



DEGREE PROJECT IN COMPUTER SCIENCE AND ENGINEERING,
SECOND CYCLE, 30 CREDITS
STOCKHOLM, SWEDEN 2018

Improved inapproximability of Max-Cut through Min-Cut

MÅRTEN WIMAN

Improved inapproximability of Max-Cut through Min-Cut

MÅRTEN WIMAN

Master in Computer Science

Date: June 15, 2018

Supervisor: Per Austrin

Examiner: Johan Håstad

Swedish title: Förbättrad ickeapproximerbarhet för Max-Cut genom
Min-Cut

School of Electrical Engineering and Computer Science

Abstract

A *cut* is a partition of a graph's nodes into two sets, and we say that an edge *crosses* the cut if it connects two nodes belonging to different sets. A *maximum cut* is a cut that maximises the number of crossing edges.

We show that for any sufficiently small $\varepsilon > 0$ it is **NP**-hard to distinguish between graphs for which at least a fraction $1 - \varepsilon$ of all edges crosses the maximum cut and graphs for which at most a fraction $1 - 1.4568\varepsilon$ of all edges crosses the maximum cut. The previous state of the art had a constant smaller than 1.375 in place of 1.4568.

Sammanfattning

Ett *snitt* är en partition av en grafs noder i två mängder, och vi säger att en kant *korsar* snittet om dess ändpunkter tillhör olika mängder. Ett *maximalt snitt* är ett snitt som maximerar antalet kanter som korsar snittet.

Vi bevisar att det för alla tillräckligt små konstanter $\varepsilon > 0$ är NP-svårt att skilja mellan grafer för vilka minst en andel $1 - \varepsilon$ av alla kanter korsar det maximala snittet och grafer för vilka högst en andel $1 - 1.4568\varepsilon$ av alla kanter korsar det maximala snittet. Detta är en förbättring jämfört med ett tidigare resultat som hade en konstant mindre än 1.375 istället för 1.4568.

Contents

1	Introduction	1
1.1	Related problems	3
2	Preliminaries	5
2.1	Constraint satisfaction problems	5
2.2	Hadamard predicates	6
2.3	Reduction from Max-Had _k to Max-Cut	8
3	Results	20
3.1	Relaxation of constraints	20
3.2	Gadget optimisation	23
3.3	Exploiting the symmetries	28
3.3.1	The symmetry group	28
3.3.2	Optimality of symmetric gadgets	30
3.3.3	Evaluation of symmetric gadgets	34
3.4	A good Had ₄ -gadget	34
4	Conclusion	37
4.1	Future work	37
	Bibliography	40

Chapter 1

Introduction

The well-known complexity class **NP** contains many examples of problems not believed to be efficiently solvable. However, some of these problems are not only hard to solve exactly, but also to solve approximately.

The perhaps simplest example is the Max-Cut problem, in which the goal is to partition the nodes in a given graph into two sets in such a way that as many as possible of the graph's edges connect nodes in different sets.

Problem 1.1 (Max-Cut). Given a graph $G = (V, E)$, a *cut* is a partition of the nodes V into two sets S and T . The *relative cut size* is the fraction of all edges $(u, v) \in E$ for which u and v belong to different sets. The *maximum cut* problem is the problem of computing the maximum relative size of any cut.

It was proven by Karp [9] that Max-Cut is **NP**-hard, which means that if there exists an efficient algorithm solving Max-Cut exactly then any problem whose solution can be efficiently verified can also be efficiently solved.

However, even approximating Max-Cut turns out to be **NP**-hard. Håstad [7] and Trevisan et al. [12] showed that it is **NP**-hard to approximate the size of the maximum cut of an arbitrary graph within a factor $\frac{16}{17} + \varepsilon$. This stands in contrast to the best known polynomial time approximation algorithm, due to Goemans and Williamson [5], which produces cuts of size at least 0.87856 times the optimum.

For graphs that are almost bipartite in the sense that the maximum relative cut size is close to 1, the Goemans-Williamson algorithm has

better approximation guarantees. Indeed, if the maximum relative cut size is $1 - \varepsilon$ then the Goemans-Williamson algorithm produces a cut of relative size at least $1 - \frac{2}{\pi}\sqrt{\varepsilon} - o(\varepsilon)$.

On the hardness side, Håstad et al. [8] showed that for any sufficiently small $\varepsilon > 0$ and any constant $C < 11/8 = 1.375$ it is NP-hard to distinguish between the following two types of graphs:

- Graphs with maximum relative cut size at least $1 - \varepsilon$.
- Graphs with maximum relative cut size at most $1 - C\varepsilon$.

In this thesis we improve on the work of Håstad et al. [8] by demonstrating a relatively efficient algorithmic way of proving inapproximability results for Max-Cut. The proposed method allows us to analyse more complex reductions than those considered by Håstad et al. [8], and we are thus able to improve their result by replacing the upper bound $C < 1.375$ with $C \leq 1.4568$.

Theorem 1.2. *For all sufficiently small constants $\varepsilon > 0$ it is NP-hard to distinguish between graphs with maximum relative cut size at least $1 - \varepsilon$ and graphs with maximum relative cut size at most $1 - 1.4568\varepsilon$.*

To prove hardness of approximating Max-Cut we use *gadgets*, which help us describe reductions to Max-Cut from a problem known as Max-Had_k. Different gadgets yield different factors of inapproximability, and determining how good a gadget is requires solving a certain graph problem which unfortunately seems to be hard to solve efficiently. Our main contribution is the observation that it is possible to prove lower bounds on the performance of gadgets by solving a much simpler problem known as *Min-Cut*.

Problem 1.3 (Min-Cut). The *minimum cut* problem is the same as the Max-Cut problem, except that instead of trying to *maximise* the relative cut size the goal is to *minimise* it and there may be some nodes that are required to belong to different sets.

Unlike the Max-Cut problem, the Min-Cut problem can easily be solved in polynomial time, which makes it possible for us to analyse larger gadgets than we would otherwise be able to.

However, in order to derive good inapproximability results, we do not only have to be able to evaluate gadgets but we also have to come up with gadgets to evaluate. We automate this process through an

algorithm capable of efficiently finding the gadget which maximises the performance lower bound mentioned above.

Finally, we present an optimisation which we estimate decreases the time needed for our computer search by almost a factor 10 000.

1.1 Related problems

An interesting generalisation of both Min-Cut and Max-Cut is the Max-2-Lin(2) problem. In the Max-2-Lin(2) problem, some of the edges in the graph are maximisation edges while others are minimisation edges, and the goal of the problem is to maximise the number of cut maximisation edges minus the number of cut minimisation edges. We can also formulate this using linear equations, as follows:

Problem 1.4 (Max-2-Lin(2)). Given a system of linear equations modulo 2 where each equation contains exactly two variables,

$$x_i + x_j = b \pmod{2},$$

the Max-2-Lin(2) problem consists of finding an assignment to the variables such that the number of satisfied equations is maximised.

The variables x_i represent which set each node belongs to, while the equations with $b = 1$ represent maximisation edges and equations with $b = 0$ represent minimisation edges.

Because the Max-2-Lin(2) problem is a generalisation of Max-Cut and Min-Cut, it is at least as hard as each of those problems. Therefore our inapproximability result for Max-Cut also holds for Max-2-Lin(2).

Theorem 1.5. *For all sufficiently small constants $\varepsilon > 0$ it is NP-hard to distinguish between Max-2-Lin(2) instances for which there exists an assignment satisfying at least a fraction $1 - \varepsilon$ of the equations and instances for which no assignment satisfies more than a fraction $1 - 1.4568\varepsilon$ of the equations.*

The Max-2-Lin(2) problem can be further generalised by changing the modulo from 2 to an arbitrary prime p , in which case we get the Max-2-Lin(p) problem. Max-2-Lin(p) is interesting because of its relation to Khot's [10] Unique Games Conjecture. Khot et al. [11] showed that the Unique Games Conjecture is true if and only if it is NP-hard

to distinguish between $\text{Max-2-Lin}(p)$ instances where more than a fraction $1 - \varepsilon$ of the equations can be satisfied and instances where no more than a fraction ε of the equations can be satisfied, when p is sufficiently large as a function of ε .

Khot et al. [11] also showed that if the Unique Games Conjecture is true then the Goemans-Williamson algorithm for Max-Cut is essentially optimal, and in particular it holds for all sufficiently small $\varepsilon > 0$ that it is NP-hard to distinguish graphs with maximum relative cut size at least $1 - \varepsilon$ from graphs with maximum relative cut size at most $1 - \frac{2}{\pi}\sqrt{\varepsilon}$, which would be an improvement compared to Theorem 1.2. However, the Unique Games Conjecture is somewhat controversial, and it is therefore worthwhile to study unconditional NP-inapproximability of Max-Cut.

Chapter 2

Preliminaries

In this chapter we provide some background that is needed for our reduction. We start by describing constraint satisfaction problems, which comprise not only the Max-Cut and Max-2-Lin(2) problems but also the Max-Had_k problem which is very important to us because our inapproximability results are based on reductions from the Max-Had_k problem.

The Max-Had_k problem is defined in Section 2.2, and in Section 2.3 we describe how *gadgets* can be used for reductions from Max-Had_k to Max-Cut.

2.1 Constraint satisfaction problems

A constraint satisfaction problem (CSP) consists of a set of variables and a set of constraints on the variables, and the goal is to find an assignment to the variables that maximises the number of satisfied constraints.

More formally, for any predicate $\phi: \{-1, 1\}^n \rightarrow \{0, 1\}$ an instance $\mathcal{I}$ of the Max- ϕ CSP consists of a set V of variables and a distribution of constraints, where each constraint $\mathcal{C}$ consists of an n -subset $\{x_1, x_2, \dots, x_n\}$ of V and a sign $\sigma_i \in \{-1, 1\}$ for each variable x_i . An assignment $A: V \rightarrow \{-1, 1\}$ is said to *satisfy* a constraint $\mathcal{C}$ if

$$\phi(\sigma_1 A(x_1), \dots, \sigma_n A(x_n)) = 1.$$

The *value* $\text{val}_{\mathcal{I}}(A)$ of an assignment A is the probability that it satisfies a random constraint sampled from $\mathcal{I}$,

$$\text{val}_{\mathcal{I}}(A) = \Pr_{\mathcal{C} \sim \mathcal{I}}[A \text{ satisfies } \mathcal{C}],$$

and the value of an instance $\mathcal{I}$ is the maximum value of any assignment,

$$\text{val}(\mathcal{I}) = \max_A \{\text{val}_{\mathcal{I}}(A)\}.$$

In the Max- ϕ constraint satisfaction problem, the objective is to determine $\text{val}(\mathcal{I})$.

Example 2.1. Max-2-Lin(2) can be modelled as a constraint satisfaction problem by letting $\phi: \{-1, 1\}^2 \rightarrow \{0, 1\}$ be the equality predicate

$$\phi(x, y) = \begin{cases} 1 & \text{if } x = y, \\ 0 & \text{if } x \neq y. \end{cases}$$

For equations of the form $x_i + x_j = 0 \pmod{2}$ we let $\sigma_1 = \sigma_2$ and for equations of the form $x_i + x_j = 1 \pmod{2}$ we let $\sigma_1 \neq \sigma_2$.

