Compiling Boolean circuits into Ising spin systems

Physics 212 Final Project

Ben Bartlett[1]

[1] *Department of Applied Physics, Stanford University, Stanford, CA 94305, USA**

The Ising model, despite its simplicity, has a rich array of properties that allow for universal computation. In this project, I show how to implement each elementary Boolean logic gate as an Ising system of 2-4 spins. I developed a Python library for compiling a sequence of Boolean logic gates into a spin system where the result of the computation is encoded in the ground state of the Hamiltonian. I provide several demonstrations of compiling complex circuits into spin systems and use Monte Carlo simulations to show that the compiled circuits encode the desired computations.

I. INTRODUCTION

The well-studied Ising model is the canonical model of ferromagnetism in statistical mechanics; its simple structure gives rise to a surprising array of interesting properties from both physical and computational perspectives. Notably, solving for the ground state of an arbitrary Ising spin system has been shown to be an NP-complete problem [1]. Since NP-complete problems are poly-time reducible to each other, Ising spin systems are capable of computing solutions to a tremendous amount of interesting problems. Indeed, all of Karp's 21 NP-complete problems [2] have been formulated as Ising-type problems where the solution is encoded in the ground state of the spin system [3].

In recent years, there has been significant interest in designing programmable Ising machines: physical devices which implement variational Hamiltonians on a network of two-level systems and perform an annealing process to find the ground state of the system. It is hoped that these physical annealers may offer a polynomial (or much less likely[1], exponential) speedup for various computationally hard optimization problems.

For this project, I explored the ability of Ising spin systems to compute Boolean logic circuits. I developed a Python-based library for compiling a sequence of high-level statements describing a Boolean circuit into a Hamiltonian of an interacting spin system. The result of the computations are encoded in the ground states of the system, which are degenerate for each logical input, and the degeneracy can be broken by applying a strong magnetic field to the input spins to specify their state. Thus, the same system can be used to compute the result of any logical input. Using Markov-chain Monte Carlo (MCMC) simulations, I demonstrate the ability of library to correctly compile complex circuits into Ising Hamiltonians.

* benbartlett@stanford.edu

[1] If any NP-complete problem is solvable in polynomial time on a quantum annealer, it would imply that NP $\subseteq$ BQP, which is thought not to be the case.

II. BOOLEAN SPIN LOGIC

I organize this section as follows: Section II A introduces notation, in Section II B I describe a method of engineering various elementary Boolean gates from few-spin systems, and in Section II C I discuss the details of how the compiler programmatically transforms a sequence of logical statements into an Ising Hamiltonian.

Most of the conceptual results in this paper are not new: Refs. [4] and [5] presented various methods of encoding elementary Boolean gates in the ground states of Ising systems. Although my derivation and formulation of the gates differs slightly from both of these papers, the main novel contribution of this paper is a Python library to compile arbitrary circuits from high-level instructions, which, to my knowledge, is the first implementation of such a library. The library includes Monte Carlo simulation methods and extensive unit tests to verify the functionality of the compiled circuits. All code written for this project can be found at `github.com/fancompute/ising-compiler`.

A. Background and notation

Let us first define the notation that I use in this paper. The most general possible Hamiltonian of a system of N interacting spins can be written as

$$H = \sum_i c_i \sigma_i + \sum_{i,j} c_{ij} \sigma_i \sigma_j + \sum_{i,j,k} \sigma_i \sigma_j \sigma_k + \cdots \tag{1}$$

where σ is the Pauli-z operator $|0\rangle\langle 0| - |1\rangle\langle 1|$ with spin eigenvalues ± 1. For notational simplicity, when discussing logical circuits in Section II, we denote Boolean variables $x = 0, 1$ as $x_i = \frac{\sigma_i + 1}{2}$.

