5-2010

The Maximum Cut Problem: Investigating Computational Approaches

Austin Powell
College of William and Mary

Austin_Powell_cover

The Maximum Cut Problem
Investigating Computational Approaches

A thesis submitted in partial fulfillment of the requirement
for the degree of Bachelors of Science in Mathematics from
The College of William and Mary

by

Austin Powell

Accepted for _____

_____
David Phillips, Director

_____
Robert Michael Lewis

_____
Rex Kincaid

_____
Virginia Torczon

Williamsburg, VA
April 19, 2010

THE MAXIMUM CUT PROBLEM: INVESTIGATING COMPUTATIONAL APPROACHES

An Honors Thesis Presented

by

AUSTIN G. POWELL

A thesis submitted in partial fulfillment of the
requirements for the degree of

BACHELOR OF SCIENCE WITH HONORS IN MATHEMATICS

2010

The College of William and Mary

ABSTRACT

THE MAXIMUM CUT PROBLEM: INVESTIGATING COMPUTATIONAL APPROACHES

2010

AUSTIN G. POWELL

B.S., THE COLLEGE OF WILLIAM AND MARY

Directed by: Professor David J. Phillips and Professor R. Michael Lewis

This thesis investigates various computational approaches to the Maximum Cut problem. It is generally believed that Maximum Cut cannot be solved exactly in polynomial time, so we approach the problem using various heuristics and approximation algorithms. We introduce a rank-penalization heuristic that generates feasible solutions to Maximum Cut. Numerical results show that these solutions are comparable to those given by the Goemans-Williamson randomized algorithm [3]. We also implement a branch and bound algorithm using a branching scheme based on optimal dual variables for the Maximum Cut semidefinite programming relaxation. In our test cases, the dual branching scheme performed consistently better than randomized or largest-degree branching schemes.

Keywords: Maximum Cut, semidefinite programming, penalty methods for optimization problems, branch and bound

TABLE OF CONTENTS

ACKNOWLEDGMENTS

First and foremost, I would like to thank my research advisors, Professor David Phillips and Professor Michael Lewis for their support and guidance throughout my research. I would also like to thank my research partner, Rachel Taylor, for the many hours she put into our joint work. I also would like to extend my gratitude towards Professor Rex Kincaid and Professor Virginia Torczon for serving on my defense committee. Finally, I would like to thank the William and Mary Mathematics Department and the NSF CSUMS program for making this research possible. This work was performed in part using computational facilities at the College of William and Mary which were provided with the assistance of the National Science Foundation, the Virginia Port Authority, Sun Microsystems, and Virginia's Commonwealth Technology Research Fund.

CHAPTER 1

INTRODUCTION

1.1 The Maximum Cut Problem

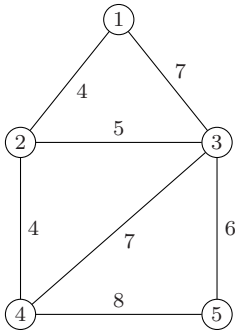
Consider the weighted undirected graph $G = (V, E)$ where V is a set of **vertices** and $E \subseteq V \times V$ is a set of **edges**. A **cut** in G is a nonempty set $S \subset V$. We denote edge weights as w_{ij} for all $(i, j) \in E$ with $w_{ij} = 0$ if $(i, j) \notin E$. In this thesis we assume all edge weights are nonnegative. For a cut $S \subset V$, we define its **cut value** as

$$w(S) = \sum_{i \in S, j \notin S} w_{ij}.$$

Our problem is to find the **maximum cut** of G, defined as

$$\max_{S \subset V} w(S).$$

Example Consider the weighted undirected graph below:



The maximum cut for this graph is $S = \{v_1, v_4\}$ and $S^C = \{v_2, v_3, v_5\}$. The weights of the edges $(v_1, v_2), (v_1, v_3), (v_2, v_4), (v_3, v_4)$, and (v_4, v_5) are included in the value of the cut, giving S a cut value of $4 + 7 + 4 + 7 + 8 = 30$. Because both of their endpoints lie in the cut, the weights of the edges $(2, 3)$ and $(3, 5)$ are not included in the value of S.

The maximum cut of a graph can be found through an exhaustive search of all possible cuts. If we let $n = |V|$, we must compute the value of 2^n different cuts. As an example, if $n = 60$, solving the problem requires the computation of over 10^{18} cut values. Thus, for graphs of this size or larger, heuristics and approximation algorithms are generally used to find cuts that are close to the maximum.

In Chapter 2, we discuss background information on optimization and several well-known optimization techniques. We also discuss the analysis of algorithms and the complexity classes P and NP.

In Chapter 3, we discuss two known approximation algorithms for Maximum Cut. We describe a $\frac{1}{2}$-approximation algorithm that uses a greedy local search [12] as well as a .87856-approximation algorithm based on semidefinite programming [3]. We also briefly describe a branch and bound algorithm for Maximum Cut that solves semidefinite relaxations with triangle inequalities using a spectral bundle method [2].

In Chapter 4, we introduce our algorithm, a nonconvex penalization heuristic for Maximum Cut. We present a rank-one penalization algorithm for Maximum Cut and further extend the algorithm to ranks greater than one. Numerical results are given at the end of Chapter 4.

In Chapter 5, we discuss our implementation of a branch and bound algorithm for Maximum Cut. We implement a branching scheme based on the Lagrangian dual problem of the Maximum Cut semidefinite programming relaxation. Numerical results are given at the end of Chapter 5.

Chapter 6 concludes the paper and comments on the given results. We also give suggestions for future research on the Maximum Cut problem.

In this paper, we assume that the reader holds a basic knowledge of linear algebra and linear programming techniques. For introductory texts in these areas, the reader is referred to [4] and [1].

CHAPTER 2

BACKGROUND

2.1 Optimization

We define an **optimization problem** as any problem of the form

$$\begin{aligned} \max \quad & f(\mathbf{x}) \\ \text{subject to} \quad & \mathbf{x} \in F, \end{aligned} \tag{2.1}$$

where $f : F \subseteq \mathbb{R}^n \to \mathbb{R}$. The function f is the **objective function** of the problem and the set F is the **feasible region**. In this case, the objective function is being maximized; to solve minimization problems, we can maximize $-f(\mathbf{x})$. If $F = \emptyset$, we say (2.1) is **infeasible**. If $\mathbf{x} \in F$, then $\mathbf{x}$ is **feasible** for (2.1).

Global and Local Maximizers For an optimization problem, a feasible solution $\mathbf{x}^*$ is a **global maximizer** of f over F if for all $\mathbf{x} \in F$, we have $f(\mathbf{x}^*) \geq f(\mathbf{x})$. It is important to note that a global maximizer does not necessarily exist for every optimization problem. For example, if $f(x) = x$ and $F = \mathbb{R}$, the problem is unbounded with an optimal objective value of ∞.

We say $\hat{\mathbf{x}}$ is a **local maximizer** of f with respect to F if there exists some $\epsilon > 0$ such that for all $\mathbf{x} \in \{\mathbf{x} \in F : |\hat{\mathbf{x}} - \mathbf{x}| < \epsilon\}$ we have $f(\hat{\mathbf{x}}) \geq f(\mathbf{x})$, where $|\cdot|$ is a norm on $\mathbb{R}^n$.

Relative Approximations Suppose for some optimization problem $\mathcal{P}$ there is an algorithm that returns a solution $\hat{\mathbf{x}}$ that is feasible for $\mathcal{P}$. The algorithm gives a **relative α-approximation** if there exists some α between zero and one such that for any instance of $\mathcal{P}$,

$$f(\hat{\mathbf{x}}) \geq \alpha f(\mathbf{x}^*).$$

If we assume a nonnegative value for $f(\mathbf{x}^*)$, then the algorithm is guaranteed to return a solution that is at least a fraction α of the optimal objective value.

The Quadratic Integer Program Maximum Cut Formulation Given an undirected graph $G = (V, E)$ with edge weights w_{ij}, we can formulate the maximum cut problem as a nonlinear integer program using antipodal variables that can assume only the values ± 1. Recall that our objective is to find a cut S that maximizes the total weight of edges between S and $S^C = V \setminus S$. For each vertex v_i in V, we let $x_i = 1$ if $v_i \in S$ or $x_i = -1$ if $v_i \in S^C$. Let n denote the total number of vertices and let w_{ij} represent the weight of the edge connecting v_i and v_j. The Maximum Cut problem can then be written as

$$\begin{aligned} \max \quad & \tfrac{1}{4} \textstyle\sum_{i=1}^{n} \sum_{j=1}^{n} w_{ij}(1 - x_i x_j) \\ \text{s.t.} \quad & x_i \in \{-1, 1\} \qquad\qquad v_i \in V. \end{aligned} \tag{2.2}$$

We can see that if v_i and v_j are on opposite sides of the cut, then $(1 - x_i x_j) = (1 - (-1)) = 2$ and the weight w_{ij} is included in the cut value. If v_i and v_j are on the same side of the cut, then $(1 - x_i x_j) = (1 - 1) = 0$ and $w_{i,j}$ is not included in the cut value.

The maximum cut problem is known to be NP-hard (see section 2.2). By restricting the decision variables to the values ± 1, we are unable to use continuous optimization techniques alone to find an optimal solution.

Relaxations of Optimization Problems In many instances of optimization problems, including Maximum Cut, it is useful to look at a **relaxation**. A relaxation is a reformulation of an optimization problem in which constraints are removed or relaxed to expand the feasible region. Formally, for any optimization problem of the form (2.1), a relaxation is formulated as

$$\begin{aligned} \max \quad & f(\mathbf{x}) \\ \text{subject to} \quad & \mathbf{x} \in \hat{F}, \end{aligned} \tag{2.3}$$

where $\hat{F} \supset F$.

Relaxations are particularly useful because they provide an upper bound on the optimal value of the original problem.

Proposition. *Let* $\mathbf{x}^*$ *be an optimal solution to an optimization problem of the form* (2.1). *Let* $\mathbf{y}^*$ *be an optimal solution to a relaxation of the form* (2.3). *Then,*

$$f(\mathbf{y}^*) \geq f(\mathbf{x}^*).$$

Proof. Because $\mathbf{y}^*$ is optimal for (2.3), we know that $f(\mathbf{y}^*) \geq f(\mathbf{y})$ for all $\mathbf{y} \in \hat{F}$. Given $\mathbf{x}^* \in F$ and $F \subset \hat{F}$, it clearly follows that $\mathbf{x}^* \in \hat{F}$ and $f(\mathbf{y}^*) \geq f(\mathbf{x}^*)$. □

2.1.1 Convex Optimization

We define convex optimization as a specific class of optimization problems involving maximizing a concave objective function over a convex feasible region.

