Graphical Stabilizer Decompositions For Counting Problems

Tuomas Laakkonen

St Catherine's College

University of Oxford

A dissertation submitted for the degree of

MSc in Mathematics and Foundations of Computer Science

Trinity 2022

Acknowledgements

Many thanks to Konstantinos Meichanetzidis and John van de Wetering for their guidance, insight, and enthusiasm, without which this dissertation would’ve been much shorter (I’ll let you decide if that’s a good thing). Thank you to Simran Chowdhry, for her inimitable brand of encouragement: “What page are you on now?", and to Julien Codsi, for sharing many engaging discussions, terrible puns, and exciting ideas.

Abstract

Building on the work of de Beaudrap, Kissinger, and Meichanetzidis [dKM21], in this dissertation, we examine the relationship between #SAT and the ZH calculus. We outline their reduction from #SAT to evaluating ZH diagrams, which shows that evaluating ZH diagrams is #P-hard, and we extend this to show that evaluating diagrams in certain fragments of the ZH calculus, is $\mathrm{FP}^{\#\mathrm{P}}$-complete. In particular, we show this holds for diagrams where all H-boxes have labels in $\mathbb{Q}[i, \sqrt{2}]$, which includes both the Clifford+T and Toffoli-Hadamard fragments natively [BKM$^+$21].

We show the action of the DPLL algorithm [DLL62] for SAT and its derivative CDP [BL99] for #SAT in terms of ZH-diagrams, and use diagrammatic methods to extend CDP to treat variables and clauses equally. Combining this with an algorithm for #2SAT by Wahlström [Wah08], we derive novel upper bounds in terms of clauses and variables for #kSAT that are independent of k, and better than brute-force for small clause densities of $\frac{m}{n} < 2.25$. This improves on the upper bound of Dubois [Dub91] whenever $\frac{m}{n} < 1.858$ and $k \geq 4$, and the average-case analysis by Williams [Wil04] whenever $\frac{m}{n} < 1.217$ and $k \geq 6$. We also obtain an unconditional upper bound of $O^*(1.88^m)$ for #4SAT in terms of clauses only.

Finally, we find novel stabilizer decompositions of certain ZH-diagrams and use this to adapt the algorithm of Kissinger and van de Wetering [Kv21] [KvV22] for evaluating ZX-diagrams using stabilizer decompositions to evaluate #2SAT instances. We examine the effect of different decomposition strategies on its runtime but conclude that the algorithm as given is not effective, and suggest avenues for further improvement.

Contents

List of Figures

CHAPTER 1

Introduction

The Boolean satisfiability problem, known as SAT, is the problem of determining whether, for some Boolean formula, there is an assignment of the variables so that the formula is true. This problem has been extensively studied and is used in practice to solve many scheduling and optimization problems since modern SAT solvers are extremely capable, and can solve instances with millions of variables.

The counting satisfiability problem, known as #SAT, instead asks how many assignments of variables there are that make a formula true. While SAT is the prototypical NP-complete problem, #SAT is even harder to solve and forms the prototypical problem for the complexity class #P [Val79]. In practice, however, #SAT solvers are also quite capable, with modern solvers capable of solving instances with tens of thousands of variables. Practical use cases for #SAT are mainly in the realm of AI, where it is used in probabilistic reasoning applications.

Recent work by Neil de Beaudrap, Aleks Kissinger, and Konstantinos Meichanetzidis [dKM21] has shown that the solution to a #SAT instance can be quite naturally encoded in the ZH calculus [BK19], a rigorous graphical language for reasoning about tensor networks. In this language, tensors and scalars are represented by diagrams. We will continue this and examine various angles of #SAT through the lens of the ZH calculus, and seek to demonstrate that diagrammatic methods, and the ZH calculus in particular, are useful and powerful tools for tackling this problem.

This dissertation is organized as follows. First, we will outline the embedding of #SAT instances into ZH-diagrams given by de Beaudrap et al [dKM21] in chapter three, and extend this in chapter four to show that for many ZH-diagrams, the reverse is also possible - that the value of a ZH-diagram can be written as a weighted sum of the solution to a small number of #SAT instances.

Then, we will examine how these ZH-diagrams can be decomposed by writing them as a sum of simpler tensors. We will use this to illustrate in chapter five how the classic DPLL algorithm [DLL62] for SAT and the similar CDP algorithm [BL99] for #SAT can be reframed in terms of ZH-diagrams. From this, we find that these diagrams can be generalized without affecting this algorithm, and derive some novel upper bounds on the runtime for the $\#SAT$ problem in certain situations.

Finally, in chapter six, we will show how we found some decompositions of a special type, called stabilizer decompositions, for certain tensors and, in chapter seven, use this to adapt an algorithm of Kissinger and van de Wetering [Kv21] to evaluate #SAT instances.

CHAPTER 2

Preliminaries

2.1 Tensors and Tensor Networks

An excellent introduction to diagrammatic reasoning about tensor networks is given by Coecke and Kissinger in Picturing Quantum Processes [CK17]. This section is a subset of the concepts presented there in chapters three and four.

For the purposes of this thesis, we will consider a tensor to be the generalization of a matrix or vector - that is to say, a tensor is a multidimensional array of numbers. The number of dimensions, the size of the dimensions themselves, and the type of number contained by the tensor can vary depending on the situation. To describe a particular number contained within the tensor, we can index it according to its position along each dimension, much like a multidimensional array is handled in computer programming, or the row and column indices of a matrix. We will consider tensors to have two distinct types of dimensions: inputs and outputs - each input dimension corresponds to a subscript index, and each output dimension corresponds to a superscript index.

Definition 2.1. *A tensor T with n inputs and m outputs over a ring $\mathcal{R}$ is a multidimensional array of size $d_1 \times d_2 \times \cdots \times d_n \times d^1 \times \cdots \times d^m$ such that*

$$T^{i^1 i^2 \cdots i^m}_{i_1 i_2 \cdots i_n} \in \mathcal{R} \quad \text{for every } i_k \in [0, d_k) \text{ and } i^k \in [0, d^k) \tag{2.1}$$

where $d_k \in \mathbb{N}$ and $d^k \in \mathbb{N}$ are called the dimensions of the kth input and output. We write $\mathcal{R}^{d^1 d^2 \cdots d^m}_{d_1 d_2 \cdots d_n}$ as the set of all such tensors. A tensor with no inputs and outputs is called a scalar, since it is just an element of $\mathcal{R}$.

For example, matrices are tensors with one input (column index) and one output (row index). An $n \times m$ matrix therefore has $d_1 = m$ and $d^1 = n$ - vectors, meanwhile, are tensors with one output and no inputs. Two operations are defined on tensors: contraction and addition, which are generalizations of matrix multiplication and addition, respectively. We will also encounter the tensor product and composition, which are special cases of contraction.

Definition 2.2. *Given two distinct tensors $A \in \mathcal{R}^{a^1\cdots a^{m_a}}_{a_1\cdots a_{n_a}}$ and $B \in \mathcal{R}^{b^1\cdots b^{m_b}}_{b_1\cdots b_{n_b}}$, and two sets of indices $O \subseteq [1, m_a]$ and $I \subseteq [1, n_b]$, where $|O| = |I| = k$ and $a^{O_i} = b_{I_i}$ for every $i \in [1, k]$, we define the contraction of A and B along O and I as a new tensor $A^O B_I$ such that*

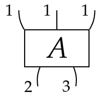
$$(A^O B_I)^{\vec{a}^{\bar{O}}\vec{b}}_{\vec{a}\vec{b}^{\bar{I}}} = \sum_{a_{O_1}=b^{I_1}=0}^{a_{O_1}-1} \cdots \sum_{a_{O_k}=b^{I_k}=0}^{a_{O_k}-1} A^{\vec{a}}_{\vec{a}} B^{\vec{b}}_{\vec{b}} \tag{2.2}$$

where $\vec{a}$ and $\vec{b}$ refer to all upper or lower indices of A and B and $\vec{a}^{\bar{O}}$ refers to all upper indices of A that are not in O, and similarly with $\vec{b}_{\bar{I}}$.

Informally, this means contraction is a sum over all possible values of all indices that are not in the output tensor, with corresponding indices set to the same value. For matrices, this corresponds to the elements of the product of two matrices (which is the contraction of the rows of one matrix with the columns of another) being dot products. Since this notation is complete index soup, we will instead consider a graphical notation for contractions, called tensor networks.

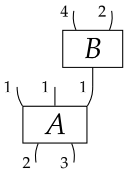
In this graphical scheme, we will represent tensors as shapes with inputs and outputs represented by loose wires. Each wire can be marked with the dimension corresponding to its index (if it is implied, this is omitted), and they are read either left to right or bottom to top: inputs are wires from the bottom or left of the shape, outputs are wires from the top or right. For instance, if a tensor A has two inputs of dimensions two and three and three outputs of dimension one, it would be represented like this:

$$A \in \mathcal{R}^{1,1,1}_{2,3} = \tag{2.3}$$



To represent a contraction, we will place multiple tensors together in a diagram, and connect the wires corresponding to the inputs and outputs that are contracted. For instance, take A as before and $B \in \mathcal{R}^{4,2}_{1}$ then we will write a contraction as follows:

$$A^3 B_1 = \tag{2.4}$$



From this representation, we can easily see the two special cases of contraction. For tensor product, we perform a contraction with no inputs or outputs, and for composition, we perform a contraction of all the inputs of one tensor, with all the outputs of another.

Definition 2.3. *The tensor product is given as:*

$$A \otimes B = \boxed{A} \otimes \boxed{B} = A^{\varnothing} B_{\varnothing} = \boxed{A} \quad \boxed{B} \tag{2.5}$$

The composition of two tensors is given as:

$$A \circ B = \boxed{A} \circ \boxed{B} = A_{\vec{a}} B^{\vec{b}} = \begin{array}{c} \boxed{A} \\ (\cdots) \\ \boxed{B} \end{array} \tag{2.6}$$

Like multiplication, we may also write composition as juxtaposition: i.e $AB = A \circ B$.

We can then see from the structure of the sum, and also directly from this diagrammatic representation that contraction is associative (up to a renaming of indices). Some other relationships, such as the following exchange identity, also become immediately clear:

$$(A \circ B) \otimes (C \circ D) = \begin{array}{c} \boxed{A} \\ (\cdots) \\ \boxed{B} \end{array} \quad \begin{array}{c} \boxed{C} \\ (\cdots) \\ \boxed{D} \end{array} = (A \otimes C) \circ (B \otimes D) \tag{2.7}$$

We will now define tensor addition, which is a very straightforward generalization of matrix addition as elementwise array addition. This will have no special diagrammatic interpretation, but rather sums of tensors can simply be represented as sums of diagrams.

Definition 2.4. *Given two tensors $A \in \mathcal{R}^{d^1 \cdots d^m}_{d_1 \cdots d_n}$ and $B \in \mathcal{R}^{d^1 \cdots d^m}_{d_1 \cdots d_n}$, define the tensor $A + B$ as follows:*

$$(A + B)^{i^1 \cdots i^m}_{i_1 \cdots i_n} = A^{i^1 \cdots i^m}_{i_1 \cdots i_n} + B^{i^1 \cdots i^m}_{i_1 \cdots i_n} \tag{2.8}$$

Clearly, for matrices and vectors, this translates directly from the definition of addition. Addition of tensors or tensor networks is also commutative and associative by the properties of the underlying ring, and contraction is linear over tensor addition. Therefore, we can define a vector space on tensors of a particular shape, seeing that scalar multiplication cA for a scalar c and tensor A is just given by $c \otimes A$ - note this vector space must have at least one basis.

Definition 2.5. *We define the computational basis [CK17, 5.3.4] of a vector space of tensors of the form $\mathcal{R}^{d^1 \cdots d^m}_{d_1 \cdots d_n}$ as*

$$\{ |x_1 x_2 \cdots x_m\rangle \otimes \langle y_1 y_2 \cdots y_n| \mid x_i \in [0, d^i),\ y_i \in [0, d_i) \} \tag{2.9}$$

where $|x_1 x_2 \cdots x_m\rangle$ is the tensor $T \in \mathcal{R}^{d^1 \cdots d^m}$ given by $T^{i^1 \cdots i^m} = \prod_{j=1}^{m} \delta_{i^j x_j} = \delta_{\vec{i}\vec{x}}$ and $\langle y_1 y_2 \cdots y_n|$ is defined by the tensor $T \in \mathcal{R}_{d_1 \cdots d_n}$ such that $T_{i_1 \cdots i_m} = \delta_{\vec{i}\vec{y}}$.

In the case that the tensors have no inputs (or outputs), then $|x_1 \dots x_m\rangle$ (or $\langle y_1 \dots y_m|$) form a basis directly. We call a tensor with no inputs a state, and a tensor with no outputs an effect. Composing a state with an effect will give a scalar and is the direct analogue of multiplying a row vector with a column vector, so it represents a dot product.

With these basis states defined, we can move on to the last pieces needed for general tensor networks: identity maps, cups, and caps. So far, we have only allowed contraction between distinct tensors - i.e not between two indices on the same tensor. In order to make this operation well defined, we introduce caps and cups.

Definition 2.6. *A cup is a two-output state tensor of the form $\mathcal{R}^{dd}$ for some dimension d, defined by*

$$\cup \;=\; \sum_{i=0}^{d-1} |ii\rangle \tag{2.10}$$

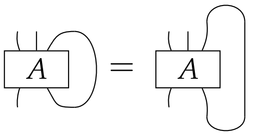
and similarly, a cap is a two-input effect tensor of the form $\mathcal{R}_{dd}$ defined by:

$$\cap \;=\; \sum_{i=0}^{d-1} \langle ii| \tag{2.11}$$

We also give the identity tensor, which has the form $\mathcal{R}_d^d$, and denote it by a straight wire:

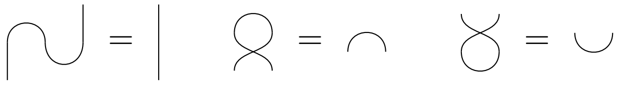
$$| \;=\; \sum_{i=0}^{d-1} |i\rangle \langle i| \tag{2.12}$$

Using caps and cups, we can represent a contraction between an input and output of the same tensor by first contracting an output with a cap and an input with a cup, and then contracting the other output of the cup with the input of the cap. Diagrammatically, this looks like this:



(2.13)

In terms of matrices, this represents the trace operation, but it is generalized to be considered over just part of a tensor. Cups and caps satisfy several nice symmetries, called the yanking equations [CK17, 4.1.3]:



(2.14)

Because of these symmetries, tensor networks have the property of *only connectivity matters*: when drawing a tensor network, the placement of the tensors, and the shape of wires connecting indices doesn't matter, only which inputs are connected to which outputs, and which order the inputs and outputs are in.

2.2 Graphical Calculi

So far, we have considered tensor networks as simply an alternative notation for tensors, and so networks are only equal when the underlying tensors are equal. We will now define several sets of specific generating tensors along with rules that govern how these tensors combine in networks, all presented in graphical format. Since we have graphical rules for manipulating these tensor networks, we can treat them as objects in their own right, called diagrams, distinct from the corresponding tensor. A set of generator diagrams together with rewrite rules is called a graphical calculus.

Definition 2.7. *A graphical calculus is a set of (possibly parameterized) generator diagrams corresponding to tensors over a ring $\mathcal{R}$, from which every diagram is constructed through contractions, and rewrite rules equating families of diagrams. Each diagram D has an associated tensor, given by $\llbracket D \rrbracket$.*

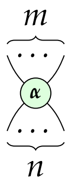
A calculus is called universal *if every tensor over $\mathcal{R}$ (with some restriction on the possible dimensions of the tensor), can be written as a diagram, up to scalar multiplication. A calculus is called* sound *if for every pair of diagrams D_1 and D_2, $\llbracket D_1 \rrbracket = \llbracket D_2 \rrbracket$ if $D_1 = D_2$ according to the rewrite rules of the calculus. A calculus is called* complete *if for every pair of diagrams D_1 and D_2, $D_1 = D_2$ according to the rewrite rules of the calculus if $\llbracket D_1 \rrbracket = \llbracket D_2 \rrbracket$.*

2.2.1 The ZX Calculus

The ZX calculus is a graphical calculus first introduced by Bob Coecke and Ross Duncan in 2008 [CD11], designed to succinctly represent the structure of quantum computations. It has been presented in several different ways, but we will present it as a graphical calculus of tensors over the complex numbers with three generators.

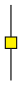
Definition 2.8 ([CD11, Figure 1])**.** *The ZX-calculus is a graphical calculus over $\mathbb{C}$ where all inputs and outputs of diagrams have dimension two. It is given by the following generators:*

- *The Z-spider, which is parameterized by a number of inputs n, outputs m, and a phase $\alpha \in \mathbb{R}$. If α is unspecified, it is assumed to be zero.*

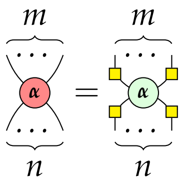
$Z_n^m[\alpha] =$ 

$$\llbracket Z_n^m[\alpha] \rrbracket = |\overbrace{0\cdots0}^{m}\rangle\,\langle\overbrace{0\cdots0}^{n}| + e^{i\alpha}\,|\overbrace{1\cdots1}^{m}\rangle\,\langle\overbrace{1\cdots1}^{n}| \tag{2.15}$$

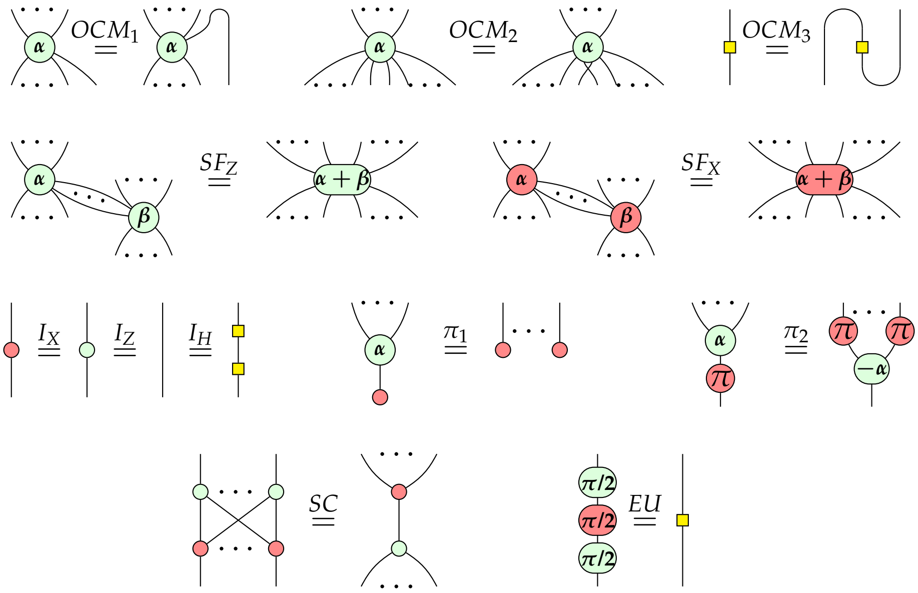
- *The Hadamard gate, which is a tensor with one input and one output.*

$H =$  $$[\![H]\!] = \frac{1}{\sqrt{2}} \left(|0\rangle \langle 0| + |1\rangle \langle 0| + |0\rangle \langle 1| - |1\rangle \langle 1| \right) \tag{2.16}$$

- *The X-spider, which is parameterized by a number of inputs n, outputs m, and a phase $\alpha \in \mathbb{R}$, and is derived from the Z-spider.*

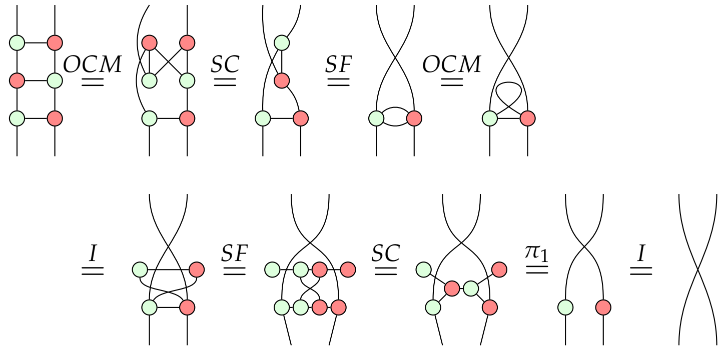
$\cdot X_n^m[\alpha] =$  (2.17)

The rewrite rules for the calculus are as follows, up to scalar multiplication:



The first three rules, $OCM_{1,2,3}$ assert that spiders are flexsymmetric [Car21, Chapter 5], and the Hadamard gate is self-transpose. This, together with the yanking equations, means that ZX-diagrams have a stronger form of only connectivity matters, in which the inputs and outputs of spiders can be deformed at will, so long as the (undirected) connectivity between elements of the diagram is maintained.

We can see that the properties of the generators of ZX-calculus are thus reflected in the shape of the diagrams used to represent them: spiders are circular because their legs can be moved about at will - they are radially symmetric - and Hadamard gates are square because they are self-transpose - vertically symmetric - but still have two distinct legs. To show an example of how we can use graphical techniques to prove identities about tensors, consider the following derivation

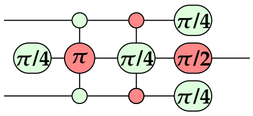


where the steps marked OCM are applications of the strong only connectivity matters property of ZX-calculus. Note that we have ignored scalar factors in the derivation, which we will always do unless stated otherwise. Also we see that several rules are applied in orientations other than those presented above, or with their colors swapped - this is permissible in ZX-calculus. Several other nice properties of ZX-calculus are known, including universality [CK17, 9.4.1] for all tensors over $\mathbb{C}$ with every dimension equal to two, soundness, and completeness (albeit with a slightly larger ruleset than presented here) [JPV19].

2.2.2 The ZH Calculus

While the ZX calculus is universal, it is not necessarily good at expressing all tensors concisely. In particular, tensors that compute functions related to the logical AND gate on basis states (i.e a tensor $\wedge$ such that $\wedge_{ij}^{k} = 1$ if $i \wedge j = k$ and $\wedge_{ij}^{k} = 0$ otherwise) are difficult to express - the smallest known decomposition for the AND gate in ZX is given [Kv21, Equation 20] as:

$$\wedge = \qquad (2.18)$$

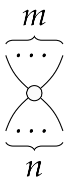


Since this is quite large, reasoning about classical logical operations in ZX can become difficult. As such, the ZH-calculus was introduced by Miriam Backens and Aleks Kissinger in 2018 [BK19], and provides a generalization of the Hadamard gate to alleviate these problems.

Definition 2.9 ([BK19]). *The ZH calculus is a graphical calculus over $\mathbb{C}$ where all inputs and outputs of diagrams have dimension two. It is given by the following generators:*

- *The Z-spider, which is parameterized by a number of inputs n and outputs m.*

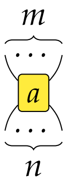
$$Z_n^m =$$



$$[\![Z_n^m]\!] = |\overbrace{0\cdots0}^{m}\rangle\langle\overbrace{0\cdots0}^{n}| + |\overbrace{1\cdots1}^{m}\rangle\langle\overbrace{1\cdots1}^{n}| \qquad (2.19)$$

- *The H-box, which is parameterized by a number of inputs n, outputs m, and a label $a \in \mathbb{C}$. If a is unspecified, it is assumed to be -1.*

$$H_n^m[a] =$$

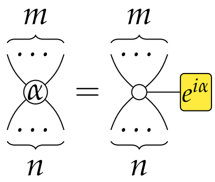


$$[\![H_n^m[a]]\!] = \sum_{i^k \in [0,1]} \sum_{i_k \in [0,1]} a^{i_1 \cdots i_n i^1 \cdots i^m} \left|i^1 \cdots i^m\right\rangle \left\langle i_1 \cdots i_n\right| \tag{2.20}$$

Note that $[\![H_n^m[a]]\!]_{i_1 \cdots i_n}^{i^1 \cdots i^m}$ equals a if $i_1 = \cdots = i_n = i^1 = \cdots = i^m$, and one otherwise - it is a generalized AND gate which equals a when it is true, and one otherwise. Additionally, we have that $[\![H_0^0[a]]\!] = a$, so it is easy to represent scalars and scalar multiplication in ZH, unlike in ZX. We will further define two derived generators, for convenience:

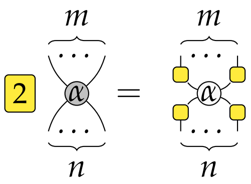
- *The labelled Z-spider, which is parameterized by a number of inputs n, outputs m, and a phase $\alpha \in \mathbb{R}$. Note that the unlabelled Z-spider is equivalent to a Z-spider labelled with $\alpha = 0$.*

$$Z_n^m[\alpha] = \quad = \tag{2.21}$$



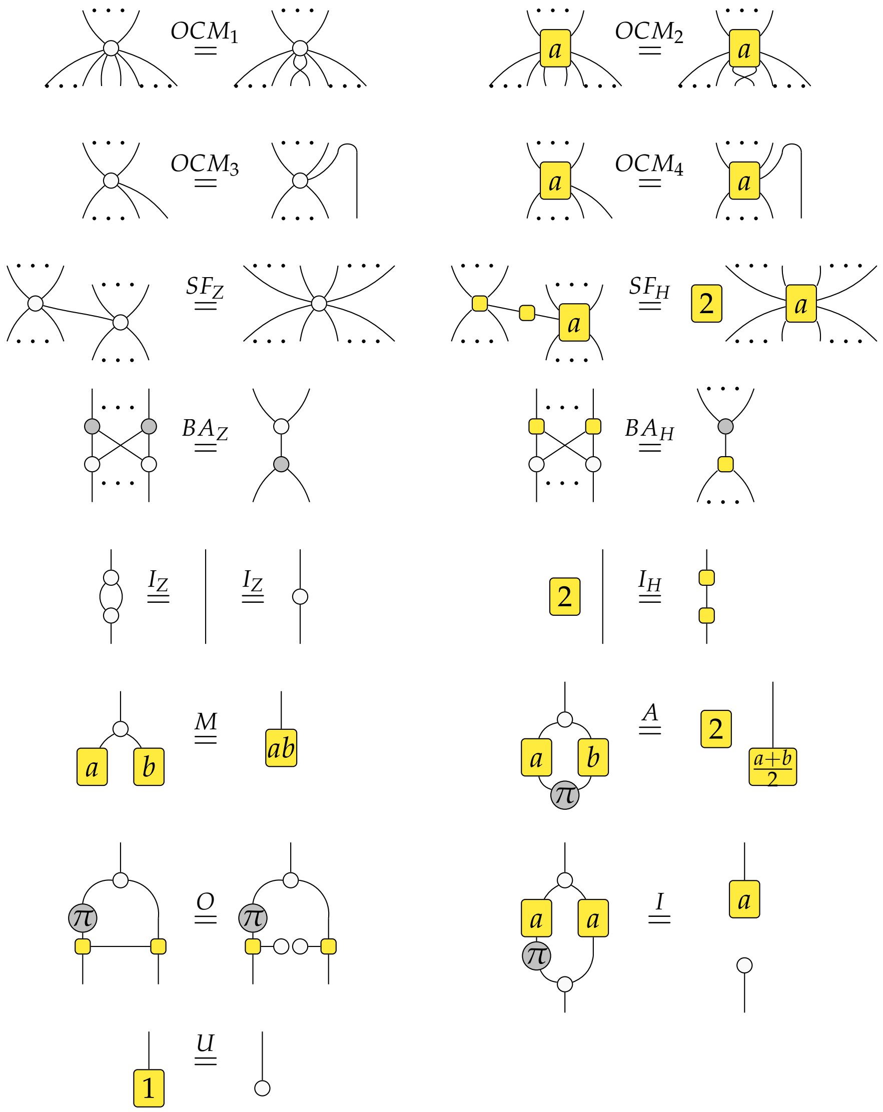
- *The X-spider, which is parameterized by a number of inputs n, outputs m, and a phase $\alpha \in \mathbb{R}$. If α is omitted, it is assumed to be zero.*

$$2 \cdot X_n^m[\alpha] = \quad = \tag{2.22}$$

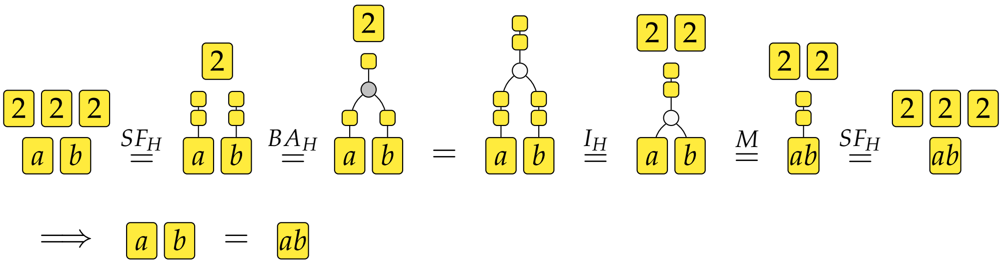


We can see then that the ZX and ZH calculus are closely related - their Z-spiders are exactly equivalent, while their X-spiders only differ by a scalar factor (although this depends on the parameters of the spider). Like ZX, ZH also allows the strong form of only connectivity matters, as all spiders and H-boxes are flexsymmetric.

The rewrite rules of ZH-calculus are as follows:



Some of these rules also apply with the colours swapped (for instance, $OCM_{1,3}$, SF_Z, I_Z, and BA_Z), like ZX-calculus, but not all of them, e.g BA_H. The ZH calculus is also universal, sound, and complete [BKM$^+$21] in the same way as ZX, but unlike ZX we will not usually ignore scalar factors in diagrams, since they are much easier to work with ZH. In particular, using the fact that scalar H-boxes are just complex numbers, we have the following:



(2.23)

2.3 Counting and Satisfiability

Boolean satisfiability is a well-known problem in computer science that asks whether a given formula of propositional logic has at least one assignment of variables that makes the formula true. This is the prototypical NP-complete problem [Coo71], and is widely regarded to be essentially difficult to compute [IP99], despite significant progress made on algorithms with extremely good performance in practice. Model counting is a generalization of satisfiability that instead asks how many assignments make the formula true.

Definition 2.10. *A Boolean formula on n variables is a function of type* $\{0,1\}^n \to \{0,1\}$*. For a given boolean formula* ϕ*,* SAT *is the problem of determining whether there exist* $x_1, x_2, \dots, x_n \in \{0,1\}$ *such that* $\phi(x_1, x_2, \dots, x_n) = 1$*, and* #SAT *is the problem of determining the value* $|\{x_1, \dots x_n \mid \phi(x_1, \dots, x_n) = 1\}|$*.*

While decision problems like SAT exist in complexity classes like P, NP, or $PSPACE$ and are defined in terms of whether a Turing machine accepts or rejects, different complexity classes exist for counting problems like #SAT. One such class is #P, which was first defined by Leslie Valiant in 1979 [Val79].

Definition 2.11 ([Val79])**.** *A counting Turing machine is a nondeterministic Turing machine which, after a computation terminates, outputs how many nondeterministic choices would result in the machine accepting, given an input of size n. It is said to run in* $f(n)$ *time, if the longest possible computation (taken over all nondeterministic choices) takes* $f(n)$ *many steps.*

#P *is the class of problems that can be computed as the output of a counting Turing machine running in polynomial time.*

Clearly, #SAT $\in$ #P, since a Turing machine that takes a polynomially sized Boolean formula as input, chooses a satisfying assignment nondeterministically and accepts if one exists, runs in polynomial time, and the number of accepting choices is exactly the number of satisfying assignments. Since SAT is not just in NP but NP-complete [Coo71], it seems natural to ask whether a similar result holds for #P. Completeness and hardness for complexity classes is often defined in terms of reductions, such as the following.

Definition 2.12. *A Turing reduction from one problem A to another problem B is defined as a Turing machine that can solve instances of A given access to an oracle for problem B. A Cook reduction is a Turing reduction that runs in polynomial time.*

A polynomial-time counting reduction is a Cook reduction in which the oracle for problem B is only allowed to be used once. Furthermore, a parsimonious reduction is a polynomial-time counting reduction where no further computation is carried out after the oracle is used.

While for decision problems, what it means for a problem to be complete for a given class is well defined, this is not so clear for counting problems. We will be using the definitions of #P-hard and #P-complete given by Valiant [Val79], but these are more permissive than some other definitions.