The above Hamiltonian allows for n-body interactions, but if we restrict ourselves to only single- and two-body interactions, we obtain the familiar Ising Hamiltonian:

$$H_{\text{Ising}} = \sum_i c_i \sigma_i + \sum_{i,j} c_{ij} \sigma_i \sigma_j = -\sum_{\langle ij \rangle} J_{ij} \sigma_i \sigma_j - \sum_i h_i \sigma_i, \tag{2}$$

where h_i is the local magnetic field applied to spin σ_i, where J_{ij} is the coupling constant between spins i, j, with

$J = +1$ denoting ferromagnetic and $J = -1$ denoting antiferromagnetic coupling, and where $\langle ij \rangle$ denotes summation over all adjacent spins i, j. Eq. 2 represents an experimentally realizable Hamiltonian that can be implemented on Ising machines.

B. Designing elementary Boolean gates

To implement a Boolean gate with I inputs and O outputs, we wish to engineer a system of $k \geq I + O$ spins which has a ground subspace spanned by degenerate ground states corresponding to the truth table of the gate. For example, consider a spin system implementing the functionally complete NAND gate: $\neg x \vee \neg y$. The system should have the ground states:

$$\mathcal{G}(H_{\text{NAND}}) = \{|001\rangle, |011\rangle, |101\rangle, |110\rangle\}, \tag{3}$$

where $x_i = 0, 1$ corresponds to $\sigma_i = -1, +1$ as introduced earlier. To provide inputs to the gate, apply strong magnetic fields to spins to deterministically set them to 0 or 1 and "connect" these spins the gate by adding a coupling term to the Hamiltonian, as discussed in the next section.

In the subsections below, we describe how to implement all elementary Boolean gates with 1-2 inputs and one output.

1. *Two-spin gates: NOT, COPY*

The simplest logic gates to implement are $\text{COPY}(x) = x$ and $\text{NOT}(x) = \neg x$. COPY is implementable simply by adding a coupling[2] to the two-spin Hamiltonian, so $H_{COPY} = -\sigma_1\sigma_2$, where σ_1 is the input spin and σ_2 is the output spin. Similarly $H_{NOT} = +\sigma_1\sigma_2$. When compiling complex Boolean circuits, the output spins of one gate are COPY-d into the input spins of another gate.

2. *Three-spin gates: NAND, AND, NOR, OR*

We can implement NAND using a system of two input spins σ_1, σ_2 and one output spin σ_3 with only two-body interactions. The most general Hamiltonian of this form as:

$$H = c_1\sigma_1 + c_2\sigma_2 + c_3\sigma_3 + c_{12}\sigma_1\sigma_2 + c_{13}\sigma_1\sigma_3 + c_{23}\sigma_2\sigma_3. \tag{4}$$

Imposing the constraints of the ground states described in Eq. 3, we obtain:

$$E_{001} = E_{110} \rightarrow c_2 = c_{12} - c_{23} \tag{5}$$
$$E_{001} = E_{101} \rightarrow c_1 = c_{12} - c_{13} \tag{6}$$
$$E_{001} = E_{110} \rightarrow c_3 = 2c_{12} - c_{13} - c_{23}. \tag{7}$$

Thus, the Hamiltonian for the NAND gate is:

$$\begin{aligned} H_{\text{NAND}} = {} & (c_{12} - c_{13})\,\sigma_1 + (c_{12} - c_{23})\,\sigma_2 \\ & + (2c_{12} - c_{13} - c_{23})\,\sigma_3 \\ & + c_{12}\sigma_1\sigma_2 + c_{13}\sigma_1\sigma_3 + c_{23}\sigma_2\sigma_3, \end{aligned} \tag{8}$$

with parameters c_{12}, c_{13}, c_{23} subject to symmetry constraints. Since NAND is symmetric with its inputs, $c_{13} = c_{23} > 0$, and to prevent $c_3 = 0$, we require $2c_{12} - c_{13} - c_{23} > 0$. In this paper, we limit coupling strength between any two spins to be $-1 \leq c_{ij} \leq 1$, so we choose $c_{13} = c_{23} = 1$ and $c_{12} = \frac{1}{2}$. Thus, the NAND Hamiltonian becomes:

$$H_{\text{NAND}} = -\frac{1}{2}\sigma_1 - \frac{1}{2}\sigma_2 - \sigma_3 + \frac{1}{2}\sigma_1\sigma_2 + \sigma_1\sigma_3 + \sigma_2\sigma_3. \tag{9}$$