Convex Sets A set $F \subseteq \mathbb{R}^n$ is a **convex set** if for any $\mathbf{x}, \mathbf{y} \in F$, the line segment between $\mathbf{x}$ and $\mathbf{y}$ is contained in F. Formally, F is convex if and only if

$$\lambda\mathbf{x} + (1-\lambda)\mathbf{y} \in F \quad \text{for all } \mathbf{x}, \mathbf{y} \in F \text{ and } \lambda \in [0,1].$$

Convex and Concave Functions A function $f : F \to \mathbb{R}$ is said to be a **convex function** if for any two points $\mathbf{x}, \mathbf{y}$, the graph of f between $\mathbf{x}$ and $\mathbf{y}$ lies below the line segment connecting $f(\mathbf{x})$ and $f(\mathbf{y})$. More precisely, f is convex if

$$f(\lambda\mathbf{x} + (1-\lambda)\mathbf{y}) \leq \lambda f(\mathbf{x}) + (1-\lambda)f(\mathbf{y}) \quad \text{for all } \mathbf{x}, \mathbf{y} \in F, \lambda \in [0,1].$$

A function g is a **concave function** if $-g$ is convex.

We have the following well-known result.

Proposition. *If $\hat{\mathbf{x}}$ is a local maximizer of a concave function g over a convex feasible region F, then $\hat{\mathbf{x}}$ is a global maximizer over F.*

Proof. If $\hat{\mathbf{x}}$ is a local maximizer of g over F, we can equivalently state that $\hat{\mathbf{x}}$ is a local minimizer of $f = -g$, where f is a convex function. Since $\hat{\mathbf{x}}$ is a local minimizer of f, for some $\epsilon > 0$, $f(\mathbf{x}) \geq f(\hat{\mathbf{x}})$ whenever $\mathbf{x} \in F$ and $|\hat{\mathbf{x}} - \mathbf{x}| < \epsilon$. Suppose there exists some $\mathbf{x}^* \in F$ such that $f(\mathbf{x}^*) < f(\hat{\mathbf{x}})$ and define $a = |\hat{\mathbf{x}} - \mathbf{x}^*|$. Let $\mathbf{y} = \frac{\epsilon}{2a}\mathbf{x}^* + (1 - \frac{\epsilon}{2a})\hat{\mathbf{x}}$. Because F is convex, $\mathbf{y} \in F$. From the convexity of f, we can see that

$$f(\mathbf{y}) \leq \frac{\epsilon}{2a} f(\mathbf{x}^*) + (1 - \frac{\epsilon}{2a}) f(\hat{\mathbf{x}}) < \frac{\epsilon}{2a} f(\hat{\mathbf{x}}) + (1 - \frac{\epsilon}{2a}) f(\hat{\mathbf{x}}) = f(\hat{\mathbf{x}}).$$

However,

$$|\mathbf{y} - \hat{\mathbf{x}}| = |(\frac{\epsilon}{2a}\mathbf{x}^* + (1 - \frac{\epsilon}{2a})\hat{\mathbf{x}}) - \hat{\mathbf{x}}| = \frac{\epsilon}{2a}|\hat{\mathbf{x}} - \mathbf{x}^*| = \frac{\epsilon}{2} < \epsilon.$$

We have a contradiction because $\hat{\mathbf{x}}$ is a local minimizer of f, but $\mathbf{y} \in F$ where $f(\mathbf{y}) < f(\hat{\mathbf{x}})$ and $|\hat{\mathbf{x}} - \mathbf{y}| < \epsilon$. Therefore, $\hat{\mathbf{x}}$ must be a global minimizer of f. Equivalently, $\hat{\mathbf{x}}$ must be a global maximizer of g. □

2.1.1.1 Semidefinite Programming

Semidefinite programming is a specific type of convex optimization that involves maximizing or minimizing over a set of **symmetric positive semidefinite** matrices. A matrix $\mathbf{A} \in \mathbb{R}^{n \times n}$ is said to be positive semidefinite (denoted $\mathbf{A} \succeq 0$) if for all $\mathbf{x} \in \mathbb{R}^n$, $\mathbf{x}^T \mathbf{A} \mathbf{x} \geq 0$.

A standard result from linear algebra [4] says that the following statements are equivalent:

- $\mathbf{A} \succeq 0$.
- There exists a matrix $\mathbf{W} \in \mathbb{R}^{n \times n}$ such that $\mathbf{A} = \mathbf{W}^T \mathbf{W}$.

- All eigenvalues of $\mathbf{A}$ are nonnegative.

By this definition, it is *not* necessary for a positive semdefinite matrix to be symmetric. However, in semidefinite programming, we are only concerned with symmetric matrices. The set of all real symmetric $n \times n$ positive semidefinite matrices is denoted S_n^+.

Proposition. *S_n^+ is a convex set.*

Proof. Let $\mathbf{A}, \mathbf{B} \in S_n^+$ where $\mathbf{A} = [a_{ij}]$ and $\mathbf{B} = [b_{ij}]$. Let $\lambda \in [0, 1]$ and consider the matrix

$$\lambda\mathbf{A} + (1 - \lambda)\mathbf{B}.$$

First, we will show that this matrix is symmetric. The (i, j) entry of this matrix is $\lambda a_{ij} + (1 - \lambda)b_{ij}$. Because $\mathbf{A}$ and $\mathbf{B}$ are symmetric, this is equal to $\lambda a_{ji} + (1 - \lambda)b_{ji}$ which is the (j, i) entry. Therefore, the matrix

$$\lambda\mathbf{A} + (1 - \lambda)\mathbf{B}$$

is symmetric.

Next, we will show that the matrix is positive semidefinite. Let $\mathbf{x}$ be any vector in $\mathbb{R}^n$. We can see that

$$\mathbf{x}^T(\lambda\mathbf{A} + (1 - \lambda)\mathbf{B})\mathbf{x} = (\lambda\mathbf{x}^T\mathbf{A} + (1 - \lambda)\mathbf{x}^T\mathbf{B})\mathbf{x} = \lambda(\mathbf{x}^T\mathbf{A}\mathbf{x}) + (1 - \lambda)(\mathbf{x}^T\mathbf{B}\mathbf{x}) \geq 0,$$

since $\mathbf{A}, \mathbf{B} \succeq 0$. Therefore,

$$\lambda\mathbf{A} + (1 - \lambda)\mathbf{B} \succeq 0.$$

Thus,

$$\lambda\mathbf{A} + (1 - \lambda)\mathbf{B} \in S_n^+,$$

and S_n^+ is convex. □

The general form of the **semidefinite program** (SDP) is as follows:

$$\begin{aligned} \max \quad & \langle \mathbf{C}, \mathbf{Y} \rangle \\ \text{subject to} \quad & \langle \mathbf{D}_i, \mathbf{Y} \rangle = d_i, \quad 1 \leq i \leq k \\ & \mathbf{Y} \in S_n^+, \end{aligned}$$

where $\mathbf{C}, \mathbf{D}_i \in \mathbb{R}^{n \times n}$ and $\langle \cdot, \cdot \rangle$ is the Frobenius inner product on $\mathbb{R}^{n \times n}$. The Frobenius inner product is defined to be

$$\langle \mathbf{A}, \mathbf{B} \rangle = Tr(\mathbf{A}^T \mathbf{B}) = \sum_{i=1}^{n} \sum_{j=1}^{n} a_{ij} b_{ij}.$$

Semidefinite Relaxation of Maximum Cut We will derive a semidefinite program relaxation from our original formulation (2.2). For a weighted graph G, the **Laplacian matrix L** is defined to be

$$\ell_{ij} = \begin{cases} -w_{ij} & \text{for } i \neq j, \\ d_i & \text{for } i = j, \end{cases}$$

where $d_i = \sum_{k=1}^{n} w_{ik}$ is the **weighted degree** of the vertex v_i. We can reformulate (2.2) as

$$\begin{aligned} \max \quad & \tfrac{1}{4} \langle \mathbf{L}, \mathbf{x}\mathbf{x}^T \rangle \\ \text{s.t.} \quad & x_i \in \{-1, 1\}. \end{aligned}$$

The matrix $\mathbf{x}\mathbf{x}^T$ is a rank-one symmetric positive semidefinite matrix such that each diagonal entry is 1. If we relax the rank-one constraint, we can formulate a relaxation of Maximum Cut as the semidefinite program,

$$\begin{aligned} \max \quad & \tfrac{1}{4} \langle \mathbf{L}, \mathbf{X} \rangle \\ \text{s.t.} \quad & x_{ii} = 1 \qquad i = 1, \ldots, n \\ & X \in S_n^+. \end{aligned} \tag{2.4}$$

As we will see in Chapter 3, the semidefinite relaxation of (2.2) is the foundation of an important approximation algorithm for Maximum Cut.

2.1.2 Integer Programming

An **integer programming** problem (IP) is a specific type of optimization problem in which the decision variables are restricted to integer values. We define an IP as any problem of the form,

$$\begin{aligned} \max \quad & f(\mathbf{x}) \\ \text{subject to} \quad & \mathbf{x} \in F, \\ & \mathbf{x} \in \mathbb{Z}^n. \end{aligned} \tag{2.5}$$

As we can see from formulation (2.2), the Maximum Cut problem is an IP. Because of the nonconvexity of $\mathbb{Z}^n$, it can be very difficult to find optimal solutions of IP's efficiently. Therefore, heuristics and approximation algorithms are generally used to solve IP's. We describe some of these approaches.

Branch and Bound Branch and Bound is an integer programming technique that finds optimal solutions by solving a series of subproblems. For an integer programming problem of the form (2.5), the feasible region is partitioned into subsets, $F_1, \ldots, F_k$, and we define problem P_i to be

$$\begin{aligned} \max \quad & f(\mathbf{x}) \\ \text{s.t.} \quad & \mathbf{x} \in F_i, \\ & \mathbf{x} \in \mathbb{Z}^n. \end{aligned}$$

The solution $\mathbf{x}_i^*$ with the greatest objective value is an optimal solution. Often, it is necessary to use the same process to further divide the subproblems. This is referred to as the **branching scheme** because the collection of subproblems can be organized as a tree where each subproblem is a **node** on the tree.

