FACULTY OF MATHEMATICS AND PHYSICS
CHARLES UNIVERSITY, PRAGUE

Implementation and Testing Of a New Algorithm For the MAX-CUT Problem

Jan Vondrák

Advisor: Dr. Martin Loebl, CSc.

April 1999

Contents

Chapter 1

Introduction

MAX-CUT is a problem of graph theory which is very simple to define, yet surprisingly hard to solve:

Given a graph, divide its vertices into two parts so that the number of edges between them is as large as possible. More generally, the edges may have arbitrary weights, and what we want to maximize is the sum of weights over all the edges between the two sets of vertices.

This is a combinatorial problem with a history of its own, but it has another motivation which comes from statistical physics. It arises in the study of the *Ising model*, which is a theoretical model of nearest-neighbor interaction in a crystallic structure. The complete description of a specific model is embodied in the *partition function* from which all fundamental physical quantities can be derived. It turns out that this function is closely analogous to the *generating function of cuts* which is a standard concept in graph theory. The value of MAX-CUT can be obtained as a by-product of the generating function of cuts, so computing the generating function is generally a more difficult task. Rather than MAX-CUT itself, this work is actually concerned with the generating function of cuts.

The general MAX-CUT problem has defied efficient solution so far, and indeed, it has been shown to be **NP-hard** [3], even in the case when all edge weights are equal to 1. In spite of that, many attempts have been made to tackle the problem with approximation and randomized algorithms. Poljak and Rendl solved a relaxation of the problem using eigenvalues [10]. A similar approach based on semidefinite programming was developed by Goemans and Williamson who presented a randomized algorithm in [6]. There are also branch-and-cut algorithms which can be applied effectively to special cases of the problem. De Simone et al. [12] were able to compute exact MAX-CUT values for large instances of the *spin glass* model. (This is a special case of the Ising model where the underlying graph is a toroidal lattice with edge weights ± 1.) Their method is based on integer programming and so far it has been the most successful way to find spin glass ground states, with time complexity approximately $O(n^3)$.

However, there has been no deterministic algorithm which would produce an exact result in polynomial time for toroidal graphs in general, for example. Polynomial-time methods for toroidal square lattices were suggested in the early 60's by Kasteleyn [8] and Kac, Ward [7]. Kac and Ward tried to calculate the partition function as a determinant of a $4n \times 4n$ matrix, and even though their original derivation was not quite exact, it demonstrated that such an approach was indeed feasible. A similar method was used by physicists Saul and Kardar [11] who managed to obtain the partition function of a 2D spin glass in estimated time $O(n^{3+\epsilon}), \epsilon < 1$.

Thanks to a recent discovery by Dr. Martin Loebl and Dr. Anna Galluccio [2], though, it has become possible to solve MAX-CUT in polynomial time for *any graph of genus bounded by a constant*. The method they suggested produces actually the complete generating function of cuts which contains information about all possible cuts in the graph. In terms of statistical physics, this means the distribution of physical states over all possible energy levels of the system. This can be of tremendous interest for anyone studying a particular Ising model.

The purpose of my work was, under Dr. Loebl's supervision, to develop this algorithm and to solve all the difficulties I would encounter on the way from a theoretical sketch to functional software. First I will summarize the theory upon which this work is based, and then I will describe in detail the algorithm itself and the implementation techniques I used. Finally, I will show a few theoretical results which allowed me to come up with some additional optimizations of the algorithm.

Finally, I would like to express my gratitude to Dr. Loebl who kept me on the right track with his helpful hints and unceasing encouragement. Thank you for your valuable time and support!

Chapter 2

MAX-CUT and Statistical Physics

This thesis is written chiefly from a combinatorialist's point of view, but due to its strong links with statistical physics, it will be helpful to clarify the connection between the two subjects at the very beginning.

2.1 Basic Definitions

Definition 2.1.1 *A graph is a pair $G = (V, E)$ where V is a finite set of vertices and E is a set of unordered pairs $\{u, v\} \subset V$ (edges). In addition, x_e denotes a symbolic variable associated with the edge $e \in E$. For a subset of edges $M \subset E$, $x(M)$ means the product of the variables associated with the edges in M.*

Definition 2.1.2 *A cut of a graph $G = (V, E)$ is a partition of its vertices into two disjoint subsets $V_1, V_2 \subset V$, and the implied set of edges between the two parts: $C(V_1, V_2) = \{\{u, v\} \in E; u \in V_1, v \in V_2\}$*

The generating function of cuts is a polynomial $\mathcal{C}(G, x)$ which equals the sum of $x(C)$ over all cuts C of G.

Definition 2.1.3 *An eulerian subgraph of graph $G = (V, E)$ is a set of edges $U \subset E$ such that each vertex is incident with an even number of edges from U.*

The generating function of eulerian subgraphs is a polynomial $\mathcal{E}(G, x)$ which equals the sum of $x(U)$ over all eulerian subgraphs U of G.

Definition 2.1.4 *A perfect matching of graph $G = (V, E)$ is a set of edges $P \subset E$ such that each vertex is incident with exactly one edge from P.*

The generating function of perfect matchings is a polynomial $\mathcal{P}(G, x)$ which equals the sum of $x(P)$ over all perfect matchings P of G.

Definition 2.1.5 The MAX-CUT Problem

Input:

Graph $G = (V, E)$

Weight function $w : E \to \mathbf{Z}$

Question:

What is the maximum possible weight of a cut of G*?*

$$MAXCUT(G) = \max_{C\,cut} \sum_{e \in C} w_e$$

2.2 The Ising Model

Now let us consider the *Ising model* of a physical system: the vertices of a graph represent particles and the edges describe interaction between pairs of particles. A common example is a planar lattice where each particle interacts only with its neighbors. Often, one adds edges connecting the first and last vertex in each row and column, representing *periodic boundary conditions* in our model. This makes the graph a *toroidal lattice*.

Next, we assign a factor J_{ij} to each edge $\{i, j\}$; this factor describes the nature of the interaction between particles i and j. A physical state of the system is an assignment of $\sigma_i \in \{+1, -1\}$ to each vertex i. This describes the two possible spin orientations the particle can take. The *Hamiltonian* (or energy function) is then defined as

$$H(\sigma) = - \sum_{\{i,j\} \in E} J_{ij} \sigma_i \sigma_j$$

One of the key questions one may ask about a specific system is:

"What is the lowest possible (*ground state*) energy?"

Before we seek an answer to this question, we should realize that the physical states (spin assignments) correspond exactly to the cuts of the underlying graph. Let us define:

$$V_1 = \{i \in V; \sigma_i = +1\}$$
$$V_2 = \{i \in V; \sigma_i = -1\}$$

Then this partition of vertices encodes uniquely the asignment of spins to particles. Of course, two states with reversed +1/-1 spins produce the same cut, but they also have the same energy, so we will unify them whenever it will be convenient.

Thus, the edges contained in the cut $C(V_1, V_2)$ are those connecting a pair of particles with different spins, and those outside the cut connect pairs with equal spins. This allows us to rewrite the Hamiltonian in the following way:

$$H(\sigma) \;=\; \sum_{\{i,j\}\in C} J_{ij} - \sum_{\{i,j\}\in E\setminus C} J_{ij} \;=\; 2w(C) - W$$

where $w(C) = \sum_{\{i,j\}\in C} J_{ij}$ denotes the weight of a cut, and $W = \sum_{\{i,j\}\in E} J_{ij}$ is the sum of all edge weights in the graph.

Clearly, if we find the value of MAX-CUT, we have found the maximum energy of the physical system. Similarly, MIN-CUT (the cut with minimum possible weight) corresponds to the minimum energy of the system. Note that if we allow negative edge weights, we can transform MAX-CUT into MIN-CUT (and vice versa) simply by reversing the value of each edge weight. So the two problems are equivalent (unlike the MIN-CUT problem with positive edge weights arising in the study of network flows, which is significantly easier).

However, the analogy between cuts and physical states goes even deeper than that. Recall the definition of the generating function of cuts:

$$\mathcal{C}(G, x) = \sum_{cut C} x(C)$$

If we assign a term $x^{J_{ij}}$ to each edge $\{i, j\}$, the generating function becomes

$$\mathcal{C}(G, x) = \sum_{cut C} x^{w(C)} = \sum_{k} C_k x^k$$

where C_k denotes the number of cuts with weight k. But this is also equal to the number of states with energy $2k - W$. Thus, if we knew the complete generating function, we would also know the distribution of physical states over all possible energy levels. In the language of physics, this information is encapsulated in the *partition function*:

$$Z(\beta) = \sum_{\sigma} e^{-\beta H(\sigma)}$$

Substituting the equalities above and counting over cuts instead of σ assignments yields:

$$Z(\beta) = 2 \sum_{cut C} e^{-\beta(2w(C)-W)} = 2e^{\beta W} \sum_{cut C} e^{-2\beta w(C)} = 2e^{\beta W} \mathcal{C}(G, e^{-2\beta})$$

So we can obtain the value of the partition function simply by substituting $e^{-2\beta}$ into the generating function of cuts. The other way round, multiplying the partition function by $\frac{1}{2}e^{-\beta W}$ and expressing it as a polynomial in $e^{-2\beta}$ yields the generating function of cuts. The difference between the two functions is merely formal; both of them encode the distribution of physical states in a similar way.

Finding the generating function of cuts is generally a harder problem than MAX-CUT, because knowing all the coefficients of the polynomial, we can simply find the highest power $x^{k_{max}}$ with a non-zero coefficient; k_{max} is then the maximum cut value. Similarly, the first non-zero term of the partition function corresponds to the ground state energy, but there is much more information contained in it. In the following, we will try to compute the complete generating function rather than MAX-CUT alone.

Chapter 3

Theory of Pfaffian Orientations

In this chapter, I will summarize the theoretical background of the MAX-CUT problem. It will take us from the generating function of cuts to that of perfect matchings which is closely related to the algebraic notion of a *Pfaffian*. Previously, this connection was only evident in planar graphs, but the recent results by Loebl and Galluccio [2] paved the way for an efficient approach applicable to any graph of bounded genus.

3.1 From Cuts to Eulerian Subgraphs

In planar graphs, there is a well-known duality between cuts and eulerian subgraphs. Because the edges incident with a vertex correspond to a dual circle (encircling the vertex and connecting all the adjacent faces), it is easy to see that a cut becomes an eulerian subgraph of the dual graph.

However, this does not work in the general case. Yet there is a duality between the generating function of cuts and eulerian subgraphs (of the same graph!), which was discovered by van der Waerden [13]. We will derive the relation for the partition function $Z(\beta)$ rather than the generating function of cuts; we know this serves our purpose just as well.

The basic trick of van der Waerden's was to substite hyperbolic functions for the exponential terms:

$$e^y = \cosh(y) + \sinh(y) = \cosh(y)(1 + \tanh(y))$$

where

$$\cosh(y) = \frac{e^y + e^{-y}}{2}, \qquad \sinh(y) = \frac{e^y - e^{-y}}{2}, \qquad \tanh(y) = \frac{\sinh(y)}{\cosh(y)}$$

Thus we obtain the partition function in the following form:

$$Z(\beta) = \sum_{\sigma} e^{-\beta H(\sigma)} = \sum_{\sigma} \prod_{\{i,j\}\in E} e^{\beta J_{ij}\sigma_i\sigma_j} =$$

$$= \sum_{\sigma} \prod_{\{i,j\}\in E} \cosh(\beta J_{ij}\sigma_i\sigma_j)(1 + \tanh(\beta J_{ij}\sigma_i\sigma_j)$$

Now we use the fact that $\cosh(-y) = \cosh(y)$, whereas $\tanh(-y) = -\tanh(y)$:

$$Z(\beta) = \prod_{\{i,j\}\in E} \cosh(\beta J_{ij}) \sum_{\sigma} \prod_{\{i,j\}\in E} (1 + \sigma_i\sigma_j \tanh(\beta J_{ij})) =$$

$$= \prod_{\{i,j\}\in E} \cosh(\beta J_{ij}) \sum_{\sigma} \sum_{U\subset E} \prod_{\{i,j\}\in U} \sigma_i\sigma_j \tanh(\beta J_{ij}) =$$

$$= \prod_{\{i,j\}\in E} \cosh(\beta J_{ij}) \sum_{U\subset E} (\prod_{\{i,j\}\in U} \tanh(\beta J_{ij}) \sum_{\sigma} \prod_{i\in V} \sigma_i^{d_U(i)})$$

where $d_U(i)$ means the number of edges in U incident with the vertex i. If we consider the sum over all spin assigments σ for a given $U \subset E$, we can see that whenever there exists a vertex i incident with an odd number of edges from U, the resulting sum will be zero. This is because the terms arising from spin assigments where $\sigma_i = +1$ will exactly cancel out the corresponding terms with $\sigma_i = -1$. On the other hand, if all vertices have even degrees in U, the sign of all contributing terms will be positive. So if n is the total number of vertices:

$$\sum_{\sigma\in\{\pm 1\}^n} \prod_{i\in V} \sigma_i^{d_U(i)} = 2^n$$

whenever U is eulerian, and zero otherwise. Finally, we get

$$Z(\beta) = 2^n \prod_{\{i,j\}\in E} \cosh(\beta J_{ij}) \sum_{U eulerian} \prod_{\{i,j\}\in U} \tanh(\beta J_{ij}) =$$

$$= 2^n \prod_{\{i,j\}\in E} \cosh(\beta J_{ij}) \quad \mathcal{E}(G')$$

So the partition function (or the generating function of cuts) can be expressed as the generating function of eulerian subgraphs of the same graph, with modified edge weights

$$w'_{ij} = \tanh(\beta J_{ij})$$

3.2 From Eulerian Subgraphs to Perfect Matchings

Now we need a graph transformation converting eulerian subgraphs into perfect matchings. The transformation I give here is based on Fisher's construction described in [1]. It is local in the sense that it only modifies each vertex in a way dependent on its degree. It can also be seen that it preserves the genus of the graph.