This three-spin system is also sufficient to implement AND, NOR, and OR using sign flips on the couplings, since these can all be written as single-bit negations of NAND: $\text{AND}(x, y) = \neg(\neg x \vee \neg y)$, $\text{OR}(x, y) = \neg(\neg x) \vee \neg(\neg y)$, and $\text{NOR}(x, y) = \neg(\neg(\neg x) \vee \neg(\neg y))$. We enumerate the Hamiltonians for these gates in Table I.

3. *Four-spin gates: XOR and XNOR*

The XOR and XNOR gates cannot be written as single-bit negations of NAND and require an ancillary spin to implement. Denoting σ_1, σ_2 as input spins, σ_3 as the output, and σ_4 as the ancilla, we follow a similar procedure as above to construct an XOR gate. (XNOR follows with a simple sign change.) The most general system of four spins with two-body interactions is:

$$\begin{aligned} H = {} & c_1\sigma_1 + c_2\sigma_2 + c_3\sigma_3 + c_4\sigma_4 + c_{12}\sigma_1\sigma_2 + c_{13}\sigma_1\sigma_3 \\ & + c_{14}\sigma_1\sigma_4 + c_{23}\sigma_2\sigma_3 + c_{24}\sigma_2\sigma_4 + c_{34}\sigma_3\sigma_4. \end{aligned} \tag{10}$$

The ancilla for the XOR gate must interact nontrivially with at least one of the ground states, or it provides no additional expressiveness to the gate. That is, not all of the $|\sigma_1\sigma_2\sigma_3\rangle$ ground states $\mathcal{G}(H_{\text{XOR}}) = \{|000\rangle |011\rangle, |101\rangle, |110\rangle\}$ can have the same ancilla state. If we make the arbitrary[3] choice that the $|\sigma_1\sigma_2\sigma_3\sigma_4\rangle$ ground states are:

$$\mathcal{G}(H_{\text{XOR}}) = \{|0001\rangle |0110\rangle, |1010\rangle, |1100\rangle\}, \tag{11}$$

[2] Here, we choose a coupling magnitude of 1 and scaled the Hamiltonians of other Boolean spin circuits to have maximum coupling between any two spins of $-1 < c_{ij} < +1$.

[3] In fact, any combination of three $\sigma_4 = 0$ and one $\sigma_4 = 1$ or the opposite parity is a valid choice for the XOR gate.

then we obtain the following relations by imposing the degeneracy of the ground state energies:

$$E_{0001} = E_{0110} \to c_{14} = -c_2 - c_3 + c_4 + c_{12} + c_{13} \quad (12)$$
$$E_{0001} = E_{1010} \to c_{24} = -c_1 - c_3 + c_4 + c_{12} + c_{23} \quad (13)$$
$$E_{0001} = E_{1100} \to c_{34} = -c_1 - c_2 + c_4 + c_{23} + c_{13}. \quad (14)$$

Substituting this into Eq. 10, we obtain:

$$\begin{aligned} H_{\text{XOR}} = {} & c_1\sigma_1 + c_2\sigma_2 + c_3\sigma_3 + c_4\sigma_4 \\ & + c_{12}\sigma_1\sigma_2 + c_{13}\sigma_1\sigma_3 + c_{23}\sigma_2\sigma_3 \\ & + (-c_2 - c_3 + c_4 + c_{12} + c_{13})\,\sigma_1\sigma_4 \\ & + (-c_1 - c_3 + c_4 + c_{12} + c_{23})\,\sigma_2\sigma_4 \\ & + (-c_1 - c_2 + c_4 + c_{23} + c_{13})\,\sigma_3\sigma_4. \end{aligned} \quad (15)$$