Throughout the process, we compute the values of feasible solutions and recognize them as lower bounds on the optimal value. Often, we solve relaxations of P_i to obtain upper bounds on the optimal value of f for the subproblem. If this upper bound is less than our best lower bound, we can eliminate the possibility that an optimal solution of the original problem lies in F_i.

Our implementation of branch and bound for Maximum Cut will be discussed in Chapter 5.

2.1.3 Penalty Methods for Optimization Problems

Consider the general optimization problem,

$$\begin{aligned} \max \quad & f(\mathbf{x}) \\ \text{subject to} \quad & \mathbf{x} \in F. \end{aligned}$$

A **penalty function** for F is a function $P : \mathbb{R}^n \to \mathbb{R}$ such that $P(\mathbf{x}) \geq 0$ for all $\mathbf{x} \in \mathbb{R}^n$ and $P(\mathbf{x}) = 0$ if and only if $\mathbf{x} \in F$ [8]. In penalty methods, we replace the original optimization problem with the problem

$$\begin{aligned} \max \quad & f(\mathbf{x}) - \rho P(\mathbf{x}) \\ \text{subject to} \quad & \mathbf{x} \in \mathbb{R}^n, \end{aligned} \tag{2.6}$$

where ρ is a positive parameter.

In the penalty function method, this optimization problem is repeatedly solved with increasing values of ρ. Intuitively, as ρ approaches infinity, $P(\mathbf{x})$ approaches zero at optimal solutions $\mathbf{x}^*$, implying that $\mathbf{x}^*$ approaches the feasible region F. Precisely, we want to define a penalty function such that

$$\lim_{\rho \to \infty} \max_{\mathbf{x} \in \mathbb{R}^n} \{f(\mathbf{x}) - \rho P(\mathbf{x})\} = \max_{\mathbf{x} \in F} f(\mathbf{x}).$$

A Penalty Function for Maximum Cut Recall that we can formulate the Maximum Cut problem as

$$\begin{aligned} \max \quad & \tfrac{1}{4}\langle \mathbf{L}, \mathbf{X} \rangle \\ \text{s.t.} \quad & x_{ii} = 1 \qquad\qquad i = 1, \ldots, n, \\ & \text{Rank}(\mathbf{X}) = 1, \\ & \mathbf{X} \in S_n^+. \end{aligned} \tag{2.7}$$

Let $\lambda_1(\mathbf{X})$ denote the greatest eigenvalue of $\mathbf{X}$ and let $\|\mathbf{X}\|$ denote the Frobenius norm of $\mathbf{X}$, which is $\|\mathbf{X}\| = \sqrt{\sum_{i=1}^{n} \sum_{j=1}^{n} x_{ij}^2}$. Equivalently, the Frobenius norm can be written as $\|\mathbf{X}\| = \sqrt{\sum_{i=1}^{n} \lambda_i(\mathbf{X})^2}$ [4]. We define our penalty function as

$$P(\mathbf{X}) = \|\mathbf{X}\|^2 - (\max\{0, \lambda_1(\mathbf{X})\})^2. \tag{2.8}$$

Because all rank-one matrices have only one nonzero eigenvalue, this penalty function expresses the distance between $\mathbf{X}$ and the nearest matrix $\mathbf{X}^* \in S_n^+$ with a rank of either zero or one [13].

Clearly we can see that $\|\mathbf{X}\|^2$ is nonnegative because each $\lambda_i^2 \geq 0$. If $\lambda_1 \geq 0$, then $(\max\{0, \lambda_1\})^2 = \lambda_1^2$ and we can express P as

$$P(\mathbf{X}) = \lambda_2(\mathbf{X})^2 + \ldots + \lambda_n(\mathbf{X})^2 \geq 0.$$

Otherwise, $(\max\{0, \lambda_1\})^2 = 0$ and $P(\mathbf{X}) = \|\mathbf{X}\|^2 \geq 0$. Therefore, our function $P(\mathbf{X})$ is nonnegative for any $n \times n$ symmetric matrix.

If $\mathbf{X}$ has a rank of zero, then $\mathbf{X}$ is a matrix of all zeroes and has no nonzero eigenvalues, implying that $P(\mathbf{X}) = 0$. If $\mathbf{X}$ is rank-one and positive semidefinite, then $\lambda_1(\mathbf{X}) > 0$ and $\lambda_k(\mathbf{X}) = 0$ for each $k \in \{2, \ldots, n\}$. This gives $P(\mathbf{X})$ a value of $\lambda_1(\mathbf{X})^2 - \lambda_1(\mathbf{X})^2 = 0$. We also note that when $\text{Rank}(\mathbf{X}) > 1$, at least one of

$\lambda_2, \ldots, \lambda_n$ is nonzero and $P(X) = \sum_{i=2}^{n} \lambda_i^2 > 0$. Therefore, P is a penalty function for Maximum Cut.

We can now replace the original problem with the problem

$$\begin{aligned} \max \quad & \tfrac{1}{4}\langle \mathbf{L}, \mathbf{X} \rangle - \rho(\|\mathbf{X}\|^2 - (\max\{0, \lambda_1(\mathbf{X})\})^2 \\ \text{subject to} \quad & x_{ii} = 1 \qquad\qquad i = 1, \ldots, n, \\ & \mathbf{X} \text{ symmetric.} \end{aligned}$$

The implementation and consequences of our penalization algorithm for Maximum Cut will be discussed in Chapter 4.

2.2 Algorithms and Complexity Classes

When developing algorithms for large scale computational problems, we require tools for effectively analyzing the efficiency of an algorithm. For a given algorithm, $\mathcal{A}$, and instance I of problem $\mathcal{P}$, the **time complexity function** measures the amount of time required by $\mathcal{A}$ to solve I. For a given $n \in \mathbb{Z}_+$, let $\mathcal{P}(n)$ denote problems of size n. We will define the time complexity function $\hat{r}_{\mathcal{A}} : \mathcal{P}(n) \to \mathbb{Z}$ to measure the number of operations ($+, -, \times, \div$, and writing to memory) required by the algorithm for an instance I of size n. We assume the "RAM" model where each operation has an equal cost. We define the **worst-case time complexity function**, $r_{\mathcal{A}}$, as

$$r_{\mathcal{A}}(n) = \sup_{I \in \mathcal{P}(n)} \hat{r}_{\mathcal{A}}(I).$$

Example Consider the following problem: Given any three vectors $\mathbf{x}, \mathbf{y}, \mathbf{z}$ of length n, find the vector $\mathbf{v} = \mathbf{x} + \mathbf{y} + \mathbf{z}$. An algorithm for solving this problem could look like

input : $\mathbf{x}, \mathbf{y}, \mathbf{z} \in \mathbb{R}^n$

output: $\mathbf{v} \in \mathbb{R}^n$

for $i = 1, \ldots, n$ **do**

  $v_i = x_i + y_i + z_i$;

end

We can write the time complexity for this algorithm as

$$\hat{r}_{\mathcal{A}}(n, \mathbf{x}, \mathbf{y}, \mathbf{z}) = \sum_{i=1}^{n} (\lceil \log x_i \rceil + \lceil \log y_i \rceil + \lceil \log z_i \rceil).$$

It is common to use $O(\cdot)$ notation to define a time complexity function.

Definition. *For any instance I of a problem, we say that a function $r(I) = O(g(I))$ if there is some nonnegative constant c such that $r(I) \leq c \cdot g(I)$.*

In the previous example, we can let $C = \max_i\{\max\{x_i, y_i, z_i\}\}$ and write the following inequality:

$$\hat{r}_{\mathcal{A}}(n, \mathbf{x}, \mathbf{y}, \mathbf{z}) = \sum_{i=1}^{n} (\lceil \log x_i \rceil + \lceil \log y_i \rceil + \lceil \log z_i \rceil) \leq 3n \lceil \log C \rceil.$$

By letting $c = 3$, and since $\mathbf{x}, \mathbf{y}, \mathbf{z}$ were arbitrary, $r_{\mathcal{A}}(n, C) = O(n \log C)$. While the use of $O(\cdot)$ notation is not as precise as exact time complexity, it allows us to see how the worst-case time complexity increases as the size of problem instances increase.

Polynomial and Exponential Time Algorithms A **polynomial function** is a function $p_n(x)$ such that $p_n(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0$ where n is a nonnegative finite integer and $a_0, \ldots, a_n$ are constants. An algorithm is said to run in **polynomial time** if its worst-case time complexity function satisfies $r_{\mathcal{A}}(n) = O(p_k(n))$ where $p_k(n)$ is some polynomial function of n. There are three different types of polynomial time algorithms. Consider an algorithm for solving an optimization

problem where n is the number of variables, m is the number of algebraic constraints, and C denotes the size of parameters.

Definition. *An algorithm, $\mathcal{A}$, is* ***strongly polynomial time*** *if $r_{\mathcal{A}}(n, m, C)$ can be bounded by a polynomial function of n and m.*

An algorithm, $\mathcal{A}$, is ***weakly polynomial time*** *if $r_{\mathcal{A}}(n, m, C)$ can be bounded by a polynomial function of n, m, and $\log C$.*

An algorithm, $\mathcal{A}$, is ***pseudo-polynomial time*** *if $r_{\mathcal{A}}(n, m, C)$ can be bounded by a polynomial function of n, m, and C.*

An algorithm whose worst-case does not run in polynomial time is typically said to be an **exponential time algorithm**.