Example 2.2. Max-Cut is a special case of Max-2-Lin(2) where all constraints have $\sigma_1 \neq \sigma_2$. Given a graph G , each node in the graph is represented by a variable in V , and the two sets are represented by the numbers -1 and 1 . A constraint consists of a uniformly random edge (u, v) in G , and the constraint is satisfied if

$$\phi(A(u), -A(v)) = 1,$$

where $\phi: \{-1, 1\}^2 \rightarrow \{0, 1\}$ is the equality predicate. The value of the instance is equal to the maximum relative cut size of G defined in Definition 1.1.

Remark 2.3. The version of Max-Cut described above is known as *unweighted* Max-Cut because the constraints are sampled uniformly among all edges in the graph. In the remainder of this thesis we will focus on *weighted* Max-Cut, where the probabilities may be different for different edges. This is not a cause of concern because unweighted Max-Cut and weighted Max-Cut are equally hard to approximate, see [3].

2.2 Hadamard predicates

Our goal is to prove that it is NP-hard to approximate Max-Cut well, and we do this using a reduction from another problem which has

already been shown to be NP-hard. The problem from which we reduce is a constraint satisfaction problem based on *Hadamard predicates*, which are defined on hypercubes and only evaluate to 1 on a relatively small set of inputs.

Before we define the Hadamard predicates, let us first introduce some notation for dealing with hypercubes.

Definition 2.4. For any non-negative integer k , we let $V_k = \{-1, 1\}^{2^k}$ denote the set of vertices in the 2^k -dimensional hypercube, and we let E_k denote the set of edges in the 2^k -dimensional hypercube.

Rather than indexing the coordinates of a node x by the numbers $1, \dots, 2^k$, we instead index them by subsets of $[k]$. For example, for a node $x \in V_2$ we write

$$x = (x_\emptyset, x_{\{1\}}, x_{\{2\}}, x_{\{1,2\}}),$$

rather than $x = (x_1, x_2, x_3, x_4)$. This convention makes the definition of the set of nodes satisfying the Hadamard predicate less cumbersome.

Definition 2.5. For any non-negative integer k and any set $S \subseteq [k]$, let $\chi_S \in V_k$ denote the node such that for all $T \subseteq [k]$, the coordinate $(\chi_S)_T$ satisfies

$$(\chi_S)_T = (-1)^{|S \cap T|}.$$

Example 2.6. If $k = 2$ then

$$\begin{aligned} \chi_\emptyset &= (1, 1, 1, 1), & \chi_{\{1\}} &= (1, -1, 1, -1), \\ \chi_{\{2\}} &= (1, 1, -1, -1), & \chi_{\{1,2\}} &= (1, -1, -1, 1). \end{aligned}$$

The Hadamard predicate is usually defined so that the nodes χ_S are the only nodes that satisfy it, but those nodes are distributed somewhat asymmetrically because $(\chi_S)_\emptyset = 1$ for all S . To make the set more symmetric we introduce a new set which also contains all nodes that are antipodal to the nodes χ_S .

Definition 2.7. For any non-negative integer k , let

$$Z_k = \{ \pm \chi_S \mid S \subseteq [k] \}.$$

We then define the Hadamard predicate to be satisfied for all nodes in Z_k , but not for any other nodes.

Definition 2.8. The 2^k -ary Hadamard predicate $\text{Had}_k: V_k \rightarrow \{0, 1\}$ is defined by

$$\text{Had}_k(x) = \begin{cases} 1 & \text{if } x \in Z_k, \\ 0 & \text{otherwise.} \end{cases}$$

It turns out that the Max-Had_k CSP is very hard to approximate. In fact, Chan [2] proved that it is NP-hard to distinguish between instances that admit almost perfect solutions and instances for which all assignments make the vector sampled by the constraints look almost uniformly random.

Theorem 2.9 (Chan [2]). *Fix a real constant $\varepsilon > 0$ and an integer $k \geq 3$. For an instance $\mathcal{I}$ of the Max-Had_k CSP it is NP-hard to distinguish between the following two cases:*

- (Completeness) $\text{val}(\mathcal{I}) \geq 1 - \varepsilon$.
- (Soundness) *For every assignment A and every vector $\alpha \in V_k$ it holds that if we sample a constraint $\mathcal{C} = ((\sigma_\emptyset, x_\emptyset), \dots, (\sigma_{[k]}, x_{[k]}))$ from $\mathcal{I}$ then*

$$\left| \Pr_{\mathcal{C} \sim \mathcal{I}}[(\sigma_\emptyset A(x_\emptyset), \dots, \sigma_{[k]} A(x_{[k]})) = \alpha] - 2^{-2^k} \right| \leq \varepsilon.$$

Remark 2.10. Note that in Theorem 2.9, $x_\emptyset, \dots, x_{[k]}$ do not refer to coordinates of some node x , but instead they refer to different variables in V , like in Section 2.1.

A predicate for which it is NP-hard to distinguish between the two cases described above is known as a *useless* predicate, as defined by Austrin and Håstad [1]. The fact that the Hadamard predicates are useless is precisely what makes them so useful for reductions, and in the next section we describe how this uselessness can be exploited to show inapproximability results for Max-Cut.

2.3 Reduction from Max-Had_k to Max-Cut

In this section we describe a reduction from Max-Had_k to Max-Cut, closely following the work of Håstad et al. [8].

The reductions we consider are based on *gadgets*, which are functions that map each constraint in an instance of one problem to a distribution over constraints in another problem. In our case the gadget will

map each constraint in a Max-Had_k instance to a distribution over constraints in a Max-Cut instance. In this way, a gadget and a Max-Had_k instance together define an instance of the Max-Cut problem, and our aim is to prove that if there exists an efficient algorithm that approximates Max-Cut well then it is possible to use the gadget reduction to distinguish between the two cases in Theorem 2.9, which is a contradiction unless $P = NP$.

The variable set in our Max-Cut instance contains all variables that are used in the Max-Had_k instance, but we also use some additional variables. Specifically, for each Max-Had_k constraint $\mathcal{C} = (\sigma_S, x_S)_{S \subseteq [k]}$ we introduce a variable $y_v^{\mathcal{C}}$ for every node $v \in V_k$. We identify the variable $y_{\sigma_S \chi_S}^{\mathcal{C}}$ with x_S for each $S \subseteq [k]$, which means that all constraints involving the variable x_S will use the same instance of that variable, thus forcing it to take the same value every time it appears.

We refer to the variables $y_{\sigma_S \chi_S}^{\mathcal{C}}$ as *primary variables* and to all other variables as *auxiliary variables*. Note that there are only 2^k primary variables per constraint $\mathcal{C}$, while there are $2^{2^k} - 2^k$ auxiliary variables. The primary variables satisfy the Hadamard predicate if

$$\text{Had}_k(\sigma_{\emptyset} y_{\sigma_{\emptyset} \chi_{\emptyset}}^{\mathcal{C}}, \dots, \sigma_{[k]} y_{\sigma_{[k]} \chi_{[k]}}^{\mathcal{C}}) = 1,$$

which is equivalent to

$$(\sigma_{\emptyset} y_{\sigma_{\emptyset} \chi_{\emptyset}}^{\mathcal{C}}, \dots, \sigma_{[k]} y_{\sigma_{[k]} \chi_{[k]}}^{\mathcal{C}}) \in Z_k.$$

We want to construct a distribution over Max-Cut constraints (i.e. edges) on these variables such that

- If there exists an assignment to the primary variables such that the Hadamard predicate is satisfied for a fraction at least $1 - \varepsilon$ of the constraints, then that assignment can be extended to all variables in such a way that a large fraction of the Max-Cut constraints are satisfied.
- If all assignments to the primary variables make the input to the Hadamard predicate look essentially random then no assignment satisfies a very large fraction of the Max-Cut constraints.

Our constraint distribution will be based on a probability distribution over E_k , the edges in the 2^k -dimensional hypercube. Because it corresponds to a distribution over Max-Cut constraints, we will refer to this edge distribution as a gadget.

Definition 2.11. A Had_k -gadget is a function $\mathcal{G}: E_k \rightarrow \mathbb{R}^{\geq 0}$ such that

$$\sum_{e \in E_k} \mathcal{G}(e) = 1.$$

Equivalently, a Had_k -gadget can be defined as a probability distribution over E_k .

To sample a Max-Cut constraint, we will first sample a constraint $\mathcal{C}$ from the Max-Had_k instance and an edge (u, v) from the gadget $\mathcal{G}$. The Max-Had_k constraint $\mathcal{C}$ has 2^{2^k} associated variables $y_v^{\mathcal{C}}$ and we want to output a Max-Cut constraint on those variables. In the following we will omit the superscript $\mathcal{C}$ but keep in mind that the auxiliary variables y_v are different for different constraints $\mathcal{C}$.

We select which Max-Cut constraint to output randomly, according to the following distribution:

- With probability $1/4$ we output the constraint $y_u \neq y_{-u}$.
- With probability $1/4$ we output the constraint $y_v \neq y_{-v}$.
- With probability $1/4$ we output the constraint $y_u \neq y_{-v}$.
- With probability $1/4$ we output the constraint $y_v \neq y_{-u}$.

The purpose of the constraints $y_u \neq y_{-u}$ and $y_v \neq y_{-v}$ is to ensure that the optimal assignment is *folded*, which means that for every $v \in V_k$ it holds that $y_v \neq y_{-v}$. If the assignment is folded then the constraints $y_u \neq y_{-v}$ and $y_v \neq y_{-u}$ are both equivalent to $y_u = y_v$, so the Max-Cut instance can actually be regarded as an instance of ‘folded Min-Cut’.

Let us now prove that for this distribution of constraints it is in fact optimal to use folded assignments.

Remark 2.12. In the following we will let $A(v)$ denote the value that an assignment A assigns to the variable y_v . This is a slight abuse of notation, since it would be more appropriate to write $A(y_v)$ instead, but this simplification should not lead to any confusion.

Lemma 2.13. *Given any assignment $A: V_k \rightarrow \{-1, 1\}$, there is a folded assignment $A': V_k \rightarrow \{-1, 1\}$ such that if we sample a random Max-Cut constraint then the probability that A' satisfies the constraint is greater than or equal to the probability that A satisfies it.*

Proof. The assignment A' will agree with A on all primary variables. For each pair of nodes $(v, -v)$ such that $A(v) = A(-v)$, at least one of the variables y_v and y_{-v} is auxiliary. Therefore we can construct the assignment A' using the following algorithm:

1. Let $A' \leftarrow A$.
2. If there exists some pair of nodes $(v, -v)$ such that $A'(v) = A'(-v)$, then let $w \in \{v, -v\}$ be a node such that y_w is auxiliary, and then flip the value of $A'(w)$.
3. Repeat Step 2 until A' is folded.