Definition 3.2.1 *Let $G = (V, E)$ be a graph embedded in an orientable surface of genus g, and $v \in V$ a vertex. Let $e_1, e_2, ..., e_d \in E$ denote the edges incident with v, ordered clockwise as they spread out from v in the embedding. Then the even splitting of v is a graph $G' = (V', E')$ where*

- $V' = V \setminus \{v\} \cup \{v_1, ..., v_d, v'_1, ..., v'_d\}$
- $E' = E \setminus \{e_1, e_2, ..., e_d\} \cup \{e'_1, e'_2, ..., e'_d\} \cup E^A$
- $E^A = \{\{v_i, v'_i\}; i = 1, ..., d\} \cup \{\{v_i, v'_{i-1}\}; i = 2, ..., d\} \cup \{\{v'_i, v'_{i+1}\}; i = 1, ..., d-1\}$

The edges $e'_i \in E'$ ("image edges") are obtained from $e_i \in E$ by replacing the vertex v by v_i. The edges in E^A will be called "auxiliary".

Lemma 3.2.2 *The graph obtained by even splitting can be embedded again in the same surface.*

Proof:
The transformation replaces a vertex $v \in V$ by a cluster of $2d$ vertices and $3d - 2$ edges. The cluster itself is a planar graph which can be embedded in a small neighborhood of the original location of the vertex v. The images of the edges incident with v can be embedded in the same way as they were in the original graph. $\square$

Definition 3.2.3 *Let $G = (V, E)$ be a graph. $G_s = (V_s, E_s)$ is then the graph obtained by successive even splitting of all vertices in V. If there are variables x_e assigned to edges $e \in E$, we assign the same variables to their images in E_s: $x_{e'} = x_e$. The auxiliary edges $f \in E_s$ get assigned $x_f = 1$.*

Theorem 3.2.4 *If G is a graph of genus g, G_s has genus g as well. With the assigment of variables described above, the generating function of perfect matchings of G_s is equal to the generating function of eulerian subgraphs of G:*

$$\mathcal{P}(G_s) = \mathcal{E}(G)$$

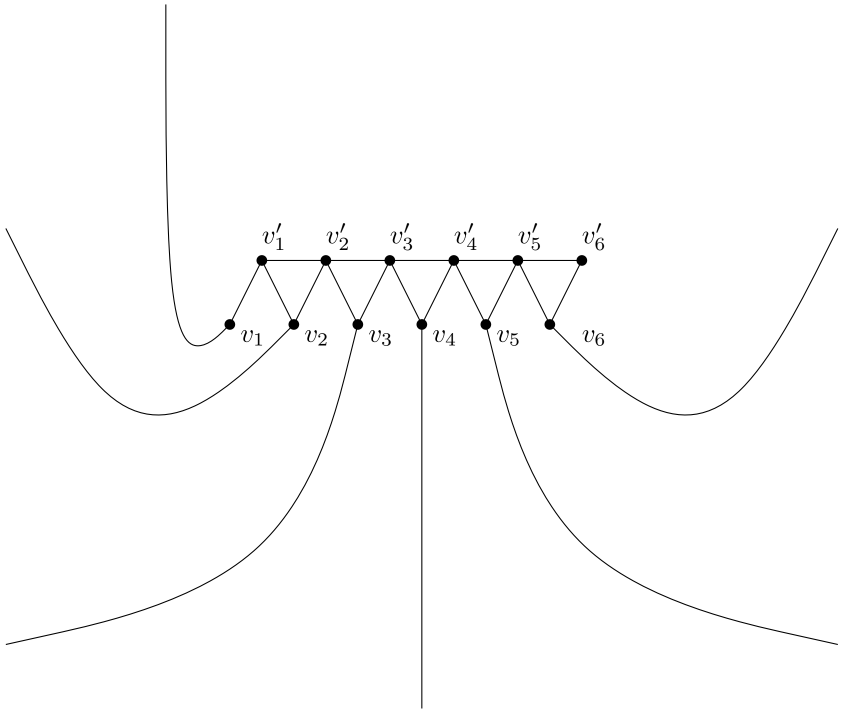

Figure 3.1: Even splitting

Proof:

From the definition of even splitting and Lemma 3.2.2, it follows that the resulting graph G_s can be embedded again in the same surface.

If M is a perfect matching in G_s, it must cover each of its vertices exactly once. Because the cluster replacing every vertex has an even number of vertices, and any of the auxiliary edges which is in M covers a pair of vertices of the cluster, there remain an even number of vertices to be covered by the image edges incident with the cluster. Therefore, every cluster coincides with an even number of image edges which are in M; in other words, these edges form the image of an eulerian subgraph of G.

Vice versa, the image of any eulerian subgraph of G can be extended (uniquely) by adding some of the auxiliary edges in G_s to make a perfect matching in G_s. Thus, there is a one-to-one correspondence between the perfect matchings of G_s and the eulerian subgraphs of G. As all the auxiliary edges have variables equal to 1, the corresponding terms contributing to either of the generating functions are equal. Consequently, the two generating functions are equal. $\square$

3.3 Perfect Matchings and Pfaffians

This section deals with the generating function of perfect matchings and an efficient way to compute it. We will start out with planar graphs for which the solution was found by Kasteleyn [8]. We shall see that the generating function can be expressed in an algebraic form very similar to a determinant.

Definition 3.3.1 *Let $G = (V, E)$ be a graph with $2n$ vertices, $\{x_e; e \in E\}$ variables assigned to edges and D an orientation (a fixed ordering of the two vertices of each edge). Let $A(D)$ denote a $2n \times 2n$ antisymmetric matrix where*

- $a_{ij} = x_{\{i,j\}}$*, if (i, j) is a directed edge in D*
- $a_{ij} = -x_{\{i,j\}}$*, if (j, i) is a directed edge in D*
- $a_{ij} = 0$*, if $\{i, j\}$ is no edge*

The Pfaffian of this matrix is defined as

$$Pf(A(D)) = \sum_P sgn(P) a_{i_1 j_1} a_{i_2 j_2} ... a_{i_n j_n}$$

where the sum is taken over all partitionings of the index set $\{1, 2, ..., 2n\}$ into pairs $i_1 < j_1$, $i_2 < j_2$,...,$i_n < j_n$ and $sgn(P)$ is the sign of the permutation $(i_1, j_1, i_2, j_2, ..., i_n, j_n)$.

Note: The ordering of the pairs is specified here only to remove multiple occurrences of the same partitioning. The resulting contribution to the Pfaffian does not depend on it, as swapping a pair of indices $\{i_k, j_k\}$ reverses both the sign of the permutation and the sign of the factor $a_{i_k j_k}$.

It can be observed that the non-zero terms contributing to the Pfaffian are exactly those corresponding to perfect matchings of G (partitionings of the set of vertices into pairs where there is an edge between each pair). However, each of them comes with a positive or negative sign, depending on the orientation D. What we would like to find is a special orientation which produces positive signs for all perfect matchings. Then the Pfaffian would be exactly equal to the generating function of perfect matchings.

Definition 3.3.2 *Let M and P be two perfect matchings. An alternating cycle is a cycle of even length which contains edges from M and P alternately. In a directed graph, if the number of edges oriented "clockwise" around a cycle is even, the cycle is called "clockwise even", otherwise it is called "clockwise odd". (Note that for cycles of even length, the definition does not depend on which orientation we call "clockwise".)*

Lemma 3.3.3 *Let M be a fixed perfect matching of graph G and D its orientation. Let $sgn(M)$ be the sign of the contribution of M to $Pf(A(D))$ and $sgn(D)$ be the sign of another perfect matching D contributing to the Pfaffian. The set of edges*

$$M \Delta P = (M \setminus P) \cup (P \setminus M)$$

is a union of disjoint alternating cycles. Let $\delta(M, P)$ denote the number of clockwise even cycles in $M\Delta P$. Then

$$sgn(P) = (-1)^{\delta(M,P)} sgn(M)$$

Proof:

The symmetric difference of two perfect matchings is a subgraph where each vertex has degree 2 or 0. Therefore $M\Delta P$ can be decomposed into disjoint cycles and as no vertex can be incident with two edges of the same matching, the cycles are alternating.

The sign of the term arising from the perfect matching M is equal to the sign of the permutation $(i_1, j_1, ..., i_n, j_n)$ where $M = \{(i_k, j_k); k = 1, ..., n\}$, each pair ordered according to the orientation D. This ordering can be transformed into that corresponding to the perfect matching P by a sequence of permutations. Each of these is responsible for one alternating cycle from $M\Delta P$, which means it must shift the vertices around this cycle and reverse the orientation of some edges if necessary. It can be verified that such a permutation has a positive sign if the cycle is clockwise odd, and a negative sign if it is clockwise even. So the overall change of sign is $(-1)^{\delta(M,P)}$. □

This observation leads to the following definition and theorem (due to Kasteleyn).

Definition 3.3.4 *An orientation D is called Pfaffian, if the alternating cycles of any two perfect matchings are all clockwise odd.*

Theorem 3.3.5 *Let G be a graph with a Pfaffian orientation D. Then*

$$Pf(A(D)) = (\pm)\mathcal{P}(G)$$

Proof:

Because the orientation is Pfaffian, the alternating cycles formed by any two perfect matchings are clockwise odd. By the lemma, all the terms constituting the Pfaffian have the same sign, which is either positive or negative. Which of the two alternatives occurs depends only on the numbering of vertices in the matrix $A(D)$. If we want to make sure that the sign will be positive, we can pick an arbitrary perfect matching, calculate the sign of the corresponding permutation, and if it comes out negative, just exchange any two vertices in the numbering. □

Theorem 3.3.6 *Every planar graph has a Pfaffian orientation.*

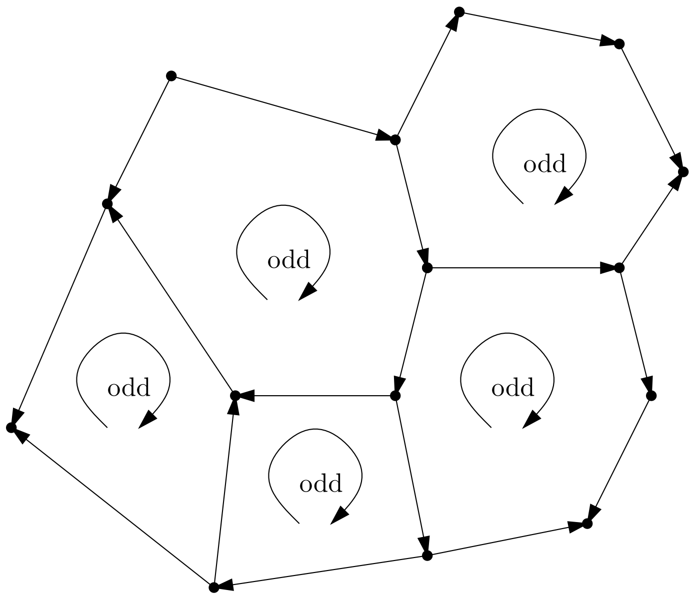

Figure 3.2: Pfaffian orientation

Proof:

First, we shall suppose that the graph is 2-connected. We will try to find such an orientation that every inner face is clockwise odd. This can be accomplished by starting out from any inner face and adding adjacent faces with suitable edge orientations. When adding a new face, there is always at least one new edge whose orientation can be chosen so that the face is clockwise odd. This process continues until we cover the whole graph. (It is not necessary that the outer face comes out clockwise odd.)

If the graph is not 2-connected, we can proceed in almost the same way, effectively ignoring the vertices and edges which do not belong to any cycle (their orientation is irrelevant here). The only difference is that the faces containing an odd number of such vertices must be clockwise even, whereas those containing an even number of them must be clockwise odd.

This scheme implies that every cycle encircling an even number of vertices is clockwise odd, whereas any cycle encircling an odd number of vertices is clockwise even. As any alternating cycle of two perfect matchings must contain an even number of vertices (there must be a perfect matching inside the circle, and no edges may cross it), all alternating cycles are clockwise odd. □

3.4 The Galluccio-Loebl Theorem

When we find a Pfaffian orientation, we can just calculate the Pfaffian of the resulting adjacency matrix and we get the generating function of perfect matchings. Supposing this can be done efficiently, we are able to solve the MAX-CUT problem, or produce the entire partition function for any planar graph.

Now that we can tackle planar graphs, what if we are presented with a graph G which is not planar, but at least it has a small genus? (I.e., it can be embedded in the surface of a sphere with a few "handles".) Then we cannot rely on the Pfaffian orientation, as it may not always exist. However, the question is answered by the *Galluccio-Loebl theorem* [2] which states that G has 4^g *relevant orientations* such that a suitable linear combination of the corresponding Pfaffians yields the desired result.

Let us define surfaces and graphs of genus g in a slightly different form than usual:

Definition 3.4.1 *A surface of genus g consists of a base B_0 and $2g$ bridges B^i_j, $i = 0, ..., g-1; j = 0, 1$.*

1. *The base B_0 is a convex $4g$-gon with vertices $a_0, ...a_{4g-1}$.*
2. *The bridge B^i_j is a quad with vertices k, l, m, n where the line segment $[k, l]$ is identified with $[a_{4i+j}, a_{4i+j+1}]$ and $[m, n]$ is identified with $[a_{4i+j+2}, a_{4i+j+3}]$.*

Definition 3.4.2 *G is a g-graph if it can be embedded in a surface of genus g, so that*

1. *All the vertices are contained in the base B_0.*
2. *Every edge uses at most one bridge B^i_j.*

Let E_0 denote the edges embedded entirely inside the base B_0 and E^i_j the edges using bridge B^i_j. Let $G_0 = (V, E_0)$ and $G^i_j = (V, E_0 \cup E^i_j)$.

