b's, then S does not contain any block-monotone subsequences. Thus, we enforce a stronger condition on the string S: We call a string *suffix-monotone* if every suffix contains at least as many a's as b's. In other words, as we go from right to left, the number of a's always leads the number of b's. For example, the string *ababa* is suffix-monotone as is the string *baabbabbaaa*.

Definition 3. *A binary string* $S = s_1 \dots s_n$, $S \in \{a, b\}^*$ *is* suffix-monotone *if for every suffix* $\overline{S}_k = s_{k+1} \dots s_n$, $0 \leq k < n$, *we have* $n_a(\overline{S}_k) \geq n_b(\overline{S}_k)$.

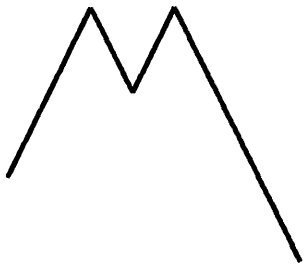

Figure 2-2: A graphical representation of the string $S = bbbbaabbaaaaaa$. We use an "up" edge to denote a 'b' and a "down" edge to denote an 'a'.

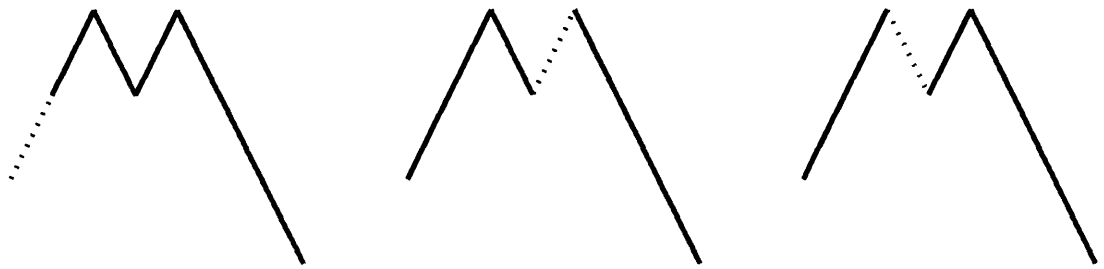

Figure 2-3: Graphical representations of some block-monotone subsequences of the string $S = bbbbaabbaaaaaa$.

Any string $S \in \{a, b\}^*$ contains a block-monotone subsequence of length at least $n_a(S)$ since the subsequence of a's is trivially block-monotone. If the string S is suffix-monotone, then $n_a(S) \geq n_b(S)$, so S contains a block-monotone subsequence with length at least half the length of S. Now we consider the following problem: Suppose S is a suffix-monotone string in $\{a, b\}^*$. Does every such string S contain a block-monotone subsequence of length more than half the length of S?

For example, suppose we have the string (see also Figure 2-2):

$$S = bbbbaabbaaaaaa.$$

The string S has length 14 and the longest block-monotone subsequence of S has length 12. Some block-monotone subsequences of S are (see also Figure 2-3):

$$--bbaabbaaaaaa,$$

$$bbbbaa--aaaaaa,$$

$$bbbb--bbaaaaaa.$$

The problem of finding the longest block-monotone subsequence of a binary string is not NP-hard. The optimal block-monotone subsequence can be found using dy-

namic programming. But we emphasize that although our applications require that we actually find block-monotone subsequences, finding optimal length block-monotone subsequences is not a hard problem. Our main goal in this section is to prove a lower bound on the length of the longest block-monotone subsequence. It seems difficult to analyze the optimal dynamic programming algorithm to show that the longest block-monotone subsequence is a large fraction of the string.

There are strings for which the longest block monotone subsequence is slightly less than .75 times the length of the string. For example, consider the string $(bbba)^3(bba)^6(a)^{12}$. The longest block-monotone subsequence is $(b)^{15}(a)^{16}$, which is only a .74 fraction of the string. The best upper bound we are aware of is a string for which the longest block-monotone subsequence is a .7115 fraction of the string:

$$bbbbbbbbbbbbbabaababababaaabaaabbbabaabaaabaaaabaaaaa.$$

Thus, our goal is to show that the longest block-monotone subsequence of a given binary string is long. By "long", we mean a constant fraction greater than .5 and less than .7115.

2.2.1 Algorithm

In this section, we give an algorithm for finding a block-monotone subsequence of a given suffix-monotone string. This algorithm does not necessarily generate the longest block-monotone subsequence and is therefore not optimal, but we show that the subsequence it does output is long. In particular, if the input string is suffix-monotone and has an equal number of a's and b's, then the algorithm outputs a block-monotone subsequence of length at least a $(2-\sqrt{2}) \approx .5857$ fraction of the input string.

The idea behind our algorithm is to move down the string—from one endpoint to the other—and if we encounter a block of a's, we keep this block and move on. If we encounter a block of b's, we match a subsequence of b's containing that block with a subsequence of a's that follows. Thus, we only keep a subsequence of b's when we can match it with a following subsequence of a's of at least the same length.

We will illustrate the idea of the algorithm with the following example, in which we show how to find a block-monotone subsequence of length $\frac{7}{12}$ the length of the input string S and give a proof sketch. Suppose we have a suffix-monotone string $S \in \{a,b\}^*$. If the first block of the string is: (i) a block of a's, then we add this block of a's to the solution and let S' be the string S with this block of a's removed. If the first block of the string is: (ii) a block of b's, let S_k be the shortest string starting at the current left endpoint such that the ratio of a's to b's is at least 1 to 2. Now we find the prefix of S_k (call it S_ℓ) such that the total number of b's in S_ℓ does not exceed the total number of a's in $S_k \setminus \{S_\ell\}$ and the total number of b's in S_ℓ plus a's in $S_k \setminus \{S_\ell\}$ is maximized. We keep all the b's in S_ℓ and all the a's in $S_k \setminus \{S_\ell\}$ for the

solution and let S' be the string $S \setminus \{S_k\}$. After both steps (i) and (ii), we recurse on the string S'.

Note that the ratio of b's to a's in any proper prefix of S_k is at least $2:1$. In any proper suffix of S_k, the ratio of a's to b's is at least $1:2$. Thus, if the length of S_ℓ is $\frac{|S_k|}{3}$, then there are least $(\frac{2}{3})(\frac{|S_k|}{3})$ b's from S_ℓ in the resulting subsequence and at least $(\frac{1}{3})(\frac{2|S_k|}{3})$ a's from $S_k \setminus \{S_\ell\}$ in the resulting subsequence, which totals $\frac{4}{9}$ of the elements in S_k. Since the ratio of the a's to b's in S_k is 1:2, if $n_a(S) = n_b(S)$, roughly three-fourths of the elements in S belong to some string S_k considered in step (ii) and roughly one-fourth of the elements in S are added to the solution set in step (i). Thus the solution contains at least $(\frac{4}{9})(\frac{3}{4}) + \frac{1}{4} = \frac{7}{12}$ of the elements in the original string. This is the main idea behind our algorithm, but we have glossed over some details. For example, since the length of $|S_k|$ is integral, the ratio of the elements considered in step (ii) may be more than 1:2. Thus, more than three-fourths of the string is considered in step (ii) and less than one-fourth of the string considered in step (i). However, in the analysis of the algorithm, we will show that this idea does lead to an algorithm that outputs a solution with length more than half that of the original string.

The algorithm has the best guarantee on the length of the block-monotone subsequence that it outputs when S_k is the shortest string in which the ratio of b's to a's is at least $\frac{1}{\sqrt{2}} : 1 - \frac{1}{\sqrt{2}}$. This leads to a block-monotone subsequence of length at least $2 - \sqrt{2}$ the length of the input string.

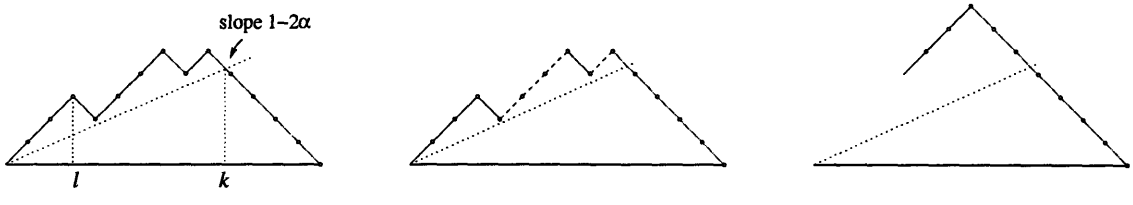

Figure 2-4: These three figures give a pictorial representation of the BLOCK-MONOTONE ALGORITHM. An up edge corresponds to an b and a down edge corresponds to a a. In the first figure, k denotes the point chosen in Step 2 (i) and ℓ denotes the point chosen in Step 2 (iii). In the second figure, the crossed-out edges represent the elements that are removed from the string. The third figure shows the string after removing the crossed-out elements, i.e. the elements that correspond to the block-monotone subsequence.

We will now precisely describe our algorithm and present its analysis to prove our main theorem.

Notation. $\alpha := 1 - \frac{1}{\sqrt{2}} \approx 0.2929$.

Definition 4. *A binary string $S = s_1 \ldots s_n$, $S \in \{a, b\}^*$ is* α-suffix-monotone *if for every suffix $\overline{S}_k = s_{k+1} \ldots s_n$, $0 \leq k < n$, we have $n_a(\overline{S}_k) \geq \alpha \cdot (n - k)$.*

BLOCK-MONOTONE ALGORITHM

Input: An α-suffix-monotone string $S = s_1 \dots s_n$.

Output: A block-monotone subsequence of S.

Let $S_i = s_1 \dots s_i$, $\overline{S}_i = s_{i+1} \dots s_n$ for $i: \ 1 < i \leq n$.

1. If $s_1 = a$:
   (i) Find the largest index k such that S_k is a block of a's and output S_k.
2. If $s_1 = b$:
   (i) Find the smallest index k such that:
   $n_a(S_k) \geq \alpha k$.
   (ii) Let $S'_\ell = s_{\ell+1} \dots s_k$ for $\ell: \ 1 \leq \ell < k$.
   (iii) Find ℓ such that:
   $n_b(S_\ell) \leq n_a(S'_\ell)$,
   $n_b(S_\ell) + n_a(S'_\ell)$ is maximized.
   (iv) Remove all the a's from S_ℓ and output S_ℓ.
   (v) Remove all the b's from S'_ℓ and output S'_ℓ.
3. Repeat algorithm on string $\overline{S}_k$.

Figure 2-5: The BLOCK-MONOTONE ALGORITHM.

This definition is less restrictive than the definition of suffix-monotone (Definition 3). For example, a suffix-monotone string is α-suffix-monotone. We can now state our main theorem.

Theorem 5. *Suppose S is an α-suffix-monotone string of length n. Then there is a block-monotone subsequence of S with length at least $n - n_b(S)(2\sqrt{2}-2)$. Furthermore, such a subsequence can be found in linear time.*

If $n_b(S) \leq \frac{1}{2}n$ and S is suffix-monotone, then Theorem 5 states that we can find a block-monotone subsequence of length at least $(2 - \sqrt{2}) > .5857$ the length of S. This is accomplished by the BLOCK-MONOTONE ALGORITHM, which is based on the ideas described previously.

2.2.2 Analysis

In this section, we prove that, on an α-suffix-monotone input string S, the BLOCK-MONOTONE ALGORITHM outputs a subsequence of length at least $n - n_b(S)(2\sqrt{2}-2)$.

First, we argue correctness, i.e. that the BLOCK-MONOTONE ALGORITHM outputs a subsequence that is block-monotone. In Step 2 (i), there always is an index k with the required property because the definition of *α-suffix-monotone* implies it

is true for $k = n$. Similarly, if $\ell = 1$, then $n_b(S_\ell) = 1 \leq n_a(S'_\ell)$. Thus there is always an ℓ that meets the requirement in Step 2 (iii). Finally, the algorithm outputs a block-monotone subsequence because whenever it outputs a subsequence of b's (in Step 2 (iv)), it also outputs at least as many a's (in Step 2 (v)). This shows that the algorithm is correct.

In this algorithm, we modify the input string by removing a's and b's. However, in order to analyze the algorithm, we will first consider a *continuous* version of the problem in which we can remove a fraction of each a or b. In the continuous version of the problem, we consider each element as a unit-length interval. For example, if $s_i = a$, then s_i is a unit-length segment labeled 'a' and if $s_i = b$, then s_i is a unit-length segment labeled 'b'. Thus, we will view the string S as a string of unit-length a- and b-segments. Suppose $s_1 = b$ and S_k is a prefix of the input string S such that $n_a(S_k) \geq \alpha k$ and $n_a(S_j) < \alpha j$ for all $j : 1 \leq j < k$ as in Step 2 (i) of the algorithm.

Let t denote the (fractional) index in the string at which $n_a(S_t) = \alpha t$. Note that there always exists a point t at which $n_a(S_t) = \alpha t$ because the string S is suffix-monotone, which implies that at least an α fraction of S is a's. The value of t may be a non-integral real number between $k-1$ and k and the string S_t may end with a fractional part of an a.

We define S'_y as the substring starting at position y up to position t. Let y be the (fractional) point in the string S_t such that $n_b(S_y) = n_a(S'_y)$. If we could keep fractional portions of the string, we could keep all the (fractions of) b-intervals in S_y and all the (fractions of) a-intervals in S'_y. At least a $(1-\alpha)$ fraction of the elements in S_y are b's, and at least an α-fraction of the elements in S'_y are a's. So for the fractional problem, the best place to cut the string is at the point $\ell = \beta t$ where:

$$\beta(1-\alpha) = (1-\beta)\alpha \implies \beta = \alpha$$

Thus, we keep a $2\alpha(1-\alpha)$ fraction of each substring considered in Step 2. Next, we are going to compute the total length of the output of our algorithm. Let T_1 represent the set of substrings (i.e. blocks of a's) that are output unmodified during the first step of the algorithm and let $|T_1|$ represent their total length. Let T_2 represent the set of substrings which are modified during the second step of the algorithm and let $|T_2|$ represent their total length. Let m be the length of the output of the algorithm. Then we have the following equations:

$$\begin{aligned} n &= |T_1| + |T_2| \\ n_b(S) &= (1-\alpha)|T_2| \\ m &= |T_1| + 2\alpha(1-\alpha)|T_2| \end{aligned}$$

Solving these three equations, we find that the total fraction of the string that remains

is:

$$m = \left(2\alpha + \frac{1}{\alpha - 1}\right) n_b(S) + n.$$

This expression is maximized for $\alpha = 1 - 1/\sqrt{2}$, which is why we assigned α this value. Substituting, we get:

$$m = n - (2\sqrt{2} - 2)n_b(S). \tag{2.1}$$

Thus, in the case where we can remove fractions of the a's and b's, the algorithm results in a string whose length is indicated in Equation (2.1).

In the integral case, we will show that the algorithm results in a string whose length is at least as large as the fraction in Equation (2.1). By definition, in the algorithm, k equals $\lceil t \rceil$. If the point y in S_t is in an a interval, then ℓ is equals $\lfloor y \rfloor$, since the point ℓ is chosen to as to maximize the quantity $n_b(S_\ell) + n_a(S'_\ell)$. In the algorithm, we keep the whole a-interval that contains t and the whole a interval that contains ℓ. In other words, in addition to keeping the b's in S_y and the a's in S'_y, we are also keeping the fraction of the a-interval that lies in S_y. Note that everything that is added to the solution set in the continuous version is also added to the solution in the algorithm; in addition, the algorithm may add more to the solution set.

If the point y in S_t is in a b-interval, then note that the (fractional) number of b's in S_y is equal to the (fractional) number of a's in S'_y. In the algorithm, the whole a interval in which t lies is included in S_k and therefore in the solution set. Thus, it must be the case that $\ell = \lceil y \rceil$. In the continuous version of the algorithm, the whole a-interval in which t lies is also included in the solution set and only part of the b-interval in which y lies is included. Thus, in the discrete case, we add at least as much to the solution set as we do in the continuous case.

This concludes the proof of Theorem 5. We now prove another simple lemma that shows we can output block-monotone subsequences with a specified number of a's and a specified number of b's. We will find this lemma useful in our folding algorithms for the 3D string folding problem, because we give an application in which we need to know the number of each type of elements in a block-monotone subsequence in advance.

Lemma 6. *We can modify the block-monotone subsequence S' output by the* BLOCK-MONOTONE ALGORITHM *so that:*

$$n_a(S') = \left\lceil \left(1 - \frac{1}{\sqrt{2}}\right) n_b(S) \right\rceil \quad \textit{and} \quad n_b(S') = \left\lceil n - \left(\frac{3}{\sqrt{2}} - 1\right) n_b(S) \right\rceil.$$

Proof. Following the notation of the proof of Theorem 5, in the fractional case, we keep:

$$\alpha(1-\alpha)|T_2| = \alpha n_b(S) \quad b\text{'s},$$

and:

$$|T_1| + \alpha(1-\alpha)|T_2| = n - \frac{1-\alpha+\alpha^2}{1-\alpha} n_b(S) \quad a\text{'s}.$$

Since these are lower bounds on what we keep in the integral case, the subsequence output by the algorithm has at least $(1-\frac{1}{\sqrt{2}})n_b(S)$ a's and $n-(\frac{3}{\sqrt{2}}-1)n_b(S)$ b's. To keep exactly the number of symbols claimed in this Lemma, it suffices to delete the excess number of a's and b's. To do this, first delete the excess b's anywhere in the output string, the result will clearly still be block-monotone. Then we delete the excess a's. Note that at this point, the number of a's exceeds the number of b's, so there will always be a block of a's strictly greater than the preceding block of b's and we can delete a's from this block. □

2.3 Open Problems

Theorem 5 states that α-suffix-monotone strings contain block-monotone subsequences of at least $2-\sqrt{2} \approx .5857$ their length. As previously, mentioned we are aware of α-suffix-monotone strings for which the longest block-monotone subsequence is only a .7115 fraction of the string. The string below is an example of a suffix-monotone string that demonstrates this upper bound:

$$bbbbbbbbbbbbbabaababababaaabaaabbbabaabaaabaaaabaaaaa.$$

The longest block-monotone subsequence of this string is $a^{18}b^{19}$, which is $\frac{37}{52} \approx 71.15\%$ of the length of the original string.

An obvious open question is to close the gap between .5857 and .7115 by improving the upper and/or lower bounds. Additionally, if the string is suffix-monotone—a stronger condition than α-suffix-monotone—then perhaps we can find block-monotone subsequences of length much more than a .5857 fraction of the string. We conjecture that suffix-monotone strings contain block-monotone subsequences at least $\frac{2}{3}$ their length.

Chapter 3

Methods II: Linear and Semidefinite Programming

In this chapter, we discuss linear and semidefinite programming approaches that can be applied to graph layout problems. Linear and semidefinite programs for combinatorial problems are obtained by relaxing integer programs. An integer program for a combinatorial problem is a set of constraints whose integral solutions correspond to solutions for the optimization problem. In general, it is NP-hard to solve an integer program, but the constraints that the variables are integral can be relaxed to form linear or semidefinite programming relaxations, which can be used to efficiently compute bounds on the values of optimal solutions.

For many graph optimization problems, integer programs that correspond to finding a maximum/minimum weight subgraph with a particular property have been studied. Linear programming relaxations of these integer programs approximate the convex hull of integer solutions in $\{0,1\}^{|E|}$. Many well-studied linear programming relaxations for the maxcut problem are of this type [Bar83, BGM85, BM86, PT95]. (The maxcut problem was defined in Chapter 1.) For example, one of these relaxations, introduced by Barahona, Grötschel and Mahjoub, is based on the integer program below, which requires that an integral solution contain no odd cycles, i.e. it is bipartite [BGM85].

$$\begin{aligned} \max \quad & \sum_{e \in E} w_e x_e & (3.1) \\ \sum_{e \in C} x_e \quad & \leq \quad |C| - 1 \quad \forall \textit{ odd cycles } C \subseteq E \\ x_e \quad & \in \quad \{0,1\} \quad \forall\, e \in E. \end{aligned}$$

As we will see in this chapter, there are other integer programs for the maxcut problem, some of which lead to stronger bounds on the value of an optimal solution. For example, there are formulations in which variables are used to indicate which vertices should be placed on which side of a cut rather than which edges should be included in a subgraph. Our goal is to find new integer programs for graph layout problems that lead to efficiently solvable relaxations.

3.1 Linear Programming

A *linear program* is the problem of optimizing a linear function subject to linear inequality constraints. The vector c has length n, the matrix A has m rows—one for each constraint—each with n entries, and the vector b has length m. The goal is to find a solution for the vector $\mathbf{x} = \{x_1, x_2, \ldots, x_n\}$ that satisfies the following:

$$\begin{aligned} &\max \ c^T \mathbf{x} \\ &A^T \mathbf{x} \leq b. \end{aligned}$$

A linear program with a polynomial number of constraints can be solved in polynomial time using the Ellipsoid algorithm, developed by Yudin and Nemirovskii [YN76] and proved to have efficient running time by Khachiyan [Kha79], or using interior point methods [Kar84]. More generally, a linear program with a polynomial-time separation oracle can be solved in polynomial time even if it has an exponential number of constraints [GLS81, GLS88]. A new, simple algorithm for this more general problem due to Bertsimas and Vempala uses random walks [BV04].

Given a solution to a linear program, a polynomial-time separation oracle is an efficient algorithm that determines if the solution is feasible, i.e. does not violate any constraints, or is infeasible. If the solution is infeasible, the separation oracle also specifies a violated constraint. For example, consider the linear programming relaxation of the aforementioned integer program for the maxcut problem:

$$\begin{aligned} & \max \sum_{e \in E} w_e x_e & & (3.2) \\ \sum_{e \in C} x_e & \leq |C| - 1 & \forall \text{ odd cycles } C \subseteq E & \\ 0 \leq & x_e \leq 1 & \forall\, e \in E. & \end{aligned}$$

Even though there could potentially be an exponential number of odd cycles and thus an exponential number of constraints, there is a well-known efficient separation oracle. In other words, given a solution to the above linear program, $\{x_e\}$, there is a polynomial-time algorithm to determine if the total value of the edge variables for any odd-cycle C is at most $|C|-1$. Finding such an algorithm is a common homework problem!

Linear programming relaxations are often used to compute upper bounds on the values of optimal solutions for maximization problems (or lower bounds on the values of optimal solutions for minimization problems). Suppose we have a combinatorial problem (e.g. the maxcut problem or the maximum acyclic subgraph problem) and a corresponding integer program for that problem. We can obtain a linear programming relaxation of this integer program by relaxing the requirement that the variables are integral. For example, from the aforementioned integer program for the maxcut problem, Equation (3.1), we obtain a linear programming relaxation by replacing the integrality constraint $x_e \in \{0, 1\}$ with the constraint $0 \leq x_e \leq 1$, Equation (3.2).

We can solve this linear programming relaxation efficiently using one of the efficient algorithms for linear programs referred to above. For a maximization problem, the optimal value of the linear programming relaxation is an upper bound on the optimal value of the integer program. Thus, a general approach to finding an upper bound for a maximization problem is to (i) find an integer program that describes the problem, (ii) relax the integrality constraints to obtain a linear programming relaxation for the problem, (iii) solve the linear program to efficiently compute a bound on an optimal integral solution.

3.1.1 Assignment Constraints

A graph layout problem can be cast as an *assignment* problem. In an assignment problem, the goal is to assign each vertex a position, a set or a label so as to optimize a particular objective function. For example, in the maxcut problem the goal is to *assign* each vertex of a given graph to S or $\overline{S}$ so as to maximize the weight of the edges with endpoints in both sets. In the linear ordering problem, the goal is to *assign* each vertex to a unique set labeled $\{1, 2, \ldots, n\}$ so as to maximize the weight of the directed edges (i, j) such that vertex i has a smaller label than vertex j.

Linear programs for assignment problems can be formulated with—what are often

referred to as—*assignment constraints.* Suppose we are given a graph $G = (V, E)$ and a set of positions P, and we want to assign each vertex to a position. We can formulate a linear program in which we have a variable x_{ip} for each vertex $i \in V$ and each position $p \in P$. If the vertex i is assigned to position p, then variable $x_{ip} = 1$, otherwise $x_{ip} = 0$. If each variable in the set $\{x_{ip}\}$ is a positive integer, then constraint (3.3) enforces the requirement that each vertex is assigned to some position.

$$\sum_{p \in P} x_{ip} = 1 \quad \forall i \in V. \tag{3.3}$$

If we want to enforce the condition that each position has at most one vertex assigned to it, as is the case in a vertex ordering problem, we can use the following constraint:

$$\sum_{i \in V} x_{ip} = 1 \quad \forall p \in P. \tag{3.4}$$

We can use these constraints to formulate another integer program for the maxcut problem, which is different the one given earlier (3.1). Our goal is to place each vertex in one of two sets. For each vertex $i \in V$, we have two variables, x_{i1} and x_{i2}. If i is placed in S, then in an integral solution, we require that $x_{i1} = 1$ and $x_{i2} = 0$. Alternatively, if i is placed in $\overline{S}$, then we require that $x_{i1} = 0$ and $x_{i2} = 1$. For each edge, $ij \in E$, we have two variables f_{ij} and b_{ij}. We will require that the variable $f_{ij} = 1$ if i is in S and j is in $\overline{S}$ and the variable $b_{ij} = 1$ if i is in $\overline{S}$ and j is in S. We can enforce these requirements with the following integer program:

$$\begin{aligned}
& & \max & \sum_{ij \in E} w_{ij}(f_{ij} + b_{ij}) & & (3.5) \\
x_{i1} + x_{i2} & = & 1 & \quad \forall i \in V & & \\
f_{ij} & \leq & \min\{x_{i1}, x_{j2}\} & \quad \forall ij \in E & & \\
b_{ij} & \leq & \min\{x_{j1}, x_{i2}\} & \quad \forall ij \in E & & \\
x_{i1}, x_{i2} & \in & \{0, 1\} & \quad \forall i \in V & & \\
f_{ij}, b_{ij} & \in & \{0, 1\} & \quad \forall ij \in E. & &
\end{aligned}$$

The constraint,

$$f_{ij} \leq \min\{x_{i1}, x_{j2}\} \quad \forall ij \in E,$$

can be enforced with the following two linear constraints:

$$f_{ij} \leq x_{i1} \quad \forall ij \in E,$$
$$f_{ij} \leq x_{j2} \quad \forall ij \in E.$$

If we replace the integrality restriction with the relaxed constraint $0 \leq x_{i1}, x_{i2} \leq 1$, $0 \leq f_{ij}, b_{ij} \leq 1$, we obtain another linear programming relaxation for the maxcut problem. How good is the bound provided by this linear program? This linear program actually does not provide a very good bound. This is because we can let $x_{i1} = x_{i2} = \frac{1}{2}$ for every $i \in V$. Then every edge contributes 1 to the objective value. Thus, it simply says that the optimal value of any maxcut is $|E|$, which is a trivial bound. Furthermore, with high probability, a random graphs with uniform edge probability has a maxcut arbitrarily close to half the edges [Pol92]. So this linear program can have an optimal value of twice the optimal integral value.

3.1.2 Graph Layout

Assignment constraints can be used to design linear programs for graph layout problems. For example, we can generalize the integer program in the previous section, (3.5), to obtain an integer program for the linear ordering problem. Given a directed graph $G = (V, A)$, our goal is to assign each vertex in the graph a label from $\{1, 2, \ldots, n\}$ so as to maximize the weight of the directed edges (i, j) such that i has a smaller label than j. For each vertex i and each label h, we have a variable $x_{ih} = 1$ if vertex i is labeled h.

$$\begin{aligned}
& & \max \quad & \sum_{ij \in A} w_{ij} \Big(\sum_{h<\ell} y_{ij}^{h\ell} \Big) \\
\sum_{h=1}^{n} x_{ih} & = & 1 \quad & \forall i \in V \\
\sum_{i=1}^{n} x_{ih} & = & 1 \quad & \forall h \in \{1, 2, \ldots, n\} \\
y_{ij}^{h\ell} & \leq & \min\{x_{ih}, x_{j\ell}\} \quad & \forall ij \in E,\ \forall i, h \in \{1, 2, \ldots, n\} \\
x_{ih} & \in & \{0, 1\} \quad & \forall i \in V,\ \forall h \in \{1, 2, \ldots, n\} \\
y_{ij}^{h\ell} & \in & \{0, 1\} \quad & \forall ij \in E,\ \forall h, \ell \in \{1, 2, \ldots, n\}.
\end{aligned}$$

We can obtain a linear programming relaxation by relaxing the constraint that the variables $\{x_{ih}\}, \{y_{ij}^{h\ell}\}$ are integral and instead require them to have values between 0 and 1. If we let each variable $x_{ih} = \frac{1}{n}$, then each edge contributes $\binom{n}{2} \cdot \frac{1}{n} = \frac{n-1}{2}$ to the objective function. Therefore, this linear program provides a very bad bound on the optimal value of a linear ordering.

However, these basic ideas of using assignment constraints can still be useful in graph layout problems. We can use this general framework to formulate linear programs for problems involving laying out graphs on square lattices, for example. In Section 4.4, we show how to use these ideas to formulate an integer program and a corresponding linear programming relaxation for the string folding problem. We will have a variable x_{iv} for each vertex i and each lattice point v. For every edge in the lattice, i.e. pair of adjacent lattice points, we have a variable $h_{(v,w)}$ that indicates whether there is 0 or 1 contact across edge (v, w) in the lattice. In other words, we will specify constraints to enforce that for each edge in the lattice, (v, w), the variable $h_{(v,w)}$ is 1 if and only if there are vertices from the input graph each labeled 1 and occupying lattice point v and w. Additionally, the semidefinite programs for the linear ordering problem that we discuss in the next section and in Chapter 6 are based on assignment constraints.

3.2 Semidefinite Programming

A semidefinite program is the problem of optimizing a linear function of a symmetric matrix subject to linear equality constraints and the constraint that the matrix is positive semidefinite. (We indicate that a matrix Y is positive semidefinite by: $Y \succeq 0$.) Inequality constraints can also be included in a semidefinite program, because they can be modeled with equality constraints using additional variables. For any $\epsilon > 0$, semidefinite programs can be solved with an additive error of ϵ in polynomial time (ϵ is part of the input, so the running time dependence on ϵ is polynomial in $\log \frac{1}{\epsilon}$) using the ellipsoid algorithm [GLS88]. Other methods can also be used to solve semidefinite programs efficiently such as interior point methods [Ali95]. Although not provably efficient, the simplex method can also be used to solve semidefinite programs [Pat96].

3.2.1 Cut Problems

Semidefinite programming yields another way to develop efficiently computable relaxations for combinatorial optimization problems. A useful feature of semidefinite programming is the structure of the semidefinite solution matrix. An $n \times n$ positive semidefinite matrix Y can be decomposed into $Y = XX^T$ where X is an $n \times n$ matrix. This decomposition lies at the heart of the Goemans-Williamson .87856-approximation algorithm for the maxcut problem [GW95].

Consider the following integer quadratic program for the maxcut problem. For a given a graph $G = (V, E)$, each vertex $i \in V$ has a corresponding vector v_i that is required to be in $\{1, -1\}$; the assignment $v_i = 1$ means that vertex i is on one side of the cut and the assignment $v_i = -1$ means that vertex i is on the other side of the cut. Each edge contributes value $\frac{w_{ij}}{2}(1 - v_i \cdot v_j)$ to the objective function: if $v_i = v_j$,

then this contribution is 0; if $v_i \neq v_j$, then this contribution is w_{ij}.

$$\begin{aligned} \max \quad & \frac{1}{2}\sum_{i<j} w_{ij}(1 - v_i v_j) \\ v_i \quad \in \quad & \{-1, 1\} \quad \forall i \in V. \end{aligned} \tag{3.6}$$

Goemans and Williamson showed that the semidefinite relaxation of this integer program can be used in an approximation algorithm for the maxcut problem [GW95]. They used the following semidefinite relaxation of the above integer program (3.6):

$$\begin{aligned} \max \quad & \frac{1}{2}\sum_{i<j} w_{ij}(1 - y_{ij}) \\ y_{ii} \quad = \quad & 1 \\ Y \quad \succeq \quad & 0. \end{aligned} \tag{3.7}$$

In the Goemans-Williamson algorithm for the maxcut problem, the matrix Y is decomposed into $Y = XX^T$. Each of the n rows of the matrix X is a unit vector. These vectors, $\{x_1, x_2, \ldots, x_n\}$, have the property that $x_i \cdot x_j = y_{ij}$. Each vertex i corresponds to a unit vector x_i in this decomposition. Thus, the relaxation (3.7) is equivalent to the relaxation (3.8):

$$\begin{aligned} \max \quad & \frac{1}{2}\sum_{i<j} w_{ij}(1 - x_i \cdot x_j) \\ x_i \cdot x_i \quad = \quad & 1 \\ x_i \quad \in \quad & \mathcal{R}^n \quad \forall i \in V. \end{aligned} \tag{3.8}$$

To obtain a feasible solution for the maxcut problem, a vector $r \in \mathcal{R}^n$ is chosen at random. A vertex i is assigned to one side of the cut if $v_i \cdot r < 0$ and to the other side if $v_i \cdot r \geq 0$. In expectation, the total edge weight crossing the cut is at least .87856 of the objective value of the semidefinite relaxation, which is at least .87856 of the value of an optimal maxcut. This relaxation is the only known relaxation that provably provides a bound of less than "all edges" for all graphs with a maxcut close to half the total edge weight.