Example We will analyze an enumeration algorithm for Maximum Cut. Let our input be a positively weighted graph $G = (V, E)$ where $|V| = n$. Let $\mathbb{P}(V)$ denote the power set of V. For any $S \subset V$, we will define the function $C : \mathbb{P}(V) \to \mathbb{R}^n$ such that

$$C_i(S) = \begin{cases} 1, & v_i \in S, \\ -1, & \text{otherwise.} \end{cases}$$

For a given cut vector, we define the function $w : \mathbb{R}^n \to \mathbb{R}$ to be

$$w(\mathbf{x}) = \frac{1}{4} \sum_{i=1}^{n} \sum_{j=1}^{n} w_{ij}(1 - x_i x_j)$$

Now we can define our algorithm.

```
input  : A weighted undirected graph G = (V, E).
output: The optimal cut value, best, and optimal cut, Cut_best.
best = 0;
Cut_best = 1;
foreach S ⊂ V do
    if w(C(S)) > best then
        best = w(C(S));
        Cut_best = C(S);
    end
end
```

For each S, the algorithm computes $C(S)$ and then $w(C(S))$. Let $W = \max_{i,j} w_{ij}$. Looking at our definitions for these functions, we see that the time to compute $C(S)$ is $O(n)$ and the time to compute $w(C(S))$ is $O(|E| \log W)$. Given that $|V| = n$, there exist 2^n subsets of V. Therefore, this algorithm executes $O(2^n)$ iterations of the loop. From this we can see that this algorithm runs in exponential time with a time complexity of $r_{\mathcal{A}}(n, |E|, W) = O(2^n |E| \log W)$.

2.2.1 The Complexity Classes P and NP

In order to analyze the relative difficulty of computational problems, theoretical computer scientists have defined several complexity classes. The classes P and NP are used to classify recognition problems in which there is a "yes" or "no" answer for any instance of the problem. For example, we can formulate Maximum Cut as a recognition problem by asking, "for a given weighted undirected graph $G = (V, E)$, is there a cut with a value of at least k?"

Definition. *A recognition problem $\mathcal{P}$ lies in P if there exists a polynomial-time deterministic algorithm that will compute a solution for any instance of $\mathcal{P}$.*

Intuitively, P is the class of problems that are known to be solvable in polynomial time. P is a subset of NP, the class of problems solvable in "non-deterministic polynomial time."

Definition. *A recognition problem $\mathcal{P}$ lies in NP if "yes" solutions to $\mathcal{P}$ can be verified in polynomial time and the length of solutions can be bounded by a polynomial.*

For Maximum Cut, verifying a "yes" solution is equivalent to computing the optimal cut value, which can be done in $O(|E| \log W)$ time. Given that a solution to Maximum Cut is an $n \times 1$ vector of 1's and -1's, the length of a solution is n. Therefore, the recognition variation of Maximum Cut is in NP.

In the NP class there is an important set of problems called NP**-Complete** problems. NP-complete problems have no known polynomial time deterministic algorithms for computing optimal solutions. Further, finding such an algorithm for *any* NP-complete problem would imply that *all* problems in NP can be solved in polynomial time and $P = NP$. It is widely believed, however, that no such algorithms exist.

Another important class of problems is the set of NP**-Hard** problems.

Definition ([1])**.** *A problem $\mathcal{P}$ is said to be NP-hard if and only if a known NP-complete problem $\mathcal{P}_0$ can be reduced to it in polynomial time.*

In other words, given an instance of $\mathcal{P}_0$, an instance of $\mathcal{P}$ can be generated such that a solution to $\mathcal{P}$ can be transformed into a solution to $\mathcal{P}_0$ in polynomial time. Similarly, finding a polynomial time algorithm for an NP-hard problem would imply that $P = NP$.

CHAPTER 3

PREVIOUS APPROACHES

3.1 Greedy Approximation Algorithm

In 1976, Sahni and Gonzalez [12] published an algorithm giving a $\frac{1}{2}$-approximation of the maximum cut of a weighted graph, $G = (V, E)$. The algorithm works as follows:

1. Let $S = \emptyset$ in the initial step.

2. If there is some $v \in S^C$ such that the cut value of $S \cup \{v\}$ is greater than that of S, move the vertex v into S.

3. If there is some $u \in S$ such that the cut value of $S \setminus \{u\}$ is greater than that of S, remove the vertex u from S.

4. Repeat steps 2 and 3 until no further improvements can be made to S, then terminate.

The algorithm will take at most $|E|$ iterations because each iteration must add an edge to the value of the cut and there are at most $|E|$ edges included in the cut. At each iteration, we consider all n vertices and all $|E|$ edges, so the greedy algorithm runs in $O(|E|(n + |E|))$ time. This algorithm is said to be "greedy" because at each step, the algorithm only makes the best local improvement. It is important to note that the algorithm can be initialized from any feasible cut. Because of this, the greedy algorithm is often used in conjunction with other heuristics and approximation algorithms as a local search for improvements.

Theorem (Sahni and Gonzalez [12])**.** *For a given weighted graph $G = (V, E)$, the greedy algorithm returns a cut S that gives a $\frac{1}{2}$-approximation of the optimal cut S^*.*

Proof. For any cut, $B \subset V$, we define the function $w : V \to \mathbb{R}$ to be

$$w(B) = \sum_{i \in B} \sum_{j \notin B} w_{ij} = \sum_{j \notin B} \sum_{i \in B} w_{ji}.$$

If we let $\bar{w}$ be the sum of all weights in G, we can see that

$$\sum_{i \in V} \sum_{j \in V} w_{ij} = 2 \sum_{(i,j) \in E} w_{ij} = 2\bar{w}.$$

Also, at the algorithm's termination, we know that for each $i \in S$,

$$\sum_{j \notin S} w_{ij} \geq \sum_{j \in S} w_{ij},$$

and for each $i \notin S$,

$$\sum_{j \in S} w_{ij} \geq \sum_{j \notin S} w_{ij}.$$

Then, we see that

$$\begin{aligned} 2w(S) = 2 \textstyle\sum_{i \in S} \sum_{j \notin S} w_{ij} &\geq \textstyle\sum_{i \in S} \sum_{j \in S} w_{ij} + \sum_{i \in S} \sum_{j \notin S} \\ &= \textstyle\sum_{i \in S} \sum_{j \in S} w_{ij} + \sum_{j \notin S} \sum_{i \in S} w_{ji} \\ &= \textstyle\sum_{i \in V} \sum_{j \in S} w_{ij}. \end{aligned}$$

Also, we know that

$$2w(S) = 2 \sum_{i \in S} \sum_{j \notin S} w_{ij} \geq \sum_{i \in V} \sum_{j \notin S} w_{ij},$$

so

$$4w(S) \geq \sum_{i \in V} \sum_{j \in V} w_{i,j} = 2\bar{w}.$$

Clearly $\bar{w}$ is an upper bound on $w(S^*)$, so we can say that

$$w(S) \geq \frac{1}{2}\bar{w} \geq \frac{1}{2} w(S^*).$$

□

Implementation In our implementation of the greedy local search, we let $\mathbf{x}$ be a vector of 1's and -1's representing the initial cut S, where $x_i = 1$ if $v_i \in S$ and $x_i = -1$ otherwise. Next, we construct the Gram matrix $\mathbf{X} = \mathbf{x}\mathbf{x}^\mathbf{T}$. Then, we construct a matrix $\bar{\mathbf{W}}$ where $\bar{w}_{ij} = x_{ij} \cdot w_{ij}$. The sum of the i^{th} column of $\bar{\mathbf{W}}$ gives the difference between the weights of uncut edges adjacent to v_i and cut edges adjacent to v_i. Therefore, a positive sum implies that placing v_i in the opposite side of the cut improves the cut value. Let i^* be the index of the column with the largest sum. If $i^* > 0$, we update the cut by letting $x_{i^*} = -x_{i^*}$ and repeat the process. Otherwise, we terminate the algorithm.

3.2 The Goemans-Williamson Randomized Approximation Algorithm

In 1995, Michel Goemans and David Williamson published a randomized approximation algorithm for Maximum Cut that returns solutions with an expected value of at least .87856 times the true optimal cut [3]. To this day, the Goemans-Williamson algorithm gives the best known approximation guarantee for Maximum Cut. It has been proven that if the Unique Games Conjecture holds [6], then it is NP-hard to approximate Maximum Cut with a better guarantee [11].

The Randomized Approximation Algorithm [3] We are given an undirected graph, $G = (V, E)$, where each $(i, j) \in E$ has a weight of w_{ij}. The algorithm begins by solving a relaxation of Maximum Cut,

$$\begin{array}{rll} \max & \frac{1}{4}\sum_{i=1}^{n}\sum_{j=1}^{n} w_{i,j}(1 - \mathbf{y}_i \cdot \mathbf{y}_j) & \\ \text{s.t.} & \|\mathbf{y}_i\| = 1 & \forall v_i \in V, \end{array} \tag{3.1}$$

where each $\mathbf{y}_i$ is a unit vector in $\mathbb{R}^n$. We will show that this vector relaxation can be solved using semidefinite programming.

Theorem. *The vector relaxation,* (3.1)*, is equivalent to the semidefinite relaxation of Maximum Cut,* (2.4)*.*

Proof. Using the properties of positive semidefinite matrices, we can show that each feasible solution to (3.1) corresponds to a feasible solution to the semidefinite program (SDP) with the same objective value.

Let $\mathbf{y}_1, \ldots, \mathbf{y}_n$ be a collection of unit vectors in $\mathbb{R}^n$. Clearly, this collection is feasible for (3.1) with an objective value of

$$\tfrac{1}{4}\sum_{i=1}^{n}\sum_{j=1}^{n} w_{i,j}(1 - \mathbf{y}_i \cdot \mathbf{y}_j).$$

Let $\mathbf{Y}$ be the $n \times n$ matrix whose columns are $\mathbf{y}_1, \ldots, \mathbf{y}_n$, and let $\mathbf{X} = \mathbf{Y^T Y}$. We note that $\mathbf{X}$ is the Gram matrix of $\mathbf{Y}$ and must therefore be positive semidefinite. By construction, the entries of $\mathbf{X}$ will be

$$x_{ij} = \mathbf{y}_i \cdot \mathbf{y}_j,$$

and because each $\mathbf{y}_i$ is a unit vector, we know that

$$x_{ii} = \mathbf{y}_i \cdot \mathbf{y}_i = 1.$$

From the equivalent definitions of a positive semidefinite matrix, we see that $\mathbf{X} \in S_n^+$ and is therefore feasible for the SDP. Simplifying the objective value of $\mathbf{X}$, we observe that

$$\begin{aligned}
\tfrac{1}{4}\langle \mathbf{L}, \mathbf{X} \rangle &= \tfrac{1}{4}\sum_{i=1}^{n}\sum_{j=1} \ell_{i,j}x_{i,j} \\
&= \tfrac{1}{4}\sum_{i=1}^{n} d_i + \sum_{i=1}^{n}\sum_{j=1}^{n} -w_{i,j}x_{i,j} \\
&= \tfrac{1}{4}\sum_{i=1}^{n}\sum_{j=1}^{n} w_{i,j} - w_{i,j}x_{i,j} \\
&= \tfrac{1}{4}\sum_{i=1}^{n}\sum_{j=1}^{n} w_{i,j}(1 - x_{i,j}) \\
&= \tfrac{1}{4}\sum_{i=1}^{n}\sum_{j=1}^{n} w_{i,j}(1 - \mathbf{y}_i \cdot \mathbf{y}_j).
\end{aligned}$$

The second equality follows from the fact that $x_{ii} = 1$ and $\ell_{ii} = d_i$. The third equation follows from the definition $d_i = \sum_{j=1}^{n} w_{ij}$. Observe that the solution to the SDP has the same objective value as the corresponding solution to (3.1).