Here, $c_1, c_2, c_3, c_4, c_{12}, c_{13}, c_{23}$ are parameters subject to symmetry constraints. Since XOR is symmetric with its inputs, we should have that $c_1 = c_2$ and $c_{13} = c_{23} > 0$. Since it is symmetric with one-input one-output exchange (such that if the output bit and either input bit in any ground state are swapped, the resulting state is also a ground state), then $c_1 = c_2 = c_3$ and $c_{13} = c_{23} = c_{12}$. If we choose $c_1 = c_2 = c_3 = c_{13} = c_{23} = c_{12} = \frac{1}{2}$, then we have from Eqs. 12 - 14 that $c_{14} = c_{24} = c_{34} = c_4$. This value should be greater than the other couplings to ensure that $\sigma_4 = +1$ causes $\text{XOR}(x_1 = 0, x_2 = 0) = 0$, so we set $c_4 = +1$. Thus, the final XOR Hamiltonian is:

$$\begin{aligned} H_{\text{XOR}} = {} & \frac{1}{2}\left(\sigma_1 + \sigma_2 + \sigma_3 + \sigma_1\sigma_2 + \sigma_1\sigma_3 + \sigma_2\sigma_3\right) \\ & + \sigma_4 + \sigma_1\sigma_4 + \sigma_2\sigma_4 + \sigma_3\sigma_4. \end{aligned} \quad (16)$$

The structure of the four-spin XOR gate is shown in Figure 1.

To implement XNOR, we simply flip the sign of all coupling constants affecting the output bit σ_3. The spin system implementations of all Boolean logic gates are enumerated in Table I.

C. Compiling complex circuits

In this section, I describe the structure of the `ising-compiler` library and how the circuit compilation process works.

The library consists of three main classes which represent different abstraction levels of the Boolean spin logic. The `IsingGraph` class[4] represents a generalization of an Ising lattice, but simulated using the `networkx` library on an arbitrary graph structure. Nodes in the graph contain properties representing the spin and local magnetic field strength, and edges have a coupling property which represents the c_{ij} interaction terms in the Hamiltonian. This class also contains the Monte Carlo simulation logic, which runs a version of the Metropolis-Hastings algorithm that is suitable for arbitrary graphs, as well as an annealing scheduler which exponentially decreases the temperature over the course of the simulation.

The `IsingCircuit` class extends `IsingGraph` with graph-constructing methods that implement the elementary Boolean gates listed in Table I, as well as circuit eval-

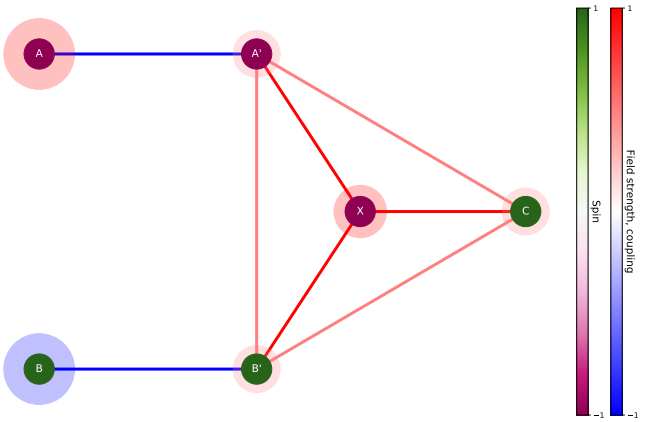

Figure 1: Four-spin implementation of an XOR gate evaluating $1 \oplus 0 = 0$. Input spins A, B are copied to gate inputs A', B'; the ancillary spin is X and the output is $C = A \oplus B$. The purple-green coloring denotes the spin of each node, which can vary depending on the inputs. The red-blue coloring denotes the field strength at each node (circles) and coupling between spins (lines), which do not vary as the inputs change.