Closely related to the maxcut problem is the maximum directed cut (dicut) problem. Given a directed weighted graph $G = (V, A)$, the dicut problem is to find a bipartition of the vertices—call these disjoint sets S_1 and S_2—that maximizes the weight of the edges directed from S_1 to S_2, i.e. the weight of the directed edges (i, j) such that vertex i is in set S_1 and vertex j is in set S_2. Goemans and Williamson [GW95] and Feige and Goemans [FG95] study semidefinite relaxations of integer programs for this problem. One such integer program is based on assignment constraints. In

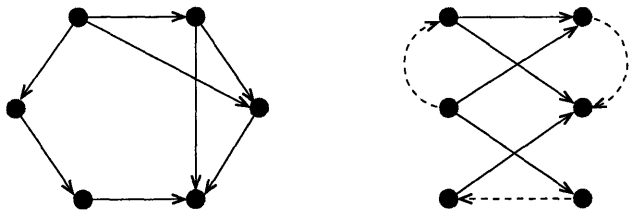

Figure 3-1: A maximum dicut for the graph on the left is shown on the right.

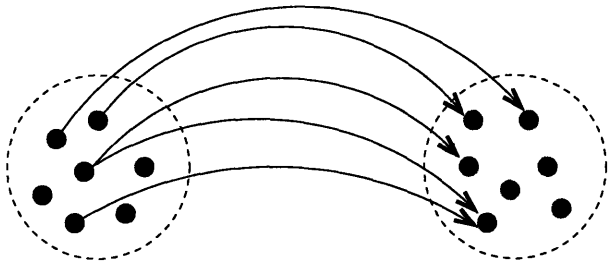

Figure 3-2: The dicut problem is to divide the vertices of a directed graph into sets S_1 and S_2 so as to maximize the weight of edges directed from S_1 to S_2.

the following integer program, each vertex i has two corresponding vectors, t_i and f_i. The vector v_0 is an arbitrary unit vector; without loss of generality, we can assume $v_0 = \{1, 0, \ldots, 0\}$. In an integral solution, if vertex i is assigned to S_1, then t_i is assigned value v_0 and f_i is assigned 0; if vertex i is assigned to S_2, then $t_i = 0$ and $f_i = v_0$. Thus, the following constraints enforce the requirements that in an integral solution, exactly one of t_i and f_i is set to v_0 for each vertex i. For a directed edge (i, j), if i is in S_1 and j is in S_2, then $t_i = f_j = v_0$, so the contribution of that edge to the objective value is w_{ij}.

$$\begin{aligned} & \max \sum_{ij \in A} w_{ij}(t_i \cdot f_j) \\ t_i \cdot f_i &= 0 \quad \forall i \in V \\ v_0 \cdot t_i + v_0 \cdot f_i &= 1 \quad \forall i \in V \\ v_0 \cdot v_0 &= 1 \\ t_i, f_i &\in \{0, v_0\} \quad \forall i \in V. \end{aligned}$$

The integrality constraint in the above integer program can be relaxed to form the following semidefinite relaxation for the dicut problem. Instead of being assigned to 0 or v_0, each t_i and f_i will be assigned a vector in $\mathcal{R}^n$. This is equivalent to the constraint that the matrix of t_i, f_i values is positive semidefinite. Note that the

following constraints imply that $f_i = v_0 - t_i$. Thus, the dimension of the solution matrix is $n + 1$.

$$\begin{aligned} & \max \sum_{ij \in A} w_{ij}(t_i \cdot f_j) \\ t_i \cdot f_i &= 0 \quad \forall i \in V \\ v_0 \cdot t_i + v_0 \cdot f_i &= 1 \quad \forall i \in V \\ v_0 \cdot v_0 &= 1 \\ t_i, f_i &\in \mathcal{R}^{n+1} \quad \forall i \in V. \end{aligned} \tag{3.9}$$

Goemans and Williamson also gave a new approximation algorithm for the dicut problem that is very similar to their algorithm for the maxcut problem [GW95]. It is based on rounding a semidefinite relaxation of the following integer program. In this integer program, a vector $v_i = v_0$ if vertex i belongs to set S_1 and $v_i = -v_0$ if vertex i belongs to set S_2. Thus, if edge (i, j) is directed from S_1 to S_2, then the contribution to the objective value is w_{ij} and if vertex i is not in S_1 or if vertex j is not in S_2, then the contribution to the objective value is 0.

$$\begin{aligned} & \max \frac{1}{4} \sum_{ij \in A} w_{ij}(1 + v_0 \cdot v_i - v_0 \cdot v_j - v_i \cdot v_j) \\ v_i \cdot v_i &= 1 \quad \forall i \in V \cup \{0\} \\ v_i &\in \{v_0, -v_0\} \quad \forall i \in V \cup \{0\}. \end{aligned} \tag{3.10}$$

We obtain an efficiently solvable relaxation by relaxing the constraint that v_i is either v_0 or $-v_0$ and requiring only that v_i is a unit vector in $\mathcal{R}^{n+1}$.

$$\begin{aligned} & \max \frac{1}{4} \sum_{ij \in A} w_{ij}(1 + v_0 \cdot v_i - v_0 \cdot v_j - v_i \cdot v_j) \\ v_i \cdot v_i &= 1 \quad \forall i \in V \cup \{0\} \\ v_i &\in \mathcal{R}^{n+1} \quad \forall i \in V \cup \{0\}. \end{aligned} \tag{3.11}$$

After solving the above semidefinite relaxation to obtain a set of solution vectors, $\{v_i\}$, a random vector $r \in \mathcal{R}^{n+1}$ is chosen. If $r \cdot v_0 < 0$, then each vertex i such that $v_i \cdot r < 0$ is placed in S_1 and the rest of the vertices are placed in S_2. If $r \cdot v_0 > 0$, then each vertex i such that $v_i \cdot r > 0$ is placed in S_1 and the rest of the vertices are placed in S_2. Goemans and Williamson showed that this algorithm has an approximation guarantee of at least .79607 [GW95].

The relaxations (3.9) and (3.11) are equivalent. Consider a solution to the above

relaxation (3.9). Let $v_i = t_i - f_i$. Then the vectors $\{v_i\}$ satisfy the constraint in (3.11) that they are unit vectors. Similarly, consider a solution $\{v_i\}$ for the relaxation (3.11). Let $t_i = \frac{v_0+v_i}{2}$ and $f_i = \frac{v_0-v_i}{2}$. Then $t_i \cdot f_i = 0$ and $v_0 \cdot (t_i + f_i) = 1$, satisfying the constraints in the relaxation (3.9). The objective function in the relaxation (3.9) can be rewritten as:

$$t_i \cdot f_i = \left(\frac{v_0 + v_i}{2}\right) \cdot \left(\frac{v_0 - v_j}{2}\right) = \frac{1}{4}(1 + v_i \cdot v_0 - v_j \cdot v_0 - v_i \cdot v_j).$$

Feige and Goemans note that constraints can be added to strengthen the relaxation (3.9). In particular, we can require that: $t_i \cdot f_j \geq 0, t_i \cdot t_j \geq 0, f_i \cdot f_j \geq 0$. Transforming these constraints to the form in the relaxation (3.11), we obtain the following constraints for all $i, j \in V$:

$$\begin{aligned} v_0 \cdot v_i + v_0 \cdot v_j + v_i \cdot v_j &\geq -1 \\ -v_0 \cdot v_i - v_0 \cdot v_j - v_i \cdot v_j &\geq -1 \\ -v_0 \cdot v_i + v_0 \cdot v_j - v_i \cdot v_j &\geq -1. \end{aligned}$$

Additionally, Feige and Goemans note that we can obtain an even stronger relaxation by allowing any vector v_k to take the role of v_0 in the above constraints:

$$\begin{aligned} v_i \cdot v_j + v_i \cdot v_k + v_j \cdot v_k &\geq -1 \\ -v_i \cdot v_j - v_i \cdot v_k + v_j \cdot v_k &\geq -1 \\ -v_i \cdot v_j + v_i \cdot v_k - v_j \cdot v_k &\geq -1 \\ v_i \cdot v_j + v_i \cdot v_k - v_j \cdot v_k &\geq -1. \end{aligned} \tag{3.12}$$

These constraints are valid because they hold for any set of variables $\{v_i, v_j, v_k\} \in \{1, -1\}$. Note that these constraints can also be used to strengthen the maxcut relaxation (3.6). Although these constraints strengthen the relaxations (3.11) and (3.6), it is an open problem how to use these constraints to improve the approximation guarantee for the maxcut or dicut problems. However, Halperin and Zwick showed how to use these so-called triangle inequalities to strengthen related relaxations and improve the approximation guarantees for several cut problems such as the maximum-$\frac{n}{2}$-bisection problem and the maximum-$\frac{n}{2}$-directed-bisection problem [HZ01]. The maximum-$\frac{n}{2}$-bisection problem is that of finding a maximum cut with the further constraint that the two sets S and $\overline{S}$ have the same cardinality (the cardinality of the two sets differs by 1 vertex if n is odd). Similarly, the maximum-$\frac{n}{2}$-directed-bisection problem is that of finding a maximum directed cut with the extra constraint that the two sets S_1 and S_2 have the same cardinality.

3.2.2 Vertex Ordering Problems

The edges in a directed cut form an acyclic subgraph. We can generalize the dicut problem to that of dividing the vertices into k labeled sets $S_1, S_2, \ldots, S_k$ so as to maximize the weight of the edges (i, j) such that vertex i is in set S_h and vertex j is in set S_ℓ and $h < \ell$. We call this the k-acyclic dicut problem. The linear ordering problem is equivalent to the n-acyclic dicut problem.

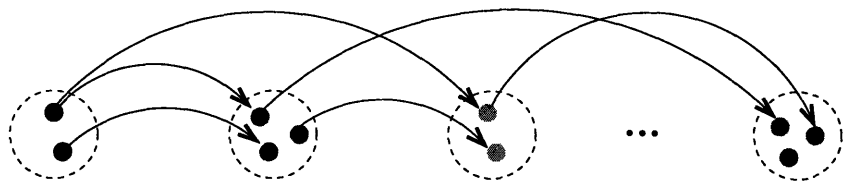

Figure 3-3: We define the *k-acyclic dicut* problem to be that of dividing the vertices of a directed graph into k sets labeled $S_1, S_2, S_3, \ldots, S_k$ so as to maximize the weight of edges directed from S_i to S_j, $i < j$.

Thus, the semidefinite programs for the dicut problem can also be generalized to formulate semidefinite programs for vertex ordering problems. In this section, we will discuss how to obtain such a formulation for vertex ordering problems. For example, in the semidefinite relaxation for the dicut problem (3.9), we use two variables t_i and f_i to represent if vertex i is assigned to S_1 or S_2. For the k-acyclic dicut problem, we can use k vectors for each vertex—as opposed to two vectors—to indicate which of the k positions vertex i occupies. Similarly, for the linear ordering problem, we will use n vectors to indicate which position vertex i occupies. We will formulate an integer program for the linear ordering problem in which v_0 is an arbitrary unit vector and vector $u_{ih} = v_0$ if vertex i is in position h in the ordering and $u_{ih} = 0$ if vertex i is not in position h. Thus, our integer program will have $n^2 + 1$ vectors. The following constraints (3.13) are valid for an integer program in which the feasible solutions are all permutations of the vertices of a given graph. Let N represent the set $\{1, 2, \ldots, n\}$.

$$\begin{aligned} v_0 \cdot v_0 &= 1 && (3.13)\\ \sum_{h=1}^{n} u_{ih} \cdot v_0 &= 1 \qquad \forall i \in V \\ \sum_{i=1}^{n} u_{ih} \cdot v_0 &= 1 \qquad \forall h \in N \\ u_{ih} &\in \{0, v_0\} \qquad \forall i \in V,\ \forall h \in N. \end{aligned}$$

Lemma 7. *There is a one-to-one correspondence between permutations of the vertices*

and feasible solutions for the set of constraints (3.13).

Proof. Consider a permutation of the vertices in which each vertex i has a unique label h in the set N. Let $u_{ih} = v_0$, where v_0 is a unit vector, and let $u_{i\ell} = 0$ for $\ell \neq h$. We will show that this solution satisfies the constraints (3.13). The first constraint $v_0 \cdot v_0 = 1$ is satisfied since v_0 is a unit vector. The second constraint,

$$\sum_{h=1}^{n} u_{ih} \cdot v_0 = 1 \quad \forall i \in V,$$

is satisfied since for each i, there is only one value of h such that $u_{ih} = v_0$. The next constraint,

$$\sum_{i=1}^{n} u_{ih} \cdot v_0 = 1 \quad \forall h \in N,$$

is satisfied since for each position h, there is only one vertex i such that $u_{ih} = v_0$. Thus, for every permutation, there is a unique feasible solution.

Now we argue that each feasible solution for the constraints (3.13) corresponds to a permutation of the vertices. Consider a feasible solution to the constraints. By the first constraint, for each vertex i, there is exactly one value of h such that $u_{ih} = v_0$, so each vertex is assigned a position. By the second constraint, for each position h, there is exactly one vertex i such that $u_{ih} = v_0$, i.e. vertex i is assigned to position h. Therefore, a feasible solution corresponds to a permutation of the vertices. □

Since a feasible solution for the constraints (3.13) corresponds to a permutation of the vertices, we can measure the weight of the forward edges with respect to a particular solution, i.e. vertex permutation, using the following objective function. Let $G = (V, A)$ be a given directed graph and let $\{u_{ih}\}, v_0$ be a feasible solution to the constraints (3.13).

$$\max \sum_{ij \in A} w_{ij} \Big(\sum_{h<\ell} u_{ih} \cdot u_{j\ell} \Big). \tag{3.14}$$

Consider an edge $(i, j) \in A$. If (i, j) is a forward edge in the vertex permutation corresponding to the given feasible solution, then there are some vectors $u_{ih}, u_{j\ell}$ such that $u_{ih} = u_{j\ell}$ and $h < \ell$. Conversely, if i comes before j in the ordering, then there is some $h < \ell$ such that $u_{ih} = u_{j\ell}$.

Unfortunately, optimizing the objective function (3.14) over the constraints (3.13) is NP-hard, since this would yield an optimal solution to the linear ordering problem, which is NP-hard. So we relax the constraint that each u_{ih} is either v_0 or 0 to the constraint that $u_{ih} \in \mathcal{R}^{n^2+1}$. We can also add constraints that are valid for integer solutions and may strengthen our semidefinite relaxation. For example, the constraint $u_{ih} \cdot u_{ih} = u_{ih} \cdot v_0$ is valid in an integral solution since u_{ih} is either 0 or equal to v_0.

Additionally, in an integral solution, it is the case that $u_{ih} \cdot u_{i\ell} = 0$, since for each vertex i, the variable u_{ih} is non-zero for exactly one value of h. Similarly, $u_{ih} \cdot u_{jh} = 0$ is a valid constraint because for each position h, the u_{ih} is non-zero for exactly one vertex i. Finally, we can also require that every pair of vectors from the set $\{u_{ih}\}$ has a non-negative dot product, since this constraint holds for an integral solution. Combining all these constraints results in the following semidefinite programming relaxation.

$$\begin{aligned}
& \max \sum_{ij \in A} w_{ij} \Big(\sum_{h<\ell} u_{ih} \cdot u_{j\ell} \Big) && (3.15) \\
v_0 \cdot v_0 &= 1 \\
\sum_{h=1}^{n} u_{ih} \cdot v_0 &= 1 \quad \forall i \in V \\
\sum_{i=1}^{n} u_{ih} \cdot v_0 &= 1 \quad \forall h \in N \\
u_{ih} \cdot u_{ih} &= u_{ih} \cdot v_0 \quad \forall i \in V,\ h \in N \\
u_{ih} \cdot u_{i\ell} &= 0 \quad \forall i \in V,\ h, \ell \in N \\
u_{ih} \cdot u_{jh} &= 0 \quad \forall i, j \in V,\ h \in N \\
u_{ih} \cdot u_{j\ell} &\geq 0 \quad i, j \in V,\ h, \ell \in N \\
u_{ih} &\in \mathcal{R}^{n^2+1} \quad \forall i \in V,\ \forall h \in N.
\end{aligned}$$

Lemma 8. *For a given graph $G = (V, A)$, the optimal value of the semidefinite program in constraints* (3.15) *is $\sum_{ij \in A} w_{ij}$.*

Proof. We will show that the total contribution to the unweighted objective function for any 2-cycle is one edge. A 2-cycle constraint can be written as follows:

$$\sum_{1 \leq h < \ell \leq n} u_{ih} \cdot u_{j\ell} + \sum_{1 \leq h < \ell \leq n} u_{jh} \cdot u_{i\ell} = 1.$$

Since $u_{ih} \cdot u_{jh} = 0$, we have:

$$\sum_{1 \leq h < \ell \leq n} u_{ih} \cdot u_{j\ell} + \sum_{1 \leq h < \ell \leq n} u_{jh} \cdot u_{i\ell} = \Big(\sum_{k=1}^{n} u_{ik} \Big) \cdot \Big(\sum_{k=1}^{n} u_{jk} \Big) = v_0 \cdot v_0 = 1.$$

Thus, we can bound the forward value for any edge by 1:

$$\sum_{1 \leq h < \ell \leq n} u_{ih} \cdot u_{j\ell} \leq 1 \quad \Rightarrow \quad \sum_{ij \in A} \sum_{1 \leq h < \ell \leq n} w_{ij} (u_{ih} \cdot u_{j\ell}) \leq \sum_{ij \in A} w_{ij}.$$

□

This semidefinite program is related to the semidefinite program for the dicut

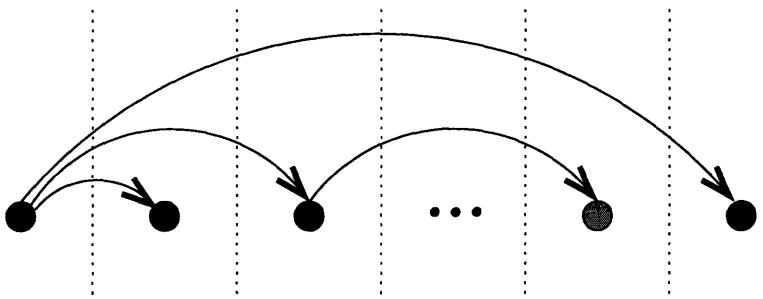

Figure 3-4: A vertex ordering can be precisely described by $n-1$ bipartitions of the vertices.

problem (3.9). A vertex ordering can be completely described by a a series of $n-1$ cuts. For example, the first cut has one vertex on one side and $n-1$ vertices on the other; the second cut has two vertices on one side and $n-2$ on the other; the k^{th} cut has k vertices on one side and $n-k$ on the other. In fact, if we consider a solution to the constraints (3.15), $\{\{u_{ih}\}, v_0\}$, we can fix a value $h \in N$ and let

$$t_i = \sum_{i=1}^{h} u_{ih}, \quad f_i = \sum_{i=h+1}^{n} u_{ih}.$$

This yields a feasible solution for the dicut semidefinite program (3.9). Moreover, suppose t_i and f_i are computed using an integral feasible solution for the constraints (3.15) (i.e. $u_{ih} \in \{0, v_0\}$). This integral solution corresponds to a vertex permutation. Then the resulting objective value for the dicut semidefinite program is exactly the weight of the edges crossing the cut (in the forward directed) that divides the first h vertices from the last $n-h$ vertices in the vertex permutation that corresponds to the integral solution.

This semidefinite program (3.15) is based on representing an ordering using 0-1 vectors (without loss of generality, we can assume that $v_0 = (1, 0, \ldots, 0)$). For example, a vertex ordering of a graph on four vertices in which vertex i is in position i in the ordering has the following representation:

$$\begin{aligned}
\{u_{11},\ u_{12},\ u_{13},\ u_{14}\} &= \{1,\ 0,\ 0,\ 0\}, \\
\{u_{21},\ u_{22},\ u_{23},\ u_{24}\} &= \{0,\ 1,\ 0,\ 0\}, \\
\{u_{31},\ u_{32},\ u_{33},\ u_{34}\} &= \{0,\ 0,\ 1,\ 0\}, \\
\{u_{41},\ u_{42},\ u_{43},\ u_{44}\} &= \{0,\ 0,\ 0,\ 1\}.
\end{aligned}$$

There are other ways to represent vertex orderings. We now discuss another way that uses $\{1, -1\}$ vectors (or $\{v_0, -v_0\}$ vectors, where v_0 is an arbitrary unit vector).

This representation is also based on the observation that a vertex ordering can be fully described by a series of $n-1$ cuts. This semidefinite program can be viewed as a generalization of the second semidefinite program discussed for the dicut problem (3.11). In this representation, each vertex i will have $n+1$ associated unit vectors, $\{v_i^0, v_i^1, v_i^2, \ldots v_i^n\}$. In an integral solution, we enforce that $v_i^0 = -1$, $v_i^n = 1$ and that v_i^h and v_i^{h+1} differ for only one value of h, $0 \leq h < n$. This position h denotes vertex i's position in the ordering. For example, suppose we have a graph G that has four vertices, arbitrarily labeled 1 through 4. Consider the vertex ordering in which vertex i is in position i. An integral description of this vertex ordering is:

$$\begin{aligned}
\{v_1^0,\ v_1^1,\ v_1^2,\ v_1^3,\ v_1^4\} &= \{-1,\ \ 1,\ \ 1,\ \ 1,\ \ 1\}, \\
\{v_2^0,\ v_2^1,\ v_2^2,\ v_2^3,\ v_2^4\} &= \{-1, -1,\ \ 1,\ \ 1,\ \ 1\}, \\
\{v_3^0,\ v_3^1,\ v_3^2,\ v_3^3,\ v_3^4\} &= \{-1, -1, -1,\ \ 1,\ \ 1\}, \\
\{v_4^0,\ v_4^1,\ v_4^2,\ v_4^3,\ v_4^4\} &= \{-1, -1, -1, -1,\ \ 1\}.
\end{aligned}$$

There is actually a connection between this representation (i.e. using $\{-1, 1\}$ variables and the previously discussed representation (i.e. using $\{0, 1\}$ variables). In integral solutions, we have the following relation between the u_{ih} and v_i^h variables:

$$v_i^h = \sum_{k=1}^{h} u_{ik} - \sum_{k=h+1}^{n} u_{ik}.$$

Note that we can assume $u_{i0} = 0$. Then $v_i^0 = -\sum_{k=1}^{n} u_{ik}$. Similarly, we have:

$$u_{ih} = \frac{v_i^h - v_i^{h-1}}{2}.$$

Thus, using this connection between the two representations, we can obtain constraints for the linear ordering problem in terms of the $\{v_i^h\}$ variables. Below, we translate each of the constraints (3.15) line by line. Note that an edge (i, j) only contributes 1 to the objective function when: $v_i^{h-1} = v_j^{\ell-1} = -1$ and $v_i^h = v_j^\ell = 1$.

Valid:	.	.	.	h-1	h	.	.	.	ℓ-1	ℓ	.	.	.
	.	.	.	-1	1	.	.	.	1	1	.	.	.
	.	.	.	-1	-1	.	.	.	-1	1	.	.	.
Invalid:	.	.	.	h-1	h	.	.	.	ℓ-1	ℓ	.	.	.
	.	.	.	-1	-1	.	.	.	1	-1	.	.	.
	.	.	.	-1	1	.	.	.	1	1	.	.	.

Figure 3-5: The invalid assignment violates the constraint $(v_i^h - v_i^{h-1}) \cdot (v_j^\ell - v_j^{\ell-1}) \geq 0$ since the lefthand side of this expression evaluates to -4 for these vectors.

$$\begin{aligned}
& \max \sum_{ij \in A} \sum_{h<\ell} \frac{1}{4} w_{ij} (v_i^h - v_i^{h-1}) \cdot (v_j^\ell - v_j^{\ell-1}) && (3.16) \\
v_0 \cdot v_0 &= 1 \\
(v_i^n - v_i^0) \cdot v_0 &= 2 \quad \forall i \in V \\
\sum_{i=1}^{n} (v_i^h - v_i^{h-1}) \cdot v_0 &= 2 \quad \forall h \in N \\
v_i^h \cdot v_i^{h-1} - v_i^h \cdot v_0 - v_i^{h-1} \cdot v_0 &= 1 \quad \forall i \in V,\ h \in N \\
(v_i^h - v_i^{h-1}) \cdot (v_i^\ell - v_i^{\ell-1}) &= 0 \quad \forall i \in V,\ h, \ell \in N \\
(v_i^h - v_i^{h-1}) \cdot (v_j^h - v_j^{h-1}) &= 0 \quad \forall i, j \in V,\ h \in N \\
(v_i^h - v_i^{h-1}) \cdot (v_j^\ell - v_j^{\ell-1}) &\geq 0 \quad i, j \in V,\ h, \ell \in N \\
v_i^h &\in \mathcal{R}^{n^2+1} \quad \forall i \in V,\ \forall h \in N.
\end{aligned}$$

Other constraints we can add are triangle inequalities which are shown (3.12). We can apply these triangle inequalities on any set of three vectors chosen from the set $\{v_i^h\}$. We can also add the following constraint, which states that the number of vertices on each side of the middle cut is $\frac{n}{2}$ for even n:

$$\sum_{i,j \in V} v_i^{\frac{n}{2}} \cdot v_j^{\frac{n}{2}} = 0.$$

In Chapter 6, we will actually focus on the following semidefinite program, which includes a subset of these constraints. These constraints are sufficient to prove that the integrality gap is small for random graphs.

$$\max \quad \frac{1}{4}\sum_{ij\in A}\sum_{1\le h<\ell\le n} w_{ij}(v_i^h - v_h^{h-1})\cdot(v_j^\ell - v_j^{\ell-1})$$

$$(v_i^h - v_i^{h-1})\cdot(v_j^\ell - v_j^{\ell-1}) \geq 0 \quad \forall i,j \in V,\ h,\ell \in [n] \tag{3.17}$$

$$v_i^h \cdot v_i^h = 1 \quad \forall i \in V,\ h \in [n]$$

$$v_i^0 \cdot v_0 = -1 \quad \forall i \in V$$

$$v_i^n \cdot v_0 = 1 \quad \forall i \in V$$

$$\sum_{i,j\in V} v_i^{\frac{n}{2}} \cdot v_j^{\frac{n}{2}} = 0 \tag{3.18}$$

$$v_i^h \in \{1,-1\} \qquad \forall i,h \in [n]. \tag{3.19}$$

3.3 Discussion

Besides the linear ordering problem, we can also model other vertex ordering problems using the semidefinite programs presented in Section 3.2.2. For example, given an undirected graph $G = (V,E)$, the goal of the minimum bandwidth problem is to assign each vertex $i \in V$ a unique label $\ell(i)$ from the set of integers $N = \{1,2,\ldots,n\}$ so as to minimize the quantity: $\max_{ij\in E} |\ell(i) - \ell(j)|$. We can use the constraints in the semidefinite program (3.15) (without the objective function) and add the additional constraint:

$$u_{ih}\cdot u_{j\ell} = 0 \quad \forall ij \in E,\ \ell,h \in N,\ |\ell - h| > b.$$

We can run a semidefinite program solver for each integral value of b from 1 through n and find the minimum value of b such that the semidefinite program is feasible. This value of b is a lower bound on the value of the minimum bandwidth. We note that we can easily extend this idea to obtain a relaxation for the minimum directed bandwidth problem as well. The input to the minimum directed bandwidth problem is an acyclic graph and the goal is to find a topological sort of the vertices that minimizes the maximum length edge. If the graph $G = (V,A)$ is acyclic, then in an integral solution, we can require that every edge is a forward edge using the following constraint for each edge $(i,j) \in A$:

$$\sum_{h\ge\ell} u_{ih}\cdot u_{j\ell} = 0. \tag{3.20}$$

Another problem we can model is the minimum linear arrangement problem. For a given undirected graph $G = (V,E)$, the goal of this problem is to assign each vertex $i \in V$ a unique label $\ell(i)$ from the set of integers $N = \{1,2,\ldots,n\}$ so as to minimize

the quantity: $\sum_{ij} |\ell(i) - \ell(j)|$. Again, we can use the constraints from the semidefinite program (3.15) substituting the following for the objective function:

$$\min \sum_{ij \in E} \sum_{h,\ell \in N} |\ell - h| u_{ij} \cdot u_{j\ell}.$$

Finally, another well-known vertex ordering problem that we can model with our semidefinite program is the *traveling salesman* problem. Given a complete, weighted undirected graph with weights obeying the triangle inequality, the goal is to find an ordering of the vertices such that the total weight of edges connecting consecutive vertices in the ordering is minimized. Let $u_{i,n+1} = u_{i1}$. Then following objective function corresponds to the traveling salesman problem:

$$\min \sum_{i,j \in V} \sum_{h=1}^{n} w_{ij} (u_{ih} \cdot u_{jh+1}).$$

Chapter 4

2D String Folding

4.1 Introduction

In this chapter, we focus on an optimization problem known as the *string folding* problem. The string folding problem is motivated by the protein folding problem, which is a central problem in computational biology. It is considered to be one of the simplest models of the protein folding problem and leads to a purely combinatorial problem.

In this combinatorial problem, the input is a string: a path graph in which each vertex except the two endpoints has degree exactly two and each end vertex has degree one. Each vertex in the string is labeled 1 or 0. Throughout this chapter, we will refer to such a path graph as a string S in $\{0,1\}^*$.

Figure 4-1: The input to the string folding problem can be viewed as a path graph.

Additionally, a particular type of lattice is specified as part of the input. In this chapter, we use a *square lattice*. A two-dimensional square lattice (three-dimensional square lattice) is a graph drawn in the (x,y)-plane ((x,y,z)-plane) in which the vertices are all points with integral coordinates. The edges connect pairs of vertices that are at distance one. Figure 4-2 illustrates a square lattice as well as a triangular lattice, which is another possible type of input lattice.

We say a vertex from the string is *placed* on a lattice point (x,y) if that vertex is assigned to lattice point (x,y). A *folding* of such an input string corresponds to

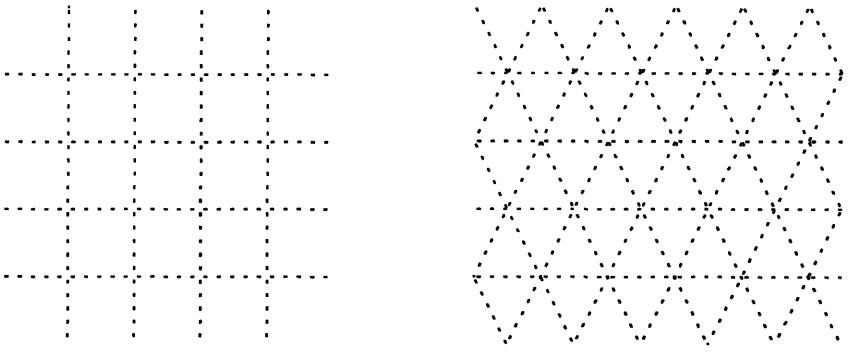

Figure 4-2: A square lattice and a triangular lattice.

placing the vertices of the graph on a lattice subject to the following three constraints:

(i) Each lattice point can have at most one vertex placed on it.