Similarly, given a feasible solution $\mathbf{X}$ to the SDP, we can use a Cholesky decomposition to factor the $\mathbf{X}$ as $\mathbf{X} = \mathbf{Y^T Y}$. Because $x_{ii} = \mathbf{y}_i \cdot \mathbf{y}_i = 1$ for every i, we have a collection of unit vectors feasible for (3.1) with the same objective value. □

Let $\mathbf{y}_i^*$ for $i = 1, \ldots, n$ denote an optimal collection of vectors to (3.1). Next, $\mathbf{r}$ is chosen to be a uniformly distributed random unit vector in $\mathbb{R}^n$. The cut S is then constructed so that

$$S = \{v_i : \mathbf{y}_i^* \cdot \mathbf{r} \geq 0\}.$$

The vector $\mathbf{r}$ defines a random hyperplane through the origin $\{\mathbf{z} : \mathbf{r}^T \mathbf{z} = 0\}$. For each vector lying above the hyperplane, the corresponding vertex is put into S.

Theorem (Goemans and Williamson [3])**.** *Let S denote the cut returned by the Goemans-Williamson randomized algorithm. The expected value of the cut, $E(w(S))$, is such that*

$$E(w(S)) > .87856 \cdot Opt,$$

where Opt is the maximum cut value for G.

Proof. First, observe that

$$E(w(S)) = \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} w_{ij} \cdot Pr(v_i \text{ and } v_j \text{ lie on opposite sides of the cut } S).$$

The probability of v_i and v_j lying on opposite sides of the cut is equivalent to the probability that the hyperplane defined by $\mathbf{r}$ separates $\mathbf{y}_i^*$ and $\mathbf{y}_j^*$. This probability is directly proportional to the angle between $\mathbf{y}_i^*$ and $\mathbf{y}_j^*$, so

$$Pr(v_i \text{ and } v_j \text{ lie on opposite sides of the cut } S) = \frac{1}{\pi} \arccos(\mathbf{y}_i^* \cdot \mathbf{y}_j^*).$$

The expected value can now be written as

$$E(w(S)) = \frac{1}{\pi} \cdot \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} w_{ij} \arccos(\mathbf{y}_i^* \cdot \mathbf{y}_j^*).$$

Next, since $\|\mathbf{y}_i^*\| = 1$,

$$\frac{\mathbf{y}_i^* \cdot \mathbf{y}_j^*}{\pi} = \frac{2}{\pi} \left(\frac{\arccos(\mathbf{y}_i^* \cdot \mathbf{y}_j^*)}{1 - \mathbf{y}_i^* \cdot \mathbf{y}_j^*} \right) \cdot \frac{1 - (\mathbf{y}_i^* \cdot \mathbf{y}_j^*)}{2} \geq \min_{0<\theta\leq\pi} \left\{ \frac{2}{\pi} \left(\frac{\theta}{1 - \cos\theta} \right) \right\} \cdot \frac{1 - \mathbf{y}_i^* \cdot \mathbf{y}_j^*}{2}.$$

Therefore, by defining

$$\alpha = \min_{0<\theta\leq\pi} \frac{2}{\pi} \cdot \frac{\theta}{1 - \cos(\theta)},$$

the expected value satisfies

$$E(w(S)) \geq \alpha \cdot \frac{1}{4} \sum_{i=1}^{n} \sum_{j=1}^{n} w_{ij}(1 - \mathbf{y}_i^* \cdot \mathbf{y}_j^*).$$

Numerical calculations show that

$$\alpha > .87856.$$

Therefore,

$$E(w(S)) > .87856 \cdot Opt_{vec},$$

where Opt_{vec} is the optimal objective value for (3.1). Given that Opt_{vec} is an upper bound on Opt, it then follows that

$$E(w(S)) > .87856 \cdot Opt.$$

□

Implementation and Analysis To implement the Goemans-Williamson algorithm for some graph $G = (V, E)$, we first solve the SDP relaxation, (2.4), to obtain an optimal matrix $\mathbf{X}^*$. For a given tolerance, ϵ, barrier interior point methods solve the Maximum Cut SDP relaxation in $O(\ln(\frac{1}{\epsilon})n^{3.5})$ time. After factoring $\mathbf{X}^*$ using a Cholesky decomposition, we obtain a collection of n unit vectors and construct S using a uniformly distributed unit vector $\mathbf{r} \in \mathbb{R}^n$. The actual realization of S can be done in $O(n + |E|)$ time. By executing multiple realizations of the random cut S, we can improve the likelihood of being close to the true optimal value. In our implementations, we chose 100 different uniformly distributed vectors and chose S to be the cut giving the highest objective value out of the 100 realized cuts.

While the Goemans-Williamson algorithm finds a cut with a high expected value, the randomized procedure makes it impossible to achieve the .87856 approximation ratio with 100% certainty. Mahajan and Ramesh found a technique for derandomizing the algorithm in $O(n^{30})$ time [9]. While the Unique Games Conjecture [6] suggests that there is no better polynomial time approximation algorithm for Maximum Cut, it is still undetermined whether or not a more efficient deterministic algorithm exists giving the same approximation ratio.

3.3 Branch and Bound for Maximum Cut Using Triangle Inequalities

In 2006, Rendl, Rinaldi, and Wiegele presented a branch and bound algorithm for Maximum Cut using semidefinite programming, triangle inequalities, and the spectral bundle method [2].

Triangle Inequality Constraints At each node of the branch and bound tree, the method given in [2] solves a semidefinite relaxation of Maximum Cut (of form

(2.4)) that is tightened by enforcing the following triangle inequality constraints on $\mathbf{X}$:

$$\begin{pmatrix} -1 & -1 & -1 \\ -1 & 1 & 1 \\ 1 & -1 & 1 \\ 1 & 1 & -1 \end{pmatrix} \begin{pmatrix} x_{ij} \\ x_{ik} \\ x_{jk} \end{pmatrix} \leq \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \end{pmatrix} \quad 1 \leq i < j < k \leq n. \tag{3.2}$$

We abbreviate these constraints as $\mathcal{A}(\mathbf{X}) \leq e$. Using the triangle inequality constraints, (3.2), we can formulate a tighter relaxation of Maximum Cut,

$$\begin{aligned} \max \quad & \langle \mathbf{L}, \mathbf{X} \rangle \\ \text{s.t.} \quad & x_{ii} = 1 \qquad i = 1, \ldots, n, \\ & \mathbf{X} \in S_n^+ \\ & \mathcal{A}(\mathbf{X}) \leq e. \end{aligned} \tag{3.3}$$

By computing tighter upper bounds at each node, the branch and bound algorithm is able to prune more nodes by bounds. While (3.3) gives us a tighter upper bound on integral solutions, it requires the addition of $4\binom{n}{3}$ constraints that increase the dimension of the problem when compared to (2.4). Rather than enforcing all triangle inequalities, the method chooses a subset of these constraints, I, that are likely to be active at an optimal solution $\mathbf{X}^*$. For more details on how I is determined, the reader is referred to [2].

In order to solve (3.3) efficiently, a spectral bundle method, presented in [5], is used rather than a traditional interior point method. With this approach, the most expensive computation is solving the basic semidefinite programming relaxation (2.4). Using these techniques, Rendl, et al, implemented a branch and bound algorithm that found exact solutions for 100 vertex instances of Maximum Cut in under an hour on average. The number of nodes solved in these instances ranged from 51 to 2925.

CHAPTER 4

RANK PENALIZATION

4.1 Rank-One Penalization

Recall from chapter 2 that we defined a penalty function for Maximum Cut as

$$P(\mathbf{X}) = \|\mathbf{X}\|^2 - (\max\{0, \lambda_1(\mathbf{X})\})^2.$$

The motivation for this penalty function comes from the fact that we can write any real symmetric matrix $\mathbf{X}$ in the form

$$\mathbf{X} = \sum_{i=1}^{n} \lambda_i(\mathbf{X}) \mathbf{v}_i \mathbf{v}_i^T$$

where $\lambda_1(\mathbf{X}) \geq \ldots \geq \lambda_n(\mathbf{X})$ are the eigenvalues of $\mathbf{X}$ and $\mathbf{v}_1, \ldots, \mathbf{v}_n$ are associated unit eigenvectors. By penalizing the influence of eigenvalues λ_2 through λ_n, we ensure that the resulting matrix is close to the rank-one matrix $\lambda_1(\mathbf{X})\mathbf{v}_1\mathbf{v}_1^T$.

Implementation We use the rank-one penalty function to construct a feasible solution to Maximum Cut. Given a tolerance $\epsilon > 0$, the algorithm works as follows:

1. Choose initial values ρ and $\mathbf{X}_0$, where $\rho > 0$ and $\mathbf{X}_0$ is a symmetric matrix with ones on the main diagonal.

2. Solve the nonconvex problem

$$\begin{array}{rll} \max & \frac{1}{4}\langle \mathbf{L}, \mathbf{X} \rangle - \rho(\|\mathbf{X}\|^2 - (\max\{0, \lambda_1(\mathbf{X})\})^2) & \\ \text{subject to} & x_{ii} = 1 & i = 1, \ldots, n, \\ & \mathbf{X} \text{ symmetric}, & \end{array} \tag{4.1}$$

   and obtain a local maximizer $\mathbf{X}^*$.

