NTNU – Trondheim
Norwegian University of
Science and Technology

Improvements of a numerical Algorithm for a certain Class of Colouring Problems

Felix Tadeus Prinz

Physics
Submission date: May 2014
Supervisor: John Ove Fjærestad, IFY

Norwegian University of Science and Technology
Department of Physics

Contents

Abstract

We re-derive an algorithm used to calculate solutions for the edge-colouring and dimer problems for planar graphs. The theoretical background for this includes Pfaffian and then as Grassmann Integrals. We develop a new algorithm which is slightly faster and not restricted to planar graphs, and use this new algorithm to find results for small square, hexagonal and kagome grids. The new algorithm is generalised to a larger class of counting problems.

Vi forklarer en algoritme for finne lsninger til dimer- of kant-fargeleggingsproblemet for flate grafer. Den teoretiske bakgrunnen for algoritmen inkluderer Pfaffianen og Grassmann integraler. Vi lager en ny algoritme som er litt raskere og fungerer utenom flate grafer, og bruker denne for finne resultater for firkant, sekskant of kagome gitre. Den nye algoritmen blir generalisert til en større klasse telle problem.

Acknowledgements

I thank my advisor John Ove Fjrestad for guidance and patience through my 2 year long work with this problem.

Terms and Symbols

N - the number of nodes of a graph
E - the number of edges of a graph
n_i - the i'th node

In this paper, I will refer to the following algorithms:
Creutz Algorithm - the algorithm developed by Creutz [1], used to solve Grassmann integrals by transforming them into a second-quantized expression which can then be numerically evaluated .
Original Algorithm - an algorithm for solving the colouring or dimer problem; obtained by using the Creutz algorithm on the colouring or dimer problem in particular; to do this the dimer problem first has to be expressed as a Grassmann integral.
New Algorithm - an algorithm for solving the colouring or dimer problem which we will develop in this paper; based on the Original Algorithm.

Graph - A graph in this context is a collection of nodes and edges, where each edge connects two nodes.
(un)Directed graph - an directed graph is a graph where the edges have a direction from one of the nodes they connect to the other. Conversely, an undirected graph has no such orientation.
Planar graph - a planar graph can be drawn on a planar, two-dimensional euclidian surface without any edges crossing each other.

Chapter 1

Introduction

Counting problem have roots in both statistical physics, theoretical mathematics and computation theory. Of special interest from the physical side are the Ising model [2] or its generalisation which is the Potts model [3]. The Ising Model can has applications in spin-interaction on lattices; and the problem can be mapped to the dimer problem.

In this thesis, we are looking at numerical algorithms that can generate solutions to two specific counting problems; the dimer and the colouring problem. In particular, we will look at an algorithm used by Fjærestad [4], which we will refer to as the Original algorithm in this thesis. We will give a slightly different derivation of this algorithm, and then we will develop a similar algorithm which is more general in it's application; it can be used for any graph instead of being confined to planar graphs. We will also generalise this new approach to a broader set of problems.

The derivation of the algorithm consists of several steps which serve to bring the dimer and colouring problem into a form which easily lends itself to algorithmic implementation. Note that the dimer problem already has a faster algorithm, which you get halfway through these steps, but this does not work for the colouring problem. We will also give a brief introduction to the theory behind each step.

P. W. Kasteleyn [5] showed in 1963 how the number of dimer distributions on a planar graph can be expressed as the Pfaffian of a certain matrix. This matrix, which is based on the edges of the graph in question, is called the Kasteleyn matrix. For the dimer problem, this is enough, since finding the Pfaffian can be done in polynomial time [6]. In 1980, S. Samuel [7] expressed Pfaffians as Grassmann integrals. Grassmann integrals are based on Grassmann variables which are anti-commuting. Creutz' [1] developed a method for evaluating Grassmann integrals using second quantisation in 1998, and showed how this could be used to efficiently implement the result as an algorithm.

Together, these results can be used to get an algorithm to solve the dimer problem, by transforming it to a Pfaffian which can then be expressed as a Grassmann integral which in turn can be second quantised and then implemented. With slight modifications, which we will keep explaining along the way, we can also get an algorithm for the colouring problem. This algorithm was used by J O Fjærstad in 2012 [4], though he arrived at it slightly differently.

We will look at what the fundamental concepts are that makes the algorithm so powerful by comparing the algorithm to just counting all possibilities. After that we will use this concept to derive a similar but even better algorithm (which, among other things, isn't restricted to planar graphs); and apply the same ideas to the colouring problem to derive an algorithm for that.

We will generalise the algorithm, and prove that it returns the correct result in the general case; which also covers the dimer and colouring versions of the algorithm.

We will give results for our algorithm for the colouring problem for the hexagonal, square and kagome grid. Several problems will be presented as examples for the generalisation of the algorithm.

We will have examples using the graph shown in figure 1.1. Most of the graphs in this thesis were made using GeoGebra, with the exception of figures 9.1 and 9.2. Those two along with the tables were created with OpenOffice.

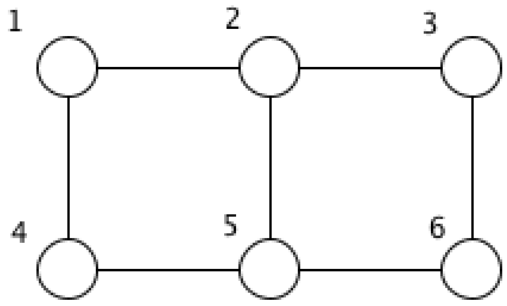

Figure 1.1: A simple undirected graph

In the following, we will often only show the nodes of the graphs, since we need the space of the edge to represent other things (such as dimers). The edges are still there, they will just not be shown. The numbers are for easy reference to the nodes, and will similarly often be omitted in the following.

Chapter 2 will give a general overview over the problems which this thesis concerns. Chapters 3 through 5 will give a brief overview over how these problems are linked to Creutz algorithm

Chapter 2

The problems

2.1 The dimer problem

Consider a general undirected graph, consisting of a number of nodes, some of which are connected by edges. Each edge is a connection between two nodes. We are interested in choosing subsets of all our edges. In order to differentiate between edges we have and haven't chosen, we will call the edges we have chosen dimers. This means that every dimer is an edge, but not every edge is a dimer.

A dimer distribution is a set of dimers where each node is connected to exactly one dimer. An example of a dimer distribution is shown in figure 2.1. There are a total of 3 possible dimer distributions for that figure.

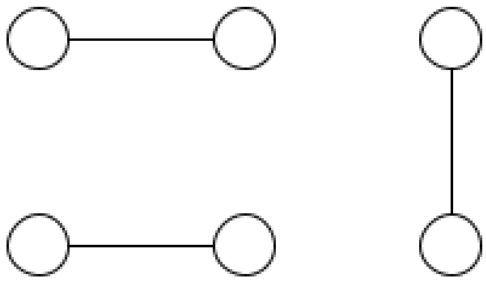

Figure 2.1: A dimer distribution

Any subset of such a distribution, where each node is connected to at most one dimer, we will call a partial dimer distribution. Note that all dimer distributions are also partial dimer distributions, but not all partial dimer distributions are dimer distributions.

There are several ways to represent such distributions. One is simply graphic, see figure 2.2. The same partial distribution can be represented as a set: $\{d_{1,4}, d_{5,6}\}$, where $d_{i,j}$ is a dimer between nodes i and j. Figure 2.3 shows a configuration which is not a partial dimer distribution since two dimers share a node.

The dimer problem is to find the number of dimer distributions for a given graph.

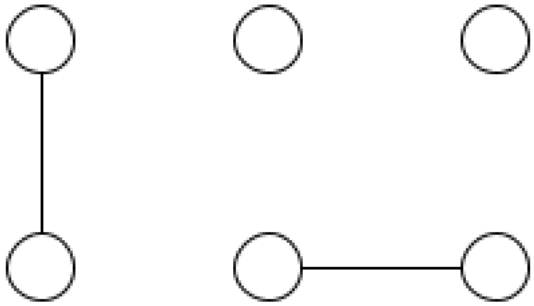

Figure 2.2: A partial distribution with two dimers

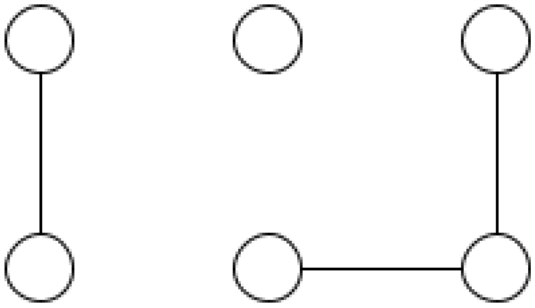

Figure 2.3: A set of edges that isn't a partial dimer distribution

2.2 The colouring problem

This problem is an extension of the dimer problem. Instead of simple dimers, consider "coloured" dimers. A colouring distribution here is one where no two dimers of the same colour share a node, and each edge has exactly one dimer. A partial colouring distribution is one where no two dimers of the same colour share a node, and each edge has at most one dimer. The goal is, once again, to find the number of distributions for the given graph. A colouring distribution for our graph has been given in figure 2.4. There are

For the case of a graph where the number of edges per node is the same for each node and equal to the number of colours, consider all dimers of one colour. Since the number of colours is equal to the number of edges per node, each node will have exactly one dimer of that colour. So each colour forms a dimer distribution. A colouring distribution consists then of a number of non-overlapping dimer distributions, one for each colour. Due to that useful property, we will be working mostly with this type of graph, especially in the first part of this thesis.

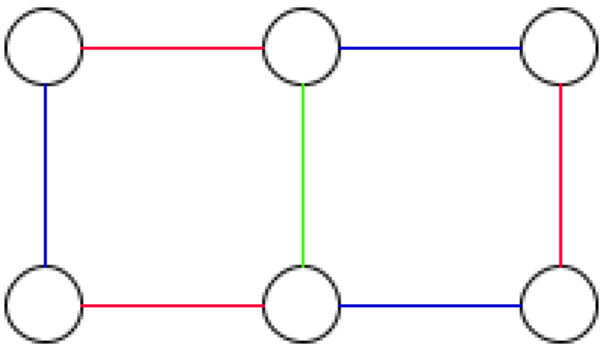

Figure 2.4: (color) A colouring distribution with 3 colours

Chapter 3

Pfaffian solution of the dimer problem

We can show that the number of possible dimer distributions for a planar graph is equal to the Pfaffian of a certain matrix, as shown by Kasteleyn [8]. A planar graph is a graph that can be drawn on a planar 2-dimensional surface without any edges crossing each other. The Pfaffian is equal to ± the square root of the determinant of the same matrix, so we have $(pf(A))^2 = det(A)$.