(ii) Each vertex must be placed on *some* lattice point.

(iii) Adjacent vertices in the string must be placed on adjacent lattice points.

For example, suppose vertex i and $i+1$ are adjacent in the input graph. On a two-dimensional square lattice, if vertex i is placed on lattice point (x, y), then vertex $i+1$ must be placed on one of four possible lattice points: $(x \pm 1, y)$ or $(x, y \pm 1)$. In a valid folding of a string, the string is laid out on the lattice so that it does not cross itself. Such a configuration on the square lattice folding is commonly referred to as a *self-avoiding walk.*

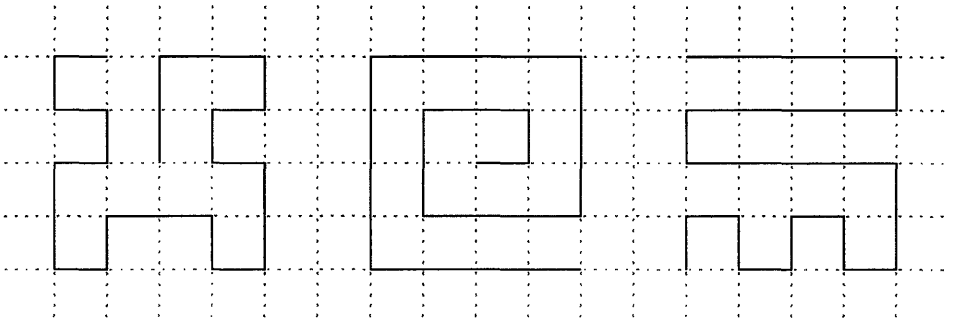

Figure 4-3: Some examples of self-avoiding walks.

There are many possible valid foldings for an input string. We are interested in finding certain types of foldings. With respect to a particular folding of an input string, we say a pair of vertices forms a *contact* if they are not adjacent on the string,

they are both labeled 1, and they are placed on neighboring lattice points. The goal of the string folding problem is to find a folding of the input string that maximizes the number of contacts.

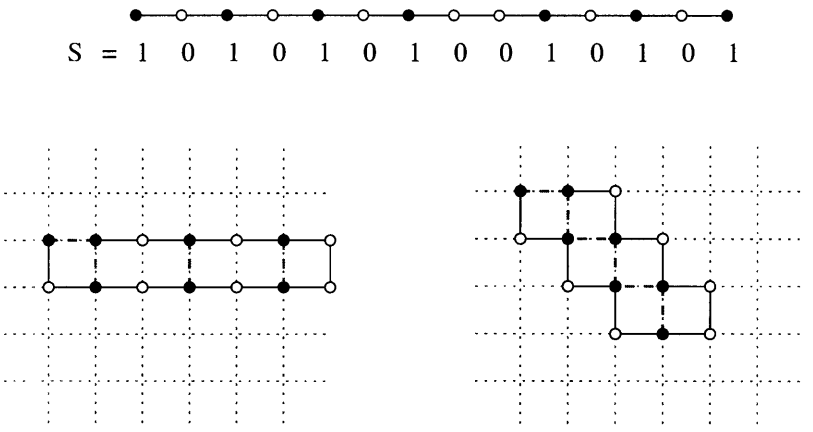

Figure 4-4: Two possible foldings for the string $S = 10101010010101$. The first folding has four contacts and the second folding has six contacts. Contacts are indicated by the dashed lines.

For example, suppose the input graph is the string 10101010010101. The first folding shown in Figure 4-4 results in four contacts. The second folding shown in Figure 4-4 results in six contacts, which is optimal. Throughout this chapter, vertices labeled 1 are denoted by black dots and vertices labeled 0 are denoted by white or unfilled dots.

4.1.1 Motivation

This string folding problem is motivated by the *protein folding* problem, a widely studied problem in the field of computational biology. A protein is a sequence of amino acids. Each sequence folds to a unique shape. The three-dimensional shape of a protein determines its function. The protein folding problem is to determine the three-dimensional shape of a protein given its amino acid sequence.

A simplified model of protein folding known as the Hydrophobic-Hydrophilic (HP) model was introduced by Dill [Dil85, Dil90]. This model abstracts the dominant force in protein folding: the propensity of hydrophobic amino acids to cluster together to avoid water. In the HP model, each amino acid residue is classified as an H (hydrophobic or non-polar) or a P (hydrophilic or polar). An optimal conformation for a string of amino acids in this model is one that has the lowest energy, which is achieved when the maximum number of H-H contacts (i.e. pairs of H's that are adjacent in the folding but not in the sequence) are present. This model is further simplified by restricting the foldings to the two-dimensional (2D) or three-dimensional

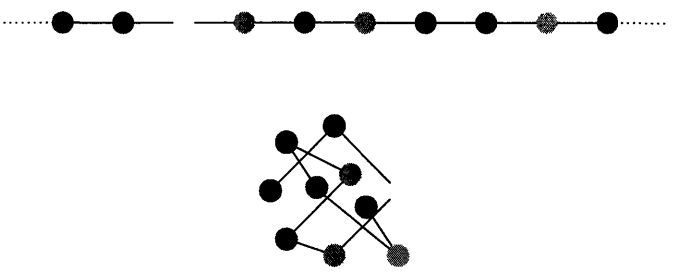

Figure 4-5: A protein is composed of a one-dimensional amino acid sequence and folds to a three-dimensional shape that determines its function.

(3D) square lattice. The protein folding problem in the hydrophobic-hydrophilic (HP) model on the 2D square lattice is combinatorially equivalent to the string folding problem described previously. We are given a string of P's and H's (instead of 0's and 1's) and our goal is to find a folding on the lattice that maximizes the number of adjacent pairs of H's (instead of 1's). Hart and Istrail give an informative discussion of the HP model and its applicability to protein folding [HI96]. In this chapter, we focus on the string folding problem on the 2D square lattice, which we refer to as the 2D string folding problem. In Chapter 5, we focus on the string folding problem on the 3D square lattice, which we refer to as the 3D string folding problem.

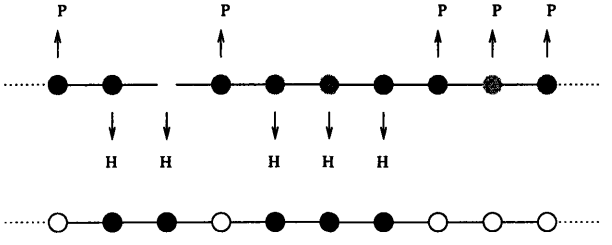

Figure 4-6: Each amino acid is classified as either an H or a P depending on its degree of hydrophobicity.

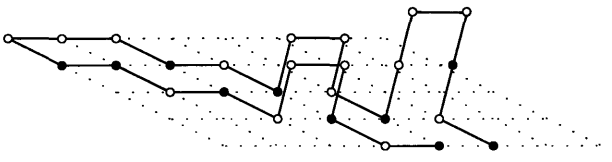

Figure 4-7: The three-dimensional HP model.

4.1.2 Previous Work

The HP lattice model of protein folding is one of the simplest models of protein folding in terms of the problem description. Nevertheless, the problem is computationally difficult to solve. In 1995, Hart and Istrail introduced the string folding problem to the theoretical computer science community and presented approximation algorithms for the string folding problem [HI96]. At this time, neither the 2D nor the 3D problem was known to be NP-hard and settling this question was a major open problem. They gave linear-time algorithms that, for a given input string, output foldings with at least $\frac{1}{4}$ and $\frac{3}{8}$ of the optimal number of contacts for the problem on the 2D and 3D square lattice, respectively. In 1998, the 2D string folding problem was shown to be NP-hard by Crescenzi, Goldman, Papadimitriou, Piccolboni and Yannakakis [CGP+98] and the 3D string folding problem was shown to be NP-hard by Berger and Leighton [BL98].

Additionally, Agarwala et al. gave approximation algorithms for the string folding problem on the 2D and 3D triangular lattice with approximation guarantees of slightly better than $\frac{1}{2}$ [ABD+97]. It is not known if the string folding problem on the 2D or 3D triangular lattice is NP-hard. More recently, Mauri, Piccolboni, and Pavesi gave another factor $\frac{1}{4}$-approximation algorithm for the 2D problem based on dynamic programming in 1999 [MPP99]. They claimed that their algorithm performed better than Hart and Istrail's algorithm in practice. Improving the approximation guarantees of $\frac{1}{4}$ and $\frac{3}{8}$ for the string folding problem on the 2D and 3D square lattices have been open problems in computational biology for several years.

4.1.3 Organization

In this chapter, we show that the the approximation guarantee for the 2D folding problem can be improved from $\frac{1}{4}$ to $\frac{1}{3}$. In Section 4.2, we discuss the combinatorial upper bound used by Hart and Istrail and describe their linear-time approximation algorithms. In Section 4.3, we present an improved linear-time $\frac{1}{3}$-approximation for the 2D string folding problem. We prove that our algorithm outputs a folding with at least $\frac{1}{3}$ as many contacts as prescribed by the simple combinatorial upper bound described in Section 4.2. In Section 4.4, we discuss a linear programming relaxation for the string folding problem and show that the bound provided by the linear program is no more than three times the optimal number of contacts. In Section 4.5, we show that both the combinatorial upper bound and our linear programming upper bounds cannot be used to obtain an algorithm with an approximation factor better than $\frac{1}{2}$. In particular, we describe a string for which the optimal folding achieves only half of the combinatorial upper bound and only half of the linear programming upper bound.

4.2 A Combinatorial Bound

Hart and Istrail gave linear-time algorithms that, for a given input string, output foldings with at least $\frac{1}{4}$ and $\frac{3}{8}$ of the optimal number of contacts for the 2D and 3D problems, respectively. For the 2D problem, they used a simple combinatorial upper bound on the optimal number of contacts possible in any folding. The 2D square lattice is a bipartite graph and a string is a bipartite graph. Therefore, when placed on the lattice, each vertex with an even index in the string can only be adjacent to a vertex with an odd index in the string and vice versa. We refer to vertices with even (odd) indices labeled 1 as *even-1's* (*odd-1's*).

Let S be a specified input string for the folding problem. Let $\mathcal{E}[S]$ denote the number of even-1's in S and let $\mathcal{O}[S]$ denote the number of odd-1's in S. Even-1's can only have contacts with odd-1's and vice versa. In any folding of an input string, each vertex (except for the two endpoints) has two vertices that are adjacent to itself on the string and on the lattice. Since each lattice point has four neighbors, each vertex can have at most two contacts. Let $M_2[S]$ denote the maximum number of contacts possible for a string S. The maximum possible number of contacts in any folding of S is:

$$M_2[S] \leq 2 \cdot \min\{\mathcal{E}[S], \mathcal{O}[S]\} + 2. \tag{4.1}$$

Hart and Istrail used this upper bound to give a $\frac{1}{4}$-approximation for the 2D problem. In other words, they gave an algorithm that outputs a folding with at least $\min\{\mathcal{E}[S], \mathcal{O}[S]\}/2$ contacts. Their algorithm is quite simple. First, choose a vertex p in the input string S such that at least half the odd-1's are on one side of p and at least half the even-1's are on the other side of p. It is easy to find such a vertex: find a vertex in S such that at least half the even-1's are on one side and at least half the even-1's are on the other side. One of these sides contains at least half of the odd-1's—let this side be the *odd side* and the other side be the *even side*. Then, place all odd-1's from the odd side in a straight line and place all the even-1's from the even side in an adjacent straight line. See Figure 4-9 for an illustration. Without loss of generality, assume $\mathcal{O}[S] \leq \mathcal{E}[S]$. Then this folding results in at least $\mathcal{O}[S]/2$ contacts. The maximum possible number of contacts is $2 \cdot \mathcal{O}[S]$. Thus, this algorithm has an approximation guarantee of $\frac{1}{4}$.

Hart and Istrail also gave a $\frac{3}{8}$-approximation for the 3D string folding problem. We discuss the 3D string folding problem in the next chapter, but we describe their 3D algorithm now since it uses the 2D algorithm that we just described as a subroutine. Let $M_3[S]$ denote the maximum number of contacts for a string S when folding on the 3D square lattice. For the 3D problem, the upper bound on the maximum number of contacts is:

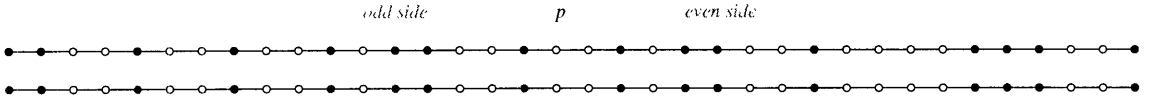

Figure 4-8: . Vertex p is chosen such that at least half the odd-1's are on one side of p and at least half the even-1's are on the other side of p.

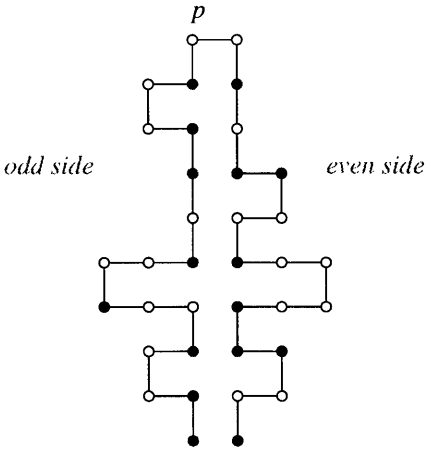

Figure 4-9: An illustration of a folding output by Hart and Istrail's $\frac{1}{4}$-approximation algorithm for the 2D string folding problem.

$$M_3[S] \leq 4 \cdot \min\{\mathcal{O}[S], \mathcal{E}[S]\} + 4. \tag{4.2}$$

Let $k = \mathcal{O}[S]/2$. Then the odd side has at least k odd-1's and the even side has at least k even-1's. The next step is to divide the odd side into segments with $\sqrt{k}$ odd-1's and divide the even side into segments with $\sqrt{k}$ even-1's. The 2D folding algorithm is then repeated $\sqrt{k}$ times in adjacent (x, y)-planes. The idea is that each of the odd-1's on the odd side has three contacts: one in the (x, y)-plane, one with the plane above and one with the plane below. Without loss of generality, assume $\mathcal{O}[S] \leq \mathcal{E}[S]$. Then, in particular, three contacts are made for $\mathcal{O}[S]/2 - c\sqrt{\mathcal{O}[S]}$ odd-1's for some constant c. This results in an algorithm with an absolute approximation guarantee of $(3/8 - O(1/\sqrt{\mathcal{O}[S]}))$ and an asymptotic approximation guarantee of $\frac{3}{8}$. See Figure 4-10 for an illustration of this algorithm.

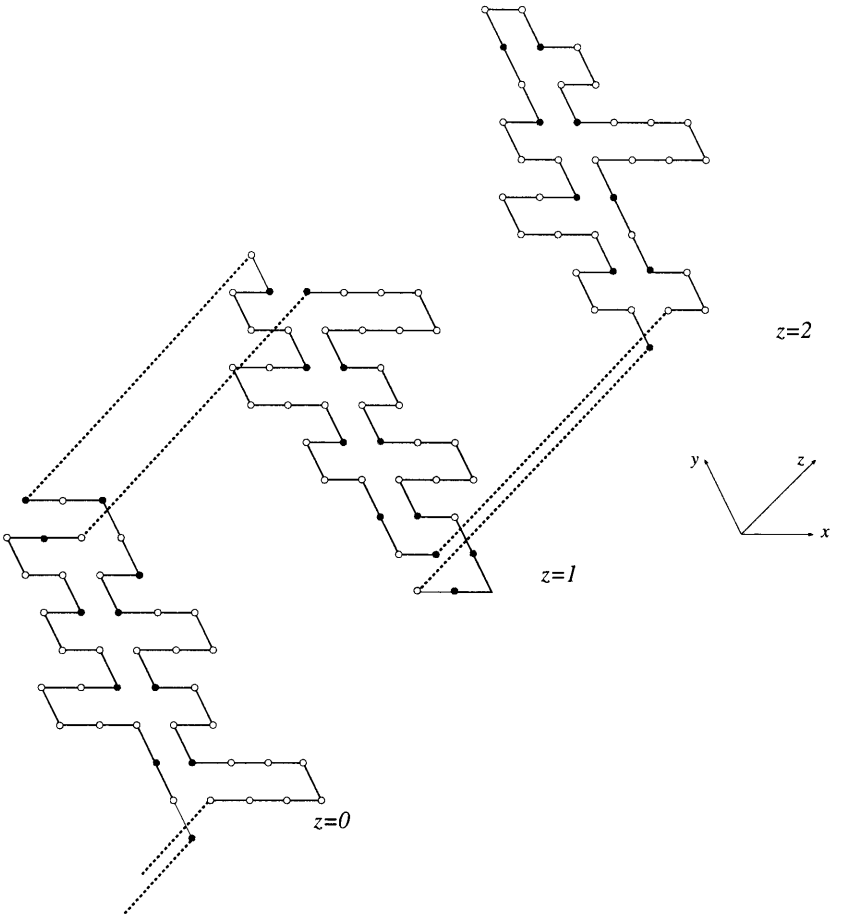

Figure 4-10: An illustration of a folding output by Hart and Istrail's $\frac{1}{4}$-approximation algorithm for the 3D string folding problem.

4.3 A Factor $\frac{1}{3}$-Approximation Algorithm

We now present a factor $\frac{1}{3}$-approximation algorithm for the string folding problem on the 2D square lattice. In Section 4.3.1, we state the algorithm itself, and in Section 4.3.2 we analyze the approximation guarantee and the running time. The approximation guarantee of $\frac{1}{3}$ for our algorithm is obtained by showing that at least $\frac{2}{3}$ of the odd-1's or at least $\frac{2}{3}$ of the even-1's average at least one contact each. Without loss of generality, we make the following assumptions about any input string S to the string folding problem.

(i) The length of S is even.

(ii) The number of odd-1's is equal to the number of even-1's, i.e. $\mathcal{O}[S] = \mathcal{E}[S]$.

If the length of the string S is odd, we can pad the string with an extra vertex labeled 0. If the string S does not have an equal number of odd-1's and even-1's, say

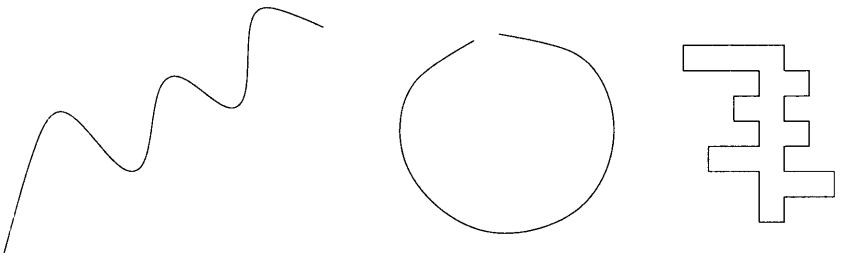

Figure 4-11: Our algorithm folds a loop rather than a string.

$\mathcal{O}[S] < \mathcal{E}[S]$, we can turn an arbitrarily chosen subset of $\mathcal{E}[S] - \mathcal{O}[S]$ even-1's into vertices labeled 0. Both of these modifications leave the quantity $\min\{\mathcal{O}[S], \mathcal{E}[S]\}$—and therefore the value of the upper bound (Equation (4.1))—unchanged.

For the sake of convenience, we consider folding a *loop* rather than a string. That is, given a string $S \in \{0,1\}^*$ (which has even length by assumption (i) above), we add an edge between the first and last vertices to obtain the loop $L(S)$. Note that the upper bound stated in Section 4.2 is also a valid upper bound for the number of contacts that can be obtained by folding a loop. Since the loop is closed, we need to demarcate which vertices are have odd indices and which vertices have even indices. It suffices to choose any vertex, label it 'odd' and call every vertex an even distance away from this vertex 'odd' and call the rest 'even'.

Lemma 9. *Let S be a string such that if we join the endpoints, we obtain the loop $L(S)$. Then a folding of the loop $L(S)$ resulting in k contacts also yields folding of the string S with at least k contacts.*

Proof. Consider any folding of $L(S)$ with k contacts. Any string that is obtained by disconnecting two adjacent vertices of $L(S)$ can assume the same configuration as this folding. So this configuration also yields at least k contacts for such a string. □

One of the combinatorial observations discussed in Section 2.1 plays a key role in our algorithm. We apply Lemma 2 from Section 2.1. Lemma 2 states that if we have a loop $L \in \{a,b\}^*$ with an equal number of a's and b's, then there is some element in the loop such that if we go in the clockwise direction, we encounter at least as many a's as b's and if we go in the counter-clockwise direction, we encounter at least as many b's as a's.

Consider the loop $L(S)$ in which each odd-1 is replaced by an a and each even-1 is replaced by a b and each 0 is ignored. By Lemma 2, there is a vertex s_i in $L(S)$ with the following properties: if we start at vertex s_i and move in the clockwise direction, we will encounter at least as many odd-1's as even-1's (i.e. at least as many a's as b's), and if we start at vertex s_{i-1} and move in the counter-clockwise direction, we will encounter at least as many even-1's as odd-1's (i.e. at least as many b's as a's). We will refer to vertex s_i as vertex p.

Consider the j^{th} odd-1 encountered if we start at vertex $p+1$ and go along $L(S)$ in the clockwise direction, and define $B_{\mathcal{O}}(j)$ to be the substring from the vertex directly following the $(j-1)^{st}$ odd-1 up to and including the j^{th} odd-1. Consider the i^{th} even-1 encountered if we start at vertex $p-2$ and move along $L(S)$ in the counter-clockwise direction, and define $B_{\mathcal{E}}(i)$ to be the substring from the vertex directly following the $i-1^{th}$ even-1 up to and including the i^{th} even-1. Let the length of $B_{\mathcal{O}}(j)$ be $\ell_{\mathcal{O}}(j)+1$ and the length of $B_{\mathcal{E}}(i)$ be $\ell_{\mathcal{E}}(i)+1$. Note that $\ell_{\mathcal{E}}(i)$ and $\ell_{\mathcal{O}}(j)$ are always odd integers. For example, given the loop corresponding to the string $S = 11010110100011$, where $p = s_7$, we have that $B_{\mathcal{O}}(1) = 01$, $B_{\mathcal{O}}(2) = 0001$, $B_{\mathcal{O}}(3) = 11$. We also have that $B_{\mathcal{E}}(1) = 01, B_{\mathcal{E}}(2) = 01$, and $B_{\mathcal{E}}(3) = 11$. See Figure 4-12 for an illustration.

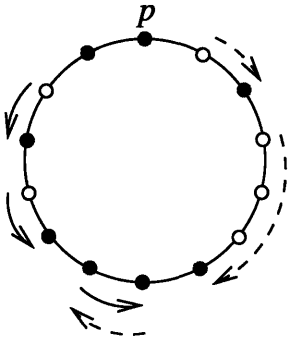

Figure 4-12: Moving clockwise from p, we have the substrings $B_{\mathcal{O}}(1)$, $B_{\mathcal{O}}(2)$, etc. Moving counter-clockwise from p, we have the substrings $B_{\mathcal{E}}(1)$, $B_{\mathcal{E}}(2)$, etc.

4.3.1 Algorithm

We now describe our STRING FOLDING ALGORITHM. Our goal is to find a folding of a given string $S \in \{0,1\}^*$ so as to maximize the number of pairs of adjacent 1's. Using Lemma 9, we consider folding the loop $L(S)$. Using Lemma 2, we find a vertex p, such that if we go around the loop $L(S)$ in the clockwise direction from p, we always encounter at least as many odd-1's as even-1's and if go around $L(S)$ in the counter-clockwise direction, we always encounter at least as many even-1's as odd-1's.

STRING FOLDING ALGORITHM(S).

Input: A loop $L(S) \in \{0,1\}^*$ and a starting point p.

Output: A folding of the string S on the 2D square lattice.

1. Lay p and $p+1$ and their adjacent vertices as shown in Figure 4-13. Let $i = j = 1$.

2. Iteration: Consider $B_{\mathcal{E}}(i)$ and $B_{\mathcal{O}}(j)$. There are four cases.

(i) $\ell_{\mathcal{E}}(i) = 1$ and $\ell_{\mathcal{O}}(j) = 1$: Fold $B_{\mathcal{E}}(i), B_{\mathcal{E}}(i+1), B_{\mathcal{O}}(j)$, and $B_{\mathcal{O}}(j+1)$ as in Figures 4-14(a) and 4-15(a). Set $i = i+2$ and $j = j+2$. The idea is to make sure there are three contacts: one between the i^{th} even-1 and j^{th} odd-1, one between the $i+1^{th}$ even-1 and j^{th} odd-1, and one between the $i+1^{th}$ even-1 and $j+1^{th}$ odd-1.

(ii) $\ell_{\mathcal{E}}(i) \geq 3$ and $\ell_{\mathcal{O}}(j) \geq 3$: Fold $B_{\mathcal{E}}(i), B_{\mathcal{E}}(i+1), B_{\mathcal{O}}(j)$, and $B_{\mathcal{O}}(j+1)$ as in Figures 4-14(b) and 4-15(b). Set $i = i+2$ and $j = j+2$. The idea is that same as in case (a), except we must move the segments $B_{\mathcal{E}}(i)$ and $B_{\mathcal{O}}(j)$ out of the way if either $\ell_{\mathcal{E}}(i) \geq 3$ or $\ell_{\mathcal{O}}(j) \geq 3$.

(iii) $\ell_{\mathcal{E}}(i) = 1$ and $\ell_{\mathcal{O}}(j) \geq 3$: Fold $B_{\mathcal{E}}(i), B_{\mathcal{O}}(j)$, and $B_{\mathcal{O}}(j+1)$ as in Figures 4-14(c) and 4-15(c). Set $i = i+1$ and $j = j+2$. The idea is to make sure there are two contacts: one between the i^{th} even-1 and the j^{th} odd-1 and one between the i^{th} even-1 and the $j+1^{th}$ odd-1.

(iv) $\ell_{\mathcal{E}}(i) \geq 3$ and $\ell_{\mathcal{O}}(j) = 1$: Fold $B_{\mathcal{E}}(i), B_{\mathcal{E}}(i+1)$, and $B_{\mathcal{O}}(j)$ as in Figure 4-14(d) and in the mirror image of Figure 4-15(c). Set $i = i+2$ and $j = j+1$. The idea here is the same as in case (c) except here there are two contacts for the j^{th} odd-1 and one contact for the i^{th} and $i+1^{th}$ even-1.

3. Repeat Step 2 while $B_{\mathcal{E}}(i)$ and $B_{\mathcal{O}}(j)$ do not overlap.

4.3.2 Analysis

Theorem 10. *Given a binary string S, the* STRING FOLDING ALGORITHM *finds a folding with at least $M_2[S]/3$ contacts, i.e. a $\frac{1}{3}$-approximation.*

Proof. Without loss of generality, assume there are k more case (c) folds than case (d) folds, where $k \geq 0$. We will count how many contacts the odd-1's are involved in. (If there are more case (d) folds than case (c) folds, we would count how many contacts the even-1's are involved in.) Consider the folding of a string S found by the algorithm. Let i^* and j^* be the value of i and j during the last iteration of the algorithm. Then $\mathcal{O}[p+1, p+2, \ldots j^*]$ denotes the number of odd-1's that are guaranteed to be used in some contact(s). How many odd-1's are not necessarily in

Figure 4-13: Placement of vertices $p-2, \ldots p+1$.

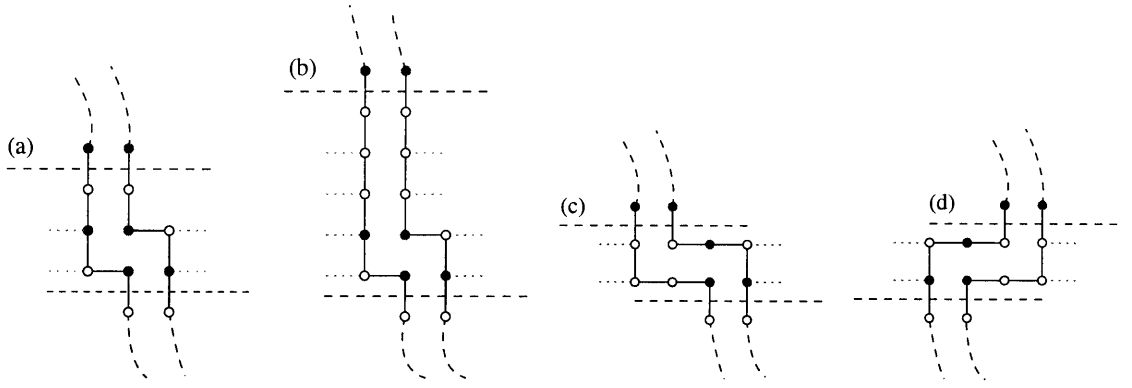

Figure 4-14: Case (a), (b), (c), and (d) folds.

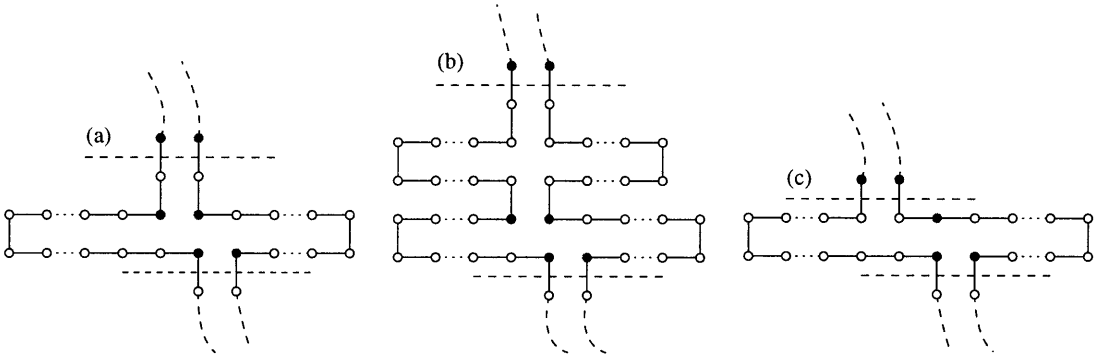

Figure 4-15: Foldings for higher values of $\ell_{\mathcal{E}}(i) \geq 5$ and $\ell_{\mathcal{O}}(j) \geq 5$.

any contacts? The odd-1's in the string $p-2, p-3, \ldots i^*$ are not necessarily used in any contacts. By Lemma 2, we have:

$$\mathcal{O}[p-2, p-3, \ldots i^*] \leq \mathcal{E}[p-2, p-3, \ldots i^*]$$

$$\mathcal{O}[S] = \mathcal{O}[p+1, p+2, \ldots j^*] + \mathcal{O}[p-2, p-3, \ldots i^*]$$

Combining equations (4.3) and (4.3), we have:

$$\mathcal{O}[S] \leq \mathcal{O}[p+1, p+2, \ldots j^*] + \mathcal{E}[p-2, p-3, \ldots i^*]$$

We assumed that there are k more case (c) folds than case (d) folds. Let's pair up each case (d) fold with a case (c) fold and call each of these pairs a (c-d)-fold. Thus, the number of odd-1's used in case (a), case (b), or (c-d) folds is $\mathcal{O}[p+1, p+2, \ldots j^*] - 2k$ and $2k$ odd-1's are used in unpaired case (c) folds, since each case (c) fold uses two odd-1's. The number of even-1's used in case (a), case (b), or case (c-d) folds is also $\mathcal{O}[p+1, p+2, \ldots j^*] - 2k$, since in these folds the number of even-1's used is the same as the number of odd-1's. Then there are k even-1's used in the extra case (c) folds. Thus,

$$\mathcal{E}[p-2, p-3, \ldots i^*] = \mathcal{O}[p+1, p+2, \ldots j^*] - k$$

Combining (4.3) and (4.3), we have:

$$\mathcal{O}[S] \leq (\mathcal{O}[p+1, p+2, \ldots j^*]) + (\mathcal{O}[p+1, p+2, \ldots j^*] - k)$$

Equation (4.3) can be rewritten as:

$$\mathcal{O}[p+1, p+2, \ldots j^*] \geq \frac{\mathcal{O}[S]}{2} + \frac{k}{2} \tag{4.3}$$

If we consider the subset of the odd-1's in the string $p+1, p+2, \ldots j^*$ involved in case (a), case (b), or (c-d) folds, we note that there are at least four contacts for every three odd-1's. (i.e. In case (a) and case (b) folds, we have three contacts for every two odd-1's, and in case (c-d) folds, we have four contacts for every three odd-1's.) In the unpaired case (c) folds, we have at least one contact for every odd-1. Thus, the number of contacts we have is at least:

$$\frac{4}{3}(\mathcal{O}[p+1, p+2, \ldots j^*] - 2k) + 2k \tag{4.4}$$

Using equation (4.3), we have that the quantity in equation (4.4) is at least:

$$(\frac{\mathcal{O}[S]}{2} - \frac{3k}{2})\frac{4}{3} + 2k = \frac{2\mathcal{O}[S]}{3} \tag{4.5}$$

Recall that $\mathcal{O}[S] = \mathcal{E}[S]$ by assumption, which implies that $M_2[S] = 2\mathcal{O}[S]$.