Definition 3.4.3 *G is a proper g-graph if it is a g-graph satisfying the following conditions*

1. *The outer face of G_0 is a cycle C_0 embedded on the boundary of B_0.*
2. *The edges in E^i_j are embedded entirely inside the bridge B^i_j and their endpoints are located on the boundary of B_0.*
3. *Every vertex is incident with at most one edge outside of E_0.*
4. *G_0 has a perfect matching M_0.*

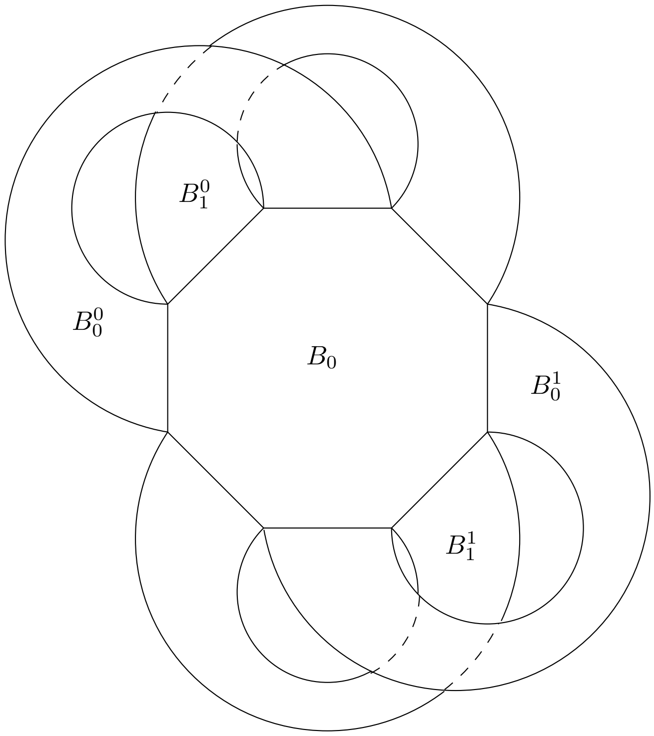

Figure 3.3: A surface of genus 2

For the purpose of the theorem, we shall require that the graph G be a proper g-graph. However, any g-graph can be transformed into a proper g-graph by splitting the edges crossing the boundary of B_0 into paths with new vertices on the boundary, and adding zero-weight edges if necessary to create a cycle on the boundary of B_0. Then we obtain a proper g-graph with only a linear increase in the number of vertices and edges, and the same generating function of perfect matchings.

Definition 3.4.4 *Let G be a proper g-graph. G_0 is a planar subgraph of G, therefore it has a Pfaffian orientation D_0. In addition, we can choose D_0 so that M_0 has a positive sign in the Pfaffian of the matrix $A(D_0)$.*

Furthermore, G_j^i can be embedded in the plane in the following way: Take the planar embedding of G_0 and add the bridge B_j^i so that it is contained in the outer face of G_0. Then the orientation D_0 can be supplemented with an orientation D_j^i of the edges in E_j^i so that the orientation $D_0 \cup D_j^i$ is still Pfaffian.

Finally, we define the 4^g relevant orientations of the entire graph G: The relevant orientation D_r (where $r \in \{+1, -1\}^{2g}$) is composed of D_0 and $r_{2i+j} D_j^i$ for $i = 0, ..., g-1$ and $j = 0, 1$ (where $-D_j^i$ means the reversed orientation D_j^i).

Theorem 3.4.5 (Gallucio, Loebl) *Let G be a proper g-graph and $D_r, r \in \{+1, -1\}^{2g}$ its relevant orientations. Let us define*

$$c_r = \prod_{i=0}^{g-1} c(r_{2i}, r_{2i+1})$$

where

$$c(+1,+1) = \frac{1}{2}$$
$$c(+1,-1) = \frac{1}{2}$$
$$c(-1,+1) = \frac{1}{2}$$
$$c(-1,-1) = -\frac{1}{2}$$

Then the generating function of perfect matchings of G is equal to a linear combination of 4^g Pfaffians:

$$\mathcal{P}(G) = \sum_{r \in \{+1,-1\}^{2g}} c_r Pf(A(D_r))$$

where $A(D_r)$ are the adjacency matrices corresponding to the relevant orientations.

The proof of this theorem is difficult and it can be found in [2]. Its great significance is that the number of Pfaffians, though increasing exponentially with g, is constant for a given g. So, assuming Pfaffians can be calculated efficiently, the generating function of perfect matchings can be computed in polynomial time for any graph of bounded genus. Needless to say, this approach is practical only for very small values of g. For example, toroidal graphs (i.e., 1-graphs) can be treated in the same way as planar graphs, only there will be four relevant orientations and four Pfaffians instead of one.

Chapter 4

Modular Arithmetic

The approach described in Chapter 3 leads to an efficient algorithm indeed, but it still leaves a number of questions open:

- How to treat the matrix elements? (They are symbolic functions.)
- How to calculate the Pfaffian in a polynomial number of steps?

4.1 Data Structure Choices

The first question involves a decision on the data structures, as well as the type of programming language to use. Consider the data flow of the program:

- **Input:** Graph G with edge weights J_{ij}.
- **Intermediate data:** Matrices containing symbolic functions of a single variable.
- **Output:** Coefficients of the partition function.

At first glance, the obvious approach seems to be: treat the matrix elements as polynomials, perform all arithmetic operations symbolically, and obtain the result in a symbolic form. For this purpose, one would most probably use a symbolic computation package like Mathematica. Indeed, this would be feasible, and perhaps the easiest way to achieve our goal. Nonetheless, I had some objections against it which made me rule it out in the end. My aim was maximum performance and robustness, so that the program could be applied to as large input graphs as possible. Using a symbolic computation package (instead of a general purpose programming language) would inevitably bring a substantial loss of speed and increase the amount of required memory. Also, it is not apparent how this symbolic computation could be parallelized.

My solution is this: we know the partition function is a polynomial in variable $x = e^{-\beta}$ with integer coefficients. Instead of computing the entire polynomial at once, we can evaluate it at sufficiently many distinct points, and then we can obtain the polynomial coefficients by interpolation. This turns the problem of treating symbolic functions into ordinary numeric calculation, albeit still cumbersome in a sense. As the expected results are exponentially large (the total number of cuts in a graph with n vertices is 2^n), I had to decide how to handle these huge numbers. Instead of using floating point (which is imprecise) or long integer arithmetic (which is possible, but slow and consuming a lot more memory), I decided to split the task once more. The idea is to use only elementary operations which can be performed in constant time - the basic operations in a finite field. Of course, the results are elements of the finite field as well, but this loss of information can be amended, as I will show. The major advantage is that the algorithm employs only simple integer operations and it can be implemented easily in any general purpose programming language, like C++.

The following lemma justifies this approach in a formal way:

Lemma 4.1.1 *Let $P(x)$ be a polynomial of degree n with integer coefficients, $m > n$ prime, and $x_1, x_2, ..., x_n$ distinct elements of the finite field $\mathcal{GF}(m)$. Then there exists a unique polynomial $Q(x)$ of degree n over $\mathcal{GF}(m)$ such that*

$$Q(x_i) = P(x_i) \bmod m, i = 0...n$$

Moreover, the coefficients of $Q(x)$ are equal to the coefficients of $P(x) \bmod m$.

Proof:

Let $P(x_i) = y_i$. Define $Q(x)$ by Lagrangian interpolation:

$$Q(x) = \sum_{i=0}^{n} \frac{\prod_{j \neq i} (x - x_j)}{\prod_{j \neq i} (x_i - x_j)} (y_i \bmod m)$$

where all operations are performed in $\mathcal{GF}(m)$. The expression is well defined because

$$(i \neq j \Longrightarrow x_i \neq x_j) \Longrightarrow \prod_{j \neq i} (x_i - x_j) \neq 0$$

and any non-zero element is invertible in a field.

$Q(x)$ satisfies the conditions $Q(x_i) = y_i \bmod m$. Now let us suppose there is another polynomial $R(x)$ over $\mathcal{GF}(m)$, such that $R(x_i) = y_i \bmod m, i = 0...n$. Then we define the difference polynomial $Z(x) = R(x) - Q(x)$, and it follows that

$$Z(x_i) = R(x_i) - Q(x_i) = 0 \pmod{m}, i = 0...n$$

However, a non-zero polynomial of degree n has at most n roots in any field, therefore $Z(x)$ is a zero polynomial and $R(x) = Q(x)$.

The polynomial $\overline{P}(x)$ whose coefficients are the coefficients of $P(x) \bmod m$, satisfies the same conditions, so by the same argument $\overline{P}(x) = Q(x)$. □

Note: Our "polynomial" can actually contain negative powers of x which would cause trouble in the interpolation formula. Therefore it is necessary to make the exponents positive before interpolation by multiplying the function by x^k for a suitable k. The degree of the polynomial is then confined by the difference of maximum and minimum cut weights. As we do not know the values of MAX-CUT and MIN-CUT in advance, we can only estimate them. For now, it will suffice to know that if the edge weights are bounded by a constant, the degree is bounded by $O(m)$, where m is the number of edges. In graphs of bounded genus, this is the same as $O(n)$, where n is the number of vertices.

4.2 Randomized MAX-CUT

To summarize the lemma, if we can supply the values of our polynomial modulo some prime p, we can also extract its coefficients modulo p. For now, we will leave aside the question of how to evaluate the polynomial, and we will suppose we have acquired the partial results (modulo p). What good can they be? Instead of a number a, we only get $a \bmod p$. There is certainly less information here than we need, even if we only want to detect whether a given coefficient is zero. (As you may recall, solving MAX-CUT amounts to detecting the first non-zero coefficient in the generating function.) However, we can still make a guess. If $a \bmod p \neq 0$, then surely $a \neq 0$. If $a \bmod p = 0$, we cannot say for sure. But if p is large enough, it is unlikely that $a \bmod p = 0$, while $a \neq 0$, because this can happen only if a is divisible by p. To estimate our chance of failing to detect a non-zero coefficient in this way, I give the following lemma.

Lemma 4.2.1 *Suppose $0 < a \leq N$, and we pick a prime $p \leq M$ at random. Then the probability that a is divisible by p is*

$$\mathcal{P} = O(\frac{\log M \log N}{M \log \log N})$$

Proof:
We use the following facts from number theory:

- The number of primes below or equal to M is $P(M) = \Theta(\frac{M}{\log M})$

- The sum of logarithms of the primes below or equal to M is $S(M) = \Theta(M)$

If $0 < a \leq N$, and a is divisible by K different primes $p_1, p_2, ..., p_K$, then

$$\log N \geq \log a \geq \sum_{i=1}^{K} \log p_i \geq S(q)$$

where $S(q)$ is the sum of logarithms of the K smallest primes, the largest of which is q. But the inequality implies that $q = O(\log N)$ and therefore

$$K = P(q) = O(\frac{q}{\log q}) = O(\frac{\log N}{\log \log N})$$

If we pick one of the primes below or equal to M at random, the probability that we choose one of a's factors by misfortune is

$$\mathcal{P} \leq \frac{K}{P(M)} = O(\frac{\log M \log N}{M \log \log N})$$

□

Now we can find the value of MAX-CUT with a randomized algorithm, and regulate our chances of success by selecting the modulus from a wide enough range of primes.

Lemma 4.2.2 *By calculating the generating function of cuts of a graph on n vertices modulo a randomly chosen prime below or equal to M, we can find the correct value of MAX-CUT with a probability of error*

$$\mathcal{P} = O(\frac{n \log M}{M \log n})$$

Proof:

Let us select a random prime $p \leq M$ and calculate the generating function of cuts modulo p. The MAX-CUT value can be found as the highest power of x with a non-zero coefficient in the polynomial. The powers of x above this have zero coefficients, and the algorithm will make no mistake here. It can return an incorrect answer only if it fails to detect the non-zero coefficient corresponding to the value of MAX-CUT. This coefficient is bounded by $N = 2^n$, so the probability of error is, by the previous lemma

$$\mathcal{P} = O(\frac{\log M \log N}{M \log \log N}) = O(\frac{n \log M}{M \log n})$$

□

4.3 The Chinese Remainder Theorem

Sometimes we may require the complete exact results, which are integer coefficients ranging from 0 to 2^n. Even then, modular arithmetic can be useful. Consider the *Chinese Remainder Theorem*:

Theorem 4.3.1 *Let us select k different prime numbers $p_1, p_2, ..., p_k$. Then for any $a_1, a_2, ..., a_k$, $0 \leq a_i < p_i$, there is exactly one x, $0 \leq x < \prod_{i=1}^{k} p_i$, such that:*

$$x \bmod p_i = a_i, i = 1...k$$

This is a well-known fact, nevertheless I give the proof here, so that it becomes apparent how the number x can be constructed from its "coordinates".

Proof: Because $P = \prod_{i=1}^{k} p_i$ is the smallest common multiple of $p_1, p_2, ..., p_k$, any two numbers x, y satisfying

$$x \bmod p_i = y \bmod p_i = a_i, i = 1...k$$

must differ by a multiple of P. Any solution can be clamped into the interval $0 \leq x < P$ by adding a suitable multiple of P and such a "normalized" solution is unique. It remains to show that a solution exists.

To prove this constructively, let us define

- $r_i = \prod_{j \neq i} p_j$
- $q_i = {r_i}^{-1}$ in $\mathcal{GF}(p_i)$, i.e. $(q_i r_i) \bmod p_i = 1$
- $x = \sum_{i=1}^{n} a_i q_i r_i$

Then x is a solution, because whenever $i \neq j$, r_i is divisible by p_j, and therefore

$$x \bmod p_j = (a_j q_j r_j) \bmod p_j = (a_j((q_j r_j) \bmod p_j)) \bmod p_j = a_j \bmod p_j = a_j$$

$\square$

Note: It is not guaranteed that the solution constructed like this falls into the interval $0 \leq x < P$. In fact, typically it would be considerably larger than P. If we want a normalized solution, we can proceed in the following way (without dividing large numbers modulo P):

1. For $i = 1...k$, calculate $b_i = (a_i q_i) \bmod p_i$
2. Calculate $x = (\sum_{i=1}^{n} b_i r_i) \bmod P$; here each term $b_i r_i$ is below P, so we only have to subtract P after each addition which overflows P.