Every time we change A' , the number of pairs of nodes $(v, -v)$ such that $A'(v) = A'(-v)$ decreases, so the algorithm terminates. Furthermore, none of the changes decrease the probability that A' satisfies a randomly sampled Max-Cut constraint because half of the Max-Cut constraints involving the variable y_w check that $y_w \neq y_{-w}$, a constraint which is satisfied after flipping the value of $A'(w)$ but not before. $\square$

Lemma 2.13 tells us that we can assume without loss of generality that the assignments are folded, in which case the constraints of the form $y_u \neq y_{-v}$ are equivalent to $y_u = y_v$.

Because our Max-Cut instance will be such that it is possible to satisfy almost all the constraints, it is natural to measure the number of falsified constraints instead of the number of satisfied constraints. For this reason we introduce the *cost* of an assignment, which measures the fraction of constraints of the form $y_u = y_v$ that are falsified.

Definition 2.14. For any assignment $A: V_k \rightarrow \{-1, 1\}$ and any Had_k -gadget $\mathcal{G}$, let the *cost* of A with respect to $\mathcal{G}$ be defined as the probability that A assigns different values to the endpoints of a random edge sampled from $\mathcal{G}$,

$$\text{cost}_{\mathcal{G}}(A) = \Pr_{(u,v) \sim \mathcal{G}} [A(u) \neq A(v)].$$

The above definition only applies to complete assignments A , which assign values to *all* variables. However, we will often have partial assignments which only assign values to the primary variables and their antipodes. We refer to such assignments as *Z-assignments*.

Definition 2.15. A *Z-assignment* is an assignment $A_Z: Z_k \rightarrow \{-1, 1\}$. We let $\mathcal{Z}$ denote the set of all folded *Z-assignments*.

We generalise the notion of cost to Z -assignments by defining it as the optimal cost when such a partial assignment is extended to an assignment on all variables. We refer to this quantity as the cost, as well, which causes no ambiguity because the two types of costs are defined on different types of assignments.

Definition 2.16. For any folded Z -assignment A_Z and any Had_k -gadget $\mathcal{G}$, let the cost of A_Z with respect to $\mathcal{G}$ be defined by

$$\text{cost}_{\mathcal{G}}(A_Z) = \min_{A \in \text{Ext}_{\text{folded}}(A_Z)} \text{cost}_{\mathcal{G}}(A),$$

where $\text{Ext}_{\text{folded}}(A_Z)$ denotes the set of all folded extensions of A_Z to the whole of V_k .

Let us now introduce a measure of how hard it is to satisfy the Max-Cut constraints generated by a gadget $\mathcal{G}$ when the primary variables are assigned uniformly at random, but the auxiliary variables are assigned optimally. This definition is useful in the soundness case of Theorem 2.9 because then all assignments to the primary variables behave as if they were essentially random.

Definition 2.17. The *random cost* of a Had_k -gadget $\mathcal{G}$ is defined by

$$\text{rand}(\mathcal{G}) = \mathbf{E}_{A_Z \sim \mathcal{Z}} [\text{cost}_{\mathcal{G}}(A_Z)],$$

where A_Z is a uniformly random folded Z -assignment.

In the completeness case of Theorem 2.9 there are assignments that do not behave randomly. Indeed, there even exist assignments such that for almost all Max- Had_k constraints $\mathcal{C}$, the corresponding Max-Cut-variables $y_v^{\mathcal{C}}$ satisfy the Hadamard predicate, which means that

$$(\sigma_{\emptyset} y_{\sigma_{\emptyset} \chi_{\emptyset}}^{\mathcal{C}}, \dots, \sigma_{[k]} y_{\sigma_{[k]} \chi_{[k]}}^{\mathcal{C}}) = b \chi_S,$$

for some $b \in \{-1, 1\}$. For folded assignments this is equivalent to

$$(y_{\chi_{\emptyset}}^{\mathcal{C}}, \dots, y_{\chi_{[k]}}^{\mathcal{C}}) = b \chi_S,$$

which means that for all $T \subseteq [k]$ we have

$$y_{\chi_T}^{\mathcal{C}} = b(\chi_S)_T = b(-1)^{|S \cap T|} = b(\chi_T)_S.$$

The value of the primary variables are therefore determined by only one of their coordinates. In other words, the primary variables are assigned according to a *dictator*, as defined below.

Definition 2.18. For $S \subseteq [k]$, the *dictator assignment* $d_S: V_k \rightarrow \{-1, 1\}$ is defined by

$$d_S(x) = x_S.$$

As in the soundness case, we are primarily concerned with partial assignments that only assign values to the variables corresponding to nodes in Z_k .

Definition 2.19. For every $S \subseteq [k]$, the *Z-dictator* $d_{S|Z}$ is the folded Z-assignment that sets $d_{S|Z}(z) = d_S(z)$ for all $z \in Z_k$.

Finally we define a measure of how hard it is to satisfy the Max-Cut constraints generated by a gadget $\mathcal{G}$ when the primary variables are assigned according to a dictator. Since we will not be able to choose which dictator they are assigned according to, we have to assume that they are assigned according to the ‘worst’ one.

Definition 2.20. The *dictator cost* of a Had_k -gadget $\mathcal{G}$ is defined by

$$\text{dict}(\mathcal{G}) = \max_{S \subseteq [k]} \text{cost}_{\mathcal{G}}(d_{S|Z}).$$

With all these definitions in place we are ready to show how to reduce from Max-Had_k to Max-Cut.

Proposition 2.21. Let $k \geq 3$ and let $\mathcal{G}$ be an arbitrary Had_k -gadget. Further let $\delta > 0$ be an arbitrary constant. Then, given a Max-Cut instance $\mathcal{I}'$, it is NP-hard to distinguish between the following two cases:

- $\text{val}(\mathcal{I}') \geq 1 - \frac{1}{2} \text{dict}(\mathcal{G}) - \delta$,
- $\text{val}(\mathcal{I}') \leq 1 - \frac{1}{2} \text{rand}(\mathcal{G}) + \delta$.

Proof. We know from Theorem 2.9 that given a Max-Had_k instance $\mathcal{I}$ it is NP-hard to distinguish between the following two cases:

- $\text{val}(\mathcal{I}) \geq 1 - \varepsilon$.
- For every assignment A and every vector $\alpha \in V_k$ it holds that if we sample a constraint $\mathcal{C} = ((\sigma_{\emptyset}, x_{\emptyset}), \dots, (\sigma_{[k]}, x_{[k]}))$ from $\mathcal{I}$ then

$$\left| \Pr[(\sigma_{\emptyset} A(x_{\emptyset}), \dots, \sigma_{[k]} A(x_{[k]})) = \alpha] - 2^{-2^k} \right| \leq \varepsilon.$$

It is therefore sufficient to prove that if we are given an oracle that distinguishes between the cases

$$\text{val}(\mathcal{I}') \geq 1 - \frac{1}{2} \text{dict}(\mathcal{G}) - \delta$$

and

$$\text{val}(\mathcal{I}') \leq 1 - \frac{1}{2} \text{rand}(\mathcal{G}) + \delta$$

for a Max-Cut instance $\mathcal{I}'$, then we can distinguish between the two cases in Theorem 2.9 in polynomial time.

Let us fix some Max-Had_k instance $\mathcal{I}$. We then construct a corresponding Max-Cut instance $\mathcal{I}'(\mathcal{G}, \mathcal{I})$ containing all the variables in $\mathcal{I}$ and some auxiliary variables, as we described earlier.

To sample a constraint from $\mathcal{I}'$ we first sample a constraint $\mathcal{C}$ from $\mathcal{I}$. The constraint $\mathcal{C}$ has a set of associated variables $\{y_v\}_{v \in V_k}$, where 2^k of the variables are primary variables and therefore shared with other constraints. We also sample an edge (u, v) from the gadget $\mathcal{G}$. Finally we sample $a \in \{u, v\}$ and $b \in \{-u, -v\}$ independently and uniformly at random, and output the constraint $y_a \neq y_b$.

Lemma 2.22 (Completeness). *Suppose that $\text{val}(\mathcal{I}) \geq 1 - \varepsilon$. Then*

$$\text{val}(\mathcal{I}') \geq 1 - \frac{1}{2} \text{dict}(\mathcal{G}) - \varepsilon.$$

Proof. Let A be an assignment such that $\text{val}_{\mathcal{I}}(A) \geq 1 - \varepsilon$. We want to construct a good assignment A' for the instance $\mathcal{I}'$, and to do so we start by assigning all the primary variables so that they agree with the corresponding variables in A .

If $\mathcal{C}$ is a random constraint sampled from $\mathcal{I}$, then with probability at least $1 - \varepsilon$ it holds that all primary variables y_z corresponding to that constraint are assigned according to some (negated) dictator $f = \pm d_T$.

Suppose that $\mathcal{C}$ is a constraint for which this holds, and let A' assign values to the auxiliary variables in an optimal way. By Lemma 2.13 we can assume without loss of generality that A' is folded, in which case all constraints of the form $y_u \neq y_{-v}$ are satisfied.

The probability that a random constraint of the form $y_u \neq y_{-v}$, where $u \neq v$, is satisfied is $1 - \text{cost}_{\mathcal{G}}(f|_Z)$, where $f|_Z$ denotes the partial assignment that assigns the value given by f to all nodes in Z_k . Since negating an assignment does not change its value, the maximum cost over all negated dictators is the same as the maximum cost over

all dictators, and therefore we can conclude that the probability that a random constraint of the form $y_u \neq y_{-v}$ is satisfied is at least $1 - \text{dict}(\mathcal{G})$.

Since the probability that a random constraint is of the form $y_u \neq y_{-v}$ is $1/2$, the probability that A' satisfies a random constraint on the variables corresponding to $\mathcal{C}$ is at least $1 - \frac{1}{2} \text{dict}(\mathcal{G})$. Since this lower bound holds for a fraction at least $1 - \varepsilon$ of all Max-Had_k constraints $\mathcal{C}$ we can conclude that

$$\text{val}_{\mathcal{I}'}(A') \geq (1 - \varepsilon) \left(1 - \frac{1}{2} \text{dict}(\mathcal{G}) \right) \geq 1 - \frac{1}{2} \text{dict}(\mathcal{G}) - \varepsilon.$$

□

Lemma 2.23 (Soundness). *Suppose that for every assignment A and every vector $\alpha \in V_k$ it holds that if we sample a random constraint $\mathcal{C} = ((\sigma_\emptyset, x_\emptyset), \dots, (\sigma_{[k]}, x_{[k]}))$ from $\mathcal{I}$ then*

$$\left| \Pr_{\mathcal{C} \sim \mathcal{I}}[(\sigma_\emptyset A(x_\emptyset), \dots, \sigma_{[k]} A(x_{[k]})) = \alpha] - 2^{-2^k} \right| \leq \varepsilon.$$

Then

$$\text{val}(\mathcal{I}') \leq 1 - \frac{1 - \varepsilon \cdot 2^{2^k}}{2} \cdot \text{rand}(\mathcal{G}).$$

Proof. Let A' be an optimal solution to the Max-Cut instance $\mathcal{I}'$. We can define a corresponding solution A to the Max-Had_k instance $\mathcal{I}$ by letting each variable in A be equal to the corresponding primary variable in A' .