Therefore, the number of contacts that the algorithm achieves is at least $M_2[S]/3$. □

The algorithm runs in $O(n)$ time where n is the number of vertices in $L(S)$. We can find point p in $O(n)$ time. Finding $B_{\mathcal{E}}(i)$ and $B_{\mathcal{O}}(i)$ and folding these blocks takes time proportional to the size of the blocks, but since each vertex is included in only one of the blocks, the total time it takes to find all the blocks and fold them is $\Theta(n)$.

4.4 A Linear Program for String Folding

We present an integer program for the 2D string folding problem and analyze the upper bounds given by its respective linear programming relaxation. Our formulation is similar to those studied previously in [GHL], which appears to contain the only other description of this problem as an integer program. However, we are able to analyze the strength of our linear programming relaxation, which has not been considered previously for this problem.

The main idea behind this linear program is to use the assignment constraints discussed in Section 3.1.1. Let I be the set of indices for the vertices in a given string S of length n, i.e. $I = \{1, \ldots, n\}$. Let V be the set of lattice points on a 2D square lattice; let $V_{\mathcal{O}}$ and $V_{\mathcal{E}}$ be the two bipartite sets of lattice points. Then we have a variable x_{iv} for each $i \in I, v \in V$. This variable $x_{iv} = 1$ if vertex i is placed on lattice point v. Since the square lattice is bipartite, we arbitrarily label one of the bipartitions "odd" and the other set "even". We will refer to these sets as $V_{\mathcal{O}}$ and $V_{\mathcal{E}}$, respectively. We will let $\mathcal{O}$ ($\mathcal{E}$) denote the set of odd (even) indices in the string S We break down $\mathcal{E}$ and $\mathcal{O}$ further as follows: $H_{\mathcal{O}}$ is the set of indices of odd-1's in S, $H_{\mathcal{E}}$ is the set of indices of even-1's in S.

Since vertices in the string with odd (even) indices can only be placed on odd (even) lattice points, we have the following constraints that require that each vertex in the string is assigned to some lattice point in an integral solution.

$$\begin{aligned} \sum_{v \in V_{\mathcal{O}}} x_{iv} &= 1, \quad \forall i \in \mathcal{O}, \\ \sum_{v \in V_{\mathcal{E}}} x_{jw} &= 1, \quad \forall j \in \mathcal{E}. \end{aligned}$$

We also require that each lattice point be occupied by at most one vertex. This requirement will be met in an integral solution if we use the following constraint:

$$\begin{aligned}\sum_{i\in\mathcal{O}} x_{iv} &\leq 1, \quad \forall v \in V_{\mathcal{O}} \\ \sum_{j\in\mathcal{E}} x_{jw} &\leq 1, \quad \forall w \in V_{\mathcal{E}}.\end{aligned}$$

Additionally, we need to enforce that in an integral solution, adjacent vertices in the string occupy adjacent lattice points. In an integral solution, the following constraint ensures that for any vertex assigned to a lattice point, the neighboring vertex with lower index is assigned to a neighboring lattice point. We denote the set of lattice points adjacent to lattice point v as $\delta(v)$. We call these constraints *connectivity* constraints.

$$\sum_{w\in\delta(v)} x_{i-1,w} \geq x_{iv} \quad \forall i \in I \setminus \{n\}, v \in V \tag{4.6}$$

$$\sum_{w\in\delta(v)} x_{i+1,w} \geq x_{iv} \quad \forall i \in I \setminus \{n\}, v \in V \tag{4.7}$$

We use the variable $h_{(v,w)}$ to record the number of contacts made across edge (v, w), i.e. $h_{(v,w)} = 1$ if there is an odd-1 on lattice point V and an even-1 on lattice point w. We also introduce additional constraints called *backbone* constraints. We have a variable E^{-}_{ivw} for each vertex $i \in I$ and each edge (v, w) in the lattice. In an integral solution, the variable $E^{-}_{ivw} = 1$ if vertex i is on lattice point v and vertex $i - 1$ is on lattice point w. Similarly, the variable $E^{+}_{ivw} = 1$ if vertex i is on lattice point v and vertex $i + 1$ is on lattice point w.

$$\begin{aligned}\sum_{w\in\delta(v)} E^{+}_{ivw} = \sum_{w\in\delta(v)} E^{-}_{ivw} = x_{iv}, \quad \forall i \in \mathcal{O}, v \in V_{\mathcal{O}} \\ \sum_{v\in\delta(w)} E^{+}_{j-1,vw} = \sum_{v\in\delta(w)} E^{-}_{j+1,vw} = x_{jw}, \quad \forall j \in \mathcal{E}, w \in V_{\mathcal{E}}\end{aligned} \tag{4.8}$$

Lemma 11. *Backbone constraints* (4.8) *imply the connectivity constraints* (4.6) *and* (4.7).

Proof. From the backbone constraints, we have:

$$x_{iv} = \sum_{w\in\delta(v)} E^{-}_{ivw}.$$

For each variable $x_{i-1,w}$, we also have:

$$x_{i-1,w} = \sum_{u \in \delta(w)} E^{+}_{i-1,wu}.$$

This last constraint implies that $x_{i-1,w} \geq E^{+}_{i-1,wv}$, since $v \in \delta(w)$. Note that $E^{+}_{i-1,wv} = E^{-}_{ivw}$. For each of terms in the first constraint in this proof, we can obtain the inequality $x_{i-1,w} \geq E^{-}_{ivw}$. Thus, we have the desired inequality:

$$x_{iv} \leq \sum_{w \in \delta(v)} x_{i-1,w}.$$

We can repeat this argument to derive constraint (4.7). □

IP$_{FOLD}$:

$$\begin{aligned}
& & \max & \sum_{(v,w) \in E} h_{(vw)} \\
\textit{subject to}: & \sum_{v \in V_{\mathcal{O}}} x_{iv} & = & 1, \quad \forall i \in \mathcal{O} \\
& \sum_{v \in V_{\mathcal{E}}} x_{jw} & = & 1, \quad \forall j \in \mathcal{E} \\
& \sum_{i \in \mathcal{O}} x_{iv} & \leq & 1, \quad \forall v \in V_{\mathcal{O}} \\
& \sum_{j \in \mathcal{E}} x_{jw} & \leq & 1, \quad \forall w \in V_{\mathcal{E}} \\
\sum_{w \in \delta(v)} E^{+}_{ivw} = \sum_{w \in \delta(v)} E^{-}_{ivw} & & = & x_{iv}, \quad \forall i \in H_{\mathcal{O}}, v \in V_{\mathcal{O}} \\
\sum_{v \in \delta(w)} E^{+}_{j-1,vw} = \sum_{v \in \delta(w)} E^{-}_{j+1,vw} & & = & x_{jw}, \quad \forall j \in H_{\mathcal{E}}, w \in V_{\mathcal{E}} \\
\sum_{i \in H_{\mathcal{O}}} E^{-}_{ivw} + \sum_{i \in H_{\mathcal{O}}} E^{+}_{ivw} + h_{(v,w)} & & \leq & \sum_{i \in H_{\mathcal{O}}} x_{iv}, \quad \forall v \in V_{\mathcal{O}} \\
\sum_{j \in H_{\mathcal{E}}} E^{-}_{j+1,vw} + \sum_{j \in H_{\mathcal{E}}} E^{+}_{j-1,vw} + h_{(v,w)} & & \leq & \sum_{j \in H_{\mathcal{E}}} x_{jw}, \quad \forall v \in V_{\mathcal{E}} \\
E^{\pm}_{ivw}, x_{iv}, x_{jw}, h_{(vw)} & & \in & \{0,1\} \quad \forall i \in \mathcal{O}, j \in \mathcal{E}, (v,w) \in E.
\end{aligned}$$

We relax the integrality constraint for the above integer program to obtain:

$$0 \leq E^{\pm}_{ivw}, x_{iv}, x_{jw}, h_{(vw)} \leq 1, \quad \forall i \in \mathcal{O}, j \in \mathcal{E}, (v,w) \in E.$$

For a specified input string S, we refer to the optimal value of the resulting linear program as $\mathrm{LP}_{FOLD}(S)$.

Lemma 12. *For any string S, the optimal solution for $LP_{FOLD}(S)$ is at most $2 \cdot \min\{\mathcal{O}[S], \mathcal{E}[S]\} + 2$.*

Proof. The optimal solution for the linear program is $\sum_{(v,w)\in E} h_{(vw)}$. Without loss of generality, we assume $\mathcal{O}[S] \leq \mathcal{E}[S]$. The last constraint in the linear program can be rewritten as follows:

$$h_{(vw)} \leq \sum_{i \in H_{\mathcal{O}}} x_{iv} - \sum_{i \in H_{\mathcal{O}}} E^{-}_{ivw} - \sum_{i \in H_{\mathcal{O}}} E^{+}_{ivw}.$$

Summing over all the edges, we have:

$$\sum_{(v,w)\in E} h_{(vw)} \leq \sum_{(v,w)\in E} \sum_{i \in H_{\mathcal{O}}} x_{iv} - \sum_{(v,w)\in E} \sum_{i \in H_{\mathcal{O}}} E^{-}_{ivw} - \sum_{(v,w)\in E} \sum_{i \in H_{\mathcal{O}}} E^{+}_{ivw}.$$

The first sum is upper bounded by $4\mathcal{O}[S]$. To show this, first we note that:

$$\sum_{v \in V_{\mathcal{O}}} x_{iv} = 1.$$

If we sum over all edges, as opposed to all odd vertices, note that each odd vertex $v \in V_{\mathcal{O}}$ is an endpoint in at most 4 edges. Thus, we have:

$$\sum_{(v,w)\in E} x_{iv} = \sum_{v \in V_{\mathcal{O}}} \sum_{w \in \delta(v)} x_{iv} = \sum_{w \in \delta(v)} \sum_{v \in V_{\mathcal{O}}} x_{iv} = \sum_{w \in \delta(v)} 1 \leq 4,$$

$$\sum_{(v,w)\in E} \sum_{i \in H_{\mathcal{O}}} x_{iv} = \sum_{i \in H_{\mathcal{O}}} \sum_{(v,w)\in E} x_{iv} \leq \sum_{i \in H_{\mathcal{O}}} 4 = 4\mathcal{O}[S].$$

Now we analyze the following sum:

$$\sum_{(v,w)\in E} \sum_{i \in H_{\mathcal{O}}, i \neq 1} E^{-}_{ivw} = \sum_{i \in H_{\mathcal{O}}, i \neq 1} \sum_{(v,w)\in E} E^{-}_{ivw}.$$

Each variable E^{-}_{ivw} is associated with a unique odd vertex, i.e. the odd vertex v. We have the following constraints for each odd vertex:

$$\sum_{w \in \delta(v)} E^{-}_{ivw} = x_{iv} \quad \forall i \in H_{\mathcal{O}}, v \in V_{\mathcal{O}}.$$

Thus, we can rewrite the sum as follows:

$$\sum_{i\in H_{\mathcal{O}},i\neq 1}\sum_{(v,w)\in E} E^{-}_{ivw} = \sum_{i\in H_{\mathcal{O}},i\neq 1}\sum_{v\in V_{\mathcal{O}}}\sum_{w\in\delta(v)} E^{-}_{ivw} = \sum_{i\in H_{\mathcal{O}},i\neq 1}\sum_{v\in V_{\mathcal{O}}} x_{iv} = \sum_{i\in H_{\mathcal{O}},i\neq 1} 1 = \mathcal{O}[S]-1.$$

Note that:

$$\sum_{(v,w)\in E} E^{-}_{ivw} = \sum_{(v,w)\in E} E^{+}_{ivw}.$$

Thus,

$$\sum_{i\in H_{\mathcal{O}},i\neq 1}\sum_{(v,w)\in E} E^{-}_{ivw} = \sum_{i\in H_{\mathcal{O}},i\neq n}\sum_{(v,w)\in E} E^{+}_{ivw} = \mathcal{O}[S]-1.$$

Therefore, we have:

$$\sum_{(v,w)\in E} h_{(vw)} \leq 4\mathcal{O}[S] - (\mathcal{O}[S]-1) - (\mathcal{O}[S]-1) \leq 2\mathcal{O}[S]+2.$$

So the maximum value of the objective function is $M_2[S] = 2\cdot\min\{\mathcal{O}[S],\mathcal{E}[S]\}+2$. □

Theorem 13. $LP_{FOLD}(S) \leq 3 \cdot IP_{FOLD}(S)$.

Proof. According to Lemma 10, we can always achieve a folding with value $M_2[S]/3$. We have $\mathrm{IP}_{FOLD}(S) \leq M_2[S]$ and $M_2[S]/3 \leq \mathrm{IP}_{FOLD}(S)$. This implies the lemma. □

4.5 Gap Examples

In this section, we examine the gaps between our upper bounds and the integral optimal values for the 2D folding problem.

4.5.1 Gap for 2D Combinatorial Bound

First, we examine the upper bound for the 2D folding problem presented in Section 4.2. Recall that the upper bound on the number of contacts possible for a folding of the string S is $M_2[S] = 2\cdot\min\{\mathcal{O}[S],\mathcal{E}[S]\}+2$. How good is this bound? In the previous section, we saw that $OPT/M_2[S] \geq 1/3$ for any string S. In this section, we describe a string S for which $OPT/M_2[S] = 1/2 + o(1)$. Thus, this upper bound cannot be used to obtain an approximation factor better than $\frac{1}{2}$.

Let $\hat{S} = \{0\}^{4k^2}\{01\}^k\{0\}^{8k^2}\{1000\}^k\{0\}^{4k^2}$ for an integer $k > 0$. We will show that no folding of $\hat{S}$ has more than $(1+o(1))M_2[\hat{S}]/2$ contacts.

Theorem 14. *No folding of $\hat{S}$ on the 2D square lattice results in more than* $(1 + o(1)) \min\{\mathcal{O}[S], \mathcal{E}[S]\}$ *contacts.*

Note that for the string $\hat{S}$, there are k even-1's and k odd-1's. Thus, $k = M_2[\hat{S}]/2$ for the string $\hat{S}$. So we need to show that no folding contains more than $(1 + o(1))k$ contacts. To prove Theorem 14, we consider two strings S_1 and S_2 such that $\hat{S} = S_1 S_2$. Let $S_1 = \{0\}^q\{01\}^k\{0\}^q$ and let $S_2 = \{0\}^q\{1000\}^k\{0\}^q$, where $q = 4k^2$. All the 1's in S_1 are even-1's and all the 1's in S_2 are odd-1's. Note that since all the 1's in S_1 are even-1's, no folding of $\hat{S}$ contains a contact between a pair of 1's from S_1. Similarly, no folding of $\hat{S}$ contains a contact between a pair of 1's from S_2, since all the 1's in S_2 are odd-1's. Thus, we can assume that all contacts are comprised of an even-1 from S_1 and an odd-1 from S_2. Therefore, it suffices to show that no folding of $\hat{S}$ results in more than $(1 + o(1))k$ contacts *between* the two strings S_1 and S_2.

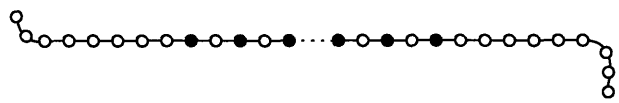

Figure 4-16: The string $S_1 = \{0\}^q\{01\}^k\{0\}^q$ is the "even string".

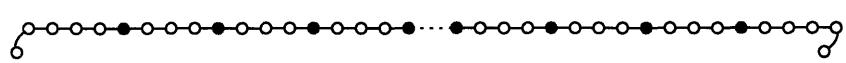

Figure 4-17: The string $S_2 = \{0\}^q\{1000\}^k\{0\}^q$ is the "odd string".

Since we are only concerned with contacts between the strings S_1 and S_2, we focus on foldings of these two strings rather than on foldings of $\hat{S}$. Note that for any folding of $\hat{S}$, there is a folding of S_1 and S_2 that has at least as many contacts as the folding of $\hat{S}$. This is because S_1 and S_2 are substrings of $\hat{S}$. Thus, proving that no folding of the two strings S_1 and S_2 results in more than $(1 + o(1))k$ contacts would prove Theorem 14.

Suppose that for each of the strings S_1 and S_2, we color one side red and the other side blue. A contact is a red-red contact if the red sides face each other in the contact, or a red-blue contact if one red side faces a blue side in the contact. Some examples of red-red contacts are illustrated in Figure 4-18. There are four types of contacts if we always consider the color of the S_1 string first: red-red, red-blue, blue-red, and blue-blue. We now show that it is only possible to have one type of contact between S_1 and S_2 in any folding. In other words, if some contact is a red-red contact, then all the contacts must be red-red contacts. Thus, we only have to consider foldings in which all contacts are of one type. If an odd-1 is involved in two contacts, both must

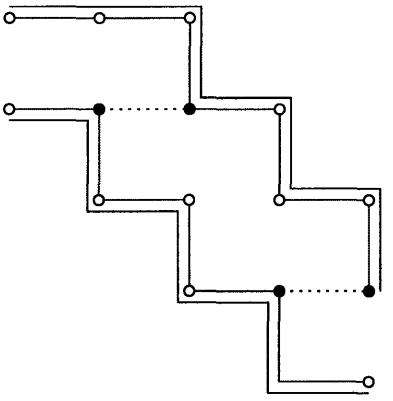

Figure 4-18: Some red-red contacts.

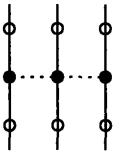

Figure 4-19: These two contacts are each of a different type.

be with even-1's on the same side of the odd-1. For example, we can ignore contacts such as those shown in Figure 4-19.

Lemma 15. *In any folding of $\hat{S}$, all contacts between S_1 and S_2 are of the same type.*

Proof. Assume for the sake of contradiction that there is some folding of S_1 and S_2 with at least two different types of contacts (of the four possible types). Let c_1 be a red-red contact and c_2 be a blue-blue contact, as shown in Figure 4-20. Suppose c_1 is a contact between x_1 and y_1 where x_1 is an even-1 in S_1 and y_1 is an odd-1 in S_2. Similarly, c_2 is a contact between x_2 and y_2, where x_2 is an even-1 in S_1 and y_2 is an odd-1 in S_2.

Then there is a closed path from y_1 to y_2 along S_2, from y_2 to x_2, from x_2 to x_1 along S_1 and from x_1 back to y_1. Note that the farthest distance between any two 1's is $2k-1$ in S_1 and $4k-1$ in S_2. Thus, the total length of this closed path is no more than $6k$. However, as shown in Figure 4-20, at least one of the substrings of 0's at the end of S_1 or S_2 is enclosed by this path. The number of 0's in this substring is $4k^2$. But this is a contradiction, because the maximum number of lattice points that can fit an enclosed area of perimeter $6k$ is $9k^2/4$. We obtain the same contradiction for the other possible arrangement of a red-red and a blue-blue contact as shown in

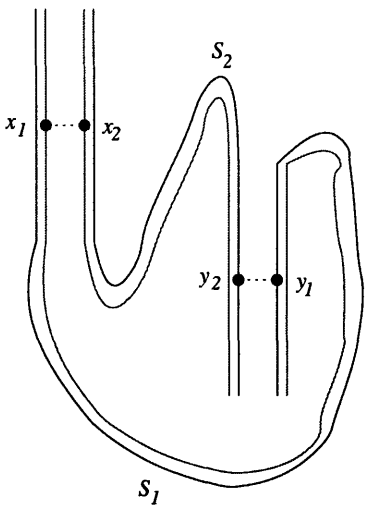

Figure 4-20: One way to connect a red-red and a blue-blue contact.

Figure 4-21. Furthermore, we obtain the same contradiction for any arrangement of a red-red contact and a red-blue contact or a red-red contact and a blue-red contact. This can be verified by inspecting the two cases (which are similar to the two possible arrangements of a red-red and a blue-blue contact) corresponding to each of these pairs of different types of contacts. □

Now we consider the case in which all contacts are of the same type. Without loss of generality, we assume they are all red-red.

Lemma 16. *There are at most* $(1 + o(1))k$ *red-red contacts between* S_1 *and* S_2.

Proof. We will show that the average number of contacts per even-1 cannot exceed $(1 + o(1))$. We note that if an even-1 has two contacts, then its two contacts much be perpendicular to each other since both are red-red contacts.

We will use the following notions in our proof. First, we assume that the contacts are ordered consecutively in a folding of S_1 and S_2. We begin folding S_1 and S_2 by considering a specific endpoint for each of these strings. Then the even-1 in S_1 and the odd-1 in S_2 that are closest to these endpoints and that make a contact are in the first contact. The next closest even-1 in S_1 and the next closest odd-1 in S_2 that make a contact are in the second contact, etc. Note that consecutive contacts, e.g. the two contacts involving x in Figure 4-22, may involve the same odd-1 or same even-1.

Next, we associate an orientation with each contact. A contact can have orientation up, down, right, or left. For example, in Figure 4-22, say that the horizontal contact involving the even-1 labeled x is the first contact in the folding, and the vertical contact involving x is the second, etc. Then the first contact has orientation up and the second contact has orientation left. A horizontal contact has orientation up if the next contact is above it and down if the next contact is below it. A vertical

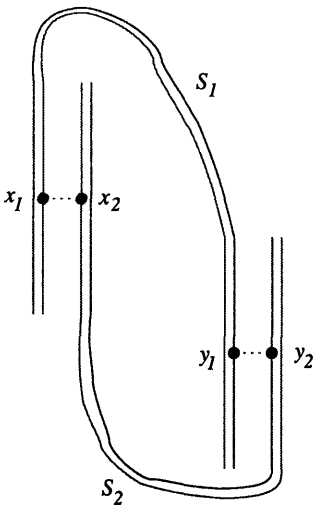

Figure 4-21: Another way to connect a red-red and a blue-blue contact.

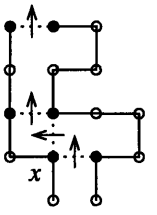

Figure 4-22: The arrows indicate the orientation of each contact.

contact has orientation left if the next contact is to its left and right if the next contact is to its right. Note that the orientation of a contact is only well-defined if the next even-1 or the next odd-1 in the string are involved in a contact.

Let x be the first even-1 to have two contacts. Without loss of generality, assume that the first of these contacts is oriented up. The two possibilities for this situation are shown in Figure 4-23. First, we consider case (a) in Figure 4-23. Say that x has an up and a left contact as in case (a). If the next even-1 also has two contacts, then its second contact will have a down orientation as shown in Figure 4-24(a). If the next even-1 has only one contact, but the next *next* even-1 has two contacts, then *its* second contact will have a down orientation, as shown in Figure 4-24(b). In other words, consider the next even-1 (call it y) after x that has contacts with two odd-1's. If all the even-1's between x and y have at least one contact, then the orientation of y's contacts makes a counter-clockwise turn. If some even-1 between x and y does not have any contacts, then the second contact of y may have a left orientation. So in this case, we are not in a downward orientation (i.e. we have not made a counter-clockwise turn), but we do not have more than one contact per even-1 on average for

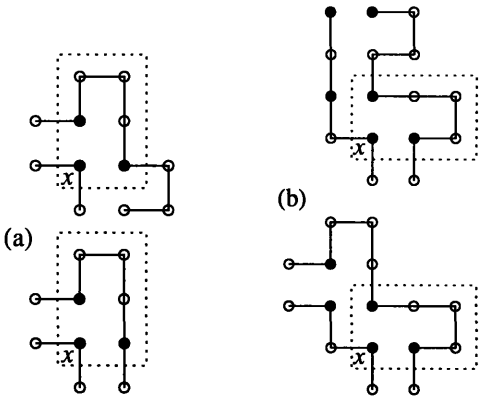

Figure 4-23: x is the first even-1 in the folding with two contacts.

the set of even-1's between x and y.

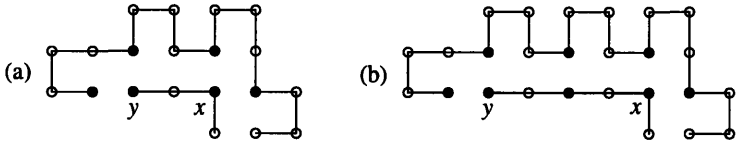

Figure 4-24: y is the next even-1 after x to have two contacts.

If the next even-1 after x has only one contact, it can have a left or a down orientation, but it cannot have an up orientation. In order for a contact to have an up orientation, we need to make a clockwise turn. However, for every clockwise turn, there will be two even-1's with no contacts. To see this, consider Figure 4-25. Now suppose r and s make a contact as shown in Figure 4-25. Note that r can be in the same situation as x is in in Figure 4-23(a) or (b). If r is in case (a) and we make another clockwise turn and then go back to case (a), etc., then we will average less than 1 contact for each even-1. If r is in the same position as x in Figure 4-23(b), then we can make a counter-clockwise turn so that the next two even-1's will have two contacts each. But in this case, we will average only one contact per even-1 over the course of a counter-clockwise and clockwise turn.

Next, we consider case (b) in Figure 4-23(b). If x is in case (b), then the even-1 that follows x will have one contact as shown in the first figure in Figure 4-23(b) or it will have two contacts and be in the same position as x is in in case (a). Thus, if we start from case (b), we can get only one more contact than if we were to start in case (a).

Therefore, the only way to fold the string so that a constant fraction of the even-

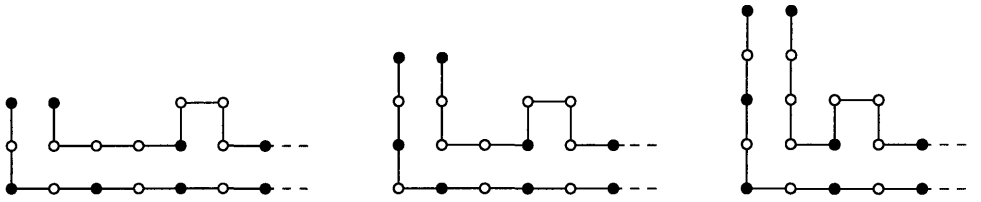

Figure 4-25: If the orientation of the contacts makes a clockwise turn, then two even-1's have no contacts.

1's are contained in more than one contact is to have more counter-clockwise turns than clockwise turns. In this case, the string forms a "spiral", as shown in Figure 4-26. Every time we make a counter-clockwise turn in this configuration, we can have an even-1 with two contacts. How many counter-clockwise turns can we make? After completing the first four counter-clockwise turns in the spiral, we have four even-1's with two contacts each. Then, one out of the next five even-1's has two contacts, then one out of next six, one out of the next seven, etc. Thus, the total number of even-1's with two contacts each is $\sqrt{2k}$. The total number of contacts is $k + \Theta(1) + \sqrt{2k} = (1 + o(1))k$. □

Theorem 14 follows from Lemmas 15 and 16.

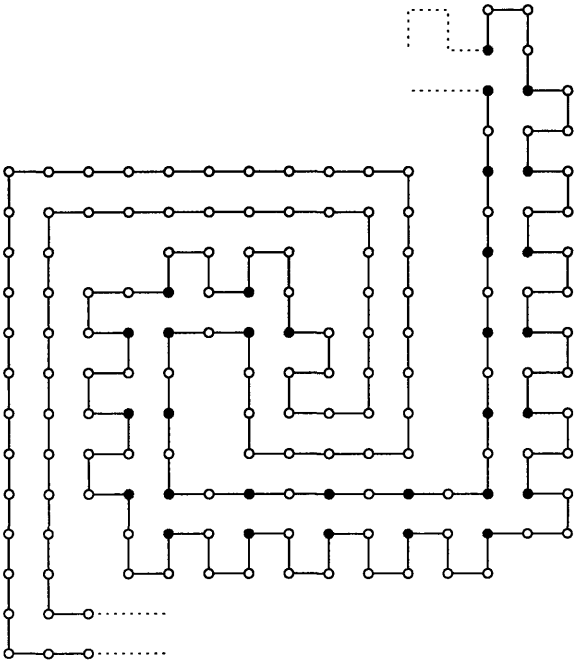

Figure 4-26: A "spiral" configuration of $\hat{S}$.

4.5.2 LP Integrality Gap

We show that the integrality gap for the linear programming relaxation in Section 4.4 is at least $2 - \epsilon$ for any $\epsilon > 0$. We demonstrate this gap using the same string that demonstrates a gap for the combinatorial bound in Section 4.5.1. Let the string $\hat{S} = \{0\}^q\{01\}^k\{0\}^{2q}\{1000\}^k\{0\}^q$, where k is a positive integer and $q = 4k^2$. In Theorem 14, it is shown that no folding of $\hat{S}$ has more than $(1 + o(1))\mathcal{O}[S]$ contacts. However, we can construct a fractional solution for LP_2 for which the objective function is $2\mathcal{O}[S] - 4$.

In Figure 4-27, we show how to fold the string $\hat{S}$ *fractionally.* Each vertex is placed on a single lattice point, except the three vertices directly following the vertex labeled y and the three vertices directly following the vertex labeled z. These six vertices are fractionally folded, so that the string is allowed to cross itself, which cannot happen in an integral folding.

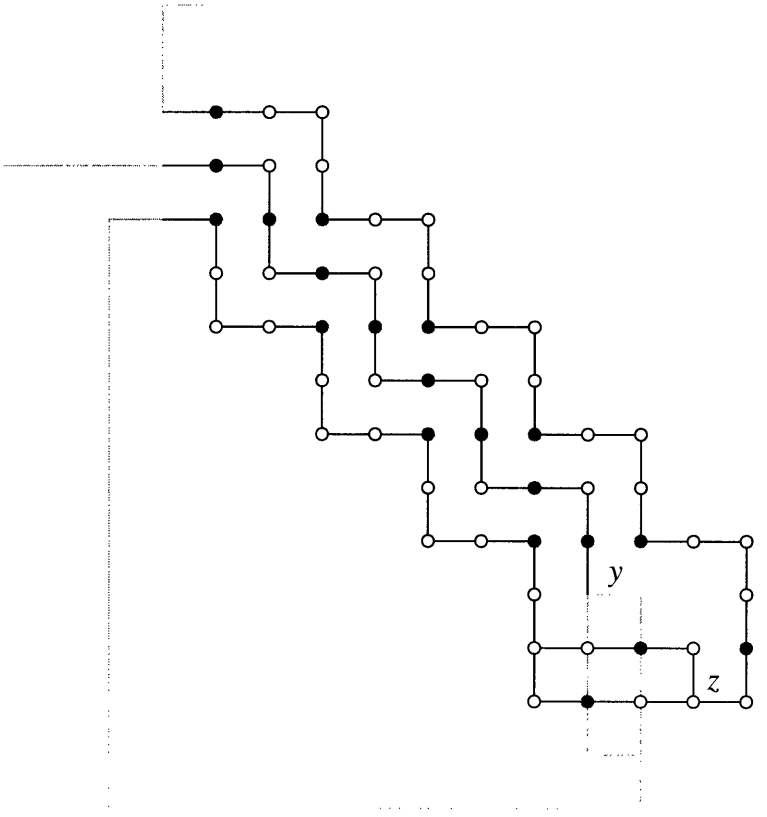

Figure 4-27: Let $S_1 = \{01\}^k$ and let $S_2 = \{0001\}^k$. The string splits in half at points y and z, which allows the string to cross itself, something not allowed in an integral solution.

4.6 Discussion and Open Problems

The main open problem related to this chapter is to find an algorithm for the 2D string folding problem with an approximation ratio greater than $\frac{1}{3}$. Possible ways to design such a new algorithm include rounding the linear programming relaxation, develop new combinatorial methods, or find strengthened upper bounds.