All in all, we can obtain *exact* results, if we choose a sufficient number of primes $p_1, p_2, ..., p_k$, so that

$$\prod_{i=1}^{k} p_i > 2^n$$

For example, if we can use primes from the interval $[\frac{M}{2}; M]$, then (assuming there are enough primes to choose from), we need

$$k = O(\frac{n}{\log M})$$

different primes. Since all the results are confined to the interval $[0; 2^n]$, the partial results modulo $p_1, p_2, ..., p_k$ determine the exact results uniquely. Furthermore, the exact results can be calculated from the partial results efficiently, using long integer addition and multiplication. The complete algorithm will consist of several repetitions of the computation producing the polynomial - in each of the finite fields. On the other hand, these computations are absolutely independent - if possible, they can be run in parallel.

Chapter 5

Computing Pfaffians

Being armed with the results of Chapter 4, we can assume that we reside in a finite field $\mathcal{GF}(m)$, and our sole purpose is to calculate a single value of the partition function at a specific point. As we know, the partition function can be transformed into the generating function of perfect matchings of a modified graph:

$$Z(\beta) = 2^n \ \left(\prod_{\{i,j\}\in E} \cosh(\beta J_{ij}) \right) \ \ \mathcal{P}(G', \tanh(\beta J_{ij}))$$

where the second argument in the generating function denotes the modified edge weights. The function is a polynomial in variable $x = e^{-\beta}$, and by substitution we get

$$Z(\beta) = P(x) = 2^n \ \left(\prod_{\{i,j\}\in E} \frac{x^{J_{ij}} + x^{-J_{ij}}}{2} \right) \ \ \mathcal{P}(G', \frac{x^{-J_{ij}} - x^{J_{ij}}}{x^{J_{ij}} + x^{-J_{ij}}}))$$

Moreover, the generating function of perfect matchings can be expressed as a linear combination of Pfaffians:

$$\mathcal{P}(G) = \sum_{r\in\{+1,-1\}^{2g}} c_r Pf(A(D_r))$$

where the Pfaffian of a matrix is defined as a sum over all partitionings of the index set into pairs:

$$Pf(A(D)) = \sum_{P} sgn(P) a_{i_1 j_1} a_{i_2 j_2} ... a_{i_n j_n}$$

The sum consists of an exponential number of terms, but (just like a determinant) it can be calculated very efficiently. My approach has been motivated by *Gaussian elimination*, and it proceeds in a similar fashion.

5.1 Basic Properties of Pfaffians

The first step is to define a basic operation which can be used to transform the matrix into a special form, while the value of the Pfaffian is preserved. For instance, determinants are invariant under elementary row operations and these can be used in Gaussian elimination to calculate a determinant or to solve a system of linear equations. Pfaffians must be treated somewhat differently, though.

Definition 5.1.1 *Let A be an antisymmetric matrix of size $2n \times 2n$ over a field $\mathcal{F}$. A cross of the matrix is the union of a row and a column of the same index. I.e., the k-th cross is the following set of elements:*

$$A_k = \{a_{ik}; 1 \leq i \leq 2n\} \cup \{a_{kj}; 1 \leq j \leq 2n\}$$

Definition 5.1.2 Cross operations:

Multiplying a cross A_k by a scalar $\alpha \in \mathcal{F}$ means simply multliplying each element of A_k by α.

Swapping crosses A_k and A_l of an antisymmetric matrix A means exchanging both the respective rows and columns. Another way to regard the swap operation is that it exchanges the values of k and l in both of the index positions. The resulting matrix B is antisymmetric again; it has elements:

- $b_{ij} = a_{ij}$, *if* $\{i, j\} \cap \{k, l\} = \emptyset$
- $b_{ik} = a_{il}, b_{il} = a_{ik}$, *if* $i \notin \{k, l\}$
- $b_{kj} = a_{lj}, b_{lj} = a_{kj}$, *if* $j \notin \{k, l\}$
- $b_{kl} = a_{lk}, b_{kl} = a_{kl}$
- $b_{kk} = b_{ll} = 0$

Adding cross A_k to cross A_l means adding first the k-th row to the l-th one, and then adding the respective columns. As the reader may verify, this order of operations is arbitrary (due to antisymmetricity), and the matrix remains antisymmetric. The result C can be described as

- $c_{ij} = a_{ij}$, *if* $l \notin \{i, j\}$
- $c_{il} = a_{il} + a_{ik}$, *if* $i \notin \{k, l\}$
- $c_{lj} = a_{lj} + a_{kj}$, *if* $j \notin \{k, l\}$

- $c_{kl} = a_{kl}$, $c_{lk} = a_{lk}$
- $c_{ll} = 0$

Lemma 5.1.3 *The Pfaffian of a matrix is a multilinear function of its crosses.*

Proof:
Let A, B be two antisymmetric matrices over a field $\mathcal{F}$ which differ only in their k-th cross, and $\alpha, \beta \in \mathcal{F}$. Let C be a matrix which is equal to A and B, except for the k-th cross where

$$C_k = \alpha A_k + \beta B_k$$

The Pfaffian of C is equal to

$$Pf(C) = \sum_P sgn(P) c_{i_1 j_1} c_{i_2 j_2} ... c_{i_n j_n}$$

The index k appears exactly once in each partitioning, so we can assume it is always one of the indices i_1, j_1. Then the element $c_{i_1 j_1}$ falls into the k-cross, so that:

$$Pf(C) = \sum_P sgn(P)(\alpha a_{i_1 j_1} + \beta b_{i_1 j_1}) c_{i_2 j_2} ... c_{i_n j_n}$$

But the remaining elements $c_{i_2 j_2}, ..., c_{i_n j_n}$ are equal in A, B and C:

$$Pf(C) = \alpha \sum_P sgn(P) a_{i_1 j_1} ... a_{i_n j_n} + \beta \sum_P sgn(P) b_{i_1 j_1} ... b_{i_n j_n} = \alpha Pf(A) + \beta Pf(B)$$

□

Lemma 5.1.4 *Swapping two crosses of a matrix reverses the sign of the Pfaffian.*

Proof:
Let A be an antisymmetric matrix, and A' the matrix obtained from A by swapping crosses A_k and A_l. If P is a partitioning of the index set into pairs $\{i_1, j_1\}, ...\{i_n, j_n\}$, and P' a partitioning into pairs $\{i'_1, j'_1\}, ...\{i'_n, j'_n\}$ obtained from P by swapping indices k and l, then

$$a'_{i'_1 j'_1} ... a'_{i'_n j'_n} = a_{i_1 j_1} ... a_{i_n j_n}$$

while

$$sgn(P') = -sgn(P)$$

Also, this operation maps the set of all partitionings onto itself, so that

$$Pf(A') = \sum_{P'} sgn(P') a'_{i'_1 j'_1} ... a'_{i'_n j'_n} = - \sum_{P} sgn(P) a_{i_1 j_1} ... a_{i_n j_n} = -Pf(A)$$

□

Lemma 5.1.5 *Adding a multiple of a cross to another cross does not change the Pfaffian of the matrix.*

Proof: If A' is obtained from A by adding the cross A_k multiplied by $\alpha \in \mathcal{F}$ to another cross A_l, then by multilinearity:

$$Pf(A') = Pf(A) + \alpha Pf(A^*)$$

where A^* contains two equal crosses: $A_l^* = A_k^* = A_k$. However, swapping these crosses should reverse the sign of $Pf(A^*)$ which is only possible if

$$Pf(A^*) = 0$$

Therefore

$$Pf(A') = Pf(A)$$

□

5.2 Pfaffian Elimination

Now we would like to use these operations to transform our matrix into a form where the Pfaffian can be determined trivially. For determinants, the obvious goal is a diagonal matrix. For Pfaffians, it is a form where only one partitioning of the index set into pairs produces a non-zero term: $\{1, 2\}, \{3, 4\}, ..., \{2n-1, 2n\}$. The only non-zero elements of the matrix should be $a_{12} = -a_{21}, a_{34} = -a_{43}, ...$ and the Pfaffian would then be equal to $a_{12} a_{34} ... a_{2n-1,2n}$.

When computing determinants, though, we need not strive for a form as perfect as a diagonal matrix. It is enough if the matrix contains non-zero elements only in its upper triangle. Then it is still just the diagonal elements that contribute to the determinant. Similarly, for Pfaffians it suffices to achieve a form satisfying the following lemma.

Lemma 5.2.1 *Let A be an antisymmetric matrix where*

$$\forall i = 2k-1, 1 \le k \le n; \forall j, 2k+1 \le j \le 2n \Longrightarrow a_{ij} = 0$$

Then

$$Pf(A) = \prod_{k=1}^{n} a_{2k-1,2k}$$

Proof:

Let us consider a partitioning P of $\{1, 2, ..., 2n\}$ into pairs. We will infer that there is only one P producing a non-zero term in the expansion of the Pfaffian.

Let us denote by $\{i_1, j_1\}$ the pair containing the index 1; for example, $i_1 = 1$. What are the possible choices for j_1 if we require that $a_{i_1,j_1} \neq 0$? $j_1 = 1$ is out of the question, and for any $j_1 > 2$, the condition of the lemma implies that $a_{i_1,j_1} = 0$. Therefore, the only possible choice is $j_1 = 2$.

Now we proceed by induction. Supposing $i_k = 2k-1, j_k = 2k$ for $k = 1, ..., m-1$, some of the remaining pairs must contain the index $2m-1$; let it be $i_m = 2m-1$. Indices $1, 2, ..., 2m-2$ are no longer relevant, since they have been used in some of the previous pairs. The condition implies that $a_{2m-1,j} = 0$ for any $j \ge 2m+1$, so the only possibility for a non-zero a_{i_m,j_m} is $j_m = 2m$.

Thus the only partitioning producing a non-zero contribution to the Pfaffian is $P = \{\{1, 2\}, \{3, 4\}, ..., \{2n-1, 2n\}\}$ and

$$Pf(A) = a_{12}a_{34}...a_{2n-1,2n}$$

□

It remains to show how this form can be achieved in a polynomial number of steps. My algorithm is analogous to Gaussian elimination and its time complexity is similar. (We will pay closer attention to the running time analysis in the next section.) It proceeds from the first cross to the last one, using the multiply-and-add cross operation to eliminate all the necessary elements as required by the lemma. In contrast to traditional numerical elimination, the plague of numerical instability need not worry us - all operations in a finite field are precise! Therefore, the only condition we impose on the pivot (which is the only element used in a divide operation) is that it is not equal to zero. If no such pivot can be found, the Pfaffian must be equal to zero, because in this case there is no partitioning producing a non-zero term. Otherwise, the value of the pivot becomes part of the result, which is accumulated in the variable Pf. In the end, the matrix acquires the form satisfying Lemma 5.2.1 and $Pf = a_{12}a_{34}...a_{2n-1,2n}$ is the Pfaffian of the matrix.

Algorithm: *(Pfaffian elimination)*

1. $Pf := 1$
2. $k := 1$
3. If $a_{2k-1,2k} = 0$, find $m > 2k$ such that $a_{2k-1,m} \neq 0$, swap the crosses A_{2k} and A_m, and set $Pf := -Pf$. If there is no such m, then $Pf := 0$; STOP.
4. $Pf := a_{2k-1,2k}Pf$
5. For $j := 2k+1, ..., 2n$; if $a_{2k-1,j} \neq 0$, eliminate it by a cross operation:

$$A_j := A_j - \frac{a_{2k-1,j}}{a_{2k-1,2k}} A_{2k}$$

6. If $k < n$, $k := k+1$, GOTO 3. Otherwise Pf is the result; STOP.

5.3 Elimination Complexity

As it stands, the algorithm needs at most $O(n^2)$ cross operations, each of which consists of $O(n)$ finite field operations which can be carried out in constant time. (Addition and multiplication are trivial, and division can be reduced to multiplication with a precalculated table of inverse elements.) The total running time is therefore at most $O(n^3)$. However, this crude estimate does not take into account the special structure of our matrix, and it can be improved considerably.

In numerical analysis, solving linear systems and computing determinants are some of the most common tasks, and they have been paid much attention in the past. The traditional approach to these problems is based on Gaussian elimination which has been known for a long time, and it still remains a viable option. It provides a safe way to obtain the result in $O(n^3)$ time and $O(n^2)$ space, regardless of the nature of the matrix. In practice, though, we usually deal with matrices which are special in some way. Often, they contain a large number of zero elements. If the number of non-zero elements is only linear in the size of the matrix (e.g., there is a constant number of non-zero elements in each row), we say that the matrix is *sparse*. Sometimes the non-zero elements form a regular pattern, allowing a specialized elimination procedure which can be very fast. Here, we will consider a more general method which was invented by Alan George [4] and later refined by Lipton, Rose and Tarjan [9]. Fortunately, their approach, though intended for Gaussian elimation, can be geared to Pfaffian elimination without much difficulty.

As the reader might expect, the speed-up can be achieved by avoiding explicit manipulation of the zeros which appear frequently in our matrix. We can also save a lot of space if we use a more sophisticated data structure than a simple array of n^2 elements, most of which are zero anyway. Nevertheless, there is more to this than meets the eye; the major difficulty comes from the fact that the number of non-zero elements increases during the elimination, and it is not trivial to estimate the total number of operations.

It will be convenient to describe the structure of the matrix during the elimination process by a *graph-theoretic model.* This will be particularly intuitive in our case, since the matrix originated from a graph anyway.

Definition 5.3.1 *Let A be an antisymmetric matrix of size $2n \times 2n$. The structure of the matrix is represented by a graph $G = (V, E)$:*

$$V = \{1, 2, ..., 2n\}$$
$$E = \{\{i, j\} \subset V; a_{ij} \neq 0\}$$

When we perform Pfaffian elimination, all elements which get modified as a result of a cross operation may become non-zero. The elements which are non-zero in the original matrix or possibly become non-zero during the elimination are called "fill-in". (Here we neglect the possibility that non-zero elements may cancel out and produce a zero; such elements are still treated as "possibly non-zero".) The corresponding set of edges E^* defines the fill-in graph $G^* = (V, E^*)$.

Clearly, the fill-in graph is related to the possible efficiency of our elimination algorithm. All elements which become non-zero have to be dealt with somehow, so the more edges G^* contains, the worse our algorithm will perform. The complexity of the algorithm is determined exactly by the fill-in graph, as shown in the following lemma.