Definition 2.13. *A problem A is* #P*-hard if there is a Cook reduction from every problem in* #P *to A. A problem is* #P*-complete if it is both in* #P *and* #P*-hard.*

While plenty of #P-complete problems exist, it seems that this definition can be generalized further. In particular, if there is a Cook reduction from some problem A to every #P-complete problem, A may not be in #P, and thus cannot be #P-complete, but it is intuitively clear that A cannot be harder than #P since it can be computed using an #P-complete oracle with only polynomial overhead. Therefore, we will call a problem $\mathrm{FP}^{\#\mathrm{P}}$-complete if it is #P-hard, but there is a Cook reduction from it to every problem that is #P-complete.

Lemma 2.1. *If there is a Cook reduction from B to A and B is* #P*-complete, then A is* #P*-hard. If there is a Cook reduction from A to B and B is* #P*-complete and A is* #P*-hard, then A is* $\mathrm{FP}^{\#\mathrm{P}}$*-complete.*

Proof. First, note that the existence of Cook reductions is transitive - if there is a reduction from A to B and B to C, there is a Cook reduction from A to C constructed by replacing the calls to the oracle for B by the reduction from B to C. Since the product of two polynomials is polynomial, this is still polynomial time.

Therefore, for the first part, for any problem C in #P there is a Cook reduction from C to B by definition, and so by assumption and transitivity, a reduction from C to A. Therefore, A is #P-hard. For the second part, for any #P-complete problem C, there is a reduction from B to C by definition, and thus from A to C by transitivity. Therefore, there is a reduction from A to every #P-complete problem, and A is #P-hard by assumption, so it is $\mathrm{FP}^{\#\mathrm{P}}$-complete. □

Returning to the matter of #SAT, it is known that this is #P-complete. In fact, this follows immediately from the original proof that SAT is NP-complete, the Cook-Levin theorem [Coo71], since this gives a reduction from any problem in NP to SAT, such that every accepting path of the Turing machine defining the problem corresponds to exactly one satisfying assignment of a Boolean formula.

For any instance of a problem in #P, applying the Cook-Levin theorem to the defining counting Turing machine as if it were a standard nondeterministic Turing machine will produce a #SAT instance that can be solved with a #SAT oracle to determine the correct solution. Thus there is a reduction from any problem in #P to #SAT, so it is #P-hard, and thus also #P-complete [Val79].

For the rest of this dissertation, we will not usually be concerned with general Boolean formulae, but rather those with a specific form. In particular, we will often use conjunctive normal form.

Definition 2.14. *A Boolean formula ϕ on n variables is in k conjunctive normal form (kCNF) with m clauses if it is of the following form*

$$\phi(x_1, \ldots, x_n) = (l_{11} \vee \cdots \vee l_{1k_1}) \wedge \cdots \wedge (l_{m1} \vee \cdots \vee l_{mk_m}) \tag{2.24}$$

where each of l_{ij} is called a literal, and is either a variable x_i or a negation of a variable $\neg x_i$, and every $k_i \leq k$. We will call quantity $\delta = \frac{m}{n}$ the density of the formula.

We will call the SAT and #SAT problems where formulae are restricted to *k*CNF form *k*SAT and #*k*SAT. Note that #*k*SAT is still #P-complete for any $k \geq 2$, even though *k*SAT is only NP-complete for $k \geq 3$, and indeed 2SAT $\in P$, as we will show in the next chapter.

For the case of $k \geq 3$, #P-completeness of #*k*SAT follows from a parsimonious reduction from general SAT to *k*SAT and then 3SAT. Whereas for #2SAT, no such reduction can exist, since 2SAT $\in P$, and so #P-completeness was proven by Valiant [Val79] via a reduction to the matrix permanent - we will elaborate on the relationship between #2SAT and #*k*SAT in chapter four.

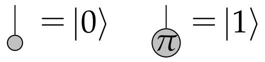
CHAPTER 3

Counting with Diagrams

This chapter is an exposition of the ideas given by Neil de Beaudrap, Aleks Kissinger, and Konstantinos Meichanetzidis in the paper "Tensor Network Rewriting Strategies for Satisfiability and Counting" [dKM21] and does not have any novel content.

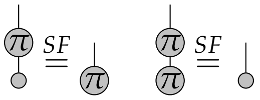
3.1 Boolean Logic in ZH-Diagrams

It was mentioned in Definition 2.9 that the ZH-calculus H-box is in fact a generalized form of the classical AND gate, and in light of this it seems natural to ask how we can represent Boolean formulae as ZH-diagrams. Since every wire in a ZH-diagram has dimension two, the computational basis consists of two states $|1\rangle$ and $|0\rangle$ which we can use to represent the true and false states of Boolean logic, respectively. In a ZH-diagram, we can represent these states as



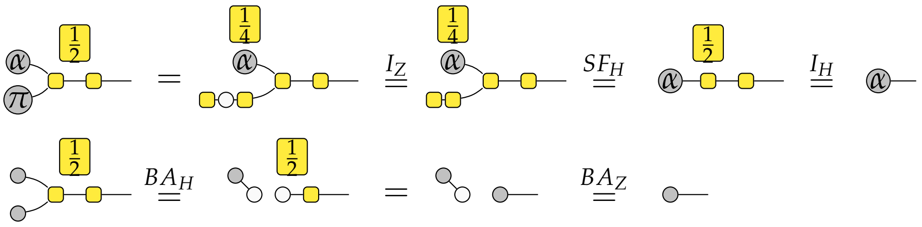
(3.1)

which can be verified by concrete calculation of the tensors. It is well-known that the set $\{\text{AND}, \text{NOT}\}$ of Boolean logic gates can be used to represent any Boolean function, so we shall seek to replicate these in ZH. For the NOT gate, we have the following:



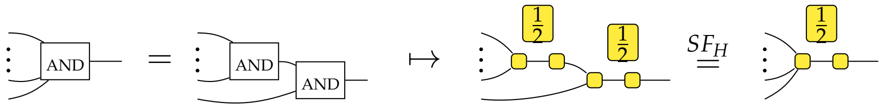
(3.2)

so we can see that $X_1^1[\pi]$ acts as a NOT gate. The AND gate can then be constructed from two H-boxes, by noting the following relations on basis states



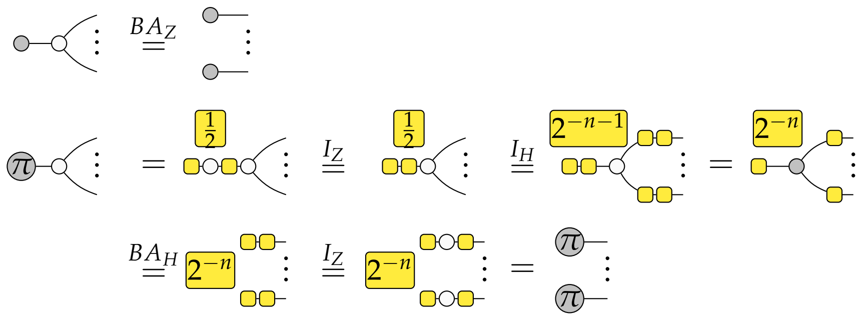
(3.3)

and then to generalize this to multiple inputs, we can stack AND gates together in an arbitrary order by associativity. This can be simplified to a single pair of H-boxes by induction using the following equation:



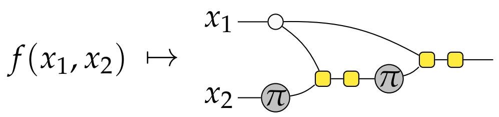
(3.4)

The diagram constructed so far can only make use of each input state once since every input is a loose wire. However, in many Boolean functions, we will need to use an input more than once. To this end, note that we can copy a basis state using a Z_1^2 spider

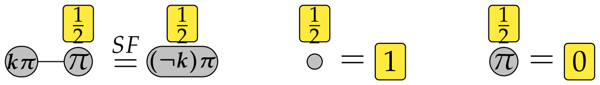


(3.5)

and so we can represent any Boolean function as a ZH-diagram. For example, the function $f(x_1, x_2) = x_1 \wedge \neg(\neg x_2 \wedge x_1)$ would be represented as:

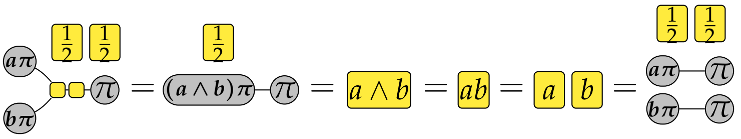


While designing a diagram that produces basis states as outputs is one embedding of a Boolean formula, we could also create a diagram that produces a scalar as output, either zero or one, which corresponds to the output of the Boolean formula directly. We can convert basis states to their corresponding scalars by applying a basis effect to the output of a diagram (also called a post-selection) since we have that



(3.6)

which can be verified by concrete calculations. Also, by exploiting the fact that multiplying scalars is the same as the AND gate when the scalars are zero or one, we can see that post-selecting after an AND gate is the same as post-selecting before the AND gate:

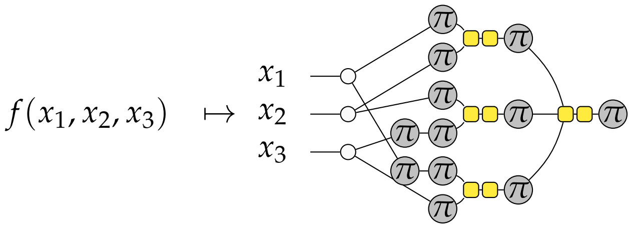


(3.7)

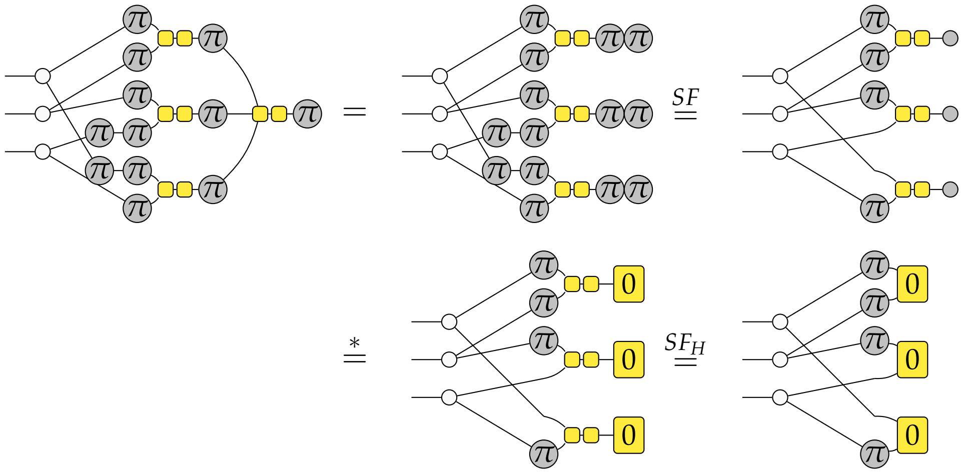
To construct a diagram that has a scalar value equal to the value of a CNF formula, we have four layers:

- First, every variable is copied zero or more times using a Z-spider.
- Then every literal is formed by negating variable copies wherever needed.
- The literals are fed into clauses, which consist of AND gates with all inputs and output negated (as this is equivalent to an OR gate by De Morgan's laws).
- The outputs of the clauses are fed into AND gates and the outputs are post-selected to give a scalar value.

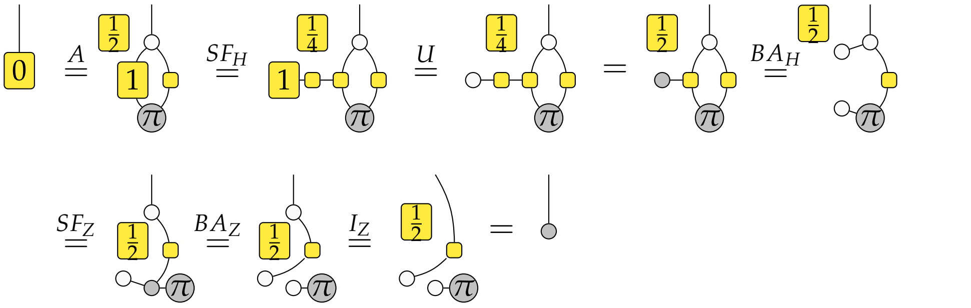
For example, with the formula $f(x_1, x_2, x_3) = (x_1 \vee x_2) \wedge (x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_3)$ we have the following diagram (ignoring scalar factors):



However, we can simplify this further, by moving the post-selections through the AND gate as before and cancelling redundant negations:



where the equality marked with a star follows from the following derivation:

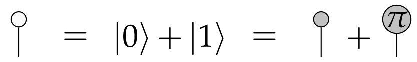


(3.8)

Therefore, in general, CNF formulae can be constructed with three layers: a Z-spider to copy each variable, then a layer of negations for every literal that is *not* negated in the formula, followed by a layer of zero-labelled H-boxes, one for each clause. Thus if we have a kCNF formula with n variables and m clauses, then the corresponding ZH-diagram will contain n Z-spiders and m H-boxes each with at most k legs.

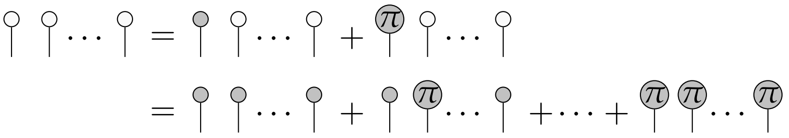
3.2 Counting from Sums of Diagrams

Since we previously defined addition on tensors and tensor networks, it is natural to define addition on diagrams to be the operation induced by addition on the underlying tensors. That is to say, for any three diagrams D_1, D_2, D_3, we will write $D_1 + D_2 = D_3$ if and only if $[\![D_1]\!] + [\![D_2]\!] = [\![D_3]\!]$. Several diagrams, such as Z-spiders, caps, cups, and identity wires are defined as sums of tensors, and so can be written as sums of diagrams. In particular, we have the following equality



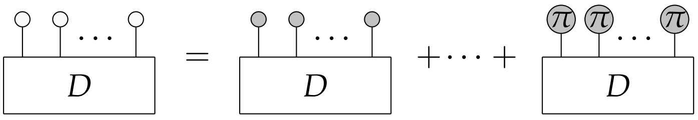
(3.9)

and since tensor addition distributes over the tensor product (as addition distributes over scalar multiplication), we can see that



(3.10)

Thus since the tensor product of Z-spiders is a sum over all possible combinations of X-spiders, and thus all possible combinations of basis states, we have the following, if D is a diagram representing a Boolean formula f:

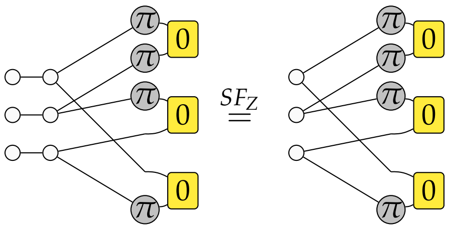


$$= \sum_{x_1,\ldots,x_n\in\{0,1\}} f(x_1,\ldots,x_n) = \sum_{f(x_1,\ldots,x_n)=1} 1 + \sum_{f(x_1,\ldots,x_n)=0} 0$$

$$= |\{x_1,\ldots,x_n \mid f(x_1,\ldots,x_n) = 1\}| \tag{3.11}$$

Therefore, plugging Z-spiders into the top of a diagram for a Boolean formula is a scalar diagram whose value is exactly the number of satisfying solutions, and so evaluating this diagram solves #SAT for that formula.

Returning to our example of a ZH-diagram of a CNF formula from earlier, we see that the equivalent #SAT diagram (ignoring scalar factors) is

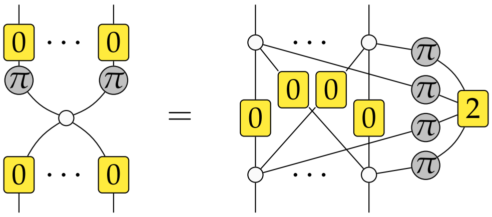


and so in general, by the SF_Z rule, the #SAT diagram for any CNF formula is the same diagram as given previously, but with the inputs to the Z-spiders removed.

The question remains, now that we have diagrams representing a #SAT instances, can we evaluate it? Since the ZH-calculus is complete, there must exist some series of rewrites which transforms such a diagram into a diagram containing just a scalar, from which we can read off the answer. However, the normal form given in the proof completeness represents every individual element of the tensor in the diagram - for a ZH-diagram with n inputs or outputs, this is 2^n elements, since every index has dimension two. While for scalar diagrams, this would appear not to be a problem, in reality, the normal form is generated by considering the diagram as a series of contractions and tensor products, and the diagrams for each of these intermediate components will have many inputs or outputs, and thus be exponentially sized.

In practice, the size of the intermediate diagrams can be minimized by choosing a good ordering for the sequence of contractions - recent work by Johnnie Gray and Stefanos Kourtis [GK21] shows that this method is highly effective for certain problems, albeit with sophisticated heuristics for determining the ordering. We will not consider efficient tensor network contraction in this dissertation - instead, we will consider diagram rewriting that maintains the polynomial size of the diagrams, and ways to evaluate these diagrams as sums of simpler diagrams.

One such rewriting rule is given by de Beaudrap et al [dKM21, Theorem 4.1] for the case of #2SAT diagrams, and allows Z-spiders to be eliminated from the diagram:



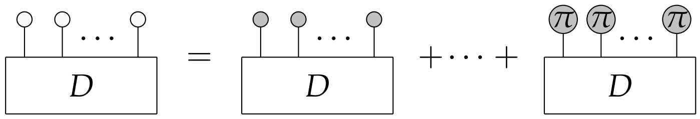
(3.12)

Unfortunately, the presence of the two-labelled H-box prevents this rule from being applied repeatedly to remove all Z-spiders from the diagram. This is to be expected: removing all Z-spiders from the diagram with this rule would provide a polynomial time algorithm for #2SAT since n applications of the rewrite would be required, and each application can be done in $O(n^2)$ elementary operations on the diagram (e.g adding or removing edges, spiders, and H-boxes) since each Z-spider is connected to at most n other Z-spiders through H-boxes, and thus $O(n^2)$ H-boxes would need to be added to connect them.

3.3 Satisfiability from Changing Rings

Since determining whether a Boolean formula is satisfiable is equivalent to determining whether the number of satisfying solutions is non-zero, it is natural to ask whether there is any way to determine if the value of a #SAT diagram given above is non-zero, without explicitly evaluating it. De Beaudrap et al give one method to do this [dKM21, Section 3], by changing the ring over which tensors are defined.

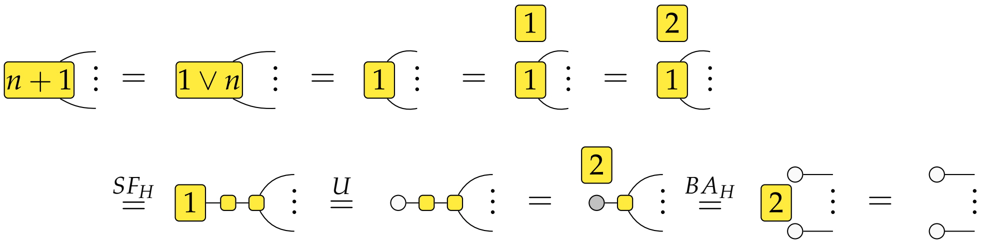
Consider the semiring $\mathcal{R}$ given by $\{0,1\}$ where $a+b = a \vee b$ and $ab = a \wedge b$. Let us consider ZH-diagrams as being defined over $\mathcal{R}$ instead of $\mathbb{C}$, then we can consider sums of scalar diagrams as sums in $\mathcal{R}$. If we have a ZH-diagram D over $\mathcal{R}$ corresponding to a Boolean formula f, we can see that:



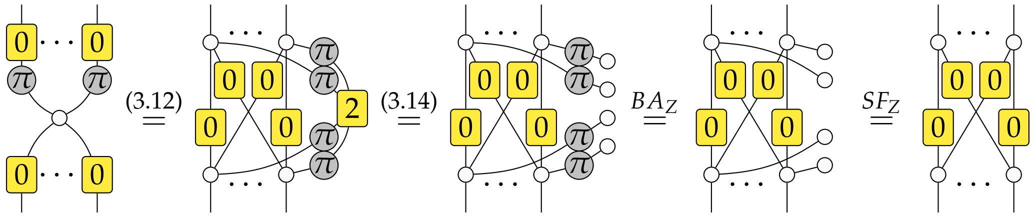
$$= \sum_{x_1,\ldots,x_n \in \{0,1\}} f(x_1,\ldots,x_n) = \bigvee_{x_1,\ldots,x_n \in \{0,1\}} f(x_1,\ldots,x_n) \qquad (3.13)$$

$$= \begin{cases} 1 & f \text{ is satisfiable} \\ 0 & f \text{ is not satisfiable} \end{cases}$$

In terms of diagrams, moving from $\mathbb{C}$ to $\mathcal{R}$ allows H-boxes with positive integer labels to be simplified into Z-spiders:

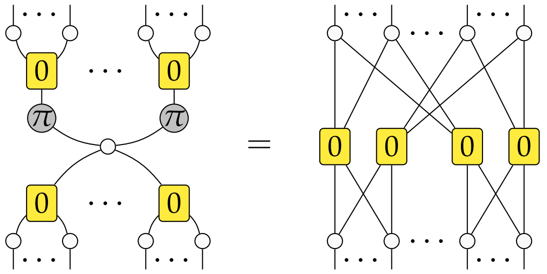


By applying this to Equation (3.12), we find that over $\mathcal{R}$ we have the following:



(3.15)

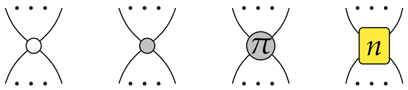
With the obstruction of the two-labelled H-box removed, this gives an algorithm for evaluating diagrams representing 2SAT instances in polynomial time - specifically $O(n^3)$, which matches the existing algorithms for 2SAT. This result can be generalized to kSAT as



(3.16)

but in this case, we lose the polynomial time evaluation property, since the arity of the H-boxes is increasing at each step and we can no longer assume that the diagram is polynomial in size. This was expected since it is generally suspected that 3SAT $\notin P$.

Some care must be taken when translating a diagram from $\mathbb{C}$ to $\mathcal{R}$ - only elements of the diagram whose underlying tensors have non-negative integer values are well-defined in $\mathcal{R}$, so even if the diagram as a whole represents a non-negative integer, this transformation is only valid if every generator of the diagram is one of the following (called the NatZH fragment by de Beaudrap et al [dKM21, Definition 3.1]),

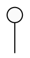
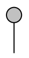


(3.17)

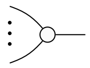
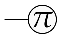
where $n \in \mathbb{Z} \geq 0$. Furthermore, since $\mathcal{R}$ is a semiring and not a ring, the scalar negative one cannot be represented in $\mathcal{R}$, so care must be taken when considering sums of diagrams, as subtraction of diagrams is not well-defined - it is only valid to write a diagram as a sum of diagrams in $\mathcal{R}$ if both summands are valid in $\mathcal{R}$, and the coefficients of the sum are non-negative.

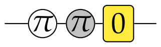
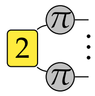
When these conditions are not met, switching from $\mathbb{C}$ to $\mathcal{R}$ is not sound. For example, consider the following fallacious argument that 3SAT $\in$ P:

1. Instead of representing true and false with the computational basis states, use the following mapping:

    $\Longleftrightarrow$ False  $\Longleftrightarrow$ True

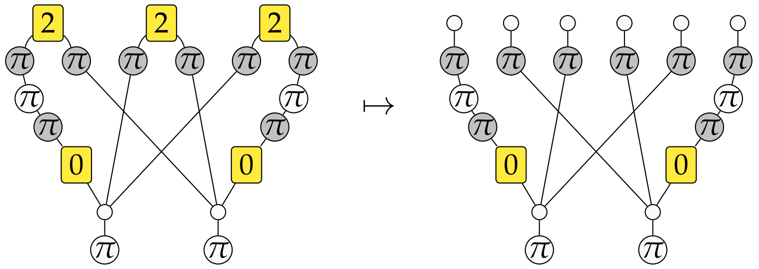
2. In this alternate basis, the OR gate and postselection look like Z-spiders, while the NOT gate is built from a zero H-box, and variables are given by two-labelled H-boxes and X-spiders:

    $\Longleftrightarrow$ OR  $\Longleftrightarrow$ Post-select

    $\Longleftrightarrow$ NOT  $\Longleftrightarrow$ Variable

   All these can be verified by concrete calculation. Note here that the output of the OR gate is weighted by the number of true values, but since we only intend to consider satisfiability, this is okay.

3. Therefore, in this alternate basis, diagrams for #3SAT look like the following, given for the function $f(x_1, x_2, x_3) = (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee \neg x_3)$, which splits completely when moving from $\mathbb{C}$ to $\mathcal{R}$:

4. Since the diagram splits completely along the variables, we are left with one disconnected component for each clause, each of bounded size. We can evaluate these by concrete calculation in $O(1)$ time, since they are bounded, and this whole process takes $O(n^3)$ time because we have at most $\frac{8n(n-1)(n-2)}{6}$ clauses - otherwise there would be duplicates that we could remove before translating the formula to a diagram.

Here the only invalid step was step three, where we moved from $\mathbb{C}$ to $\mathcal{R}$, even though the Z-spiders in the NOT gates and post-selections contained negative values.

CHAPTER 4

Completeness and Universality

This chapter shows an analogue of the "well-known", but surprisingly hard to reference, fact that tensor contraction is in some cases #P-complete. This particular presentation in terms of ZH-diagrams is, as far as the author is aware, novel.

So far, we have seen that #SAT instances can be embedded in ZH-diagrams, and so evaluating scalar ZH-diagrams must be at least as hard as solving #SAT. Is it the case that the converse is also true - that solving #SAT is at least as hard as evaluating ZH-diagrams? This question has been previously considered for general tensor networks [DHM02], where it has been shown that this depends on the generators of the tensor network and the ring they are considered over.

In this chapter, we will answer this question in the affirmative for certain subsets of ZH-diagrams. We will proceed by starting with a small fragment of ZH-diagrams and showing that these are $\mathrm{FP}^{\#\mathrm{P}}$-complete, before showing that there are reductions from larger classes of diagrams into this form, and ending with an approximation of general ZH-diagrams over $\mathbb{C}$.

4.1 Completeness for ZH_{π}^{0}

In order to consider the problem of evaluating ZH diagrams formally, we first define the problem $\mathrm{Eval}(F)$ which is the task of finding the complex number corresponding to a scalar diagram that exists in fragment F of a graphical calculus. A fragment is a set of diagrams built from arbitrary combinations of a fixed subset of generators - for example, the NatZH fragment we encountered previously.

Definition 4.1. *For a given fragment F, the problem* $\mathrm{Eval}(F)$ *is defined as follows:*

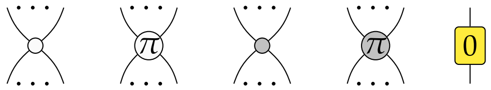
> **Input** *A scalar diagram $D \in F$ consisting of n generators and wires in total, where any parameters of the generators of D can be expressed in $O(\mathrm{poly}(n))$ bits.*

Output *A number $z \in \mathbb{C}$ such that $[\![D]\!] = z$.*

The runtime of an algorithm for Eval(F) *is defined in terms of the parameter n.*

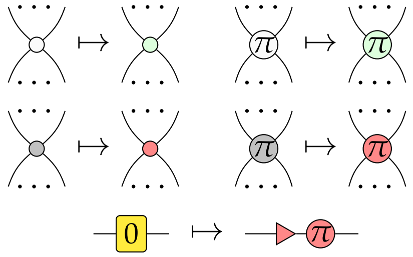
The first fragment we will consider is ZH_π^0, which is a slight generalization of the #SAT diagrams given in the last chapter. This fragment is slightly bigger than NatZH because it allows negativity.

Definition 4.2. *The fragment ZH_π^0 is the subset of ZH-diagrams consisting of the following generators connected by arbitrary wires:*



(4.1)

It is interesting to note that ZH_π^0 is equivalent (up to scalar factors) to the $\Delta\mathrm{ZX}_\pi$ fragment of the ΔZX calculus, which is a universal, sound, and complete, extension of ZX-calculus first given by Renaud Vilmart [Vil19] shortly before the introduction of ZH-calculus. In particular, we can embed ZH_π^0 into $\Delta\mathrm{ZX}_\pi$ as follows:



This is significant because several of the lemmas we will use later are either direct translations or generalizations of the rules of ΔZX. Other presentations of ZX-calculus, for instance, the work of Ng and Wang [NW18], have the triangle above as a derived generator, and some completeness results for general ZX-calculus [JPV19] require a rule that is derived from an equation on triangles, and thus zero H-boxes.

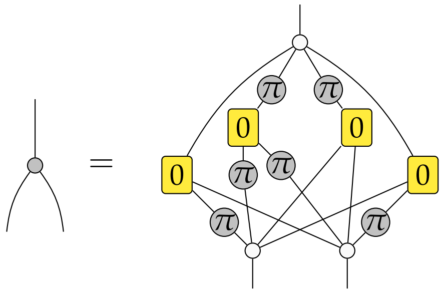
Theorem 4.1. Eval(ZH_π^0) *is #P-hard.*

Proof. This essentially follows directly from the construction of #SAT diagrams by de Beudrap et al [dKM21] as detailed in the previous chapter.

To reduce #2SAT to Eval(ZH_π^0), take a 2CNF formula and construct the associated #SAT ZH-diagram. Since the formula is 2CNF, each zero H-box representing a clause has at most two legs. If there is an H-box with no legs, then return zero, since this H-box multiplies the whole diagram by the scalar zero. For every zero H-box with one leg, apply Equation (3.8) to transform it into an X-spider.

The remaining generators are zero H-boxes with two legs, Z-spiders for variables and X-spiders for negations, all of which are in ZH^0_π. Therefore, the diagram is now in ZH^0_π and the oracle for $\mathrm{Eval}(\mathrm{ZH}^0_\pi)$ can be applied to give the answer. Since #2SAT is #P-complete, $\mathrm{Eval}(\mathrm{ZH}^0_\pi)$ is #P-hard by Lemma 2.1. □

Lemma 4.1. *The following diagram equivalence holds:*



(4.2)

This is derived from the Tseytin transformation [Tse83] of the XOR operation (which is a standard method of translating boolean formulae into 3CNF).

Proof. This can be verified by concrete calculation of the underlying tensors. □

Theorem 4.2. $\mathrm{Eval}(\mathrm{ZH}^0_\pi)$ *is* $\mathrm{FP}^{\#\mathrm{P}}$*-complete*

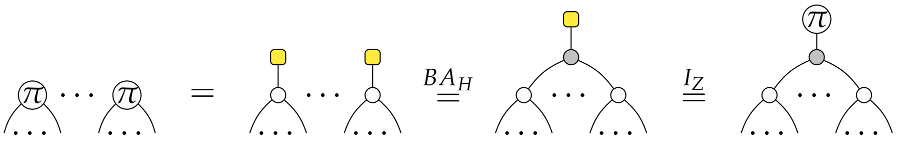
Proof. Since $\mathrm{Eval}(\mathrm{ZH}^0_\pi)$ is #P-hard and #3SAT is #P-complete, it suffices to prove that there is a Cook reduction from $\mathrm{Eval}(\mathrm{ZH}^0_\pi)$ to #3SAT by Lemma 2.1.

First, note that for simplicity this reduction is not tight - it can be made much tighter with the addition of many case distinctions and several extra rewrite rules.

We can reduce $\mathrm{Eval}(\mathrm{ZH}^0_\pi)$ to #3SAT as follows. Given a scalar diagram D in ZH^0_π, proceed as follows:

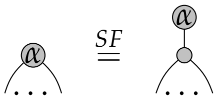
1. Any spiders or H-boxes with no legs should be removed from the diagram. Evaluate them by concrete calculation and multiply their values together to get a scalar multiplier c for the diagram. If there are no such spiders, set $c = 1$.