There is no evidence that the combinatorial upper bound given in Equation (4.1) cannot be used to obtain an algorithm with an approximation guarantee as high as $\frac{1}{2}$. In fact, for any particular string, it does not seem hard to find a folding in which the number of contacts is at least half of this upper bound. Thus, intuitively, it seems that one should be able to design an approximation algorithm in which the number of contacts is at least half of this upper bound. We formalize this intuition with the following conjecture.

Conjecture 17. *For every binary string, there is a folding in which the number of contacts is at least half of the combinatorial upper bound in Equation* (4.1).

This conjecture can be settled in the affirmative if we could find an algorithm achieving the stated number of contacts. It would be disproved if one could find a family of strings in which the optimal number of contacts is asymptotically less than half of the combinatorial upper bound.

Another question to address is, can the methods used in the STRING FOLDING ALGORITHM be used to improve the approximation guarantee? It is easy to show that there is a family of strings for which the algorithm achieves no more than $\frac{1}{3}$ of the combinatorial upper bound, i.e. let $S = 11111\ldots1001001001001\ldots$ such that the substring containing consecutive 1's contains as many 1's as in the substring containing 0's. If the starting point of the 2D folding algorithm is the point between these two substrings, then the algorithm outputs a folding in which the number of contacts is $\frac{1}{3}$ of the combinatorial upper bound. However, if we use a different starting point, namely a starting point in the middle of the first substring, the approximation ratio will be at least $\frac{3}{8}$ since we will only use case (a) and case (b) folds. (Note that there is also a simple folding for this string with an approximation guarantee of $\frac{1}{2}$: place the first substring so that the first half is adjacent to the second half and do the same for the second substring.)

Jothi and Raghavachari implemented a variation of the STRING FOLDING ALGORITHM in which they tested *all possible* starting points in our 2D folding algorithm. Experimentally, they are unable to find a string for which this algorithm has an approximation ratio of less than $\frac{3}{8}$ with respect to the combinatorial upper bound [JR]. They pose the following conjecture.

Conjecture 18. **[JR]** *Given a binary string S of length n, if the* STRING FOLDING ALGORITHM *is run n times with each of n possible choices of a starting point p, the number of contacts in the best resulting folding is at least $\frac{3}{8}$ of the combinatorial upper bound.*

Chapter 5

3D String Folding

5.1 Introduction

In this chapter, we consider the string folding problem on the three-dimensional (3D) square lattice, i.e. the 3D version of the string folding problem discussed in Chapter 4. The string folding problem is defined in Section 4.1. We will review it here, but for a precise problem statement, we refer the reader to Section 4.1.

The 3D string folding problem is a simple combinatorial problem. The input to the string folding problem is a string: a path graph in which each vertex except the two endpoints has degree exactly two and each end vertex has degree one. Each vertex in the string is labeled 1 or 0. Throughout this chapter, we will refer to such a path graph as a string S in $\{0,1\}^*$.

We will fold the string on the 3D square lattice, which is a graph in the (x,y,z)-plane in which the vertices are all points with integral coordinates. The edges connect pairs of vertices that are at distance one. We say a vertex from the string is *placed* on a lattice point (x,y,z) if that vertex is assigned to lattice point (x,y,z). A *folding* of such an input string corresponds to placing the vertices of the graph on a lattice subject to the following three constraints:

(i) Each lattice point can have at most one vertex placed on it.

(ii) Each vertex must be placed on *some* lattice point.

(iii) Adjacent vertices in the string must be placed on adjacent lattice points.

For example, suppose vertex i and $i+1$ are adjacent in the input graph. On a 3D

square lattice, if vertex i is placed on lattice point (x, y, z), then vertex $i+1$ must be placed on one of six possible lattice points: $(x \pm 1, y, z)$, $(x, y \pm 1, z)$, or $(x, y, z \pm 1)$.

There are many possible valid foldings for an input string. We are interested in finding certain types of foldings. With respect to a particular folding of an input string, we say a pair of vertices forms a *contact* if they are not adjacent on the string, they are both labeled 1, and they are placed on neighboring lattice points. The goal of the string folding problem is to find a folding of the input string that maximizes the number of contacts.

Figure 5-1 shows a 3D folding for a string. Throughout the figures in this chapter, vertices labeled 1 are denoted by black dots and vertices labeled 0 are denoted by white or unfilled dots.

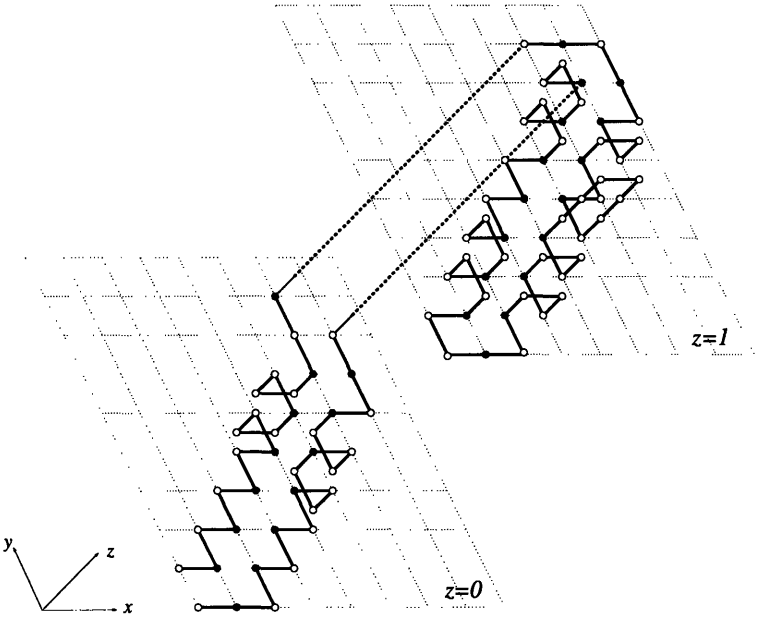

Figure 5-1: In this folding, all contacts are formed on or between the 2D planes $z = 0$ (lower) and $z = 1$ (upper). Black dots represent 1's and white dots represent 0's.

5.1.1 Background

Berger and Leighton proved that the 3D string folding problem is NP-hard [BL98]. On the positive side, Hart and Istrail gave a simple algorithm with an approximation guarantee of $\frac{3}{8}OPT - \Theta(\sqrt{OPT})$ [HI96]. This algorithm uses their $\frac{1}{4}$-approximation algorithm for the 2D string folding as a subroutine and is described in Section 4.2. Improving on the approximation guarantee of $\frac{3}{8}$ for the 3D folding problem has been an open problem for almost a decade.

5.1.2 Organization

We use some of the combinatorial methods from Chapter 2 to obtain a slightly improved approximation guarantee of $\frac{3}{8}+\epsilon$ for the 3D folding problem, where ϵ is a small positive constant. First, we present a new 3D folding algorithm in Section 5.2. Our algorithm produces a folding with $\frac{3}{8}OPT - \Theta(1)$ contacts, improving the absolute approximation guarantee of $\frac{3}{8} - O(\frac{1}{\sqrt{OPT}})$ for Hart and Istrail's algorithm. In Section 5.3, we show that if the input string is of a certain special form, we can modify our algorithm to yield $\frac{3}{4}OPT - O(\delta(S))$ contacts, where $\delta(S)$ is the number of transitions in the input string S from sequences of 1's in odd positions in the string to sequences of 1's in even positions. In Section 5.3.2, we reduce the general 3D folding problem to the special case above, yielding a folding algorithm producing $.439 \cdot OPT - O(\delta(S))$ contacts. This reduction is based on a simple combinatorial theorem about binary strings discussed in Section 2.2. In Section 5.4, we present a different combinatorial algorithm that achieves $.375 \cdot OPT + \Omega(\delta(S))$ contacts. Finally, we combine these two algorithms removing the dependence on $\delta(S)$ in the approximation guarantee and obtain an algorithm with a slightly improved approximation guarantee of .37501 for the 3D folding problem.

5.2 A Diagonal Folding Algorithm

We will use the same notation as we did for the 2D folding problem (defined in Section 4.2). For the 3D problem, Hart and Istrail used a simple combinatorial upper bound on the optimal number of contacts possible in any folding [HI96]. The 3D square lattice is a bipartite graph and a string is a bipartite graph. Therefore, when placed on the lattice, each vertex with an even index in the string can only be adjacent to a vertex with an odd index in the string and vice versa. We refer to vertices with even (odd) indices labeled 1 as *even-1's* (*odd-1's*).

Let $S \in \{0,1\}^*$ be an input string for the folding problem. Let $\mathcal{E}[S]$ denote the number of even-1's in S and let $\mathcal{O}[S]$ denote the number of odd-1's in S. Even-1's can only have contacts with odd-1's and vice versa. In any folding of an input string, each vertex (except for the two endpoints) has two vertices that are adjacent to itself on the string and on the lattice. Since each lattice point has six neighbors, each vertex can have at most four contacts. Let $M_3[S]$ denote the maximum number of contacts possible for a string S. The maximum possible number of contacts in any folding of S is:

$$M_3[S] \leq 2 \cdot \min\{\mathcal{E}[S], \mathcal{O}[S]\} + 4. \tag{5.1}$$

We now present an algorithm that produces a folding with at least $\frac{3}{8}OPT - \Theta(1)$ contacts in the worst case, thereby improving the *absolute* approximation guarantee

of the algorithm of Hart and Istrail [HI96] (see Section 4.2). Our algorithm is based on *diagonal folds*. The algorithm guarantees that contacts form on and between two adjacent 2D planes. Each point in the 3D lattice has an (x, y, z)-coordinate, where x, y, and z are integers. We will fold the string so that all contacts occur on or between the planes $z = 0$ and $z = 1$. The DIAGONAL FOLDING ALGORITHM is described below and illustrated in Figure 5-1.

DIAGONAL FOLDING ALGORITHM

Input: a binary string S.

Output: a folding of the string S.

1. Let $k = \min\{\mathcal{O}[S], \mathcal{E}[S]\}$.
2. Divide S into two strings such that $S_{\mathcal{O}}$ contains at least half the odd-1's and $S_{\mathcal{E}}$ contains at least half the even-1's. We can do this by finding a point on the string such that half of the odd-1's are on one side of this point and half the odd-1's are on the other side. One of these sides contains at least half of the even-1's. We call this side $S_{\mathcal{E}}$ and the remaining side $S_{\mathcal{O}}$. Then we replace all the even-1's in $S_{\mathcal{O}}$ with 0's and replace all the odd-1's in $S_{\mathcal{E}}$ with 0's.
3. Place the first odd-1 in $S_{\mathcal{O}}$ on lattice point $(1, 1, 1)$ and the next odd-1 in $S_{\mathcal{O}}$ on lattice point $(2, 2, 1)$ and so on. For the first $\frac{k}{4}$ of the odd-1's in $S_{\mathcal{O}}$, place the i^{th} odd-1 on lattice point $(i, i, 1)$. Then place the $(k/4 + 1)$ odd-1 on lattice point $(k/4-1, k/4+1, 1)$. For the first $\frac{k}{4} - 1$ of the even-1's in $S_{\mathcal{E}}$, place the i^{th} even-1 on lattice point $(i, i+1, 1)$. Use the dimensions $z \geq 1$ to place the strings of 0's between consecutive odd-1's in $S_{\mathcal{O}}$ and the strings of 0's between consecutive even-1's in $S_{\mathcal{E}}$.
4. Place the $(k/4 + 2)$ odd-1 in $S_{\mathcal{O}}$ on lattice point $(k/4 - 2, k/4 + 1, 0)$. Then place the $(k/4 + i)$ odd-1 in $S_{\mathcal{O}}$ on lattice point $(k/4 - i + 1, k/4 - i + 2, 0)$. Place the $(k/4)$ even-1 in $S_{\mathcal{E}}$ on lattice point $(k/4 - 1, k/4 - 1, 0)$. Place the $(k/4 + i)$ even-1 in $S_{\mathcal{E}}$ on lattice point $(k/4-i-1, k/4-i-1, 0)$. Use the dimensions $z \leq 0$ to place the strings of 0's between consecutive 1's in $S_{\mathcal{O}}$ or $S_{\mathcal{E}}$.

Lemma 19. *The* DIAGONAL FOLDING ALGORITHM *produces a folding with at least* $\frac{3}{8}OPT - O(1)$ *contacts.*

Proof. Without loss of generality, we assume that $k = \mathcal{O}[S]$. Consider the i^{th} odd-1 from the first half of $S_{\mathcal{O}}$. It is placed on lattice point $(i, i, 1)$. In Step 2, this odd-1 forms contacts with the even-1's on the lattice points $(i, i+1, 1)$ and $(i-1, i, 1)$. In Step 3, it forms a contact with the lattice point $(i, i, 0)$. Thus, each odd-1 from the first half of $S_{\mathcal{O}}$ has three contacts. Now consider an odd-1 with an index $k/4+i$, where i ranges from 3 and $\frac{k}{4}$. Each such odd-1 is placed on lattice point $(k/4-i+1, k/4-i+2, 0)$. In Step 3, it forms contacts with even-1's on the lattice points $(k/4-i+1, k/4-i+1, 0)$ and $(k/4-i+2, k/4-i+2, 0)$. In Step 2, it forms a contact with the even-1 on lattice

Figure 5-2: In this string, there are two switches.

point $(k/4 - i + 1, k/4 + i + 2, 1)$. Thus, it also has 3 contacts. By (5.1), we see that an upper bound on the number of contacts is $OPT \leq 4\mathcal{O}[S] = 4k + 2$. We obtain 3 contacts for $\frac{k}{2} - 3$ of the odd-1's. Thus, the number of contacts in the resulting folding is at least $\frac{3}{8}OPT - 9$. □

5.3 Improved Diagonal Folding Algorithms

As the number of 1's placed on the diagonal in the DIAGONAL FOLDING ALGORITHM increases, the length of the diagonal of the resulting folding (i.e. the length equals $\frac{1}{2}\min\{\mathcal{O}[S], \mathcal{E}[S]\}$) increases in a direction parallel to the line $x = y$. The height of the folding may also increase depending on the maximum distance between consecutive odd-1's in $S_{\mathcal{O}}$ or consecutive even-1's in $S_{\mathcal{E}}$. However, regardless of the input string, the resulting folding has the same constant width in the direction parallel to the line $x = -y$. In other words, the resulting folding can be enclosed in a box of infinite height and depth and constant width. Therefore, this third direction is relatively unused and leaves room which we take advantage of in our improved folding algorithms.

We will take advantage of this unused space by modifying the DIAGONAL FOLDING ALGORITHM. We say a *switch* is a transition from a sequence of consecutive odd-1's to a sequence of consecutive even-1's. For example, for the string $S = 100\underline{1}0001010110\underline{1}101011$, $\delta(S) = 2$ since there are two transitions (underlined) from a maximal sequence of consecutive odd-1's to a sequence of even-1's. We denote the number of switches in a string S by $\delta(S)$.

In this section, we present two algorithms with the following approximation guarantees:

(i) $.375OPT + \frac{\delta(S)}{256}$,

(ii) $.439OPT - 16\delta(S)$.

The first approximation guarantee is preferred when there are many switches in the string S, i.e. the value of $\delta(S)$ is large. And the second approximation guarantee is preferred when there are few switches in the string S, i.e. the value of $\delta(S)$ is small. These two algorithms lead to an approximation guarantee that is independent of $\delta(S)$. For any input string S, we output the folding from the algorithm that results in the most contacts for that string. The output guarantee of this combination of the two algorithms is lowest if the two approximation guarantees are equal, i.e. $\frac{3}{8}OPT + \frac{\delta(S)}{256} = .439OPT - 16\delta(S)$, which occurs when $\delta(S) = .0039990237OPT$, yielding an approximation guarantee of at least .37501562.

Theorem 20. *There is a linear time algorithm for the 3D folding problem that outputs a folding with at least* $.37501 \cdot OPT - O(1)$ *contacts for any input string* S.

In Section 5.3.1, we give an algorithm for the 3D string folding problem that has an approximation guarantee of $\frac{3}{4}OPT - 16\delta(S) - O(1)$ for a special class of strings. In Section 5.3.2, we show how to apply this algorithm to any string to obtain a factor $.439 - 16\delta(S)$ approximation algorithm. To apply this algorithm, we show that given any input string S, we can find a subsequence that belongs to the special class of strings. This subsequence will contain at least a .5857-fraction of the vertices in the original input string. Thus, we obtain an algorithm with an approximation guarantee of $\frac{3}{4} \cdot (.5857)OPT - 16\delta(S) - O(1) = .439OPT - 16\delta(S) - O(1)$. Finally, in Section 5.4, we give an algorithm for the 3D string folding problem with an approximation guarantee of $\frac{3}{8}OPT + \frac{\delta(S)}{256}$.

5.3.1 An Algorithm for a Special Class of Strings

We will now give an algorithm that has an approximation guarantee of $\frac{3}{4}OPT - 16\delta(S) - O(1)$ for a special class of strings. First we will describe this class of strings.

We define *consecutive odd-1's* (*consecutive even-1's*) to be odd-1's (even-1's) that are not separated by even-1's (odd-1's). For example, in the string 101000110001101, there is a sequence of three consecutive odd-1's followed by two consecutive even-1's followed by two consecutive odd-1's. A string S belongs to the special class of strings if it can be divided into two substrings, $S_{\mathcal{O}}$ and $S_{\mathcal{E}}$ such that $S_{\mathcal{O}}$ is *odd-monotone* and $S_{\mathcal{E}}$ is *even-monotone.*

Definition 21. *A string* $S_{\mathcal{O}}$ *is called* odd-monotone *(*even-monotone*) if every maximal sequence of consecutive even-1's (odd-1's) is immediately preceded by at least as many consecutive odd-1's (even-1's).*

For example, the string 10101100011 is odd-monotone and the string 010001010110 1101011 is even-monotone.

Theorem 22. *Let* $S = S_{\mathcal{O}}S_{\mathcal{E}}$ *such that* $S_{\mathcal{O}}$ *is an odd-monotone string and* $S_{\mathcal{E}}$ *is an even-monotone string and* $\mathcal{O}[S_{\mathcal{O}}] = \mathcal{E}[S_{\mathcal{E}}]$ *and* $\mathcal{E}[S_{\mathcal{O}}] = \mathcal{O}[S_{\mathcal{E}}]$. *Then there is a linear time algorithm that folds the string* S *achieving* $\frac{3}{4}OPT - 16\delta(S) - O(1)$ *contacts.*

The main idea behind the algorithm referred to in Theorem 22 is to partition the elements in $S_{\mathcal{O}}$ and $S_{\mathcal{E}}$ into *main-diagonal elements* and *off-diagonal elements.* We then use the DIAGONAL FOLDING ALGORITHM to fold the main-diagonal elements along the direction $x = y$ and the off-diagonal elements into branches along the direction $x = -y$. All 1's will make three contacts except for a constant number of 1's—for each switch in the strings $S_{\mathcal{O}}$ and $S_{\mathcal{E}}$—sacrificed to align the off-diagonal branches and a constant number of 1's for each repetition of the DIAGONAL FOLDING ALGORITHM. This yields the claimed number of $\frac{3}{4}OPT - O(\delta(S)) - O(1)$ contacts.

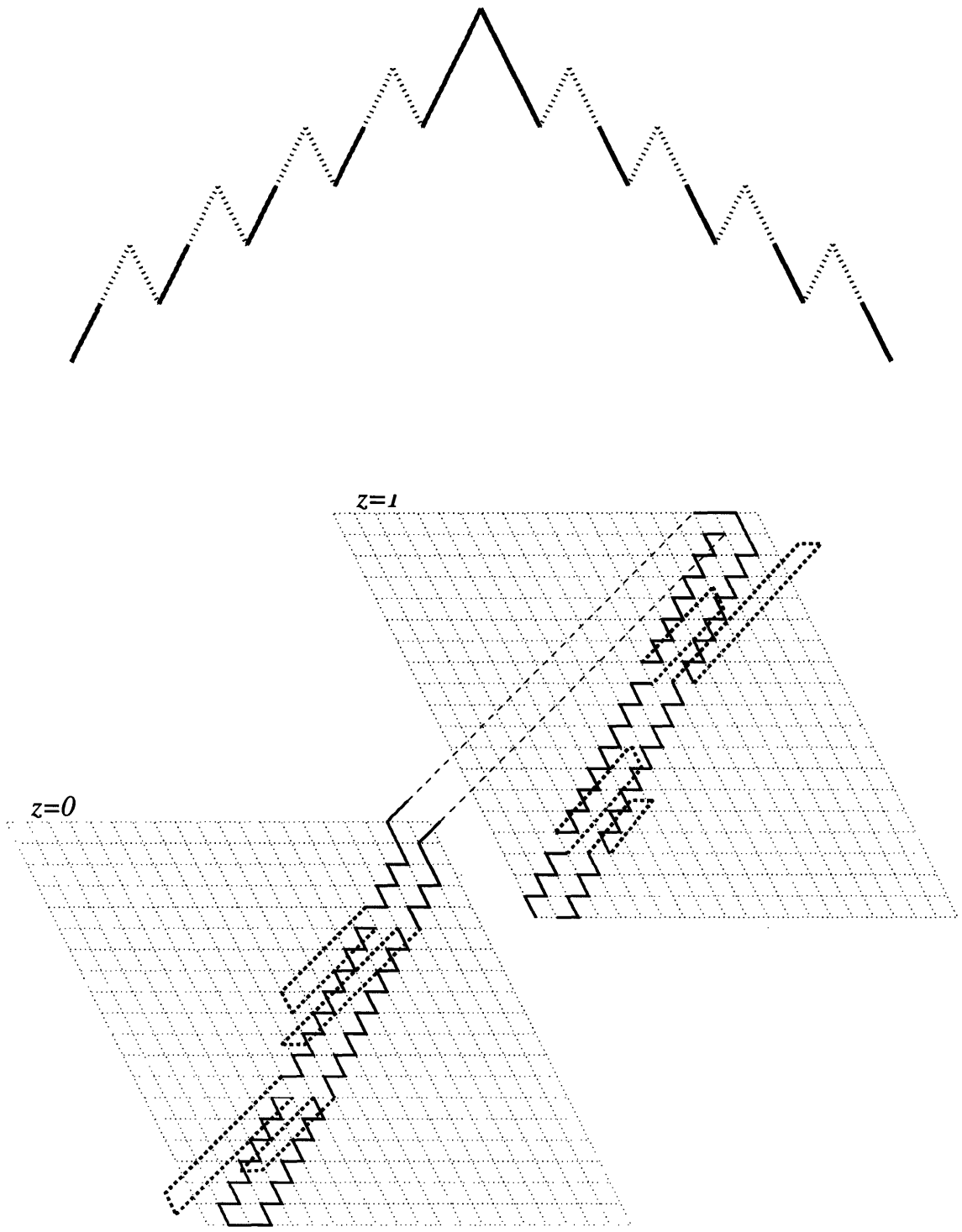

Figure 5-3: . If the strings $S_{\mathcal{O}}$ and $S_{\mathcal{E}}$ are odd-monotone and even-monotone, respectively, then we can divide the vertices in $S_{\mathcal{O}}$ and $S_{\mathcal{E}}$ into main-diagonal elements and off-diagonal elements. The main-diagonal elements are denoted by solid lines and the off-diagonal elements are denoted by dotted lines.

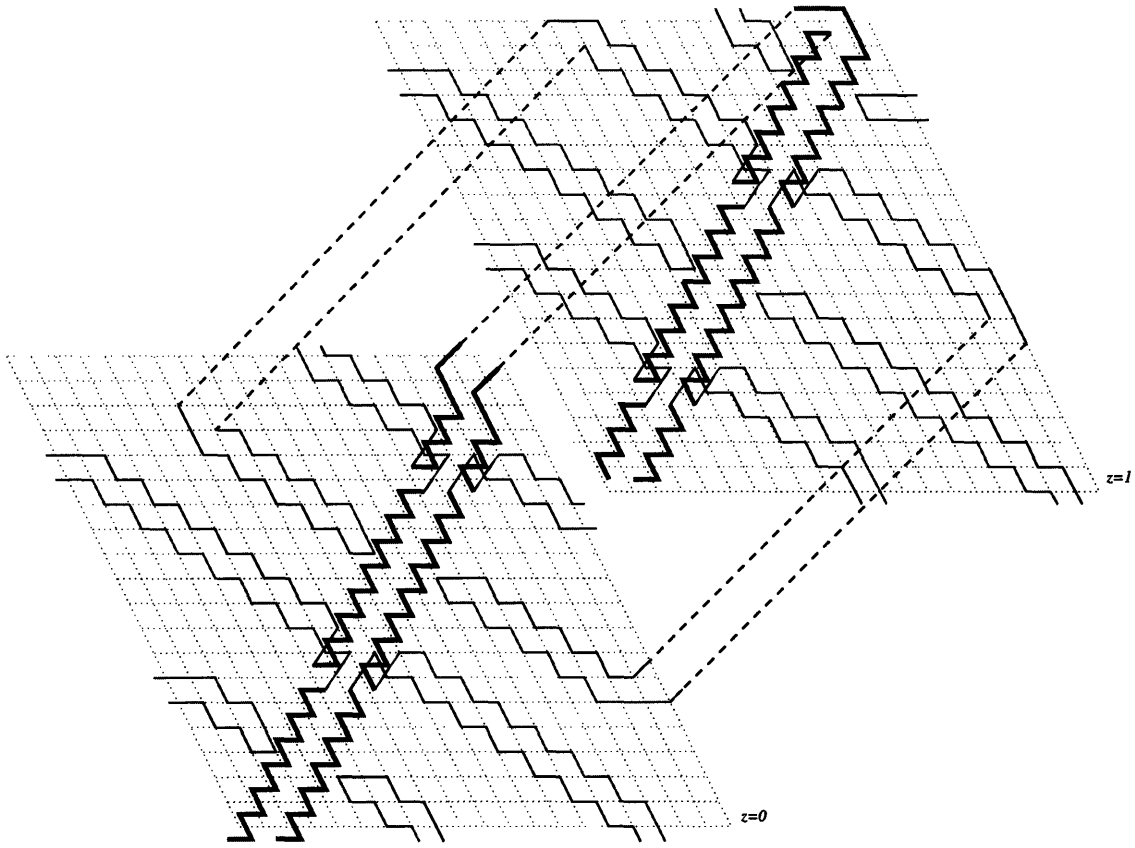

Figure 5-4: The DIAGONAL FOLDING ALGORITHM is used on the main-diagonal elements (bold) and the off-diagonal elements.

To precisely define *main-diagonal* and *off-diagonal* elements, we use additional notation. We use 0^k and 1^k (for some integer $k \geq 0$) to refer to the strings consisting of k 0's and k 1's, respectively. By writing $S = \mathcal{E}^k$ for some integer k, we mean that S is of the form $S = 0^{2i_0+1}10^{2i_1+1}10^{2i_2+1}10^{2i_3+1}\ldots 0^{2i_{k-1}+1}10^{i_k}$ for integers $i_j \geq 0$, and all the 1's in S are even-1's. Likewise, we write $S = \mathcal{O}^k$ to refer to a string of the same form where all 1's are odd-1's, i.e. $S = 10^{2i_1+1}10^{2i_2+1}\ 10^{2i_3+1}\ldots 0^{2i_{k-1}+1}10^{i_k}$. So we can express any string $S_\mathcal{E}$ as $S_\mathcal{E} = \mathcal{E}^{a_1}\mathcal{O}^{b_1}\mathcal{E}^{a_2}\mathcal{O}^{b_2}\ldots\mathcal{E}^{a_k}\mathcal{O}^{b_k}$ for $k = \delta(S_\mathcal{E})$ and integers a_i and b_i. If $S_\mathcal{E}$ is even-monotone, then $a_i \geq b_i$ for all i. We can express any string $S_\mathcal{O}$ as $S_\mathcal{O} = \mathcal{O}^{c_1}\mathcal{E}^{d_1}\mathcal{O}^{c_2}\mathcal{E}^{d_2}\ldots\mathcal{O}^{c_\ell}\mathcal{E}^{d_\ell}$ for $\ell = \delta(S_\mathcal{O})$ and integers c_i and d_i. If $S_\mathcal{O}$ is even-monotone, then $c_i \geq d_i$ for all i.

Definition 23. *For an odd-monotone string $S_\mathcal{O} = \mathcal{O}^{c_1}\mathcal{E}^{d_1}\mathcal{O}^{c_2}\mathcal{E}^{d_2}\ldots\mathcal{O}^{c_\ell}\mathcal{E}^{d_\ell}$, the first set of $c_i - d_i$ odd-1's in each block, i.e. the elements $\mathcal{O}^{c_1-d_1}\mathcal{O}^{c_2-d_2}\ldots\mathcal{O}^{c_\ell-d_\ell}$, are the* main-diagonal elements *and the remaining elements $\mathcal{O}^{d_1}\mathcal{E}^{d_1}\mathcal{O}^{d_2}\mathcal{E}^{d_2}\ldots\ \mathcal{O}^{d_\ell}\mathcal{E}^{d_\ell}$ are the* off-diagonal elements *in $S_\mathcal{O}$.*

For even-monotone strings, we define main-diagonal and off-diagonal elements analogously. In our modified algorithm, it will be useful to have $S_\mathcal{E}$ and $S_\mathcal{O}$ in a

special form. Two sets of off-diagonal elements in $S_{\mathcal{O}}$, $\mathcal{O}^{d_i}\mathcal{E}^{d_i}$ and $\mathcal{O}^{d_{i+1}}\mathcal{E}^{d_{i+1}}$, are separated by $c_{i+1} - d_{i+1}$ odd-1's that are main-diagonal elements. We want them to be separated by a number of main-diagonal elements that is a multiple of eight. This will guarantee that the off-diagonals used to fold the off-diagonal elements are regularly spaced so that none of the off-diagonal folds interfere with each other. We will use the following simple lemma.

Lemma 24. *For any odd-monotone string $S_{\mathcal{O}}$ it is possible to change at most $8\delta(S_{\mathcal{O}})$ 1's to 0's so that the resulting string S' is of the form $S' = \mathcal{O}^{a_1}\mathcal{E}^{b_1}\mathcal{O}^{a_2}\mathcal{E}^{b_2}\ldots\mathcal{O}^{a_k}$, where $a_i - b_i$ is a positive multiple of eight for $1 \leq i < k$.*

Proof. Suppose that $S_{\mathcal{O}}$ initially is of the form

$$S_{\mathcal{O}} = \mathcal{O}^{\alpha_1}\mathcal{E}^{\beta_1}\mathcal{O}^{\alpha_2}\mathcal{E}^{\beta_2}\ldots\mathcal{O}^{\alpha_\ell}.$$

First, we convert all $\mathcal{E}^{\beta_i}$ with $\beta_i \leq 8$ into 0's. This will merge some maximal sequences of odd-1's, yielding a string of the form

$$\mathcal{O}^{a_1}\mathcal{E}^{\gamma_1}\mathcal{O}^{a_2}\mathcal{E}^{\gamma_2}\ldots\mathcal{O}^{a_k}$$

with $k \leq \ell$. For each i, we then convert $(\gamma_i - a_i) \bmod 8$ even-1's of $\mathcal{E}^{\gamma_i}$ into 0's, yielding a string of the desired form. □

We note that there is an analogous version of Lemma 24 for even-monotone strings. Additionally, we want the number of even-1's deleted from $S_{\mathcal{O}}$ to equal the number of odd-1's deleted from $S_{\mathcal{E}}$. In other words, we want the number of main-diagonal elements in each of the strings to be equal. Without loss of generality, assume that number of even-1's deleted (i.e. changed to 0's) in $S_{\mathcal{O}}$ is greater than the number of odd-1's deleted in $S_{\mathcal{E}}$. Then, starting from the end of $S_{\mathcal{E}}$, we simply delete consecutive odd-1's until the number of deleted elements in each string is equal. With this preparation, we can now state our folding algorithm.