There are several ways to define the Pfaffian, but the one we will use is the following:

Let A by an antisymmetric nxn matrix with matrix elements $a_{i,j}$. The the Pfaffian of A is defined as

$$pf(A) = \sum_P \delta_P \prod_{l=1}^{n} a_{i_l,j_l} \tag{3.1}$$

Here P is any permutation of i's and j's with exactly one i or j taking each value between 1 and n, and $i_l < j_l$ for all l and $i_m < i_{m+1}$ for all m. The δ_P is a factor of ±1 determined by how many swaps you have to make to get permutation P from the standard permutation. For an even number of swaps, it is +1; for an odd number, it is -1.

Given a graph with n nodes, let K be an antisymmetric nxn matrix with $a_{i,j} = \pm 1$ if there is a connection between i and j; and $a_{i,j} = 0$ else. We will establish which elements are + and - later.

We now want to show that there is a 1-to-1 relation between the terms of the Pfaffian of K and the valid dimer distributions on the graph.

Each dimer $d_{i,j}$ corresponds to a factor $a_{i,j}$ in a term of the Pfaffian. So a dimer configuration $\{d_{i_1,j_1}, d_{i_2,j_2}, ..., d_{i_k,j_k},\}$ corresponds to the term $\delta_P \prod a_{i,j} = \delta_P a_{i_1,j_1} a_{i_2,j_2} ... a_{i_k,j_k}$ and vice versa.

The Pfaffian orders indexes such that $i_l < j_l$ for all l and $i_m < i_m + 1$ for all m. We can order the dimers in a dimer distribution the same way. Since the Pfaffian takes all such permutations, there is a Pfaffian term for any valid dimer configuration. However, there are more terms in the Pfaffian than there are dimer configurations.

Consider now a term of the Pfaffian for which there is no dimer configuration. This term must have at least one factor $a_{i,j}$ for which there is no edge between i and j; and therefore no possible dimer $d_{i,j}$. Since there is no dimer $d_{i,j}$, there cannot be a connection in the graph between i and j, as per the definition of dimers. And due to the definition of the matrix K, this means that $a_{i,j}$ in this case must be 0. Since one factor of the term is 0, the entire term is 0. Thus, all terms in the Pfaffian that do not correspond to a dimer configuration are 0.

A term of the Pfaffian that does correspond to a dimer configuration will have $a_{i,j} = \pm 1$ for all

it's elements. Combined with the δ_P, this means that the entire term has value ± 1. In order for the Pfaffian to be useful to count dimer configuration, we'd like the value to be positive for all terms. Then, we would have #dimer configurations = Pf(K).

3.1 Loops

In order to have all the terms in the pfaffian positive, it is enough to make sure that one term is positive, and that the product of any two terms in positive. The dimer equivalent of this is a superposition of two valid dimer configurations. These superpositions have two elements: double-dimers and loops. The double-dimers occur when both configurations have a dimer at the same place.

The loops have alternating edges from each configuration, and therefore must have an even number of edges. They surround either no nodes, or a set of dimers and loops. Therefore, they must always have an even number of interior nodes. This will become important later.

Consider a factorization of the product into factors corresponding to the double-dimers and loops in the superposition. This is trivial for the matrix elements $a_{i,j}$. In order to see how to divide the $\delta_{P_1}\delta_{P_2}$, we need to look closer at permutations.

3.2 Permutations and Partitions

Each dimer set can be identified by a partition of the nodes into pairs. For a partition, it doesn't matter in which order we have the pair, or in which order we have the elements. In other words, there are many permutations that define the same partition into pairs. (A permutation defines a partition into pairs in the following way: Let the first elements 2i and 2i-1 be a pair for all natural numbers i) We would expect the same to hold for each term of the Pfaffian of K, as the Dimer Problem and the Pfaffian are analogous. We will show now that this is really true.

Any two permutations defining the same partition into pairs can be transformed into each other by repeatedly using two operations: swapping the position of two pairs, and swapping the position of two elements in a pair. For both of these two operations on the pfaffian term we want to show that

$$\delta_P \prod a_{i,j} = \delta_{P'} \prod a_{i',j'} \tag{3.2}$$

where the order of the i's and j's determines the permutation P; and the right side is the same but after the operation. For instance, if we have a small graph of 6 nodes and the matrix elements $a_{1,3}a_{2,5}a_{4,6}$, then P = 132546.

Swapping two pairs does not affect the values of the matrix elements of the pairs, as they commute. If the pairs have indices i_1, j_1 and i_2, j_2, then the permutation will be changed such that i_1 is swapped with i_2 and j_1 with j_2. That is an even number of swaps, so the δ_P doesn't change.

Swapping the elements of a pair does affect the delta element, since it is just one swap, so $\delta'_P = -\delta_P$. The pair also corresponds to an element $a_{i,j}$, and swapping its indices gives $a_{i',j'} = a_{j,i} = -a_{i,j}$ (here we use that the matrix K is antisymmetric). The two factors of -1 cancel each other out, so that the modified and the original term and equal, as expected.

We see now that we can write each term of the pfaffian in any order that we want, as long as it defines the correct partition. Remember from the previous section that we want the product of two terms to be one; specifically we needed to split the expression into one for each loop. We can

easily do this for the $a_{i,j}$ elements, and only need to split the delta.

For each of the terms, order the pairs such that all elements of each loop and dimer are ordered together and the sections representing the loops and double-dimers are in the same order in both terms. The change to the permutation from each loop or dimer is now independent, so we can divide $\delta_{P_1}\delta_{P_2}$ into a product of δ_f; one for each loop or double-dimer. Each δ_f is 1 if you need to an even number of swaps to change the permutation for the loop/double-dimer from term 1 to the permuation for the loop/dimer from term 2. If it takes an odd number of swaps, δ_f is -1.

In the product, each double-dimer corresponds to a factor $\delta_f a_{i,j}^2$, which is always 1, and can thus be disregarded. (δ_f is 1 because the dimer is the same in each configuration.)

For a loop of length r, let $\{1, 2, ...r\}$ be indices for the nodes in the loop, ordered clockwise. We have $a_{i_1,i_2}a_{i_3,i_4}...a_{i_{r-1},i_r}$ from configuration 1 and $a_{i_r,i_1}a_{i_2,i_3}a_{i_4,i_5}...a_{i_{r-2},i_r-1}$ from configuration 2. We get the following permutations:

1 2 3 4 ... r-1 r
r 1 2 3 ... r-2 r-1

To get from one to the other, we need to get r to the left of 1; we do this by swapping r first the r-1 then r-2 and so on until we swap it with 1. That means we need r-1 swaps; which is an odd number since r is even. Thus, for each loop, you δ_f is -1. This means that for each loop, we want the product of $a_{i,j}$ to be -1. Figuring out how to choose the elements of K such that that is case for all loops is difficult, but doing the same on the graph is not; thanks to directed graphs.

3.3 Directed graph

We now direct our graph, such that each connection has a direction. Let $a_{i,j} = 1$ if the direction of the connection is from i to j, and -1 otherwise. For each loop in any superposition, we now want to have an odd number of arrows nodeing clockwise in order to get all the terms of the Pfaffian positive. A such loop is called clockwise odd.

We can reduce this problem to all of the smallest polygons being clockwise odd. If so, we can use induction to prove that any loops with an even number of interior nodes are clockwise odd too. All the loops in superpositions of dimer configurations do have an even number of interior nodes, as explained in the previous section.

The induction is done as follows:

Let l_1 be a loop with x interior nodes and l_2 a loop with y interior nodes, such that the two loops are adjacent at only one continuous length and do not share any polygons. Assume that loops with an even number of interior nodes have are clockwise odd, and loops with an odd number interior nodes clockwise even. This can be expressed more easily in mod 2:

#clockwise arrows + 1 $\equiv$ #interior nodes (mod 2)

Let l_3 be the loop surrounding all the polygons inside both loop 1 and loop 2 and no others. Loops 1 and 2 touch each other at a number of edges $s > 0$. Then loop 3 will consist of all edges from loops 1 and 2 except for these s edges. The s edges do have s-1 nodes in between them which are not interior to either 1 or 2, but are interior to 3. So loop three has (x+y+(s-1)) interior nodes.

Looking at the clockwise directed arrows in loop 3, we see that the loop get all the arrows from loops 1 and 2 except those located at the contacting part. For each of the s edges between loops 1 and 2, exactly one of the loops has the arrow of that side clockwise.

Let us now check whether the mod 2 equation holds for loop 3:

#clockwise arrows + 1 ≡ #interior nodes (mod 2)
#clockwise arrows + 1 ≡ (x - 1 + y -1 + s) + 1 ≡ x + y + s + 1 (mod 2)
#interior nodes ≡ x + y + (s-1) ≡ x + y + s + 1 (mod 2)

The only thing left to do is to show that any planar graph can be directed such that any minimal polygons are clockwise odd. This is easily done as follows: start with one minimal polygon; choose arrows such that it is clockwise odd. Now add polygons one by one such that all the chose polygons at any time form one area without any holes. For each polygon you add, choose it's non-directed sides such that it becomes clockwise odd. By choosing polygons in this way, any new polygon will have at least one side that has not yet had a direction assigned to it; which is enough to ensure that the polygon is clockwise odd.

3.4 A note on Non-planar Graphs

While the approach detailed here only works for planar graphs, it is possible to extend beyond that. For instance, for any graph that can be embedded on a toroid, the number of dimer distributions can be expressed as a combination of 4 pfaffians, as shown by Kasteleyn [8].

3.5 Pfaffians for the Colouring Problem

Recall that we can represent the colouring problem as a set of non-overlapping dimer problems. Each of these dimer problems can be represented by a Pfaffian. However, we don't have a good way to formalise the non-overlapping criteria at this point. It is possible, but very clumsy. We will get back to that in the next chapter, where Grassmann variables offer a more elegant solution. For now, note that the pfaffian consists of one term for each dimer distribution. Each of these distributions is in turn represented as the product of all its dimers. Thus it would make sense to represent a colouring distribution as a product of all its dimers with an added variable to denote colour. So we can represent a coloured dimer distribution as:

$$\prod_{l=1}^{n} K^{c}_{i_l,j_l} \tag{3.3}$$

and a colouring distribution will have the form:

$$\prod_{c} prod_{l=1}^{n} K^{c}_{i_l,j_l} \tag{3.4}$$

The product of c dimer distributions will contain terms of the form above, including all the colouring distributions for the graph. However, they will also contain several terms that aren't colouring distributions, because they violate the non-overlapping condition. Our limitation that no two different colours should occupy the same dimer. So, $K^{c_1}_{i,j}$ and $K^{c_2}_{i,j}$ should not both be present in the product for any i, j and pair of distinct c_1 and c_2. This will be easier to formalise in the next chapter.