Lemma 5.3.2 *Let G^* denote the fill-in graph of matrix A, m the number of edges of G^* and $d(i)$ the degree of vertex i in G^*. Then the space required by Pfaffian elimination is*

$$O(m)$$

and the total number of operations is

$$O(\sum_{k=1}^{n} d(2k-1)d(2k))$$

Proof:

The edges of G^* represent all the relevant elements of A, and it is possible to devise a data structure encoding the matrix which needs only $O(m)$ space. (See Section 6.5 for details.)

As for the number of operations, please refer to the description of the algorithm in the previous section: To eliminate the cross A_{2k}, we need to perform a cross multiplication and addition costing $d(2k)$ operations for every $j > 2k$ where $a_{2k-1,j} \neq 0$. The number of such elements is at most $d(2k-1)$, so we have $O(d(2k-1)d(2k))$ operations for every repetition of the main cycle.

There are constant overheads associated with looping over the rows and columns, and occasionally a cross swapping which costs $O(d(2k-1))$, but these are terms of lesser order. (Unless some $d(i) = 0$ which, of course, implies that $Pf(A) = 0$ without calculating anything.)

The total number of operations is bounded by $O(\sum_{k=1}^{n} d(2k-1)d(2k))$. □

How can we affect the density of G^*? The choice that we have is *the ordering of the rows and columns* of A. As we know, swapping two crosses of A only reverses the sign of the Pfaffian, and by repeating this operation, we can achieve any permutation π. But the fill-in graph depends on π and so we can try to optimize for minimum fill-in.

Definition 5.3.3 *Let A be an antisymmetric matrix of size $2n \times 2n$ and*

$$\pi : \{1, 2, ..., 2n\} \to \{1, 2, ..., 2n\}$$

a permutation. We will denote by $A^{(\pi)}$ the re-ordered matrix defined as

$$a^{(\pi)}_{\pi(i)\pi(j)} = a_{ij}$$

By $G^*_\pi = (V, E^*_\pi)$, we will denote the fill-in graph corresponding to Pfaffian elimination performed on $A^{(\pi)}$. I.e., $\{i, j\} \in E^*_\pi$ if $a^{(\pi)}_{\pi(i)\pi(j)} \neq 0$ or it (possibly) becomes non-zero in the elimination process.

In the following, we will consider only such permutations, or orderings, π that

$$M = \{\{\pi^{-1}(1), \pi^{-1}(2)\}, \{\pi^{-1}(3), \pi^{-1}(4)\}, ..., \{\pi^{-1}(2n-1), \pi^{-1}(2n)\}\}$$

is a perfect matching in G. This will ensure that the pivot elements $a^{(\pi)}_{2k-1,2k}$ are non-zero, and the analysis of the fill-in graph will be somewhat simplified.

Also, we can realize that in the k-th step of elimination, we can skip the operations affecting any elements a_{ij}, where $\min\{i, j\} \leq 2k$. This is because these elements can no longer influence the values of the pivot elements which constitute the Pfaffian in the end.

Lemma 5.3.4 *Vertices $\{i, j\}$ form an edge of the fill-in graph G^*_π, if and only if there exists a path of an odd length $2l+1, l \geq 0$ between the two vertices*

$$(i, x_1, y_1, x_2, y_2, ..., x_l, y_l, j)$$

in the original graph G, so that $\forall p = 1...l$:

$$\{x_p, y_p\} \in M$$

$$\min\{\pi(i), \pi(j)\} > \max\{\pi(x_p), \pi(y_p)\}$$

(In other words, there is an alternating path in G with respect to the perfect matching M where all the inner vertices precede i and j in the ordering π.)

Proof:

First, let us suppose $\{i, j\} \in E^*_\pi, \pi(i) < \pi(j)$. This means that either $a^{(\pi)}_{\pi(i)\pi(j)} \neq 0$ or the element has been modified as a result of a cross operation. The first possibility implies $\{i, j\} \in E$, and thus there exists a path of length 1 between vertices i and j for which the statement of the lemma is trivial.

In the latter case, we proceed by induction on $\pi(i)$. Considering the elimination algorithm, the element $a^{(\pi)}_{\pi(i)\pi(j)}$ must have been affected by an operation, adding a multiple of cross $A^{(\pi)}_{2k}$ to $A^{(\pi)}_{\pi(i)}$, so as to eliminate the element $a^{(\pi)}_{2k-1,\pi(j)}$ $(2k < \pi(i))$. This operation modifies the element $a^{(\pi)}_{\pi(i)\pi(j)}$ only if

- $a^{(\pi)}_{\pi(i),2k-1} \neq 0$
- $a^{(\pi)}_{2k,\pi(j)} \neq 0$

otherwise the operation is trivial (an addition of zero).

By the induction hypothesis, there are alternating paths connecting the pairs of vertices $\{i, \pi^{-1}(2k-1)\}$ and $\{\pi^{-1}(2k), j\}$, where all the inner vertices precede the endpoints in the ordering π. But $2k < \pi(i) < \pi(j)$, and $\{\pi^{-1}(2k-1), \pi^{-1}(2k)\} \in M$, so the union of the paths from i to $\pi^{-1}(2k-1)$ and from $\pi^{-1}(2k)$ to j together with the edge $\{\pi^{-1}(2k-1), \pi^{-1}(2k)\}$ forms an alternating path satisfying the claim of the lemma.

Conversely, let us suppose there exists an alternating path

$$(i, x_1, y_1, ..., x_l, y_l, j)$$

$$\forall p = 1..l : \{x_p, y_p\} \in M, \pi(x_p) < \pi(y_p) < \pi(i) < \pi(j)$$

and let us proceed by induction on l.

If $l = 0$, the path boils down to a single edge $\{i, j\} \in E$. Then, of course, $\{i, j\} \in E^*_\pi$ as well.

If $l > 0$, let us pick the vertex y_p with maximum $\pi(y_p)$ on the path. Because $\{x_p, y_p\} \in M$, we know that $\pi(x_p) = \pi(y_p) - 1 = 2k$ for some $1 \leq k \leq n$. So we have alternating paths connecting the pairs of vertices $\{i, 2k-1\}$ and $\{2k, j\}$ which are shorter than l and satisfying the required condition. By the induction hypothesis, elements $a^{(\pi)}_{\pi(i),2k-1}$ and $a^{(\pi)}_{2k,\pi(j)}$ become non-zero during the elimination process, and it will certainly happen before the algorithm starts eliminating the cross $A^{(\pi)}_{2k-1}$. (After that, elements with an index less than $2k+1$ are never modified again!) But when eliminating the cross $A^{(\pi)}_{2k-1}$, the algorithm is going to carry out an operation modifying the element $a^{(\pi)}_{\pi(i)\pi(j)}$ which will (possibly) make it non-zero. $\square$

So far, our analysis of the algorithm has given us a characterization of the fill-in graph which is an important concept if we want to estimate the complexity of our algorithm. But it is still not clear how the ordering π should be chosen so as to achieve optimal performance. To follow in the wake of [9], we need to introduce another notion.

5.4 Separators and Nested Dissection

In graph algorithms, it is often beneficial to split a graph into two parts which have, loosely speaking, "very little in common". This enables one to apply the "divide and conquer" method which deals with each half of the graph separately (using the same scheme recursively), and then combines the two partial results to produce a final result. In order to make this approach efficient, it is desirable that there are not many edges between the two halves of the graph. Or rather, it is useful to find a *separator* which, if deleted, breaks up the rest of the graph into two separate components. Of course, a good separator is substantially smaller than the entire graph, and it is also necessary than the same property holds for the subgraphs as well. We will define this notion in a rather general way.

Definition 5.4.1 *Let $\mathcal{S}$ be a class of graphs such that*

$$G_1 \in \mathcal{S}, G_2 \subset G_1 \Longrightarrow G_2 \in \mathcal{S}$$

The class $\mathcal{S}$ satisfies an $f(n)$-separator theorem, if for some $\alpha \geq \frac{1}{2}, \beta > 0$:

$$\forall G = (V, E) \in \mathcal{S}; V = A \cup B \cup C$$

where

- $|A|, |B| < \alpha|V|$
- $|C| < \beta f(|V|)$
- $u \in A, v \in B \Rightarrow \{u, v\} \notin E$

The following result can be found in [5].

Theorem 5.4.2 *Graphs of bounded genus satisfy a $\sqrt{n}$-separator theorem.*

The reason why we need separators is that with their help, we will be able to construct an ordering which is very suitable for Pfaffian elimination. The ordering can be found in linear time by a numbering algorithm which was described in [9]. Although it was designed for Gaussian elimination, I will show that the bounds given in [9] hold for Pfaffian elimination as well.

The following procedure assigns numbers from a given range $[a; b]$ to a subset W of the vertices of G. The initial call should be NumberVertices (G, $V(G)$, 1, $|V(G)|$);

Algorithm: (*nested dissection*)

NumberVertices (G, W, a, b)

1. If $W = \emptyset$, RETURN.
2. Find a separator C, dividing $V(G)$ into disjoint parts A, B and C. Denote by G_1 the induced subgraph on vertices $A \cup C$ and by G_2 the induced subgraph on $B \cup C$. Denote $k = |A \cap W|$, $l = |B \cap W|$.
3. NumberVertices (G_1, $A \cap W$, a, $a + k - 1$);
4. NumberVertices (G_2, $B \cap W$, $a + k$, $a + k + l - 1$);
5. Number the vertices in $C \cap W$ arbitrarily from $a + k + l$ to b.

The algorithm bears the name of *nested dissection*, because it cuts the graph recursively into halves, and the numbers assigned to the vertices of each component form a compact block. The vertices of the separator get the largest numbers, so that a path between two vertices from different components A and B (which must cross the separator somewhere) always contains a vertex with a larger number than the endpoints. Considering the ordering as a permutation on the index set of the adjacency matrix, the effect on the elimination scheme is that there will be no edges between A and B in the fill-in graph. To put it another way, the elimination of the submatrix corresponding to A does not affect the submatrix corresponding to B; there will be no fill-in elements $a_{ij}, i \in A, j \in B$. This intuitive reasoning leads to a theorem which was proved in [9] (as a by-product of the analysis of Gaussian elimination).

Theorem 5.4.3 *Let A be an antisymmetric matrix, and let the graph G representing it belong to a class satisfying a $\sqrt{n}$-separator theorem. If π is the ordering found by the numbering algorithm, then the number of edges of the fill-in graph G^*_π is*

$$m = O(n \log n)$$

and the vertex degrees $d(i)$ of G^*_π satisfy

$$\sum_{i=1}^{2n} d(i)^2 = O(n^{\frac{3}{2}})$$

This has important consequences for our elimination algorithm.

Corollary 5.4.4 *For any graph satisfying a $\sqrt{n}$-separator theorem, Pfaffian elimination can be performed in $O(n \log n)$ space and $O(n^{\frac{3}{2}})$ time.*

Proof:

As we know from Lemma 5.3.2, our Pfaffian elimination algorithm requires $O(m)$ space where m is the number of edges of the fill-in graph. From the same lemma, the running time is bounded by $\sum_{k=1}^{n} d(2k-1)d(2k)$, where $d(i)$ is the degree of vertex i in G^*_π.

Due to Theorem 5.4.3,

$$m = O(n \log n)$$
$$\sum_{i=1}^{2n} d(i)^2 = O(n^{\frac{3}{2}})$$

So the bound on the space complexity holds, and because

$$d(2k-1)d(2k) = \frac{d(2k-1)^2 + d(2k)^2 - (d(2k-1) - d(k))^2}{2} \le \frac{d(2k-1)^2 + d(2k)^2}{2}$$

the time complexity is as claimed as well:

$$\sum_{k=1}^{n} d(2k-1)d(2k) \le \frac{1}{2} \sum_{i=1}^{2n} d(i)^2 = O(n^{\frac{3}{2}})$$

□

Thus, for the graphs of interest for us (which are graphs of bounded genus), Pfaffian elimination can be carried out substantially faster than for general graphs (in $O(n^{\frac{3}{2}})$, rather than $O(n^3)$ time).

Chapter 6

Implementation for Toroidal Lattices

One of the main goals of my work was to develop a functional implementation of the algorithm described here, or at least a special case of it. The program, in its most general form, would accept any graph of genus g with integer edge weights, find the relevant orientations, and calculate the appropriate Pfaffians, eventually producing the complete partition function. The reader may notice that this would entail some further issues which I did not address; mainly the algorithmic aspects of proper g-embeddings and relevant orientations. Indeed, my implementation does not incorporate all the features provided by the algorithm as it was described. After consulting literature on the computation efforts in statistical physics so far, it seemed more natural to focus on a rather special case which rests on some physical grounds, and has already been paid attention in the past - *toroidal lattices*.

6.1 Toroidal Lattices

Toroidal lattices seem suitable as models for many physical phenomena in crystallic structures. These are finite orthogonal grids in the plane with additional edges connecting the boundary vertices. In the physical interpretation, the vertices of the lattice represent the particles of a solid crystallic substance, which interact only with their close neighbors. The nature of the interaction is described by the factors assigned to the edges of the lattice. The additional edges, connecting the first and the last vertex in each row and column, represent *periodic boundary conditions*. A special case of this model is the *spin glass* where the edge weights can only be equal to ± 1. The partition function we are going to compute is the starting point for any investigation into the physical properties of the system (see Section 2.2).