Proof of Theorem 22: By the correctness of the DIAGONAL FOLDING ALGORITHM, it suffices to consider whether some off-diagonals intersect each other. The first step of the algorithm ensures that all off-diagonal branches are spread apart by multiples of eight on the main-diagonal. Thus, neighboring branches do not intersect. Furthermore, branches off the upper ($z = 1$) plane do not intersect with branches off the lower ($z = 0$) place due to Step 4. Changing the plane when the main diagonal has a length $\equiv 2 \mod 4$ ensures that branches on the upper plane will follow diagonals $x = -y + 8k$ for some k, and branches on the lower plane follow diagonals $x = -y + 8k + 4$ for some k. Thus, branches are at least four lattice points apart, showing that the folding is non-intersecting.

It remains to analyze the number of contacts produced by the folding. The DIAGONAL FOLDING ALGORITHM produces three contacts for almost every 1 in the string

OFF-DIAGONAL FOLDING ALGORITHM

Input: A binary string $S = S_{\mathcal{O}}S_{\mathcal{E}}$, such that $S_{\mathcal{O}}$ is odd-monotone, $S_{\mathcal{E}}$ is even-monotone, $\mathcal{O}[S_{\mathcal{O}}] = \mathcal{E}[S_{\mathcal{E}}]$ and $\mathcal{E}[S_{\mathcal{O}}] = \mathcal{O}[S_{\mathcal{E}}]$.

Output: A folding of the string S.

1. Change at most $8\delta(S)$ 1's to 0's in $S_{\mathcal{O}}$ and $S_{\mathcal{E}}$ to yield the form specified in
2. Change at most $8\delta(S)$ 1's to 0's in $S_{\mathcal{O}}$ and $S_{\mathcal{E}}$ so that for each maximal block of odd-1's (even-1's) and following maximal block of even-1's (odd-1's) in $S_{\mathcal{O}}$ ($S_{\mathcal{E}}$), the number of odd-1's (even-1's) and even-1's (odd-1's) differ by a multiple of eight (see Lemma 24).
3. Run DIAGONAL FOLDING ALGORITHM on *main-diagonal* elements along the direction $x = y$ and change from plane $z = 0$ to $z = 1$ when the length of the main diagonal equals $4 \cdot \lfloor \mathcal{O}[S_{\mathcal{O}}]/8 \rfloor + 2$.
4. Run DIAGONAL FOLDING ALGORITHM on the *off-diagonal* elements along the direction $x = -y$. The *off-diagonal* elements attached to the *main-diagonal* elements on the plane $z = 1$ are folded along the diagonals $x = -y + 8k$. The *off-diagonal* elements attached to the *main-diagonal* elements on the plane $z = 0$ are folded along the diagonals $x = -y + 8k + 4$. (See Figure 5-5.)

S. So it suffices to bound the number of 1's in S that do not receive three contacts. The following is an exhaustive list: (i) the up to $8\delta(S)$ 1's changed into 0's in Step 2; (ii) a constant number of 1's at the ends of the main-diagonal (see Lemma 19) and because we fold over at a length $\equiv 2 \mod 4$ in Step 3; (iii) in Step 4, for each of the at most $\delta(S)$ off-diagonal branches: at most three 1's at the end of each branch (by Lemma 19), and at most five 1's to connect the off-diagonal branch to the main-diagonal (see Figure 5-5). So in summary, up to $16\delta(S) + O(1)$ 1's might not receive three contacts, so that we obtain $3\mathcal{O}[S] - 16\delta(S) - O(1) \geq \frac{3}{4}OPT - 16\delta(S) - O(1)$ contacts. □

5.3.2 Relating Folding to String Properties

We will now show how to apply the algorithm for the special class of strings from the previous section to the general string folding problem. The main idea is given a string, find a long subsequence that has the special form required by the OFF-DIAGONAL FOLDING ALGORITHM. If we can keep most of the elements, then using this algorithm, we would get very close to the performance guarantee of $\frac{3}{4}OPT - 16\delta(S) - O(1)$ contacts.

Thus, the combinatorial problem that we want to solve is the following: given a string $S \in \{0,1\}^*$ such that $\mathcal{E}[S] = \mathcal{O}[S]$, we want to divide the string into two sub-

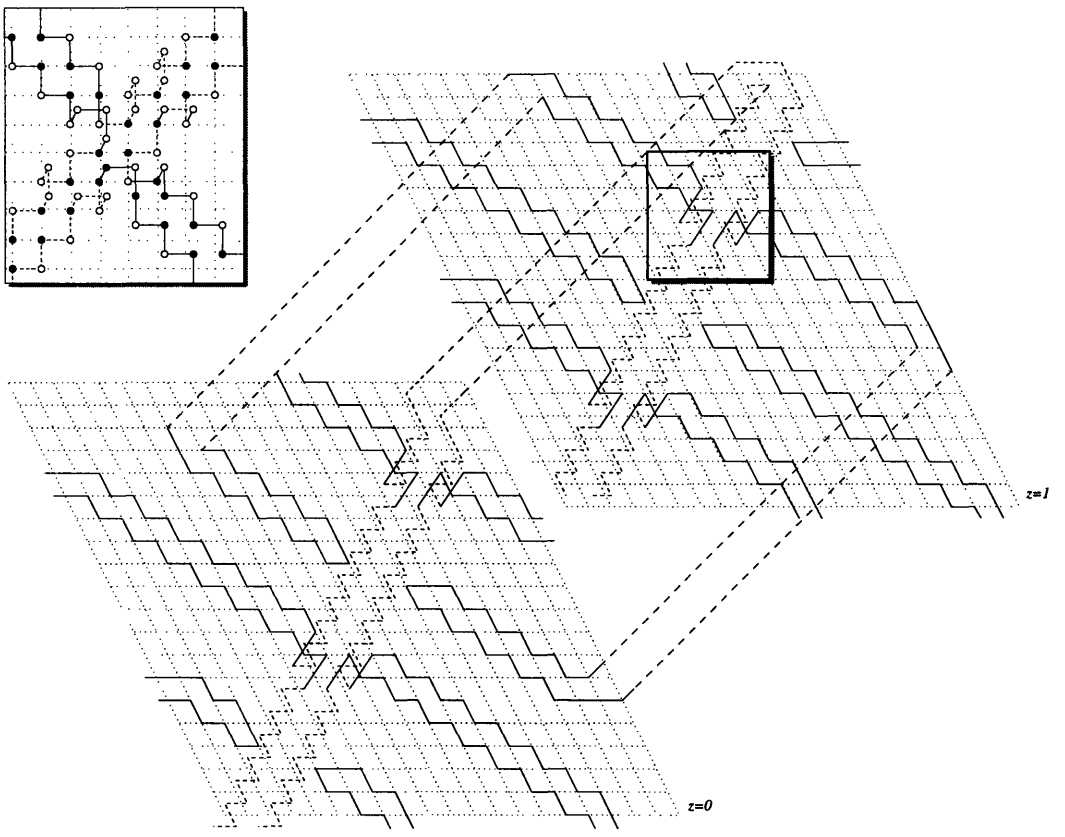

Figure 5-5: Folding the *off-diagonal* elements in Step 4 of the OFF-DIAGONAL FOLDING ALGORITHM. The *main-diagonal* elements are represented by the dashed lines on the main diagonal. The *off-diagonal* elements are represents by the solid lines on the off-diagonals. This figure shows how the repetitions of the DIAGONAL FOLDING ALGORITHM on the off-diagonals interleave and thus so not interfere with each other. The closeup gives an example of how the off-diagonal folds are connected to the main diagonal.

strings such that one contains an even-monotone subsequence and the other contains an odd-monotone subsequence and the number of 1's contained in these monotone subsequences is as large as possible, since the 1's in these subsequences are the 1's that will have contacts in the OFF-DIAGONAL FOLDING ALGORITHM.

Given a string $S \in \{0,1\}^*$, we will treat it as a loop $L(S)$ by attaching its endpoints. In other words, we are only going to consider foldings of the string that place the first and last element of S on adjacent lattice points. (If S has odd length, we can add a 0 to the end of the string and fold this string instead of S; a folding of this augmented string will yield a valid folding of the original string.)

Lemma 25. *Let $L(S) \in \{0,1\}^*$ be a loop, and $k = \min\{\mathcal{O}[S], \mathcal{E}[S]\}$. Then it is possible to change some 1's in $L(S)$ to 0's so that there is a partition $L(S) = S_{\mathcal{O}}S_{\mathcal{E}}$*

with $S_{\mathcal{O}}$ and $S_{\mathcal{E}}$ odd- and even-monotone, respectively, $\mathcal{O}[S_{\mathcal{O}}] = \mathcal{E}[S_{\mathcal{E}}]$, $\mathcal{E}[S_{\mathcal{O}}] = \mathcal{O}[S_{\mathcal{E}}]$, and $\mathcal{O}[S_{\mathcal{O}}] + \mathcal{O}[S_{\mathcal{E}}] \geq (2 - \sqrt{2})k$. Furthermore, this partition can be constructed in linear time.

Proof. We first apply Lemma 2 from Chapter 2 to cut the string into two substrings. This lemma states that we can always find a vertex $p \in L(S)$ such that as we start in that position and go clockwise, we encounter at least as many odd-1's as even-1's and as we go in the counter-clockwise direction, we encounter at least as many even-1's as odd-1's. We cut the loop $L(S)$ at such a point p and then make another cut in $L(S)$ at a point such that the number of 1's in each of the two resulting strings is equal. We refer to these strings as $S_{\mathcal{O}}$ and $S_{\mathcal{E}}$, respectively. Note that in these two strings, the number of odd-1's in one string equals the number of even-1's in the other and vice versa, i.e. $\mathcal{O}[S_{\mathcal{O}}] = \mathcal{E}[S_{\mathcal{E}}]$ and $\mathcal{E}[S_{\mathcal{O}}] = \mathcal{O}[S_{\mathcal{E}}]$.

Now we have two substrings $S_{\mathcal{O}}$ and $S_{\mathcal{E}}$. The substring $S_{\mathcal{O}}$ has the property that every suffix (or prefix–depending on how you view the string) has at least as many odd-1's as even-1's and $S_{\mathcal{E}}$ has the property that every suffix has at least as many even-1's as odd-1's.

We want to change the minimum number of 1's to 0's in $S_{\mathcal{O}}$ and $S_{\mathcal{E}}$ so that the resulting substrings are odd-monotone and even-monotone, respectively, and $\mathcal{O}[S_{\mathcal{O}}] = \mathcal{E}[S_{\mathcal{E}}]$ and $\mathcal{E}[S_{\mathcal{O}}] = \mathcal{O}[S_{\mathcal{E}}]$, since these are the conditions required by Theorem 22. Consider a binary string $S'_{\mathcal{E}}$ corresponding to the subsequence of 1's in $S_{\mathcal{E}}$ in which each odd-1 is replaced by an a and each even-1 is replaced by a b. The problem of changing the minimum number of 1's to 0's in $S_{\mathcal{E}}$ so that the resulting string is odd-monotone is equivalent to finding the longest *block-monotone* subsequence in the string $S'_{\mathcal{E}}$. A subsequence is *block-monotone* if every block of a's is immediately followed by a block of at least as many b's. (For the string $S_{\mathcal{O}}$, we have the same problem stated with a's and b's inverted: we want to find the longest subsequence in which every block of b's is immediately followed by a block of at least as many a's.) By Lemma 6, we can furthermore choose these subsequences such that $\mathcal{O}[S'_{\mathcal{O}}] = \mathcal{E}[S'_{\mathcal{E}}]$ and $\mathcal{E}[S'_{\mathcal{O}}] = \mathcal{O}[S'_{\mathcal{E}}]$ after the transformation. □

Lemma 25 implies that every 3D folding instance can be converted into the case required by Theorem 22 by converting not too many 1's into 0's. We obtain the following corollary of Lemma 25 and Theorem 22.

Corollary 26. *There is a linear time algorithm for the 3D folding problem that generates at least $.439 \cdot OPT - 16\delta(S) - O(1)$ contacts.*

Proof. Given an input string S, first obtain $S_{\mathcal{O}}$ and $S_{\mathcal{E}}$ with Lemma 25. Note that the number of switches does not increase from S to $S_{\mathcal{O}}S_{\mathcal{E}}$. Since the number of 1's is reduced by a factor of $(2 - \sqrt{2})$, the optimal number of contacts might also have been decreased by that factor. Applying Theorem 22 to $S_{\mathcal{O}}$ and $S_{\mathcal{E}}$ therefore leads to a folding with at least $\frac{3}{4}(2 - \sqrt{2})OPT - 16\delta(S) - O(1) > .439 \cdot OPT - 16\delta(S) - O(1)$ contacts. □

5.4 Another 3D String Folding Algorithm

In this section, we give a case-based algorithm that has an approximation guarantee of $.375 + O(\delta(S))$. Consider the substrings $S_{\mathcal{O}}$ and $S_{\mathcal{E}}$ of the loop $L(S)$ such that $\mathcal{O}[S_{\mathcal{O}}] = \mathcal{E}[S_{\mathcal{E}}]$ and $\mathcal{E}[S_{\mathcal{O}}] = \mathcal{O}[S_{\mathcal{E}}]$. We can do this by cutting the loop $L(S)$ at the point p chosen according to Lemma 2 in Chapter 2 and cutting $L(S)$ at a point so that the two resulting substrings have an equal number of 1's. Furthermore, in this section, we can assume that $\mathcal{O}[S_{\mathcal{O}}] = \mathcal{E}[S_{\mathcal{O}}]$. If we have $\mathcal{O}[S_{\mathcal{O}}] > \mathcal{E}[S_{\mathcal{O}}]$, then the algorithms we describe below will have strictly better approximation ratios than what we prove.

We will consider the following modified version of the string $S_{\mathcal{O}}$. For every sequence of consecutive even-1's, we turn all but one of them into a 0. For example, we would transform the string 1101011 into 1100001. Abusing notation, we will from now on refer to this modified string as $S_{\mathcal{O}}$. We will divide the even-1's in $S_{\mathcal{O}}$ into the following disjoint categories. Suppose each of these categories has $\delta_1 k, \delta_2 k, \delta_3 k$, and $\delta_4 k$ even-1's respectively, where $k = \mathcal{O}[S]$. Without loss of generality, we assume that $\delta_1 + \delta_2 + \delta_3 + \delta_4 \geq \delta/2$, i.e. half the switches occur in $S_{\mathcal{O}}$.

Each even-1 in $S_{\mathcal{O}}$ falls in exactly one of the following categories:

1. Even-1's in blocks of 1's of length at least ten or in a block of 1's of length nine that begins with an odd-1.
2. Even-1's in blocks of 1's of length at least two and at most nine that begin or end with an even-1.
3. Even-1's in blocks of length one.
4. Even-1's in blocks of length at least three and at most seven that begin and end with an odd-1.

For each of the four cases above, we will show how to slightly modify the DIAGONAL FOLDING ALGORITHM so that it gives an approximation guarantee of $\frac{3}{8} + c_i\delta_i$ for some constant c_i. In the DIAGONAL FOLDING ALGORITHM, one way to account for contacts is to attribute $\frac{3}{2}$ of a contact to each odd-1 on the main diagonal and $\frac{3}{2}$ of a contact to each even-1 on the main diagonal. The main idea behind the modifications of the algorithm is to fold the string so that some odd-1's may no longer be on the main diagonal (thus losing $\frac{3}{2}$ contacts per odd-1) but form more than $\frac{3}{2}$ contacts per odd-1 with neighboring even-1's (making use of the switches). In some of the modifications (such as Case 2) we do not actually remove any of the odd-1's from the main diagonal; due to the nature of the switches, we can still get $O(1)$ contacts per switch. We will first prove a lemma that we will use in several of the cases.

Lemma 27. *Suppose we delete (i.e. change 1's to 0's) i odd-1's in $S_{\mathcal{O}}$. Then we can re-divide S into substrings $S_{\mathcal{O}}$ and $S_{\mathcal{E}}$ so that we again have $\mathcal{E}[S_{\mathcal{E}}] = \mathcal{O}[S_{\mathcal{O}}]$. If we run the* DIAGONAL FOLDING ALGORITHM *on these new strings $S_{\mathcal{E}}$ and $S_{\mathcal{O}}$, we will obtain a folding with at least $\frac{3}{2}(\mathcal{O}[S] - i)$ contacts on the main diagonal.*

Proof. We use Lemma 2 from Chapter 2 to choose s_1 so that $\mathcal{O}[S_i] \geq \mathcal{E}[S_i]$ for all $i = 1, \ldots n$, where $S_i = s_1 \ldots s_i$. If we define $\widetilde{S}_i := s_n s_{n-1} \ldots s_i$, then again by Lemma 2 we have $\mathcal{E}[\widetilde{S}_i] \geq \mathcal{O}[\widetilde{S}_i]$ for all $i = 1, \ldots n$.

If we remove i odd-1's from $S_{\mathcal{O}}$, then the main diagonal fold of $S_{\mathcal{O}}$ would be much shorter than that of $S_{\mathcal{E}}$. However, if we move $s_i = s_{i+j}$ for some j so that once again $\mathcal{O}[S_{\mathcal{O}}] = \mathcal{E}[S_{\mathcal{E}}]$, then the number of odd-1's in $S_{\mathcal{O}}$ is at least $\frac{\mathcal{O}[S]-i}{2}$. Thus, we obtain at least $\frac{3}{2}(\mathcal{O}[S] - i)$ contacts on the main diagonal. □

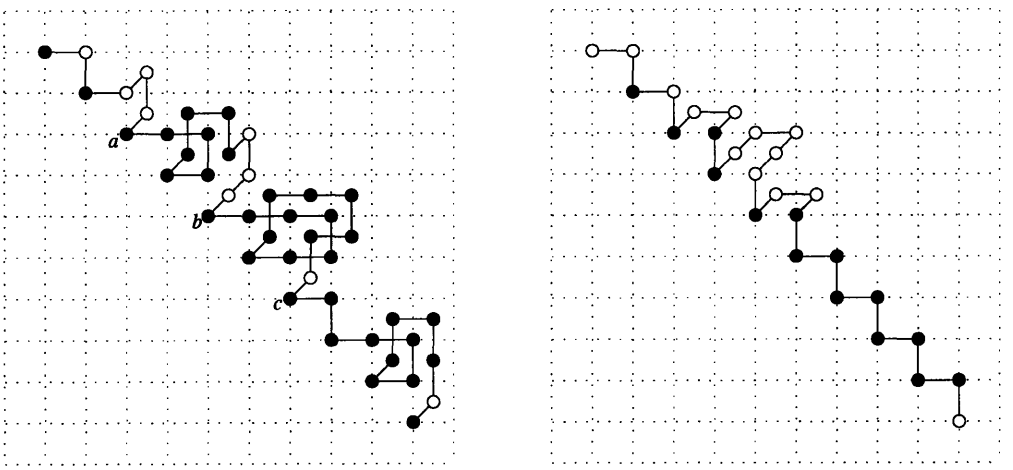

Figure 5-6: Cases 1 and 2. The first figure shows a folding for even-1's in Case 1. At point a, the folding for a block of 1's of length nine that starts with an odd-1 begins. Note that three odd-1's are not placed on the main diagonal, but five contacts – in addition to those that will be formed on the main diagonal – are obtained. At point b, a block of 1's of length 13 is folded. Here, five odd-1's are not placed on the main diagonal, but eight additional contacts are formed off the main diagonal. At point c, a block of 1's of length 11 is folded. It is basically the same folding as used for blocks of length nine. The second figure shows even-1's in Case 2. For at least half of the blocks of 1's of length at least two and at most nine that begin or end with an even-1, we can get an extra contact by placing an even-1 adjacent to an odd-1 on the main diagonal.

Case 1

Lemma 28. *There is a modification of the* DIAGONAL FOLDING ALGORITHM *with approximation guarantee at least $\frac{3}{8} + \frac{\delta_1}{40}$.*

Proof. An even-1 in Case 1 occurs in a block of 1's of length at least ten or in a block of 1's of length nine beginning with an odd-1. Suppose we have a block of 11 1's that begins with an odd-1, which will give the worst case approximation ratio. Then we fold this block as in Figure 5-6 starting at the point labeled c. Note that three odd-1's from $S_{\mathcal{O}}$ that would be on the main diagonal in the DIAGONAL FOLDING ALGORITHM are not placed on the main diagonal. Thus, the main diagonal will be shorter – at least $\frac{3\delta_1 k}{5}$ shorter, because for every five even-1's in Case 1, we take at least three odd-1's off the main diagonal. By Lemma 27 we can then assume that the length of the main diagonal is:

$$\frac{1}{2}\left(\mathcal{O}[S] - \frac{3\delta_1 \mathcal{O}[S]}{5}\right).$$

For every odd-1 in $S_{\mathcal{O}}$ on the main diagonal, we obtain three contacts. For every three odd-1's in $S_{\mathcal{O}}$ off the diagonal (corresponding to five even-1's in Case 1), we obtain five contacts. Thus, the approximation guarantee is:

$$\left(\frac{3}{2}\left(\mathcal{O}[S] - \frac{3\delta_1 \mathcal{O}[S]}{5}\right) + \frac{5\delta_1 \mathcal{O}[S]}{5}\right)\frac{1}{4\mathcal{O}[S]} = \frac{3}{8} - \frac{9\delta_1}{40} + \frac{\delta_1}{4} = \frac{3}{8} + \frac{\delta_1}{40}.$$

□

Case 2

Lemma 29. *There is a modification of the* DIAGONAL FOLDING ALGORITHM *with approximation guarantee at least* $\frac{3}{8} + \frac{\delta_2}{32}$.

Proof. An even-1 in Case 2 is in a block of 1's of length at least two and at most nine that begins or ends with an even-1. In this case, the main diagonal will remain the same length as in the DIAGONAL FOLDING ALGORITHM. We will obtain extra contacts by placing even-1's from $S_{\mathcal{O}}$ next to odd-1's on the main diagonal. This is shown in Figure 5-6.

For at least half of the blocks (in $S_{\mathcal{O}}$) of 1's of length at least two and at most nine that begin or end with even-1's, we can get an extra contact by placing an even-1 adjacent to an odd-1 on the main diagonal. We may only be able to do this for half of the blocks, because the folding in Figure 5-6 will work only for an even-1 followed immediately by an odd-1 or an odd-1 followed immediately by an even-1, but does not allow alternating between these two cases. Among these types of blocks, the worst case is a block of eight 1's that begins or ends with an even-1. Such a block uses four even-1's from Case 2. If all the Case 2 even-1's fell in this category, we could get an extra contact for half of them, which is one per eight switches. This ratio is better for block lengths other than eight. In particular, note that a block of length nine that begins with an even-1 must also end with an even-1, so we always get a contact for one of the two ends of such a block. In summary, we get the following approximation

guarantee:

$$\left(\frac{3\mathcal{O}[S]}{2} + \frac{\delta_2 \mathcal{O}[S]}{8}\right) \frac{1}{4\mathcal{O}[S]} = \frac{3}{8} + \frac{\delta_2}{32}.$$

□

Case 3

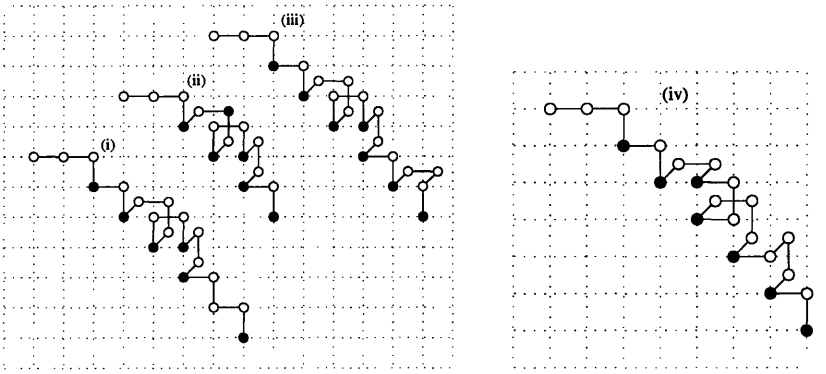

Figure 5-7: Case 3.

Lemma 30. *There is a modification of the* DIAGONAL FOLDING ALGORITHM *with approximation guarantee at least* $\frac{3}{8} + \frac{\delta_3}{32}$.

Proof. An even-1 in Case 3 is in a block of length 1. Thus, substrings containing such an even-1 look like: 1001001, 100001001, etc. In other words, an even-1 in Case 3 is in a substring $10^{q_1}10^{q_2}1$ where q_1 and q_2 are both positive even integers. Consider the string $10^i10^{q_1}10^{q_2}10^j1$ where i and j are odd integers, i.e. the first two 1's and last two 1's in the string are odd-1's and the middle 1 is an even-1. (We can assume for now that there is no even-1 between the first two odd-1's or the last two odd-1's because as we will discuss later, if there are two Case 3 even-1's that share an odd-1 as a neighbor, our folding will only use one of these even-1's.) We will use four different modifications of the DIAGONAL FOLDING ALGORITHM based on the values of i and j. We name these types of even-1's as follows: (i) $i \geq 3, j \geq 3$; (ii) $i = 1, j = 1$; (iii) $i \geq 3, j = 1$; (iv) $i = 1, j \geq 3$. See Figure 5-7 for illustrations of the foldings for each of these types. We now distinguish two cases: first, if more than half of the Case 3 even-1's are of type (i),(ii) or (iii), and second, if more than half are of type (iv).

Suppose that more than half of the Case 3 even-1's are of types (i)-(iii). The foldings for these three types can be used consecutively (as opposed to the folding of (iv), which cannot be applied right after itself). However, we can only guarantee a contact for half of the even-1's in these three types because we may have, for example,

$10^i10^{q_1}10^{q_2}1001001$, i.e. two even-1's that are both adjacent to the same odd-1. In this case, we can only get an extra contact for one such even-1.

We note that the approximation guarantee obtained is a linear combination of the approximation guarantees for the three types, weighted by their relative frequency. The worst case therefore occurs if half the of Case 3 even-1's are of a single type, (i),(ii) or (iii). Since they change the length of the main diagonal, types (i) and (ii) are worse than (iii).

Since types (i) and (ii) either remove an odd-1 from the main diagonal (type (ii)) or result in some even-1's from $S_\mathcal{E}$ not having contacts on the main diagonal (type (i)), they are worse than type (iii). Both of these types have the same approximation guarantee. We will just analyze the case when half the Case 3 even-1's are type (i). The folding modification for this type changes the length of the main diagonal to at least:

$$\frac{1}{2}\left(\mathcal{O}[S] + \frac{\delta_3\mathcal{O}[S]}{4}\right).$$

This is because we assumed that at least half of the Case 3 even-1's are of types (i)-(iii) and we can use half of these even-1's. For each even-1 in Case 3, we lose 1 odd-1 on the main diagonal and we gain 2 contacts per even-1 off the main diagonal. Therefore, the approximation guarantee is:

$$\left(3\left(\frac{1}{2}\left(\mathcal{O}[S] + \frac{\delta_3\mathcal{O}[S]}{4}\right) - \frac{\delta_3\mathcal{O}[S]}{4}\right) + \frac{2\delta_3\mathcal{O}[S]}{4}\right)\frac{1}{4\mathcal{O}[S]} =$$
$$\frac{3}{8} + \delta_3\left(\frac{3}{8} - \frac{3}{4} + \frac{1}{2}\right)\frac{1}{4\mathcal{O}[S]} = \frac{3}{8} + \frac{\delta_3}{32}.$$

In the other case, when more than half of Case 3 even-1's are of type (iv), per type (iv) even-1 we obtain 2 contacts and one odd-1 is not used on the main diagonal. Therefore, in this case the approximation guarantee is better than $\frac{3}{8} + \frac{\delta_3}{32}$. □

Case 4

Lemma 31. *There is a modification of the* Diagonal Folding Algorithm *with approximation guarantee at least* $\frac{3}{8} + \frac{\delta_4}{24}$.

Proof. In Case 4, even-1's occur in blocks of length at least 3 and at most 7 that begin and end with an odd-1. Consider all the odd-1's that occur in blocks of length at least 3 and at most 7 and that begin and end with an odd-1. Note that the number of such odd-1's is at least $\frac{4\delta_4}{3}$ since the ratio of odd-1's to even-1's in this case is at least 4 to 3. To deal with Case 4, we will cut the loop $L(S)$ into two pieces in a particular way. Previously, we cut the loop $L(S)$ into two pieces to secure certain properties. Here, we will cut the loop $L(S)$ into two pieces in the following (different)

way: Let s_0 be an element in $S_{\mathcal{O}}$ that divides $S_{\mathcal{O}}$ into two parts, each containing half the odd-1's of Case 4 (i.e. odd-1's that are in blocks with Case 4 even-1's). This will be one of the new points at which we cut $L(S)$. Then we find another point such that one string contains at least half the odd-1's and the other string contains at least half the even-1's. For these new strings, let us call them $S'_{\mathcal{O}}$ and $S'_{\mathcal{E}}$, note that now $S'_{\mathcal{E}}$ contains at least half of the $\mathcal{O}[S]$ odd-1's that were in blocks with the Case 4 even-1's. Thus, we can apply the Case 2 folding to $S'_{\mathcal{E}}$, i.e. $S'_{\mathcal{E}}$ now contains blocks of 1's that begin with odd-1's. This gives the following the approximation guarantee:

$$\left(\frac{3\mathcal{O}[S]}{2} + \frac{1}{4}\frac{4\delta_4\mathcal{O}[S]}{3}\frac{1}{2}\right)\frac{1}{4\mathcal{O}[S]} = \frac{3}{8} + \frac{\delta_4}{24}.$$

□

Lemma 32. *We can modify the* DIAGONAL FOLDING ALGORITHM *to create a folding with* $\frac{3}{8}OPT + \frac{\delta(S)}{256} - O(1)$ *contacts for any binary string* S.

Proof. Setting all the approximation guarantees equal, we have:

$$\frac{\delta_1}{40} = \frac{\delta_2}{32} = \frac{\delta_3}{32} = \frac{\delta_4}{24}.$$

Using the fact that $\delta_1 + \delta_2 + \delta_3 + \delta_4 = \frac{\delta}{2}$, we obtain that when $\delta_1 \geq \frac{5\delta}{32}$, we should use the Case 1 modification. This implies that the approximation guarantee for the four cases is at least:

$$\frac{3}{8} + \frac{5\delta}{32}\frac{1}{40} = \frac{3}{8} + \frac{\delta}{256}.$$

□

5.5 Discussion

We have described an algorithm for the 3D string folding problem that slightly improves on the previously best-known algorithm to yield an approximation guarantee of .37501. The contribution of this algorithm is not so much the actual gain in the approximation ratio, but the demonstration that the previously best-known algorithm is not optimal, even though there have been no improvements for almost a decade. Our algorithm capitalizes on combinatorial properties of the string rather than using purely geometric ideas. New geometric ideas are most likely necessary to improve the approximation guarantee significantly.

In conclusion, improvement on the 2D algorithm does not immediately lead to an improvement for the 3D case because it might be the case that the 2D folding is asymmetric and therefore cannot be "layered". Thus, despite the fact that the first algorithm of Hart and Istrail [HI96] for the 3D problem used their 2D algorithm

as a subroutine, improving the approximation ratios for the two problems seems to present different difficulties.

Chapter 6

Linear Ordering

6.1 Introduction

Vertex ordering problems comprise a fundamental class of combinatorial optimization problems that, on the whole, is not well understood. For the past thirty years, combinatorial methods and linear programming techniques have failed to yield improved approximation guarantees for many well-studied vertex ordering problems such as the linear ordering problem and the traveling salesman problem. Semidefinite programming has proved to be a powerful tool for solving a variety of cut problems, as first exhibited for the maximum cut problem [GW95]. Since then, semidefinite programming has been successfully applied to many other problems that can be categorized as cut problems such as coloring k-colorable graphs [KMS98], maximum-3-cut [GW04], maximum k-cut [FJ97], maximum bisection and maximum uncut [Ye01, HZ01, YZ03], and correlation clustering [CGW03], to name a few. In contrast, there is no such comparably general approach for approximating vertex ordering problems.

In this chapter, we focus on a well-studied and notoriously difficult combinatorial optimization problem known as the linear ordering problem. Given a complete directed weighted graph, the goal of the linear ordering problem is to find an ordering of the vertices that maximizes the weight of the forward edges. A *vertex ordering* is defined as a mapping of each vertex $i \in V$ to a unique label $\ell(i)$. An edge $(i, j) \in A$ is a *forward edge* with respect to an ordering if $\ell(i) < \ell(j)$. Without loss of generality, we can assume that the labels are integers chosen from the range $\{1, 2, \ldots, n\}$, where $n = |V|$.