3. If $|n - \lambda_1| < \epsilon$, then round $\mathbf{X}^*$ to a rank-one matrix that is feasible to Maximum Cut by rounding positive entries to 1 and negative entries to -1. Otherwise, let $\rho = 2\rho$ and let $\mathbf{X}_0 = \mathbf{X}^*$ and return to step 2.

The method used for solving this problem searches for a local maximizer and requires a starting point $\mathbf{X}_0$. There will be multiple local maximizers in the feasible region because any feasible cut has an associated local maximizer. Therefore, choosing different starting values can potentially return different values of $\mathbf{X}^*$.

The motivation for the stopping criterion comes from the fact that

$$\sum_{i=1}^{n} \lambda_i(\mathbf{X}) = n$$

for any feasible $\mathbf{X}$. Thus, when $\mathbf{X}$ is rank-one, $\lambda_1 = n$.

Stationarity Conditions Let

$$f(\mathbf{X}) = \frac{1}{4}\langle \mathbf{L}, \mathbf{X} \rangle - \rho(\|\mathbf{X}\|^2 - (\max\{0, \lambda_1(\mathbf{X})\})^2)$$

denote the objective function. Without loss of generality, we can eliminate the $\frac{1}{4}$ constant. At any local maximizer $\mathbf{X}^*$ the Karush-Kuhn-Tucker conditions are

$$\nabla f(\mathbf{X}^*) = \mathbf{L} - 2\rho(\mathbf{X}^* - \lambda_1 v_1 v_1^T) + \Lambda = 0,$$

where Λ is a diagonal matrix that represents the Lagrange multipliers associated with the constraints $x_{ii} = 1$. We use these constraints to solve for Λ:

$$\begin{aligned}
\mathbf{X}^* &= \tfrac{1}{2\rho}\mathbf{L} + \lambda_1 v_1 v_1^T + \tfrac{1}{2\rho}\Lambda \\
\tfrac{1}{2\rho}\Lambda &= I - \operatorname{diag}(\tfrac{1}{2\rho}\mathbf{L} + \lambda_1 v_1 v_1^T) \\
\Lambda &= 2\rho(I - \operatorname{diag}(\lambda_1 v_1 v_1^T)) - \operatorname{diag}(\mathbf{L}).
\end{aligned}$$

Substituting this into the original equation yields

$$\begin{aligned}
&\mathbf{L} - 2\rho(\mathbf{X}^* - \lambda_1 v_1 v_1^T) + 2\rho(I - \operatorname{diag}(\lambda_1 v_1 v_1^T)) + \operatorname{diag}(\mathbf{L}) = 0 \\
&\mathbf{L} = 2\rho(\mathbf{X}^* - \lambda_1 v_1 v_1^T) - 2\rho(I - \operatorname{diag}(\lambda_1 v_1 v_1^T)) + \operatorname{diag}(\mathbf{L}).
\end{aligned}$$

Now take the inner product of both sides with $\mathbf{X}^*$:

$$\langle \mathbf{L}, \mathbf{X}^* \rangle = 2\rho(\|\mathbf{X}^*\|^2 - \lambda_1^2) - 2\rho(n - \lambda_1) + tr(\mathbf{L}),$$
$$\langle \mathbf{L}, \mathbf{X}^* \rangle = 2\rho(\|\mathbf{X}^*\|^2 - \lambda_1^2) + 2\rho(\lambda_1 - n) + 2\bar{w}$$

or

$$\tfrac{1}{4}\langle \mathbf{L}, \mathbf{X}^* \rangle = \tfrac{\rho}{2}(\|\mathbf{X}^*\|^2 - \lambda_1^2) + \tfrac{\rho}{2}(\lambda_1 - n) + \tfrac{\bar{w}}{2} \tag{4.2}$$

where $\bar{w}$ is the sum of all weighted edges of the graph. It is important to note that $\|\mathbf{X}^*\|^2 - \lambda_1^2$ is nonnegative, but $\lambda_1 - n$ can be negative.

Analysis From (4.2), we observe that a local maximizer to (4.1) has a greater objective value when λ_1 is large. Because we know that

$$\sum_{i=1}^{n} \lambda_i = n$$

for any feasible solution, if $\sum_{i=2}^{n} \lambda_i \geq 0$, we have $\lambda_1 \leq n$. Conversely, if $\sum_{i=2}^{n} \lambda_i < 0$, then $\lambda_1 > n$. In this sense, intermediate local solutions that are positive semidefinite are undesirable, given that all eigenvalues are nonnegative.

From this, we found that by starting with small values of ρ, around $\frac{1}{512}$ or $\frac{1}{256}$, initial iterations of the algorithm typically return local solutions with λ_1 values much greater than n. In nearly all cases, these initial values of ρ led to better rank-one solutions than greater initial ρ values did.

In choosing an initial $\mathbf{X}_0$, we found that the optimal solution to the semidefinite relaxation (2.4) consistently led to high-value cuts. While choosing a rank-one matrix as the initial $\mathbf{X}_0$ typically resulted in fewer iterations, the final solutions were typically close to $\mathbf{X}_0$, even when the objective value of the associated cut was far from optimal.

The nonconvexity of (4.1) makes it difficult to exactly predict the behavior of the algorithm; however, the algorithm typically returns cuts comparable to those given by the Goemans-Williamson algorithm (numerical results given at the end of this

chapter). Solutions to (4.1) are computed using the exact Hessian in initial iterations (in $O(n^3)$ time) and the asymptotically exact approximate Hessian in later iterations (in $O(n^2)$ time).

4.2 Penalization for Ranks Greater than One

Recall that our rank-one penalty function is

$$P(\mathbf{X}) = \|\mathbf{X}\|^2 - (\max\{0, \lambda_1(\mathbf{X})\})^2.$$

Suppose that we want to enforce the constraints

$$\begin{aligned} &\text{Rank}(\mathbf{X}) \leq k, \\ &\mathbf{X} \in S_n^+, \end{aligned}$$

where $\mathbf{X} \in \mathbb{R}^{n \times n}$ is the decision variable and k is an integer constant such that $2 \leq k < n$. To enforce these constraints, we can use the penalty function

$$P_k(\mathbf{X}) = \|\mathbf{X}\|^2 - \sum_{i=1}^{k} (\max\{0, \lambda_i(\mathbf{X})\})^2.$$

Similar to rank-one penalization, this penalty function penalizes the influence of the eigenvalues λ_{k+1} through λ_n. After penalization, a local maximizer of P_k will be close to the rank-k matrix $\sum_{i=1}^{k} \lambda_i \mathbf{v}_i \mathbf{v}_i^T$ provided ρ is sufficiently large.

Implementation Using the rank-k penalty function, we can use the following steps to construct a feasible cut:

1. Choose initial values ρ and $\mathbf{X}_0$, where $\rho > 0$ and $\mathbf{X}_0$ is a symmetric matrix with ones on the main diagonal.

2. Solve the nonconvex problem

$$\begin{array}{lll} \max & \frac{1}{4}\langle \mathbf{L}, \mathbf{X}\rangle - \rho(\|\mathbf{X}\|^2 - \sum_{i=1}^{k}(\max\{0, \lambda_1(\mathbf{X})\})^2) & \\ \text{subject to} & x_{ii} = 1 & i = 1, \ldots, n, \\ & \mathbf{X} \text{ symmetric.} & \end{array} \tag{4.3}$$

and obtain a local maximizer $\mathbf{X}^*$.

3. If $|n - \sum_{i=1}^{k} \lambda_k| < \epsilon$, where ϵ is a given tolerance, round $\mathbf{X}^*$ to a rank-k positive semidefinite solution. Otherwise, let $\rho = 2\rho$ and let $\mathbf{X}_0 = \mathbf{X}^*$ and return to step 2.

4. Factor the rank-k positive semidefinite solution using a Cholesky decomposition and use the Goemans-Williamson randomized rounding algorithm to construct a cut S.

When rounding solutions in step 3, we first construct the matrix $\mathbf{W} \in \mathbb{R}^{n\times k}$ where the i^{th} column of W is the vector $\lambda_i \mathbf{v}_i$. After normalizing the k-dimensional row vectors of $\mathbf{W}$, we then define our rank-k positive semidefinite solution to be the Gram matrix $\mathbf{W}\mathbf{W}^T$.

Using the same process as for rank-one penalization, we can derive similar stationarity conditions. For any local maximum $\mathbf{X}^*$, we have that

$$\frac{1}{4}\langle \mathbf{L}, \mathbf{X}^*\rangle = \frac{\rho}{2}(\|\mathbf{X}^*\|^2 - \sum_{i=1}^{k} \lambda_i) + \frac{\rho}{2}((\sum_{i=1}^{k} \lambda_i) - n) + \frac{\bar{w}}{2}.$$

From this, we found that using the same initial conditions as with rank-one penalization gave the most successful results.

Analysis for $k = 2$ When $k = 2$, we can take the final rank-two solution and enumerate all possible partitions given by the Goemans-Williamson algorithm in $O(n)$

time. Our process for enumerating these partitions is based on "Procedure CUT" from [10]. After factoring the solution using a Cholesky decomposition, let $\theta_1, \ldots, \theta_n$ be the directions of each of the n unit vectors acquired. Procedure CUT is then:

input : $\theta_1, \ldots, \theta_n$.

output: Feasible cut S^*.

Let $\alpha = 0$, $V^* = -\infty$, $i = 1$. Let j be the smallest index such that $\theta_j > \pi$, if existent; otherwise let $j = n + 1$. Let $\theta_{n+1} = 2\pi$.

while $\alpha \leq \pi$ **do**

1. Let $\mathbf{r}$ be the unit vector with direction α. Use the Goemans-Williamson rounding method to construct a cut S with objective value V;
2. If $V > V^*$, let $V^* = V$ and $S^* = S$;
3. If $\theta_i \leq \theta_j - \pi$, let $\alpha = \theta_i$ and let $i = i + 1$; otherwise let $\alpha = \theta_j - \pi$ and let $j = j + 1$;

end

Because of our ability to use Procedure CUT on rank-two solutions, we found that using rank-one and rank-two penalization methods were more successful for Maximum Cut than higher-rank penalization methods.

4.3 Numerical Results

The Goemans-Williamson, rank-one penalization, and rank-two penalization algorithms were implemented in MATLAB 7.8 and results were obtained on the College of William and Mary's cluster computing system, SciClone. Tests were run on a Dell PowerEdge SC1435 dual core 2.6 GHz. The SDP relaxation for each problem was solved using SDPT3 version 4.0 [7].