Definition 6.1.1 *A toroidal lattice of size $a \times b$ is a graph $G = (V, E)$, where*

$$V = \{[x; y]; 0 \le x < a, 0 \le y < b\}$$
$$E = E_a \cup E_b \cup E_\alpha \cup E_\beta$$
$$E_a = \{\{[x; y], [x + 1; y]\}; 0 \le x < a - 1, 0 \le y < b\}$$
$$E_b = \{\{[x; y], [x; y + 1]\}; 0 \le x < a, 0 \le y < b - 1\}$$
$$E_\alpha = \{\{[a - 1; y], [0; y]\}; 0 \le y < b\}$$
$$E_\beta = \{\{[x; b - 1], [x; 0]\}; 0 \le x < a\}$$

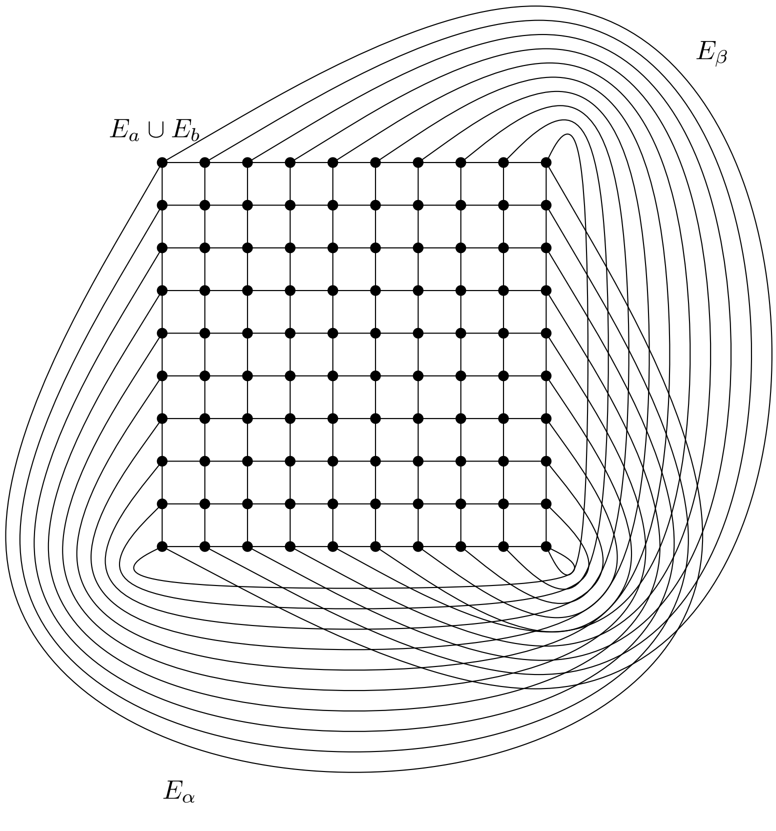

Figure 6.1: Toroidal lattice

Obviously, any toroidal lattice is a 1-graph where $E_a \cup E_b$ can be embedded in the base B_0 (see section 3.4), E_α can be embedded on the bridge B_0^0, and E_β on the bridge B_1^0.

6.2 Graph Transformation and Embedding

Prior to embedding the graph in any surface, we have to carry out the transformations described in sections 3.1 and 3.2. The first one only modifies the edge weights, while the second one turns every vertex into a cluster of vertices. As the degrees of all vertices in a toroidal lattice are equal to 4, the construction given in 3.2 would produce a cluster of 8 vertices for every vertex in G. However, I found a more economical solution for this special case which needs only 6 vertices per cluster.

Definition 6.2.1 *If $G = (V, E)$ is a 4-regular g-graph, and the neighbors of $v \in V$ are denoted by $n_0(v), n_1(v), n_2(v), n_3(v)$ in the clockwise order, we define the 4-splitting of G as a graph $G' = (V', E')$, where*

$$V' = \{v_i; v \in V, i = 0, 1, ..., 5\}$$
$$E' = E^* \cup E^A$$
$$E^* = \{\{v_i, w_j\}; v, w \in V, w = n_i(v), v = n_j(w)\}$$
$$E^A = \bigcup_{v \in V} \{\{v_0, v_1\}, \{v_1, v_4\}, \{v_4, v_0\}, \{v_2, v_3\}, \{v_3, v_5\}, \{v_5, v_2\}, \{v_4, v_5\}\}$$

The edges in E^* are "images" of the edges in E; they receive their respective edge weights. The edges in E^A are "auxiliary" edges which get weights equal to 1.

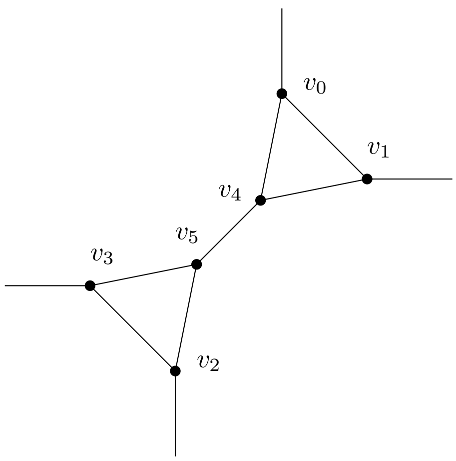

Figure 6.2: Vertex cluster of the modified lattice

Lemma 6.2.2 *The 4-splitting of a 4-regular proper g-graph G is a proper g-graph as well.*

Proof:

Every vertex v can be replaced by a cluster of 6 vertices embedded in a small ball surrounding the original location of v. The auxiliary edges can be embedded inside these balls, and E^* can be embedded in the same way as their images in E. By suitable arrangement of the boundary clusters, we can make the outer circle lie exactly on the boundary. Every vertex is incident with at most one edge from E^*, and the edges in E^A are embedded in the base, so there are never two edges using a bridge incident with the same vertex. Finally,

$$M = \bigcup_{v \in V} \{\{v_0, v_1\}, \{v_2, v_3\}, \{v_4, v_5\}\}$$

is a perfect matching embedded entirely in the base B_0. □

Lemma 6.2.3 *The generating function of eulerian subgraphs of a 4-regular graph G is equal to the generating function of perfect matchings of its 4-splitting G'.*

Proof: Let M denote a perfect matching of G'. Consider the subgraph G_v induced on a cluster of vertices $\{v_0, ..., v_5\}$; the perfect matching M must cover every vertex of G_v exactly once. The edges of M which are inside G_v cover an even number of vertices, so likewise, an even number of vertices of G_v must be covered by edges going out from G_v; these are elements of E^*. For each cluster, there is an even number of incident edges belonging to $M \cap E^*$, therefore the pre-image of $M \cap E^*$ in G forms an eulerian subgraph.

Conversely, if U is an eulerian subgraph in G, the image of U in G' is a matching $K \subset E^*$ which covers an even number of vertices of each cluster G_v. It is easy to see that this matching can be extended in a unique way with some auxiliary edges to a perfect matching M.

So the eulerian subgraphs of G and the perfect matchings of G' are in a one-to-one correspondence, and since the auxiliary edges have weights equal to 1, the corresponding terms in the generating functions are equal. □

6.3 Relevant Orientations

It remains to find the relevant orientations of this modified toroidal lattice. As you may recall, we need to find a Pfaffian orientation of the base graph and then extend the Pfaffian orientation to each of the bridges separately.

If we denote the vertices of $G' = (V', E^A \cup E_a \cup E_b \cup E_\alpha \cup E_\beta)$ by triples of integers:

$$V' = \{[x; y; i]; 0 \leq x < a, 0 \leq y < b, 0 \leq i < 6\}$$

one of the possible Pfaffian orientations of the base graph $(V', E^A \cup E_a \cup E_b)$ is the following:

$$D^A = \bigcup_{0 \leq x < a, 0 \leq y < b} \{([x; y; 0], [x; y; 1]), ([x; y; 1], [x; y; 4]), ([x; y; 4], [x; y; 0]),$$

$$([x; y; 2], [x; y; 3]), ([x; y; 3], [x; y; 5]), ([x; y; 5], [x; y; 2]), ([x; y; 4], [x; y; 5])\}$$

$$D_a = \{([x+1; y; 3], [x; y; 1]); 0 \leq x < a-1, 0 \leq y < b\}$$

$$D_b = \{([x; y; 2], [x; y+1; 0]); 0 \leq x < a, 0 \leq y < b-1\}$$

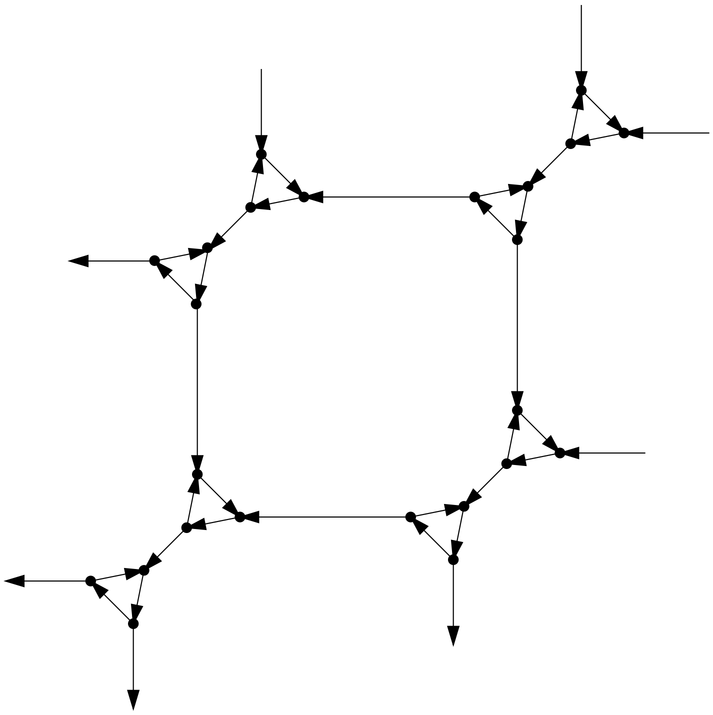

Figure 6.3: Pfaffian orientation of a planar lattice

As you can see, the orientation is devised in such a way that every inner face is clockwise odd, i.e. the orientation is Pfaffian.

If we add the first bridge, in order to obtain a Pfaffian orientation, we can define:

$$D_\alpha = \{([a-1; y; 1], [0; y; 3]); 0 \le y < b\}$$

Similarly, the edges on the second bridge should be oriented like this:

$$D_\beta = \{([x; 0; 0], [x; b-1; 2]); 0 \le x < a\}$$

The four relevant orientations are then obtained by taking the four combinations of reversing the orientation of D_α and/or D_β.

6.4 Separators in Toroidal Lattices

For fast Pfaffian elimination, we need a recursive scheme which describes how to find a small separator for any toroidal lattice. Obviously, if G is an $a \times b$ modified toroidal lattice, the sets of vertices

$$C_\alpha = \{[0; y; i]; 0 \le y < b, 0 \le i < 6\} \cup \{[\lfloor \frac{a}{2} \rfloor; y; i]; 0 \le y < b, 0 \le i < 6\}$$

$$C_\beta = \{[x; 0; i]; 0 \le x < a, 0 \le i < 6\} \cup \{[x; \lfloor \frac{b}{2} \rfloor; i]; 0 \le x < a, 0 \le i < 6\}$$

are separators of sizes $12b$ and $12a$, respectively. The smaller of them is certainly an $O(\sqrt{n})$-separator. The components of $G \setminus C_\alpha$ ($G \setminus C_\beta$) are toroidal lattices as well, so the same separator scheme can be applied recursively. Moreover, after cutting the graph in both dimensions, the remaining components will be planar lattices which have even smaller separators - only $6a$ (or $6b$) vertices in a row of clusters are necessary, instead of the two rows of clusters which form C_α and C_β.

Using the nested dissection algorithm from Section 5.4, we obtain a numbering of the vertices which we will use for Pfaffian elimination. This can be done only once in advance, because the structure of the graph remains the same, but we have to calculate a lot of Pfaffian values.

6.5 Elimination and Modular Machines

The Pfaffian elimination itself is by far the most time-consuming part of the whole program, therefore its implementation was crucial for the algorithm's performance.

My original idea was that the eliminating process for a given graph is always the same, regardless of the initial values in the matrix. It is the structure of non-zero elements that matters, and once it is fixed, the sequence of operations in Pfaffian elimination is determined (except for one exception which I leave aside now). So we can actually generate these operations as a stream of pseudo-code which works with a set of registers, some of which are initially loaded with the matrix values. The total number of registers needed is equal to the maximum size of fill-in, which is bounded by $O(n \log n)$. The instructions of our pseudo-code include the basic operations in a finite field:

- $r_0 := -r_1 \pmod p$
- $r_0 := r_1 + r_2 \pmod p$
- $r_0 := r_1 - r_2 \pmod p$
- $r_0 := r_1 * r_2 \pmod p$
- $r_0 := r_1 / r_2 \pmod p$
- $r_0 := r_1 + r_2 * r_3 \pmod p$
- $r_0 := r_1 - r_2 * r_3 \pmod p$

(The last two instructions could be split into more elementary operations, but they are included to make the code more efficient - these are the most frequent instructions in Pfaffian elimination.)

Having generated the pseudo-code for a given graph, we can evaluate the Pfaffian for any input values by running an interpreter of this code which I call a *modular machine*. Optimally, one could manufacture a processor which would run this code directly!

However, this approach has several drawbacks. First, it occupies a lot more memory than necessary - its space complexity is the same as its time complexity, which is $O(n^{\frac{3}{2}})$ (for 1 Pfaffian value). Of course, we can re-use the memory for different evaluations and finite fields, and it turns out that in reality, we are running sooner out of time than out of space. What is worse from a theoretical point of view, though, this algorithm may not always give the correct answer!

I claimed that the sequence of operations we are to perform is determined uniquely. But if you recall the algorithm of Pfaffian elimination, there is a conditional branching in it which does actually depend on the input values. It is the choice of the pivot element, which must not be zero. Even though we start out with an adjacency matrix where all the pivots are non-zero (as I explained in Section 5.3), it may happen that an element becomes zero as a result of a modular operation. We neglected this possibility in Section 5.3, but it is very important here. Now if this happens, there is little we can do but discard the whole Pfaffian computation as erroneous! (The instruction stream contains a division by zero and we do not have enough information to circumvent it.) Even though we can amend this by repeating the computation with another element of the finite field, I decided to implement also a more robust method which never fails.