Gate	Hamiltonian
COPY	$-\sigma_1\sigma_2$
NOT	$+\sigma_1\sigma_2$
AND	$-\frac{1}{2}\sigma_1 - \frac{1}{2}\sigma_2 + \sigma_3 + \frac{1}{2}\sigma_1\sigma_2 - \sigma_1\sigma_3 - \sigma_2\sigma_3$
NAND	$-\frac{1}{2}\sigma_1 - \frac{1}{2}\sigma_2 - \sigma_3 + \frac{1}{2}\sigma_1\sigma_2 + \sigma_1\sigma_3 + \sigma_2\sigma_3$
OR	$+\frac{1}{2}\sigma_1 + \frac{1}{2}\sigma_2 - \sigma_3 + \frac{1}{2}\sigma_1\sigma_2 - \sigma_1\sigma_3 - \sigma_2\sigma_3$
NOR	$+\frac{1}{2}\sigma_1 + \frac{1}{2}\sigma_2 + \sigma_3 + \frac{1}{2}\sigma_1\sigma_2 + \sigma_1\sigma_3 + \sigma_2\sigma_3$
XOR	$\frac{1}{2}(\sigma_1 + \sigma_2 + \sigma_3 + \sigma_1\sigma_2 + \sigma_1\sigma_3 + \sigma_2\sigma_3)$ $+\sigma_4 + \sigma_1\sigma_4 + \sigma_2\sigma_4 + \sigma_3\sigma_4$
XNOR	$\frac{1}{2}(\sigma_1 + \sigma_2 - \sigma_3 + \sigma_1\sigma_2 - \sigma_1\sigma_3 - \sigma_2\sigma_3)$ $+\sigma_4 + \sigma_1\sigma_4 + \sigma_2\sigma_4 - \sigma_3\sigma_4$

Table I: Ising formulations of all elementary Boolean logic gates.

[4] I initially implemented two other versions of the `IsingGraph` class built using NumPy and PyTorch backends, both of which operate only on square n-dimensional lattices. The NumPy backend is significantly more performant than the `networkx` version, but I did not have time to finish the logic for embedding the graph structure of a compiled circuit into a planar lattice. (An outline of how this would work is presented in Section IV.) The PyTorch version allows for automatic differentiation of the spin system parameters, which could be used to inverse design more compact Boolean circuits for a desired truth table, although there was also insufficient time to pursue this idea as well.