Let $\mathcal{C}$ be a random Max-Had_k constraint sampled from $\mathcal{I}$, and let $\{y_v\}_{v \in V_k}$ be the set of corresponding variables in A' . Then it holds for all $\alpha \in V_k$ that

$$\left| \Pr_{\mathcal{C} \sim \mathcal{I}}[(\sigma_\emptyset y_{\sigma_\emptyset \chi_\emptyset}, \dots, \sigma_{[k]} y_{\sigma_{[k]} \chi_{[k]}}) = \alpha] - 2^{-2^k} \right| \leq \varepsilon.$$

By Lemma 2.13 we can assume without loss of generality that $y_v \neq y_{-v}$ for all $v \in V_k$. Therefore

$$\left| \Pr_{\mathcal{C} \sim \mathcal{I}}[(y_{\chi_\emptyset}, \dots, y_{\chi_{[k]}}) = \alpha] - 2^{-2^k} \right| \leq \varepsilon.$$

For every $\alpha \in V_k$, let A_α denote the folded Z -assignment such that $(A_\alpha(y_{\chi_\emptyset}), \dots, A_\alpha(y_{\chi_{[k]}})) = \alpha$. With probability $1/2$, a randomly sampled constraint is of the form $y_v \neq y_{-v}$, and such constraints are always satisfied when the assignment is folded. Only with probability $1/2$ is a

randomly sampled constraint of the form $y_u \neq y_{-v}$ for $u \neq v$, which means that

$$\text{val}(\mathcal{I}') = 1 - \frac{1}{2} \sum_{\alpha \in V_k} \text{cost}_{\mathcal{G}}(A_{\alpha}) \Pr \left[(y_{\chi_{\emptyset}}, \dots, y_{\chi_{[k]}}) = \alpha \right].$$

Since $\Pr_{\mathcal{C} \sim \mathcal{I}} \left[(y_{\chi_{\emptyset}}, \dots, y_{\chi_{[k]}}) = \alpha \right] \geq 2^{-2^k} - \varepsilon$ for all α it follows that

$$\text{val}(\mathcal{I}') \leq 1 - \frac{2^{-2^k} - \varepsilon}{2} \cdot \sum_{\alpha \in V_k} \text{cost}_{\mathcal{G}}(A_{\alpha}).$$

By definition we have

$$\text{rand}(\mathcal{G}) = 2^{-2^k} \sum_{\alpha \in V_k} \text{cost}_{\mathcal{G}}(A_{\alpha}),$$

so we can conclude that

$$\text{val}(\mathcal{I}') \leq 1 - \frac{1 - \varepsilon \cdot 2^{2^k}}{2} \cdot \text{rand}(\mathcal{G}).$$

□

Now set

$$\varepsilon = 2^{-2^k} \delta.$$

In the completeness case we then have

$$\text{val}(\mathcal{I}') \geq 1 - \frac{1}{2} \text{dict}(\mathcal{G}) - \delta,$$

and in the soundness case we have

$$\text{val}(\mathcal{I}') \leq 1 - \frac{1}{2} \text{rand}(\mathcal{G}) + \delta.$$

This means that if we are given an oracle that distinguishes between $\text{val}(\mathcal{I}') \geq 1 - \frac{1}{2} \text{dict}(\mathcal{G}) - \delta$ and $\text{val}(\mathcal{I}') \leq 1 - \frac{1}{2} \text{rand}(\mathcal{G}) + \delta$ for any Max-Cut instance $\mathcal{I}'$, we can distinguish between the completeness and the soundness case for any Max-Had_k instance $\mathcal{I}$, as desired. □

Most of the time we do not really care about the exact values of $\text{dict}(\mathcal{G})$ and $\text{rand}(\mathcal{G})$, but only the relation between them. For this reason we introduce the *score* of a gadget, which captures this relation.

Definition 2.24. The *score* of a Had_k -gadget $\mathcal{G}$ is defined by

$$\text{score}(\mathcal{G}) = \frac{\text{rand}(\mathcal{G})}{\text{dict}(\mathcal{G})}.$$

We can then state the following variant of Proposition 2.21:

Proposition 2.25. *Let $k \geq 3$ and let $\mathcal{G}$ be an arbitrary Had_k -gadget. Further let $0 < \varepsilon < \frac{1}{2} \text{dict}(\mathcal{G})$ and $\delta > 0$ be arbitrary constants. Then, given a Max-Cut instance $\mathcal{I}'$, it is **NP-hard** to distinguish between the following two cases:*

- $\text{val}(\mathcal{I}') \geq 1 - \varepsilon,$
- $\text{val}(\mathcal{I}') \leq 1 - \text{score}(\mathcal{G}) \cdot \varepsilon + \delta.$

Proof. By Proposition 2.21 it is **NP-hard** to distinguish between the following two cases for a given Max-Cut instance $\mathcal{I}$:

- $\text{val}(\mathcal{I}) \geq 1 - \frac{1}{2} \text{dict}(\mathcal{G}) - \delta',$
- $\text{val}(\mathcal{I}) \leq 1 - \frac{1}{2} \text{rand}(\mathcal{G}) + \delta'.$

Given a Max-Cut instance $\mathcal{I}$ with optimal solution A , let us construct a new Max-Cut instance $\mathcal{I}'$ with the same set of variables as $\mathcal{I}$ but where the constraints are sampled using the following strategy:

- With probability p output a fixed constraint that is satisfied by A .
- With probability $1 - p$ output a random constraint from $\mathcal{I}$.

Then it holds for any assignment A' that

$$\text{val}_{\mathcal{I}'}(A') \leq p + (1 - p) \text{val}_{\mathcal{I}}(A').$$

When $A' = A$ the above inequality becomes an equality and therefore

$$\text{val}(\mathcal{I}') = p + (1 - p) \text{val}(\mathcal{I}).$$

Hence it is **NP-hard** to distinguish between the following two cases:

- $\text{val}(\mathcal{I}') \geq p + (1 - p)(1 - \frac{1}{2} \text{dict}(\mathcal{G}) - \delta'),$
- $\text{val}(\mathcal{I}') \leq p + (1 - p)(1 - \frac{1}{2} \text{rand}(\mathcal{G}) + \delta').$

Let

$$p = 1 - \frac{\varepsilon}{\frac{1}{2} \text{dict}(\mathcal{G}) + \delta'}.$$

Then

$$p + (1 - p) \left(1 - \frac{1}{2} \text{dict}(\mathcal{G}) - \delta' \right) = 1 - \varepsilon,$$

and

$$p + (1 - p) \left(1 - \frac{1}{2} \text{rand}(\mathcal{G}) + \delta' \right) = \frac{1 - \varepsilon \cdot \text{score}(\mathcal{G}) + \frac{2\delta'(1+\varepsilon)}{\text{dict}(\mathcal{G})}}{1 + \frac{2\delta'}{\text{dict}(\mathcal{G})}}.$$

If we fix ε and let $\delta' \rightarrow 0$ then

$$\frac{1 - \varepsilon \cdot \text{score}(\mathcal{G}) + \frac{2\delta'(1+\varepsilon)}{\text{dict}(\mathcal{G})}}{1 + \frac{2\delta'}{\text{dict}(\mathcal{G})}} \rightarrow 1 - \varepsilon \cdot \text{score}(\mathcal{G})$$

and therefore it holds for all sufficiently small δ' that

$$p + (1 - p) \left(1 - \frac{1}{2} \text{rand}(\mathcal{G}) + \delta' \right) \leq 1 - \varepsilon \cdot \text{score}(\mathcal{G}) + \delta.$$

Hence it is NP-hard to distinguish between the following two cases:

- $\text{val}(\mathcal{I}') \geq 1 - \varepsilon,$
- $\text{val}(\mathcal{I}') \leq 1 - \text{score}(\mathcal{G}) \cdot \varepsilon + \delta.$

□

Because of Proposition 2.25 we are interested in gadgets with large scores. Håstad et al. [8] showed that there exists a Had_3 -gadget $\mathcal{G}$ such that $\text{score}(\mathcal{G}) = 11/8$, and they also showed that this is optimal among all Had_3 -gadgets. This means that in order to improve on their result it is necessary to look beyond Had_3 , and we will therefore focus on Had_4 -gadgets.

Unfortunately it seems to be computationally difficult to compute $\text{score}(\mathcal{G})$ for arbitrary gadgets $\mathcal{G}$. Since V_k contains 2^{2^k} nodes there exist $2^{2^{2^k}}$ assignments, of which $2^{2^{2^k}-1}$ are folded. A naive algorithm which simply checks all folded assignments therefore runs in time $\mathcal{O}(2^{2^{2^k-1}})$, which is far from feasible even for $k = 3$. Håstad et al. [8] nevertheless managed to analyse their gadget, but their methodology is specific to Had_3 -gadgets and does not generalise to Had_4 -gadgets.

Our main contribution is an algorithm that computes lower bounds on $\text{score}(\mathcal{G})$ in time $2^{\mathcal{O}(2^k)}$ for arbitrary gadgets $\mathcal{G}$. We also use this algorithm to search for good Had_4 -gadgets, and we present a Had_4 -gadget $\mathcal{G}$ such that $\text{score}(\mathcal{G}) > 1.4568$.

Chapter 3

Results

Because of Proposition 2.25 we are interested in gadgets $\mathcal{G}$ such that $\text{score}(\mathcal{G})$ is large, but unfortunately we know of no good algorithms for computing $\text{score}(\mathcal{G})$ exactly. However, if we could prove good lower bounds on $\text{score}(\mathcal{G})$, then that could still yield improved inapproximability results for Max-Cut.

In the next section we describe a general method of computing such lower bounds, which works quite well in practice.

3.1 Relaxation of constraints

Recall that we defined the cost of a folded Z -assignment A_Z with respect to a gadget $\mathcal{G}$ as the minimum cost of any *folded* extension of A_Z to the whole of the hypercube V_k . The problem of finding the optimal extension of A_Z is highly reminiscent of the *minimum cut problem*, which we now restate more formally.

Problem 3.1 (Min-Cut, restated). Given a weighted, undirected graph $G = (V, E)$ and two disjoint sets of nodes $S, T \subseteq V$, the *minimum cut problem* consists of finding two sets $A \supseteq S$ and $B \supseteq T$ partitioning V such that the total weight of all edges (u, v) for which $u \in A$ and $v \in B$ is minimised.

The difference between our problem and the minimum cut problem is that we have additional requirements on the cuts, because we require the assignments to be folded.

Our key idea is to ignore the requirement that the assignments must be folded. If we can compute the minimum cost of any extension

of A_Z then that clearly provides us with a lower bound on the minimum cost of the folded extensions of A_Z , but a priori it is not obvious how good that lower bound is.