The second version always performs the elimination from scratch, just as it was described in Section 5.2. The matrix is represented internally as an array of linked lists, one list for each row of the matrix, and as the matrix is antisymmetric, only the upper triangle needs to be maintained. The elimination proceeds by modifying elements sequentially as they are ordered in the list, so there is no substantial overhead in the access to matrix elements. Still, it may happen than a pivot element is zero, and then we have to find another candidate for the pivot and exchange two crosses of the matrix. This operation is quite costly - all element lists have to be traversed and relinked where necessary, which may take $O(n \log n)$ (the total number of elements) at worst. The total running time, therefore, is $O(n^2 \log n)$. Practical results show, however, that this version performs only slightly worse than the first one (which is $O(n^{\frac{3}{2}})$). The space required in this approach is constant per non-zero element in the matrix, i.e. $O(n \log n)$ in total.

Both versions are implemented in the program, and which of them should be used can be decided before compilation.

6.6 Energy Bounds

Now we know everything we need to calculate a single value of the partition function. As I explained in Chapter 4, we will work in a fixed finite field, and we will evaluate the partition function at sufficiently many distinct points, so that we are able to determine the polynomial uniquely by interpolation. The necessary number of such points depends on the possible degree of the polynomial; or rather, on the maximum number of independent coefficients in the polynomial.

Considering the partition function as a polynomial $P(x)$ in variable $x = e^{-\beta}$, we have

$$P(x) = \sum_{\sigma} x^{H(\sigma)}$$

The range of possible exponents of $P(x)$ is exactly the range of possible energies

$$H(\sigma) = -\sum_{\{i,j\} \in E} J_{ij} \sigma_i \sigma_j$$

Because $\sigma_i = \pm 1$, the energy values are obviously limited by the absolute sum of all edge weights:

$$\forall\sigma : |H(\sigma)| \leq \sum_{\{i,j\}\in E} |J_{ij}|$$

Generally, if the edge weights are bounded by a constant, the number of values we have to calculate is $O(n)$. For the special case of toroidal lattices, though, I found a more precise bound which can be found in linear time.

Lemma 6.6.1 *Let S_{xy} denote the square consisting of vertices $[x;y],[x+1;y],[x+1;y+1],[x;y+1]$ (addition taken modulo size of the lattice). Let $\alpha,\beta,\gamma,\delta$ denote the weights of the four edges forming the square. Define*

$$E_{xy} = \max\{|\alpha+\beta+\gamma+\delta|, |\alpha+\beta-\gamma-\delta|, |\alpha-\beta+\gamma-\delta|, |\alpha-\beta-\gamma+\delta|\}$$

$$E_{max} = \frac{1}{2}\sum_{[x;y]\in V} E_{xy}$$

Then

$$\forall\sigma : |H(\sigma)| \leq E_{max}$$

Proof:

If the spins assigned to the vertices of S_{xy} are denoted by $\sigma_a,\sigma_b,\sigma_c,\sigma_d$ and the edge weights are $\alpha,\beta,\gamma,\delta$ as in the lemma, we define:

$$H_{xy}(\sigma) = -\alpha\sigma_a\sigma_b - \beta\sigma_b\sigma_c - \gamma\sigma_c\sigma_d - \delta\sigma_d\sigma_a$$

The total energy $H(\sigma)$ can be expressed as a sum of these "partial energies" over all squares S_{xy}:

$$H(\sigma) = -\sum_{\{i,j\}\in E} J_{ij}\sigma_i\sigma_j = \frac{1}{2}\sum_{[x;y]\in V} H_{xy}(\sigma)$$

(Every edge appears twice in the last sum, hence the factor of $\frac{1}{2}$.)

The possible values of $H_{xy}(\sigma)$ are determined by the configurations of spins $\sigma_a,\sigma_b,\sigma_c$ and σ_d. There are $2^4 = 16$ possible spin assignments, but they yield only 8 different energy values:

$$E_1^+ = \alpha+\beta+\gamma+\delta$$
$$E_1^- = -\alpha-\beta-\gamma-\delta$$
$$E_2^+ = \alpha+\beta-\gamma-\delta$$
$$E_2^- = -\alpha-\beta+\gamma+\delta$$

$$E_3^+ = \alpha - \beta + \gamma - \delta$$
$$E_3^- = -\alpha + \beta - \gamma + \delta$$
$$E_4^+ = \alpha - \beta - \gamma + \delta$$
$$E_4^- = -\alpha + \beta + \gamma - \delta$$

Four of these are non-negative, while their counterparts are non-positive. As we define

$$E_{xy} = \max\{|E_1^+|, |E_2^+|, |E_3^+|, |E_4^+|\} = \max\{|E_1^-|, |E_2^-|, |E_3^-|, |E_4^-|\}$$

it follows that

$$\forall\sigma : |H_{xy}(\sigma)| \leq E_{xy}$$

Summing up over all squares, we get

$$\forall\sigma : |H(\sigma)| \leq E_{max}$$

□

Note: This is stronger than the trivial bound, because

$$E_{xy} \leq |\alpha| + |\beta| + |\gamma| + |\delta|$$

and therefore

$$E_{max} \leq \sum_{\{i,j\}\in E} |J_{ij}|$$

In the case of *spin glass*, where the edge weights are either $+1$ or -1:

- $E_{xy} = 4$, if an even number of the edge weights in S_{xy} are positive
- $E_{xy} = 2$, if an odd number of the edge weights in S_{xy} are positive

If the edge weights are distributed randomly with equal probabilities, both of these cases are equally probable. Then the expected value of E_{max} is $1.5ab$, whereas the trivial bound would be $2ab$. Experimental results (Section 7.2) show that the average values of MAX-CUT and MIN-CUT on such lattices are approximately $\pm 1.4ab$, so here the bound is not far from optimal (in a randomized sense).

Moreover, in Section 2.2 we expressed the energy of a given state as

$$H(\sigma) = 2w(\sigma) - W$$

where $w(\sigma)$ is the weight of the corresponding cut and W is the sum of all edge weights. Since the edge weights are integers, the cut weights are integers as well. Consequently, the energy levels can differ only by gaps of 2, and they are all either even or odd.

So the polynomial $P(x)$ (multiplied by a suitable power of x) contains only powers of x^2 and as its degree is at most $d = 2E_{max}$, its coeffiecients can be determined uniquely from only $\lfloor \frac{d}{2} \rfloor + 1 = E_{max} + 1$ values.

6.7 Special Optimizations

Finally, I came up with a few optimizations which are applicable only under additional conditions on the input lattice. Without these optimizations, the program accepts any toroidal lattice with integer edge weights. Then it uses the bounds described in the previous section, so that $m = E_{max} + 1$ polynomial values must be calculated for each finite field.

In many cases, though, the polynomial acquires a special form where the number of independent coefficients is smaller. Then we can get away with a smaller number of polynomial values, thus reducing the running time even further. One of these cases is the *spin glass* model where a constraint is imposed that the edge weights can be only equal to ± 1.

Lemma 6.7.1 *Let G be a graph where for any $u \in V$, the star*

$$S_u = \{\{u, v\} \in E; v \in V\}$$

has an even weight $w(S_u)$. Then any cut in G has an even weight.

Proof:
Let $C(V_1, V_2)$ be a cut in G and let us define

$$E_1 = \{\{u, v\} \in E; u, v \in V_1\}$$

Then we can express the weight of the cut as

$$w(C) = \sum_{e \in C} w_e = \sum_{u \in V_1} w(S_u) - 2 \sum_{e \in E_1} w_e$$

If $w(S_u)$ is even for any vertex $u \in V$, then the entire expression must be even. $\square$

Obviously, a toroidal lattice with edge weights ± 1 satisfies the condition of even-weight stars, so its cuts have even weights, too. But then the energy levels must differ not only by gaps of 2, but by gaps of 4! This reduces the number of non-zero coefficients in the polynomial by half.

Another restriction comes into play whenever the lattice has an even size, i.e., it is a bipartite graph.

Lemma 6.7.2 *Let G be a bipartite graph representing an Ising model. Then for any energy level E, the number of states with energy E equals the number of states with energy $-E$.*

Proof:
Let us define a map R on the set of all spin assigments $\sigma : V \to \{+1, -1\}$:

- $R\sigma(v) = \sigma(v)$, if $v \in V_1$
- $R\sigma(v) = -\sigma(v)$, if $v \in V_2$

R is a permutation of all possible spin assignments. If $H(\sigma)$ is the energy of the state σ, then the energy of $R\sigma$ is equal to

$$H(R\sigma) = -\sum_{i\in V_1, j\in V_2} J_{ij}(R\sigma)_i(R\sigma)_j = \sum_{i\in V_1, j\in V_2} J_{ij}\sigma_i\sigma_j = -H(\sigma)$$

Thus R is a permutation which maps the states with energy E to states with energy $-E$ and vice versa. □

If we restrict ourselves to even-sized lattices with edge weights ± 1, the energy levels will be separated by intervals of 4 and the state distribution will be symmetric around zero. Then the partition function can be written as

$$Z(\beta) = \sum_{j=0}^{m} A_j(e^{-4\beta j} + e^{4\beta j})$$

or

$$Z(\beta) = \sum_{j=0}^{m} A_j(e^{2-4\beta j} + e^{2+4\beta j})$$

(depending on whether the sum of all the edge weights is divisible by 4.)

In both cases, the partition function can be expressed as a polynomial with only $m+1$ non-zero coefficients, where $m = \lfloor \frac{E_{max}}{4} \rfloor$. For example, in the first case:

$$Z(\beta) = \sum_{j=0}^{m} B_j(e^{-2\beta} + e^{2\beta})^{2j} = P(y)$$

is a polynomial of degree m in variable $y = (e^{-2\beta} + e^{2\beta})^2$. Its coefficients can be obtained by choosing a set of $m+1$ values for e^{β}, calculating the appropriate $y_0, ..., y_m$ and $P(y_0), ..., P(y_m)$ and performing Lagrangian interpolation. What we get are the coefficients $B_0, ..., B_m$. The relation between these and the coefficients $A_0, ..., A_m$ can be inferred from the two expressions of $Z(\beta)$ above:

$$A_j = \sum_{k=j}^{m} \binom{k}{j} B_k$$

6.8 Interpolation Complexity

When we get the values of our polynomial:

$$P(y_i) = z_i$$

we can proceed by Lagrangian interpolation to obtain its coefficients. The general formula is

$$P(y) = \sum_{i=0}^{n} \frac{\prod_{j \neq i}(y - y_j)}{\prod_{j \neq i}(y_i - y_j)} z_i$$

(where all operations are performed in our finite field).

If we simply followed this formula, we could reconstruct the polynomial in $O(n^3)$ time (because multiplying two polynomials of degrees d_1, d_2 takes $O(d_1 d_2)$ time).

But we can do better, if we split the set of indices $\{0, 1, ..., n\}$ into two parts I_1, I_2 of roughly the same size and rewrite the formula in the following way:

$$P(y) = Q_1(y)P_1(y) + Q_2(y)P_2(y)$$

where

$$Q_1(y) = \prod_{j \in I_2}(y - y_j)$$

$$P_1(y) = \sum_{i \in I_1} \frac{\prod_{j \in I_1, j \neq i}(y - y_j)}{\prod_{j \neq i}(y_i - y_j)} z_i$$

$$Q_2(y) = \prod_{j \in I_1}(y - y_j)$$

$$P_2(y) = \sum_{i \in I_2} \frac{\prod_{j \in I_2, j \neq i}(y - y_j)}{\prod_{j \neq i}(y_i - y_j)} z_i$$

Then $P_1(y)$ and $P_2(y)$ are polynomials of degree $\lceil \frac{n}{2} \rceil$ which can be reconstructed recursively in the same way. $Q_1(y)$ and $Q_2(y)$ are polynomials whose coefficients can be calculated by expanding the respective products in $O(n^2)$ time. So the time we need for interpolation of a polynomial of degree n is:

$$T(n) \leq O(n^2) + 2T(\lceil \frac{n}{2} \rceil)$$

This sort of recurrence implies that

$$T(n) = O(n^2)$$

Even this could be improved, if we plugged in a more sophisticated method to multiply polynomials (using discrete Fourier transform, as described in [14]). But the approach I described here makes the interpolation significantly faster than the rest of the algorithm, so there was little reason to optimize it any further.

6.9 Chinese Composition Complexity

The last part of the algorithm is the composition of the partial results from the finite fields into the complete partition function. Please recall that if

$$x \bmod p_i = a_i, i = 1...k$$

then x can be expressed as a linear combination

$$x = \sum_{i=1}^{k} a_i q_i r_i$$

where

$$r_i = \prod_{j \neq i} p_j$$

$$q_i = {r_i}^{-1} \quad (in\ \mathcal{GF}(p_i))$$

In order to implement this sort of computation, I had to create a simple library supporting long integer arithmetic. I defined a new type called HugeNumber which is internally represented as an array of short integers

$$0 \leq a_0, a_1, ..., a_n < 2^{16}$$

This structure represents the number

$$X = a_0 + a_1 2^{16} + a_2 2^{32} + ... + a_m 2^{16m}$$

The library supports the following operations:

- Create a new HugeNumber equal to $X, 0 \leq X < 2^{16}$ ($O(1)$ time)
- Multiply a HugeNumber of size m by a short integer ($O(m)$ time).
- Add two HugeNumbers of size m ($O(m)$ time).

The coefficients of the partition function are bounded by 2^n, so the length of our HugeNumbers will be $m = O(n)$. Every application of the Chinese Remainder Theorem amounts to $O(n)$ long multiplications by the constants q_i and r_i (which can be prepared in advance) and $O(n)$ long additions (see Section 4.3 for details). The time required to obtain one coefficient of the result is therefore $O(n^2)$. The partition function has $O(n)$ coefficients, so the total time complexity of the composition step is $O(n^3)$.

6.10 Summary

Let us review the algorithm once more, this time with the technical details pertaining to the special case of an $a \times b$ toroidal lattice:

1. Select primes $p_1, p_2, ..., p_k < 2^{16}$ so that

$$\prod_{i=1}^{k} p_i > 2^{ab}$$

   For each of them, repeat the following steps of the algorithm, performing all operations in $\mathcal{GF}(p_i)$.