Chapter 4

Grassmann Variables and Integrals

In this section we will be introducing Grassmann variables. See also "Principles of Quantum Mechanics" by Shankar [9], where there is a good introduction to Grassmann variables and integrals with links to quantum states and operators.

Grassmann variables are anti-commuting, so that we have

$$\psi_1\psi_2 = -\psi_2\psi_1 \tag{4.1}$$

for all ψ_1 and ψ_2. This implies among other things that

$$\psi\psi = -\psi\psi \implies \psi^2 = -\psi^2 \implies \psi^2 = 0 \tag{4.2}$$

for all ψ. Now consider a function $f(\psi)$. The Taylor series of such a function will be linear in ψ, since all terms of order 2 or higher in ψ are equal to 0 since $\psi^2 = 0$. As such all functions that have will be equal to a finite polynomial. Of particular interest to us is the case of e^{ψ}:

$$e^{\psi} = 1 + \psi \tag{4.3}$$

and in general we have

$$f(\psi) = a\psi + b \tag{4.4}$$

where a and b can be functions both of real numbers and other grassmann variables; but must be constant with regard to ψ. Next we will define integration over grassmann variables. Since we have already established that all functions are linear at most, and we want integration to be be a linear functional (like normal integration), we only need to define two base cases in order to define all integration over grassmann variables:

$$\begin{aligned} \int d\psi\psi &= 1 \\ \int d\psi 1 &= 0 \end{aligned} \tag{4.5}$$

((this should possibly be numbered as two separate equations))

This gives us for an arbitrary function:

$$\int d\psi a\psi + b = a \tag{4.6}$$

When integrating over several variables, it is important to keep track of the order, as they are anti-symmetric variables, and changing the order can change the sign of your final expression. In

particular, the following two expression give different results:

$$\begin{aligned} \int d\psi_1 d\psi_2 \psi_1 \psi_2 &= -1 \\ \int d\psi_2 d\psi_1 \psi_1 \psi_2 &= 1 \end{aligned} \tag{4.7}$$

Since we will need to work with many variables, it is useful to define how the product symbol should work for these variables, which basically just means that we need to define the order in which the terms should be multiplied.

$$\prod_{i=1}^{n} d\psi_i \equiv d\psi_n d\psi_{n-1} \dots d\psi_2 d\psi_1 \tag{4.8}$$

Given these basic definitions, we will now prove that the pfaffian can be expressed in terms of a grassmann integral as follows:

$$pf(K) = \int \left(\prod_{i=1}^{n} d\psi_i \right) \exp\left(\frac{1}{2} \sum_{j,k=1}^{n} \psi_j K_{j,k} \psi_k \right) \tag{4.9}$$

In order to do this, we will use induction and the recursive definition of the Pfaffian:

$$pf(K) = \sum_{i=2}^{n} (-1)^i k_{1,i} pf(K_{\setminus\{1,i\}}) \tag{4.10}$$

Here, $k_{1,i}$ and is the element of K at position (1,i) and $K_{\setminus\{1,i\}}$ is the matrix K with both rows and columns 1 and i removed. Given this recursive definition, we only need to show that the grassmann integral follows the same recursive pattern, as well as establishing a base case where the formula holds. The base case is easily found as the 2x2 matrix. Since we require the matrix to the anti-symmetric to get a non-trivial result, we only need to consider anti-symmetric matrices:

$$pf \begin{pmatrix} 0 & a \\ -a & 0 \end{pmatrix} = a \tag{4.11}$$

For the integral corresponding to the matrix, we get

$$\begin{aligned} &\int d\psi_2 d\psi_1 \exp\left(\frac{1}{2} \left(\psi_1 a \psi_2 + \psi_2 (-a) \psi_1 \right) \right) \\ &= \int d\psi_2 d\psi_1 \left(1 + \frac{1}{2} \psi_1 a \psi_2 \right) \left(1 - \frac{1}{2} \psi_2 a \psi_1 \right) \end{aligned} \tag{4.12}$$

because, as you may recall, $e^{\psi} = 1 + \psi$. This leaves us with four terms. The rules for Grassmann variables tell us that the quadratic term is 0, while the rules for Grassmann integration tell us that the constant term is 0. The remaining two terms are:

$$
\begin{aligned}
&\int d\psi_2 d\psi_1 \left(\frac{1}{2}\psi_1 a\psi_2 - \frac{1}{2}\psi_2 a\psi_1\right) \\
&= \int d\psi_2 d\psi_1 \left(\frac{1}{2}a(\psi_1\psi_2 - \psi_2\psi_1)\right) \\
&= \int d\psi_2 d\psi_1 \left(\frac{1}{2}a(\psi_1\psi_2 + \psi_1\psi_2)\right) \\
&= \int d\psi_2 d\psi_1 a\psi_1\psi_2 \\
&= a
\end{aligned}
\tag{4.13}
$$

As required, both give the same result, a. Next we will prove the integration step, staring with our Grassmann expression:

$$
\begin{aligned}
&\int \left(\prod_{i=1}^{n} d\psi_i\right) \exp\left(\frac{1}{2}\sum_{j,k=1}^{n} \psi_j K_{j,k}\psi_k\right) \\
&= \int \left(d\psi_1 \prod_{i=2}^{n} d\psi_i\right) \left(1 + \sum_{j=2}^{n} \psi_1 K_{1,j}\psi_j\right) \exp\left(\frac{1}{2}\sum_{k,l=2}^{n} \psi_k K_{k,l}\psi_l\right) \\
&= \int \left(\prod_{i=2}^{n} d\psi_i\right) (-1) \left(\sum_{j=2}^{n} K_{1,j}\psi_j\right) \exp\left(\frac{1}{2}\sum_{k,l=2}^{n} \psi_k K_{k,l}\psi_l\right)
\end{aligned}
\tag{4.14}
$$

Note that each element of the sum can contain no more than one ψ_j, since $\psi_j^2 = 0$. This means that all elements in the seconds sum that contain ψ_j give no contribution to the final results, and can thus be removed. Integrating out ψ_j gives us a factor $(-1)^{j-1}$, since we need to move $d\psi_j$ next to ψ_j, which takes j-1 moves, and each moves gives a factor of -1. (the previous factor of -1 came from exchanging ψ_1 and ψ_j in order to have ψ_1 and $d\psi_1$ next to each other). Thus we now have:

$$
\sum_{j=2}^{n} (-1)^j K_{1,j} \int \left(\prod_{i=2}^{n} d\psi_i\right) \exp\left(\frac{1}{2}\sum_{k,l=\setminus\{1,j\}}^{n} \psi_k K_{k,l}\psi_l\right) \tag{4.15}
$$

Here we have the same recursion as for the Pfaffian, proving that the grassmann expression and the Pfaffian are equal.

4.1 Colouring Problem

We will now use develop a Grassmann Integral formulation for the colouring problem, see also Fjærestad [4].

From section 3.5 we have that the number of colouring distributions is equal to a product of Pfaffians with a restriction. Let us first look at the product and then come back to the restriction.

We have:

$$\begin{aligned}\prod_c pf(K^c) &= \int \prod_c \left(\left(\prod_{i=1}^n d\psi_i^c \right) \exp\left(\frac{1}{2} \sum_{j,k=1}^n \psi_j^c K_{j,k} \psi_k^c \right) \right) \\ &= \int \left(\prod_{i=1}^n \prod_c d\psi_i^c \right) \prod_c \left(\exp\left(\frac{1}{2} \sum_{j,k=1}^n \psi_j^c K_{j,k} \psi_k^c \right) \right) \\ &= \int \left(\prod_{i=1}^n \prod_c d\psi_i^c \right) \exp\left(\frac{1}{2} \sum_c \sum_{j,k=1}^n \psi_j^c K_{j,k} \psi_k^c \right)\end{aligned} \tag{4.16}$$

Now, our restriction was that we cannot have $K_{i,j}^{c_1}$ and $K_{i,j}^{c_2}$ together present in any of products. This is easily enforced by making use of the properties of Grassmann variables, in particular that $\psi^2 = 0$. By adding a Grassmann variable for each $k_{i,j}$, we ensure that is can be used only by one of the colours for each distribution; if it were used by two or more we would get at least a double instance of that Grassmann variable, and hence get 0.

However, as we are dealing with Grassmann variables, this leads to sign problems. We can avoid that by observing that a pair of Grassmann variables always commutes, and thus doesn't give us any additional sign trouble. Thus we now have the following formula for the number of colouring distributions:

$$\int \left(\prod_{i=1}^n \prod_c d\psi_i^c \right) \left(\prod_{l,m=1}^n d\bar{\xi}_{l,m} d\xi_{l,m} \right) \exp\left(\frac{1}{2} \sum_c \sum_{j,k=1}^n \psi_j^c K_{j,k} \xi_{j,k} \bar{\xi}_{j,k} \psi_k^c \right) \tag{4.17}$$

Chapter 5

Creutz' Algorithm

5.1 Second Quantisation

Creutz algorithm [1] is intended to numerically solve Grassman integrals of the following form:

$$Z = \int \left(\prod_i^n d\psi_i \right) exp(S(\psi)) \tag{5.1}$$

which is exactly the expression we got for our grassmann integral.