Although the problem is NP-hard [Kar72], it is easy to estimate the optimum to within a factor of $\frac{1}{2}$: In any ordering of the vertices, either the set of forward

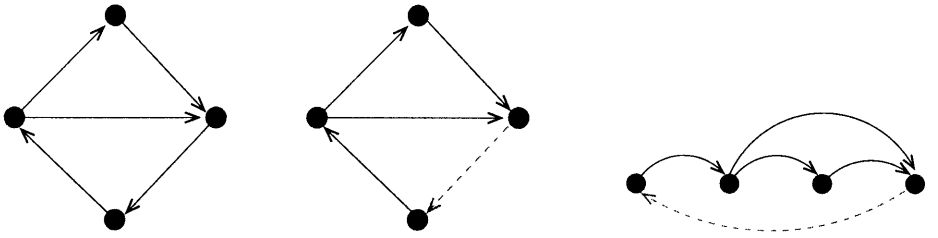

Figure 6-1: The maximum acyclic subgraph of the graph on the left is shown with a corresponding optimal vertex ordering.

edges or the set of backward edges accounts for at least half of the total edge weight. It is not known whether the maximum can be estimated to a better factor using a polynomial-time algorithm. Approximating the problem to within better than $\frac{65}{66}$ is NP-hard [NV01].

The linear ordering problem is also known as the maximum acyclic subgraph problem. Given a weighted directed graph, the maximum acyclic subgraph problem is that of finding the maximum weight subgraph that contains no cycles. The forward edges in any linear ordering comprise an acyclic subgraph and a topological sort of an acyclic subgraph yields a linear ordering of the vertices in which all edges in the acyclic subgraph are forward edges.

6.1.1 Background

Improving upon the best-known approximation guarantee of $\frac{1}{2}$ for the linear ordering problem has been an open problem since the 1970's—from the inception of the field of approximation algorithms. It is one of the most fundamental graph optimization problems for which no non-trivial approximation algorithm is known. Another problems that falls into this category is, for example, the *vertex cover* problem. Given an undirected graph, the vertex cover problem is to find a minimum cardinality subset of the vertices such that every edge in the graph has at least one endpoint in this set of vertices. The vertices in any minimal matching has size at most twice the size of the minimum vertex cover. No better approximation factor is known and there is a gap between this approximation ratio and the best known hardness of 1.36 [DS02].

A major open problem is to close the gap between the best-known approximation guarantee of $\frac{1}{2}$ and the best-known hardness of $\frac{65}{66}$ for the linear ordering problem. Combinatorial methods have failed to yield algorithms with a constant-factor approximation guarantee of more than $\frac{1}{2}$. The goal of most approaches for improving the approximation guarantee for an NP-hard maximization problem is to find a good upper bound on the value of an optimal solution. For a directed graph, $G = (V, A)$, a trivial bound on the size of an optimal linear ordering is the total edge weight, which is $|A|$ if the graph is unweighted. This is the upper bound used in the combinatorial

factor $\frac{1}{2}$-approximation algorithm in which a arbitrary ordering of the vertices is considered, and either the set of forward edges or the set of backward edges is selected. At least one set contains at least half the total edge weight. Thus, the "all edges" or "total edge weight" bound can be no more than twice as large as optimal. The major open problem is to find an efficiently computable bound that is strictly less than twice the value of an optimal solution for all graphs.

Linear programming formulations are often used to compute upper bounds on the optimal values of instances of NP-hard maximization problems. A classical integer program for the linear ordering problem has a variable x_{ij} for each edge (i, j) in the graph.

$$\begin{aligned} & \max \sum_{ij \in A} w_{ij} x_{ij} \\ \sum_{ij \in C} x_{ij} & \leq |C| - 1 \quad \forall \text{ cycles } C \in A \\ x_{ij} & \in \{0, 1\}. \end{aligned}$$

In a solution for this integer program, at least one edge (i, j) in each cycle C has value $x_{ij} = 0$. Thus, a solution comprised of all edges (i, j) such that $x_{ij} = 1$ corresponds to an acyclic subgraph. The linear programming relaxation is obtained by relaxing the constraint that the x_{ij} variables are integral.

$$\begin{aligned} & \max \sum_{ij \in A} w_{ij} x_{ij} \\ \sum_{ij \in C} x_{ij} & \leq |C| - 1 \quad \forall \text{ cycles } C \in A \\ 0 \leq x_{ij} & \leq 1. \end{aligned} \tag{6.1}$$

This linear programming relaxation has an exponential number of constraints. However, there is a simple polynomial-time separation oracle, so the upper bound it provides can be computed in polynomial time. The separation oracle is as follows: Given a solution $\{x_{ij}\}$ to the linear program, let $y_{ij} = 1 - x_{ij}$. Find the shortest cycle. If the shortest cycle C' has value less than 1, then the total value of the x_{ij} variables corresponding to the cycle C' is greater than $|C'| - 1$.

There is another well-studied linear programming relaxation for this problem. It is a relaxation of the following integer program in which there is a variable x_{ij} for every pair of vertices $i, j \in V$, i.e. there is a variable for every edge in the complete graph on n vertices. If for a pair of vertices $i, j \in V$, there is no edge (i, j) in the graph, then the edge weight $w_{ij} = 0$. This integer program has a constraint for each directed 2-cycle and each directed 3-cycle in the complete directed graph on the vertices of

the input graph $G = (V, A)$. This integer program is due to Potts [Pot80]. Integer solutions correspond to acyclic subgraphs. A proof of this can be found in [New00]. In contrast to the previous linear program (6.1), the linear programming relaxation of this integer program has a polynomial number of constraints.

$$\begin{aligned} & \max \sum_{i,j \in V} w_{ij} x_{ij} \\ x_{ij} + x_{ji} &= 1 \quad \forall i, j \in V \\ x_{ij} + x_{jk} + x_{ki} &\leq 2 \quad \forall i, j, k \in V \\ 0 \leq x_{ij} &\leq 1. \end{aligned} \tag{6.2}$$

Both of these relaxations, (6.1) and (6.2), have the same optimal value [NV01]. The quality of a linear programming relaxation in terms of the upper bound it provides is usually measured by the *integrality gap*. The integrality gap is the maximum ratio of the optimal fractional solution to the optimal integral solution taken over all graphs with non-negative edge weights. For example, if there is a graph with a maximum acyclic subgraph of half the edges for which the optimal value of a linear programming relaxation is all of the edges, then this example would demonstrate that the integrality gap of this relaxation is 2.

The integrality gap of both of these relaxations is $2-\epsilon$ for any $\epsilon > 0$ [NV01, New00]. Since the gap is arbitrarily close to 2, in the worst case the upper bound provided by these linear relaxations is no better than the "all edges" bound. Thus, it is unlikely that these relaxations can be used to approximate the problem to within a factor greater than $\frac{1}{2}$. The graphs used to demonstrate these integrality gaps are random graphs in which each edge in the complete undirected graph on n vertices is chosen with uniform edge probability of approximately $2^{\sqrt{\log n}}/n$ and then randomly directed. For sufficiently large n, such a random graph has a maximum acyclic subgraph close to half the edges with high probability. However, each of the polyhedral relaxations studied provide an upper bound for these graphs that is asymptotically close to all the edges, which exceeds the integral optimal by a factor arbitrarily close to 2.

6.1.2 Organization

We first discuss a new semidefinite programming relaxation for the linear ordering problem. A vertex ordering for a graph $G = (V, E)$ with n vertices can be fully described by a series of $n-1$ cuts. We use this simple observation to relate cuts and orderings and to relate cut and ordering semidefinite programs. This and other ideas behind the development of our semidefinite programming relaxation are thoroughly discussed in Chapter 3.

Next we show that for sufficiently large n, if we choose a random directed graph on n vertices with uniform edge probability $p = \frac{d}{n}$ (i.e. every edge in the complete undi-

rected graph on n vertices is chosen with probability p and then directed randomly), where $d = \omega(1)$, with high probability, the bound provided by our semidefinite program for this graph will be no more than 1.64 times the integral optimal. In particular, the graphs used in [NV01] to demonstrate integrality gaps of 2 for the widely-studied polyhedral relaxations fall into this category of random graphs, i.e. each edge in these graphs is chosen with probability $2^{\sqrt{\log n}}/n$. The main idea is that our semidefinite relaxation provides a "good" bound on the value of an optimal linear ordering for a graph if it has no small roughly balanced bisection. With high probability, a random graph with uniform edge probability contains no such small balanced bisection. These results are also presented in [New04].

6.2 Relating Cuts and Orderings

The techniques we apply to study efficiently computable upper bounds for the linear ordering problem are based on semidefinite programming techniques applied by Goemans and Williamson to the maximum cut problem [GW95]. We discuss their methods and the connections between cut problems and vertex ordering problems.

6.2.1 Relaxations for Cut Problems

Given an undirected weighted graph $G = (V, E)$, the maximum cut (maxcut) problem is to find a bipartition $(S, \overline{S})$ of the vertices that maximizes the weight of the edges crossing the partition. The maxcut problem is one of Karp's original NP-complete problems [Kar72]. Because it is unlikely that there exist efficient algorithms for such NP-hard optimization problems, a common approach is to find an efficient *ρ-approximation algorithm.* A *ρ-approximation algorithm* is a polynomial time algorithm that produces a solution with value at least ρ times that of an optimal solution. In 1976, Sahni and Gonzales [SG76] gave a $\frac{1}{2}$-approximation algorithm for the maxcut problem. Their greedy algorithm iterates through the vertices in an arbitrary order and adds vertex i to S or $\overline{S}$ depending on which placement maximizes the weight of the edges crossing the cut so far. For nearly twenty years, $\frac{1}{2}$ was the best constant factor approximation known. Linear programming relaxations have been studied in order to find improved bounds on the value of an optimal solution [Bar83, BGM85, BM86, PT95], but the bounds provided by these relaxations were shown to be larger than the optimal by a factor of 2 in the worst case [Pol92].

In 1993, Goemans and Williamson used a semidefinite programming relaxation to obtain a breakthrough .87856-approximation algorithm for the maxcut problem [GW95]. The goal of their algorithm is to assign each vertex $i \in V$ a vector $v_i \in \{1, -1\}$ so as to maximize the weight of the edges (i, j) such that $v_i \neq v_j$. Integer solutions for the following quadratic program correspond to integer solutions for the maxcut problem in which each vertex i in S is assigned $v_i = 1$ and each vertex i in $\overline{S}$ is assigned

$v_i = -1$. That is, an edge (i, j) such that $v_i = 1$ and $v_j = -1$, contributes value w_{ij} to the objective function and an edge (i, j) such that $v_i = v_j$ contributes 0 to the objective function.

$$\begin{aligned} \max \quad & \frac{1}{2}\sum_{i<j} w_{ij}(1 - v_i \cdot v_j) \\ v_i \quad \in \quad & \{1, -1\} \qquad \forall i \in V. \end{aligned} \tag{6.3}$$

A semidefinite programming relaxation is obtained by relaxing the constraint that $v_i \in \{1, -1\}$ to the constraint $v_i \in \mathcal{R}^n$ and adding the constraints that the vectors v_i are unit vectors and that the matrix of dot products $v_i \cdot v_j$ is positive semidefinite.

$$\begin{aligned} \max \quad & \frac{1}{2}\sum_{i<j} w_{ij}(1 - v_i \cdot v_j) \\ v_i \cdot v_i \quad = \quad & 1 \qquad \forall i \in V \\ v_i \quad \in \quad & \mathcal{R}^n \qquad \forall i \in V. \end{aligned} \tag{6.4}$$

Goemans-Williamson gave an algorithm for the maxcut problem in which they first solve this semidefinite programming relaxation, then choose a random hyperplane $r \in \mathcal{R}^n$, and finally place a vertex i in S if $r \cdot v_i < 0$ and in $\overline{S}$ if $r \cdot v_i \geq 0$. The expected value of the edges crossing such a cut is at least .87856 of optimal [GW95].

A closely related graph optimization problem for which Goemans and Williamson also gave a radically improved approximation guarantee is the maximum directed cut (dicut) problem. Given a directed weighted graph $G = (V, A)$, the dicut problem is to find a bipartition of the vertices—call these disjoint sets S_1 and S_2—that maximizes the weight of the directed edges (i, j) such that vertex i is in set S_1 and vertex j is in set S_2. In 1993, the best-known approximation guarantee for the dicut problem was $\frac{1}{4}$. This approximation factor can be achieved by randomly assigning the vertices to either S_1 or S_2. The expected weight of the edges (i, j) such that i falls in S_1 and j falls in S_2 is $\frac{1}{4}$ the total edge weight.

Goemans and Williamson gave a greatly improved .79607-approximation algorithm for the dicut problem. Solutions for the following quadratic program correspond to dicuts, i.e. only edges (i, j) such that $v_i = -1$ and $v_j = 1$ contribute weight w_{ij} to the objective function.

$$\begin{aligned} \max \quad & \sum_{i<j} w_{ij} \frac{(1 - v_i \cdot v_j + v_i \cdot v_0 - v_j \cdot v_0)}{4} \\ v_i \quad & \in \quad \{1, -1\} \quad \forall i \in V \cup \{0\}. \end{aligned} \tag{6.5}$$

The semidefinite relaxation that Goemans and Williamson use for their algorithm is obtained by replacing the requirement that v_i be integral with the constraint that the vectors v_i are unit vectors in $\mathcal{R}^{n+1}$ and the matrix of dot products $v_i \cdot v_j$ is positive semidefinite.

$$\begin{aligned} \max \quad & \sum_{i<j} w_{ij} \frac{(1 - v_i \cdot v_j + v_i \cdot v_0 - v_j \cdot v_0)}{4} \\ v_i \cdot v_i \quad & = \quad 1 \quad \forall i \in V \cup \{0\} \\ v_i \quad & \in \quad \mathcal{R}^{n+1} \quad \forall i \in V \cup \{0\}. \end{aligned} \tag{6.6}$$

6.2.2 A Relaxation for the Linear Ordering Problem

We can generalize the semidefinite programming relaxation for the dicut problem [GW95] to obtain a new semidefinite programming relaxation for the linear ordering problem. The development of this semidefinite relaxation is discussed thoroughly in Chapter 3. We describe a vertex ordering using $n + 1$ unit vectors for each vertex. Each vertex $i \in V$ has $n + 1$ ($n = |V|$) associated unit vectors: $v_i^0, v_i^1, v_i^2, \ldots v_i^n$. In an integral solution, we enforce that $v_i^0 = -1$, $v_i^n = 1$ and that v_i^h and v_i^{h+1} differ for only one value of h, $0 \leq h < n$. This position h denotes vertex i's position in the ordering. For example, suppose we have a graph G that has four vertices, arbitrarily labeled 1 through 4. Consider the vertex ordering in which vertex i is in position i. An integral description of this vertex ordering is:

$$\begin{aligned} \{v_1^0,\ v_1^1,\ v_1^2,\ v_1^3,\ v_1^4\} &= \{-1,\ 1,\ 1,\ 1,\ 1\}, \\ \{v_2^0,\ v_2^1,\ v_2^2,\ v_2^3,\ v_2^4\} &= \{-1, -1,\ 1,\ 1,\ 1\}, \\ \{v_3^0,\ v_3^1,\ v_3^2,\ v_3^3,\ v_3^4\} &= \{-1, -1, -1,\ 1,\ 1\}, \\ \{v_4^0,\ v_4^1,\ v_4^2,\ v_4^3,\ v_4^4\} &= \{-1, -1, -1, -1,\ 1\}. \end{aligned}$$

This representation of an ordering can be viewed as a generalization of the dicut semidefinite program. If we fix an h, and consider the set of vectors $\{v_i^h\}$ in an integral solution, which corresponds to an actual vertex ordering. If we plug these vectors into the dicut semidefinite program (6.5), then the value of the objective function on this

set of vectors is exactly the weight of the forward edges crossing the cut obtained by partitioning the first h vertices and the last $n-h$ vertices in the ordering.

In an integral solution, we can enforce that for each vertex i, the vectors v_i^h and v_i^{h+1} differ for only one value of h between 1 and n, by setting $(v_i^h - v_i^{h-1}) \cdot (v_j^\ell - v_j^{\ell-1}) \geq 0$. Consider the first value of h such that $v_i^{h-1} = -1$ and $v_i^h = 1$. Then $(v_i^h - v_i^{h-1}) = 2$. If there is any other values of j, ℓ such that $v_j^\ell = -1$ and $v_j^{\ell-1} = 1$, then $(v_i^h - v_i^{h-1}) \cdot (v_j^\ell - v_j^{\ell-1}) = -4$, which violates the constraint.

We also enforce that the sum of the dot products of the $v_i^{\frac{n}{2}}$ vectors sum to 0: $\sum_{i,j \in V} v_i^{\frac{n}{2}} \cdot v_j^{\frac{n}{2}} = 0$. For the sake of convenience and without loss of generality, we assume that n is even, possibly by adding a dummy vertex. If n is even, then in an integral solution, exactly half of the vectors in the set $\{v_i^{\frac{n}{2}}\}$ are 1 and exactly half are -1. So the sum of the vectors in this set, i.e. $\sum_{i \in V} v_i^{\frac{n}{2}}$, is 0.

The set of constraints below form an integer quadratic program for the linear ordering problem based on the description of an ordering using unit vectors. A set of vectors $\{v_i^h, v_i^{h-1}, v_j^\ell, v_j^{\ell-1}\}$ contributes value w_{ij} to the objective function exactly when $v_i^{h-1} = v_j^{\ell-1} = -1$ and $v_i^h = v_j^\ell = 1$, i.e. when vertex i is in position h in the ordering and vertex j is in position ℓ in the ordering.

$$
\begin{aligned}
& & \max \quad & \frac{1}{4} \sum_{ij \in A} \sum_{1 \leq h < \ell \leq n} w_{ij} (v_i^h - v_h^{h-1}) \cdot (v_j^\ell - v_j^{\ell-1}) & \\
(P) \quad & (v_i^h - v_i^{h-1}) \cdot (v_j^\ell - v_j^{\ell-1}) & \geq \quad & 0 \quad \forall i,j \in V,\ h,\ell \in [n] & (6.7) \\
& v_i^h \cdot v_i^h & = \quad & 1 \quad \forall i \in V,\ h \in [n] & \\
& v_i^0 \cdot v_0 & = \quad & -1 \quad \forall i \in V & \\
& v_i^n \cdot v_0 & = \quad & 1 \quad \forall i \in V & \\
& \sum_{i,j \in V} v_i^{\frac{n}{2}} \cdot v_j^{\frac{n}{2}} & = \quad & 0 & \\
& v_i^h & \in \quad & \{1,-1\} \qquad \forall i,h \in [n]. &
\end{aligned}
$$

Let $G = (V, A)$ be a directed graph. By $P(G)$, we denote the optimal value of the integer quadratic program P on the graph G. We obtain a semidefinite programming relaxation for the linear ordering problem by relaxing the last constraint in (P) to: $v_i^h \in \mathcal{R}^{n^2+1}$. We denote the optimal value of the relaxation of P on the graph G as $P_R(G)$. There are many additional inequalities that we can add, but we will focus on the relaxation of (P) since the corresponding relaxation is strong enough to prove the results in this chapter. Additional constraints are discussed in Chapter 3.

6.2.3 Cuts and Uncuts

Suppose we have a directed graph $G = (V, A)$ and we are given a set of unit vectors $\{v_i\} \in \mathcal{R}^{n+1}$, $0 \leq i \leq n$. We define the *forward* value of this set of vectors as the value obtained if we compute the value of the dicut semidefinite programming

relaxation [GW95, FG95] using these vectors. Specifically, the forward value for this set of vectors is:

$$\frac{1}{4}\sum_{ij\in A} w_{ij}(v_0+v_i)\cdot(v_0-v_j) \;=\; \frac{1}{4}\sum_{ij\in A} w_{ij}(1-v_i\cdot v_j+v_0\cdot v_i-v_0\cdot v_j). \tag{6.8}$$

In an integral solution for the dicut problem, there will be edges that cross the cut in the backward direction, i.e. they are not included in the dicut. For a specified set of unit vectors, we can view the vectors as having *backward* value. We define the backward value of the set of vectors $\{v_i\}$ as:

$$\frac{1}{4}\sum_{ij\in A} w_{ij}(v_0-v_i)\cdot(v_0+v_j) \;=\; \frac{1}{4}\sum_{ij\in A} w_{ij}(1-v_i\cdot v_j-v_0\cdot v_i+v_0\cdot v_j). \tag{6.9}$$

This can be obtained by replacing v_i by $-v_i$ for $i \neq 0$ in the forward value. The *difference* between the forward and backward value of a set of vectors $\{v_i\}$ is:

$$\frac{1}{2}\sum_{ij\in A} w_{ij}(v_i\cdot v_0-v_j\cdot v_0). \tag{6.10}$$

Lemma 33. *If a directed unweighted graph $G=(V,A)$ has a maximum acyclic subgraph of $(\frac{1}{2}+\delta)|A|$ edges, then there is no set of unit vectors $\{v_i\}$ such that the difference between the forward and backward value of this set of vectors exceeds $2\delta|A|$.*

Proof. We show that given a unit vector solution $\{v_i\}$ to the semidefinite program which maximizes the objective function (6.10), we can find an integral solution (i.e. an actual cut) in which the difference of forward and backward edges crossing the cut is exactly equal to the objective value. If the difference of an actual cut exceeds $2\delta|A|$, e.g. suppose it is $(2\delta+\epsilon)|A|$, then we can find an ordering with $(\frac{1}{2}+\delta+\epsilon/2)|A|$ forward edges, which is a contradiction. This ordering is found by taking the cut that yields $(2\delta+\epsilon)|A|$ more forward than backward edges and ordering the vertices in each of the two sets greedily so as to obtain at least half of the remaining edges.

Suppose we have a set of unit vectors $\{v_i\}$ such that the value of equation (6.10) is at least $(2\delta+\epsilon)|A| = \beta|A|$. We show that we can find an actual cut such that the difference between the forward and the backward edges is at least $\beta|A|$. Note that $v_0\cdot v_i$ is a scalar quantity since v_0 is a unit vector that without loss of generality is $(1,0,0,\ldots)$. Thus, our objective function can be written as $\frac{1}{2}\sum_{ij\in A} w_{ij}(z_j-z_i)$ where $1\geq z_i\geq -1$. This results in the following linear program. We claim that there is an optimal solution to the following linear program that is integral.

$$\begin{aligned} & \max \quad \frac{1}{2}\sum_{ij\in A} w_{ij}(z_j - z_i) \qquad (6.11) \\ -1 \quad \leq \quad & z_i \quad \leq \quad 1, \quad \forall i \in V. \end{aligned}$$

To show this, note that we are optimizing a linear objective function over a polytope that is the cube with vertices in $\{-1,1\}^n$. The vertices of the polytope are integral and so there is always an optimal solution that is integral. Thus, the integral solution obtained must have difference of forward and backward edges that is equal to the objective value (6.11). □

By Lemma 33, if a directed graph has a maximum acyclic subgraph close to half the total edge weight, then there are no cuts that have a high difference of forward and backward value. We will show that if $P_R(G)$ is large, i.e. if the linear ordering semidefinite programming value for a graph G is large, then the backward value for all sets of vectors $\{v_i^h\}$ for each fixed h is small. If a maximum acyclic subgraph of G is close to half the edges, the the forward value for all sets of vectors will also be small by Lemma 33. In particular, the sum of the forward and backward value across defined by the vectors $\{v_i^{\frac{n}{2}}\}$ will be small. In other words, the value of the maxcut semidefinite program (6.4) evaluated at the vectors $\{v_i^{\frac{n}{2}}\}$ will be small. By constraint $\sum_{i,j\in V} v_i^{\frac{n}{2}} \cdot v_j^{\frac{n}{2}} = 0$, these vectors are very "spread out". Thus, we can show that if $P_R(G)$ is large, then in expectation, a random hyperplane will find a cut with small edge weight crossing it such that at least a constant fraction of the vertices are on each side of the cut.

We will also find a discussion of the following problem useful. Consider the problem of finding a balanced partition of the vertices of a given graph (i.e. each partition has size $\frac{n}{2}$) that maximizes the weight of the edges that do *not* cross the cut. This problem is referred to as the max-$\frac{n}{2}$-uncut problem by Halperin and Zwick [HZ01]. Below is an integer quadratic program for the max-$\frac{n}{2}$-uncut problem.

$$\begin{aligned} & & \max \quad & \sum_{ij\in A} w_{ij}\frac{(1 + v_i \cdot v_j)}{2} \\ (T) \qquad & \sum_{i,j\in V} v_i \cdot v_j \quad = \quad & 0 & \\ & v_i \cdot v_i \quad = \quad & 1 \qquad & \forall i \in V \\ & v_i \quad \in \quad & \{1,-1\} \quad & \forall i \in V. \end{aligned}$$

We obtain a semidefinite programming relaxation for the max-$\frac{n}{2}$-uncut problem by relaxing the last (integrality) constraint to: $v_i \in \mathcal{R}^n$, $\forall i$. We denote the value of the relaxation of T on the graph G as $T_R(G)$.

Lemma 34. *Let $G = (V, A)$ be an unweighted graph and let ϵ, δ be positive constants. Suppose the maximum acyclic subgraph of G is $(\frac{1}{2} + \delta)|A|$. If $P_R(G) \geq (1 - \epsilon)|A|$, then $T_R(G) \geq (1 - 2\epsilon - 2\delta)|A|$.*

Proof. For each edge $ij \in A$, using constraint (6.7), we have:

$$\sum_{1 \leq h < \ell \leq n} (v_i^h - v_i^{h-1}) \cdot (v_j^\ell - v_j^{\ell-1}) \leq \tag{6.12}$$

$$\sum_{1 \leq h \leq \frac{n}{2}, 1 \leq \ell \leq \frac{n}{2}} (v_i^h - v_i^{h-1}) \cdot (v_j^\ell - v_j^{\ell-1}) + \tag{6.13}$$

$$\sum_{n \geq h > \frac{n}{2}, n \geq \ell > \frac{n}{2}} (v_i^h - v_i^{h-1}) \cdot (v_j^\ell - v_j^{\ell-1}) + \tag{6.14}$$

$$\sum_{1 \leq h \leq \frac{n}{2}, n \geq \ell > \frac{n}{2}} (v_i^h - v_i^{h-1}) \cdot (v_j^\ell - v_j^{\ell-1}). \tag{6.15}$$

For each edge, we refer to the quantity (6.12) as the forward value for that edge with respect to $P_R(G)$. The same term summed instead over $h > \ell$ is referred to as the *backward* value of the edge with respect to $P_R(G)$. We can simplify the terms above. Let $v_i = v_i^{\frac{n}{2}}$.

$$\begin{aligned}
a_{ij}^{LL} &= \sum_{1 \leq h \leq \frac{n}{2}, 1 \leq \ell \leq \frac{n}{2}} (v_i^h - v_i^{h-1}) \cdot (v_j^\ell - v_j^{\ell-1}) = (v_i + v_0) \cdot (v_j + v_0), \\
a_{ij}^{RR} &= \sum_{\frac{n}{2} < h \leq n, \frac{n}{2} < \ell \leq n} (v_i^h - v_i^{h-1}) \cdot (v_j^\ell - v_j^{\ell-1}) = (v_0 - v_i) \cdot (v_0 - v_j), \\
a_{ij}^{LR} &= \sum_{1 \leq h \leq \frac{n}{2}, \frac{n}{2} \leq \ell \leq n} (v_i^h - v_i^{h-1}) \cdot (v_j^\ell - v_j^{\ell-1}) = (v_i + v_0) \cdot (v_0 - v_j), \\
a_{ij}^{RL} &= \sum_{\frac{n}{2} < h \leq n, 1 \leq \ell \leq \frac{n}{2}} (v_i^h - v_i^{h-1}) \cdot (v_j^\ell - v_j^{\ell-1}) = (v_0 - v_i) \cdot (v_0 + v_j).
\end{aligned}$$

Lemma 35. $\sum_{ij \in A} \frac{1}{4}(a_{ij}^{RR} + a_{ij}^{LL} + a_{ij}^{LR} + a_{ij}^{RL}) = |A|$.

Proof. For every edge $ij \in A$, we have:

$$a_{ij}^{LL} + a_{ij}^{RR} + a_{ij}^{LR} + a_{ij}^{RL} = 4.$$

□

By definition, we have $P_R(G) \leq \sum_{ij \in A} \frac{1}{4}(a_{ij}^{LL} + a_{ij}^{RR} + a_{ij}^{LR})$. By assumption, $P_R(G) \geq (1 - \epsilon)|A|$, so we have:

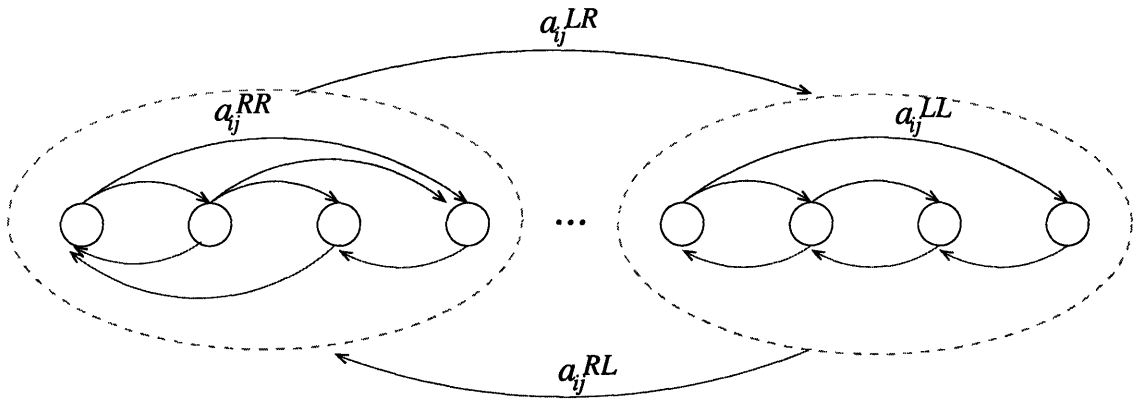

Figure 6-2: Summing a_{ij}^{LR} over all the edges results in the *forward* value and summing a_{ij}^{RL} over all the edges results in the *backward* value.

$$\sum_{ij \in A} \frac{1}{4} a_{ij}^{RL} \leq \epsilon |A|.$$

The above inequality says that the *backward* value of the vectors $\{v_i\}$ (i.e. quantity (6.9)) is at most the *backward* value of $P_R(G)$. By Lemma 33, the difference of the edges crossing the cut in the forward direction and the edges crossing the cut in the backward direction is at most $2\delta|A|$.

$$\sum_{ij \in A} \frac{1}{4} (a_{ij}^{LR} - a_{ij}^{RL}) \leq 2\delta |A|.$$

This implies that the forward value cannot exceed the backward value by more than $2\delta|A|$. Thus, we can bound the forward value as follows:

$$\sum_{ij \in A} \frac{1}{4} a_{ij}^{LR} \leq (\epsilon + 2\delta)|A|.$$

This implies that if we sum the quantity $\frac{1}{4}(a_{ij}^{LL} + a_{ij}^{RR})$ over all edges in A, then the total value of this sum is at least $(1 - 2\epsilon - 2\delta)|A|$.

$$\sum_{ij \in A} \frac{1}{4} (a_{ij}^{LL} + a_{ij}^{RR}) = \sum_{ij \in A} \frac{1 + v_i \cdot v_j}{2}.$$

Thus, we have:

$$\sum_{ij \in A} \frac{1 + v_i \cdot v_j}{2} \geq (1 - 2\epsilon - 2\delta)|A|.$$

□

6.3 Balanced Bisections of Random Graphs

A *bisection* of a graph is a partition of the vertices into two equal (or with cardinality differing by one if n is odd) sets. We use a related definition in this section.

Definition 36. *A γ-bisection of a graph for $\gamma \leq \frac{1}{2}$ is the set of edges that cross a cut in which each set of vertices has size at least γn.*

Suppose we choose an undirected random graph on n vertices in which every edge is present with probability $p = \frac{d}{n}$. The expected degree of each vertex is d and the expected number of edges is $\frac{dn}{2}$. We call such a class of graphs $G_{n,p}$.