2. Extract the phases from all π-phase Z-spiders as follows:



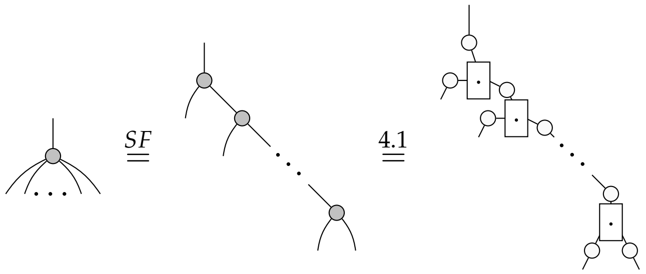
(4.3)

3. Unfuse the phase of every X-spider with at least two legs:



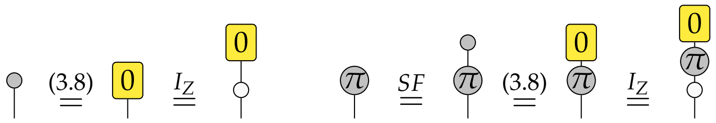
(4.4)

4. For any X-spiders with at least three legs, split them and apply Lemma 4.1, fusing Z-spiders between applications:



(4.5)

5. Replace X-spiders with one leg as follows:



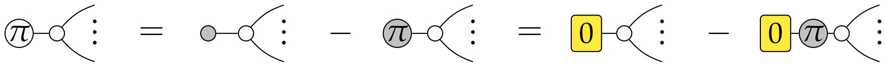
(4.6)

6. Excepting the single Z-spider with a π-phase, use the SF_Z rule to fuse Z-spiders wherever possible. If there are two two-legged zero H-boxes connected together directly, introduce a Z-spider between them using the I_Z rule.

At this point, we can argue that this diagram follows the form of a #3SAT diagram except perhaps for a single π-phase Z-spider:

- Every Z-spider has a zero phase due to step two, is not connected to other Z-spiders due to step six, and is only connected to X-spiders that are NOT gates by steps four and five.
- Every H-box has label zero, is only connected to X-spiders that are NOT gates by steps four and five, and is not connected to other H-boxes directly by step six.
- Every X-spider is a NOT gate: after step three only X-spiders with one leg or at least three legs remain. The first case is eliminated in step five, and the second in step four.

Finally, to remove the π-phase Z-spider introduced in step two, we can write the diagram as a sum of two diagrams which don't contain this phase:



(4.7)

Since these two diagrams are #3SAT diagrams associated with some Boolean functions f_1 and f_2, read these functions off the diagram by converting each Z-spider into a variable and each H-box into a clause. Then, using the #3SAT oracle, find the number of solutions z_1 and z_2 of f_1 and f_2. The value of the original ZH_{π}^{0} diagram D is then given by $z = c(z_1 - z_2)$.

Since there are at most n each of generators and wires, it is easy to see that this runs in polynomial time in n: steps one, two, three, and five run in time $O(n)$ since they perform a constant amount of work per generator. Step four runs in time $O(n)$ since it performs a constant amount of work per edge. Step six runs in time $O(n)$ since it does a constant amount of work per generator and $O(n)$ of them were introduced in step four. Therefore, the reduction above is a Cook reduction, and $\mathrm{Eval}(\mathrm{ZH}_{\pi}^{0})$ is $\mathrm{FP}^{\#\mathrm{P}}$-complete. □

It is interesting to note that this reduction requires the use of #3SAT despite the fact that #2SAT is also #P-complete. The only generators that are blocking this are X-spiders and Z-spiders with π-phases. We saw from step two that it is possible to eliminate the π-phases in exchange for X-spiders, and we will see in the next section that it is possible to eliminate the X-spiders in exchange for π-phases, but it is not possible to eliminate both.

Therefore, if we wanted to find a reduction from $\mathrm{Eval}(\mathrm{ZH}_{\pi}^{0})$ to #2SAT, perhaps we could find a way to write an X-spider as a #2SAT diagram? Unfortunately, this is not possible unless $P = NP$ - it is known that Boolean formulae that have both 2CNF and XOR clauses are equivalent to 3CNF formulae, and X-spiders are equivalent to XOR clauses, so we could solve 3SAT in polynomial time if we could rewrite X-spiders in #2SAT form by switching the underlying ring from $\mathbb{C}$ to $\mathcal{R}$ as shown before.

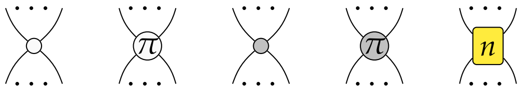
Another way we could find a reduction to #2SAT would be to write X-spiders as sums of polynomial many #2SAT diagrams. However, if this is possible, unless $P = NP$, at least one of the coefficients of the sum must be negative - otherwise we can still make the switch from $\mathbb{C}$ to $\mathcal{R}$ and solve 3SAT by evaluating the disjunction of 2SAT instances. Therefore, it seems unlike that a reduction of this type is possible, and indeed this makes sense given that the original proof of $\#P$-completeness for #2SAT by Valiant [Val79] exploits a multiplicative, not additive, structure.

4.2 Negativity is All You Need

Now that we have one fragment of ZH-calculus which is $\mathrm{FP}^{\#\mathrm{P}}$-complete, we can move to larger fragments. We will examine four fragments $\mathrm{ZH}_{\pi/4}^{\mathbb{Q}} \supseteq \mathrm{ZH}_{\pi/2}^{\mathbb{Q}} \supseteq \mathrm{ZH}_{\pi}^{\mathbb{Q}} \supseteq \mathrm{ZH}_{\pi}^{\mathbb{Z}} \supseteq \mathrm{ZH}_{\pi}^{0}$ and show that they can all be reduced to ZH_{π}^{0} and thus are all $\mathrm{FP}^{\#\mathrm{P}}$-complete. Far from being purely academic, the largest of these fragments,

$\mathrm{ZH}^{\mathbb{Q}}_{\pi/4}$, is large enough for many practical uses of ZH-calculus including expressing quantum computations ergonomically, which is what ZH-calculus was originally designed for.

Definition 4.3. $\mathrm{ZH}^{\mathbb{Z}}_{\pi}$ *is the fragment of ZH-calculus given by the following generators*

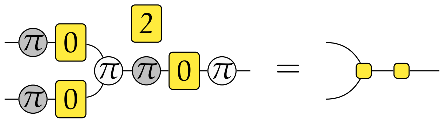


(4.8)

where $n \in \mathbb{Z}$.

By far the most difficult reduction is the first one, from $\mathrm{ZH}^{\mathbb{Z}}_{\pi}$ to ZH^{0}_{π}, as it requires a method of encoding countably many generators. In order to establish this, first, we will show that arbitrarily-sized AND gates can be implemented in ZH^{0}_{π} if we use negative numbers in our tensors in the form of π-phase Z-spiders.

Lemma 4.2. *The following diagram equivalence holds:*

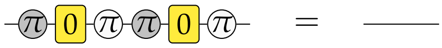


(4.9)

This is translated from a characterization of the Toffoli gate first given by Ng and Wang [NW18] and is a generalization of the $\Delta\mathrm{ZX}_{\pi}$ *rules* HT *and* BW.

Proof. This can be verified by concrete calculation of the tensors. □

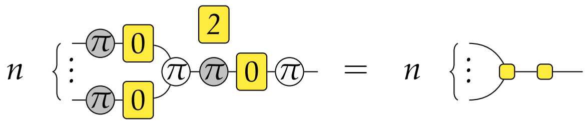
Lemma 4.3. *The following diagram equivalence holds:*



(4.10)

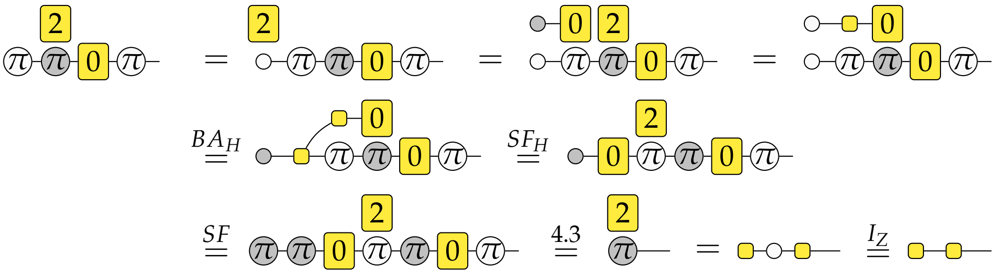
Proof. This can be verified by concrete calculation of the tensors. □

Lemma 4.4. *The following diagram equivalence holds for all* $n \geq 0$*:*



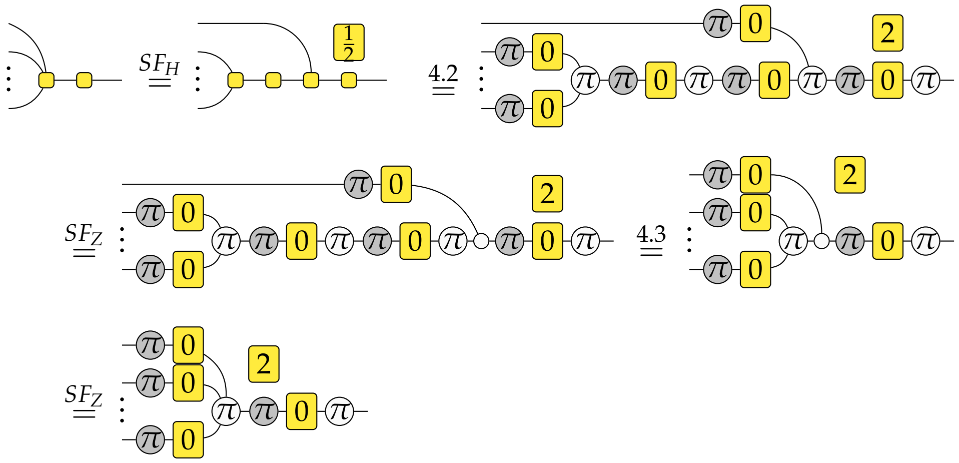
(4.11)

Proof. We proceed by induction on n. For the case of $n = 0$:



(4.12)

For the case of $n = k+1$, assuming the result holds for k:



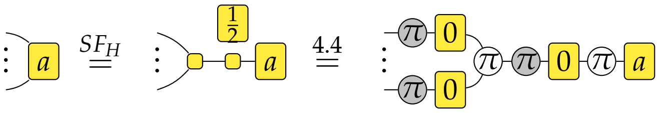
(4.13)

□

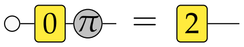
The previous lemmas represent the most technical part of the reduction. With these cases handled, we can show the whole reduction to ZH_π^0 by constructing all integers labelled H-boxes from zero H-boxes (this was inspired by the method in [BKM$^+$21], but uses a different construction, as their method requires exponentially sized diagrams). We will also explore several other applications of these lemmas in the next chapter.

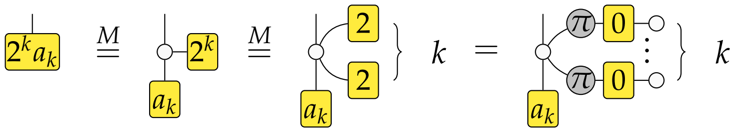
Theorem 4.3. *There is a polynomial-time counting reduction from the problem* $\mathrm{Eval}(\mathrm{ZH}_\pi^{\mathbb{Z}})$ *to* $\mathrm{Eval}(\mathrm{ZH}_\pi^0)$ *and we have that* $\mathrm{Eval}(\mathrm{ZH}_\pi^{\mathbb{Z}})$ *is* $\mathrm{FP}^{\#\mathrm{P}}$*-complete.*

Proof. In order to rewrite a diagram from $\mathrm{ZH}_\pi^{\mathbb{Z}}$ into ZH_π^0, we only need to rewrite all of the H-boxes into zero H-boxes with two legs. First, apply H-box fusion and Lemma 4.4 to rewrite all incompatible H-boxes into H-boxes with one leg:



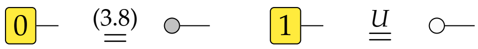
(4.14)

Then to construct a one-legged H-box with label $a \geq 0$, write $a = a_0 + 2^1 a_1 + 2^2 a_2 + \cdots + 2^m a_m$ where $m = O(\mathrm{poly}(n))$. This is always possible since by the definition of $\mathrm{Eval}(\mathrm{ZH}_\pi^{\mathbb{Z}})$, all the labels must have at most $O(\mathrm{poly}(n))$ bits. Then since  we have that



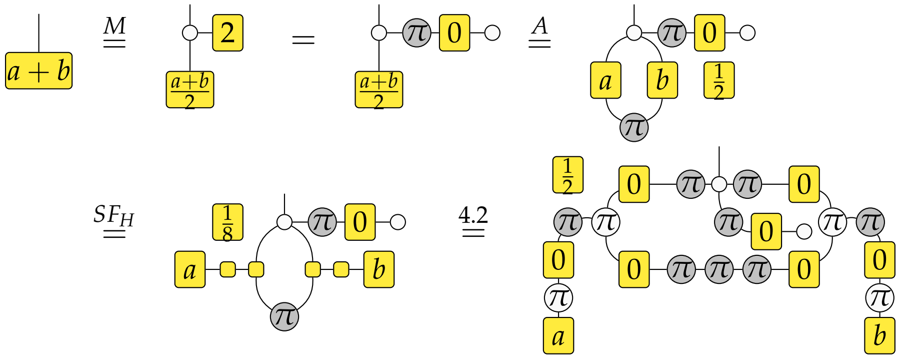
(4.15)

and as $a_k \in \{0,1\}$, we can rewrite a_k in ZH_π^0 as



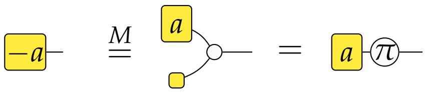
(4.16)

and then we can combine these together to make an a-labelled H-box using an addition gadget:



(4.17)

Finally, if we need to construct an $a < 0$ labelled H-box, we can use the following:



(4.18)

We can see this rewrite runs in polynomial time because there are $O(n)$ H-boxes, each of which requires $O(m)$ addition gadgets, and each term of the addition requires $O(m)$ spiders and H-boxes since we need k two-labelled H-boxes to represent $2^k a_k$, and $k \leq n$. Therefore, the following is a polynomial-time counting reduction from $\text{Eval}(\text{ZH}_\pi^\mathbb{Z})$ to $\text{Eval}(\text{ZH}_\pi^0)$:

- Apply this rewrite, after which we will be left with a ZH_π^0 diagram with some additional scalar H-boxes of $\frac{1}{2}$.

- Remove these from the diagram and let c be their product.

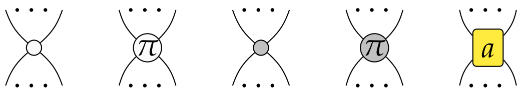
- Submit the resulting diagram to the oracle for $\text{Eval}(\text{ZH}_\pi^0)$ to obtain a value z, and then return cz.

Furthermore, by Lemma 2.1 and Theorem 4.2, $\text{Eval}(\text{ZH}_\pi^\mathbb{Z})$ is $\text{FP}^{\#\text{P}}$-complete. □

4.3 Injecting Magic States

Now we can move on to considering even larger fragments fairly easily. Firstly, we will show that incorporating rational numbers in H-box labels can be done by rewriting up to some scalar factors, and then by using gadgets to copy certain "magic states" - that is, one-legged H-boxes with specific labels which we can split as sums - we will introduce complex numbers and factors of $\sqrt{2}$ into the labels.

Definition 4.4. $\mathrm{ZH}_{\pi}^{\mathbb{Q}}$ *is the fragment of ZH-calculus given by the following generators*

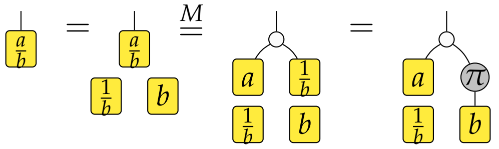


(4.19)

where $a \in \mathbb{Q}$.

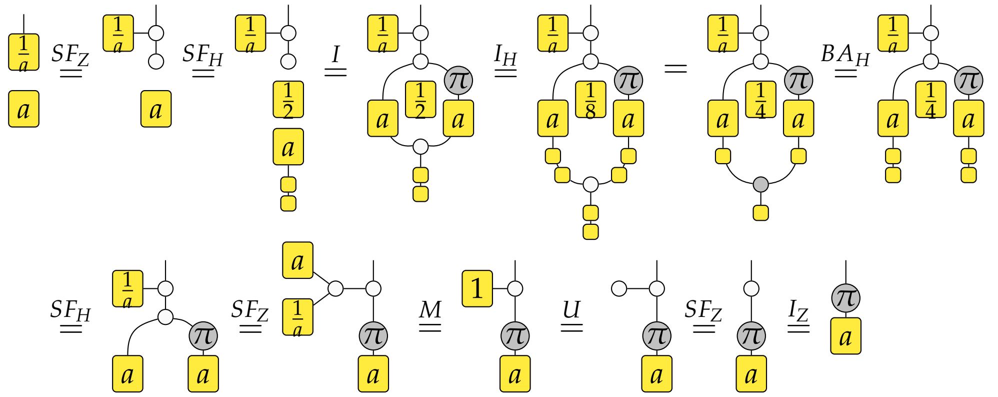
Theorem 4.4. *There is a polynomial-time counting reduction from* $\mathrm{Eval}(\mathrm{ZH}_{\pi}^{\mathbb{Q}})$ *to the problem* $\mathrm{Eval}(\mathrm{ZH}_{\pi}^{\mathbb{Z}})$*, and* $\mathrm{Eval}(\mathrm{ZH}_{\pi}^{\mathbb{Q}})$ *is* $\mathrm{FP}^{\#\mathrm{P}}$*-complete.*

Proof. We can rewrite a diagram from $\mathrm{ZH}_{\pi}^{\mathbb{Q}}$ to $\mathrm{ZH}_{\pi}^{\mathbb{Z}}$ as follows. First, note that only the H-boxes need to be rewritten. Therefore, apply the SF_H rule to unfuse the label from every H-box, as in the first step of the previous reduction, so that we must only consider rewriting H-boxes with one leg. These can be translated as follows



(4.20)

since we have that:



(4.21)

Therefore, a reduction from $\mathrm{Eval}(\mathrm{ZH}_{\pi}^{\mathbb{Q}})$ to $\mathrm{Eval}(\mathrm{ZH}_{\pi}^{\mathbb{Z}})$ is given by:

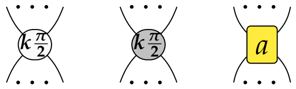
1. Perform the rewrite specified above. This clearly happens in polynomial time since a constant amount of work is performed per generator, of which there are $O(n)$.

2. This rewrite will create additional scalar factors in the diagram, so extract these from the diagram and let c be their product, which can be computed in $O(\mathrm{poly}(n))$ time since there are $O(n)$ of them and they each have $O(\mathrm{poly}(n))$ number of bits.

3. Use the oracle for $\mathrm{Eval}(\mathrm{ZH}_{\pi}^{\mathbb{Z}})$ to evaluate the remaining diagram and obtain a value z, and then return cz.

This is a polynomial-time counting reduction because the oracle is only used once, and since this is also a Cook reduction, $\mathrm{Eval}(\mathrm{ZH}_{\pi}^{\mathbb{Q}})$ is $\mathrm{FP}^{\#\mathrm{P}}$-complete by Lemma 2.1 and Theorem 4.3. □

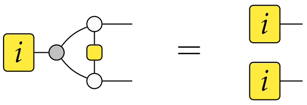
Definition 4.5. $\mathrm{ZH}_{\pi/2}^{\mathbb{Q}}$ *is the fragment of ZH-calculus given by the following generators*



(4.22)

where $k \in \mathbb{Z}$ and $a \in \mathbb{Q}[i]$ – the value of a must have rational real and imaginary parts.

Lemma 4.5. *The following diagram equivalence holds:*

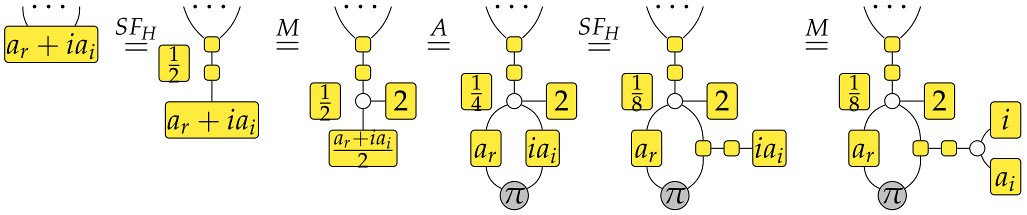


(4.23)

Proof. This can be checked by concrete calculations on the underlying tensors. □

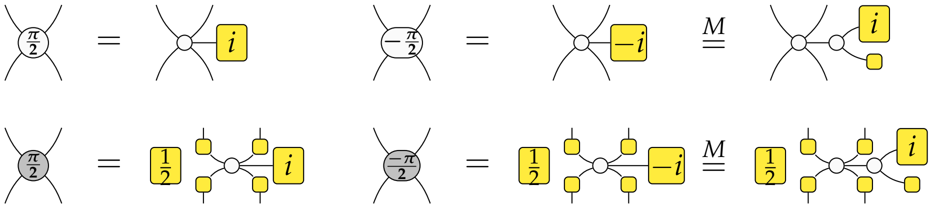
Theorem 4.5. *There is a Cook reduction from* $\mathrm{Eval}(\mathrm{ZH}_{\pi/2}^{\mathbb{Q}})$ *to* $\mathrm{Eval}(\mathrm{ZH}_{\pi}^{\mathbb{Q}})$*, and we have that* $\mathrm{Eval}(\mathrm{ZH}_{\pi/2}^{\mathbb{Q}})$ *is* $\mathrm{FP}^{\#\mathrm{P}}$*-complete.*

Proof. To perform a reduction from $\mathrm{Eval}(\mathrm{ZH}_{\pi/2}^{\mathbb{Q}})$ to $\mathrm{Eval}(\mathrm{ZH}_{\pi}^{\mathbb{Q}})$, we will first normalize the diagram to extract all the imaginary components as seperate spiders. To do this for H-boxes, first write each label, a, as $a = a_r + ia_i$ where $a_r, a_i \in \mathbb{Q}$, and then apply the following:



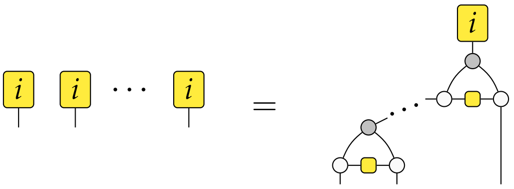
(4.24)

For each spider, if k is even, we do not need to do anything, since then it is in $\mathrm{ZH}_{\pi}^{\mathbb{Q}}$, otherwise, if k is odd, we do the following:



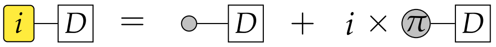
(4.25)

Then we fold up the i-labelled H-boxes into a single H-box by making use of Lemma 4.5:



(4.26)

At this point we have a $\mathrm{ZH}_{\pi}^{\mathbb{Q}}$ diagram D connected to a single i-labelled, so we can split it as a sum:

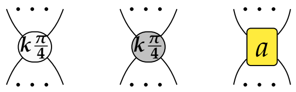


(4.27)

Then we can use the $\mathrm{Eval}(\mathrm{ZH}_{\pi}^{\mathbb{Q}})$ oracle to evaluate these two diagrams to obtain z_1 and z_2, and return $z_1 + iz_2$. This whole reduction runs in polynomial time since the rewriting does a constant amount of work per generator. Therefore, by Lemma 2.1 and Theorem 4.4, $\mathrm{Eval}(\mathrm{ZH}_{\pi/2}^{\mathbb{Q}})$ is $\mathrm{FP}^{\#\mathrm{P}}$-complete. □

Now we are finally ready to move on to the largest fragment, which contains all H-boxes with rational numbers, i and $\sqrt{2}$ as labels. The method of proof will be very similar to the previous case and proceeds by introducing another magic state.

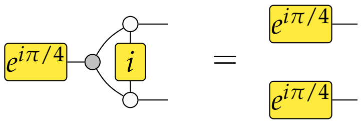
Definition 4.6. $\mathrm{ZH}_{\pi/4}^{\mathbb{Q}}$ *is the fragment of ZH-calculus given by the following generators*



(4.28)

where $k \in \mathbb{Z}$ and $a \in \mathbb{Q}[i, \sqrt{2}]$ – the value of a must be a linear combination of 1, i, and $\sqrt{2}$ with rational coefficients.

Lemma 4.6. *The following diagram equivalence holds:*



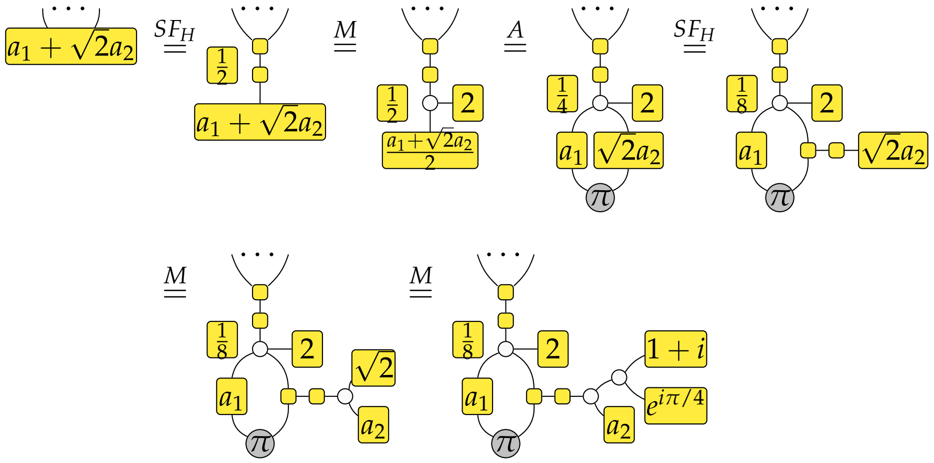
(4.29)

Proof. This can be checked by concrete calculations on the underlying tensors. □

Theorem 4.6. *There is a Cook reduction from* $\mathrm{Eval}(\mathrm{ZH}_{\pi/2}^{\mathbb{Q}})$ *to* $\mathrm{Eval}(\mathrm{ZH}_{\pi/4}^{\mathbb{Q}})$*, and we have that* $\mathrm{Eval}(\mathrm{ZH}_{\pi/4}^{\mathbb{Q}})$ *is* $\mathrm{FP}^{\#\mathrm{P}}$*-complete.*

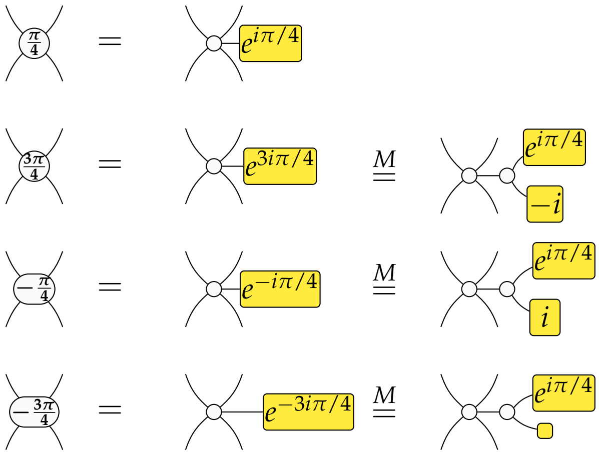
Proof. To perform a reduction from $\mathrm{Eval}(\mathrm{ZH}_{\pi/4}^{\mathbb{Q}})$ to $\mathrm{Eval}(\mathrm{ZH}_{\pi/2}^{\mathbb{Q}})$, we will first normalize the diagram to extract all the $\sqrt{2}$ components as seperate $e^{i\frac{\pi}{4}}$-labelled H-boxes. To do this for H-boxes, first write each label, a, as $a = a_1 + \sqrt{2}a_2$ where

$a_1, a_2 \in \mathbb{Q}[i]$, and then apply the following:



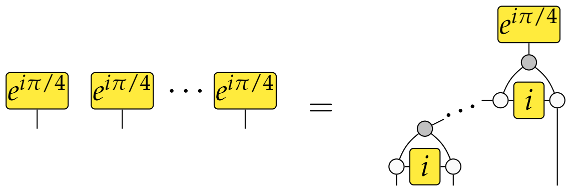
(4.30)

For X-spiders, apply the definition to translate them into Z-spiders, and for Z-spiders, if k is even, we do not need to do anything, since then it is in $\mathrm{ZH}_{\pi}^{\mathbb{Q}}$, otherwise, if k is odd, we do the following:



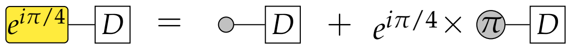
(4.31)

Then we fold up the $e^{i\frac{\pi}{4}}$-labelled H-boxes into a single H-box by making use of Lemma 4.6:



(4.32)

At this point we have a $\mathrm{ZH}_{\pi/2}^{\mathbb{Q}}$ diagram D connected to a single $e^{i\frac{\pi}{4}}$-labelled H-box, so we can split it as a sum:



(4.33)

Then we can use the $\mathrm{Eval}(\mathrm{ZH}_{\pi/2}^{\mathbb{Q}})$ oracle to evaluate these two diagrams to obtain z_1 and z_2, and return $z_1 + e^{i\frac{\pi}{4}} z_2$. This whole reduction runs in polynomial time since the rewriting does a constant amount of work per generator. Therefore, by Lemma 2.1 and Theorem 4.5, $\mathrm{Eval}(\mathrm{ZH}_{\pi/4}^{\mathbb{Q}})$ is $\mathrm{FP}^{\#\mathrm{P}}$-complete. □

Therefore, by a chain of reductions, we have shown that $\mathrm{ZH}_{\pi}^{\mathbb{Z}}$, $\mathrm{ZH}_{\pi}^{\mathbb{Q}}$, $\mathrm{ZH}_{\pi/2}^{\mathbb{Q}}$, and $\mathrm{ZH}_{\pi/4}^{\mathbb{Q}}$ are all $\mathrm{FP}^{\#\mathrm{P}}$-complete and diagrams in these fragments can be written as a sum of at most eight $\#3SAT$ instances, since we introduce two magic states with two terms each, and the original reduction from ZH_{π}^{0} to #3SAT requires two terms.

While we could continue this process for larger and larger fragments $\mathrm{ZH}_{\pi/2^k}^{\mathbb{Q}}$ for $k \geq 2$, which allows for arbitrarily small angles in Z-spiders and H-box labels, or indeed show completeness for all fixed k by induction, this technique cannot be used to show completeness for the full ZH-calculus. Note that we can approximate any ZH-diagram with a diagram in $\mathrm{ZH}_{\pi/2}^{\mathbb{Q}}$, since the rationals are dense in the reals, but we will not attempt to formalize this here as the error analysis required to bound the size of the approximation is difficult.

CHAPTER 5

Backtracking Algorithms for #SAT

The first and second parts of this chapter give an exposition of existing work on #SAT *algorithms, but are presented in terms of diagrams. The third part gives a novel extension to the DPLL algorithm to handle larger clauses and uses it to obtain new upper bounds on the runtime for low-density instances.*

So far, we have seen that some ZH-diagrams are equivalent to #SAT instances and that some other diagrams can be rewritten into this form. Given this, one natural question is can the process of solving a #SAT instance be kept in diagrammatic form, and if so, what does it look like?

In this chapter, we will provide an overview of the DPLL algorithm [DLL62] for solving #SAT and SAT problems, and show how it corresponds to operations on diagrams. We will then describe some methods of obtaining upper bounds on the running time of this algorithm, including an exposition of the bound obtained by Dahllöf et al [DJW02a] for #2SAT, and measure how the running time depends on the density of the input formulae. Finally, we apply our completeness results from the last chapter to introduce an extension to the algorithm to solve #kSAT independent of k for formulae with low density. In particular, we find a runtime of less than $O(\text{poly}(n) \cdot 2^n)$ for formulae with $\delta < 2.25032$, beating the worst-case bounds of Dubois [Dub91] whenever $\delta < 1.858$ and $k \geq 4$, and the expected-time bounds for #kSAT by Williams [Wil04] for $k \geq 6$ whenever $\delta < 1.217$.

5.1 Interpreting DPLL Diagrammatically

The Davis-Putnam-Logemann-Loveland (DPLL) algorithm is an algorithm for solving SAT that was first introduced in 1962 [DLL62], in the context of automated theorem proving. It is essentially an optimized depth-first search over all possible assignments of variables in a Boolean formula. The algorithm is based on the following three rules:

Unit Propogation: The following rewrite holds for any clauses A_i and B_i not containing some literal x:

$$
\begin{aligned}
x \land (A_1 \lor x) \land \cdots \land (A_n \lor x) \land (B_1 \lor \neg x) \land \cdots \land (B_m \lor \neg x) & \\
= B_1 \land \cdots \land B_n &
\end{aligned}
\tag{5.1}
$$

This can be applied to a formula when it contains a clause of only one literal x, and it removes all clauses which will become true because x must be true and removes x from all clauses in which it is required to be false. Note that often, the application of this rule will create more clauses of size one, causing a chain of cancellations.