In order to get Creutz algorithm, we need to introduce fermonic second-quantisation instead of the grassmann variables. We exchange a variable $|\psi_i\rangle$ for creation and annihilation operators a_i and $a_i^\dagger$. These act on base states $|n_1, n_2, ..., n_N\rangle$. A general state $|A\rangle$ is a superposition of these base states:

$$|A\rangle = \sum_{n_1,n_2,...,n_N} c_{n_1,n_2,...,n_N} |n_1, n_2, ..., n_N\rangle \tag{5.2}$$

In particular, we denote the empty state $|E\rangle$ and the full state $|F\rangle$, defined such that $a_i|E\rangle = 0$ for all i, and $|F\rangle = \prod a_i^\dagger |E\rangle$ Rewriting the integral using second quantisation, it can be shown that:

$$Z = \langle E|exp(S(a))|F\rangle \tag{5.3}$$

Now we gather all terms of S containing a_i and call that S_i. The remaining terms $\tilde{S}_i$, and by definition does not contain any instance of a_i. We have $S = S_i + \tilde{S}_i$, and thus

$$\begin{aligned} Z &= \langle E| \exp(\tilde{S}_i(a) + S_i(a))|F\rangle \\ Z &= \langle E| \exp(\tilde{S}_i(a))(1 + S_i(a))|F\rangle \end{aligned} \tag{5.4}$$

Since $\tilde{S}_i$ contains no a_i, we have that $\langle E|exp(\tilde{S}_i)$ is empty at position i. Applying the occupation number operator $\hat{n}_i = a_i^\dagger a_i$ will therefore give 0, and we can insert a term as follows:

$$\begin{aligned} Z &= \langle E| \exp(\tilde{S}_i(a))(1)(1 + S_i(a))|F\rangle \\ Z &= \langle E| \exp(\tilde{S}_i(a))(1 + 0)(1 + S_i(a))|F\rangle \\ Z &= \langle E| \exp(\tilde{S}_i(a))(1 + \hat{n}_i)(1 + S_i(a))|F\rangle \end{aligned} \tag{5.5}$$

This new term, $1+\hat{n}_i$, is called the projection operator, and acting with it on any state will project that state to be empty at position i. This means that $S_i(1+\hat{n}_i)$ is always 0, and we can insert an

extra term S_i before the projection operator.

$$\begin{aligned} Z &= \langle E| \exp(\tilde{S}_i(a))(1)(1+\hat{n}_i)(1+S_i(a))|F\rangle \\ Z &= \langle E| \exp(\tilde{S}_i(a))(1+S_i(a))(1+\hat{n}_i)(1+S_i(a))|F\rangle \\ Z &= \langle E| \exp(\tilde{S}_i(a)) \exp(S_i(a))(1+\hat{n}_i)(1+S_i(a))|F\rangle \\ Z &= \langle E| \exp(\tilde{S}_i(a)+S_i(a))(1+\hat{n}_i)(1+S_i(a))|F\rangle \\ Z &= \langle E| \exp(S(a))(1+\hat{n}_i)(1+S_i(a))|F\rangle \end{aligned} \tag{5.6}$$

For simplicity, we now define an operator O_i.

$$O_i = (1-n_i)(1+S_i) \tag{5.7}$$

By induction over i, we have

$$Z = \langle E|exp(S(a)) \prod_i O_i|F\rangle \tag{5.8}$$

Since the product over O_i contains the projection operator for all i, and no other instances of the creation operator, we have

$$\prod_i O_i|F\rangle = |E\rangle \tag{5.9}$$

and thus we can neglect the exp(S(a)) term:

$$Z = \langle E| \prod_i O_i|F\rangle \tag{5.10}$$

For the dimer problem, from the Grassmann integral we have:

$$\begin{aligned} S &= \frac{1}{2}\sum_{i,j} K_{ij}a_i a_j \\ S_i &= \frac{1}{2}\sum_{j\in nn(i)} K_{ij}a_i a_j \end{aligned} \tag{5.11}$$

where nn(i) is the set of nodes connected by an edge to node i (its neighbors). We do not need to consider any other values of j because $K_{ij} = 0$ for any i and j that aren't neighbours.

For the colouring problem, again using the Grassmann integrals we worked out in chapter 4, we have (with second-quantisation giving us $\psi_i^c \to a_i^c$ and $\xi_{i,j} \to b_{i,j}$):

$$S_i = \frac{1}{2}\sum_{j\in n(i)}\sum_c b_{i,j}\bar{b}_{i,j}K_{ij}a_i^c a_j^c \tag{5.12}$$

For the b operators to work, we need to change our base state by adding m variables for the b operators to work on. However, we won't need all pairs of i and j, just the ones the lie within n(i) for all i's. We thus have as our base state:

$$|n_1^{c_1}, n_1^{c_2}, ..., n_1^{c_c}, n_2^{c_1}, ..., n_N^{c_1}, ..., n_N^{c_c}, ...m_{i_1,j_1}, \overline{m}_{i_1,j_1}, m_{i_2,j_2}, \overline{m}_{i_2,j_2}, ..., m_{i_E,j_E}\overline{m}_{i_E,j_E}\rangle \tag{5.13}$$

5.2 Computer Implementation

When implementing the algorithm above, we have to make some concession to the way computers actually work. Most importantly, they can't represent a superposition directly, and must instead represent it as a list of all its component base states. These base states consist of N variables each for the dimer problem (which correspond to the n_i of the base state) and N*c + 2E variables for the colouring problem (c variables for each of the N nodes, corresponding to the n variables and 2E variables, corresponding to the m and $\overline{m}$ variables).

These variables can take the value of 0 or 1, so each variable can be represented by booleans fairly naturally. However, there is an even better representation. Computers store integers in binary anyway, so each digit of the integer can be used as a 0/1 variable. In that way we only need to store one integer instead of many booleans. This is especially useful as there are binary operations for integers which allow you to change individual digits of a number fairly easily. We will also need a second integer for each base state, which represents the prefactor.

To sum up, we represent a generic state $|\psi\rangle$ by a number of integers, two for each base state.

We start with $|\psi\rangle = |F\rangle$, which is the state consisting solely of the full base state. We then sequentially apply the operators O_i (see equation 5.7) to each of the base states that our current $|\psi\rangle$ consists of. We translate these to binary operations on our integers. 1 simply leaves the base state as it is; n_i changes just the prefactor; multiplying it with the value at digit i or the base state. S_i is of course a bit more complicated, but it can be broken down into three different operations; K_{ij}, a_i and b_{ij}. K_{ij} is just a number, so is just changes the prefactor. a_i changes the digit at place i from 1 to 0 if it was 1, and changes the prefactor to 0 if it was 0. Similarly, b_{ij} affects the digit as the position assigned to it.

After we have applied all our operators O_i, we take the state we end up with, and find the prefactor of the base state that is equivalent to $|E\rangle$. So, $\langle E|\prod_i O_i|F\rangle$ is a projection of $\prod_i O_i|F\rangle$ onto $|E\rangle$. Thus, all other base states will give 0 contribution to the result, and the prefactor in front of $|E\rangle$ is the answer we want.

5.3 Understanding the algorithm

There is a lot going on here, and it isn't obvious why a series of seemingly random steps gives us a decent algorithm for calculating dimer distributions. We are not interested in understanding why these steps work. Instead, we will look at what makes this algorithm so much more powerful than just generating all possible solutions and counting them. In chapter 6 we will use those insights to develop a better algorithm for the specific problem we are looking at, which should also be easier to comprehend. Note that the algorithm we develop will not be able to be used in every way that the original Creutz algorithm could be used; it will be intended specifically for the Dimer and Colouring Problem or related problems.

We will now consider how a state is represented in the original algorithm. We don't care about the specific implementation. Instead, we are looking at general concepts we can find here, and there are three which are interesting: firstly, this is a superposition. That means that we can handle a very large amount of states (all the substates of the superposition) simultaneously; at least on paper. When we get to implementing the algorithms, however, we have to represent and process all of those individually. So this does help to make the algorithm easier to understand, but not to make it faster.

The second important concept here is that the algorithm builds all its states in parallel, step by

step; instead of building them one at a time and then counting when you have built all of them. This is a quite potent idea, since it makes it a lot easier to ensure that you have all states.

The last, and most important, thing to look at is how each of the substates of the superpositions are defined, and compare this to a "natural" representation of a dimer distribution. A natural representation of dimers could for instance be a list of all the dimers present in the graph; or a list of all edges, with different values for edges that have dimers and edges that do not. These representations are **complete**. One such representation gives us full information of the system.

On the other hand, the Creutz algorithm uses nodes as the basis for its representation for the system. This is an **incomplete** representation. Several dimer distributions can be represented in with a single state. This is a powerful idea, and is exactly what makes the Creutz algorithm so powerful. It means we do not need to deal with each dimer distribution individually, instead only concerning us with states that can represent a lot of dimer distributions at the same time.

Chapter 6

The New Algorithm

In this chapter we will develop the New Algorithm based on the insights from section 5.3. In particular, we want to develop states that can handle as many partial distributions at the same time as possible, while being as simple as possible.

6.1 New states

Let us first consider an algorithm which uses complete representations for our dimer distributions. To do that, we use partial dimer distributions as our states.

So, our pseudo algorithm is now:

1. start with a system containing only the empty partial dimer distribution
2. for each node, take each partial dimer distribution in our system, and do the following:
a) if the partial dimer distribution has a dimer at that node, keep it as it is
b) if the partial dimer distribution does not have a dimer at that node, remove it from our system. Add a new partial dimer distribution for every possible dimer connected to that node.
3. After we have gone through all the nodes, count all of the partial dimer distribution which use all nodes (and thus are dimer distributions)

We want to expand on this by introducing states where one state can represent several partial dimer distributions. These states have several requirements:

1 - We need to be able to associate each partial distribution with exactly one state, so as to avoid counting a dimer distribution twice.
2 - We need a way to keep track of how many partial distributions are represented by a state.
3 - We need the generic distribution to have the same restrictions that any of the basic states it represents have.

The last point is the critical one. It is needed due to the way we set up our algorithm - we build all distributions in parallel one step at a time (analogous to the Original algorithm). Each step adds another dimer to each partial dimer distribution; or leaves the distribution as it is; see step 2 in our pseudo algorithm above. The restriction here is that the new dimer cannot touch any existing dimer. If we instead work with states, then they need to have exactly the same restrictions as any partial dimer distribution it contains. If the state has fewer restrictions; then we will get invalid distributions; and if the state has too many restrictions, then some distributions will potentially not be counted, and our final count will be wrong.

However, this also lends itself beautifully as a definition of a state. We define a state as the smallest

set of information that contains the exact restrictions of a partial dimer distribution. Incidentally, this gives us a state very similar to the states used in the Creutz algorithm: For each node, we have a 0/1 variable indicating whether it already has a dimer. We name these variables n_i, where i indicates the node. Any new dimer must be between two nodes that do not have a dimer; as it would otherwise conflict with previous dimers. This means for our state that a new dimer must be between two points whose variables indicate that there is no dimer there; and adding a dimer will change the two nodes to reflect that they are now occupied.