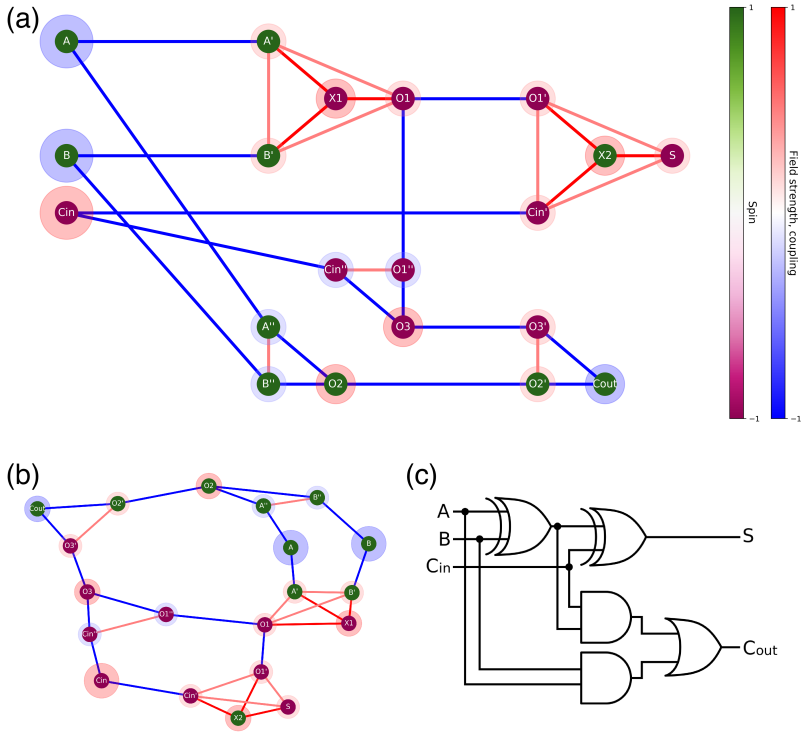

Figure 2: Ising graph implementing a full adder circuit. (a) The spin system, manually laid out to highlight the correspondence of the spins with the gates in the circuit. Primed spins denote copies, X are ancillae, O are intermediate outputs, and S, C_{out} are the sum and carry circuit outputs. Purple-green denotes spin and red-blue denotes field strength and coupling between spins. (b) Default layout of the same graph. (c) The logical circuit implementing a full adder.

uation functions. An `IsingCircuit()` instance starts off containing an empty graph; as gates are added to the circuit, subgraphs representing the corresponding gates are progressively added to the spin graph. Gates take references to the input spin nodes as arguments and return a reference to the output spin; any inputs to the gates are `COPY`-d, as some gates apply magnetic fields directly to the input spins. Nodes in the circuit can be designated as logical inputs and outputs: when the circuit is evaluated, a dictionary is passed in containing the input values, and strong magnetic fields in the appropriate direction are applied to the corresponding spins. The class contains methods for evaluating single-shot MC simulations of the circuit, computing expectation values of spins in the circuit, and computing counts of the most frequent outcomes over an ensemble of runs.

Finally, the `IsingALU` class extends the functionality of `IsingCircuit` to build various high-level arithmetic logic modules. The methods have the same signatures as the gate methods, taking a set of input spins as arguments and returning a list of output spins. Currently implemented are circuits for half adders, full adders, and ripple-carry adders. (See Section III for demonstrations.)

The library includes an extensive set of unit tests (in `tests_nx.py`) which use MCMC simulations to verify that the ground states of gates and compiled circuits do indeed encode the correct result of the computation with an accuracy above some threshold.

III. SIMULATIONS

A. Full adder circuit

Program 1 contains Python code for implementing a full adder circuit using the `ising-compiler` library. The logical circuit and compiled spin system are shown in Figure 2. Using an annealing timescale of 200000 iterations, the spin systems achieves over 85% accuracy for all possible input states.

```
from ising_compiler.gates_nx import IsingCircuit

# Create an empty Ising graph
circuit = IsingCircuit()

# Create input spins
A = circuit.INPUT("A")
B = circuit.INPUT("B")
Cin = circuit.INPUT("Cin")

# Intermediate gates
```

```
O1 = circuit.XOR(A, B)
O2 = circuit.AND(A, B)
O3 = circuit.AND(O1, Cin)

# Add sum and c_out spins
S = circuit.XOR(O1, Cin, out = "S")
Cout = circuit.OR(O2, O3, out = "Cout")

# Register S and C as output spins
circuit.OUTPUT(S)
circuit.OUTPUT(Cout)

# Evaluate the circuit with inputs A=1, B=1, Cin=0
out = circuit.evaluate_input(
          {"A": 1, "B": 1, "Cin": 0},
          mode = 'binary',
          epochs = 20000,
          anneal_temperature_range = [1, 1e-4])

# >> out = {"S": 0, "Cout": 1}
```

Program 1: Simulation and evaluation of a spin system implementing a full adder circuit, which takes two single-bit inputs A, B and a carry bit C_{in}, returning a sum bit S and a carry bit C_{out}.

B. Ripple-carry adder

A ripple-carry adder adds two n-bit binary integers and consists of a circuit of consecutive full adders. An implementation of an 8-bit ripple-carry adder is shown in Figure 3.