Pure Literal Elimination: If there is a formula g such that every clause contains a literal x

$$g = (A_1 \lor x) \land \cdots \land (A_n \lor x) \tag{5.2}$$

then g is satisfiable. Therefore, since we only care about satisfiability, if we have a formula f such that $f = f' \land g$ where f' doesn't contain x or $\neg x$, then we can replace f with f'.

Variable Branching: For any variable x appearing in a formula f, f is only unsatisfiable if both of the following formulae are unsatisfiable:

$$f_1 = f \land x \qquad f_2 = f \land \neg x \tag{5.3}$$

Suppose f is satisfiable, then for any satisfying assignment of f, either x must be true, in which case f_1 is satisfiable, or x must be false, so f_2 is satisfiable. If f is unsatisfiable, then clearly neither f_1 or f_2 can be satisfiable, since any satisfying assignment would also satisfy f.

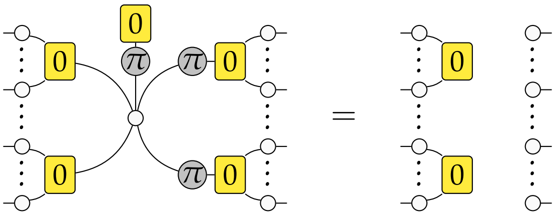
With these three observations, we can define the full algorithm $DPLL(f)$:

1. If f contains the clauses x and $\neg x$ for some variable x, then f is unsatisfiable.
2. If f has no clauses remaining then f is satisfiable.
3. Apply unit propagation and pure literal elimination to f until it is no longer possible.
4. Pick a variable x that occurs in f and apply variable branching to generate f_1 and f_2.
5. Return $DPLL(f_1) \lor DPLL(f_2)$, only evaluating $DPLL(f_2)$ if $DPLL(f_1)$ is false.

We can see that this must terminate because when applying variable branching, clauses of one literal are created, which means that in both branches at least one variable will be removed due to unit propagation. Since we start with n variables, this means we can have at most 2^n recursive calls, although in practice it is fewer than this, since the first recursive branch is often satisfiable, and the second one is then not taken.

To interpret DPLL diagrammatically we will first see how the three rules apply to the ZH-diagrams for SAT instances detailed in chapter three. Note that throughout we will be considering diagrams over the semiring $\mathcal{R}$ as before.

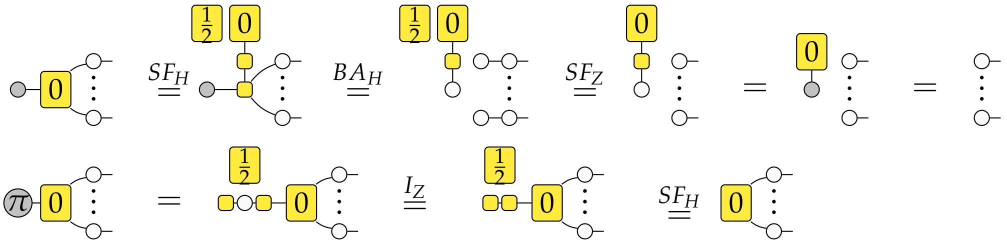
Lemma 5.1. *The following diagrammatic equivalent to the unit propagation rule holds (without loss of generality, we assume the literal to be propagated is not negated):*



(5.4)

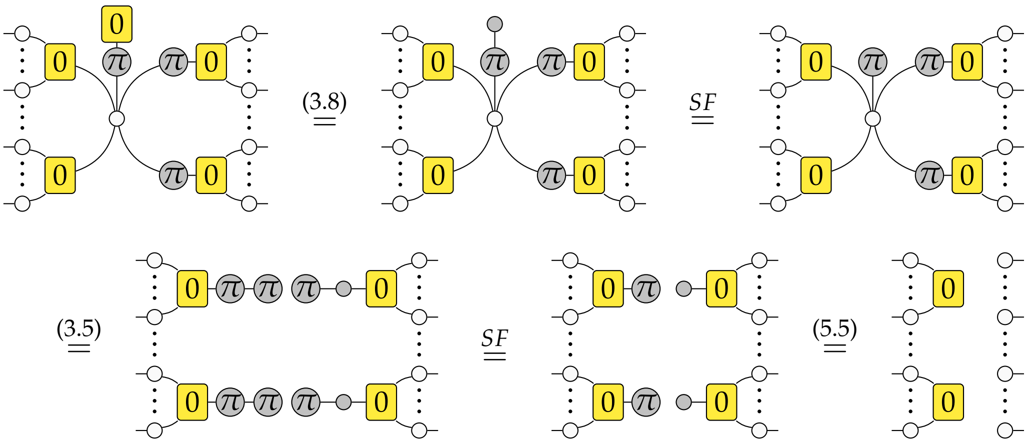
It can be read as follows - on the left-hand side, the zero H-box with one leg is the clause with a single non-negated literal x, the H-boxes on the left represent the clauses $B_i \vee \neg x$ while the H-boxes on the right represent the clauses $A_i \vee x$. On the right-hand side, we see that the clauses B_i remain, while the clauses $A_i \vee x$ have been removed entirely. Because the variable x is now no longer mentioned, the Z-spider representing it is also removed.

Proof. First, note that



(5.5)

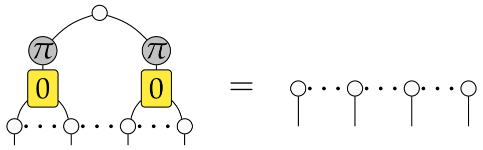
and so we have that



(5.6)

which completes the proof. □

Lemma 5.2. *The following diagrammatic equivalent to the pure literal elimination rule holds (without loss of generality, we assume the pure literal is not negated):*

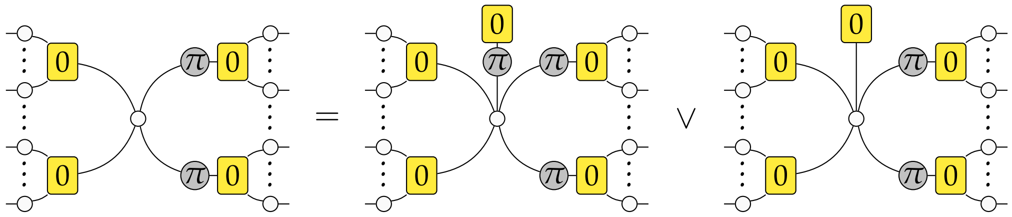


(5.7)

This is read as follows - on the left-hand side, we have a variable that is only present in clauses non-negated, and on the right-hand side, both the variable and the clauses containing it have been eliminated.

Proof. This follows directly from Equation (3.16) in the case where the top or bottom of the diagram is empty. □

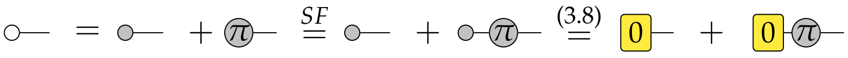
Lemma 5.3. *The following diagrammatic equivalent to the variable branching rule holds:*



(5.8)

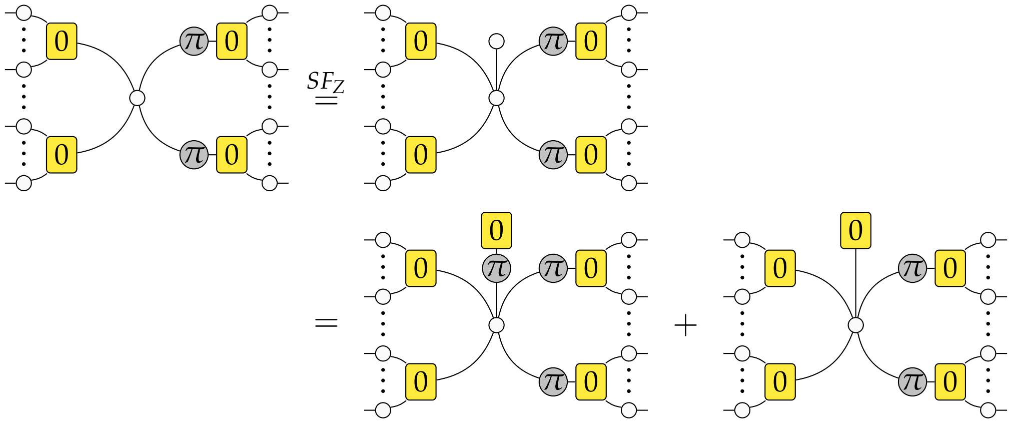
On the left-hand side, we have a variable connected to arbitrary clauses, whereas on the right-hand side we have two terms, each with a clause of one literal introduced onto the variable.

Proof. First, note that we can write a Z-spider by the following sum:



(5.9)

Therefore, we can rewrite any SAT diagram as a sum according to



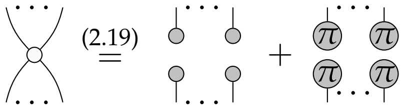
(5.10)

but in $\mathcal{R}$ sums are logical OR operations, so this completes the proof. □

While this shows that we can interpret DPLL diagrammatically as expanding a SAT diagram into a sum of diagrams (that is actually a logical OR) and rewriting each one, this particular formulation is not particularly clean in terms of diagrams. We can reformulate this as an equivalent set of rewrites in terms of propagating post-selections, and applying the definition of the Z-spider:

- Firstly, Equation (3.5) represents the fact that post-selecting on a variable post-selects all the clauses it is connected to, and Equation (5.5) corresponds to post-selecting a clause either making the whole clause true and removing it or removing that literal from the clause. Finally, Equation (3.8) says that a clause with one literal is the same as a post-selection. Taken together, these are equivalent to unit propagation.

- Variable branching is exactly equivalent to the definition of the Z-spider



(5.11)

  which can be interpreted as 'either every instance of a variable is true, or every instance is false', so we have a sum of two terms post-selecting the clauses as such.

- Pure literal elimination is more complicated and does not correspond directly to the action of post-selection, but rather to the observation that when performing variable branching on a pure literal, one branch will be a diagram that is exactly a subset of the other (in terms of clauses), and so the whole sum is satisfiable if that term is also satisfiable.

Note that of all the steps of the DPLL algorithm, only pure literal elimination is invalid when considered for ZH-diagrams over $\mathbb{C}$ instead of $\mathcal{R}$. Indeed, it is even known that #MONOTONE-2SAT, which is #2SAT restricted to instances where every literal is pure, is #P-complete! Clearly, the decision variant of this problem, MONOTONE-2SAT, is not only solvable in polynomial time but solvable in constant time - every instance is satisfiable.

However, the argument that DPLL terminates does not depend on pure literal elimination, so that means that applying this process without pure literal elimination to #SAT diagrams over $\mathbb{C}$ will also terminate and compute the correct result, and allows us to define an analogous algorithm for #SAT, which we will call $\#DPLL(f)$, and was first published as the algorithm CDP by Birnbaum and Lozinskii in 1999 [BL99]:

- If f contains the clauses x and $\neg x$ for some variable x, then f is unsatisfiable, so it has zero satisfying assignments.
- If f contains no variables, then it has one satisfying solution, the empty solution.
- Apply unit propagation to f as much as possible.
- Pick a variable x that occurs in f and apply variable branching to generate f_1 and f_2.
- Return $\#DPLL(f_1) + \#DPLL(f_2)$.

Modifications of this algorithm are used extensively in practice, with some of the best solvers like sharpSAT [Thu06] and Cachet [SBB$^+$04] making use of this technique, and all the best-known theoretical upper bounds on runtime are based on careful analysis of this algorithm. Note that we left the choice of variable to branch on unspecified - choosing this wisely is crucial to obtaining good runtimes, as we will see in the next section.

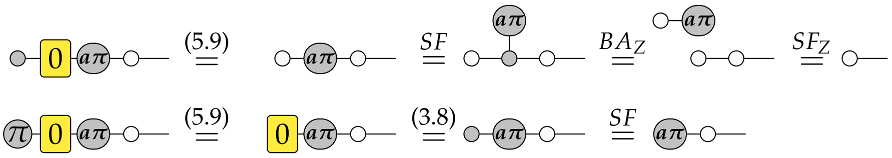
5.2 A Close Look at #2SAT

While practical solvers exist for general #SAT problems, most theoretical results concern either the case of #3SAT or #2SAT. In this dissertation, we will be focusing on #2SAT because it is particularly amenable to the diagrammatic methods we will consider in the next chapter. For this section, we will present in diagrammatic form an algorithm for #2SAT published by Dahllöf et al in 2002 [DJW02a].

5.2.1 Simplifying #2SAT Diagrams

For #2SAT diagrams, we are working with Z-spiders for variables, X-spiders for negation, and zero-labelled H-boxes with at most two legs as clauses (i.e a subset of the fragment of ZH_{π}^{0}). In this setting, we can give some specialized rewrites. We start with the clause post-selection rule, Equation (5.5):

Lemma 5.4. *The following equivalent to Equation* (5.5) *for* #2SAT *holds:*

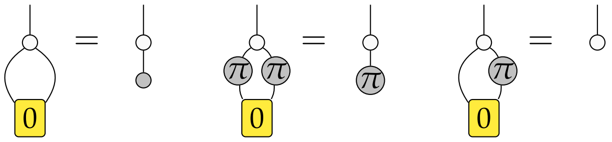


(5.12)

We can see that in one of the cases, a post-selected clause is just removed, and in the other case, it becomes a post-selection on the other variable the clause is connected to.

We can also remove some clauses from the instance. For instance, we have the following rules that allow the rewriting of clauses that connect a variable to itself, or two clauses that connect the same two variables:

Lemma 5.5. *The following rewrite for* #2SAT *holds:*

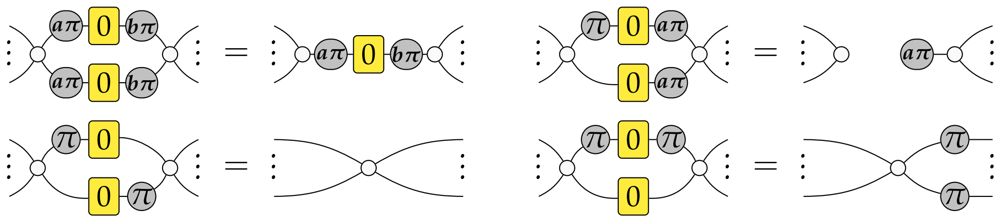


(5.13)

This can be interpreted as saying that $x \vee x = x$, $\neg x \vee \neg x = \neg x$, and that $x \vee \neg x$ is always true, so any such clause can be removed.

Proof. These follow from Lemmas 5.1 and 8.3 in the paper of Backens et al [BKM+21]. □

Lemma 5.6. *The following rewrite for* #2SAT *holds:*



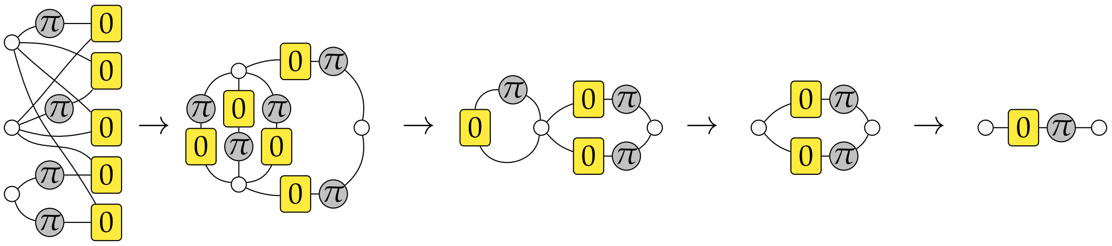
(5.14)

This lemma provides a characterization of all possible Boolean functions on two variables and allows pairs of clauses on the same variables to be replaced with the deletion of clauses, post-selections or merging of variables.

Proof. This can be derived for the case of one input and output wire from concrete calculations of the underlying tensors, then for more wires, this follows by the SF_Z rule. □

If we apply these rewrite rules to a #2SAT diagram as much as possible, we end up with each clause and pair of clauses connecting distinct sets of variables. This means that the diagram now forms a graph, where the clauses are edges and the variables are vertices. Each edge is labelled with whether or not its variables are negated, and is directed to indicate which variable corresponds to which label. For example, we can see how the following instance is transformed from a formula into a diagram, and then simplified to graph form:

$$f(x_1, x_2, x_3) = (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_2) \wedge (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_3) \wedge (\neg x_2 \vee x_3)$$



Unfortunately, these rewrites cannot be applied recursively during the DPLL algorithm, since the algorithm maintains the structure of the graph. This is easy to see because unit propagation only removes clauses and variables, so it can't create situations where these rules would apply, and variable branching only adds clauses with one literal, to which these rules don't apply.

One other observation we can make diagrammatically is that if the diagram for a formula forms two disconnected components, then it can be written as a tensor product of two other scalar diagrams. But the tensor product of scalars is just multiplication, which means if a formula $f(\vec{x})$ contains two sets of variables which are not present together in any clauses, we can can split $f(\vec{x}) = f_1(\vec{x}_1) \wedge f_2(\vec{x}_2)$, and so we have that:

$$|\{\vec{x} \mid f(\vec{x}) = 1\}| = |\{\vec{x}_1 \mid f_1(\vec{x}_1) = 1\}| \cdot |\{\vec{x}_2 \mid f_2(\vec{x}_2) = 1\}| \tag{5.15}$$

This technique is used to great effect in all modern #SAT solvers, starting with Relsat, which was originally published by Bayardo and Schrag in 1997 [BS97].

5.2.2 Branching Numbers

In order to analyze algorithms that branch recursively like DPLL, we will need make to extensive use of *branching numbers*. Suppose we have an algorithm A, where the runtime is parameterized by some variable n, and each step of the algorithm for some instance ϕ consists of a series of cases defined by conditions C_i and subalgorithms A_i,

$$A(n,\phi) = \begin{cases} A_1(n,\phi) & \text{if } C_1(n,\phi) \\ \vdots \\ A_k(n,\phi) & \text{if } C_k(n,\phi) \end{cases} \tag{5.16}$$

and further suppose that each A_i calls A recursively some m number of times, along with a polynomial amount of other work, i.e

$$A_i(n) = f_i(A(n_{i1},\phi_{i1}), A(n_{i2},\phi_{i2}), \ldots, A(n_{im},\phi_{im})) \tag{5.17}$$

where the runtime of f_i is $O(\text{poly}(n))$, all $n_{ij} < n$, and each ϕ_{ij} can be determined in polynomial time. In this case we can see that if $T(n)$ is the runtime of $A(n,\phi)$, then the runtime of each $A_i(n,\phi)$ is

$$T_i(n) = O(\text{poly}(n)) + \sum_{j=1}^{m} T(n_{ij}) \tag{5.18}$$

and so the overall algorithm has a worst-case runtime of:

$$T(n) = \max_i T_i(n) = O(\text{poly}(n)) + \max_i \sum_{j=1}^{m} T(n_{ij}) \tag{5.19}$$

It has then been shown [Epp03] that we can bound the runtime of the algorithm by

$$T(n) = O(\text{poly(n)} \cdot 2^{\alpha_{max} n}) \qquad \alpha_{max} = \max_i \log_2 \tau(n - n_{i1}, \ldots, n - n_{im}) \tag{5.20}$$

where $\tau(k_{i1}, \ldots, k_{im})$ is the unique positive real root of the equation

$$1 = x^{-k_{i1}} + \cdots + x^{-k_{im}} \tag{5.21}$$

which is guaranteed to exist by Descartes' rule of signs [AJS98]. Each $\tau(\ldots)$ is called a branching number, and in general, we will refer to an algorithm as having an α-value of some constant α' if the runtime is bounded by $O(\text{poly}(n) \cdot 2^{\alpha' n})$.

5.2.3 An Algorithm for #2SAT

We will now show diagrammatically an algorithm $\#DPLL_2(f)$ for #2SAT, published by Dahllöf et al in 2002 [DJW02a], which achieves a runtime of $O(\text{poly(n)} \cdot 1.3247^n)$ – $\alpha = 0.4057$. Note throughout this that we refer to the number of clauses a variable x occurs in as the degree $d(x)$ of the variable – in the diagram, this is equivalent to the arity of the Z-spider corresponding to x – we will let $d(f)$ represent the maximum degree of any variable in a formula f. Additionally, we will refer to the neighbours of a variable x as the variables that appear in a clause with x.

Definition 5.1 ([DJW02a]). *The algorithm $\#DPLL_2(f)$ is as follows:*

1. *If f has at most four variables, compute the answer by exhaustive search.*
2. *If f contains the clauses x and $\neg x$ for some variable x, then f is unsatisfiable, so it has zero satisfying assignments.*
3. *If f contains no variables, then it has one satisfying solution, the empty solution.*
4. *Apply unit propagation and Lemmas 5.6 and 5.5 to f as much as possible.*
5. *If f consists of disjoint components $f_1, \dots, f_k$, then return $\prod_{i=1}^{k} \#DPLL_2(f_i)$.*
6. *If $d(f) = 3$, then return $\#DPLL_2^3(f)$.*
7. *Otherwise, pick a variable with $d(x) = d(f)$ and apply variable branching to generate f_1 and f_2.*
8. *Return $\#DPLL_2(f_1) + \#DPLL_2(f_2)$.*

We can see that this has a subalgorithm for the case where $d(f) \leq 3$, and this is given as follows. Here the function $L_f(x)$ is defined as the number of neighbours of x that have neighbours that are not also neighbours of x, and if $L_f(x) > 0$ then $B_f(x)$ is defined as one such neighbour of x.

Definition 5.2 ([DJW02a]). *The algorithm $\#DPLL_2^3(f)$ is as follows:*

1. *Apply steps one to five of $\#DPLL_2(f)$.*
2. *If $d(f) \leq 2$ then we have one of the following cases:*
   (a) *If f is a cycle, pick x as any variable.*
   (b) *If f is a chain, pick x as the variable closest to the center of the chain.*
3. *If $d(f) = 3$, pick a variable y such that $f(y) = 3$, and then we have one of the following cases:*
   (a) *If $L_f(y) = 1$, then let $x = B_f(y)$.*

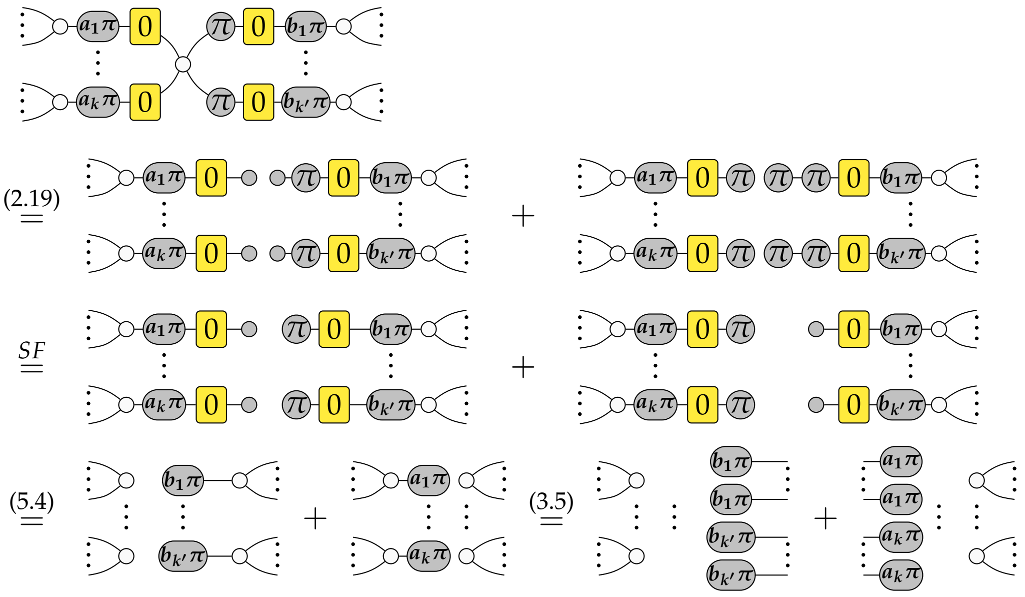
(b) *If* $L_f(y) > 1$, *then let* $x = y$.

4. *Apply variable splitting to* x *to generate* f_1 *and* f_2, *and return* $\#DPLL_2^3(f_1) + \#DPLL_2^3(f_2)$.

First we will give a lemma that is useful for both $\#DPLL_2^3$ and $\#DPLL_2$.

Lemma 5.7. *In a* #2SAT *diagram, branching on a variable* x *will eliminate* $k+1$ *variables in one branch, and* $k'+1$ *variables in the other branch, where* $k'+k = d(x)$, *for some* $0 \leq k \leq d(x)$.

Proof. Note that any variable is in k clauses negated and k' clauses not-negated for some $k \leq d(x)$, so

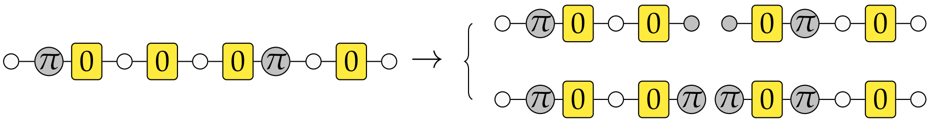


(5.22)

which completes the proof. □

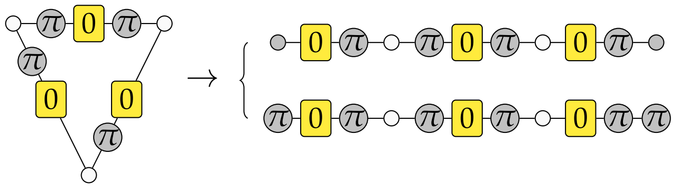
Now we will start by analysing $\#DPLL_2^3$, following the original analysis of [DJW02a], and we will analyse it in terms of the number of clauses m, not the variables n. The first step applies the termination conditions and unit propagation, and this takes polynomial time (specifically $O(m^2)$, since there are at most $O(m)$ variables and m clauses). It also decomposes f into disjoint components if possible, and this cannot increase the runtime of the final algorithm - since they are disjoint, no unit propagation will occur across components, and so they must be decomposed individually. From this, we will now assume that f is connected.

In the case that $d(f) \leq 2$, we have that either f is a cycle or a chain since it is connected. If f is a chain, then branching on the variable that is closest to the center of the chain will divide it into two components of size $\lfloor \frac{n}{2} \rfloor$ and $\lceil \frac{n}{2} \rceil$:



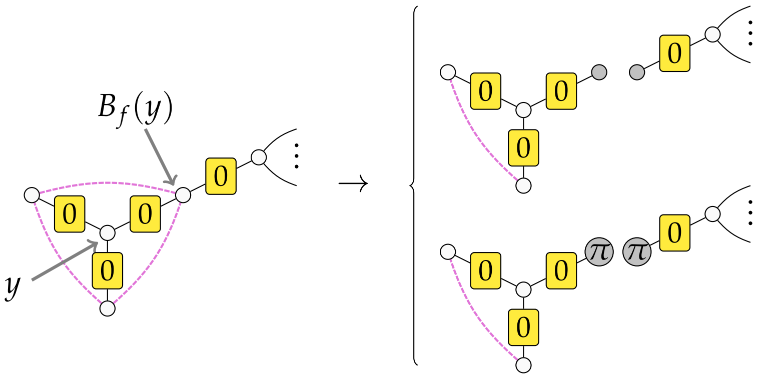
(5.23)

These components will be handled separately by the next step of the algorithm, so are essentially evaluating four terms, two of size $\lfloor \frac{m}{2} \rfloor$ and two of size $\lceil \frac{m}{2} \rceil$, giving a runtime bound of $T(m) \leq 4T(m/2)$ as m increases. An application of the master theorem then yields $T(m) = O(m^4)$. If f is a cycle, then picking any vertex and branching on it will yield two chains, so the runtime is also $O(m^4)$:



(5.24)

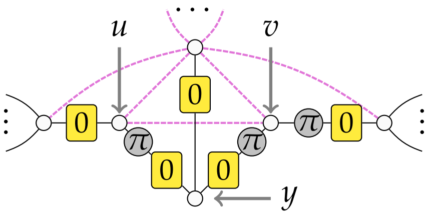
In the case that $d(f) = 3$, we pick $d(y) = 3$, and we know that $L_f(y) > 0$. This is because if $L_f(y) = 0$, then the diagram can only have four variables – y plus its three neighbours, so it must have been removed in the first step. In the case that $L_f(y) = 1$, we have the following situation



(5.25)

where the pink wires indicate places where clauses might exist. As indicated, branching on $B(y)$ will split the diagram into two parts, one of which has three variables, and then the rest. In either branch, at least four clauses are removed, since the component of three variables will be removed entirely by exhaustive search in the next step, and thus the runtime is bounded by $T(n) = 2T(n-4)$, which has a branching number of $\tau(4,4)$.

In the case that $L_f(y) > 1$, we have the following situation



(5.26)

but by Lemma 5.7, if we branch on y, y is always eliminated, and either u and v are both eliminated in one branch, in which case we remove at least five clauses in one branch and three in the other, or u and v are eliminated in different branches, in which case we remove at least four clauses in each branch. Therefore, the branching number, in this case, is either $\tau(4,4)$ or $\tau(3,5)$.

Therefore, we have $\alpha_{max} = \max\{\log_2(\tau(4,4)), \log_2(\tau(5,5))\} = 0.2556$, and the runtime of the whole algorithm $\#DPLL_2^3(f)$ is bounded above by $O(\text{poly}(m) \cdot 2^{0.2556 \cdot m})$. However, since we only call $\#DPLL_2^3(f)$ when f has degree at most three, we have $m \leq \frac{3n}{2}$, and thus the runtime is bounded by $O(\text{poly}(n) \cdot 2^{0.2556 \cdot \frac{3n}{2}}) = O(\text{poly}(n) \cdot 2^{0.3835 \cdot n})$.

Now to analyze $\#DPLL_2(f)$, similarly to $\#DPLL_2^3(f)$, the first five steps are polynomial or don't change the final runtime. If $d(f) \leq 3$, then the branching number is $\tau(4,4)$ or $\tau(3,5)$. Otherwise we pick x with $d(x) = d(f) > 3$ to branch on, so by Lemma 5.7 the branching number is $\tau(1+k, 1+d(f)-k)$ where $0 \leq k \leq d(f)$. However, it can be shown that $\max_k \tau(1+k, 1+c-k) = \tau(1, 1+c)$, and also that $\tau(1, 1+a) \geq \tau(1, 1+b)$ whenever $a \leq b$. Therefore, we have that the branching number when $d(f) > 3$ is bounded above by $\tau(1, 1+d(f)) \leq \tau(1,5)$. Thus the algorithm overall has a branching number of $\max\{\tau(1,5), \tau(3,5), \tau(4,4)\} = \tau(1,5) = 1.3247$, so the runtime is bounded by $O(\text{poly}(n) \cdot 2^{0.4057 \cdot n})$.

5.2.4 Phase Transitions in Density

It is well-known that the SAT problem has a phase transition in terms of density - that is, there is a critical density δ_{crit} at which instances transition from 'easy' to 'hard'. This is known both theoretically, where it has been proven that for DPLL-style algorithms, $\delta_{crit} = 4.63$ is the critical point for 3SAT [MPZ02], and has been demonstrated extensively in practice. For #SAT, no similar theoretical results are known - the first proven phase transition in a #P-complete problem is counting the number of independent sets of a graph, which Allan Sly showed [Sly10] has a phase transition in 2010.

Practically, however, most algorithms for solving #SAT do exhibit a 'hardest' density, below and above which instances become easier. This has been measured for #3SAT by Birnbaum and Lozinskii [BL99], Bayardo and Schrag [BS97], Darwiche [Dar02], and others, who observe values of δ_{crit} ranging from 1.2 to 2.1 depending on the algorithm. However, so far as the author can find, similar experiments have not been published for #2SAT.