In addition, we need to add a counter, since we currently have no way of representing how many elemental states are represented by our state. This is done by simply defining our system as a linear combination of our states. To use a similar notation as the Creutz algorithm:

$$\psi = \sum x|n_1, n_2, n_3, ..., n_N\rangle$$

$n_i \in \{0, 1\}$ here represents the whether node i has a dimer or not. Also define $|E\rangle$ as the empty state (which represents the empty partial dimer distribution, with no dimers) and $|F\rangle$ as the full state, the state where each node has exactly one dimer. (which represents all dimer distributions)

The only difference to the states used in the Original algorithm is actually that we take x to count a number of states, so it is always a positive number. By contrast, Creutz' algorithm has states that can take negative values for x, because the creation and annihilation operators mean you have to keep track of the sign you your states.

At this point, it is worth noting that it does not matter what we assign as 0's or as 1's. One of the two must show that a node is occupied by a dimer, but we can choose which one. Nor does it matter whether we build our dimer distributions by starting with an empty state and adding dimers or starting with a full state and removing dimers. The end result is exactly the same.

Because it seems easier to think about building a dimer distribution by adding dimers, we will start with an empty state, as opposed to the Original Algorithm. But, again, this is exactly the same thing except for a cognitive difference. For programming convenience, we will assign a 0 to states that contain a dimer.

6.2 Example

Consider an basic state (which, as stated before, is a partial dimer distribution in this case), see figure 6.1. In set notation, this state would be represented as $\{(1, 2), (4, 5)\}$

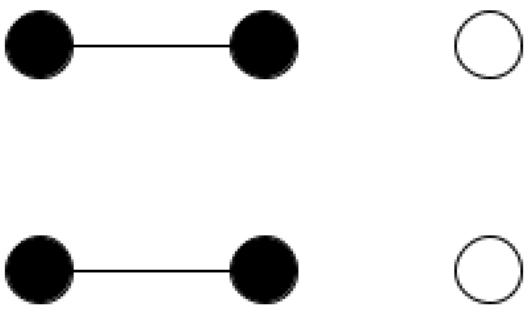

Figure 6.1: A partial dimer distribution, with occupied nodes marked

Note that we have marked which nodes these dimers use. Those nodes are what we use to define the state, which is shown in figure 6.2. With our notion, it is represented as $|1, 1, 0, 1, 1, 0\rangle$. In this case there are 2 partial dimer distributions that match this state; 6.1 and 6.1.

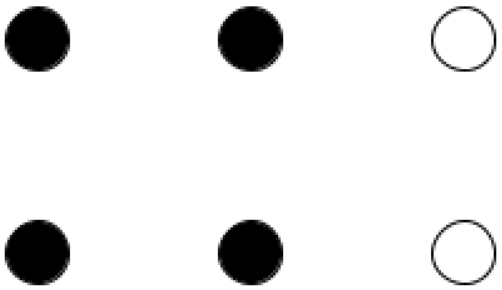

Figure 6.2: A state

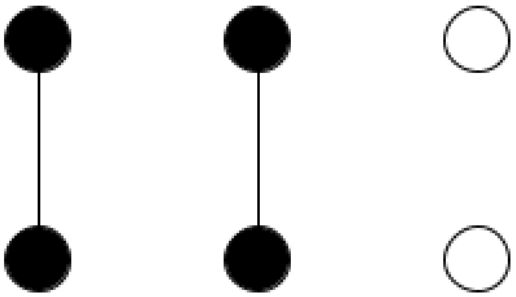

Figure 6.3: A partial dimer distribution, with occupied nodes marked

6.3 New algorithm for the Dimer Problem

With our new understanding of the states, let us consider the rest of the algorithm. It will be quite similar to our pseudo algorithm in section 6.1; though we use introduce formal notation for our states.

1. start with the empty state, $|E\rangle$. 2. for each node, take each state in our system, and apply the following operator to that state:

$$O|n_1, ..., n_N\rangle = \delta_{n_i,0}|n_1, ..., n_N\rangle + \delta_{n_i,1} \sum_{j \in n(i)} \delta_{n_j,1}|\{n\}_{\setminus\{i,j\}}\rangle$$

where n(i) is the set of all neighbours of the node i (all nodes with are connected by an edge to node i) 3. After we have gone through all the nodes, look at the counter in front of the full state, $|F\rangle$. That is the number of dimer distributions.

Note that the operator can be split into two parts, which as directly analogous to our original step 2:
a) if the state has a 0 at that node, keep it as it is ($\delta_{n_i,0}|n_1, ..., n_N\rangle$)

b) if the state has a 1 at that node, remove it from our system. Add a new state for every possible dimer connected to that node ($\delta_{n_i,1} \sum_{j \in n(i)} \delta_{n_j,1} |\{n\}_{\setminus\{i,j\}}\rangle$)

Our algorithm is quite close to Original algorithm. The important difference here is that we don't need the factor $-A_{ij} r_{ij,\{n\}}$ in the operator; all our contributions are positive.

6.4 New algorithm for the Colouring problem

Here we will extend the algorithm we developed to the colouring problem. For this problem the difference between our algorithm and the Original algorithm will be larger than for the dimer problem.

Let us first consider what our distributions are; what we are actually counting. For the colouring problem, this is a graph where each edge has exactly one colour assigned such that no two edges share both a colour and a node; and that the total number of different colours is smaller than some integer c. (most of the interesting problems of this kind have a fixed number of E edges per node, and c = E) A partial distribution is thus a graph where some edges have colours assigned with the same limitations as above.

We now want to define states, similar to the dimer case. To find what our states are, we need to ask the following question: If we have a partial distribution, what are the restrictions on adding to it? We will then define those limitations as our state, with all the partial distributions that have the same limitations represented by that state.

The limitations are as follows: - we cannot assign a colour to an edge which connects to a node which already has an edge with that colour - we cannot assign a colour to an edge which already has a colour.

This means we need c 0/1 variables on each node to keep track of whether that node has edges with the various colours, and we also need a 0/1 variable on each edge to keep track of whether that edge has a colour yet.

However, we can eliminate the need for the edge variables by changing what we iterate over. By iterating over the edges, we know that each edge we have looked at has a colour assigned, and each edge we have not looked at does not. So we are implicitly keeping track of where we have colours, and do not need an explicit variable to do so for us.

Our state thus consists of c 0/1 variables for each node, one for each colour. Each variable signifies whether the node it belongs to already has an adjacent edge which has the the variables' colour assigned. We again define the empty state as the state which corresponds to no edges with a colour assigned, and the full state as the state where all edges have a colour assigned.

Our algorithm is as follows:

1. start with the empty state, $|E\rangle$.
2. for each edge, take each state in our system, and apply the following operator to that state:

$$O_i |n_1, ..., n_{c*N}\rangle = \sum_{j=0;c-1} \delta_{n^{c_j}_{p1(i)},1} \delta_{n^{c_j}_{p2(i)},1} |\{n\}_{\setminus\{n^{c_j}_{p1(i)}, n^{c_j}_{p2(i)}\}}\rangle$$

where p1(i) and p2(i) are the two nodes which the edge is adjacent to.
3. After we have gone through all the nodes, look at the counter in front of the full state, $|F\rangle$. That is the number of colouring distributions.

Chapter 7

Generalisation

7.1 A Generalisation of the dimer and colouring problems

Given the algorithm for the colouring problem derived in the previous section, we can see that it is curiously linked with the problem description for the colouring problem. A colouring distribution here is one where no two dimers of the same colour share a node, and each edge has exactly one dimer. In the algorithm, we are iterating over the edges, and our state is based on the nodes and colours. It seems there is a link between the iteration and the "each edge has exactly one dimer" part, as well as the state and the "no two dimers of the same colour share a node" part.

So what is the difference between these two parts? And can we find a similar link for the dimer problem?

The "no two dimers of the same colour share a node" part is a restriction. It describes something that must not happen for the colouring distribution to be valid. The opposite would be a requirement, where something must be true to create a valid colouring distribution. Asking for "each edge has exactly one dimer" isn't quite a requirement, in fact it can be split into two part: a requirement ("at least one dimer") and another restriction ("no more than one dimer").

As for the dimer problem, the problem description is that each node must have exactly one dimer, which can similarly be split into a requirement "each node must have at least one dimer" and a restriction "each node must not have more than one dimer". In each case, we iterate over the requirement, and use the state to enforce the restriction. For instance, in the dimer case, since each node must have no more than one dimer, we will need one variable on each node keeping track of how many dimers there are there connected to that nodes; with these variables taking the value 0 or 1 (we cannot have fractional or negative dimers connected to a node, and the restriction prohibits more than 1 dimer per node - hence only 0 or 1 are acceptable values)

Dimer Problem
Requirement: Each node must have at least one dimer.
Restriction: Each node must have no more than one dimer.

Colouring problem
Requirement: Each edge must have at least one coloured dimer.
Restriction: Each node must not have more than one dimer of each colour.

We can expand this idea to a general problem. If we want to count some sets which all have an iterable Requirement and a Restriction. We call these sets solutions. We also define a partial solution, which respects the Restriction, but not necessarily all of the Requirement. This partial solution will then have a partial restriction, which is based on which parts of the Restriction is

has already used and which can still be used. For instance, in the dimer case,

0. define states, such that each state obeys the Restriction and keeps track of how much of the Restriction we have already used. These states may fulfil any fraction of the Requirement. In particular note the state $|E\rangle$, which doesn't fulfil any of the partial requirement. Note that each state represents all of the partial solutions to our original that have those exact partial restriction. Add a counter variable to keep track of how many those are. Define an operator O_i which takes a state and creates a new state for each of the smallest possible changes that enforce part i of the Requirement, while making sure that the Requirement is will upheld and keeping track of the updated partial requirement.

1. start with the empty state, $|E\rangle$.
2. for each step in the Requirement, take each state in our system, and apply the operator O_i to that state.
3. After we have gone through all the nodes, add the prefactors of all the remaining states. That is the number of solutions.

7.2 Proof

In this section, we will prove that the generic algorithm we developed in the last chapter actually returns the correct result. Specifically, we want to know whether the answer we get is equal to the number of sets that fulfil both the Restriction and the Requirement of the problem. In order to do this, we need to prove two things: First, given a valid set, that it is counted exactly once. Secondly, that we do not count any anything that is not a valid set. The second is easy to show by simply considering how we have defined our states. We literally require all our states to obey the Restriction, so this is trivially given.