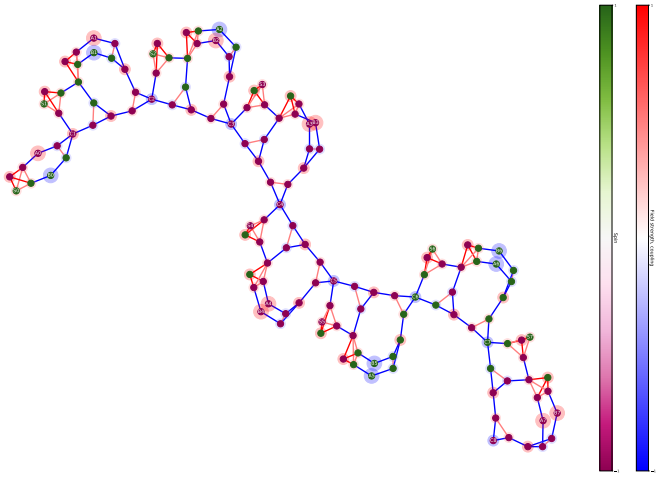

Figure 3: Ising implementation of an 8-bit ripple-carry adder computing $100 + 99 = 199$. The circuit is little-endian ordered (least significant bit first, e.g. $A_0A_1 \cdots A_7 + B_0B_1 \cdots B_7 = S_0S_1 \cdots S_7(C_8)$), so this corresponds to $00100110 + 11000110 = 11100011(0)$.

C. Circuit satisfiability

The circuit satisfiability (CSAT) problem is an NP-complete decision problem of determining whether a Boolean circuit with one output has some combination of inputs which makes the output true. This can be thought of as a sort of "circuit inversion" problem which is particularly suitable for Ising formulations of Boolean circuits. To solve this problem with a spin system, we compile the Boolean circuit and then treat the output node as the inputs, applying a strong magnetic field to force the spin to evaluate to true, and then allow the system to anneal to the lowest energy state.

Since each gate is implemented by a Hamiltonian which has degenerate ground states with a known energy for all logically correct evaluations, it is trivial to compute the ground state energy of the compiled circuit if a logically consistent solution is admissible. If the system consistently evolves into a state with a higher total energy, it indicates that the circuit may not be satisfiable.

Although this approach naturally lends itself to circuit inversion, it should be noted that one would expect that larger circuits would require an exponentially longer annealing time, so it is unlikely that this approach could provide more than a polynomial speedup over classical methods.

A demonstration of solving CSAT over satisfiable and un-satisfiable spin circuits is shown in Figure 4.

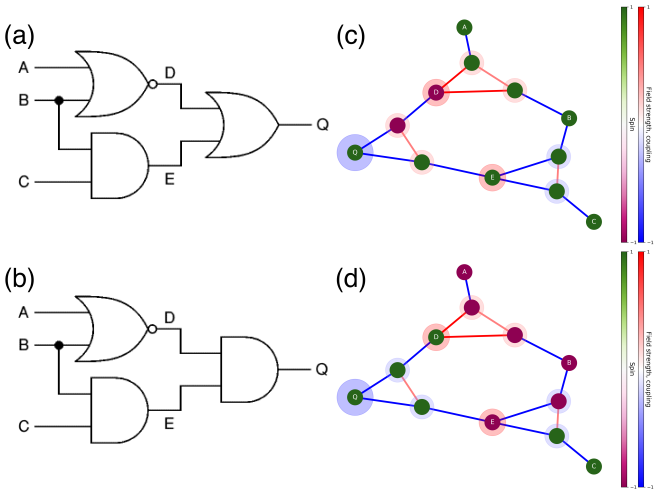

Figure 4: Circuit inversion for satisfiable and unsatisfiable circuits. (a) Satisfiable Boolean circuit. (b) Unsatisfiable Boolean circuit. (c,d) Spin system implementing the respective Boolean circuits. The final energy of (d) is higher than that of (c) since there is no logically consistent set of inputs that yields $Q = 1$.