There is, however, reason to believe that the lower bound should be good, because our partial assignments A_Z are always folded. It seems plausible that the optimal extension of A_Z should preserve this symmetry even when it is not forced to, and this intuition actually turns out to be correct in many (but not all) cases.

Let us therefore use a new definition of the cost of a partial assignment, which does not require the assignment's extensions to be folded.

Definition 3.2. For any Z -assignment $A_Z: Z_k \rightarrow \{-1, 1\}$ and any Had_k -gadget $\mathcal{G}$, let the *relaxed* cost of A_Z with respect to $\mathcal{G}$ be defined by

$$\widetilde{\text{cost}}_{\mathcal{G}}(A_Z) = \min_{A \in \text{Ext}(A_Z)} \text{cost}_{\mathcal{G}}(A),$$

where $\text{Ext}(A_Z)$ denotes the set of all extensions of A_Z to the whole of V_k .

We also introduce a new definitions of $\text{rand}(\mathcal{G})$ that uses our new definition of cost.

Definition 3.3. The *relaxed* random cost of a Had_k -gadget $\mathcal{G}$ is defined by

$$\widetilde{\text{rand}}(\mathcal{G}) = \mathbf{E}_{A_Z \sim \mathcal{Z}} [\widetilde{\text{cost}}_{\mathcal{G}}(A_Z)],$$

where A_Z is a uniformly random folded Z -assignment.

Observation 3.4. The *relaxed* random cost is a lower bound on the random cost,

$$\widetilde{\text{rand}}(\mathcal{G}) \leq \text{rand}(\mathcal{G}).$$

With our new relaxed definition of cost, we can reduce the problem of evaluating gadgets to the Min-Cut problem, which is solvable in polynomial time. To do this, we let the graph G be the 2^k -dimensional hypercube with edge weights given by the gadget $\mathcal{G}$, and we let S be the set of nodes whose variables are assigned the value 1 by the partial assignment A_Z and we let T be the set of nodes whose variables are assigned the value -1 .

In Algorithm 3.5 we show how to compute an optimal extension of any Z -assignment A_Z with respect to a gadget $\mathcal{G}$. The algorithm

includes a call to MINCUT, where any algorithm for computing minimum cuts may be used (and in particular the algorithm may return any minimum cut if there are several to choose from). However, since solving Min-Cut constitutes the main bottleneck of finding and evaluating gadgets in our work it is preferable to use an efficient implementation, and we have chosen to use the preflow algorithm by Goldberg and Tarjan [6], implemented in the Lemon graph library [4].

We also include the procedure RELAXEDRANDOMCOST which computes $\widetilde{\text{rand}}(\mathcal{G})$ for an arbitrary gadget $\mathcal{G}$ by computing the relaxed cost of each folded Z -assignment. In Section 3.3 we show how this step can be optimised by exploiting the fact that many of the Z -assignments are symmetric and therefore have the same cost.

Algorithm 3.5 Evaluating gadgets

```

1: procedure OPTIMALRELAXEDEXTENSION( $k, A_Z, \mathcal{G}$ )
2:   Let  $G = (V_k, E_k)$  be the  $2^k$ -dimensional hypercube.
3:    $S \leftarrow \{z \in Z_k : A_Z(z) = 1\}$ 
4:    $T \leftarrow \{z \in Z_k : A_Z(z) = -1\}$ 
5:    $(A, B) \leftarrow \text{MINCUT}(G, S, T)$ 

6:   return the complete assignment setting  $y_v = \begin{cases} 1 & \text{if } v \in A \\ -1 & \text{if } v \in B \end{cases}$ 

7: procedure RELAXEDRANDOMCOST( $k, \mathcal{G}$ )
8:    $C \leftarrow 0$ 
9:   for  $A_Z \in \mathcal{Z}$  do
10:     $A \leftarrow \text{OPTIMALRELAXEDEXTENSION}(k, A_Z, \mathcal{G})$ 
11:     $C \leftarrow C + \frac{1}{|\mathcal{Z}|} \sum_{(u,v) \in E_k} \mathcal{G}(u,v)[A(u) \neq A(v)]$ 
12:   return  $C$ 

```

Algorithm 3.5 is sufficiently efficient to evaluate any Had_4 -gadget in reasonable time (our implementation takes less than a day per gadget). However, we also need a way of finding good gadgets to evaluate. In the next section we present an iterative algorithm which models the gadget optimisation problem as a linear program using a small subset of all assignments as constraints.

3.2 Gadget optimisation

Our goal in this section is to find gadgets $\mathcal{G}$ such that $\text{score}(\mathcal{G})$ is large. In the previous section we presented a relatively efficient algorithm that computes a lower bound on $\text{rand}(\mathcal{G})$, but in order to be able to bound $\text{score}(\mathcal{G})$ from below we also need an upper bound on $\text{dict}(\mathcal{G})$.

Fortunately it is in some sense easier to show upper bounds for the cost of a partial assignment than it is to show lower bounds, because in order to show an upper bound it is sufficient to find a single assignment achieving a small cost whereas proving a lower bound requires showing that no assignment has a small cost.

The task is further facilitated by the fact that we only require upper bounds for the costs of Z -dictators, and for a partial assignment that is assigned in accordance with a dictator it seems natural that the extension should assign all variables according to that dictator. This idea yields the following simple upper bound:

Lemma 3.6. *For any Had_k -gadget $\mathcal{G}$ it holds that*

$$\text{dict}(\mathcal{G}) \leq \max_{S \subseteq [k]} \{\text{cost}_{\mathcal{G}}(d_S)\}.$$

Proof. By definition we have

$$\text{dict}(\mathcal{G}) = \max_{S \subseteq [k]} \{\text{cost}_{\mathcal{G}}(d_{S|Z})\}.$$

For any $S \subseteq [k]$ it holds that d_S is a folded extension of $d_{S|Z}$, which means that

$$\text{cost}_{\mathcal{G}}(d_{S|Z}) = \min_{A \in \text{Ext}_{\text{folded}}(d_{S|Z})} \{\text{cost}_{\mathcal{G}}(A)\} \leq \text{cost}_{\mathcal{G}}(d_S).$$

Hence

$$\text{dict}(\mathcal{G}) \leq \max_{S \subseteq [k]} \{\text{cost}_{\mathcal{G}}(d_S)\}.$$

□

The upper bound in Lemma 3.6 can be simplified further if we assume that the gadget $\mathcal{G}$ is ‘symmetric’. In Section 3.3 we clarify what we mean by a symmetric gadget, but for now let us just require that for every $S \subseteq [k]$, the total weight of all edges (u, v) such that u and v differ only in the S -coordinate is the same.

This is a much weaker form of symmetry than what is discussed in Section 3.3, but this weak symmetry is sufficient to simplify Lemma 3.6 because $\text{cost}_{\mathcal{G}}(d_S)$ is exactly the total weight of all edges between nodes that differ in the S -coordinate. The idea behind the simplification is that if the total weight of all edges between nodes that differ in the S -coordinate is the same for all $S \subseteq [k]$, then it must be equal to the total weight of *all* edges divided by the number of subsets $S \subseteq [k]$.

Proposition 3.7. *Suppose that $\mathcal{G}$ is a Had_k -gadget such that the sum*

$$\sum_{(u,v) \in E_k} \mathcal{G}(u,v) \cdot [u_S \neq v_S]$$

is independent of the set $S \subseteq [k]$. Then

$$\text{dict}(\mathcal{G}) \leq 2^{-k}.$$

Proof. By the definition of cost we have

$$\text{cost}_{\mathcal{G}}(d_S) = \sum_{(u,v) \in E_k} \mathcal{G}(u,v) [d_S(u) \neq d_S(v)],$$

and since $d_S(u) = u_S$ and $d_S(v) = v_S$ it follows that

$$\text{cost}_{\mathcal{G}}(d_S) = \sum_{(u,v) \in E_k} \mathcal{G}(u,v) [u_S \neq v_S].$$

The symmetry assumption therefore implies that $\text{cost}_{\mathcal{G}}(d_S)$ is independent of S , so

$$\text{cost}_{\mathcal{G}}(d_S) = \max_{S' \subseteq [k]} \{\text{cost}_{\mathcal{G}}(d_{S'})\},$$

for all $S \subseteq [k]$. Now Lemma 3.6 implies that

$$\begin{aligned} \text{dict}(\mathcal{G}) &\leq \max_{S \subseteq [k]} \{\text{cost}_{\mathcal{G}}(d_S)\} \\ &= 2^{-k} \cdot \sum_{S \subseteq [k]} \max_{S' \subseteq [k]} \{\text{cost}_{\mathcal{G}}(d_{S'})\} \\ &= 2^{-k} \cdot \sum_{S \subseteq [k]} \text{cost}_{\mathcal{G}}(d_S). \end{aligned}$$

For every edge (u,v) , the nodes u and v differ in exactly one coordinate, which means that

$$\sum_{S \subseteq [k]} \text{cost}_{\mathcal{G}}(d_S) = \sum_{(u,v) \in E_k} \mathcal{G}(u,v) \sum_{S \subseteq [k]} [u_S \neq v_S] = \sum_{(u,v) \in E_k} \mathcal{G}(u,v).$$

We can therefore conclude that

$$\text{dict}(\mathcal{G}) \leq 2^{-k} \cdot \sum_{(u,v) \in E_k} \mathcal{G}(u,v).$$

The desired result now follows because our gadgets are defined as probability distributions, meaning that $\sum_{(u,v) \in E_k} \mathcal{G}(u,v) = 1$. $\square$

Proposition 3.7 tells us that if we restrict our attention to gadgets that satisfy some moderate symmetry requirements, then we can bound the dictator cost from above. In order to find gadgets $\mathcal{G}$ such that $\text{score}(\mathcal{G})$ is large we therefore focus on trying to maximise $\text{rand}(\mathcal{G})$.

Trevisan et al. [12] developed a general approach for optimising gadgets by using *linear programs*.

Problem 3.8. A *linear program* is an optimisation problem where, given a matrix $A \in \mathbb{R}^{m \times n}$ and vectors $b \in \mathbb{R}^m$ and $c \in \mathbb{R}^n$, one has to find a vector $x \in \mathbb{R}^n$ that maximises the scalar product $c \cdot x$ subject to $Ax \leq b$ and $x \geq 0$, where the inequalities must hold componentwise.