Since this will be instructive to compare against algorithms presented in later chapters, the dependence of solving difficulty on formula density was measured for several different algorithms:

- $\#DPLL_2$ from above, and $\#DPLL$ of Birnbaum and Lozinskii [BL99], both based on plain DPLL (and implemented by the author).
- Ganak, a probabilistic exact solver written by Sharma at el [SRSM19] that is based on the conflict-driven clause learning technique of SAT solving (SharpSAT [Thu06], on which Ganak is based, was also tested but it showed identical performance).
- miniC2D, a solver based on compiling CNF formulae to binary decision diagrams written by Oztok and Darwiche [OD15].

Random instances of n variables and m edges were sampled by creating empty graphs on n vertices and adding m edges uniformly at random, and then turning each edge into a clause by picking the polarity and ordering of the literals randomly. Instances were sampled in this way to ensure that the simplifications in Lemmas 5.6 and 5.5 are not applicable.

For each combination of n and m, a hundred instances were sampled, and the same instances were provided to all of the algorithms. For the DPLL-based algorithms ($\#DPLL$, $\#DPLL_2$ and Ganak), the total number of recursive calls was measured, whereas for miniC2D the size of the compiled decision diagram was measured, as these essentially represent the 'effort' required to solve the instance, but is not subject to the instability of time measurements - solving time was also measured, but the trends over density are totally eclipsed by other factors at this scale. The results are presented in Figure 5.1.

We can observe that the hardest density for all the algorithms appears to be around $\delta = 1$. It is known that there is a transition between satisfiability and unsatisfiable for 2SAT instances, which has $\delta_{crit} = 1$, and so it seems reasonable to suggest that these two are related. One possible explanation is that there are two competing effects: the number of solutions and the number of clauses - effort increases with more clauses but decreases with fewer solutions. Indeed, the results for Ganak (and even miniC2D, when rescaled) are remarkably close to the geometric mean of density and bit length of the number of solutions, as shown in Figure 5.2.

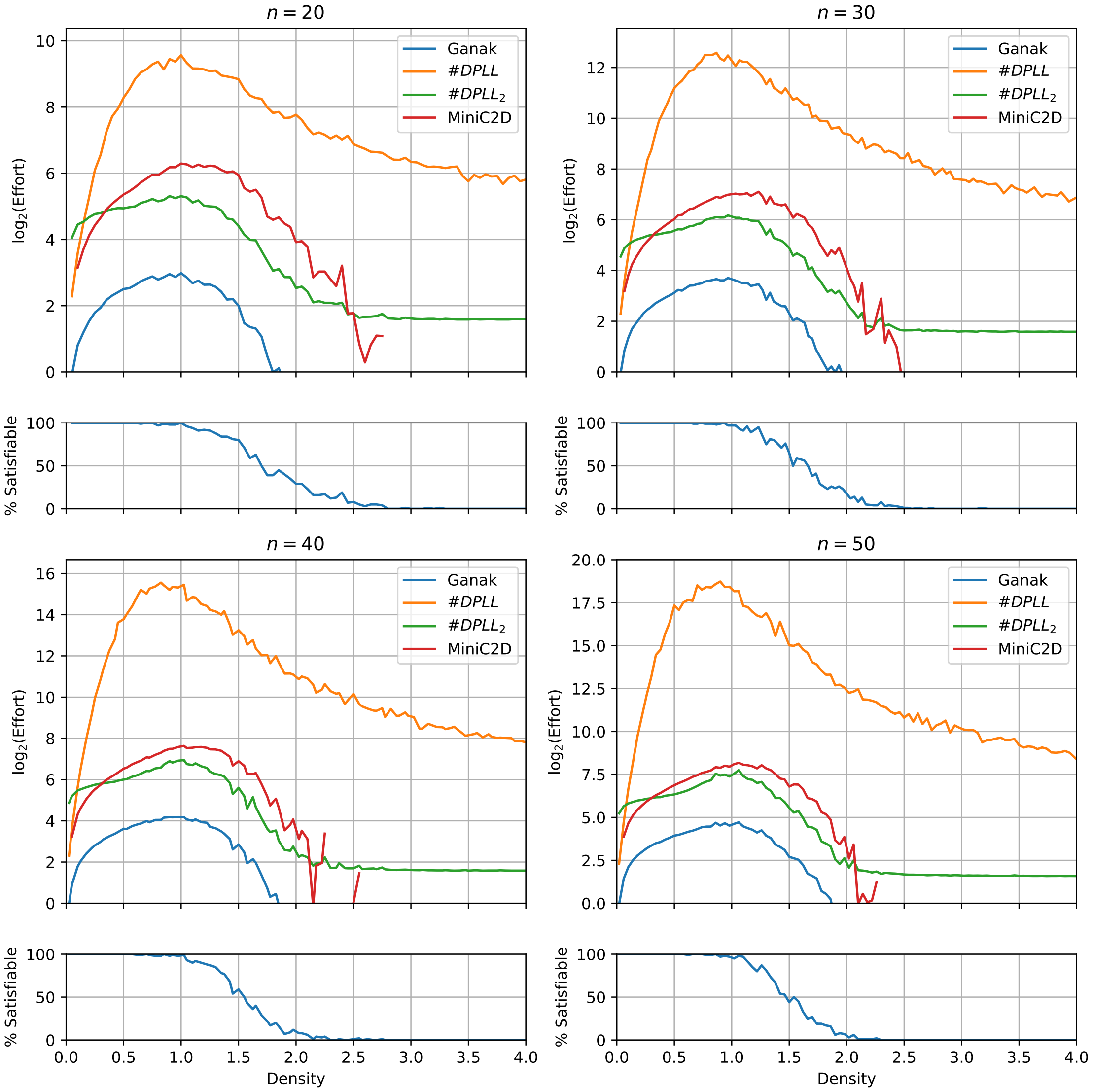


Figure 5.1: The effort required to solve #2SAT instances of a fixed density. Four plots are presented, one each for $n = 20$, 30, 40, and 50 variables. Below each plot is a separate plot of the percentage of instances which were satisfiable at each density. The hardest density for all algorithms appears to coincide with the transition in satisfiability.

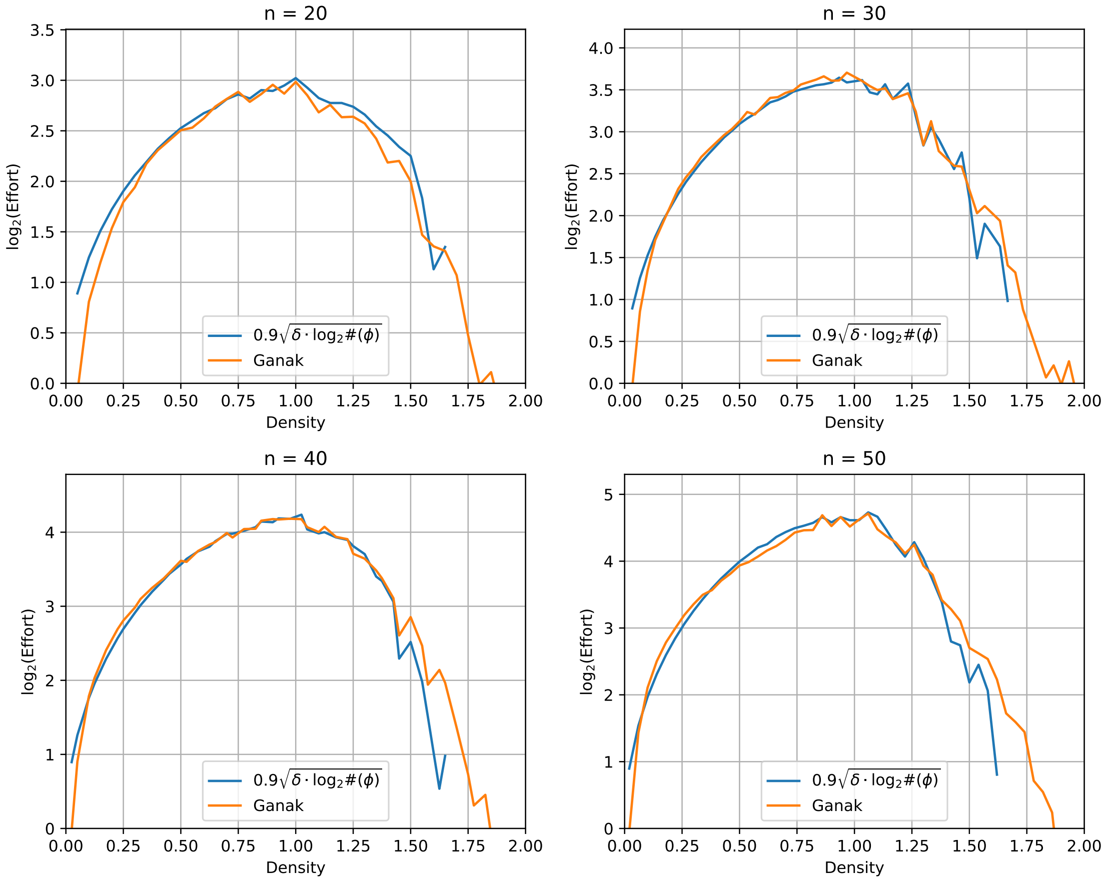
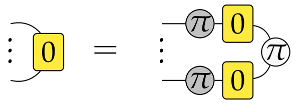


Figure 5.2: A comparison between the effort required by Ganak to solve #2SAT instances of a fixed density, and the geometric mean of the number of solutions and the density. Up to a scalar factor, these are fairly similar.

5.3 Augmenting DPLL with Negative Variables

In the previous chapter, we saw that it is possible to reduce arbitrary ZH-diagrams within certain fragments to the ZH_{π}^{0} fragment of the ZH calculus. In particular, these fragments include all the diagrams representing #kSAT instances. In this section, we will use these techniques to obtain upper bounds on the runtime of #kSAT that is independent of k. Firstly, we have the following lemma, which is the embedding of arbitrary arity #SAT clauses into ZH_0.

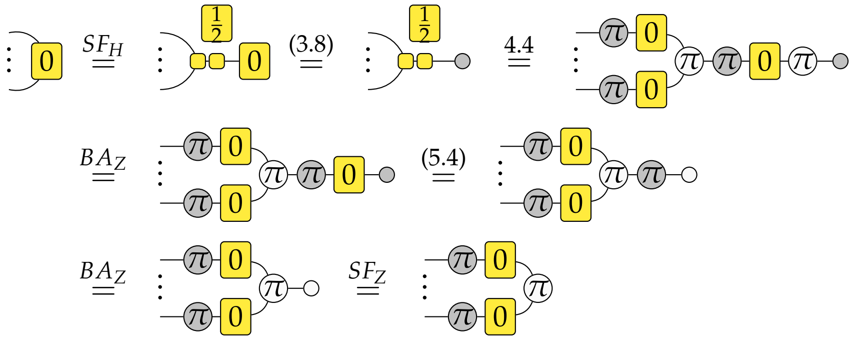
Lemma 5.8. *The following diagrammatic equivalence holds:*



(5.27)

This directly generalizes the BW *axiom of the* ΔZX*-calculus.*

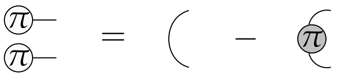
Proof. We can see the following



(5.28)

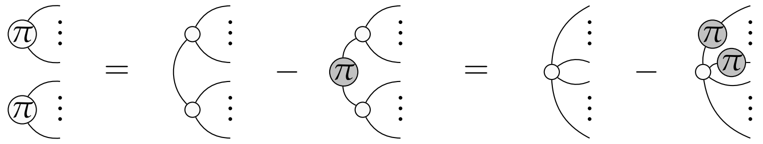
which completes the proof. □

If we applied this to every clause in a #SAT diagram with n variables and m clauses, we would have a ZH_{π}^{0} diagram with $n+m$ spiders. Furthermore, with the exception of π-phases on m of these spiders, this diagram would represent a #2SAT instance, as there are no X-spiders present that are not $X_1^1[\pi]$-spiders (logical NOT gates). Therefore, one strategy for solving #kSAT might be to write such a diagram as a sum of diagrams that remove these π-phases and then apply a #2SAT solver to these. As discussed in the previous chapter, it is unlikely that such a sum can be expressed in polynomially many terms, but it is easy to express in exponentially many. In particular, we can apply the following decomposition recursively



(5.29)

and since each decomposition removes two π-phases, we will end up with $2^{\frac{m}{2}} = 1.4142^m$ diagrams. We can also see that each of these diagrams will have only $n + \frac{m}{2}$ spiders, since:



(5.30)

Therefore, if we were to use the $\#DPLL_2$ algorithm from the previous section to evaluate each of these diagrams, we would end up with a total runtime of $O(\text{poly}(n,m) \cdot 1.4142^m 1.3247^{n+\frac{m}{2}}) = O(\text{poly}(n,m) \cdot 2^{0.4057n+0.7028m})$.

5.3.1 Upper Bounds on #*k*SAT

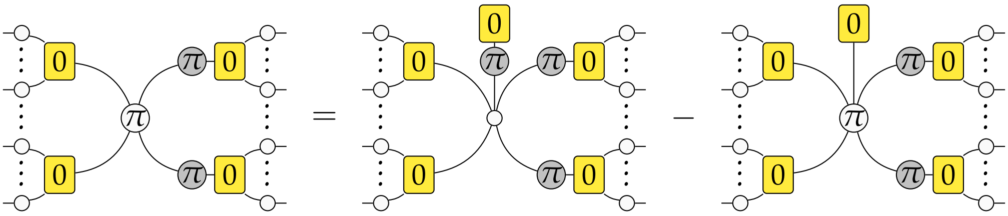
However, there are better algorithms. In particular, for #2SAT, there is a history of steadily improving bounds on the worst-case runtime, starting with Dubois in 1991 [Dub91], who gave a $O(\text{poly}(n)1.6180^n)$ algorithm, followed by Dahllöf et al [DJW02a] who presented $\#DPLL_2$ in 2002, and refined it in 2003 to give a $O(\text{poly}(n)1.2561^n)$ bound. Furer and Kasiviswanathan [FK07] reanalyze this to obtain $O(\text{poly}(n)1.2461^n)$, and finally Wahlström [Wah08] introduced a significantly more complicated analysis of the same to obtain the current best-known bound of $O(\text{poly}(n)1.2377^n)$.

For #*k*SAT with $k > 2$, bounds better than the $O(\text{poly}(n,m)2^n)$ runtime of brute-force enumeration are also known. For $k = 3$, Dahllöf et al [DJW02a] give an upper bound of $O(\text{poly}(n)1.6894^n)$ in the same paper that presents $\#DPLL_2$, which was improved by Kutzkov [Kut07], who obtained $O(\text{poly}(n)1.6423^n)$. For larger k, the results of Dubois provide a bound of $O(\text{poly}(n)c_k^n)$ where $c_k \to 2$ as $k \to \infty$. Similarly, Williams [Wil04] presents an analysis of the average-case behaviour, finding a runtime of $O(\text{poly}(n){c'_k}^n)$, also with $c'_k \to 2$ (although $c'_k < c_k$ for all k). In general, it is conjectured that any algorithms for #*k*SAT must have a worst-case runtime of $O(\text{poly}(n,m)c_k^n)$ where $c_k \to 2$ as $k \to \infty$ – this can be seen either as a consequence of the Strong Exponential Time Hypothesis (SETH) [CIP09], or as its own hypothesis #SETH – although this conjecture is not held universally [Wil21].

5.3.2 Negative Variables

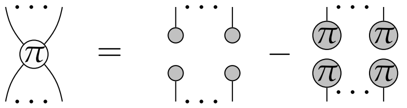
Returning to the previous #SAT algorithm, we can apply the #2SAT algorithm of Wahlström [Wah08] to get a better upper bound of $O(\text{poly}(n,m) \cdot 2^{0.3068n+0.6538m})$. However, rather than writing #SAT diagrams as a sum of #2SAT diagrams, a more powerful observation is to note that DPLL-style algorithms are able to evaluate diagrams with π-phase Z-spiders natively! Indeed, only slight variations to introduce sign changes for unit propagation and variable branching are needed to handle Z-spiders with π-phases, and are given as follows:

Lemma 5.9. *The following diagrammatic equivalent to the variable branching rule holds:*



(5.31)

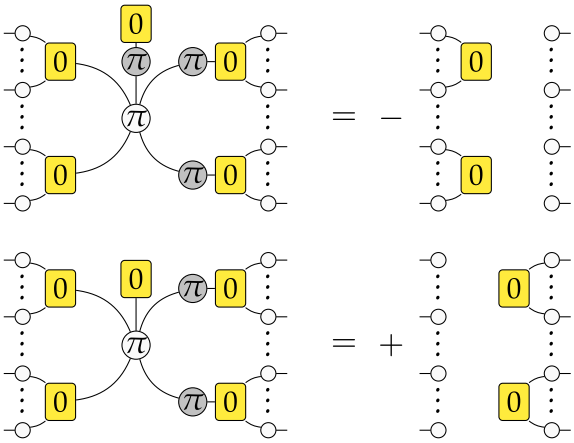
Proof. This follows from the sum



(5.32)

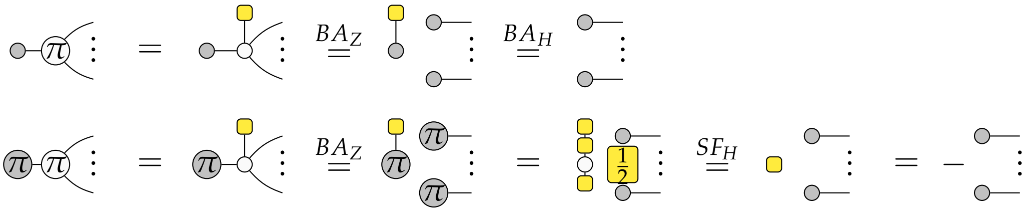
together with the proof of Lemma 5.3. □

Lemma 5.10. *The following diagrammatic equivalent to the unit propagation rule holds:*



(5.33)

Proof. This follows from the identities



(5.34)

together with the proof of Lemma 5.1. □

Therefore, we can define a variant of #SAT, #SAT$_\pm$ which allows variables to be marked as 'positive' or 'negative'. Then diagrammatically, positive variables are represented by Z-spiders with no phase and negative variables by Z-spiders with π-phases. By exploiting the branching rule above, we have the following variant of DPLL to handle these instances, which we will call $\#DPLL_\pm(f)$.

Definition 5.3. *For a given Boolean CNF formula f on variables $x_1, \dots, x_n$, and a set of variables N (the negative variables), the algorithm $\#DPLL_{\pm}(f)$ computes the value*

$$\#DPLL_{\pm}(f) = \sum_{x_i} (-1)^{\sum_{x_j \in N} x_j} f(\vec{x}) \tag{5.35}$$

and is given by the following:

- *If f contains the clauses x and $\neg x$ for some variable x, then f is unsatisfiable, so it has zero satisfying assignments.*
- *If f contains no variables, then it has one satisfying solution, the empty solution.*
- *Apply unit propogation to f as much as possible.*
- *Pick a variable x that occurs in f and apply variable branching to generate f_1 and f_2.*
- *If x is positive ($x \notin N$), return $\#DPLL(f_1) + \#DPLL(f_2)$, otherwise if x is negative ($x \in N$) return $\#DPLL(f_1) - \#DPLL(f_2)$.*

With a smart choice of the variables to branch on, the worst-case runtime of this algorithm can be bounded in exactly the same way as the regular $\#DPLL$. In particular, this means that the $O(\text{poly}(n)1.2377^n)$ bound of Wahlström [Wah08] can be adapted directly. Therefore, by applying Lemma 5.8 to any #SAT diagram and then applying $\#DPLL_{\pm}$ to the resulting diagram directly, we can evaluate #SAT instances in time $O(\text{poly}(n,m)1.2377^{n+m}) = O(\text{poly}(n,m)2^{0.3068n+0.3068m})$, which is certainly better than the bound given by decomposing into a sum of diagrams. From now on, we will refer to this method as $\#DPLL_{\pm}^{\to 2}$.

It remains to ask, when is $\#DPLL_{\pm}^{\to 2}$ actually useful? First, note that if only positive variables are picked for branching, the action of $\#DPLL_{\pm}$ on a translated #SAT diagram is *exactly* the same as the action of regular $\#DPLL$ on the original diagram. Therefore, we would only expect gains when decomposing some of the negative variables (i.e clauses), and thus it is natural to suspect that this bound will only be useful for instances with few clauses.

For diagrams with a fixed maximum density δ_{max}, we have that $m \leq n\delta_{max}$, and so the runtime of $\#DPLL_{\pm}^{\to 2}$ is bounded by $O(\text{poly}(n, \delta_{max})2^{0.3068(1+\delta_{max})n})$. Firstly, we can see that this is better than the naive $O(\text{poly}(n,m)2^n)$ whenever $\delta_{max} < 2.2503$. Since this is independent of k, it means that for any $\delta_{max} < 2.2503$ and sufficiently large k, this beats both the worst-case bound of Dubois [Dub91] and the average-case bound of Williams [Wil04]. Since it is suspected that it is not possible to do better than $O(\text{poly}(n, m)2^n)$ in general (SETH), this indicates that the 'hardest' density of #SAT must be some $\delta > 2.2503$.

For the decision problem SAT, similar bounds in terms of clauses are known independently of k - in particular, there is a classic result of Monien and Speckenmeyer from 1980 [MSV81] that SAT can be solved in $O(\text{poly}(n, m)2^{\frac{m}{3}})$ time, which was improved to $O(\text{poly}(n, m)2^{0.30897m})$ by Hirsch in 2000 [Hir00], and implies a better than brute-force runtime for $\delta_{max} < 3.2366$. Generally, there are two general approaches taken by algorithms for SAT - either bounding runtime in terms of maximum density (i.e runtime measured in terms of m) or in terms of maximum clause size (i.e runtime measured in terms of n and k) - it has been shown that in the limit of large k and large δ, these have the same runtime [CIP06].

While similar bounds in terms of m have been previously obtained for the specific cases of #2SAT and #3SAT - $O(\text{poly}(n, m)1.1892^m)$ was obtained for #2SAT and $O(\text{poly}(n, m)1.4142^m)$ for #3SAT by Zhou et al [ZYZ10], later improved to a runtime of $O(\text{poly}(n, m)1.1710^m)$ for #2SAT by Wang and Gu [WG13] - as far as the author is aware, this is the first bound in terms of m for #SAT independent of k. Other similar bounds have been obtained for #CircuitSAT, which concerns counting satisfying assignments of Boolean formulae that are not in CNF form - e.g Golovnev et al [GKST16] find a better than brute-force runtime whenever a formula has $\delta < 2.99$, although in this context δ refers to the ratio of binary logic gates needed to express the formula, not clauses.

Concretely, $\#DPLL_{\pm}^{\rightarrow 2}$ is better than the average-case bounds of Williams [Wil04] whenever $\delta_{max} < 1.217$ and $k \geq 6$, and better than the worst-case bounds of Dubois [Dub91] whenever $k \geq 3$ and $\delta_{max} < 1.858$. Clearly, it offers no improvement on #2SAT, but it is also not applicable to #3SAT - when $\delta < 1.6$, the $O(\text{poly}(n, m)1.4142^m)$ bound of Zhou et al [ZYZ10] is sharper, and when $\delta \geq 1.6$ the $O(\text{poly}(n)1.6423^n)$ bound of Kutzkov [Kut07] is sharper.

Finally, noting that $\#DPLL_{\pm}^{\rightarrow 2}$ maps a #SAT instance of m clauses to a #2SAT instance of km clauses (with negative variables), and applying the upper bound of $O(\text{poly}(n, m)1.1710^m)$ on #2SAT found by Wang and Gu [WG13], we can bound the runtime of $\#DPLL_{\pm}^{\rightarrow 2}$ purely in terms of clauses, not mentioning variables, as $O(\text{poly}(n, m)1.1710^{km})$. This does not offer any improvement over known bounds for $k = 2$ or $k = 3$, and for $k > 4$, it is worse than the naive $O(\text{poly}(n, m)2^m)$ upper bound (which follows from branching on all the negative variables, so that there are no clauses left in the diagram). However, for $k = 4$, this yields a time of $O(\text{poly}(n, m)1.8803^m)$. As far as the author can tell this is the best bound for #4SAT in terms of m, although it is probably of limited interest.

CHAPTER 6

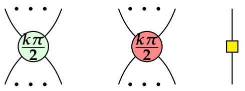
Stabilizer Decompositions

The first and second parts of this chapter are exposition and do not contain any novel content. The third part contains several unsuccessful techniques that extend existing work, while the fourth details some novel decompositions found using the author's implementation of previously known methods.

In this dissertation so far, we have seen various identities that allow us to write specific diagrams as the sum of other diagrams - we call these graphical decompositions. They have been used in various contexts to compute some quantity by writing it as the sum of other quantities that are easier to compute. In this section, we will describe a certain class of graphical decompositions, stabilizer decompositions, and how such decompositions can be found automatically. Finally, we will present novel stabilizer decompositions that were found by these methods for certain diagrams of interest.

Stabilizer diagrams are all diagrams that exist in the stabilizer fragment StabZX of the ZX-calculus. We will also consider a ZX-diagram to have a stabilizer region, or stabilizer part if some part of this diagram consists only of generators from this fragment. The fragment is given as follows.

Definition 6.1 ([Kv22, Definition 4.1.1]). StabZX *is the fragment of the ZX-calculus given by the following generators*



(6.1)

where $k \in \mathbb{Z}$.

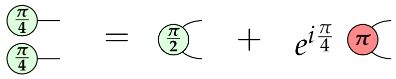
These diagrams are of interest because of a famous result of Gottesman and Knill that Eval(StabZX) can be computed in polynomial time [AG04]. We will expand on this further soon, but this implies that if a scalar ZX-diagram with n generators can be written as a sum of k StabZX-diagrams, then it can be evaluated in $O(k \cdot \text{poly}(n))$ time. Therefore, if we can show an upper bound on the value of k for some family of diagrams, we can put an upper bound on the time required to evaluate them.

Definition 6.2. *A stabilizer decomposition of size k for some ZX-diagram D is a set of* StabZX*-diagrams $D_1, \ldots, D_k$ and $c_1, \ldots, c_k \in \mathbb{C}$ such that*

$$[\![D]\!] = c_1[\![D_1]\!] + c_2[\![D_2]\!] + \cdots + c_k[\![D_k]\!] \tag{6.2}$$

The stabilizer rank $\chi(D)$ of a ZX-diagram D is the smallest $k \in \mathbb{N}$ such that D has a stabilizer decomposition of size k.

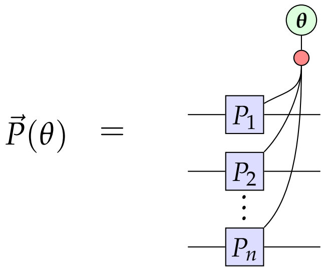
Note that in this definition, all scalar diagrams D have stabilizer rank one, since $[\![D]\!] = z = z \cdot 1 = z[\![E]\!]$ for some $z \in \mathbb{C}$, where E is the empty diagram, which is certainly a stabilizer diagram. However, this is not useful in practice because we don't know in advance what the coefficient of decomposition is. Instead, we will focus on finding stabilizer decompositions and ranks for non-scalar diagrams. For example, the following diagram can be shown to have a stabilizer rank of two, with the following decomposition of size two [KvV22]:



(6.3)

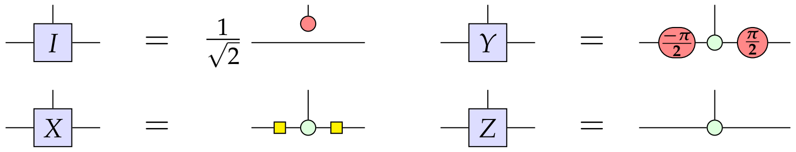
While StabZX can be characterized as all tensor networks consisting of only the specified generators, there is also another characterization, as products of *Pauli exponentials*. In order to show this, we will first define Pauli exponentials, and then show how stabilizer diagrams can be brought into a normal form in terms of them.

Definition 6.3 ([Kv22, 5.1.5]). *A Pauli exponential $\vec{P}(\theta)$ of size n is a tensor with n inputs and n outputs. It is parameterized by a sequence $\vec{P} = P_1P_2 \cdots P_n$ where $P_i \in \{I, X, Y, Z\}$ and an angle $\theta \in \mathbb{R}$. Diagrammatically, it is defined by*



(6.4)

where the blue boxes are called Pauli boxes and are defined by

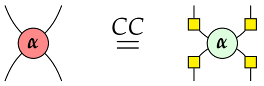


(6.5)

To convert StabZX-diagrams into a sequence of Pauli exponentials, we will apply two rewriting rules called local complementation and pivoting to reduce these diagrams to a normal form called 'graph state with local Cliffords' (GSLC) form. In the context of ZX-diagrams, this was introduced Duncan and Perdrix [DP09], although the normal form itself was previously introduced by Van den Nest [dNDDM04] in the context of quantum computing.

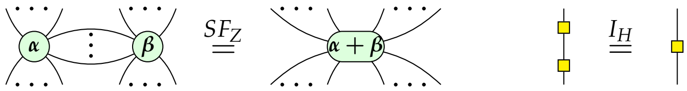
However, before these rules can be used, we must first transform StabZX-diagrams so that they consist of only Z-spiders connected together by Hadamard gates (known as a graph-like diagram). This can be accomplished through the following procedure [Kv22, Proposition 4.1.8]:

- Apply the colour change rule to transform all X-spiders into Z-spiders and Hadamards:



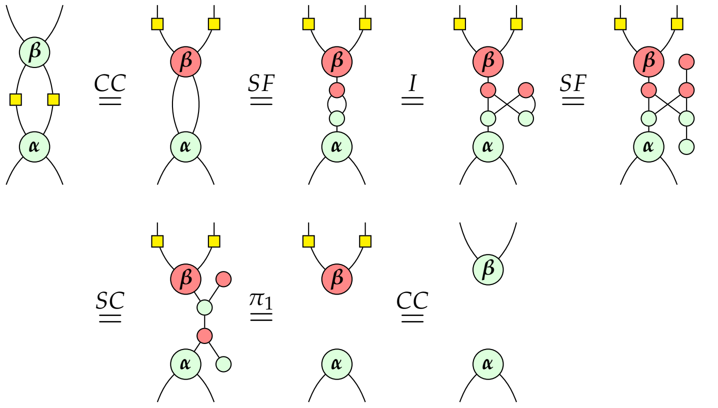
(6.6)

- Apply the spider fusion (SF_Z) and Hadamard cancellation (I_H) as much as possible:



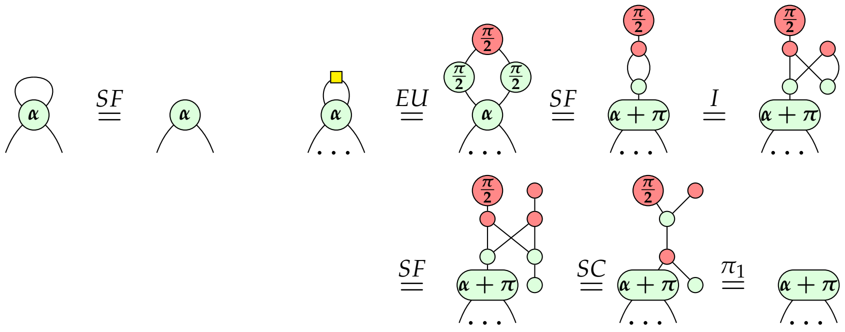
(6.7)

- Remove pairs of Hadamards between spiders as follows:



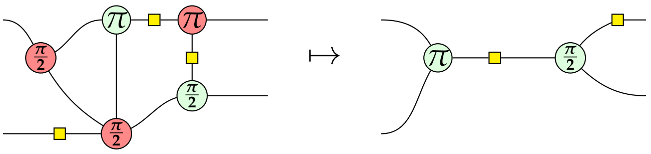
(6.8)

- Remove self-loops using the following two equations:



(6.9)

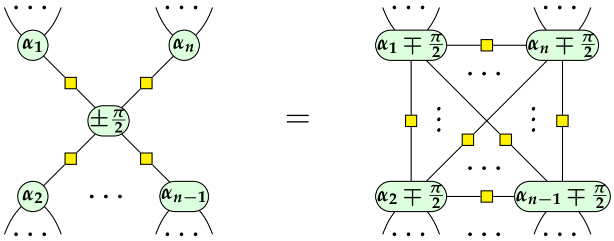
For example, we can see the action of this procedure on the following diagram:



(6.10)

Now that we can represent diagrams in graph-like form, we are ready to apply the two rewriting rules. These rules can be used to remove Z-spiders with $\pm\frac{\pi}{2}$ phases or pairs of Z-spiders with 0 and π phases from the diagram, assuming that they are not connected to any inputs or outputs. We will call a spider internal if it is not connected to any inputs or outputs, Pauli if it has phase 0 or π, and proper Clifford if it has phase $\pm\frac{\pi}{2}$. A pair of spiders is called connected if there is a Hadamard gate that connects them.