Now consider a valid solution. We will now prove that at each step, the algorithm contains exactly one state that represents a partial solution for that solution exactly once. To prove this, we use induction.

Initial: the initial state for our algorithm is a set containing only the empty state, which represents a partial solution for all solutions including ours, and it is present exactly once.

Step: We now assume that a set of states does contain exactly one which represents a partial solution for our solution, and want to prove that it still does this after one step. This means applying our operation O to the set of states. We do not need to consider the entire set of states, since any partial solution can only be built by adding partial solution. So a state that doesn't contain any partial solution for our solution cannot contain one after being subjugated to the operator O, since it just adds to the partial solutions.

Thus we just need to look at the state that represents our partial solution. When applying the operator, it ensures that a particular part of the Requirement is fulfilled. This can already be true, in which case the operator simply leaves the state as it; and thus also still leaves us with exactly one partial solution of our solution. If this particular part of the Requirement isn't met yet, the operator will create a new state for each of the smallest changes that can be made to fulfil that requirement. Now, only one of these changes can be true for our solution, since the solution cannot contain two distinct ways to fulfil that requirement. We thus have exactly one of the new states representing a partial solution for our solution.

Chapter 8

Implementation and Discussion

In this section we will discuss the implementation of the algorithm for the colouring problem, and discuss how well it works, how it scales and how it compares to other algorithms.

Coding was done in Python, for the actual code see the appendix.

Since our algorithm iterates over the edges of the graph, it made sense to represent the graph as a list of edges. The program works for any graph given, which is defined in the **createLinks** function. To change the graph that you wish to calculate, simply change that function. The createLinks function also has a few graphs that it can create built in. Specifically, any regular square, hexagonal, triangular or kagome grids.

Nodes are identified by integers. Now, it might seem reasonable to identify them by 1, 2, 3, 4, ... ; but this approach neglects two important points. First, in programming, it is normal to start sequences at 0, since all the core iteration functions do the same. Secondly, we need c variables for each node. Thus we denote the nodes by 0, c, 2c, 3c etc. Our colours will have values ranging from 0 to c-1; and each variable is labeled as m+d where m is the value of the node and d the value of the colour.

This gives us a total of Nc variables for one base state, as we want according to our theory. Since each of these variables is a simple 0/1 variable, we can represent the entire state by an integer of length Nc since computers store integers as binary anyway. Using the operator on a base state is done with the **process** function.

Our total state is simply a list of all our states with associated values. Since we unfortunately can't actually process all the base states in our state simultaneously, we also need a temporary state. Otherwise, already processed states might be processed again in the same step.

8.1 Runtimes and Scaling

While the program is theoretically capable of solving any graph, there are practical limitation because computers aren't infinitely fast. One of the most important things about a program like this is therefor knowledge of how the runtime scales with problem size.

Let N be the number of nodes, c the number of colour and E the number of edges for our graph. We can put a hard limit on the runtime scaling by observing that we have Nc variables, and thus 2^{Nc} possible base states. Each base state is processed at most once per step, and each step corresponds to an edge. Thus we have a runtime on the order of $O(E \times 2^{Nc})$ at most.

We can do a lot better than that for cases where the number of colours is equal to the number of edges per node. This was also an important point in the paper by Creutz [1]. In that case, once all the edges around a node have been processed, all variables associated with the node are occupied for all base states. This means that the number of base states gets a lot smaller. By carefully choosing the order in which we process, we can drastically reduce the runtime scaling. Let us say that we have an $X \times Y$ grid, where $X \geq Y$ (This also implies $Y \leq \sqrt{N}$). Then we get a cross-section of the graph which is currently being processed, which is Y. All other nodes are either fully processed or not processed at all; so they don't add to the number of base states we currently can have.

This gives us a runtime on the order of $O(E \times 2^{Yc})$, which is at most $O(E \times 2^{\sqrt{N}c})$.

8.2 Comparisons to the Original algorithm

The main difference between the New algorithm and the Original algorithm is the New Algorithm does not require any specific topology for the graph. By contrast, the Original algorithm either can't process complicated topologies, or requires several evaluations to compensate for the topology. For example, the toroidal case which we have worked with in this thesis requires 4 Pfaffians for the dimer case, and 4^c for the colouring case. There some symmetry between these, so the actual number you need to calculate is lower; for instance for the square with c=4 we get 35 distinct cases instead of $4^4 = 256$ [4]. However, even 35 distinct cases still means that you have to run the Original Algorithm 35 times to get the correct answer.

Beyond that, the New algorithm is also a bit simpler because there is no need to keep track of signs. Everything in the new algorithm is always positive.

When it comes to runtime, the New algorithm a bit faster than the Original; the removal of variables for the edges helps, as does the lack of compensation for topology. It still grows with the same order of magnitude with size, though.

8.3 The graphs

We have implemented several regular grids with periodic boundary conditions so that the number of edges per node is constant for the entire grid. This means that the grids can be embedded on a torus.

Implementation of the square grid is very easy, since all we need to know is the number of each node. By knowing the row and column of a node, we can infer its number; see figure 8.1.

For the other grids, they were transformed to fit the square grid too, since that was the easiest way to find the numbering for the nodes. The difference is only in the connection between the nodes, at least for the hexagonal and triangular grids, figures 8.2 and 8.3 respectively.

For the kagome grid (see figure 8.4), it is a bit more complicated, since we can't fill all of the nodes in a square grid; but that is mostly a problem for implementation to make sure that each node is assigned the correct number.

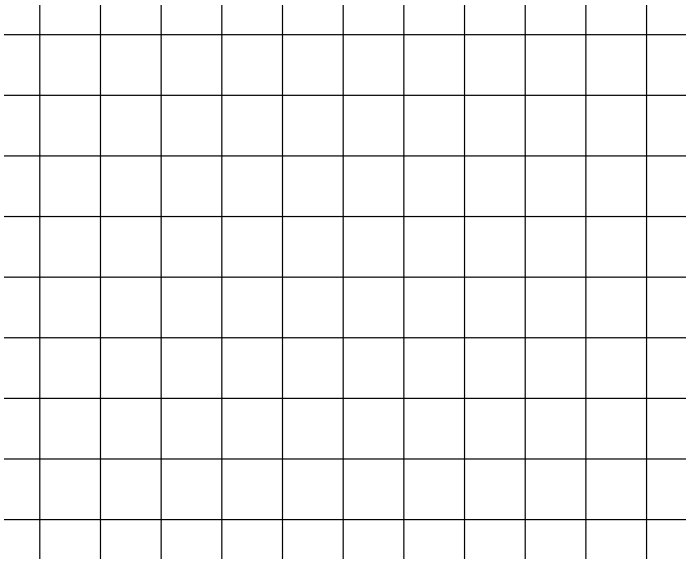

Figure 8.1: A square grid

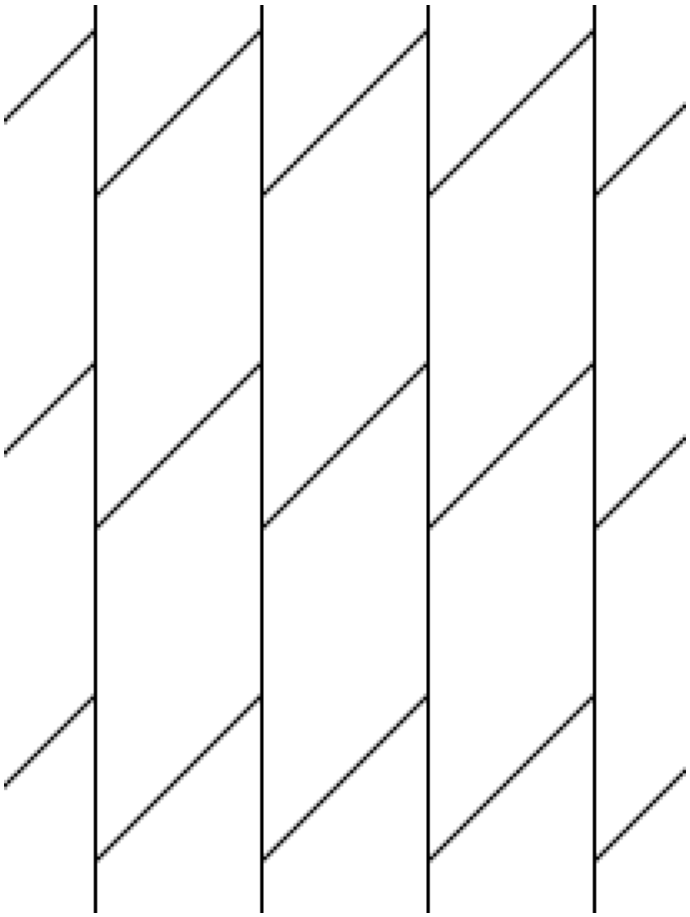

Figure 8.2: A hexagonal grid

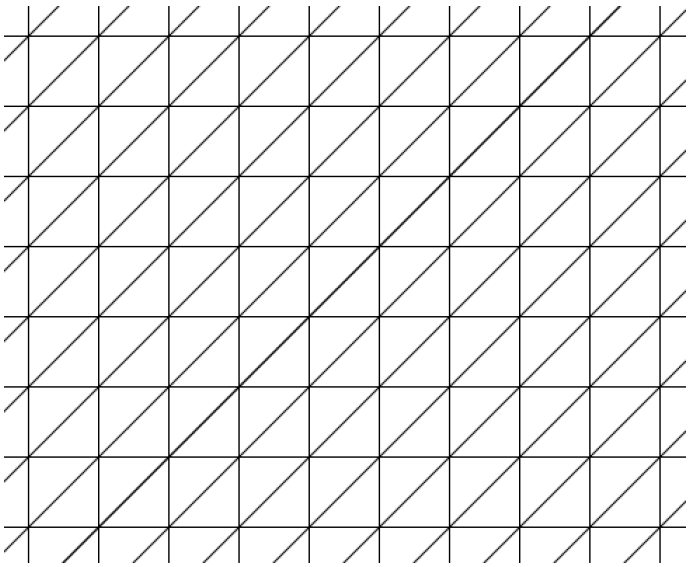

Figure 8.3: A triangular grid

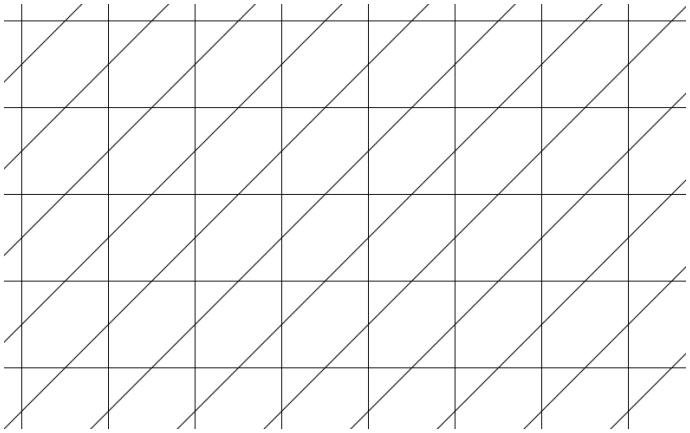

Figure 8.4: A kagome grid

Chapter 9

Results for the colouring problem

Table 9.1: Results for the square grid

Z	y	2	3	4	5	6	7
x							
2		96	192	1536	7680	47616	279552
3		192	0	1440	0	14688	0
4		1536	1440	44160	243840	3629568	35750400
5		7680	0	243840	0	17988960	0
6		47616	14688	3629568	17988960	1072652544	15664050528
7		279552	0	35750400	0	15664050528	0
8		1683456	168480	451209216	1840646400	589685031168	10486277124715400
9		10076160	0	5158471680	0	12978406493280	0
10		60481536	1998432	62658256896	209977817280	412755112206336	10237845291845200000