Håstad et al. [8] observed that the linear programming approach by Trevisan et al. [12] could in principle be used to find gadgets $\mathcal{G}$ that maximise $\text{rand}(\mathcal{G})$ subject to $\sum_{e \in E_k} \mathcal{G}(e) = 1$. Indeed, we can consider the weights $\mathcal{G}(e)$ as variables, and our objective is to assign non-negative real values to these variables so that $\mathbb{E}_{A_Z \sim \mathcal{Z}}[\text{cost}_{\mathcal{G}}(A_Z)]$ is maximised, where A_Z is a uniformly random folded Z -assignment. If we introduce, for each Z -assignment A_Z , a variable ℓ_{A_Z} representing a lower bound on $\text{cost}_{\mathcal{G}}(A_Z)$ then we can write the optimisation problem as follows:

$$\begin{aligned} &\text{Maximise} && 2^{-k} \cdot \sum_{A_Z \in \mathcal{Z}} \ell_{A_Z} \\ &\text{subject to} && \sum_{e \in E_k} \mathcal{G}(e) = 1, \\ &&& \forall e \in E_k: \mathcal{G}(e) \geq 0, \\ &&& \forall A_Z \in \mathcal{Z}: \forall A \in \text{Ext}_{\text{folded}}(A_Z): \ell_{A_Z} \leq \text{cost}_{\mathcal{G}}(A). \end{aligned}$$

Note that this optimisation problem is actually a linear program, because the constraints of the form $\ell_{A_Z} \leq \text{cost}_{\mathcal{G}}(A)$ can be written as

$$\ell_{A_Z} \leq \sum_{(u,v) \in E_k} \mathcal{G}(u,v)[A(u) \neq A(v)],$$

which is a linear inequality.

Unfortunately, as observed by Håstad et al. [8], it is infeasible to solve this linear program even for $k = 3$. The reason is that there exist 2^{2^k-1} folded assignments A and thus there are more than 2^{2^k-1} constraints.

However, we observe that it may not actually be necessary to include all those constraints. Instead we propose an iterative approach where we start with a linear program with no constraints other than those ensuring that the gadget is a probability distribution. We then pick a gadget $\mathcal{G}$ arbitrarily and compute for each Z -assignment A_Z the optimal folded extension $A \in \text{Ext}_{\text{folded}}(A_Z)$ with respect to $\mathcal{G}$. Then the constraint $\ell_{A_Z} \leq \text{cost}_{\mathcal{G}}(A)$ is added to the linear program, but *only* for the optimal folded extension A . When this has been done for all Z -assignments A_Z , we solve the new linear program to find a new gadget $\mathcal{G}$, for which we determine the optimal folded extensions A of all Z -assignments A_Z , and in this way we get new constraints $\ell_{A_Z} \leq \text{cost}_{\mathcal{G}}(A)$ which we add to the linear program. This process can then be repeated until we have found a good gadget.

The problem with the algorithm described above is that we do not know how to efficiently find an optimal folded extension A of a given Z -assignment A_Z . However, as described in Section 3.1, we *do* know how to find the optimal extension when it is not required to be folded. For this reason we will allow constraints $\ell_{A_Z} \leq \text{cost}_{\mathcal{G}}(A)$ also for extensions A which are not folded.

To recapitulate, we start with a linear program $\mathcal{LP}_{\text{init}}$ defined as follows:

$$\begin{aligned} &\text{Maximise} && 2^{-k} \cdot \sum_{A_Z} \ell_{A_Z} \\ &\text{subject to} && \sum_{e \in E_k} \mathcal{G}(e) = 1, \\ &&& \forall e \in E_k: \mathcal{G}(e) \geq 0. \end{aligned}$$

We then iteratively pick a gadget $\mathcal{G}$ given by the optimal solution to the linear program, except in the first iteration where we have to pick the gadget in some other way because the initial linear program does not have an optimal solution. For this gadget we then compute the optimal, not necessarily folded, extensions A of all folded Z -assignments A_Z , and add the corresponding constraints $\ell_{A_Z} \leq \text{cost}_{\mathcal{G}}(A)$ to the linear program, after which the process is repeated. The pseudocode is

shown in Algorithm 3.9.

Algorithm 3.9 Searching for gadgets

```

1: procedure GADGETSEARCH( $k$ )
2:    $\mathcal{LP} \leftarrow \mathcal{LP}_{\text{init}}$ 
3:   Let  $\mathcal{G}$  be the uniform gadget, for which  $\forall e \in E_k: \mathcal{G}(e) = 1/|E_k|$ 
4:    $\mathcal{S} \leftarrow \emptyset$ 
5:   while  $\mathcal{G} \notin \mathcal{S}$  do
6:      $\mathcal{S} \leftarrow \mathcal{S} \cup \{\mathcal{G}\}$ 
7:     for  $A_Z \in \mathcal{Z}$  do
8:        $A \leftarrow \text{OPTIMALRELAXEDEXTENSION}(k, A_Z, \mathcal{G})$ 
9:       Add the constraint  $\ell_{A_Z} \leq \text{cost}_{\mathcal{G}}(A)$  to  $\mathcal{LP}$ 
10:    Solve  $\mathcal{LP}$ 
11:    Set  $\mathcal{G}(e)$  according to the optimal solution of  $\mathcal{LP}$ 
12:   return  $\mathcal{G}$ 

```

Note that the algorithm keeps track of which gadgets have already been analysed. Doing so allows it to terminate whenever it sees some gadget $\mathcal{G}$ for the second time, because analysing that gadget again would yield no new constraints and thus the optimal solution to the linear program would continue to be the gadget $\mathcal{G}$.

Assuming that a polynomial time algorithm is used to solve the linear programs, each iteration of Algorithm 3.9 runs in time $2^{\mathcal{O}(2^k)}$. The number of iterations depends on how OPTIMALRELAXEDEXTENSION is implemented, because for some gadgets there may be several optimal relaxed extensions and which constraint is added to the linear program depends on which extension is chosen. In practice, however, it does not seem to matter much which extension is chosen.

Even though the number of iterations varies depending on which extensions are chosen, when a gadget is finally returned it is guaranteed to have the optimum random relaxed cost. This is because the linear program provides an upper bound on the random relaxed cost of any gadget, and this upper bound is tight for the final gadget $\mathcal{G}$ found by the algorithm since all assignments A used to compute $\widetilde{\text{rand}}(\mathcal{G})$ are added as constraints in the linear program.

We estimate that our implementation of Algorithm 3.9 would terminate within a few weeks in the case $k = 4$, but in the next section we show how an optimisation brings this down to ten minutes.

3.3 Exploiting the symmetries

The hypercubes on which our gadgets are defined are very symmetric objects, and in this section we use this fact to show that we without loss of generality can restrict our attention to gadgets that exhibit the same symmetries. The benefit of this is twofold: Firstly it means that the linear program in Algorithm 3.9 can be simplified because we can assume that many of the edges have the same weight, but more importantly symmetric gadgets are much easier to evaluate because for such gadgets there are not very many significantly different Z -assignments.

3.3.1 The symmetry group

The *symmetry group* of a geometric object is the set of affine transformations under which the object is invariant, with composition of functions as the group operation. For example, the symmetry group of a triangle consists of the identity transformation, the rotations by 120° and 240° around the triangle's centre and the reflections in the triangle's three medians.

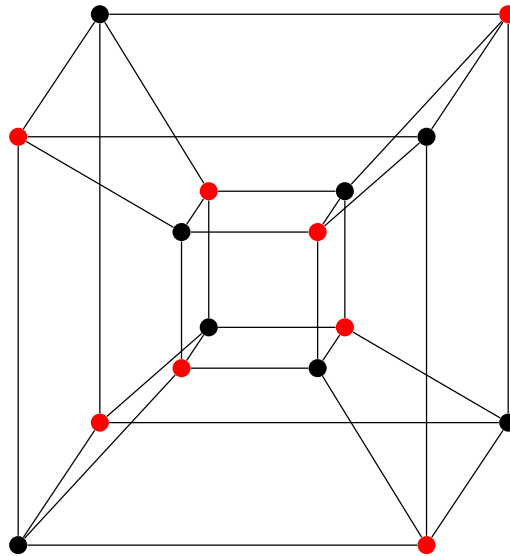


Figure 3.1: The four-dimensional hypercube with the nodes belonging to Z_2 coloured black.

We are interested in the symmetry group of the 2^k -dimensional hypercube where the nodes in Z_k are considered 'special', and we will

denote this group by G_k . In Figure 3.1 we visualise the hypercube in the case $k = 2$.

Definition 3.10. Let G_k be the group consisting of all bijective linear maps $L: \mathbb{R}^{2^k} \rightarrow \mathbb{R}^{2^k}$ such that

$$\forall v \in V_k: Lv \in V_k,$$

and

$$\forall z \in Z_k: Lz \in Z_k,$$

with the group operation being composition of functions.

The symmetry group G_k naturally induces a group action on the set of nodes V_k , but we can also extend the action to the set of edges E_k , the set of Z -assignments $A_Z: Z_k \rightarrow \{-1, 1\}$, the set of assignments $A: V_k \rightarrow \{-1, 1\}$ and the set of gadgets $\mathcal{G}: E_k \rightarrow \mathbb{R}^{\geq 0}$ by letting the group act on the underlying nodes.

Definition 3.11. We let G_k act on edges $(u, v) \in E_k$, Z -assignments $A_Z: Z_k \rightarrow \{-1, 1\}$, assignments $A: V_k \rightarrow \{-1, 1\}$ and Had_k -gadgets $\mathcal{G}$ by letting

- $L(u, v) = (Lu, Lv)$,
- $\forall z \in Z_k: LA_Z(z) = A_Z(Lz)$,
- $\forall v \in V_k: LA(v) = A(Lv)$,
- $\forall e \in E_k: L\mathcal{G}(e) = \mathcal{G}(Le)$,

where L is an arbitrary element of G_k .

The *orbit* of a node $v \in V_k$ is the set of nodes that are ‘symmetric’ to v with respect to the group action. More precisely, the orbit of v is defined by

$$\text{Orb}(v) = \{Lv : L \in G_k\}.$$

Similarly, if X is any edge, Had_k -gadget or partial assignment, then we define the orbit of X by

$$\text{Orb}(X) = \{LX : L \in G_k\}.$$

In the next section we will show that gadgets belonging to the same orbit have similar properties, and then use that result to show that the optimal gadget is very symmetric.

3.3.2 Optimality of symmetric gadgets

Our goal in this subsection is to show there is a symmetric gadget which is at least as good as any other gadget, meaning that no other gadget has a larger random relaxed cost.

However, before we do that we will prove a lemma which states that if the same transformation is applied both to a gadget and to a corresponding assignment, then the cost of the assignment remains unchanged. Intuitively, this lemma is true because when both the gadget and the assignment are transformed, the assignment still looks the same from the perspective of the gadget.

Lemma 3.12. *Let $\mathcal{G}$ be a Had_k -gadget and let A_Z be a folded Z -assignment. Then it holds that*

$$\widetilde{\text{cost}}_{L\mathcal{G}}(LA_Z) = \widetilde{\text{cost}}_{\mathcal{G}}(A_Z),$$

for any $L \in G_k$.