The algorithms were implemented for the following Maximum Cut instances from [14]:

- g05.60.0 - g05.60.9. Randomly generated unweighted graphs with 60 vertices and .5 edge density.

- pw01.100.0 - pw01.100.9. Randomly generated weighted graphs with 100 vertices, .1 edge density, and weights chosen uniformly from $\{1, \ldots, 10\}$.

- pw05.100.0 - pw05.100.9. Randomly generated weighted graphs with 100 vertices, .5 edge density, and weights chosen uniformly from $\{1, \ldots, 10\}$.

- pw09.100.0 - pw09.100.9. Randomly generated weighted graphs with 100 vertices, .9 edge density, and weights chosen uniformly from $\{1, \ldots, 10\}$.

For each graph, the following tables give the true maximum cut value and the objective value of the integral cut obtained from the Goemans-Williamson, rank-one penalization, and rank-two penalization algorithms. The Goemans-Williamson rounding technique was realized 100 times for each instance and the cut with the greatest objective value was returned. The rank-one and rank-two penalization algorithms were initialized with $\rho = \frac{1}{512}$ and $\mathbf{X}_0 = \mathbf{X}^*$ where $\mathbf{X}^*$ is optimal to the Maximum Cut SDP relaxation for the associated graph.

From the given results, it seems that rank-one penalization produces better cuts as the edge density increases. For the pw05 and pw09 graphs, rank-one penalization outperforms the Goemans-Williamson algorithm.

Graph	True Max	G-W	Rank-one	Rank-two
g05.60.0	536	535	534	535
g05.60.1	532	532	531	530
g05.60.2	529	529	524	529
g05.60.3	538	536	536	536
g05.60.4	527	527	526	527
g05.60.5	533	533	531	531
g05.60.6	531	528	526	528
g05.60.7	535	531	531	533
g05.60.8	530	525	527	525
g05.60.9	533	532	528	532
pw01.100.0	2019	1998	1996	1997
pw01.100.1	2060	2052	2055	2055
pw01.100.2	2032	1996	2007	2013
pw01.100.3	2067	2050	2030	2056
pw01.100.4	2039	2020	2015	2025
pw01.100.5	2108	2063	2102	2068
pw01.100.6	2032	2011	2007	2020
pw01.100.7	2074	2060	2053	2040
pw01.100.8	2022	2009	2010	2002
pw01.100.9	2005	1999	1966	1986

Graph	True Max	G-W	Rank-one	Rank-two
pw05.100.0	8190	8129	8161	8156
pw05.100.1	8045	7963	7934	7994
pw05.100.2	8039	7987	7980	7986
pw05.100.3	8139	8087	8136	8136
pw05.100.4	8125	8063	8106	8074
pw05.100.5	8169	8103	8139	8149
pw05.100.6	8217	8135	8176	8159
pw05.100.7	8249	8176	8198	8208
pw05.100.8	8199	8151	8181	8185
pw05.100.9	8099	8062	8082	8080
pw09.100.0	13585	13540	13572	13551
pw09.100.1	13417	13339	13335	13381
pw09.100.2	13461	13380	13389	13446
pw09.100.3	13656	13613	13624	13629
pw09.100.4	13514	13441	13495	13491
pw09.100.5	13574	13543	13546	13558
pw09.100.6	13640	13547	13549	13578
pw09.100.7	13501	13454	13476	13478
pw09.100.8	13593	13545	13553	13558
pw09.100.9	13658	13638	13655	13655

CHAPTER 5

A BRANCH AND BOUND ALGORITHM FOR MAXIMUM CUT

5.1 The Maximum Cut SDP Dual Problem

In our branch and bound algorithm for Maximum Cut, we implement a branching scheme based on dual variables. We first discuss the dual problem for the Maximum Cut SDP and our motivation for implementing a dual branching scheme.

Recall the SDP relaxation for Maximum Cut:

$$\begin{aligned} \max \quad & \tfrac{1}{4}\langle \mathbf{L}, \mathbf{X} \rangle \\ & x_{ii} = 1 \qquad i = 1, \ldots, n \\ & \mathbf{X} \in S_n^+. \end{aligned} \tag{5.1}$$

If $\mathbf{X}$ is a matrix, we define the function $\text{diag}(\mathbf{X})$ to be the main diagonal of $\mathbf{X}$ written as a vector. If $\mathbf{x}$ is a vector, $\text{diag}(\mathbf{x})$ is the square matrix with the entries of $\mathbf{x}$ on the main diagonal, and zeros everywhere else. By introducing the dual variables, $\mathbf{Z} \in \mathbb{R}^{n \times n}$ and $\mathbf{u} \in \mathbb{R}^n$, the Lagrangian for (5.1) is

$$L(\mathbf{X}, \mathbf{Z}, \mathbf{u}) = \langle \mathbf{L}, \mathbf{X} \rangle - \langle \mathbf{Z}, \mathbf{X} \rangle + \mathbf{u}^T(\text{diag}(\mathbf{X}) - \mathbf{1}).$$

Note that without loss of generality, we have disregarded the $\frac{1}{4}$ constant. This gives us the Lagrange dual function,

$$g(\mathbf{Z}, \mathbf{u}) = \max_{\mathbf{X} \succeq 0} (\langle \mathbf{L}, \mathbf{X} \rangle - \langle \mathbf{Z}, \mathbf{X} \rangle + \mathbf{u}^T \text{diag}(\mathbf{X}) - \sum_{i=1}^{n} u_i).$$

By minimizing g, we can now write the Lagrangian dual problem as

$$\begin{aligned} \min \quad & -\textstyle\sum_{i=1}^n u_i \\ \text{s.t.} \quad & \mathbf{L} + \text{diag}(\mathbf{u}) = \mathbf{Z} \\ & \mathbf{Z} \preceq 0, \end{aligned}$$

or equivalently as

$$\begin{aligned} \min \quad & \textstyle\sum_{i=1}^n u_i \\ \text{s.t.} \quad & \text{diag}(\mathbf{u}) \succeq \mathbf{L} \end{aligned} \tag{5.2}$$

Complimentary Slackness Conditions Let $\mathbf{Z}^*, \mathbf{u}^*$ be optimal solutions to (5.2), and let $\mathbf{X}^*$ denote an optimal solution to the primal (5.1). In this case, the optimal objective values of the dual and primal problems are equal, so from complimentary slackness it must hold that $(\mathbf{Z}^*)(\mathbf{X}^*) = \mathbf{0}$, where $\mathbf{0}$ is the matrix of all zeros. Given the dual constraint $\text{diag}(\mathbf{u}^*) - \mathbf{Z}^* = \mathbf{L}$ and the fact that the off-diagonal entries of $\text{diag}(\mathbf{u}^*)$ are zeros, we know that the off-diagonal entries of $\mathbf{Z}^*$ are such that $z_{ij} = w_{ij}$. We can use the fact that the diagonal entries of $\mathbf{X}^*$ are ones to see that

$$z_{ii}^* \cdot 1 + \sum_{j=1}^{n} w_{ij} x_{ij}^* = 0.$$

Thus the diagonal entries of $\mathbf{Z}^*$ are such that $z_{ii}^* = -\sum_{j=1}^{n} w_{ij} x_{ij}^*$.

We can now look at the $\mathbf{u}^*$ variable. From the dual equality constraint, we see that $u_i^* - z_{ii}^* = \ell_{ii}$. Substituting in the value just derived for z_{ii}^*, we see that

$$u_i^* = d_i - \sum_{j=1}^{n} w_{ij} x_{ij}^*.$$

If $\mathbf{X}^*$ is an integral solution for Maximum cut, the entries are all 1's and -1's. In this, the value w_{ij} is added to u_i^* whenever $x_{ij} = -1$ and the edge (i, j) is included in the cut. Otherwise, w_{ij} is subtracted from the degree of node i. In integral cases the value of u_i^* directly reflects the contribution of the vertex v_i to the optimal cut. If $\mathbf{X}^*$ is not an integral solution, the values of u_i^* give us an approximate value for the contribution of the vertex v_i to the optimal cut.

Because the variable u_i^* is a reflection of the contribution of v_i to the optimal cut, our branch and bound algorithm fixes the vertices with higher associated dual variables first. In doing so, we hope that our earlier branching decisions will allow us to quickly approach the optimal cut so that more nodes can be pruned.

5.2 Depth-First Branch and Bound Algorithm

In this section, we describe a depth-first branch and bound algorithm for Maximum Cut. In the initial step of the algorithm, we compute lower bounds of the optimal cut by using the Goemans-Williamson algorithm, rank-one penalization, and rank-two penalization, each followed by a greedy local search. Next, we determine the branching order by computing the optimal dual variables to the SDP relaxation. The indices of the vertices of the graph are then re-indexed so that v_i is the vertex with the i^{th} largest associated dual value u_i^*. We then begin branching from the root node.

5.2.1 Branching

Without loss of generality, we can fix v_1 in the cut set S. This will have no effect on the final result because the weight w_{ij} is included in the cut if $v_i \in S$ and $v_j \in S^C$, or if $v_i \in S^C$ and $v_j \in S$. From the root node, we create two subproblems. In the first subproblem, we fix v_2 in the set S. In the second subproblem, we fix v_2 in S^C.