Table 9.2: Results for the kagome grid

Z	y	2	4	6	8	10	12
x							
2		0	96	0	1536	0	24576
4		96	768	6144	49152	393216	3145728
6		0	6144	0	3342336	0	2491416576
8		1536	49152	3342336			
10		0	393216	0			
12		24576	3145728	2491416576			
14		0	25165824				
16		393216	201326592				
18		0					

The results for these graphs have been verified in three ways, to ensure the absence of coding errors. Firstly, for very small grids, the correct number of colourings was derived by hand, and then compared to the results from the program. Secondly, for the square grid, the results were compared to previous results [4]. Thirdly, the results were compared to analytical results in the limit $N \to \infty$.

Table 9.3: Results for the hexagonal grid

Z	y	2	4	6	8	10	12
x							
2		12	24	48	96	192	384
3		24	48	120	408	1284	4752
4		48	96	408	2160	8208	36096
5		96	192	1284	8208	49416	317352
6		192	384	4752	36096	317352	3226032
7		384	768	17412	185184	1946964	30749232
8		768	1536	68088	916032	12153168	317511600
9		1536	3072	266232	4285632	80964996	3384078480
10		3072	6144	1058808	20484096	529208112	35145601224

For large grids, the number of colourings Z goes approximately as a constant W raised to the power of N,

$$Z \sim W^N \tag{9.1}$$

This gives us in the thermodynamic limit

$$W = \lim_{N\to\infty} Z^{1/N} \tag{9.2}$$

Of the grids which we look at in this thesis, W has been found for the square grid [10] and for the hexagonal grid [11]. We have

$$W_{square} = \frac{\Gamma^2(1/4)}{\sqrt{2}\pi^{3/2}} \approx 1.669253683 \tag{9.3}$$

$$W_{hexagonal} = \sqrt{\frac{3\Gamma^3(1/3)}{4\pi^2}} \approx 1.208717703 \tag{9.4}$$

These values seem reasonable compared to the W values we get from our program, see figure 9.1 for the hexagonal grid, and for a discussion for the square grid see [4]. Note that higher Y values give results that are closer to the thermodynamic limit.

Figure 9.2 shows the W values for the kagome grid. To my knowledge, no analytical result for W in the thermodynamic limit exists yet for this grid. The extrapolation lines aren't as strong as for the hexagonal grid, because we have fewer data points to work with. Still, we can use this to make a general estimate of W for the kagome grid in the thermodynamic limit to 1.39

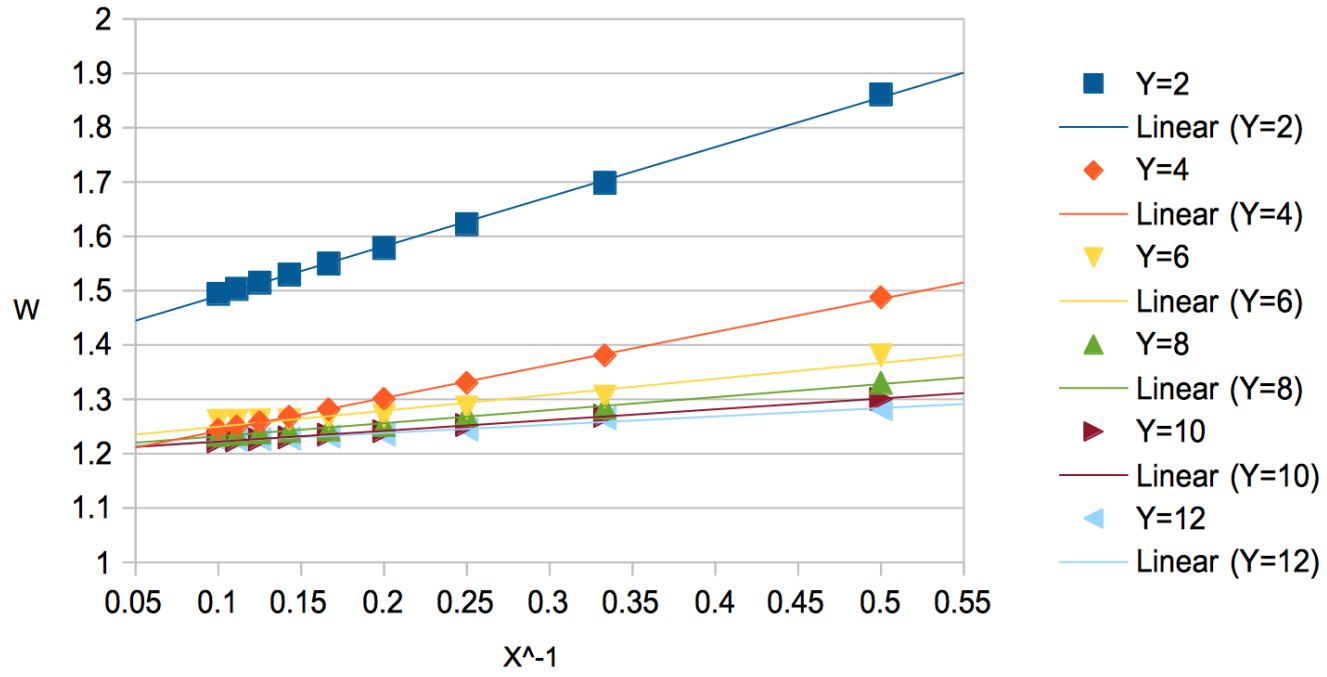

Figure 9.1: (color) W values for the hexagonal grid. The lines are the linear extrapolation of our data

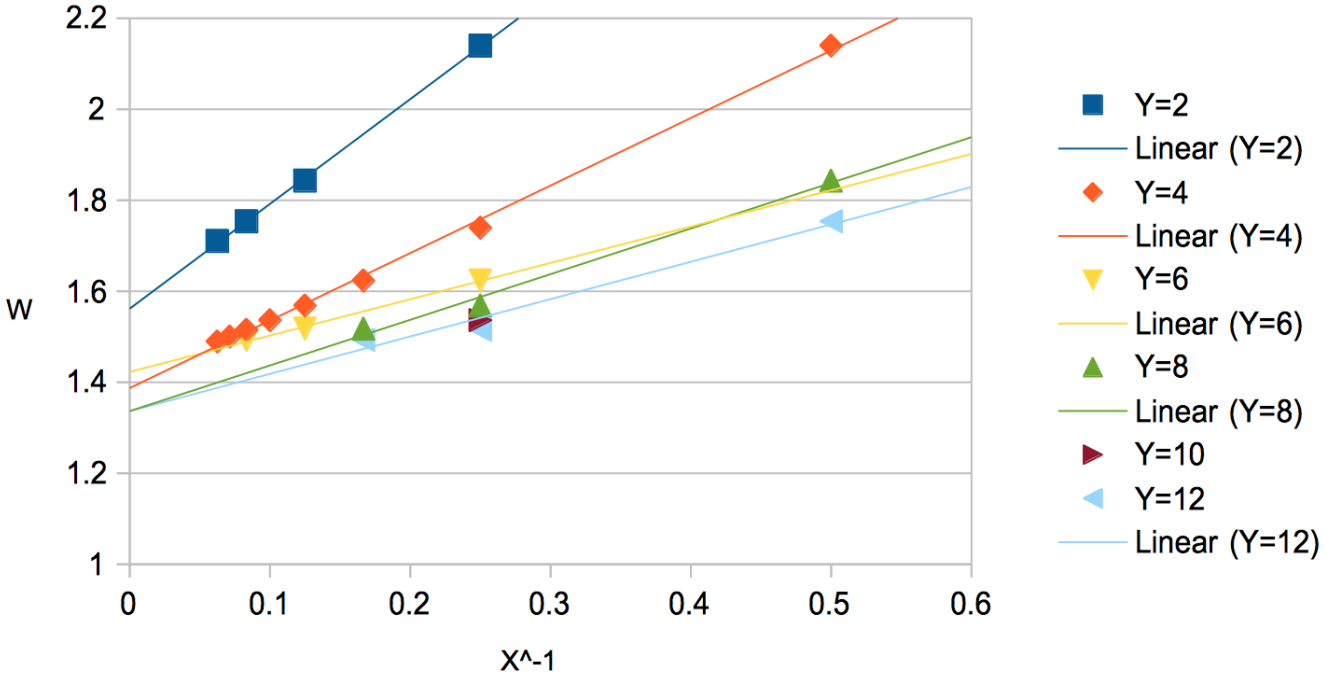

Figure 9.2: W values for the kagome grid

Chapter 10

Other problems

In chapter 7, we expanded the New algorithm to a general class of problems; any that can be represented as counting sets which have a restriction and a requirement. In this chapter, we will look at a few of the other problems which fall into this category, how their algorithm would look, and how effective it would be.