IV. UNFINISHED WORK

I was unable to complete a few ideas which I originally wanted to pursue for this project. Most notably, the Ising model is typically introduced on a two-dimensional square lattice. With some additional engineering, it is possible to embed the spin graphs generated by the compiler into a 2D planar graph. Each elementary Boolean gate can be reformulated with additional ancillae to fit in a 2D square lattice, as shown in Figure 5. The four-spin XOR and XNOR gates are slightly more complicated to embed, but can be reformulated as a system of 9 spins. [4]

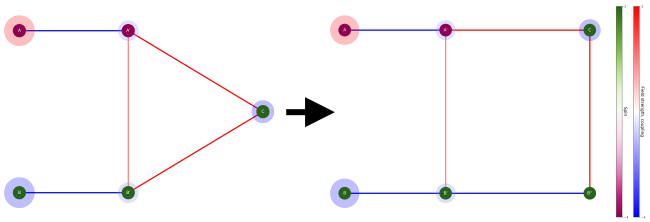

Figure 5: By adding an additional ancilla, the NAND gate can be reformulated to fit in a square lattice.

Once a set of elementary Boolean gates is derived which fits in a 2D square lattice, the main engineering challenge is figuring out an optimal layout and "wiring" the gates together. The problem of embedding a graph into a grid of minimum area is NP-hard [6], so a heuristic approach would likely need to be used for larger circuits. The approach I had laid out but not had time to implement was to connect gates with strings of COPY operations and place the gates in a grid to minimize the number of times the "wires" cross, as follows:

- Divide the grid into "columns" and place all inputs at the leftmost column.
- For each gate that is added to the circuit, place the gate in the lowest column available. (If the gate has inputs which are outputs of column i, place the gate in column $i + 1$.)
- For each column, compute an ensemble of orderings of the gates within rows and choose the row ordering which minimizes the number of "wire crossings" from the outputs of the previous column.
- For each column, use A^* (or some other pathfinding algorithm) to compute routes for the COPY wires to connect the inputs of the column to the outputs of the previous column. After the route of each wire is calculated, update the cost matrix for the pathfinder to increase the weights of nodes which are connected to wires to a high but finite value so that crossings are unfavorable but not impossible.
- After computing the wire routes, for each crossing, embed a SWAP gate to allow the wires to cross over each other.

I originally built versions of the `IsingGraph` class using NumPy and PyTorch backends which were more performant, but were limited to computing on n-dimensional square lattices. Once I realized that most of the effort of implementing a fully planar Ising compiler was a challenge of software engineering rather than statistical mechanics, I chose not to pursue this further.

V. CONCLUSION

In this project I showed that the Ising model, despite its simple structure, is sufficiently expressive to encode universal computation via Boolean logic in the ground states many-spin systems. Building from previous work [4, 5], I derived Ising formulations of all elementary Boolean logic gates and created a Python software package which can compile a sequence of gates into an Ising Hamiltonian over an arbitrary graph. I presented several demonstrations of the compiler implementing complex logical circuits and showed that the correct results of the computations were encoded in the ground state of the spin network.

[1] F. Barahona, "On the computational complexity of ising spin glass models," Journal of Physics A: Mathematical and General **15**, 3241–3253 (1982).

[2] Richard M. Karp, "Reducibility among combinatorial problems," (2010).

[3] Andrew Lucas, "Ising formulations of many NP problems," Frontiers in Physics **2**, 1–14 (2014).

[4] Mile Gu and Alvaro Perales, "Encoding universal computation in the ground states of Ising lattices," Physical Review E - Statistical, Nonlinear, and Soft Matter Physics **86**, 1–7 (2012).

[5] J. D. Whitfield, M. Faccin, and J. D. Biamonte, "Ground-state spin logic," Epl **99** (2012), 10.1209/0295-5075/99/57004.

[6] Michael Formann and Frank Wagner, "The VLSI layout problem in various embedding models," Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) **484 LNCS**, 130–139 (1991).