Proof. Let A be an extension of A_Z to the whole of V_k such that $\text{cost}_{\mathcal{G}}(A)$ is minimised. Then

$$\widetilde{\text{cost}}_{\mathcal{G}}(A_Z) = \text{cost}_{\mathcal{G}}(A),$$

and since LA is an extension of LA_Z it also holds that

$$\widetilde{\text{cost}}_{L\mathcal{G}}(LA_Z) \leq \text{cost}_{L\mathcal{G}}(LA).$$

By definition it holds that

$$\begin{aligned} \text{cost}_{L\mathcal{G}}(LA) &= \sum_{(u,v) \in E_k} L\mathcal{G}(u,v)[LA(u) \neq LA(v)] \\ &= \sum_{(u,v) \in E_k} \mathcal{G}(Lu, Lv)[A(Lu) \neq A(Lv)], \end{aligned}$$

and since L is bijective we can conclude that

$$\text{cost}_{L\mathcal{G}}(LA) = \sum_{(u,v) \in E_k} \mathcal{G}(u,v)[A(u) \neq A(v)] = \text{cost}_{\mathcal{G}}(A).$$

Hence

$$\widetilde{\text{cost}}_{L\mathcal{G}}(LA_Z) \leq \text{cost}_{L\mathcal{G}}(LA) = \text{cost}_{\mathcal{G}}(A) = \widetilde{\text{cost}}_{\mathcal{G}}(A_Z).$$

Repeating the same argument but using L^{-1} in place of L , LA_Z in place of A_Z and $L\mathcal{G}$ in place of $\mathcal{G}$, we find that

$$\widetilde{\text{cost}}_{L^{-1}(L\mathcal{G})}(L^{-1}(LA_Z)) \leq \widetilde{\text{cost}}_{L\mathcal{G}}(LA_Z),$$

and therefore

$$\widetilde{\text{cost}}_{LG}(LA_Z) = \widetilde{\text{cost}}_{\mathcal{G}}(A_Z),$$

as desired. $\square$

We will now prove that we without loss of generality can assume that the best gadget is symmetric, in the sense that its orbit has size 1.

Proposition 3.13. *For any Had_k -gadget $\mathcal{G}$ there exists a gadget $\bar{\mathcal{G}}$ such that*

$$\widetilde{\text{rand}}(\bar{\mathcal{G}}) \geq \widetilde{\text{rand}}(\mathcal{G})$$

and

$$|\text{Orb}(\bar{\mathcal{G}})| = 1.$$

Proof. To prove this we first show that all gadgets in $\text{Orb}(\mathcal{G})$ are equally good, and we then combine all those gadgets into an ‘average’ gadget $\bar{\mathcal{G}}$ which turns out to be at least as good as the gadgets in $\mathcal{G}$ ’s orbit while also being very symmetric.

Lemma 3.14. *For any gadget $\mathcal{G}' \in \text{Orb}(\mathcal{G})$ it holds that*

$$\widetilde{\text{rand}}(\mathcal{G}') = \widetilde{\text{rand}}(\mathcal{G}).$$

Proof. Let us write

$$\mathcal{G}' = L\mathcal{G},$$

where $L \in G_k$. Since L induces a bijection on the set of folded Z -assignments, it holds that

$$\mathbf{E}_{A_Z \sim \mathcal{Z}} \left[\widetilde{\text{cost}}_{L\mathcal{G}}(A_Z) \right] = \mathbf{E}_{A_Z \sim \mathcal{Z}} \left[\widetilde{\text{cost}}_{L\mathcal{G}}(LA_Z) \right],$$

where A_Z is a uniformly random folded Z -assignment. This implies that

$$\widetilde{\text{rand}}(L\mathcal{G}) = \mathbf{E}_{A_Z \sim \mathcal{Z}} \left[\widetilde{\text{cost}}_{L\mathcal{G}}(LA_Z) \right],$$

and if we apply Lemma 3.12 we get

$$\widetilde{\text{rand}}(L\mathcal{G}) = \mathbf{E}_{A_Z \sim \mathcal{Z}} \left[\widetilde{\text{cost}}_{\mathcal{G}}(A_Z) \right] = \widetilde{\text{rand}}(\mathcal{G}).$$

$\square$

Lemma 3.14 implies that within an orbit, all gadgets are equally good. Let us now combine those gadgets into a single ‘average’ gadget $\bar{\mathcal{G}}: E_k \rightarrow \mathbb{R}^{\geq 0}$ defined by

$$\bar{\mathcal{G}}(e) = \frac{1}{|\text{Orb}(\mathcal{G})|} \sum_{\mathcal{G}' \in \text{Orb}(\mathcal{G})} \mathcal{G}'(e).$$

In the next lemma we prove that $\bar{\mathcal{G}}$ is not worse than the gadgets in $\mathcal{G}$ ’s orbit.

Lemma 3.15. *It holds that*

$$\widetilde{\text{rand}}(\bar{\mathcal{G}}) \geq \widetilde{\text{rand}}(\mathcal{G}).$$

Proof. Let $A_Z: Z_k \rightarrow \{-1, 1\}$ be a folded Z -assignment sampled uniformly at random and let $A: V_k \rightarrow \{-1, 1\}$ be an extension of A_Z that minimises $\text{cost}_{\bar{\mathcal{G}}}(A)$. Then

$$\widetilde{\text{cost}}_{\bar{\mathcal{G}}}(A_Z) = \text{cost}_{\bar{\mathcal{G}}}(A),$$

and by the definition of $\bar{\mathcal{G}}$ it follows that

$$\widetilde{\text{cost}}_{\bar{\mathcal{G}}}(A_Z) = \frac{1}{|\text{Orb}(\mathcal{G})|} \sum_{\mathcal{G}' \in \text{Orb}(\mathcal{G})} \text{cost}_{\mathcal{G}'}(A).$$

It also holds for all $\mathcal{G}' \in \text{Orb}(\mathcal{G})$ that

$$\text{cost}_{\mathcal{G}'}(A) \geq \widetilde{\text{cost}}_{\mathcal{G}'}(A_Z),$$

so

$$\widetilde{\text{cost}}_{\bar{\mathcal{G}}}(A_Z) \geq \frac{1}{|\text{Orb}(\mathcal{G})|} \sum_{\mathcal{G}' \in \text{Orb}(\mathcal{G})} \widetilde{\text{cost}}_{\mathcal{G}'}(A_Z).$$

Taking the expectation over all folded Z -assignments A_Z , we get

$$\begin{aligned} \mathbf{E}_{A_Z \sim \mathcal{Z}} [\widetilde{\text{cost}}_{\bar{\mathcal{G}}}(A_Z)] &\geq \mathbf{E}_{A_Z \sim \mathcal{Z}} \left[\frac{1}{|\text{Orb}(\mathcal{G})|} \sum_{\mathcal{G}' \in \text{Orb}(\mathcal{G})} \widetilde{\text{cost}}_{\mathcal{G}'}(A_Z) \right] \\ &= \frac{1}{|\text{Orb}(\mathcal{G})|} \sum_{\mathcal{G}' \in \text{Orb}(\mathcal{G})} \widetilde{\text{rand}}(\mathcal{G}'). \end{aligned}$$

By Lemma 3.14 it holds that $\widetilde{\text{rand}}(\mathcal{G}') = \widetilde{\text{rand}}(\mathcal{G})$ for all $\mathcal{G}' \in \text{Orb}(\mathcal{G})$, so we can conclude that

$$\widetilde{\text{rand}}(\bar{\mathcal{G}}) \geq \frac{1}{|\text{Orb}(\mathcal{G})|} \sum_{\mathcal{G}' \in \text{Orb}(\mathcal{G})} \widetilde{\text{rand}}(\mathcal{G}) = \widetilde{\text{rand}}(\mathcal{G}).$$

□

To conclude the proof of Proposition 3.13 we only need to show that $|\text{Orb}(\mathcal{G})| = 1$, which holds if and only if $\bar{\mathcal{G}}$ is invariant under all transformations in G_k .

Lemma 3.16. *Let $L \in G_k$ be arbitrary. Then*

$$L\bar{\mathcal{G}} = \bar{\mathcal{G}}.$$

Proof. By linearity we have, for each $e \in E_k$, that

$$L\bar{\mathcal{G}}(e) = \frac{1}{|\text{Orb}(\mathcal{G})|} \sum_{\mathcal{G}' \in \text{Orb}(\mathcal{G})} L\mathcal{G}'(e).$$

When applying the transformation L on each element of $\text{Orb}(\mathcal{G})$ we simply get a permutation of the orbit, which means that

$$L\bar{\mathcal{G}}(e) = \frac{1}{|\text{Orb}(\mathcal{G})|} \sum_{\mathcal{G}' \in \text{Orb}(\mathcal{G})} \mathcal{G}'(e) = \bar{\mathcal{G}}(e).$$

□

□

By Proposition 3.13 we can restrict our search to gadgets that are invariant under all transformations in G_k . Such gadgets have very few different edge weights, because if $\mathcal{G}$ satisfies $|\text{Orb}(\mathcal{G})| = 1$ then for any $e \in E_k$ and any $L \in G_k$,

$$\mathcal{G}(Le) = L\mathcal{G}(e) = \mathcal{G}(e),$$

so all edges in the same orbit have the same weight. This means that the linear program in Algorithm 3.9 can be made much smaller by using only a single variable for each orbit, rather than one variable for each edge in each orbit. For instance, in the case $k = 4$ the number of variables representing edge weights decreases from $2^{19} = 524\,288$ to 7.

Making the linear program smaller makes the search more efficient, but in the next subsection we describe an even more important optimisation which makes evaluating symmetric gadgets much faster.

3.3.3 Evaluation of symmetric gadgets

If a gadget is symmetric then there are many Z -assignments to which the gadget looks the same, which means that if we compute the cost of one Z -assignment the result can be reused for other Z -assignments. This is formalised in the following proposition:

Proposition 3.17. *Let $\mathcal{G}$ be a Had_k -gadget such that $|\text{Orb}(\mathcal{G})| = 1$. Let A_Z be an arbitrary folded Z -assignment and let $A'_Z \in \text{Orb}(A_Z)$. Then*

$$\widetilde{\text{cost}}_{\mathcal{G}}(A'_Z) = \widetilde{\text{cost}}_{\mathcal{G}}(A_Z).$$

Proof. Let us write $A'_Z = LA_Z$, where $L \in G_k$. By Lemma 3.12 it holds that

$$\widetilde{\text{cost}}_{L\mathcal{G}}(LA_Z) = \widetilde{\text{cost}}_{\mathcal{G}}(A_Z).$$

Because $|\text{Orb}(\mathcal{G})| = 1$ it holds that $L\mathcal{G} = \mathcal{G}$, and therefore

$$\widetilde{\text{cost}}_{\mathcal{G}}(LA_Z) = \widetilde{\text{cost}}_{\mathcal{G}}(A_Z),$$

as desired. □

Proposition 3.17 dramatically improves the performance of Algorithm 3.9 because there are much fewer orbits of folded Z -assignment than there are folded Z -assignments. For instance, for $k = 4$ there are $2^{16} = 65\,536$ folded Z -assignments, but each of them belongs to one of only 8 different orbits, so for $k = 4$ the algorithm's performance improves by almost a factor 10 000.