Collapsed Graph In order to formulate each subproblem, we construct a new graph by collapsing v_2 into v_1 to create the collapsed vertex v_c. In the case of the first subproblem, where $v_1, v_2 \in S$, the edge (v_1, v_2) is deleted because it cannot possibly be included in the value of the cut. We then define the weights of any edges adjacent to v_1 or v_2 such that

$$w_{cj} = w_{1j} + w_{2j}.$$

We then construct the Laplacian matrix of this graph, $\mathbf{L}_{c_1}$ and solve the SDP relaxation at the first child node,

$$\begin{aligned} \max \quad & \tfrac{1}{4}\langle \mathbf{L}_{c_1}, \mathbf{X} \rangle \\ \text{s.t.} \quad & x_{ii} = 1 \qquad i = 1, \ldots, n \\ & \mathbf{X} \in S_n^+. \end{aligned}$$

For the second subproblem, we want to construct a Laplacian matrix, $\mathbf{L}_{c_2}$, so that we can solve an SDP relaxation where we fix $v_1 \in S$ and $v_2 \in S^C$. First, we define the weights of any edges adjacent to v_1 or v_2 such that

$$w_{cj} = w_{1j} - w_{2j}.$$

This means that if $v_c \in S$ and $v_j \in S^C$, the weight w_{1j} is included in the cut, but w_{2j} is not. Similarly, if $v_c \in S$ and $v_j \in S$, the weight w_{2j} is included in the cut, but w_{1j} is not. The weight of the cut edge (v_1, v_2), if it exists, is reflected in $\mathbf{L}_{c_2}$ by defining

$$\ell_{c_2 11} = d_1 + d_2 + 2w_{12}.$$

We can now solve the SDP relaxation at the second child node,

$$\begin{aligned} \max \quad & \tfrac{1}{4}\langle \mathbf{L}_{c_2}, \mathbf{X} \rangle \\ \text{s.t.} \quad & x_{ii} = 1 \qquad i = 1, \ldots, n \\ & \mathbf{X} \in S_n^+. \end{aligned}$$

After solving both relaxations and computing their objective values, we treat the child node with the highest objective value as the new root node and repeat the branching process iteratively. Because the size of the Laplacian is reduced at each iteration, the size of the subproblems decreases at each level and solutions are computed more quickly.

At each node of the branch and bound tree, we implement the Goemans-Williamson rounding, rank-one penalization, and rank-two penalization, along with a greedy local search, to attempt to improve the best lower bound.

5.2.2 Pruning

The advantage of the branch and bound algorithm comes from our ability to eliminate, or prune, subproblems as we traverse the branch and bound tree. We can eliminate these problems by either pruning by bounds, or pruning by optimality.

Pruning by Bounds Suppose for some subproblem we have fixed the vertices $v_{i_1}, \ldots, v_{i_k} \in S$ and $v_{i_{k+1}}, \ldots, v_{i_\ell} \in S^C$. We know that the optimal objective value of the SDP relaxation at that node is an upper bound on all cuts such that $v_{i_1}, \ldots, v_{i_k} \in S$ and $v_{i_{k+1}}, \ldots, v_{i_\ell} \in S^C$. If the optimal objective value of this SDP is less than our best lower bound, we know that none of these cuts can be the optimal cut. Therefore, we can prune this node by bounds and do not need to branch any further from the node.

Pruning by Optimality Suppose for some subproblem we have fixed the vertices $v_{i_1}, \ldots, v_{i_k} \in S$ and $v_{i_{k+1}}, \ldots, v_\ell \in S^C$. Suppose that the optimal solution to the SDP relaxation at this node, $\mathbf{X}^*$, is rank-one. We know that the objective value of $\mathbf{X}^*$ is an upper bound on all cuts such that $v_{i_1}, \ldots, v_{i_k} \in S$ and $v_{i_{k+1}}, \ldots, v_\ell \in S^C$. We also know that $\mathbf{X}^*$ is feasible for Maximum Cut, and is therefore a lower bound on all such cuts. Because none of these cuts can have a higher objective value, we have found the optimal integral solution to the subproblem. Therefore, we can prune this node by optimality and do not need to branch any further from the node. Further, if the objective value of $\mathbf{X}^*$ is greater than our best lower bound, we can update our lower bound.

When we reach a pruned node of the branch and bound tree, we move back up the tree to the nearest node that has not been pruned and continue branching. The algorithm terminates when all nodes have either been pruned or solved and returns the optimal cut S^* for G.

5.3 Numerical Results

Our branch and bound algorithm was implemented in MATLAB 7.8 and results were obtained on SciClone. Tests were run on a Dell PowerEdge SC1435 dual core 2.6 GHz. The SDP relaxation for each problem was solved using SDPT3 version 4.0 [7].

The algorithms were implemented for the following Maximum Cut instances from [14]:

- g05.60.0 - g05.60.9. Randomly generated unweighted graphs with 60 vertices and .5 edge density.
- pw05.60.0 - pw05.60.9. Randomly generated weighted graphs with 60 vertices, .5 edge density, and weights chosen uniformly from $\{1, \ldots, 10\}$.
- pw09.60.0 - pw09.60.9. Randomly generated weighted graphs with 60 vertices, .9 edge density, and weights chosen uniformly from $\{1, \ldots, 10\}$.

For each graph, the following table gives the number of nodes solved prior to the termination of the branch and bound algorithm for three different branching schemes. First, we implemented a random branching scheme. Next, we defined our branching order so that we fixed higher degree vertices first. Finally, we implemented our dual branching scheme. At each node, rank-one penalization, rank-two penalization, and 100 realizations of the Goemans-Williamson rounding scheme were implemented, along with a greedy local search, in an attempt to improve lower bounds. The rank-one and rank-two penalization algorithms were initialized with $\rho = \frac{1}{512}$ and $\mathbf{X}_0 = \mathbf{X}^*$ where $\mathbf{X}^*$ is optimal to the Maximum Cut SDP relaxation for the associated graph. A "DNF" indicates that the branch and bound algorithm did not finish in the 40 hours allowed by SciClone's job scheduler.

In the graphs tested, branch and bound using a dual branching scheme consistently ran more efficiently than branch and bound using highest-degree or randomized branching.

Graph	Random	Degree	Dual
g05.60.0	4988	4236	2680
g05.60.1	3516	2404	1450
g05.60.2	12908	12908	7894
g05.60.3	1374	976	582
g05.60.4	DNF	DNF	DNF
g05.60.5	1650	1176	1136
g05.60.6	DNF	11490	10076
g05.60.7	8330	11514	10726
g05.60.8	DNF	DNF	8960
g05.60.9	13538	DNF	13538

Graph	Random	Degree	Dual
pw05.60.0	3966	1736	892
pw05.60.1	5000	1978	1596
pw05.60.2	11044	4490	3386
pw05.60.3	6198	4474	3132
pw05.60.4	1218	520	356
pw05.60.5	7948	4188	3682
pw05.60.6	4618	3390	1758
pw05.60.7	DNF	12618	11170
pw05.60.8	12462	8330	2992
pw05.60.9	1842	1464	674

Graph	Random	Degree	Dual
pw09.60.0	DNF	DNF	DNF
pw09.60.1	DNF	13624	10422
pw09.60.2	2172	1932	780
pw09.60.3	8780	7738	4160
pw09.60.4	6894	9622	6342
pw09.60.5	12052	11846	8324
pw09.60.6	14442	DNF	DNF
pw09.60.7	5820	2950	1882
pw09.60.8	2728	3584	2090
pw09.60.9	3940	4982	3928

CHAPTER 6

CONCLUSION

This paper focused on computational approaches to the Maximum Cut problem. We discussed several previous approaches to the problem and introduced a rank-penalization heuristic. We then implemented a branch and bound algorithm for Maximum Cut using a dual branching scheme and used both rank-penalization heuristics and the Goemans-Williamson approximation algorithm [3] to compute lower bounds.

There are still several unanswered questions surrounding our rank-penalization heuristic. In future research, we would hope to find some initial conditions that would give us an approximation guarantee for Maximum Cut. Similarly, we would also hope to tweak the algorithm to ensure that we do not end up at a bad local maximizer of (4.1) that is far from optimal. Our results suggest that rank-one penalization is more successful for dense graphs, but we hope to run further tests to confirm this.

While our dual branching scheme for branch and bound seemed to perform consistently better than random or weighted-degree branching schemes, we hope to prove theoretical results involving dual branching. In particular, we hope to show that given dual branching, after solving (or pruning) all nodes on the k^{th} level of the branch and bound tree, we have a

$$\frac{k + \alpha(n-k)}{n}$$

approximation of the maximum cut where α is a constant between zero and one.

Ultimately, we hope to find new techniques and theoretical results in order to implement an even more efficient branch and bound algorithm for Maximum Cut. In future implementations, we hope to utilize the triangle inequality constraints and bundle methods presented in [2].

BIBLIOGRAPHY

[1] Bertsimas, D., and Tsitsiklis, J. *Introduction to Linear Optimization.* Athena Scientific and Dynamic Ideas, LLC, Belmont, 1997.

[2] F. Rendl, G. Rinaldi, and Wiegele, A. A branch and bound algorithm for max-cut based on combining semidefinite and polyhedral relaxations.

[3] Goemans, M. X., and Williamson, D. P. Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming. *Journal of the Association for Computing Machinery 42*, 6 (1995), 1115–1145.

[4] Horn, R. A., and Johnson, C. R. *Matrix Analysis.* Cambridge University Press, New York, 1985.

[5] I. Fischer, G. Gruber, F. Rendl, and Sotirov, R. Computational experience with a bundle approach for semidefinite cutting plane relaxations of max-cut and equipartition. *Math. Programming 105* (2006), 451–469.

[6] Khot, S. On the power of unique 2-prover 1-round games. *STOC '02: Proceedings of the thiry-fourth annual ACM symposium on Theory of computing* (2002), 767–775.

[7] Kim-Chuan Toh, M. J. Todd, and Tutuncu, R. H. Sdpt3 version 4.0 - a matlab software for semidefinite-quadratic-linear programming. http://www.math.nus.edu.sg/mattohkc/sdpt3.html, 2009.

[8] Luenberger, D. G. *Linear and Nonlinear Programming*, second ed. Addison-Wesley, Menlo Park, 1984.

[9] Mahajan, Sanjeev, and Ramesh, H. Derandomizing approximation algorithms based on semidefinite programming. *SIAM Journal of Computing 28*, 5 (1999), 1641–1663.

[10] S. Burer, R. D. C. Monteiro, and Zhang, Yin. Rank-two relaxation heuristics for max-cut and other binary quadratic programs. *SIAM Journal of Optimization 12*, 2 (2001), 503–521.

[11] S. Khot, G. Kindler, E. Mossel, and O'Donnell, R. Optimal inapproximability results for max-cut and other 2-variable csps. *Electronic Colloquium on Computational Complexity*, 101 (2005).

[12] Sahni, S., and Gonzalez, T. P-complete approximation problems. *Journal of the Association for Computing Machinery 23*, 3 (1976), 555–565.

[13] Torgerson, W. S. Multidimensional scaling: I. Theory and method. *Psychometrika 17*, 4 (December 1952), 401–419.

[14] Wiegele, A. A collection of some max cut and quadratic 0-1 programming instances, 2006.