2. Choose $m+1$ elements $x_0, ..., x_m \in \mathcal{GF}(p_i)$, where m is the maximum possible degree of the partition function as a polynomial $P(y)$ in variable $y = e^{-2\beta}$. (Avoid $x_j = 0$ because the elements must be invertible.) Calculate $y_0 = x_0^2$, $y_1 = x_1^2$,...,$y_m = x_m^2$, make sure these numbers are pairwise distinct, and for each of them, repeat the next step.

3. Construct the four matrices corresponding to the four relevant orientations of the modified toroidal lattice, with hyperbolic functions $\tanh(\beta J_{ij})$ substituted for edge weights. For every occurrence of $e^{-\beta}$, substitute $e^{-\beta} = x_i$ and calculate the four Pfaffians. From these values, calculate the value of $P(y_i)$.

4. Obtain the coefficients of $P(y)$ (modulo p_i) by interpolation in $\mathcal{GF}(p_i)$.

5. Apply the Chinese Remainder Theorem and compose the results from all the finite fields to obtain the complete partition function.

Notes:

- The optimization described in Section 6.7 would affect Step 2., where instead of $y_i = x_i^2$ we would have to substitute $y_i = (x_i^2 + x_i^{-2})^2$ and express the partition function in variable $y = (e^{-2\beta} + e^{2\beta})^2$. Finally, in Step 4. we would have to recalculate the coefficients as described at the end of Section 6.7.

- The number of finite fields is $O(n)$, assuming there is a sufficient number of primes in the range where our hardware is able to perform modular arithmetics in constant time. Due to this constraint, the algorithm is actually unable to work properly for an arbitrarily large input, and any complexity analysis in the sense of asymptotic behavior is meaningless! On the other hand, the limit on the size of lattices we are able to process is well beyond our practical possibilities. The product of all primes below 2^{16} is approximately $2^{2^{16}}$, which means we can process lattices up to 2^{16} vertices. Such a computation would last $10,000$ years on a typical PC. If this should prove insufficient, we could still move to 32-bit arithmetics and the limit of 2^{32} vertices which would be enough to keep all our computing resources busy for more than the lifetime of our universe.

- If the edge weights are bounded by a constant, the number of evaluations in each of the fields is $O(n)$. A single evaluation of the polynomial takes $O(n^{1.5})$, so the computation in each finite field takes $O(n^{2.5})$ time. The total time complexity of Step 3. is $O(n^{3.5})$.

- The interpolation in Step 4. and the final composition in Step 5. take $O(n^3)$ time in total, so they are faster than the Pfaffian evaluation in Step 3. The sequential time complexity of the whole algorithm (under the restrictions mentioned above) is therefore $O(n^{3.5})$.

- Steps 2., 3. and 4. can be parallelized easily. The computation in each of the finite fields can be performed independently, and the communication is cheap - in Step 5., we only have to send the results modulo each of the primes; i.e., data of size $O(n^2)$. With this degree of parallelization ($O(n)$ processors are needed), Steps 2., 3. and 4. take $O(n^{2.5})$ time, whereas Step 5. takes $O(n^3)$. To remove this obstacle, we could parallelize it as well; every processor would produce one of the $O(n)$ coefficients in $O(n^2)$ time. Then the total parallel time complexity would be $O(n^{2.5})$.

Chapter 7

Experiments

In order to test the software and compare the results with some constants known from statistical physics, I carried out a number of experiments with various toroidal lattices.

The lattices were generated as random spin glass instances, with bond values equal to ± 1, with equal and independent probabilities. For each of the instances, I calculated the entire partition function and saved the exact coefficients for further examination.

7.1 Performance Overview

For illustration, I give here a table of the running time required for various parts of the computation (sequentially on a Pentium/200 MHz machine):

lattice size	*1 Pfaffian*	*1 Finite Field*	*Complete partition function*
10×10	0.03 s	5 s	40 s
14×14	0.1 s	30 s	7 min
20×20	0.25 s	150 s	1 hour
30×30	1 s	20 min	20 hours
40×40	3 s	2 hours	8 days
50×50	6 s	6 hours	40 days
100×100	90 s	14 days	24 years

Notes:

- *1 Pfaffian* - this is the basic building block, the Pfaffian elimination of a single matrix; in principle, each of these could be executed in parallel, if $O(n^2)$ processors were available. However, then the communication and composition of results would become the bottleneck of the entire computation.

- *1 Finite Field* - this means the computation of the partition function in a single finite field; it is actually the parallel complexity of the algorithm using $O(n)$ processors, or it can be understood as the complexity of a sequential randomized algorithm to find the MAX-CUT value (see Section 4.2).

- *Complete partition function* - this is the total sequential running time of the computation in all the finite fields plus the Chinese composition step which produces the exact coefficients of the partition function.

7.2 Experimental Results

As the time complexity increases radically with growing lattice size, I decided to distribute the computing resources at my disposal evenly among the individual lattice sizes. The number of experiments I managed to carry out depended on the lattice size as follows:

lattice size	*number of instances*
10×10	4000
12×12	2000
14×14	320
16×16	40
18×18	16
20×20	16

The result for every instance of 2D spin glass consists of a set of coefficients encoding the respective partition function. From these data, we can read the ground state energy, and many other physical quantities as well.

For example, the following table contains the coefficients of the partition function for one of the instances of a 10×10 spin glass.

Note:
The coefficients (*state density*) are given in hexadecimal representation.

Energy	*State density*
± 146	2A98
± 142	D2370
± 138	22E7944
± 134	404AE630
± 130	5ADCB509C
± 126	6807537D18
± 122	63E7B88DD8A
± 118	52763A1F5940
± 114	3B8B889AAF3AA
± 110	261DD9A9D372E2
± 106	15D93D8305E6168
± 102	B4D2319A85AA17A
± 98	54E88EA7166BB7B8
± 94	245D324F6ED395B30
± 90	E41775910FE6D027E
± 86	5214E9981080B1F620
± 82	1B2C711A03D611A886E
± 78	84A212848DFDAF282DE
± 74	255482CAA03D154B191C
± 70	9B3D4ED06B32DC14461E
± 66	254886746B75802D54D1C
± 62	84703FBB00527B8EFA664
± 58	1B2FE679B4DC6FCA5BB28A
± 54	5292DB7E20DC5F7BE75A64
± 50	E7E97093889E9C53BD208A
± 46	25A3BED84A96AD03DE03DFA
± 42	5A5C70CD6ABE3577EB95F50
± 38	C8819EF3D0242B7ECD2BFAA
± 34	19B24B00DFBC92BE606B51C0
± 30	30AE6D99684DF31CB73B62D8
± 26	553075C4A4CA2756E55EA65E
± 24	89ACD50C32927101B13BBEA8
± 18	CD724DE0740F5C839BA22C1E
± 14	11B0ABEB258DA7B518753F57E
± 10	167F702483452588171D9047C
± 6	1A68F3867A15BE3008464799E
± 2	1C9D585B7FF8DA9B6CFA4374C

For each of the instances of a 2D spin glass model, I calculated two fundamental physical quantities:

- *ground state energy* $\overline{e_0}$ - which is the minimum energy level with a non-zero number of states, in other words the MIN-CUT value
- *ground state entropy* $\overline{s_0}$ - which is the logarithm of the number of states with the minimum energy

These values can be obtained easily from the partition function. The following table shows the results, averaged and normalized per vertex of the lattice:

lattice size	$\overline{e_0}$	$\overline{s_0}$
10×10	-1.39 ± 0.05	0.09 ± 0.03
12×12	-1.39 ± 0.04	0.09 ± 0.02
14×14	-1.40 ± 0.03	0.08 ± 0.02
16×16	-1.40 ± 0.03	0.08 ± 0.02
18×18	-1.40 ± 0.02	0.08 ± 0.01
20×20	-1.38 ± 0.02	0.09 ± 0.01

These constants have been previously estimated by Monte Carlo simulations, numerical calculations and high-temperature expansion methods. For example, Kardar and Saul in [11] report

$$\overline{e_0} = -1.404 \pm 0.002$$

$$\overline{s_0} = 0.075 \pm 0.002$$

These values should be valid in the thermodynamic limit, as lattice size tends to infinity. Therefore, we can expect discrepances due to various finite-size effects; nonetheless, the numbers I obtained are in fair agreement with the results of [11].

Chapter 8

Conclusion

The main goal of this thesis was to work out the technical details of the polynomial-time algorithm for the MAX-CUT problem suggested by Dr. Martin Loebl and Dr. Anna Gallucio and to develop a functional implementation. I succeeded in that, and I created a program which produces the complete generating function of cuts, or the partition function of the Ising model on any toroidal lattice. I focused on the *spin glass* model which has been recently one of the most thoroughly studied models in statistical physics. I exploited its special properties for additional optimizations of the algorithm. I used a technique based on *modular arithmetic* which allowed me to split the space and time required by the algorithm into a large number of independent parts. These can be processed either sequentially or in parallel, depending on available hardware.

In comparison with previous approaches, the key advantages of my implementation are:

- it produces the *complete generating function of cuts* in $O(n^{3.5})$ time - this is especially useful for the applications in statistical physics
- the algorithm can be *parallelized* easily - with $O(n)$ processors we get a linear speed-up
- if only the value of the MAX-CUT is needed, we have a *randomized algorithm* running in $O(n^{2.5})$ time to find it
- the implementation is *self-contained* - it does not rely on any libraries for symbolic computation or long arithmetic

As an application of the algorithm, I computed the partition functions for a variety of instances of the 2D spin glass model. I used the results to calculate two fundamental physical quantities: ground state energy and ground state entropy. The values I obtained are in agreement with results obtained by other methods in statistical physics.

Appendix A

Included Software

Included with this thesis is a PC floppy disk containing the implementation of the algorithm which produces the partition function for any toroidal lattice with integer edge weights. This appendix describes briefly how the software can be used. (More information can be found on the floppy disk itself.)

A.1 Compilation

The sources can be compiled under any Unix environment with the `gcc` compiler simply by running `make` in the `source` directory.

Some compile-time options can be specified by modifying the definitions at the beginning of the source file `ising.cpp`:

- `MIN SPLIT SIZE` is the lattice size limit for recursive splitting in the numbering algorithm
- `MATRIX REORDER = 1/0` decides whether the numbering algorithm should be used at all; if not, the lattice vertices are numbered naturally row-by-row
- `GENERATE CODE = 1/0` decides whether the "modular machine" should be used to calculate Pfaffians (faster but consuming more memory)
- `POLY SYMMETRIC = 1/0` decides whether the program can expect the lattice to have an even size and ± 1 edge weights, which allows some special optimizations

Running `make` should produce 3 executable files in the `bin` subdirectory: `genmesh`, `ising` and `chinese`.

A.2 Usage

To make the software parallelizable in a simple way (just by sending text files over a network), I divided it into three separate executables with command-line parameters:

1. `ising field-size input-lattice`

   reads the lattice coefficients from the file `input-lattice` which has the following format:

   `lattice-width lattice-height`

   `horizontal edge weights, one row per line`

   `vertical edge weights, one row per line`

   (Any line beginning with the # character is ignored.)

   The program chooses the nearest prime above or equal to `field-size` and performs all operations modulo this number p. On the standard output, it produces the coefficients of the partition function $\pmod p$.

   The input for `ising` can also be generated with the following tool.

2. `genmesh a b`

   creates an input file for `ising` containing a random spin glass lattice of size $a \times b$

3. `chinese func1 func2 func3 ....`

   accepts several files containing the output of `ising`, each of which contains the partition function modulo some prime number (these results may have been computed on different machines in parallel). The purpose of this program is the implementation of the Chinese Remainder Theorem, which states that if the partial results have been calculated modulo primes $p_1, p_2, ..., p_k$, we can reconstruct the results modulo $\prod_{i=1}^{k} p_i$. The output contains the composite coefficients (in hexadecimal representation).

   It is important to realize that it is beyond the power of this program to decide whether the selection of primes is sufficient to guarantee an *exact* result! The user must either supply the results from a large enough number of finite fields (refer to Section 4.3 for details) or use the partial results, aware of their imprecision.

Bibliography

[1] M.E. Fisher. On the dimer solution of planar Ising models. *Journal of Mathematical Physics 7*, 10, 1966.

[2] A. Galluccio and M. Loebl. A theory of Pfaffian orientations I: Perfect matchings and permanents. *Technical Report 452, IASI-CNR*, 1997.

[3] M. R. Garey and D. S. Johnson. *Computers and Intractability.* W. H. Freeman and Company, New York, 1979.

[4] Alan George. Nested dissection of a regular finite element mesh. *SIAM Journal of Numerical Analysis*, 10, 1973.

[5] Gilbert, Hutchinson, and Tarjan. A separator theorem for graphs of bounded genus. *Journal of Algorithms*, 5, 1984.

[6] M. X. Goemans and D. P. Williamson. Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming. *J. of ACM*, 42:1115–1145, 1995.

[7] M. Kac and J.C. Ward. A combinatorial solution of the two-dimensional Ising model. *Physical Review*, 88, 1952.

[8] P.W. Kasteleyn. Graph theory and crystal physics. In *Graph theory and theoretical physics*, New York, 1967. Academic Press.

[9] Lipton, Rose, and Tarjan. Generalized nested dissection. *SIAM Journal of Numerical Analysis*, 16, 1979.

[10] S. Poljak and F. Rendl. Solving the max-cut problem using eigenvalues. *Discr. Appl. Math.*, 62:249–278, 1995.

[11] Lawrence Saul and Mehran Kardar. The 2D Ising spin glass: exact partition functions in polynomial time. *Nuclear Physics*, B 432, 1994.

[12] C. De Simone, M. Diehl, M. Jünger, P. Mützel, G. Reinelt, and G. Rinaldi. Exact ground states of 2D Ising spin glasses. *Jour. of Stat. Physics*, 84:1363–1371, 1996.

[13] B.L. van der Waerden. Die lange reichweite der regelmassigen atomanordnung in mischkristallen. *Z. Physik*, 118:473, 1941.

[14] C. K. Yap. *Fundamental Problems in Algorithmic Algebra*. Princeton University Press, Princeton, 1993.