3.4 A good Had_4 -gadget

In this section we present the gadget $\mathcal{G}$ constructed by Algorithm 3.9 for $k = 4$.¹ When G_4 acts on the 16-dimensional hypercube, it induces 8 different node orbits and 7 different edge orbits. In Table 3.1 we present, for each node orbit, a representative node in that orbit and the size of the orbit. We also label the orbits by different letters in order to be able to more easily refer to them.

¹The code used to find the gadget is available at <https://github.com/Gullesnuffs/MaxCutInapproximability>

Orbit	Representative node															Size
Z	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	32
A	-1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	512
B	-1	-1	1	1	1	1	1	1	1	1	1	1	1	1	1	3 840
C	-1	-1	-1	1	1	1	1	1	1	1	1	1	1	1	1	17 920
D	-1	-1	-1	-1	1	1	1	1	1	1	1	1	1	1	1	1 120
E	-1	-1	-1	1	-1	1	1	1	1	1	1	1	1	1	1	26 880
F	-1	-1	-1	1	-1	1	1	1	-1	1	1	1	1	1	1	14 336
G	-1	-1	-1	1	-1	1	1	1	-1	1	1	1	1	1	-1	896

Table 3.1: The node orbits when G_4 acts on V_4 .

The different edge orbits connect nodes in different orbits, and we give the edge orbits labels based on the orbits of the nodes they are incident to. For instance, all edges that are incident to the nodes in the Z -orbit belong to the same orbit and we refer to this orbit as the ZA -orbit because those edges are also incident to nodes in the A -orbit.

In Table 3.2 we present the sizes of the different edge orbits, as well as the weights assigned by $\mathcal{G}$ to each edge in the orbit. In order to avoid fractions we multiply the weights by their least common denominator, which is 11 943 966 720. We also include the approximate total weight of all edges in each orbit.

Orbit	Size	Weight	Total normalised weight
ZA	512	4 389 840	0.188
AB	7 680	292 656	0.188
BC	53 760	54 610	0.246
CD	17 920	102 042	0.153
CE	215 040	10 488	0.189
EF	215 040	1 995	0.036
FG	14 336	0	0

Table 3.2: The edge orbits when G_4 acts on V_4 , together with the weights assigned by $\mathcal{G}$ to edges in each orbit. The weights have been multiplied by 11 943 966 720. The total normalised weight denotes the fraction of the total weight that is assigned to each orbit.

Using Algorithm 3.5 with the optimisation based on Proposition

3.17, we can compute that for this gadget we have

$$\widetilde{\text{rand}}(\mathcal{G}) = \frac{331\,440\,289}{3\,640\,066\,048}.$$

Since $\widetilde{\text{rand}}(\mathcal{G})$ is a lower bound on $\text{rand}(\mathcal{G})$ we can conclude that

$$\text{rand}(\mathcal{G}) \geq \frac{331\,440\,289}{3\,640\,066\,048}.$$

Since $\mathcal{G}$ is symmetric it follows by Proposition 3.7 that

$$\text{dict}(\mathcal{G}) \leq \frac{1}{16}$$

and therefore

$$\text{score}(\mathcal{G}) = \frac{\text{rand}(\mathcal{G})}{\text{dict}(\mathcal{G})} \geq \frac{331\,440\,289}{227\,504\,128} > 1.45685.$$

Finally, our main result follows by applying Proposition 2.25 to the gadget $\mathcal{G}$.

Theorem 3.18. *Let $\varepsilon < 1/32$ be an arbitrary constant. Then, given a Max-Cut instance $\mathcal{I}$, it is **NP**-hard to distinguish between the following two cases:*

- $\text{val}(\mathcal{I}) \geq 1 - \varepsilon$,
- $\text{val}(\mathcal{I}) \leq 1 - 1.45685\varepsilon$.

Chapter 4

Conclusion

We have shown improved inapproximability results for the Max-Cut problem, using a novel algorithm which searches for and evaluates gadget reductions from the Max-Had_k problem. The algorithm computes lower bounds on the inapproximability factors achieved by different gadgets in polynomial time in the size of the gadgets. This allows us to easily evaluate gadget reductions from Max-Had₄, which was previously out of reach of algorithmic approaches.

The main drawback of the algorithm is that it only provides lower bounds on the inapproximability factors rather than computing them exactly, but the lower bounds seem to be quite good in practice.

4.1 Future work

A possible next step would be to search for gadget reductions from Max-Had_k to Max-Cut for $k \geq 5$. We believe that such gadgets can be used to prove that it is NP-hard to distinguish between Max-Cut instances $\mathcal{I}$ with $\text{val}(\mathcal{I}) \geq 1 - \varepsilon$ and instances with $\text{val}(\mathcal{I}) \leq 1 - C\varepsilon$ for constants C considerably larger than the value 1.4568 achieved by our Had₄-gadget. However, there is a limit to how good inapproximability results can be proven in this way, because Håstad et al. [8] showed that $\text{score}(\mathcal{G}) \leq 2.542$ for all Had_k-gadgets $\mathcal{G}$.

Håstad et al. [8] proposed studying a particular family of Had_k-gadgets, and in a conjecture known as the Game Show Conjecture they hypothesised that as $k \rightarrow \infty$ the scores of their gadgets would converge to $3/2$. However, in view of our result showing that the best Had₄-gadget has a considerably larger score (at least 1.4568) than the

best Had_3 -gadget (with score 1.375), it is tempting to predict that for $k \geq 5$ there exist Had_k -gadgets with scores exceeding $3/2$, and perhaps even $k = 5$ is sufficient to break the $3/2$ barrier.

Evaluating Had_5 -gadgets may actually be feasible using our algorithm, although the computational resources required would be far greater than what is needed to evaluate Had_4 -gadgets. The main bottleneck of the algorithm consists of computing minimum cuts in the 2^k -dimensional hypercube, which is easy in the case $k = 4$ because then the graph only contains $2^{16} = 65\,536$ nodes and $2^{19} = 524\,288$ edges. When $k = 5$, on the other hand, the graph contains $2^{32} \approx 4.3 \cdot 10^9$ nodes and $2^{36} \approx 6.9 \cdot 10^{10}$ edges, which makes the computation of minimum cuts much more challenging. However, there are several ways to optimise the algorithm further:

- The hypercubes in which we want to compute minimum cuts have some symmetries that can be exploited. After fixing a Z -assignment A_Z , the number of symmetries in the hypercube is much smaller than before, but some symmetries still remain and therefore some of the nodes and edges may be considered ‘identical’, effectively decreasing the size of the graph.
- The standard way of computing minimum cuts uses the Max-Flow Min-Cut theorem, which states that the size of the minimum cut separating two sets S and T is equal to the maximum amount of flow that can be sent from S to T along the graph’s edges. It follows that if we can compute a lower bound on the maximum flow then we automatically get a lower bound on the size of the minimum cut. Since any feasible flow provides a lower bound on the maximum flow, approximating the maximum flow may in some cases be considerably easier than computing it exactly.
- If the gadget to be evaluated assigns a zero weight to some edge, then that edge can be removed. For the Had_4 -gadget presented in Section 3.4 this optimisation would only decrease the number of edges by roughly 2.7%, but there actually exist fairly good Had_4 -gadgets which assign zero weights to all edges in the orbits CE , EF and FG , and removing all those edges would result in an 85% reduction in the size of the graph. In the case $k = 5$ we expect there to be even more edges that are either not

very important or even detrimental to the score of a gadget, so it would make sense to restrict the search to gadgets with non-zero weights only on a relatively small set of edges.

Another interesting avenue for future research is the problem of finding the optimal Had_4 -gadget. This requires evaluating gadgets exactly, which we do not know how to do efficiently. However, even though the evaluation algorithm presented in this thesis only computes lower bounds on the scores of gadgets, it is possible to adapt it so that it computes the score exactly but no longer is guaranteed to run in polynomial time.

The adaptation relies on the observation that if any of the minimum cuts satisfy the foldedness requirement, meaning that antipodal nodes belong to different sets, then the cost is equal to the relaxed cost. If, on the other hand, none of the minimum cuts respect foldedness then we can pick some pair of antipodal nodes for which the foldedness requirement is violated and ‘guess’ which one of the antipodal nodes should belong to which side of the cut. There are two options for the guess, and for each option we can recurse and if all minimum cuts still violate the foldedness requirement we can then pick a new pair of antipodal nodes and guess which node should be on which side of the cut. We can continue recursing in this way until a folded minimum cut is found.

Because the branching factor is 2 in each step of the recursion, this algorithm may not run in polynomial time. However, if the recursion depth is not too large then it might still be feasible to use this algorithm to evaluate Had_4 -gadgets exactly.

Bibliography

- [1] Per Austrin and Johan Håstad. ‘On the usefulness of predicates’. In: *ACM Transactions on Computation Theory (TOCT)* 5.1 (2013), p. 1.
- [2] Siu On Chan. ‘Approximation resistance from pairwise-independent subgroups’. In: *Journal of the ACM (JACM)* 63.3 (2016), p. 27.
- [3] Pierluigi Crescenzi, Riccardo Silvestri, and Luca Trevisan. ‘On weighted vs unweighted versions of combinatorial optimization problems’. In: *Information and Computation* 167.1 (2001), pp. 10–26.
- [4] Balázs Dezső, Alpár Jüttner, and Péter Kovács. ‘LEMON—an open source C++ graph template library’. In: *Electronic Notes in Theoretical Computer Science* 264.5 (2011), pp. 23–45.
- [5] Michel X Goemans and David P Williamson. ‘Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming’. In: *Journal of the ACM (JACM)* 42.6 (1995), pp. 1115–1145.
- [6] Andrew Goldberg and Robert Tarjan. ‘A new approach to the maximum-flow problem’. In: *Journal of the ACM (JACM)* 35.4 (1988), pp. 921–940.
- [7] Johan Håstad. ‘Some optimal inapproximability results’. In: *Journal of the ACM (JACM)* 48.4 (2001), pp. 798–859.
- [8] Johan Håstad, Sangxia Huang, Rajsekar Manokaran, Ryan O’Donnell, and John Wright. ‘Improved NP-Inapproximability for 2-Variable Linear Equations’. In: *Theory Of Computing* 13.19 (2017), pp. 1–51.
- [9] Richard M Karp. ‘Reducibility among combinatorial problems’. In: *Complexity of computer computations*. Springer, 1972, pp. 85–103.

- [10] Subhash Khot. 'On the power of unique 2-prover 1-round games'. In: *Proceedings of the thirty-fourth annual ACM symposium on Theory of computing*. ACM. 2002, pp. 767–775.
- [11] Subhash Khot, Guy Kindler, Elchanan Mossel, and Ryan O'Donnell. 'Optimal inapproximability results for MAX-CUT and other 2-variable CSPs?' In: *SIAM Journal on Computing* 37.1 (2007), pp. 319–357.
- [12] Luca Trevisan, Gregory B Sorkin, Madhu Sudan, and David P Williamson. 'Gadgets, approximation, and linear programming'. In: *SIAM Journal on Computing* 29.6 (2000), pp. 2074–2097.