Lemma 6.1 ([Kv22, Lemma 4.2.9]). *Any internal proper Clifford spider can be removed from a ZX-diagram. The following rewriting rule called local complementation, holds*

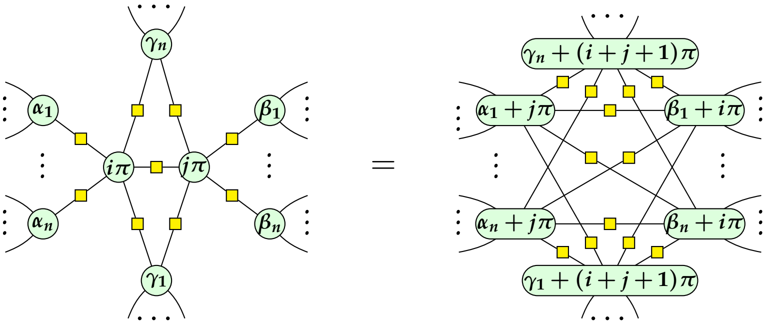


α_1
α_n
$\pm\frac{\pi}{2}$
α_2
α_{n-1}
$=$
$\alpha_1 \mp \frac{\pi}{2}$
$\alpha_n \mp \frac{\pi}{2}$
$\alpha_2 \mp \frac{\pi}{2}$
$\alpha_{n-1} \mp \frac{\pi}{2}$

(6.11)

up to a scalar factor, where the connections implied on the right-hand side form a complete graph where every spider is connected to every other spider.

Lemma 6.2. *[Kv22, Lemma 4.2.10] Any pair of connected internal Pauli spiders can be removed from a ZX-diagram. The following rewriting rule, called pivoting, holds*



(6.12)

up to a scalar factor, where $i, j \in \mathbb{Z}$, and the connections implied on the right-hand side form a complete tripartite graph, where every spider on the left is connected to every spider on the right, top, and bottom.

Note that as a consequence of these two rules on scalar diagrams, we can see a weak version of Gottesman and Knill's result directly.

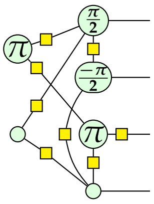
Theorem 6.1 ([Kv22, Proposition 4.4.1])**.** Eval(StabZX) *is computable in polynomial time.*

Proof. Given any scalar StabZX-diagram D with n spiders, apply the procedure above to transform it into graph-like form. Apply Lemmas 6.1 and 6.2 as much as possible, interleaving Equation (6.8) where necessary to remove double edges. This step takes $O(n^3)$ time, since the initial rewrite does a constant amount of work per wire and generator (which is $O(n^2)$ time), and there can be at most $O(n)$ applications of pivoting and local complementation (since they remove at least one Z-spider each time), each of which takes $O(n^2)$ time (because in the worst case we may add $O(n^2)$ Hadamards and edges).

After this, the diagram cannot have any proper Clifford spiders - as every spider in a scalar diagram is internal, they would've been removed by an application of Lemma 6.1. The diagram also cannot have any pairs of connected Pauli spiders, as they would've been removed by an application of Lemma 6.2. Since in a StabZX-diagram, every Z-spider is either Pauli or proper Clifford, this means the diagram can only consist of disconnected Pauli spiders.

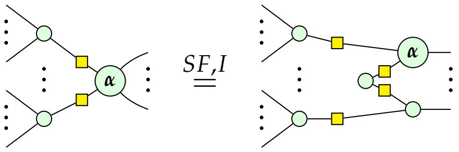
Finally, the tensor of the remaining diagram can be evaluated numerically in linear time, since the whole diagram is a tensor product $\llbracket D' \rrbracket = \llbracket Z_1 \rrbracket \otimes \llbracket Z_2 \rrbracket \otimes \cdots \llbracket Z_n \rrbracket = \llbracket Z_1 \rrbracket \cdot \llbracket Z_2 \rrbracket \cdots \llbracket Z_n \rrbracket$, where each Z_i is a scalar diagram consisting of one Pauli spider with no wires, so they can be evaluated in constant time, and there are at most n of them. □

Suppose we have a state represented as a StabZX-diagram, then we can transform it into a graph-like diagram, and apply local complementation and pivoting as much as possible. Then the only remaining spiders are those connected to outputs and interior Pauli spiders that are only connected to spiders that are connected to outputs. This is known as the affine-with-phases normal form [Kv22, Definition 4.3.1], and might look something like the following:



(6.13)

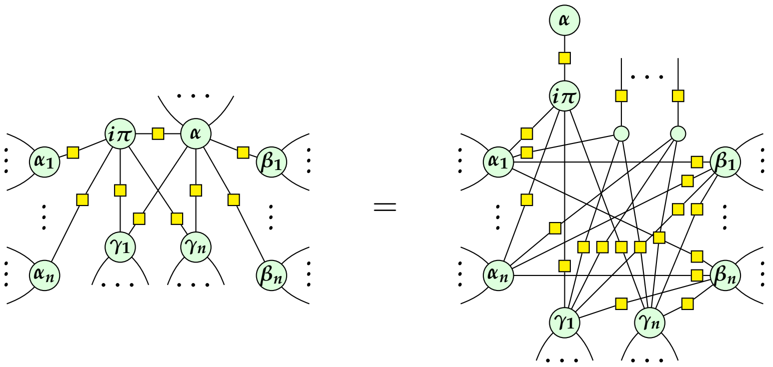
We can rewrite this so that every spider that is connected to an output is connected to exactly one output by unfusing an interior Pauli spider as follows:



(6.14)

In order to simplify this further into the GSLC normal form, we will use the following variation to the pivoting rule, called the boundary pivot. This is incredibly messy, but the exact pattern of connections is not important, only that it can, in principle, be done.

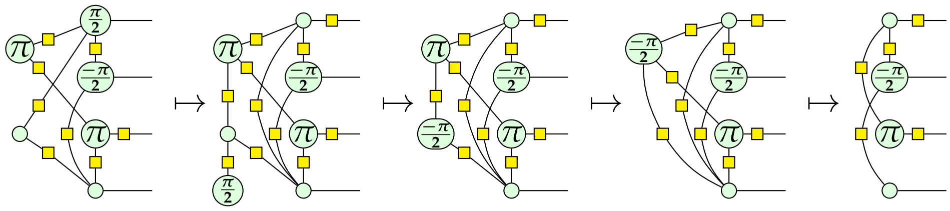
Lemma 6.3 ([Kv22, Lemma 4.3.5])**.** *Any internal Pauli spider can be simplified. The following rewriting rule, called boundary pivoting, holds*



(6.15)

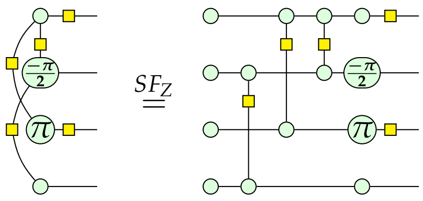
up to a scalar factor, where $i \in \mathbb{Z}$, and the connections implied on the right-hand side form a complete tripartite graph, where every spider on the left is connected to every spider on the right, top, and bottom.

Given a StabZX-diagram that is in affine-with-phases form, this can be applied to all interior spiders. If the interior spider is connected to a Pauli spider, then the extra spider introduced and the original interior spider can be removed directly by a pivot, whereas if the interior spider is connected to a proper Clifford spider, then the extra spider introduced can be removed by a local complementation, which then allows the interior spider to be removed by a local complementation. For example, we have the following:



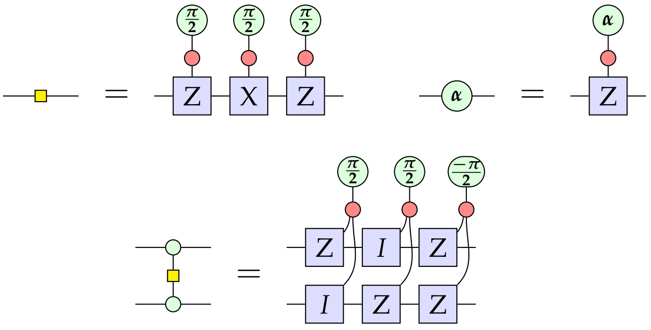
(6.16)

Once all the interior spiders have been removed in this manner, we are left with a diagram in GSLC form. This can now be expressed as a product of Pauli exponentials. First, we will unfuse all phases on the spiders, Hadamards connecting the spiders to each other, and Hadamards connecting the spiders to outputs. For example:



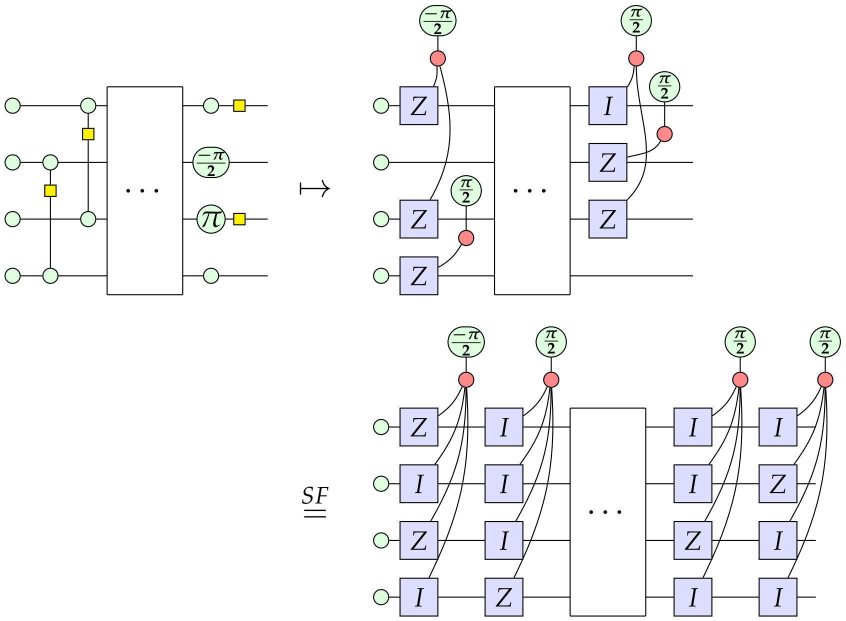
(6.17)

Now we can use the following identities (which can be verified by concrete calculation) to rewrite these as Pauli exponentials [Kv22, Lemma 5.1.14]:



(6.18)

Finally, by considering any unused wire at every stage to be a Pauli I box, we can write the whole diagram as a tensor product of Z-spiders, followed by a series of Pauli exponentials:



(6.19)

This shows an equivalence between all states reachable as a product of Pauli exponentials with angles $\theta = k\frac{\pi}{2}$ for $k \in \mathbb{Z}$ and StabZX diagrams. We have just shown such diagrams can be rewritten as Pauli exponentials, but also note that any Pauli exponential with $\theta = k\frac{\pi}{2}$ is in fact a StabZX-diagram, so they are exactly equivalent.

6.1 Searching by Simulated Annealing

The following section is an exposition of the work of Bravyi, Smith, and Smolin from the paper "Trading Classical and Quantum Computational Resources" [BSS16]

In order to search for stabilizer decompositions, we first need to define some measure of the distance of states, in order to guide the search in the right direction. The measure we will use is called fidelity, and it is defined in terms of the underlying tensor, via the process of vectorization.

Definition 6.4. *Given some tensor over $\mathbb{C}$ with n outputs each of dimension d and no inputs, $A^{i_1 \cdots i_n}$, the vectorization* $\text{vec}(A)$ *is a vector over $\mathbb{C}$ of size d^n given by:*

$$\text{vec}(A)_i = A^{i_1 \cdots i_n} \quad \textit{where} \quad i = i_1 + di_2 + d^2 i_3 + \cdots + d^{n-1} i_n \tag{6.20}$$

That is, the values i_k are given by the base d expansion of i for each i.

Definition 6.5. *Given two tensors over $\mathbb{C}$ with n outputs, $A^{i_1 \cdots i_n}$ and $B^{i_1 \cdots i_n}$, the fidelity of A and B is given by:*

$$\mathcal{F}(A, B) = \frac{|\text{vec}(A) \cdot \text{vec}(B)|}{||\text{vec}(A)|| \, ||\text{vec}(B)||} \tag{6.21}$$

In the case of ZX and ZH-diagrams, $d = 2$, and so vectorization corresponds to writing out the elements of the tensor in binary ascending order. Note that by the Cauchy-Schwartz inequality, $0 \leq \mathcal{F}(A, B) \leq 1$ with $\mathcal{F}(A, B) = 1$ if and only if A and B are proportional. Therefore, if we search for stabilizer decompositions by optimizing the fidelity between the target state and all possible stabilizer decompositions with a fixed number of terms, we will find such a decomposition if it exists.

One particularly nice property of fidelity is that when one of the tensors is given by a linear combination of other tensors, it is possible analytically to maximize the fidelity over all possible coefficients of the combination. First note the following relationship between the Euclidean distance and dot product - for any vectors $\vec{a}$ and $\vec{b}$ such that $||\vec{a}|| = ||\vec{b}|| = 1$ we have

$$||\vec{a} - \vec{b}|| = \sqrt{(\vec{a} - \vec{b}) \cdot (\vec{a} - \vec{b})} = \sqrt{\vec{a} \cdot \vec{a} + \vec{b} \cdot \vec{b} - 2\vec{a} \cdot \vec{b}} = \sqrt{2(1 - \vec{a} \cdot \vec{b})} \tag{6.22}$$

and therefore, because $\sqrt{\cdot}$ is convex, minimizing $||\vec{a} - \vec{b}||$ over all choices $\vec{b}$ also minimizes $1 - \vec{a} \cdot \vec{b}$. Furthermore, it also minimizes $1 - |\vec{a} \cdot \vec{b}|$, as if $\vec{a} \cdot \vec{b} < 0$, then $1 - \vec{a} \cdot \vec{-b} < 1$ while $1 - \vec{a} \cdot \vec{b} > 1$ so it cannot be minimal, which is a contraction and thus we have $\vec{a} \cdot \vec{b} \geq 0$ and $1 - \vec{a} \cdot \vec{b} = 1 - |\vec{a} \cdot \vec{b}|$.

Now assume that $||\text{vec}(A)|| = 1$ and let

$$B = \alpha_1 B_1 + \alpha_2 B_2 + \cdots + \alpha_k B_k \tag{6.23}$$

for some k and coefficients $\vec{\alpha}$ such that $||\text{vec}(B)|| = 1$. Equivalently, let us write that $\text{vec}(B) = B' \vec{\alpha}$ where B' is the matrix such that the ith column is $\text{vec}(B_i)$.

Then to maximize $\mathcal{F}(A,B)$ over all choices of α_i it is sufficient to minimize $1 - \mathcal{F}(A,B) = 1 - |\text{vec}(A) \cdot \text{vec}(B)|$, which is equivalent to minimizing $||\text{vec}(A) - \text{vec}(B)|| = ||\text{vec}(A) - B'\vec{\alpha}||$. This transforms the problem of maximizing fidelity into an ordinary least squares instance, which can be solved by a QR decomposition.

Definition 6.6. *Given a matrix M over $\mathbb{C}$ with n rows and m columns, a (reduced) QR decomposition of M is given by $M = QR$ such that Q is an orthogonal matrix (i.e $Q^TQ = I$) with n rows and m columns and R is an upper triangular square matrix with m rows and columns.*

It can be shown [Wei] by vector calculus that the minimal solution to $||\text{vec}(A) - B'\vec{\alpha}||$ is given analytically by $(B')^T B\vec{\alpha} = (B')^T\text{vec}(A)$ (this is known as the normal equation). To solve this, we can substitute $B' = QR$ and note that $(B')^T B\vec{\alpha} = R^TQ^TQR\vec{\alpha} = R^TR\vec{\alpha} = (B')^T\text{vec}(A) = R^TQ^T\text{vec}(A)$, and therefore we have that $\vec{\alpha} = R^{-1}Q^T\text{vec}(A)$. This implies that $\text{vec}(B) = B'\vec{\alpha} = QRR^{-1}Q^T\text{vec}(A) = QQ^T\text{vec}(A)$, and so the maximal fidelity of A and B over all choices of α_i is given by:

$$\begin{aligned} \mathcal{F}(A,B) &= |\text{vec}(A) \cdot \text{vec}(B)| = |\text{vec}(A) \cdot QQ^T\text{vec}(A)| \\ &= |\text{vec}(A)^T QQ^T\text{vec}(A)| = |\text{vec}(A)^T QQ^TQQ^T\text{vec}(A)| \\ &= ||QQ^T\text{vec}(A)||^2 \end{aligned} \tag{6.24}$$

Therefore, in order to search for a stabilizer decomposition of size k, we need only search for a set of k StabZX-states which can be combined linearly to form the target state, not for the specific coefficients of the combination. One method for this, published by Bravyi, Smith, and Smolin in 2016 [BSS16], is to use the equivalence between StabZX-states and Pauli exponentials given above to perform a random walk over all possible combination - in particular, they used the method of simulated annealing [KGV83], which is detailed below. This method keeps track of a parameter $T \in \mathbb{R}$, known as the temperature, and performs a random walk over the state space. When T is high, a step of the walk is chosen that may increase or decrease the target function, but as T decreases, the probability of choosing a step which decreases the target function is lowered. Assuming T decreases sufficiently slowly over the course of the walk, it has been shown that this process converges to the global optimum when the target function is sufficiently smooth. Since our search space is discrete, this guarantee does not apply, but the method appears to work well in practice regardless.

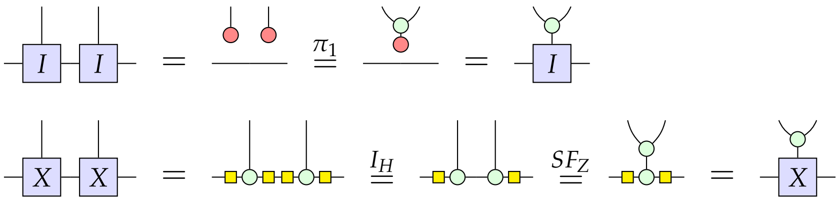
Definition 6.7 ([BSS16, Appendix A])**.** *Given some target stabilizer state S with n outputs and $k \in \mathbb{Z}$, the following algorithm, called simulated annealing, attempts to find a stabilizer decomposition of S into k terms. It is parameterized by a decreasing sequence $T_0, T_1, \dots, T_m \in \mathbb{R}$ which specifies the cooling schedule, as well as a function* $\mathrm{accept}(f_{prev}, f_{proposed}, T) : \mathbb{R}^3 \to [0,1]$ *which gives the probability that a step of the walk will be accepted at the given temperature.*

1. *Initialize k stabilizer states $U_1, U_2, \dots U_k$ to some fixed value. Let $U = \alpha_1 U_1 + \alpha_2 U_2 + \dots + \alpha_k U_k$ throughout.*
2. *Set the temperature as $T = T_0$, and let $s = 0$.*
3. *Compute $f_{prev} = \mathcal{F}(S, U)$ as the maximum fidelity over all choices of α_j.*
4. *Pick a random integer $1 \leq i \leq k$ and save the value of U_i as U'.*
5. *Choose a Pauli string $\vec{P}$ of size n uniformly at random, and apply the Pauli exponential $\vec{P}(\frac{\pi}{2})$ to U_i.*
6. *Compute $f_{proposed} = \mathcal{F}(S, U)$ as the maximum fidelity over all choices of α_j. If $f_{proposed} = 1$, then we are done, and can return U.*
7. *Pick $r \in [0, 1)$ uniformly randomly. If $r <$ $\mathrm{accept}(f_{prev}, f_{proposed}, T_s)$, set $f_{prev} = f_{proposed}$, otherwise set $U_i = U'$.*
8. *Update $s = s + 1$, and if $s = m$ we are done, otherwise go to step four.*

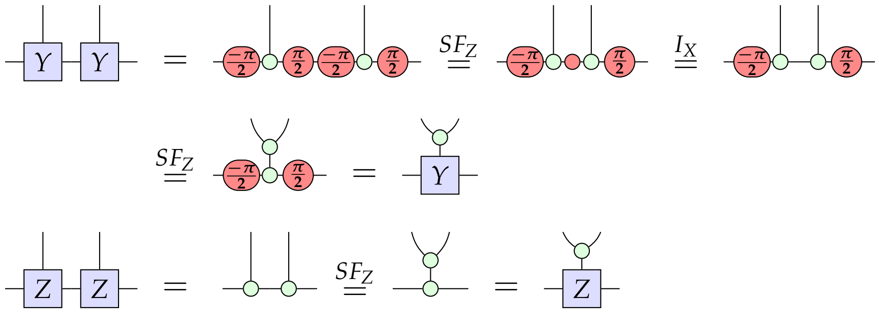
In order to converge, every point in the state space must be accessible from every other point in the state space by some series of steps. There are two potential obstacles to this - firstly, we fix the angle of the Pauli exponentials applied to be $\frac{\pi}{2}$, and secondly, it may be that applying a particular Pauli exponential cuts down the accessible search space. Thankfully, both of these can be alleviated with the following lemma.

Lemma 6.4 ([Kv22, 5.3.2])**.** *We have that $\vec{P}(\theta_1) \circ \vec{P}(\theta_2) = \vec{P}(\theta_1 + \theta_2)$ for any Pauli string $\vec{P}$ and angles $\theta_1, \theta_2 \in \mathbb{R}$.*

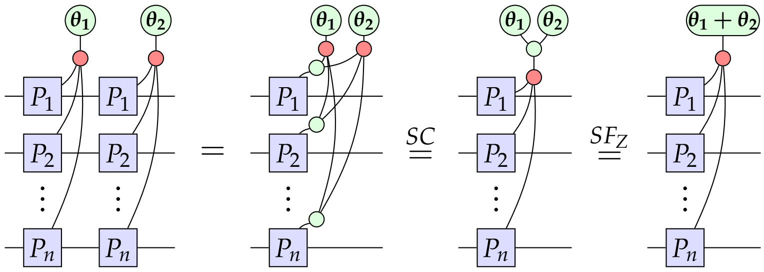
Proof. First, note that we can combine any adjacent Pauli boxes of the same type, up to a scalar factor:



(6.25)



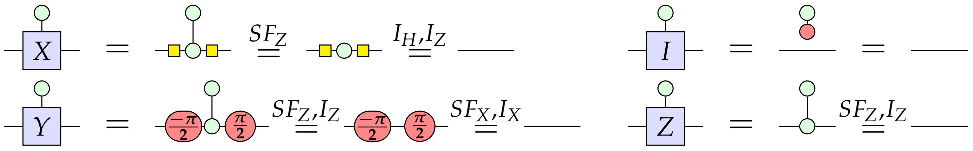
With this, we can see the following



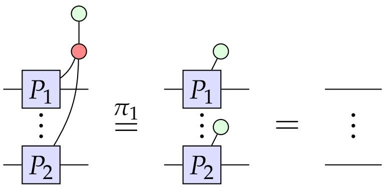
(6.26)

which completes the proof. □

Firstly, this implies that we can form any $\vec{P}(\theta)$ for $\theta = k\frac{\pi}{2}$ with $k \in \mathbb{Z}$, since $\vec{P}(\theta) = \vec{P}(\frac{\pi}{2})^{\circ k}$ for $k \geq 0$ (which we can always assume as angles are considered modulo 2π). Secondly, this shows that the Pauli exponentials form a group if we take the identity to be the identity tensor on n inputs and outputs, since we have $\vec{P}(\theta) \circ \vec{P}(-\theta) = \vec{P}(0)$ and $\vec{P}(0) \circ \vec{P}(\theta) = \vec{P}(\theta)$, and we can see that for any $\vec{P}$, $\vec{P}(0)$ is the identity:



(6.27)



Because of this, any stabilizer diagram is reachable from any other stabilizer diagram by a sequence of Pauli exponentials.

In terms of implementation, the stabilizer states U_i are kept in memory as their vectorized forms, $\text{vec}(U_i)$, so that the QR decomposition of U' can be computed directly, and therefore $O(k2^n)$ memory is required to operate the algorithm. To apply Pauli operators when the states are stored in this form, it has been shown that is possible to write $\text{vec}(\vec{P}(\theta) \circ S) = \cos\theta\text{vec}(S) + \sin\theta M\text{vec}(S)$, where $M = PD$ is the product of a permutation matrix and a diagonal matrix. Therefore, each application of a Pauli product costs only $O(2^n)$ time, and the dominant step of the algorithm is the QR decomposition for finding the maximum fidelity, which can be done in $O(k^2 2^n)$ time [TI97, Lecture 10], so the algorithm executes in $O(mk^2 2^n)$ time total. In principle, this can be reduced to $O(k^2 2^n + mk2^n)$ by making use of specialized algorithms for updating QR decompositions [HL08] when a column is changed, but this is quite complicated and so was not used for this project.

The author's implementation of this algorithm is available online [The], under a permissive open-source license and integrates with the popular Python package NumPy to allow stabilizer decompositions to be found for arbitrary states in vectorized form.

6.2 Other Search Methods

In addition to simulated annealing, several other methods were tried to pick suitable stabilizer states $U_1, U_2, \ldots, U_k$ so that a stabilizer decomposition of size k can be found by maximizing the fidelity between $U = \alpha_1 U_1 + \cdots + \alpha_k U_k$ and the target state S. In particular, we will detail two strategies which were implemented: stabilizer testing for cancellation, and a genetic algorithm approach.

6.2.1 Genetic Algorithms

Given any stabilizer state S with n outputs, we can reduce it to GSLC form described above. This form can be uniquely specified using $\frac{n^2+5n}{2}$ binary variables: $\frac{n(n-1)}{2}$ variables are required to specify the adjacency matrix of which Z-spiders are connected to which other Z-spiders via Hadamard edges, $2n$ variables can be used to specify the phase of each Z-spider (since the phase must be one of $\{0, \frac{\pi}{2}, \pi, \frac{-\pi}{2}$, a two-bit number suffices to encode this), and the remaining n variables specify whether each Z-spider is connected to its output by a Hadamard gate or just an edge.

Therefore, for a vector of size $k\frac{n^2+5n}{2}$ with binary entries, we can define an objective function given by interpreting this as k stabilizer states in GSLC form, and finding the maximal fidelity between the sum of these states and some target state. We can then use any generic high-dimensional discrete optimization algorithm to try and find stabilizer decompositions of size k by maximizing this objective function.

Since the number of dimensions is so high for any practical application, genetic algorithms [KCK21] are one of the only algorithms equipped to deal with such problems. This was implemented using the PyGAD Python library [Gad20], but in practice never converged, even when a stabilizer decomposition of the requested size was known to exist. As such, this method will not be considered further in this work - it seems that the maximum fidelity may be a bad metric for this style of optimization, and more tuning is required.

6.2.2 Stabilizer Testing for Cancellation

Suppose that the states A and B have stabilizer decompositions into n and m terms given by $A = \alpha_1 A_1 + \cdots + \alpha_n A_n$ and $B = \beta_1 B_1 + \cdots + \beta_m B_m$, then $A \otimes B$ has a stabilizer decomposition into nm terms given by $A \otimes B = \alpha_1\beta_1 A_1 \otimes B_1 + \alpha_1\beta_2 A_1 \otimes B_2 + \cdots + \alpha_n\beta_m A_n \otimes B_m$. However, it may be the case that some of these terms cancel out - we may have that $\alpha_i\beta_j A_i \otimes B_j + \alpha_k\beta_l A_k \otimes B_l$ is actually a stabilizer state, and so there is a decomposition of $A \otimes B$ into $nm - 1$ terms. For instance, this was noted in practice by Kocia [Koc22].

However, this technique can be extended by applying one stabilizer decomposition but using a different stabilizer decomposition of the remaining non-stabilizer part of the diagram in each of the terms. For instance, suppose A has a stabilizer decomposition of size n given by $\alpha_1 A_1 + \cdots + \alpha_n A_n$, and B has n (not necessarily unique) stabilizer decompositions of size $m_1, m_2, \ldots, m_n$ given by $B = \beta_{i1} B_{i1} + \cdots + \beta_{im_i} B_{im_i}$ for all $1 \leq i \leq n$. Then there is a stabilizer decomposition of $A \otimes B$ into $m_1 + m_2 + \cdots + m_n$ terms given by

$$\begin{aligned} A \otimes B = \alpha_1\beta_{11} A_1 \otimes B_{11} + \cdots + \alpha_1\beta_{1m_1} A_1 \otimes B_{1m_1} + \cdots \\ + \alpha_n\beta_{n1} A_n \otimes B_{n1} + \cdots + \alpha_n\beta_{nm_n} A_n \otimes B_{nm_n} \end{aligned} \tag{6.28}$$

and similarly to before, it may be the case that some pairs of these terms cancel out and can be replaced by their sum.

This technique was used to search for stabilizer decompositions of tensor products of a state with itself. We shall use $S^{\otimes n}$ to refer to a state S tensored with itself n times. Given a state S we proceed as follows:

1. Start with a list of decompositions of $S^{\otimes 1}$.

2. Randomly pick small integers i and j and a decomposition of $S^{\otimes i}$ into n terms. Further pick n decompositions of $S^{\otimes j}$ randomly with replacement, and create a decomposition of $S^{\otimes (i+j)}$ as above.

3. For each pair of terms in this new decomposition, check if their sum is actually a stabilizer state. If it is, replace these terms with their sum. Repeat until no terms can be further combined.

4. If any terms were removed from this decomposition, add it to the list of decompositions.

5. Return to step two.

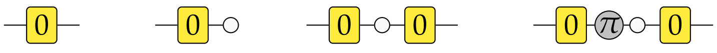
This process can be terminated when decompositions of large enough tensor products have been considered.

In order to test whether a sum of two stabilizer states is also a stabilizer state, we can find the vectorization of the sum and use the previous simulated annealing method to try and find a stabilizer decomposition of size one. Many optimizations are available in this case - the maximum fidelity is given by the dot product of the two states, and so the QR decomposition of B' need not be computed at all - and so this test can be done extremely quickly.

Another option that was tested is to use an algorithm dedicated to testing if a state is a stabilizer state, such as those presented Damanik [Dam18]. Since these algorithms are designed to be run on quantum computers, this was implemented using the IBM Qiskit toolkit [GTN$^+$21] to run a simulation of a quantum computer.

6.3 Decompositions Found

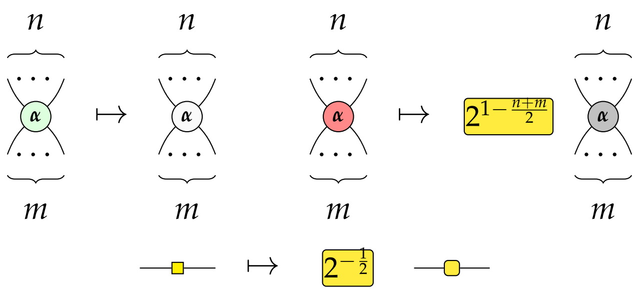
In the next chapter, we will apply stabilizer decompositions to ZH-diagrams for #kSAT instances. As such, we are interested in finding stabilizer decompositions of the following four tensors



(6.29)

which we will call $H_1^1[0]$, H_{state}, H_{link}^+, and H_{link}^- respectively, as well as tensor products of these with themselves.

Note that so far, we have only considered stabilizer decompositions of ZX-diagrams, as stabilizer diagrams are defined as ZX-diagrams. However, we can define an equivalent fragment StabZH of the ZH calculus, by noting the following embedding of StabZX up to scalar factors:



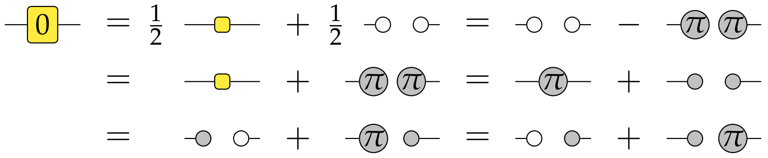
(6.30)

In this way, we can see that Eval(StabZH) is also computable in polynomial time, and all the rewriting rules for StabZX apply to StabZH up to scalar factors since the ZX calculus is sound.