10.1 Weighted dimer problem

The extension to a weighted graph for the dimer problem is fairly straight forward. Instead of implicitly giving each edge the weight 1, we are instead giving it a weight ω_{ij}. Each dimer distribution has then a value which is the product of the weights of all its dimers. With our algorithm, we can find the maximal or minimal value, or we can find the sum of all the dimer distributions' values. Implicitly, this also gives us the average, since we already know how to find the total number of dimer distributions.

The requirement and restriction stay exactly the same, the only differences in order to add this functionality is to add a table which stores all the values, and change our operator. For the sum, we have:

$$O_i|n_1, ..., n_N\rangle = \delta_{n_i,0}|n_1, ..., n_N\rangle + \delta_{n_i,1} \sum_{j \in n(i)} \delta_{n_j,1}\omega_{ij}|\{n\}_{\setminus\{i,j\}}\rangle \tag{10.1}$$

For the minimal or maximal values, we store for each base state the current extreme value, instead of the sum. This means that instead of simply adding values to our database, we instead need to compare the new values we get from the use of the operator to the ones stored from before, and only change them if the current extreme value has been beaten.

10.2 Node colouring problem

Similar to the edge colouring problem, the node colouring problem is of interest to theoretical physics and has been studied numerically with algorithms, see e.g. [12].

The node colouring problem is closely related to the edge colouring problem. This can be seen easily when you consider that each edge colouring problem can be restated as a node colouring problem (by replacing each edge with a node. Two nodes in the restated problem share an edge if they shared a node before).

Requirement: All nodes must be have a colour.
Restriction: No edge can have two identical colours.

There are several ways to formulate the restriction, for instance you could say "No node can have the same colour as any adjacent node." The point of stating the restriction based on the edges is that this makes for easily handled states. Such an edge can have only c+2 different states: Either none of the adjacent edges have a colour, or exactly one edge has one colour (in which case we need to keep track of which colour that is, so that the other edge can't take the same), or both have a colour, in which case we can safely ignore the edge, since no further nodes are connected to it.

10.3 Travelling Salesman Problem

The travelling sale man problem, or TSP, asks for the shortest path between a given number of cities such that each city is visited exactly once. There are different versions depending on whether you start in the same city as you finish or not (basically, whether your path is a loop).

The TSP can be represented by a set of weighted dimers between nodes (cities), and we are interested in the smallest set of dimers that fulfils the following:

Requirement: All nodes must be connected by dimers.
Restriction: No node must have more than two dimers.

The connectivity basically means that each new dimer we add in our operator must have at least one node which is already connected. This is fine as long as we already have a dimer, so we must simply define one node as a starting node (before we have any dimers), which we connect everything else to.

State: Basically, nodes can have three different values: fully connected, partially connected and unconnected. Mostly, there will be two partially connected nodes, representing the endnodes of our current path; while the connected nodes represent all the nodes that our path goes through, without caring about the order (though we can record this order as part of our value for the state if we are interested in it) The value consists of the length of our state, which is the smallest length for any path between the two partially connected node which goes through all of the fully connected nodes and none of the unconnected.

Chapter 11

Summary and Conclusions

We have shown the theoretical background for the Original Algorithm for the dimer and edge-colouring problems, going through Pfaffians, Grassmann Integrals and Second Quantization. We then looked at the basic principles which make the Original Algorithm effective, especially building up solutions in parallel and having states which represent several partial distributions at once.

We proceeded to develop our own algorithm based on these insights. The algorithm we developed in this thesis is more flexible than previous algorithms for this problem, especially due the the ability to handle any kind of graph. Still, the inherent limitations of exponentially growing computing time makes it unfeasible to compute exact solutions for large grids using this method. It can be used to get a very rough estimate of the thermodynamical limit for grids, by looking at the progression of $Z^{1/N}$ for small grids. We found exact values for small square, hexagonal and kagome grids, and compared our results with the analytical thermodynamic limit. For the kagome grid, where no such analytical result exists yet, we estimated W in the thermodynamic limit to be about 1.39

Looking closer at the link between the original problem and the New Algorithm, we generalised our algorithm to any problem that concerns a set of solutions with one or more restrictions and one or more requirements. The generalised algorithm could be very useful for any problem for which the computation time is polynomial.

Bibliography

[1] M. Creutz, "Evaluating grassmann integrals," *Phys.Rev.Lett.*, vol. 81, no. 3555-3558, 1998.

[2] K. Binder, "Ising model."

[3] F. Y. Wu, "The potts model," *Rev. Mod. Phys.*, vol. 54, pp. 235–268, Jan 1982.

[4] J. O. Fjærestad, "Dimer and fermionic formulations of a class of colouring problems," *Journal of Physics A: Mathematical and Theoretical*, vol. 45, no. 7, p. 075001, 2012.

[5] P. W. Kasteleyn, "Dimer statistics and phase transitions," *Journal of Mathematical Physics*, vol. 4, no. 2, pp. 287–293, 1963.

[6] M. Mahajan, P. Subramanya, and V. Vinay, "A combinatorial algorithm for pfaffians," in *Computing and Combinatorics* (T. Asano, H. Imai, D. Lee, S.-i. Nakano, and T. Tokuyama, eds.), vol. 1627 of *Lecture Notes in Computer Science*, pp. 134–143, Springer Berlin Heidelberg, 1999.

[7] S. Samuel, "The use of anticommuting variable integrals in statistical mechanics. i. the computation of partition functions," *Journal of Mathematical Physics*, vol. 21, no. 12, pp. 2806–2814, 1980.

[8] P. W. Kasteleyn, "The statistics of dimers on a lattice : I. The number of dimer arrangements on a quadratic lattice," *Physica*, vol. 27, pp. 1209–1225, Dec. 1961.

[9] R. Shankar, *Principles of QM, 2nd edition*. Plenum Press, 1994.

[10] D. D. Cont and B. Nienhuis, "The packing of two species of polygons on the square lattice," *Journal of Physics A: Mathematical and General*, vol. 37, no. 9, p. 3085, 2004.

[11] R. J. Baxter, "Colorings of a hexagonal lattice," *Journal of Mathematical Physics*, vol. 11, no. 3, pp. 784–789, 1970.

[12] A. Bedini and J. L. Jacobsen, "A tree-decomposed transfer matrix for computing exact potts model partition functions for arbitrary graphs, with applications to planar graph colourings," *Journal of Physics A: Mathematical and Theoretical*, vol. 43, no. 38, p. 385001, 2010.

Appendix - Code

```
import sys

# for quicker solving, let the x value be the larger one
x = 2 # size in x-direction
y = 2 # size in y-direction
l = True # l=True means periodic boundary conditions
c = 3 # number of colors
gridtype = 1    # 0 square
                # 1 hexagonal
                # 2 triangle
                # 3 kagome (3-6 colouring)


class ColourCount():
    def __init__ (self, x, y, loopy, colour, gridtype):
        self.colors = colour
        self.size = x*y
        self.links = []
        self.createLinks(x, y, loopy, gridtype)
        self.linknum = len(self.links)
        initstate = (1 << (self.size * self.colors)) - 1
        self.system = { initstate : 1 }
        self.__main__()

    def __main__(self):
        self.tempsystem = {}
        j = 0
        s = len(self.links)
        for link in self.links :
            j += 1
            print ("Step " + str(j) + "/" + str(s) +
                " (size: " + str(len(self.system)) + ")")
            for state in self.system :
                self.process(link, state)
            self.system = self.tempsystem
            self.tempsystem = {}
        if 0 in self.system:
            print ("Result: " + str(self.system[0]))
        else:
            print (0)
```

```
def process(self, link, state):
    tempstates = []
    for c in range (self.colors):
        if (testBits(state, (link[0]+c, link[1]+c))):
            newstate = clearBits(state, (link[0]+c, link[1]+c))
            tempstates.append(newstate)
    for s in tempstates:
        if s in self.tempsystem:
            self.tempsystem[s] += self.system[state]
        else:
            self.tempsystem[s] = self.system[state]


def createLinks(self, x, y, loopy, gridtype):
    n = self.colors
    print (gridtype)
    if (gridtype == 0):
        for i in range (x):
            for j in range (y):
                self.links.append((n*(i+j*x), n*((i+1)%x+j*x)))
                self.links.append((n*(i+j*x), n*(i+((j+1)%y)*x)))
    elif (gridtype == 1):
        if (y%2 == 0):
            for i in range (x):
                for j in range (y):
                    self.links.append((n*(i+j*x), n*(i+((j+1)%y)*x)))
                    if (j%2 == 0):
                        self.links.append((n*(i+j*x),
                            n*((i+1)%x+((j+1)%y)*x)))
        else:
            print ("For a hexagonal grid,
                    the y values has to be divisible by 2.")
    elif (gridtype == 2):
        for i in range (x):
            for j in range (y):
                self.links.append((n*(i+j*x), n*((i+1)%x+j*x)))
                self.links.append((n*(i+j*x), n*(i+((j+1)%y)*x)))
                self.links.append((n*(i+j*x), n*((i+1)%x+((j+1)%y)*x)))
    elif (gridtype == 3):
        self.size = int(x*y*3/4)
        if (x%2 == 0 and y%2 == 0):
            for i in range (x):
                for j in range (y):
                    if (j%2 == 0):
                        self.links.append((n*(i+int(j*x*3/4)),
                            n*((i+1)%x+int(j*x*3/4))))
                        if (i%2 ==0):
                            self.links.append((n*(i+int(j*x*3/4)),
                                    n*(int(i/2)+int(j*x*3/4)+x)))
                        else:
                            self.links.append((n*(i+int(j*x*3/4)),
                                    n*((int(i/2)+1)%
```

```
                                        (int(x/2))+int(j*x*3/4)+x)))
                        else:
                            if (i%2 == 0):
                                self.links.append((n*(int(i/2)+
                                        int((j-1)*x*3/4)+x),
                                        n*((i)+int(((j+1)%y)*x*3/4))))
                                self.links.append((n*(int(i/2)+
                                        int((j-1)*x*3/4)+x),
                                        n*((i+1)+int(((j+1)%y)*x*3/4))))
            else:
                print ("For a star grid, both the x and y
                        values have to be divisible by 2.")




def testBit (integer, offset):
    mask = 1 << offset
    return not (integer & mask) == 0

def clearBit (integer, offset):
    mask = ~(1 << offset)
    return (integer & mask)

def testBits (integer, alist):
    for offset in alist:
        if not testBit(integer, offset):
                return False
    return True

def clearBits (integer, alist):
    for offset in alist:
        integer = clearBit(integer, offset)
    return integer


a = ColourCount(x,y,l,c,gridtype)
```