Lemma 37. *For any fixed positive constants ϵ, γ, if we choose a graph $G \in G_{n,p}$ on n vertices for a sufficiently large n with $p = \frac{d}{n}$ and $d = \omega(1)$, then every γ-bisection contains at least $(1-\epsilon)\gamma(1-\gamma)nd$ edges with high probability.*

Proof. We use the principle of deferred decisions. First, we choose a $\gamma n, (1-\gamma)n$ partition of the vertices. Thus $\gamma(1-\gamma)n^2$ edges from the complete graph on n vertices cross this cut. Then we can choose the random graph G by picking each edge with probability $p = \frac{d}{n}$. The expected number of edges from G crossing the cut is $\mu = (\gamma(1-\gamma)n^2)(\frac{d}{n}) = \gamma(1-\gamma)dn$. For each edge in the complete graph that crosses the cut, we have the indicator random variable X_i such that $X_i = 1$ if the edge crosses the cut and $X_i = 0$ if the edge does not cross the cut. Let $X = \sum X_i$, i.e. X is the random variable for the number of edges that cross the cut. By Chernoff Bound, we have:

$$\Pr[X < (1-\epsilon)\gamma(1-\gamma)dn)] < e^{-\frac{\epsilon^2\gamma(1-\gamma)dn}{2}}.$$

We can union bound over all the possible γ-bisections. There are less than 2^n ways to divide the vertices so that at least γn are in each set. Thus, the probability that the minimum γ-bisection of G is less than a $(1-\epsilon)$ fraction of its expectation is:

$$\Pr[\min \gamma\text{-bisection}(G) < (1-\epsilon)\gamma(1-\gamma)nd] < \frac{2^n}{e^{\frac{\epsilon^2\gamma(1-\gamma)dn}{2}}}.$$

Let $d = \omega(1)$. Then for any fixed positive constants γ, ϵ, this probability will be arbitrarily small for sufficiently large n. □

6.4 A Contradictory Cut

In this section, we prove our main theorem. Suppose we choose a directed random graph on n vertices in the following way: we include every edge in the complete undirected graph with probability $p = \frac{d}{n}$. Then we randomly direct each edge. We call this class of graphs $\vec{G}_{n,p}$. Note that if we randomly choose a graph from $\vec{G}_{n,p}$, the underlying undirected graph is randomly chosen from $G_{n,p}$.

Theorem 38. *For sufficiently large n, $d = \omega(1)$, and $p = \frac{d}{n}$, if we randomly choose a graph $\vec{G} \in \vec{G}_{n,p}$, then with high probability, the ratio $P_R(\vec{G})/P(\vec{G}) \leq 1.64$.*

We divide the proof of Theorem 38 into two cases: (i) when $d = o(n)$ and (ii) when $d = \Omega(n)$. The proof of both cases is similar. As stated above, we choose a directed graph $\vec{G}$ from $\vec{G}_{n,p}$ since the linear ordering problem is defined for directed graphs. However, in our proof, we really only rely on properties of the underlying *undirected* graph, which we refer to as G (chosen from $G_{n,p}$).

The main idea behind the proof is that since with high probability, every γ-bisection of random graph is very close to $\gamma(1-\gamma)dn$ edges, we can weight the edges in the complete graph in such a way so that with high probability, every γ-bisection of the complete graph has negative weight. We refer to this weighted graph as G_w. Then we can show that if the value $P_R(\vec{G})$ is "high", we can use a random hyperplane to find a γ-bisection of the weighted graph G_w that is non-negative for some constant $\gamma > 0$.

Let E represent the edges in the complete undirected graph K_n for some fixed n. Let $A \subseteq E$ represent the edges both in the directed graph $\vec{G}$ and the underlying undirected graph G, which is chosen at random from $G_{n,p}$. Let ϵ_1 be a small positive constant whose value can be arbitrarily small for sufficiently large n. We weight the edges in E as follows:

$$w_{ij} = -\frac{n}{(1-\epsilon_1)d} \quad \text{if} \quad ij \in A, \tag{6.16}$$

$$w_{ij} = 1 \quad \text{if} \quad ij \in E - A. \tag{6.17}$$

We refer to this weighted graph as G_w. If we choose a random $\frac{1}{2}$-bisection of G_w, the expected contribution of the edges from A to the bisection is roughly $-\frac{n^2}{4}$. The expected contribution of the edges from $E-A$ is roughly $\frac{n^2}{4} - \frac{dn}{4}$. Thus, in expectation, the bisection should have a negative value.

Lemma 39. *Let $\gamma > 0$ be a fixed positive constant. If G_w is chosen on n vertices, for sufficiently large n, then with high probability every γ-bisection of G_w has negative weight.*

Proof. By Lemma 37 with high probability a γ-bisection of G has at least $(1-\epsilon_1)\gamma(1-\gamma)nd$ edges. Thus, with high probability the total weight of the edges in the minimum γ-bisection of G_w is at most:

$$\begin{aligned}
|E-A| + |A|\left(-\frac{n}{(1-\epsilon_1)d}\right) &= \\
\gamma(1-\gamma)n^2 - (1-\epsilon_1)\gamma(1-\gamma)nd + (1-\epsilon_1)\gamma(1-\gamma)nd\left(-\frac{n}{(1-\epsilon_1)d}\right) &= \\
\gamma(1-\gamma)\left(n^2 - (1-\epsilon_1)nd + (1-\epsilon_1)nd\left(-\frac{n}{(1-\epsilon_1)d}\right)\right) &= \\
\gamma(1-\gamma)\left(-(1-\epsilon_1)nd\right) &< 0.
\end{aligned}$$

□

If the value of $P_R(\vec{G})$ is high, i.e. at least $(1-\epsilon_1)|A|$ for some small constant $\epsilon_1 > 0$, and the maximum acyclic subgraph is $(\frac{1}{2}+2\delta)|A|$, then by Lemmas 33 and 34, we have $T_R(G) \geq (1-2\epsilon-2\delta)|A|$. Let $\epsilon_2 = 2\epsilon + 2\delta$.

To prove our next lemma, we use the following theorem from [GW95]. The quantity W stands for the weight of the edges cut by a random hyperplane and $E[W]$ stands for the expected value of the edge weight cut by a random hyperplane.

Theorem 2.7 [**GW**] Let $W_- = \sum_{i<j} w_{ij}^-$, where $x^- = \min(0,x)$. Then

$$\{E[W] - W_-\} \geq \alpha\left\{\frac{1}{2}\sum_{i<j} w_{ij}(1 - v_i \cdot v_j) - W_-\right\}.$$

We will apply this theorem to the graph G_w and show if the value of $T_R(G)$ is high, i.e. $T_R(G) \geq (1-\epsilon_2)|A|$, and if $\epsilon_2 < .36$, then the expected weight cut by a random hyperplane is non-negative. Moreover, the expected weight is βn^2, where β is a small positive constant. Thus, at least $\sqrt{\beta}n$ vertices appear on each side. So setting $\gamma = \sqrt{\beta}$, we will be able to prove the lemma.

Lemma 40. *Let $\gamma > 0$ be a small fixed constant (e.g. $\gamma = .01$). For sufficiently large n, choose a graph $G \in G_{n,p}$, for $p = \frac{d}{n}$ where $d = \omega(1)$. Let G_w be the weighted graph obtained by weighting each edge in G with edge weight $-\frac{n}{(1-\epsilon_1)d}$ and each remaining edge in the complete graph on n vertices with edge weight 1.*

Let $\{v_i\}$, $i \in V$ be a set of unit vectors that satisfy the following constraints:

$$\sum_{i,j\in V} v_i \cdot v_j = 0 \tag{6.18}$$

$$\sum_{ij \in A} \frac{1 + v_i \cdot v_j}{2} \geq (1 - \epsilon_2)|A|. \tag{6.19}$$

If $\epsilon_2 < .36$, then we can find a γ-bisection of G_w with a strictly positive value.

Proof. We use Goemans-Williamson's random hyperplane algorithm to show that we can find a cut that is roughly balanced and has a strictly positive value. Let W represent the total weight of the edges that cross the cut obtained from a random hyperplane. Let W_- denote the sum of the negative edge weights, i.e. $W_- = -|A|\frac{n}{(1-\epsilon_1)d}$. Applying Theorem 2.7 from [GW], we have:

$$\begin{aligned} E[W] &\geq \alpha \left\{ \frac{1}{2} \sum_{i<j} w_{ij}(1 - v_i \cdot v_j) - W_- \right\} + W_- \\ &\geq \alpha \left\{ \sum_{i<j:w_{ij}>0} w_{ij} \frac{1 - v_i \cdot v_j}{2} + \sum_{i<j:w_{ij}<0} |w_{ij}| \frac{1 + v_i \cdot v_j}{2} \right\} + W_-. \end{aligned}$$

To bound $E[W]$, we need to determine the values of three quantities:

(i) $\sum_{i<j:w_{ij}>0} w_{ij} \frac{1-v_i \cdot v_j}{2}$,

(ii) $\sum_{i<j:w_{ij}<0} |w_{ij}| \frac{1+v_i \cdot v_j}{2}$,

(iii) W_-.

By definition $W_- = -|A|\frac{n}{(1-\epsilon_1)2d}$, so we know quantity (iii). We will now compute quantities (i) and (ii). First, we will compute quantity (i), i.e. we want to calculate the value of $\sum_{i<j:w_{ij}>0} \frac{1-v_i \cdot v_j}{2}$, which is equal to quantity (i) since all edges with positive edge weight have weight 1.

By condition (6.18), we have that $\sum_{i,j \in V} v_i \cdot v_j = 0$ and therefore $\sum_{i<j} \frac{1-v_i \cdot v_j}{2} = \frac{n^2}{4}$. By condition (6.19), we have that $\sum_{i<j:w_{ij}<0} \frac{1-v_i \cdot v_j}{2} \leq \epsilon_2|A|$.

$$\begin{aligned} \sum_{i<j:w_{ij}>0} \frac{1 - v_i \cdot v_j}{2} &= \sum_{i<j} \frac{1 - v_i \cdot v_j}{2} - \sum_{i<j:w_{ij}<0} \frac{1 - v_i \cdot v_j}{2} \\ &\geq \sum_{i<j} \frac{1 - v_i \cdot v_j}{2} - \epsilon_2|A| \\ &= \frac{n^2}{4} - \epsilon_2|A|. \end{aligned}$$

Thus, quantity (i) is at least $\frac{n^2}{4} - \epsilon_2|A|$. By constraint (6.19), quantity (ii) is at least $(1-\epsilon_2)|A|\frac{n}{(1-\epsilon_1)d}$. Now we have:

$$E[W] \geq \alpha\left\{(\frac{n^2}{4} - \epsilon_2|A|) + \frac{n}{(1-\epsilon_1)d}(1-\epsilon_2)|A|\right\} - \frac{n}{(1-\epsilon_1)d}|A|.$$

For large enough n, we can choose ϵ_1 to be arbitrarily small. (Recall that ϵ_1 is a small positive constant such with high probability the minimum γ-bisection of G for some fixed positive constant γ has at least $(1-\epsilon_1)\gamma(1-\gamma)2nd$ edges.) Let us assume that $|A| = (1 \pm \epsilon)dn$, where $d = \omega(1)$ and $d = o(n)$ and ϵ is an arbitrarily small positive constant. With exponentially high probability, the number of edges in A is $(1-\epsilon)\frac{dn}{2} \leq |A| \leq (1+\epsilon)\frac{dn}{2}$. This can be seen using one application of a Chernoff Bound. The expected weight of the edges cut by a random hyperplane, $E[W]$, can be bounded from below by a value arbitrarily close to the following (i.e. because ϵ and ϵ_1 can be made arbitrarily small):

$$(\frac{3\alpha}{4} - \frac{1}{2} - \frac{\alpha\epsilon_2}{2})n^2 - o(n^2) \geq (.1585 - \frac{\alpha\epsilon_2}{2})n^2 - o(n^2). \tag{6.20}$$

If the value of ϵ_2 is such that the quantity on line (6.20) is strictly greater than βn^2 for some positive constant β, then we will have a contradiction for sufficiently large n. Note that if this value is at least βn^2, then each side of the cut contains at least $\sqrt{\beta}n$ vertices, so it is a $\sqrt{\beta}$-bisection. So we have:

$$(.1585 - \frac{\alpha\epsilon_2}{2})n^2 \geq \beta n^2 \quad \Rightarrow \quad (.1585 - \frac{\alpha\epsilon_2}{2}) \geq \beta.$$

This value will be strictly positive (i.e. greater than .00035) as long as $\epsilon_2 < .36$. Thus, it must be the case that $\epsilon_2 > .36$. Setting $\gamma = \sqrt{.00035} > .01$, there must be a γ-bisection of positive value if $\epsilon_2 < .36$. This contradicts Lemma 39: For any fixed constant $\gamma > 0$ and for sufficiently large n, every γ-bisection of G_w has negative weight.

To conclude the proof, we need to prove the case when $d = \Omega(n)$, i.e. $d = cn$ for some constant $c < 1$. In this case, we will weight the edges in G_w differently, but the idea is the same. We weight the edges as follows:

$$w_{ij} = \frac{c-1}{c} \qquad \text{if} \quad ij \in A, \tag{6.21}$$

$$w_{ij} = 1 \qquad \text{if} \quad ij \in E - A. \tag{6.22}$$

With these edge weights, we can modify Lemma 39 to show that with high proba-

bility, every γ-bisection of G_w has weight no more than $\epsilon_1 n^2$ for some arbitrarily small positive constant ϵ_1 and sufficiently large n. We again apply Theorem 2.7 [GW95]. We need to compute the quantities (i), (ii), and (iii) denoted above. First, note that quantity (i) is the same as above. Namely,

$$\sum_{i<j:w_{ij}>0} w_{ij}\frac{1-v_i\cdot v_j}{2} = \frac{n^2}{4} - \epsilon_2|A|.$$

Next we compute quantity (ii):

$$\sum_{i<j:w_{ij}<0} |w_{ij}|\frac{1+v_i\cdot v_j}{2} = (1-\epsilon_2)|A|\frac{(1-c)}{c}.$$

Next we compute quantity (iii). We have $W_- = |A|\frac{(c-1)}{c}$. Thus, we have:

$$\begin{aligned} E[W] &\geq \alpha\left\{(\frac{n^2}{4} - \epsilon_2|A|) + \frac{(1-c)}{c}(1-\epsilon_2)|A|\right\} + \frac{(c-1)}{c}|A| \\ &= \alpha\left\{\frac{3n^2}{4} - \frac{\epsilon_2 n^2}{2} - \frac{cn^2}{2}\right\} - \frac{n^2}{2} + \frac{cn^2}{2} \\ &= \left(\frac{3\alpha}{4} - \frac{\epsilon_2\alpha}{2} - \frac{c\alpha}{2} - \frac{1}{2} + \frac{c}{2}\right)n^2. \end{aligned}$$

With high probability, all γ-bisections are arbitrarily small, i.e. $\epsilon_1 n^2$ for an arbitrarily small positive constant ϵ_1. Thus, we have:

$$\left(\frac{3}{2} - \frac{1}{\alpha} + c(\frac{1}{\alpha} - 1)\right) < \epsilon_2.$$

The quantity on the left is minimized when $c = 0$. Thus, $\epsilon_2 > .36$. □

We now prove our main theorem.

Proof of Theorem 38: We fix very small positive constants γ, ϵ_1 and choose sufficiently large n. We choose a random undirected graph G from $G_{n,p}$ and randomly direct each edge to obtain the graph $\vec{G}$. We weight the edges of the undirected graph K_n as discussed previously (we use weights (6.16) and (6.17) for $d = \omega(1), o(n)$ and (6.21) and (6.22) for $d = \Theta(n)$) and obtain G_w. By Lemma 39, the minimum γ-bisection of G_w is negative with high probability. Thus, with high probability, if we

solve the linear ordering semidefinite program and obtain $P_R(\vec{G})$, then equation (6.19) holds for the set of vectors $\{v_i = v_i^{\frac{n}{2}}\}$ only when $\epsilon_2 > .36$.

Suppose the maximum acyclic subgraph of $\vec{G}$, i.e. $P(\vec{G})$ is $(\frac{1}{2} + \delta)|A|$ for some positive constant δ. Then for the set of vectors $\{v_i = v_i^{\frac{n}{2}}\}$, the value of $P_R(\vec{G})$ is upper bounded by:

$$P_R(\vec{G}) \leq \sum_{ij \in A} \frac{1}{4}(a_{ij}^{LL} + a_{ij}^{RR} + a_{ij}^{LR}).$$

By Lemma 40:

$$\sum_{ij \in A} \frac{1}{4}(a_{ij}^{LL} + a_{ij}^{RR}) = \sum_{ij \in A} \frac{1 + v_i \cdot v_j}{2} \leq .64|A|.$$

By Lemma 33, the difference between forward and backward measured according to the $\{v_i\}$ vectors is:

$$\sum_{ij \in A} \frac{1}{4}(a_{ij}^{LR} - a_{ij}^{RL}) \leq (2\delta)|A|.$$

Thus,

$$\sum_{ij \in A} \frac{1}{4}(a_{ij}^{LR}) \leq (.18 + \delta)|A|.$$

So we can upper bound the value of $P_R(\vec{G})$ by $(.82+\delta)|A|$. Thus, with high probability, for the graph $\vec{G}$, we have:

$$\frac{P_R(\vec{G})}{P(\vec{G})} \leq \frac{.82 + \delta}{.5 + \delta} \leq \frac{.82}{.5} = 1.64.$$

□

6.5 Discussion and Conjectures

In this chapter and in Chapter 3, we make a connection between cuts and vertex orderings of graphs in order to obtain a new semidefinite programming relaxation for the linear ordering problem. We show that the relaxation is "good" on random graphs chosen with uniform edge probability $\frac{\omega(1)}{n}$, i.e. if we choose such a graph at random, then with high probability, the ratio of the semidefinite programming bound to the integral optimal is at most 1.64.

In [HZ01], Halperin and Zwick give a .8118-approximation for a related problem that they call the max $\frac{n}{2}$-directed-uncut problem. Given a directed graph, the goal of this problem is to find a bisection of the vertices that maximizes the weight of the edges that cross the cut in the forward direction plus the weight of the edges that do not cross the cut. We note that a weaker version of Theorem 38 follows from their

Figure 6-3: This graph demonstrates the worst integrality gap that we are aware of for the semidefinite programming relaxation for the linear ordering problem (P). For a 3-cycle with five isolated vertices, the gap is about 2.25.

.8118-approximation algorithm. This is because their semidefinite program for the max $\frac{n}{2}$-directed uncut problem is the sum over all edges of terms a_{ij}^{LL}, a_{ij}^{RR}, and a_{ij}^{LR}. (To obtain their .8118-approximation, they use a semidefinite program that includes the triangle inequalities (3.12).) If for some directed graph $G = (V, A)$, $P_R(G)$ has value at least $(1 - \epsilon)|A|$, then the value of their semidefinite programming relaxation also has at least this value. Thus, if ϵ is arbitrarily small, we can obtain a directed uncut of value close to .8118 of the edges, which is a contradiction for a random graph with uniform edge probability. With high probability, the largest directed uncut of a random directed graph is arbitrarily close to $\frac{3}{4}$ of the edges. In this chapter, our goal was to give a self-contained proof of this theorem.

We would like to comment on the similarity of this work to the work of Poljak and Delorme [DP93] and Poljak and Rendl [PR95] on the maxcut problem. Poljak showed that the class of random graphs with uniform edge probability could be used to demonstrate an integrality gap of 2 for several well-studied polyhedral relaxations for the maxcut problem [Pol92]. These same graphs can be used to demonstrate an integrality gap of 2 for several widely-studied polyhedral relaxations for the linear ordering problem [NV01]. The similarity of these results stems from the fact that the polyhedral relaxations for the maxcut problem are based on odd-cycle inequalities and the polyhedral relaxations for the linear ordering problem are based on cycle inequalities. Poljak and Delorme subsequently studied an eigenvalue bound for the maxcut problem that is equivalent to the bound provided by the semidefinite programming relaxation used in the Goemans-Williamson algorithm [GW95]. Despite the fact that random graphs with uniform edge probability exhibit worst-case behavior for several polyhedral relaxations for the maxcut problem, Delorme and Poljak [DP93] and Poljak and Rendl [PR95] experimentally showed that the eigenvalue bound provides a "good" bound on the value of the maxcut for these graphs. This experimental evidence was the basis for their conjecture that the 5-cycle exhibited a worst-case integrality gap of 0.88445 for the maxcut semidefinite relaxation [DP93, Pol92]. The gap demonstrated for the 5-cycle turned out to be very close to the true integrality gap of .87856 [FS].

For the semidefinite relaxation of the integer program (P), the worst-case integrality gap of which we are aware is for the graph that contains a directed 3-cycle and isolated vertices. (This was discovered in a joint effort with Prahladh Harsha.) For

example, for a graph G made up of a directed 3-cycle with five isolated vertices, the optimal value $P_R(G)$ is about 2.25. As the number of isolated vertices increases, the value of $P_R(G)$ seems to increase. It was too difficult computationally to run graphs with more than eight vertices.

In closing, we conjecture that our semidefinite programming relaxation provides a "good" bound on the optimal value of a linear ordering for all graphs.

Conjecture 41. *The integrality gap of the semidefinite programming relaxation of* (P) *is at most* $2 - \epsilon$ *for some positive constant* ϵ.

Bibliography

[ABD+97] Richa Agarwala, Serafim Batzoglou, Vlado Dancik, Scott E. Decatur, Martin Farach, Sridhar Hannenhalli, S. Muthukrishnan, and Steven Skiena. Local rules for protein folding on a triangular lattice and generalized hydrophobicity in the HP model. *Journal of Computational Biology*, 4(2):275–296, 1997. Extended abstract also appeared in proceedings of *RECOMB* 1997 and proceedings of *SODA* 1997.

[Ali95] Farid Alizadeh. Interior point methods in semidefinite programming with applications to combinatorial optimization. *SIAM Journal on Optimization*, 5:13–51, 1995.

[Bar83] Francisco Barahona. The max-cut problem in graphs not contractible to K_5. *Operations Research Letters*, 2:107–111, 1983.

[BGM85] Francisco Barahona, Martin Grötschel, and Ali Ridha Mahjoub. Facets of the bipartite subgraph polytope. *Mathematics of Operations Research*, 10:340–358, 1985.

[BKRV00] Avrim Blum, Goran Konjevod, R. Ravi, and Santosh Vempala. Semidefinite relaxations for minimum bandwidth and other vertex-ordering problems. *Theoretical Computer Science*, 235:25–42, 2000. Extended abstract appear in proceedings of *STOC* 1998.

[BL76] Kellogg S. Booth and George S. Lueker. Testing the consecutive ones property, interval graphs, and graph planarity using PQ tree algorithms. *Journal of Computer and System Sciences*, 13:335–379, 1976.

[BL98] Bonnie Berger and Tom Leighton. Protein folding in the hydrophobic-hydrophilic (HP) model is NP-complete. *Journal of Computational Biology*, 5(1):27–40, 1998. Extended abstract in *Proceedings of Second Annual International Conference on Research in Computational Molecular Biology (RECOMB)*, pages 30–39, New York, 1998.

[BM86] Francisco Barahona and Ali Ridha Mahjoub. On the cut polytope. *Mathematical Programming*, 36:157–173, 1986.

[BS97] Bonnie Berger and Peter Shor. Tight bounds for the maximum acyclic subgraph problem. *Journal of Algorithms*, 25(1):1–18, 1997. Extended abstract in *Proceedings of Symposium on Discrete Algorithms (SODA)*, pages 236–243, San Francisco, 1990.

[BV04] Dimitris Bertsimas and Santosh Vempala. Solving convex programs by random walks. *Journal of the ACM*, 51(4):540–556, 2004.

[CGP+98] Pierluigi Crescenzi, Deborah Goldman, Christos H. Papadimitriou, Antonio Piccolboni, and Mihalis Yannakakis. On the complexity of protein folding. *Journal of Computational Biology*, 5(3):423–465, 1998. Extended abstract appear in proceedings of *STOC* 1998.

[CGW03] Moses Charikar, Venkatesan Guruswami, and Anthony Wirth. Clustering with qualitative information. In *Proceedings of the 44th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 524–533, Boston, 2003.

[CHN03] Robert D. Carr, William E. Hart, and Alantha Newman. Discrete optimization models for protein folding. Technical report, Sandia National Laboratories, Albuquerque, New Mexico, 2003.

[CS98] Benny Chor and Madhu Sudan. A geometric approach to betweeness. *SIAM Journal on Discrete Mathematics*, 11:511–523, 1998.

[CW58] Hollis B. Chenery and Tsunehiko Watanabe. International comparisons of the structure of production. *Econometrica*, 26(4):487–521, 1958.

[Dil85] Kenneth A. Dill. Theory for the folding and stability of globular proteins. *Biochemistry*, 24:1501, 1985.

[Dil90] Kenneth A. Dill. Dominant forces in protein folding. *Biochemistry*, 29:7133–7155, 1990.

[dlV83] W. Fernandez de la Vega. On the maximum cardinality of a consistent set of arcs in a random tournament. *Journal of Combinatorial Theory, Series B*, 35:328–332, 1983.

[DP93] Charles Delorme and Svatopluk Poljak. The performance of an eigenvalue bound in some classes of graphs. *Discrete Mathematics*, 111:145–156, 1993. Also appeared in *Proceedings of the Conference on Combinatorics*, Marseille, 1990.

[DS02] Irit Dinur and Shmuel Safra. On the importance of being biased. In *Proceedings of the 34th Annual Symposium on the Theory of Compututing (STOC)*, pages 33–42, 2002.

[FG95] Uriel Feige and Michel X. Goemans. Approximating the value of two prover proof systems with applications to MAX-2-SAT and MAX DICUT. In *Proceedings of the Third Israel Symposium on Theory of Computing and Systems*, pages 182–189, 1995.

[FJ97] Alan Frieze and Mark R. Jerrum. Improved approximation algorithms for MAX-k-Cut and MAX BISECTION. *Algorithmica*, 18:61–77, 1997.

[FS] Uriel Feige and Gideon Schechtman. On the optimality of the random hyperplane rounding technique for MAX-CUT. *Random Structures and Algorithms*. To appear.

[GHL] Harvey J. Greenberg, William E. Hart, and Guiseppe Lancia. Opportunities for combinatorial optimization in computational biology. *INFORMS Journal of Computing*. To appear.

[GKK74] Fred Glover, Ted Klastorin, and Darwin Klingman. Optimal weighted ancestry relationships. *Management Science*, 20:B1190–B1193, 1974.

[GLS81] Martin Grötschel, László Lovász, and Alexander Schrijver. The ellipsoid method and its consequences in combinatorial optimization. *Combinatorica*, 1:169–197, 1981.

[GLS88] Martin Grötschel, László Lovász, and Alexander Schrijver. *Geometric Algorithms and Combinatorial Optimization*. Springer-Verlag, Berlin, 1988.

[GW95] Michel X. Goemans and David P. Williamson. Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming. *Journal of the ACM*, 42:1115–1145, 1995.

[GW04] Michel X. Goemans and David P. Williamson. Approximation algorithms for MAX-3-CUT and other problems via complex semidefinite programming. *STOC 2001 Special Issue of Journal of Computer and System Sciences*, 68:442–470, 2004.

[HI96] William E. Hart and Sorin Istrail. Fast protein folding in the hydrophobic-hydrophilic model within three-eighths of optimal. *Journal of Computational Biology*, 3(1):53–96, 1996. Extended abstract appeared in proceedings of *STOC* 1995.

[HR94] Refael Hassin and Shlomi Rubinstein. Approximations for the maximum acyclic subgraph problem. *Information Processing Letters*, 51(3):133–140, 1994.

[HZ01] Eran Halperin and Uri Zwick. A unified framework for obtaining improved approximation algorithms for maximum graph bisection problems. In *Proceedings of Eighth Conference on Integer Programming and Combinatorial Optimization (IPCO)*, pages 210–225, Utrecht, 2001.

[JR] Raja Jothi and Balaji Raghavachari. Protein folding in the hydrophobic-hydrophilic model: How good is theory in practice? *In preparation.*

[Kar72] Richard M. Karp. Reducibility among combinatorial problems. In *Complexity of Computer Computations*, pages 85–104. Plenum Press, New York, 1972.

[Kar84] Narendra Karmarker. A new polynomial-time algorithm for linear programming. *Combinatorica*, 4:373–395, 1984.

[Kha79] Leonid G. Khachiyan. A polynomial algorithm in linear programming. *Doklady Akademiia Nauk SSR*, 244:1093–1096, 1979.

[KMS98] David R. Karger, Rajeev Motwani, and Madhu Sudan. Improved graph coloring via semidefinite programming. *Journal of the ACM*, 45(2):246–265, 1998.

[KO69] Bernhard Korte and Walter Oberhofer. Zur triangulation von input-output matrizen. *JahrBücher für National Ökonomie und Statistik*, 182:398–433, 1969.

[MPP99] Giancarlo Mauri, Antonia Piccolboni, and Giulio Pavesi. Approximation algorithms for protein folding prediction. In *Proceedings of the 10th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 945–946, 1999.

[New00] Alantha Newman. Approximating the maximum acyclic subgraph. Master's thesis, Massachusetts Institute of Technology, Cambridge, MA, June 2000.

[New02] Alantha Newman. A new algorithm for protein folding in the HP model. In *Proceedings of the 13th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 876–884, San Francisco, 2002.

[New04] Alantha Newman. Cuts and Orderings: On semidefinite relaxations for the linear ordering proble. In *Proceedings of the 7th International Workshop on Approximation Algorithms for Combinatorial Optimization Problems (APPROX)*, Cambridge, 2004.

[NR04] Alantha Newman and Matthias Ruhl. Combinatorial problems on strings with applications to protein folding. In *Proceedings of the 6th Latin American Theoretical Informatics Conference*, pages 369–378, Buenos Aires, 2004.

[NV01] Alantha Newman and Santosh Vempala. Fences are futile: On relaxations for the linear ordering problem. In *Proceedings of the Eighth Conference on Integer Programming and Combinatorial Optimization (IPCO)*, pages 333–347, 2001.

[Pat96] Gabor Pataki. Cone-LP's and semidefinite programs: Geometry and a simplex-type method. In *Proceedings of the Fifth Conference on Integer Programming and Combinatorial Optimization (IPCO)*, pages 162–174, Vancouver, 1996.

[Pol92] Svatopluk Poljak. Polyhedral and eigenvalue approximations of the max-cut problem. *Sets, Graphs and Numbers, Colloqiua Mathematica Societatis Janos Bolyai*, 60:569–581, 1992.

[Pot80] Chris Potts. An algorithm for the single machine sequencing problem with precedence constraints. *Mathematical Programming Study*, 13:78–87, 1980.

[PR95] Svatopluk Poljak and Franz Rendl. Computing the max-cut by eigenvalues. *Discrete Applied Mathematics*, 62(1–3):249–278, September 1995.

[PT95] Svatopluk Poljak and Zsolt Tuza. Maximum cuts and large bipartite subgraphs. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 20:181–244, 1995.

[SG76] Sartaj Sahni and Teofilo Gonzalez. P-complete approximation problems. *Journal of the ACM*, 23(3):555–565, 1976.

[Spe87] Joel Spencer. *Ten Lectures on the Probabilistic Method.* SIAM, Philadelphia, 1987.

[Tho95] Bo Vincent Thomsen. The Archaeological Seriation Problem–a case study in combinatorial optimization. Master's thesis, University of Copenhagen, Copenhagen, Denmark, December 1995.

[Ye01] Yinyu Ye. A .699-approximation algorithm for max-bisection. *Mathematical Programming*, 90(1):101–111, 2001.

[YN76] D.B. Yudin and A.S. Nemirovskii. Informational complexity and effective methods of solution for convex extramal problem. *Ekonomika i Maternaticheski Metody*, 12:357–369, 1976.

[YZ03] Yinyu Ye and Jiawei Zhang. Approximation of dense-n/2-subgraph and the complement of min-bisection. *Journal of Global Optimization*, 25(1):55–73, 2003.