Since our methods for finding stabilizer decompositions require only the vectorization of the underlying tensor to work, the fact that these are defined by ZH-diagrams is irrelevant, and we can translate the decompositions back in ZH-diagrams by adjusting the scalar factors as above. Note also that we can transform any diagram into a state by applying cups to the inputs, and because of the yanking equations, applying caps to the corresponding outputs on the terms of the decomposition will yield a decomposition for the original diagram.

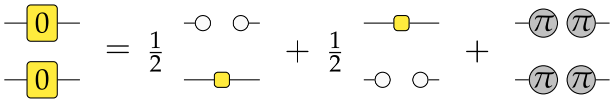
6.3.1 Stabilizer Testing for Cancellation

This method was used to find decompositions for $H_1^1[0]^{\otimes 2}$ and $H_1^1[0]^{\otimes 3}$. The algorithm was initialized with the following six decompositions of $H_1^1[0]$, which were found by manually inspecting the tensor:

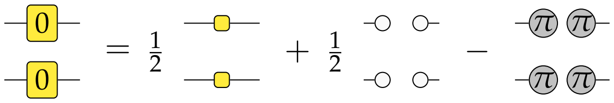


(6.31)

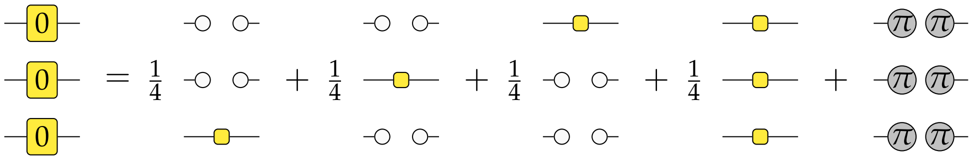
Using the simulated annealing method to test whether sums of states are stabilizers, the algorithm was run for several hundred iterations, and the following decompositions were found:



(6.32)



(6.33)



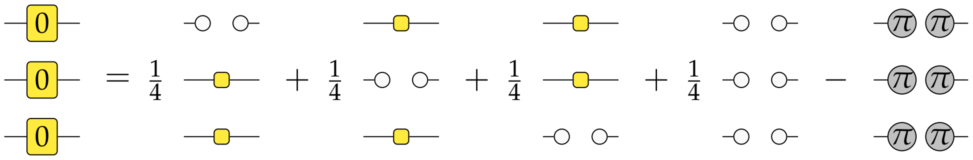
(6.34)

Whilst discovered automatically by this method, decomposition (6.33) was also previously found manually by John van de Wetering.

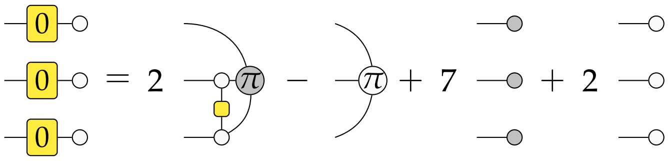
6.3.2 Simulated Annealing

Bravyi, Smith, and Smolin [BSS16, Appendix A] suggest the following parameters for the simulated annealing algorithm: the cooling schedule is defined by

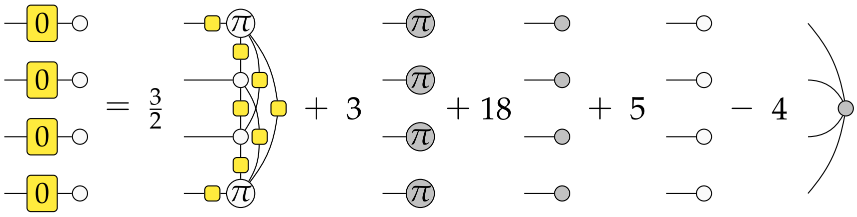
$T_0 = 1$ and $T_m = \frac{1}{4000}$ with the remaining steps interpolated by a geometric sequence as $T_i = (T_m)^{\frac{i}{m}}$, where $m = 100000$, and the probability of accepting a step on the walk is given by $\text{accept}(f_{prev}, f_{proposed}, T) = e^{-\frac{f_{prev}-f_{proposed}}{T}}$. Except for m, which was increased to 10^6, these were left unchanged. Using this method, we obtained the following decompositions for the diagrams $H_1^1[0]^{\otimes 3}$, $H_{state}^{\otimes 3}$ and $H_{state}^{\otimes 4}$, and $(H_{link}^+)^{\otimes 2}$ and $(H_{link}^-)^{\otimes 2}$:



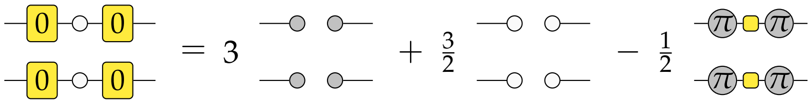
(6.35)



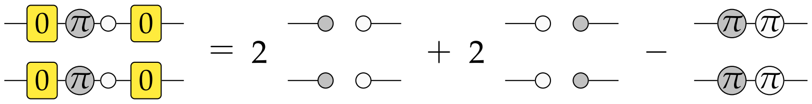
(6.36)



(6.37)



(6.38)



(6.39)

In each case, the ZH-diagram corresponding to each term was obtained from their vectorizations by utilizing the genetic algorithm method described above, since it operates directly on GSLC form ZX-diagrams. Indeed this use case is the only time when the method converged.

CHAPTER 7

Solving #2SAT with Decompositions

This chapter extends an algorithm of Kissinger and van de Wetering [Kv21] [KvV22] to give a novel method for solving #2SAT, in an approach suggested by John van de Wetering and Konstantinos Meichanetzidis. The expected runtime of several variations on this algorithm are then analyzed and implemented.

Previously, in chapter five examining the DPLL algorithm, we saw that writing a #SAT diagram as a sum of other diagrams and applying simplification works well both theoretically and in practice. A similar technique has previously also been used to great success for evaluating ZX-diagrams that exist in a certain fragment of the ZX calculus known as Clifford+T. This had previously been explored indirectly by Bravyi, Smith, and Smolin [BSS16], and further by Qassim et al [QPG21]. However, it was first presented in this form by Aleks Kissinger and John van de Wetering [Kv21] [KvV22], and recently improved by Julien Codsi [Cod22]. The technique consists of alternating between replacing a non-stabilizer part of the diagram with its stabilizer decomposition, and applying a simplification routine known as Clifford simplification.

Definition 7.1 ([Kv21, 2.3])**.** *The Clifford simplification routine for ZX-diagrams is as follows:*

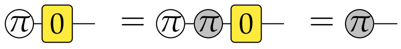
1. *Ensure that the diagram is in graph-like form.*
2. *Apply the spider fusion rule SF_Z and identity rules I_Z and I_H as much as possible.*
3. *Apply local complementation simplification, Lemma 6.1, and pivot simplification, Lemma 6.2, wherever possible.*
4. *Apply boundary pivoting, Lemma 6.3, wherever doing so would not increase the size of the diagram.*
5. *If any simplification occurred, return to step two, otherwise terminate.*

While in practice, the worst-case time bound of this algorithm is the same bound as found by Qassim et al [QPG21] without diagrammatic simplification routines, this interleaving routine appears to work well in practice, saving orders of magnitude of time compared to no simplification. As such, we will attempt to apply a similar method to #2SAT diagrams, using the stabilizer decompositions developed in the last chapter, and the rewriting rules presented earlier for #2SAT. This yields an algorithm for $\text{Eval}(\text{ZH}_\pi^0)$ which we will call #Decompose. The author acknowledges John van de Wetering and Konstantinos Meichanetzidis for suggesting this approach. #Decompose is defined in much the same way as $\#DPLL$, and is given as follows.

Definition 7.2. *Given a* ZH_π^0*-diagram* D*, the algorithm* #Decompose *is given as follows:*

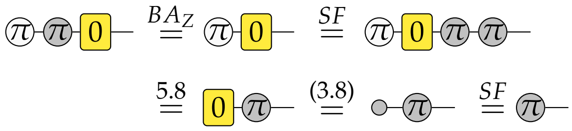
1. *Apply the Clifford simplification routine, adapted to ZH-diagrams via additional scalar factors described in the previous chapter.*
2. *Apply the rewriting rules given in Lemmas 5.4, 5.5, and 5.6, and Lemma 7.1 as much as possible.*
3. *Repeat the previous two steps until no further simplification occurs.*
4. *If the diagram is empty, or consists of just scalars, multiply them together and return this. If the diagram has a zero scalar anywhere, then return zero.*
5. *If* D *consists of disconnected components* $D = D_1 \otimes \cdots \otimes D_k$*, then recursively evaluate* $c_1 = \text{\#Decompose}(D_1)$*, ...,* $c_k = \text{\#Decompose}(D_k)$ *and return* $c_1 \ldots c_k$*.*
6. *If the diagram only contains one* $H_1^1[0]$*-box, then apply one of the decompositions given in* (6.31) *to write* $D = D_1 + D_2$*.*
7. *If it is possible to apply one of the decompositions* (6.38), (6.39), (6.37), *or* (6.36), *in this order of preference, pick a random place to apply it and write* $D = D_1 + \cdots + D_k$*.*
8. *Otherwise, pick a decomposition from* (6.32), (6.33), (6.35), *or* (6.34) *and a place to apply it using some strategy, then write* $D = D_1 + \cdots + D_k$*.*
9. *Recursively evaluate* $c_1 = \text{\#Decompose}(D_1)$*,* $c_2 = \text{\#Decompose}(D_2)$*, ...,* $c_k = \text{\#Decompose}(D_k)$*, and return* $c_1 + c_2 + \cdots + c_k$*.*

Lemma 7.1. *The following rewrite for* ZH_π^0 *diagrams holds:*



(7.1)

Proof. We have that



(7.2)

which completes the proof. □

7.1 Upper Bounds and Limiting Behaviour

Note that we have left the choice of which decomposition to apply and where to apply it undefined in step eight. In this section, we will examine several strategies for choosing decompositions and their expected behaviour in the limiting case of small or large density diagrams. First we will put an upper bound on the runtime of the whole algorithm, independent of any selection strategy. Unlike variants of #*DPLL*, #Decompose is designed to decompose clauses, not variables. As such, we will analyze the scaling of the algorithm in terms of m.

Theorem 7.1. *The algorithm* #Decompose *has a worst-case runtime upper bound of* $O(\text{poly}(n, m)2^{0.7740m})$.

Proof. First note that any decomposition that removes k $H[0]_1^1$-boxes removes k clauses since each clause consists of a $H[0]_1^1$-box and some stabilizer parts of the diagram, which will be removed by Clifford simplification after the $H_1^1[0]$-box is removed. Therefore, for each step, we can establish a polynomial time bound or a branching number:

1. The first three steps run in polynomial time, since each of the rewriting rules can be applied in time $O(\text{poly}(n, m))$, and there can be at most $O(\text{poly}(n, m))$ applicable instances before either no more simplification can be performed or the diagram is empty.

2. The fourth step takes $O(1)$ time.

3. As in #*DPLL*, component analysis in the fifth step cannot increase the runtime of the algorithm.

4. If the sixth step applies, then the diagram will be evaluated as the sum of two stabilizer diagrams, which will be eliminated in polynomial time by the first step of the algorithm, and thus the whole step takes polynomial time.

5. If the seventh step applies, we have one of the following branching numbers: $\tau(4,4,4)$, $\tau(4,4,4,4,4)$, or $\tau(3,3,3,3)$. The maximum of these is given by $\tau(3,3,3,3) = 2^{\frac{2}{3}} \approx 1.5874$.

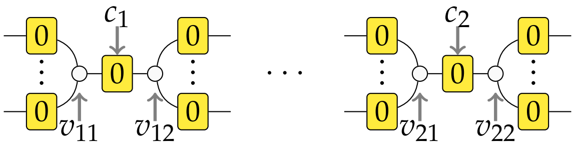
6. Otherwise if the eighth step applies, we can pick decomposition (6.35) or (6.34) to obtain a branching number of $\tau(3,3,3,3,3) = 5^{\frac{1}{3}} \approx 1.7100$.

Therefore, we have $\alpha = \max\{\log_2 \tau(3,3,3,3), \log_2 \tau(3,3,3,3,3)\} \leq 0.7740$. Note that in the cases for steps seven and eight, we ignored any cancellation that might occur as a result of applying the decompositions. If this occurs it can only decrease the runtime, meaning we have obtained an upper bound. □

While in the above proof we assumed that no cancellation occurs when applying stabilizer decompositions, this cancellation is the whole motivation for applying this method. Therefore, we should aim to pick a strategy for applying these decompositions that maximizes the amount of cancellation, at least in the average case. We present three strategies for this: applying decompositions to clauses connected to vertices of maximum degree, applying decompositions clustered at low-degree vertices, and applying decompositions to a frontier to maximize Clifford simplification.

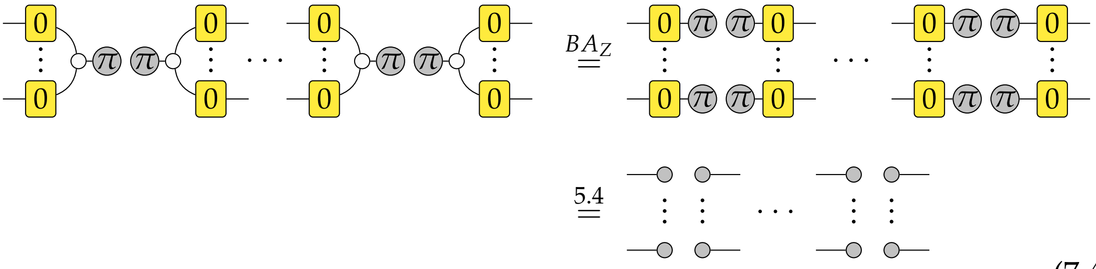
7.1.1 Clauses with High Degree Variables

Given some #2SAT-diagram, suppose we could pick two $H_1^1[0]$-boxes, c_1 and c_2, that were seperated far apart in the diagram, and are each connected (we will assume that all $H_1^1[0]$-boxes are not connected through NOT gates without loss of generality), to two Z-spiders each with high degrees, v_{11}, v_{12}, v_{21}, and v_{22}, as below:



(7.3)

Then if we apply the decomposition (6.32) or (6.33), no cancellation will occur in the first and second terms, but in the third term, we see the following:



(7.4)

This removes the two clauses c_1 and c_2, but also all the other neighbouring clauses of v_{ij}. Therefore, if v_{ij} has degree Δ_{ij}, we will remove $2 + \Delta$ clauses in total, where $\Delta = \sum_{ij}(\Delta_{ij} - 1)$. Therefore, this decomposition has a branching number of $\tau(2, 2, 2 + \Delta)$. If we performed the same analysis for three $H_1^1[0]$-boxes using decomposition (6.35) or (6.34), then we would find a branching number of $\tau(3, 3, 3, 3, 3 + \Delta)$ (since similarly, cancellation only happens in the last term).

If we assume $\Delta = 0$ (i.e the worst case upper bound, as above), then we obtain $\tau(3, 3, 3, 3, 3) = 1.7100 < \tau(2, 2, 2) = 1.7321$. Interestingly, however, we have:

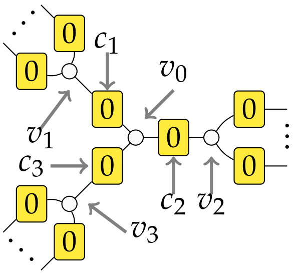
$$\begin{aligned} \lim_{\Delta\to\infty} \tau(2, 2, 2 + \Delta) &= \sqrt{2} \approx 1.4142 \\ \lim_{\Delta\to\infty} \tau(3, 3, 3, 3, 3 + \Delta) &= \sqrt[3]{4} \approx 1.5874 \end{aligned} \tag{7.5}$$

And in fact, $\tau(2, 2, 2 + \Delta) < \tau(3, 3, 3, 3, 3 + \Delta)$ for all $\Delta \geq 1$. Therefore, if we pick c_1 and c_2 such that Δ is maximized, we would actually expect the decompositions (6.32) and (6.33) to perform better in almost all cases.

Ideally, we could assume $\Delta \geq 2$, since when $\Delta \leq 1$, we can apply the decompositions (6.36) and (6.37), which have better branching numbers and use this to obtain a lower upper bound in general. While this is true for a #2SAT-diagram which we assumed above, this decomposition does not preserve the structure of the diagram like variable branching does in $\#DPLL$, and so we may have the situation where some of the neighbours of v_{ij} are not $H_1^1[0]$-boxes, but in fact stabilizer parts of the diagram introduced by previous decompositions. Therefore, this upper bound does not apply to #Decompose. In practice, however, this method may still be effective.

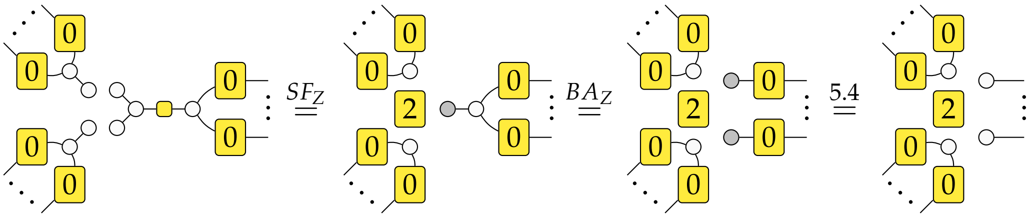
7.1.2 Eliminating Low Degree Variables

While in the previous section, we decomposed clauses that were far apart in order to exploit unit propagation as much as possible, we can instead decompose clauses that are maximally close together to exploit Clifford simplification. Specifically, suppose we have four variables $v_{0,1,2,3}$ and three clauses $c_{1,2,3}$ in the following arrangement:



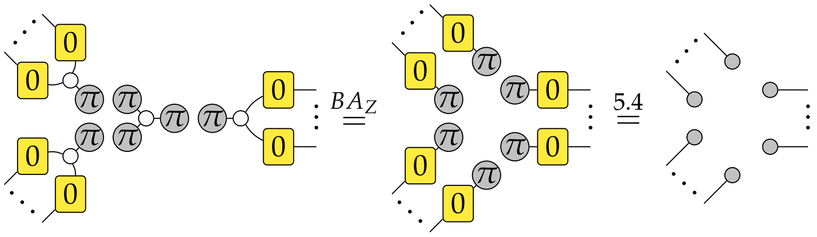
(7.6)

Then by applying decomposition (6.34), we observe the following cancellation. In the first three terms, we obtain the following (up to a rotation):



(7.7)

In the fourth term, no cancellation occurs, but in the fifth term, we have the following (throughout, we have assumed that all $H_1^1[0]$-boxes are connected to Z-spiders directly, but the cancellation is maintained even when this is not the case):



(7.8)

Therefore, if we assume that v_i has degree Δ_i, then the branching number for this case is given by $\tau(2+\Delta_1, 2+\Delta_2, 2+\Delta_3, 3, \Delta_1+\Delta_2+\Delta_3)$. Assuming that $\Delta_i \to \infty$, we find that

$$\tau(2+\Delta_1, 2+\Delta_2, 2+\Delta_3, 3, \Delta_1+\Delta_2+\Delta_3) \to 1 \tag{7.9}$$

and consequently $\alpha \to 0$, which would imply that higher densities become easier, exactly as we saw for $\#DPLL$. However, we can't assume that this will hold up in practice - increasing δ will, on average, increase Δ_i, but will also decrease the likelihood that a variable with degree three exists in the first place. Therefore, in practice, we need to pair this with one of the other two strategies to handle this case. Additionally, it is possible to perform a similar analysis for variables of degree two (or two variables of degree one) with decomposition (6.32).

7.1.3 Maximizing Clifford Simplification

Finally, we will consider the case of optimizing decomposition placement to maximize the amount of Clifford simplification taking place, since that is the main novelty of this method. While the previous method did utilize Clifford simplification, it did not make use of the fact that previous decompositions may leave stabilizer parts to the diagram that can be combined with another decomposition to yield something that is simplifiable.

In particular, notice that the major Clifford simplification rules, Lemmas 6.1 and 6.2, do not apply unless the stabilizer part of the diagram is quite large (it must contain at least three Z-spiders connected by Hadamard gates). Therefore, it may be helpful to try and cluster decompositions close together to make larger stabilizer regions in the diagram. One strategy for this is as follows:

- Maintain a list of all variables whose clauses have been decomposed thus far. Initially, this is empty.
- In order to select a decomposition, pick three random clauses that are connected to a variable in the list.
- If the list is empty, pick randomly from all clauses.
- If the diagram has been split into disconnected components and the component being considered does not have any variables on the list, pick clauses randomly in this component.
- Find the variables connected to these clauses and add them to the list if they hadn't been already.

Therefore, during operation, we start with some random 'seed' clauses and grow a frontier of decompositions around these. This results in an expanding stabilizer part to the diagram, inside which simplifications can occur. While this may seem worse than the previous strategy, as that guarantees simplification, this instead permits simplification across the whole frontier, and not just localized to one Z-spider, which may be beneficial - we have given three different strategies that are all optimized for particular situations, and so in practice, some combination of the three may be better than any one alone.

7.2 Practical Implementation

Now that we have a complete algorithm, we will see how it performs empirically. However, this testing does not focus on performance compared to existing algorithms for this problem, but rather on the relative efficacy of the different strategies described above, as well as the limits of what can be expected from this method. Simply put, this is because this method is much slower than existing algorithms - even the most basic forms of $\#DPLL$ like the CDP algorithm of Birnbaum and Lozinskii [BL99]. We will discuss in the next section why this is, and what might be done to improve the situation.

The algorithm was implemented in the Rust programming language and is built on top of QuiZX, a library for representing and simplifying ZX-diagrams.

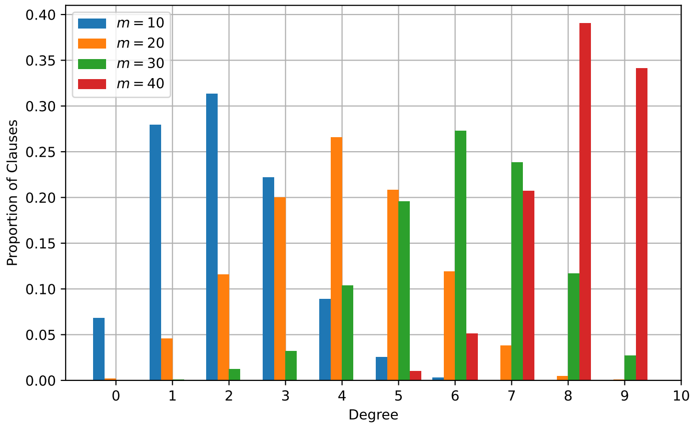


Figure 7.1: The degree distribution of random formulae generated by first picking a random graph with a fixed number of edges. Generated from 200 samples with $n = 10$ variables and $m = 10, 20, 30, 40$.

This was adapted for ZH-calculus by adding a new type of node to diagrams that represents an $H_1^1[0]$-box. The algorithm #Decompose was implemented as specified above, except that step seven (the application of the decompositions (6.38), (6.39), (6.36), and (6.37)) was removed, to make it easier to see the effects of changing the decomposition selection strategy.

Similarly to the phase-transition measurements from chapter five, the number of recursive calls (which we call the number of terms) made by the algorithm was measured in lieu of execution time, since time measurements are inherently non-deterministic and prone to noise. Random #2SAT instances with m clauses and n variables were generated by picking m distinct pairs of variables uniformly at random (i.e a random graph with m edges), and then assigning each variable of the pair a uniformly random polarity (either negated or not) to produce clauses. To aid in reproducibility, samples were generated in this manner for $n = 10$ and $m = 10, 20, 30, 40$, and the distribution of degrees is presented in Figure 7.1.

Note that the rest of the plots will make extensive use of the violinplot style - in this style, each set of samples is represented by a partially transparent symmetrical region aligned with the x-axis value of the measurement. The width of the region indicates the density of samples at that y-axis value, and the marker bars indicate the median sample. All data will be generated by taking 200 random samples as described above for every pair of m and n.

First, we compare the difference between the strategy of decomposing clauses at high-degree variables with two clauses and three clauses per decomposition. As discussed, we would expect the growth rate to converge to $O(\text{poly}(n, m)\sqrt{2}^m)$

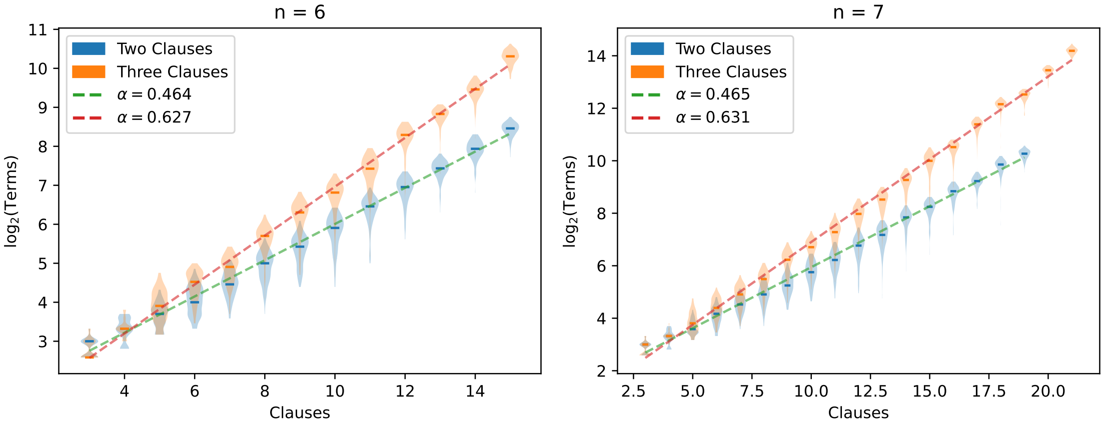


Figure 7.2: The number of terms needed to evaluate #2SAT diagrams by decomposing either two or three clauses connected to high-degree variables. Linear fits with corresponding α-values are given.

and $O(\text{poly}(n,m)\sqrt[3]{4}^m)$ for two and three clauses, respectively, as m (and thus Δ) increases. This corresponds to α-values of $\log_2(\sqrt{2}) = \frac{1}{2}$ and $\log_2(\sqrt[3]{4}) = \frac{2}{3}$. We measure these α-values directly by examining the gradient of a linear fit of the base-2 logarithm of the number of terms against the number of clauses. Results are presented for $n = 6$ and $n = 7$ in Figure 7.2.

Indeed, we can see that the measured α-values of $\alpha = 0.46$ for the two clause decomposition and $\alpha = 0.63$ for the three clause decomposition over the whole range are close to the expected values but slightly lower. This is almost certainly because of cascading unit propagation, due to its independence from the number of clauses. In order to observe the convergence, the α-value of the first decomposition for each instance was also calculated exactly through empirical calculations of the branching number, given how many clauses were cancelled in each term. These results are presented in Figure 7.3 in terms of the density.

We can see that the α-values converge quickly to the expected values. Note that we would expect larger α for low δ, and indeed many samples do have this, however, there appears to be a bimodal distribution of α-values at low δ. One possible explanation for the second, lower, mode is that at low δ, it is more likely for the graph to be disconnected, so when this is the case, the α-value is much lower.

Moving on to the strategy of optimizing Clifford simplification by placing decompositions close to previous decompositions, the number of terms needed to solve #2SAT instances with $n = 6$ and $n = 7$ was measured and compared to the previous strategy of decomposing highly-connected clauses. The results are presented in Figure 7.4. We can see that the Clifford simplification strategy

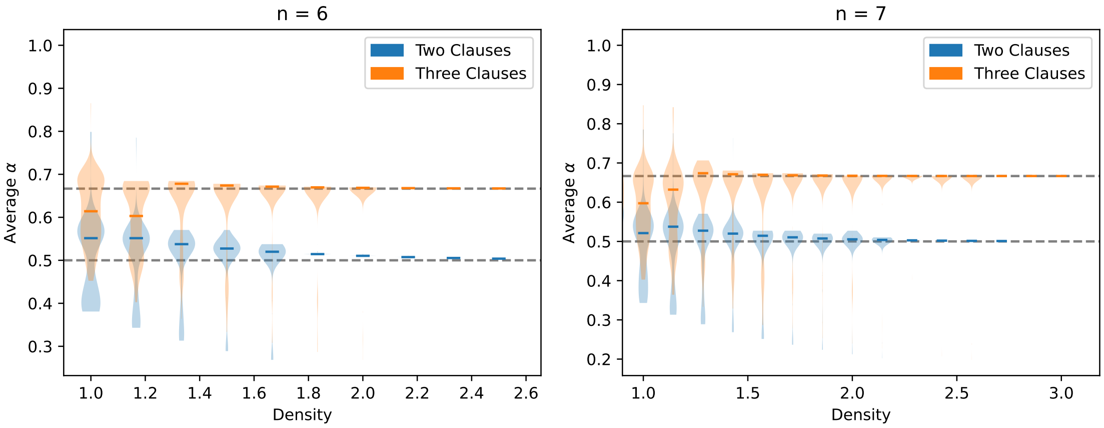


Figure 7.3: Average α value of the first decomposition of a diagram evaluated by decomposing two or three highly connected clauses.

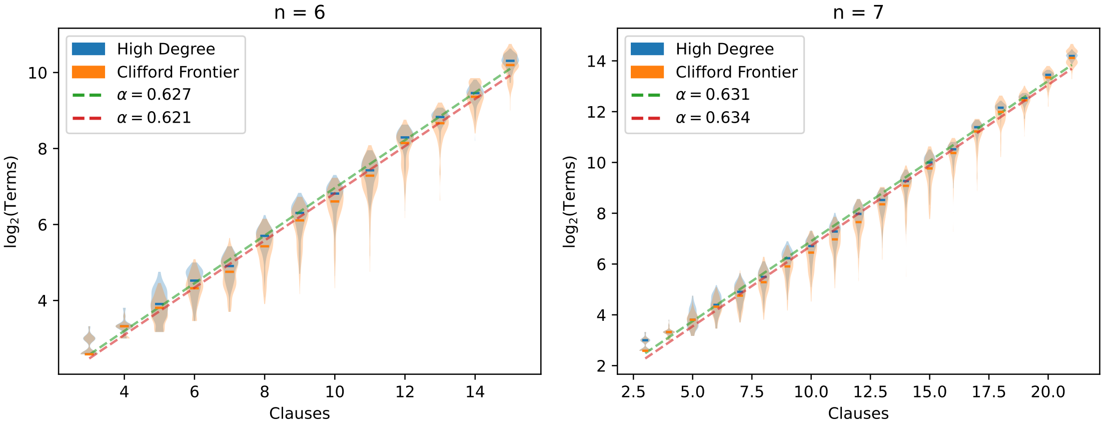


Figure 7.4: The number of terms needed to evaluate #2SAT diagrams by decomposing three clauses near previous decompositions vs three clauses connected to high-degree variables. Linear fits with corresponding α-values are given.

appears to be largely identical, with a measured α-value of 0.63 as before. The median number of terms is slightly lower across the range, but the spread of samples significantly overlaps, so we cannot say conclusively which is better.

In order to evaluate whether this strategy is successful at introducing Clifford simplification, the average proportion of clauses that were simplified out after two decompositions was measured for both strategies. Since two decompositions is the minimum number before Clifford simplification can occur, we would expect to see some simplification if the Clifford simplification strategy is effective. This was measured for $n = 7$ and $n = 8$ and is presented in Figure 7.5.

We can see that indeed at low densities, some extra simplification occurs with the Clifford strategy, although the spread is very wide and overlaps significantly.

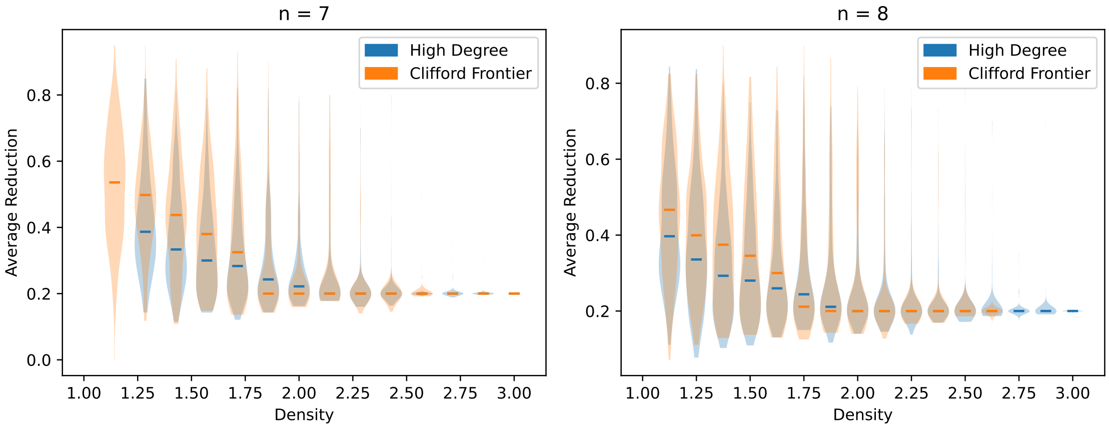


Figure 7.5: The average proportion of terms simplified after two decompositions, either by decomposing three clauses near previous decompositions vs three clauses connected to high-degree variables.

One reason this may not happen at higher densities is that the degree of Z-spiders is higher on average, so two decompositions may not be enough to generate a Z-spider which has no connected clauses, and thus might be removed by Clifford simplification.

In order to implement the strategy of picking decompositions clustered around low-degree vertices, it was modified to use the previous strategy of decomposing two clauses at high-degree variables whenever no low-degree variable is available. This was compared to just the high-degree technique alone, and the number of terms required is presented in Figure 7.6. We can see that this strategy has two distinct modes of behaviour - at lower densities, the number of terms is fairly flat, before making a sharp transition at a certain density when low-degree variables are no longer present. The flat region is probably a balance between the increasing number of Clauses and the decreasing branching number.

To see this transition more clearly, the α-value of the first decomposition was measured empirically, as before, and is presented in Figure 7.7. As predicted, we can see that α decreases towards zero, as the density increases since Δ_i increases, but then when no low-density clauses are present, it transitions to the expected $\alpha = \frac{1}{2}$ of the high-degree method. Also, note that this transition does not occur at the same point for $n = 6$ and $n = 7$ - this may be because the minimum degree of a random graph depends on both the density and the number of vertices n, shown by Riordan and Selby. This is shown experimentally in Figure 7.8, which shows how the proportion of random instances with a minimum degree of at most three varies with n and δ.

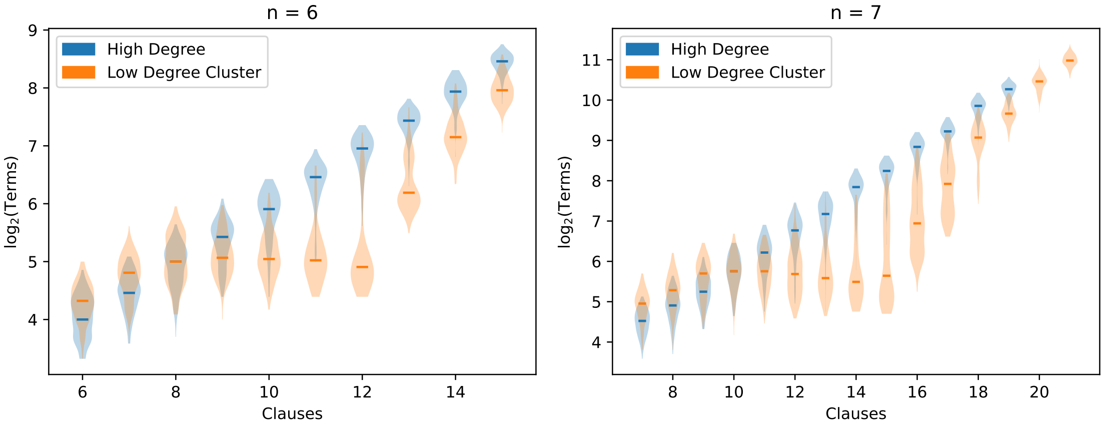


Figure 7.6: The number of terms needed to evaluate #2SAT diagrams by decomposing three clauses around a low-degree variable vs two clauses connected to high-degree variables.

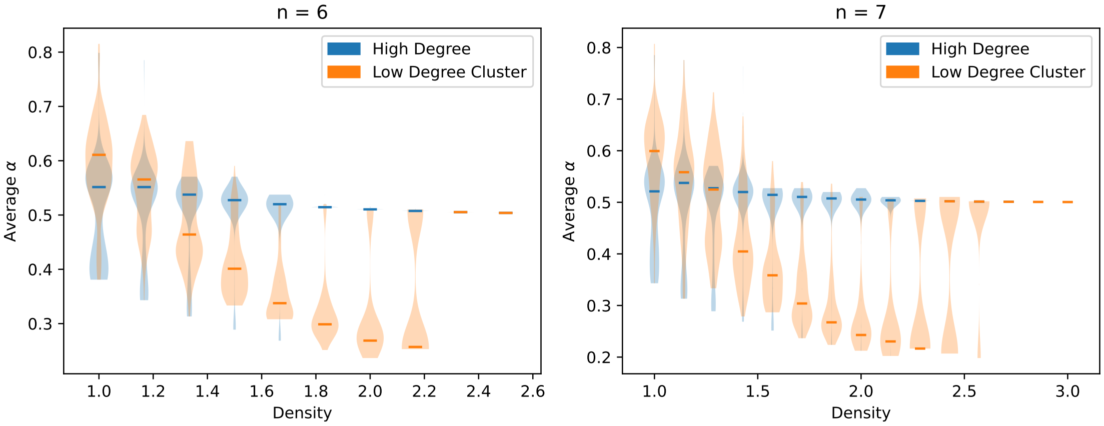


Figure 7.7: Average α value of the first decomposition of a diagram evaluated by decomposing three clauses around a low-degree variable, and two clauses at high-degree variables.

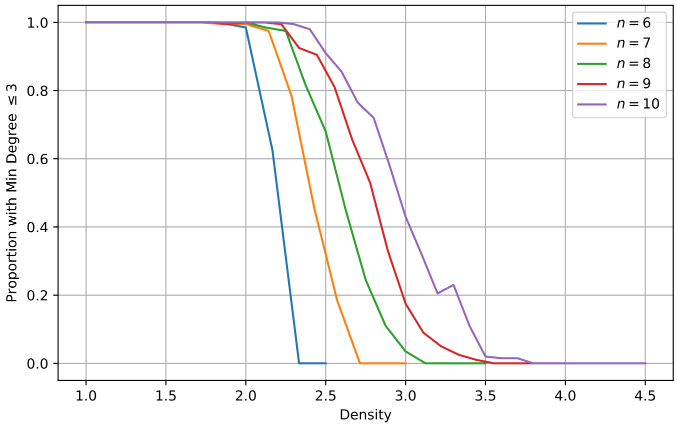


Figure 7.8: The proportion of randomly sampled #2SAT instances which have a minimum degree of at most three according to n and δ.

Finally, an attempt was made to measure the best possible performance of this method: instances were sampled, and the decompositions were picked by considering 100 random decomposition locations, substituting and simplifying these, and measuring the branching number of each choice empirically. The decomposition with the lowest α-value is picked at each step. The number of terms required by this strategy is compared to the previous strategies in Figure 7.9.

This also exhibits two distinct behaviour modes, and so in order to characterize the transition point, the number of terms was measured for $n = 6$ to 10 and is compared to density in Figure 7.10. We can see that the transition point appears earlier, around $\delta = 1.6$ to $\delta = 1.8$, as opposed to $\delta \geq 2$, and does not appear to change as n increases.

Furthermore, we observe little to no scaling as n increases, especially in the linear region, suggesting that the branching number is essentially dependent on the density. This is unexpected because if n increases while keeping δ constant, the number of clauses increases, which we would expect to require more terms to decompose. Since the instances sampled here are so small, further data collection and analysis would be required to determine the behaviour of this method conclusively.

7.3 Challenges and Future Directions

All the instances used in this chapter have been very small, with $n \leq 10$ and $m \leq 45$, and this is because the algorithm operates very slowly (e.g hours per instance for $m = 45$). A natural question is, why is this method so slow, even

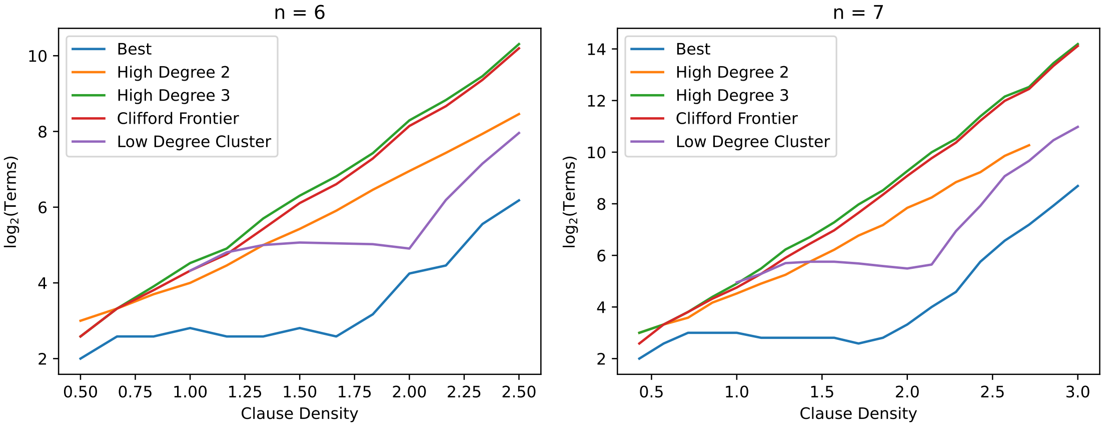


Figure 7.9: The number of terms needed to evaluate #2SAT diagrams by picking the best decomposition from 100 random options at each step compared to all previous strategies. Note that only the median values are shown to reduce clutter. High Degree 2 and 3 refer to decomposing two and three clauses connected to high-degree variables, respectively.

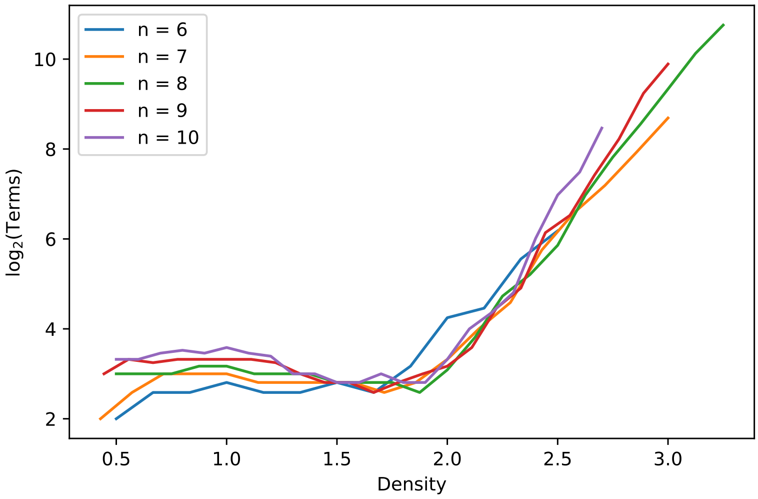


Figure 7.10: The number of terms needed to evaluate #2SAT diagrams by picking the best decomposition from 100 random options compared to itself for $n = 6, 7, 8, 9, 10$.

though the algorithm it is inspired by for Clifford+T is very effective? There are several factors at play.

Firstly, there is one crucial implementation choice that makes the rewriting steps slow: $H_1^1[0]$-boxes and NOT gates are represented explicitly in the diagram as vertices, and not as edges like in $\#DPLL_2$. This has several consequences:

1. All operations, like checking connectivity between variables or finding the variables that a clause connects to, require walking multiple steps through the graph. This is $O(1)$, but much more costly than the native graph operations.

2. The rewrite rules given by Lemmas 5.4, and 7.1 are implemented in $O(m)$ time and not $O(n)$ time.

3. The rewrite rule Lemma 5.5 requires $O(m)$ time - if clauses were considered edges, this could be amortized as $O(1)$ by incorporating it into the operation of adding a new edge.

4. The rewrite rule given by Lemma 5.6, is implemented in $O(m^2)$ time, and requires this much time to check if it is even applicable! If clauses were considered edges, this could also be implemented in amortized $O(1)$ time by incorporating it into the operation of adding new edges.

5. While Clifford simplification usually takes at most $O(n)$ time to check if rewriting can occur, since our diagram has $O(n+m)$ vertices, it takes at least $O(n+m)$ time, even if no simplification occurs.

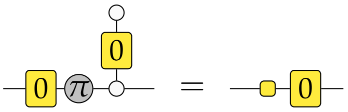
In total, this means the cost of rewriting at each step is at least $O(m^2)$, compared to $O(n^2)$ that Kissinger and van de Wetering argue for their algorithm. In the worst case, where $m = O(n^2)$, this gives a runtime of $O(n^4)$! For $n = 10$, this is already a factor of 100 times slower, not taking into account the extra time required for basic graph operations.

Secondly, in the algorithm for Clifford+T diagrams, Clifford simplification is guaranteed after every decomposition and may cascade. However, in #Decompose, note that a $\frac{\pi}{2}$ phase is never introduced into any Z-spider by any of the decompositions or simplifications. This is problematic because it means that pivoting and boundary pivoting are the only operative simplifications.

These require a pair of Pauli vertices connected via Hadamard gates and connected to their neighbours only with Hadamard gates to operate, and so it means this means at least two layers of decompositions must take place before this is ever applicable in a #2SAT diagram, and in practice, it may take many more layers if the degrees of the variables are high - this was seen in practice in Figure 7.5, where simplifications only occur at low-density. In contrast, in the method for Clifford+T diagrams, all diagrams are converted to graph-like form, and every pair of Z-spiders is connected by Hadamard edges, so Clifford simplification is directly applicable.

Now that we've examined why this method is not performing well, what could be done to remedy it? One of the most important steps would be to reimplement the algorithm to consider $H_1^1[0]$-boxes and their attached NOT gates to be edges between Z-spiders, not generators in their own right. While this may seem to be a departure from the main thesis of this dissertation that the ZH calculus is a valuable tool for examining the #SAT problem, this need only be an implementation detail.

Secondly, clearly, $H_1^1[0]$-box decompositions are not an adequate way of introducing stabilizer regions to the diagram that can be simplified. To this end, looking for more rewriting rules that can be used to introduce stabilizer parts to the diagram without decompositions would be very valuable. For instance, the following rule allows a special case of the pure literal rule for degree-one variables (which is normally only applicable to SAT) to be applied to #SAT in exchange for introducing a Hadamard gate:



(7.10)

In particular, rewrites like this that also remove $H_1^1[0]$-boxes would be very powerful and show the usefulness of the method.

CHAPTER 8

Conclusion

In this dissertation, we explored how the ZH calculus could be applied to analyze the #SAT problem. We outlined the work of de Beudrap, Kissinger, and Meichanetzidis [dKM21] showing that #SAT instances can be embedded into ZH-diagrams and extended it to show that evaluating a large subset of ZH-diagrams is not only #P-hard but $\mathrm{FP}^{\#\mathrm{P}}$-complete.

Motivated by this equivalence, we showed how the classic algorithm of Davis, Putnam, Loveland, and Logemann [DLL62] for SAT and its variants for #SAT can be expressed in the ZH calculus. Using diagrammatic methods, we gave a novel extension of this algorithm to consider clauses and variables equally and used this to show an upper bound on the runtime of #kSAT that is independent of k and improves on the existing upper bounds for small densities, as well as a novel unconditional upper bound for #4SAT in terms of clauses.

We employed stabilizer decompositions to give an algorithm for #2SAT incorporating Clifford simplification that is directly derived from the algorithm for evaluating ZX-diagrams developed by Kissinger and van de Wetering [Kv21]. In order to facilitate this, several novel stabilizer decompositions were found using the method of Bravyi, Smith, and Smolin [BSS16], and a generalization of the method of Kocia [Koc22]. The algorithm was implemented and different decomposition strategies were evaluated to measure how Clifford simplification can be used effectively in this instance. It was found that this method cannot be effective without modifications, and some potential directions of improvement were suggested.

Finally, we will conclude with some ideas for future research directions:

- While embedding general #SAT into ZH calculus was demonstrated by de Beudrap, Kissinger, and Meichanetzidis, other similar problems may also have nice embeddings, either into ZH calculus or other graphical calculi.

In particular, it would be interesting to explore the connection between #1-in-3SAT and the ZW calculus, or how MaxSAT might be embedded in a graphical calculus.

- Resolution is a technique (shown graphically as Equation (3.12)) that was originally used by Davis and Putnam [DP60] to solve SAT but has fallen out of favour since then for SAT and #SAT because it tends to increase the size of clauses, leading to inefficiency. However, for 2SAT and #2SAT, this is not the case, and in fact, the resolution rule shows that 2SAT can be evaluated in polynomial time [dKM21, 4.3]. Therefore, one possible avenue of research is to see how resolution can be used graphically to simplify #2SAT instances.
- Since we have shown that $\mathrm{Eval}(\mathrm{ZH}^{\mathbb{Q}}_{\pi/4})$ is $\mathrm{FP}^{\#\mathrm{P}}$-complete, it would be interesting to try and apply this reduction to practical ZH-diagrams. In particular, ZH-diagrams for representing simulations of quantum computers, for which the calculus was originally introduced, exist in the $\mathrm{ZH}_{\frac{\pi}{4}}$ fragment and so could be evaluated in this way. Given that state-of-the-art #SAT solvers can solve instances with hundreds of thousands of variables, this may be an efficient way to evaluate these ZH-diagrams, and could be compared to other methods of simulating quantum computers, such as the algorithm of Kissinger and van de Wetering that we adapted earlier.

Bibliography

[AG04] Scott Aaronson and Daniel Gottesman. Improved Simulation of Stabilizer Circuits. *Physical Review A*, 70(5):052328, November 2004. Comment: 15 pages. Final version with some minor updates and corrections. Software at http://www.scottaaronson.com/chp.

[AJS98] Bruce Anderson, Jeffrey Jackson, and Meera Sitharam. Descartes' Rule of Signs Revisited. *The American Mathematical Monthly*, 105(5):447–451, 1998.

[BBC$^+$01] Béla Bollobás, Christian Borgs, Jennifer T. Chayes, Jeong Han Kim, and David B. Wilson. The Scaling Window of the 2-SAT Transition. *Random Structures & Algorithms*, 18(3):201–256, May 2001. Comment: 57 pages. This version updates some references.

[BHv09] A. Biere, M. Heule, and H. van Maaren. *Handbook of Satisfiability*. IOS Press, Incorporated, 2009.

[BK19] Miriam Backens and Aleks Kissinger. ZH: A Complete Graphical Calculus for Quantum Computations Involving Classical Nonlinearity. *Electronic Proceedings in Theoretical Computer Science*, 287:23–42, January 2019. Comment: In Proceedings QPL 2018, arXiv:1901.09476.

[BKM$^+$21] Miriam Backens, Aleks Kissinger, Hector Miller-Bakewell, John van de Wetering, and Sal Wolffs. Completeness of the ZH-calculus, March 2021. Comment: 64 pages, many many diagrams.

[BL99] E. Birnbaum and E. L. Lozinskii. The Good Old Davis-Putnam Procedure Helps Counting Models. *Journal of Artificial Intelligence Research*, 10:457–477, June 1999.

[BS97] Roberto J. Bayardo and Robert C. Schrag. Using CSP look-back techniques to solve real-world SAT instances. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence and Ninth Conference on Innovative Applications of Artificial Intelligence*, AAAI'97/IAAI'97, pages 203–208, Providence, Rhode Island, July 1997. AAAI Press.

[BSS16] Sergey Bravyi, Graeme Smith, and John Smolin. Trading classical and quantum computational resources. *Physical Review X*, 6(2):021043, June 2016. Comment: 14 pages, 4 figures.

[Car21] Titouan Carette. *Wielding the ZX-calculus, Flexsymmetry, Mixed States, and Scalable Notations*. Theses, Université de Lorraine, November 2021.

[CD11] Bob Coecke and Ross Duncan. Interacting Quantum Observables: Categorical Algebra and Diagrammatics. *New Journal of Physics*, 13(4):043016, April 2011. Comment: 81 pages, many figures. Significant changes from previous version. The first sections contain a gentle introduction for physicists to the graphical language, and its use in quantum computation.

[CIP06] C. Calabro, R. Impagliazzo, and R. Paturi. A duality between clause width and clause density for SAT. In *21st Annual IEEE Conference on Computational Complexity (CCC'06)*, pages 7 pp.–260, July 2006.

[CIP09] Chris Calabro, Russell Impagliazzo, and Ramamohan Paturi. The Complexity of Satisfiability of Small Depth Circuits. In Jianer Chen and Fedor V. Fomin, editors, *Parameterized and Exact Computation*, Lecture Notes in Computer Science, pages 75–85, Berlin, Heidelberg, 2009. Springer.

[CK17] Bob Coecke and Aleks Kissinger. *Picturing Quantum Processes: A First Course in Quantum Theory and Diagrammatic Reasoning*. Cambridge University Press, Cambridge, 2017.

[Cod22] Julien Codsi. *Cutting Edge Graphical Stabilizer Decompositions for Classical Simulation of Quantum Circuits*. PhD thesis, University of Oxford, 2022.

[Coo71] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, pages 151–158, New York, NY, USA, May 1971. Association for Computing Machinery.

[Dam18] Raja Oktovin Parhasian Damanik. Optimality in Stabilizer Testing. Report, August 2018.

[Dar02] Adnan Darwiche. A compiler for deterministic, decomposable negation normal form. In *Eighteenth National Conference on Artificial Intelligence*, pages 627–634, USA, July 2002. American Association for Artificial Intelligence.

[DHM02] Carsten Damm, Markus Holzer, and Pierre McKenzie. The complexity of tensor calculus. *Computational Complexity*, 11(1/2):54–89, June 2002.

[DJW02a] Vilhelm Dahllöf, Peter Jonsson, and Magnus Wahlström. Counting Satisfying Assignments in 2-SAT and 3-SAT. In Oscar H. Ibarra and Louxin Zhang, editors, *Computing and Combinatorics*, Lecture Notes in Computer Science, pages 535–543, Berlin, Heidelberg, 2002. Springer.

[DJW02b] Vilhelm Dahllöf, Peter Jonsson, and Magnus Wahlström. Counting Satisfying Assignments in 2-SAT and 3-SAT. In Oscar H. Ibarra and Louxin Zhang, editors, *Computing and Combinatorics*, Lecture Notes in Computer Science, pages 535–543, Berlin, Heidelberg, 2002. Springer.

[dKM21] Niel de Beaudrap, Aleks Kissinger, and Konstantinos Meichanetzidis. Tensor Network Rewriting Strategies for Satisfiability and Counting. *Electronic Proceedings in Theoretical Computer Science*, 340:46–59, September 2021. Comment: In Proceedings QPL 2020, arXiv:2109.01534.

[DLL62] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, July 1962.

[dNDDM04] Maarten Van den Nest, Jeroen Dehaene, and Bart De Moor. Graphical description of the action of local Clifford transformations on graph states. *Physical Review A*, 69(2):022316, February 2004. Comment: 8 pages, 1 figure. Former title: "The evolution of graph states under local Clifford operations as graph transformations". Replaced with revised version. Minor changes. To appear in Phys. Rev. A.

[DP60] Martin Davis and Hilary Putnam. A Computing Procedure for Quantification Theory. *Journal of the ACM*, 7(3):201–215, July 1960.

[DP09] Ross Duncan and Simon Perdrix. *Graphs States and the Necessity of Euler Decomposition*, volume 5635. 2009. Comment: 15pages, 38 figures.

[Dub91] Olivier Dubois. Counting the number of solutions for instances of satisfiability. *Theoretical Computer Science*, 81(1):49–64, April 1991.

[Epp03] David Eppstein. Quasiconvex Analysis of Backtracking Algorithms, July 2003. Comment: 12 pages, 2 figures. This revision includes a larger example recurrence and reports on a second implementation of the algorithm.

[FK07] Martin Fürer and Shiva Prasad Kasiviswanathan. Algorithms for Counting 2-Sat Solutions and Colorings with Applications. In Ming-Yang Kao and Xiang-Yang Li, editors, *Algorithmic Aspects in Information and Management*, Lecture Notes in Computer Science, pages 47–57, Berlin, Heidelberg, 2007. Springer.

[Gad20] Ahmed Gad. PyGAD - Python Genetic Algorithm! — PyGAD 2.17.0 documentation, 2020.

[GK21] Johnnie Gray and Stefanos Kourtis. Hyper-optimized tensor network contraction. *Quantum*, 5:410, March 2021. Comment: 22 pages, 10 figures.

[GKST16] Alexander Golovnev, Alexander S. Kulikov, Alexander V. Smal, and Suguru Tamaki. Circuit Size Lower Bounds and #SAT Upper Bounds Through a General Framework. In Piotr Faliszewski, Anca Muscholl, and Rolf Niedermeier, editors, *41st International Symposium on Mathematical Foundations of Computer Science (MFCS 2016)*, volume 58 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 45:1–45:16, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[GTN$^+$21] Jay Gambetta, Matthew Treinish, Paul Nation, Paul Kassebaum, Diego M. Rodríguez, qiskit-bot, Salvador de la Puente González, Shaohan Hu, Kevin Krsulich, Laura Zdanski, Jessie Yu, David McKay, Juan Gomez, Lauren Capelluto, Travis-S-IBM, Julien Gacon, lerongil, Rafey Iqbal Rahman, Steve Wood, Drew, Joachim Schwarm, Luciano Bello, MELVIN GEORGE, Manoel Marques, Omar Costa Hamido, RohitMidha23, Sean Dague, Shelly Garion, tigerjack, and Yuri Kobayashi. Qiskit/qiskit: Qiskit 0.25.0. Zenodo, April 2021.

[Hir00] Edward A. Hirsch. New Worst-Case Upper Bounds for SAT. *Journal of Automated Reasoning*, 24(4):397–420, May 2000.

[HL08] Sven Hammarling and Craig Lucas. Updating the QR factorization and the least squares problem. http://eprints.maths.manchester.ac.uk/1192/, November 2008.

[IP99] R. Impagliazzo and R. Paturi. Complexity of k-SAT. In *Proceedings. Fourteenth Annual IEEE Conference on Computational Complexity (Formerly: Structure in Complexity Theory Conference) (Cat.No.99CB36317)*, pages 237–240, May 1999.

[JPV19] Emmanuel Jeandel, Simon Perdrix, and Renaud Vilmart. Completeness of the ZX-Calculus. March 2019.

[KCK21] Sourabh Katoch, Sumit Singh Chauhan, and Vijay Kumar. A review on genetic algorithm: Past, present, and future. *Multimedia Tools and Applications*, 80(5):8091–8126, February 2021.

[KGV83] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by Simulated Annealing. *Science*, 220(4598):671–680, May 1983.

[Koc22] Lucas Kocia. Improved Strong Simulation of Universal Quantum Circuits, June 2022.

[Kut07] Konstantin Kutzkov. New upper bound for the #3-SAT problem. *Information Processing Letters*, 105(1):1–5, December 2007.

[Kv21] Aleks Kissinger and John van de Wetering. Simulating quantum circuits with ZX-calculus reduced stabiliser decompositions. September 2021.

[Kv22] Aleks Kissinger and John van de Wetering. *Picturing Quantum Software*. (Draft), 2022.

[KvV22] Aleks Kissinger, John van de Wetering, and Renaud Vilmart. Classical simulation of quantum circuits with partial and graphical stabiliser decompositions. February 2022.

[LvK21] Louis Lemonnier, John van de Wetering, and Aleks Kissinger. Hypergraph Simplification: Linking the Path-sum Approach to the ZH-calculus. *Electronic Proceedings in Theoretical Computer Science*, 340:188–212, September 2021. Comment: In Proceedings QPL 2020, arXiv:2109.01534.

[MPZ02] M. Mézard, G. Parisi, and R. Zecchina. Analytic and Algorithmic Solution of Random Satisfiability Problems. *Science*, 297(5582):812–815, August 2002.

[MSV81] Burkhard Monien, Ewald Speckenmeyer, and Oliver Vornberger. Upper bounds for covering problems. *Methods of operations research*, 43:419–431, 1981.

[NW18] Kang Feng Ng and Quanlong Wang. Completeness of the ZX-calculus for Pure Qubit Clifford+T Quantum Mechanics, January 2018. Comment: 26 pages.

[OD15] Umut Oztok and Adnan Darwiche. A top-down compiler for sentential decision diagrams. In *Proceedings of the 24th International Conference on Artificial Intelligence*, IJCAI'15, pages 3141–3148, Buenos Aires, Argentina, July 2015. AAAI Press.

[QPG21] Hammam Qassim, Hakop Pashayan, and David Gosset. Improved upper bounds on the stabilizer rank of magic states. *Quantum*, 5:606, December 2021. Comment: A preliminary version of our results was reported in the first author's Ph.D thesis.

[RS00] Oliver Riordan and Alex Selby. The Maximum Degree of a Random Graph. *Combinatorics, Probability and Computing*, 9(6):549–572, November 2000.

[SBB$^+$04] Tian Sang, Fahiem Bacchus, Paul Beame, Henry A. Kautz, and Toniann Pitassi. Combining Component Caching and Clause Learning for Effective Model Counting. In *SAT 2004 - The Seventh International Conference on Theory and Applications of Satisfiability Testing, 10-13 May 2004, Vancouver, BC, Canada, Online Proceedings*, 2004.

[Sly10] Allan Sly. Computational Transition at the Uniqueness Threshold, May 2010. Comment: 33 pages.

[SRSM19] Shubham Sharma, Subhajit Roy, Mate Soos, and Kuldeep S. Meel. GANAK: A Scalable Probabilistic Exact Model Counter. pages 1169–1176, 2019.

[The] [Redacted] The author. Cliffs.

[Thu06] Marc Thurley. sharpSAT – Counting Models with Advanced Component Caching and Implicit BCP. In Armin Biere and Carla P. Gomes, editors, *Theory and Applications of Satisfiability Testing - SAT 2006*, Lecture Notes in Computer Science, pages 424–429, Berlin, Heidelberg, 2006. Springer.

[TI97] Lloyd N. Trefethen and David Bau III. *Numerical Linear Algebra*. SIAM, June 1997.

[Tse83] G. S. Tseitin. On the Complexity of Derivation in Propositional Calculus. In Jörg H. Siekmann and Graham Wrightson, editors, *Automation of Reasoning: 2: Classical Papers on Computational Logic 1967–1970*, Symbolic Computation, pages 466–483. Springer, Berlin, Heidelberg, 1983.

[Val79] Leslie G. Valiant. The Complexity of Enumeration and Reliability Problems. *SIAM Journal on Computing*, 8(3):410–421, August 1979.

[Vil19] Renaud Vilmart. A ZX-Calculus with Triangles for Toffoli-Hadamard, Clifford+T, and Beyond. *Electronic Proceedings in Theoretical Computer Science*, 287:313–344, January 2019. Comment: In Proceedings QPL 2018, arXiv:1901.09476. Contains Appendix.

[Wah08] Magnus Wahlström. A Tighter Bound for Counting Max-Weight Solutions to 2SAT Instances. In Martin Grohe and Rolf Niedermeier, editors, *Parameterized and Exact Computation*, Lecture Notes in Computer Science, pages 202–213, Berlin, Heidelberg, 2008. Springer.

[Wei] Eric W. Weisstein. Normal Equation. https://mathworld.wolfram.com/.

[WG13] Honglin Wang and Wenxiang Gu. The Worst Case Minimized Upper Bound in #2-SAT. In Wei Lu, Guoqiang Cai, Weibin Liu, and Weiwei Xing, editors, *Proceedings of the 2012 International Conference on Information Technology and Software Engineering*, Lecture Notes in Electrical Engineering, pages 675–682, Berlin, Heidelberg, 2013. Springer.

[Wil04] Ryan Williams. On Computing k-CNF Formula Properties. In Enrico Giunchiglia and Armando Tacchella, editors, *Theory and Applications of Satisfiability Testing*, Lecture Notes in Computer Science, pages 330–340, Berlin, Heidelberg, 2004. Springer.

[Wil21] Ryan Williams. On the Usefulness of the Strong Exponential Time Hypothesis. https://simons.berkeley.edu/talks/tbd-270, February 2021.

[ZYZ10] Junping Zhou, Minghao Yin, and Chunguang Zhou. New worst-case upper bound for #2-SAT and #3-SAT with the number of clauses as the parameter. In *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence*, AAAI'10, pages 217–222, Atlanta, Georgia, July 2010. AAAI Press.