Classically Programming a Quantum Annealer

Virginia Tech CS Departmental Seminar

Scott Pakin

7 May 2019
Outline

• Background
• Problem statement
• Solution approach
• Enhancements
• Conclusions
Quantum Computing: Basic Concepts

• A classical bit is a scalar 0 or 1

• Four operations can be applied to a classical bit
  – Set to 0; set to 1; flip 0↔1; and do nothing (identity)

• A quantum bit (qubit) is a complex-valued 2-vector

How much “0-ness”

\[
\begin{pmatrix}
1 \\
0
\end{pmatrix}, \begin{pmatrix}
0 \\
1
\end{pmatrix}, \begin{pmatrix}
0 \\
-1
\end{pmatrix}, \begin{pmatrix}
\frac{1}{\sqrt{2}} \\
\frac{-1}{\sqrt{2}}
\end{pmatrix}, \begin{pmatrix}
\frac{i-\frac{3}{2}}{\sqrt{12i-1}}
\end{pmatrix}, \ldots
\]

How much “1-ness”

• Infinitely many operations can be applied to a qubit
  – Envision a qubit as a unit sphere
  – Any rotation in 3-space is a valid operation

• We say that a qubit that is not purely 0 or 1 lies in a superposition of 0 and 1
  – \(n\) qubits can effectively represent \(2^n\) values simultaneously
Quantum Computing: Basic Concepts (cont.)

• Measurement
  – While qubits can be in superpositions during a computation, one can observe only pure values (0s and 1s)
  – Amount of “0-ness” and “1-ness” determines the probability of observing a 0 or a 1
  – (All quantum computation is fundamentally stochastic)

• Correlations can be introduced among qubits
  – This is called entanglement
  – For example, one can prepare a quantum state such that all qubits will be measured as 0 or all qubits will be measured as 1

• Programming challenges
  – I/O bottleneck: Can effectively work with $2^n$ Boolean values during a computation but can input/output only $n$ Booleans
  – What problems require massive computation but neither input nor output much data?
  – How to cancel out the probability of non-solutions, leaving only solutions?

• A simpler form of quantum computing: quantum annealing
  – Subject of the rest of this talk
  – Caveat: not yet proven to deliver the full computational power of what quantum computing is capable of
Quantum Annealing

• Think simulated annealing in hardware
• Find the coordinates of the minimum value in an energy landscape
• Conceptual approach
  – Drop a bunch of rubber balls on the landscape, evaluating the function wherever they hit
  – Hope that one of the balls will bounce and roll downhill to the global minimum
• Problem: Commonly get stuck in a local minimum
• Solution: Use *quantum tunneling* to cut through tall, narrow barriers
How Quantum Annealing Works

• Approach due to Kadowaki and Nishimori, 1998
• Start in a trivial energy landscape
  – Qubits initialized to the solution to this known, trivial problem
• Gradually transition to the problem state
  – Decrease transverse-field strength
  – Increase longitudinal-field strength
• Premise (adiabatic theorem)
  – Sufficiently gradual transition → qubits remain in solution state
How Quantum Annealing Works (cont.)
Case Study: D-Wave Systems

- Commercial quantum annealer
- We have one installed at LANL
  - One of four customer installations
  - Can also pay for remote access to systems at D-Wave headquarters
- Current generation: D-Wave 2000Q, with up to 2048 qubits
- Try it yourself
  - D-Wave Leap: [https://cloud.dwavesys.com/leap](https://cloud.dwavesys.com/leap)
  - Free account, but limited compute time per month
Building Block: The Unit Cell

- Logical topology
  - 8 qubits arranged in a bipartite graph
- Physical implementation
  - Based on rf-SQUIDs
  - Flux qubits are long loops of superconducting wire interrupted by a set of Josephson junctions (weak links in superconductivity)
  - “Supercurrent” of Cooper pairs of electrons, condensed to a superconducting condensate, flows through the wires
  - Large ensemble of these pairs behaves as a single quantum state with net positive or net negative flux
  - ...or a superposition of the two (with tunneling)
  - Entanglement introduced at qubit intersections

Logical view

Physical view
A Complete Chip

• Logical view
  – “Chimera graph”: 16×16 unit-cell grid
  – Qubits 0–3 couple to north/south neighbors; 4–7 to east/west
  – Inevitably incomplete

• Physical view
  – Chip is about the size of a small fingernail
  – Can even make out unit cells with the naked eye
Cooling

• Chip must be kept extremely cold for the macroscopic circuit to behave like a two-level (qubit) system
  – Much below the superconducting transition temperature (9260 mK for niobium)
• Dilution refrigerator
• Nominally runs at 15 mK
• LANL’s D-Wave 2000Q happens to run at 12.26 mK
  – That’s 0.01°C above absolute zero
  – For comparison, interstellar space is far warmer: 2700 mK
What You Actually See

• A big, black box
  – 10’×10’×12’ (3m×3m×3.7m)
  – Mostly empty space
  – Radiation shielding, dilution refrigerator, chip + enclosure, cabling, tubing
  – LANL also had to add a concrete slab underneath to reduce vibration

• Support logic
  – Nondescript classical computers
  – Send/receive network requests, communicate with the chip, etc.
Outline

• Background
• Problem statement
• Solution approach
• Enhancements
• Conclusions
A Typical, Contemporary Instruction Set Architecture

- Large number of instructions with relatively few operands apiece
- Each performs a relatively simple, stateful operation
A Quantum Annealer’s Instruction Set Architecture

• One instruction with 8,064 operands (D-Wave 2000Q)
• Performs a very complex but stateless operation: minimizing a quadratic pseudo-Boolean function

\[
\arg \min_{\sigma} \left( \sum_{i=0}^{N-1} h_i \sigma_i + \sum_{i=0}^{N-2} \sum_{j=i+1}^{N-1} J_{i,j} \sigma_i \sigma_j \right)
\]

• You provide the real-valued \( h_i \) and \( J_{i,j} \) coefficients, and the hardware solves for the Boolean \( \sigma_i \) variables that minimize the expression
  – In this talk, “Boolean” means \{−1, +1\}, not \{0, 1\}
• This is an NP-hard problem
Problem Characteristics

$$\arg\min_\sigma \left( \sum_{i=0}^{N-1} h_i \sigma_i + \sum_{i=0}^{N-2} \sum_{j=i+1}^{N-1} J_{i,j} \sigma_i \sigma_j \right)$$

• A program is merely a list of real-valued $h$ and $J$ coefficients
  – No other inputs
  – Output is a list of Boolean values, $\sigma$

• This is a *classical* Hamiltonian function
  – All real-valued coefficients

• Thesis: We can map any classical problem into this form
  – How? That’s what the rest of this talk is about
  – Why would we want to? We’ll answer that soon enough…
Goal and Motivation

➤ How can we compile from ordinary—or even ordinary-ish—source code to a minimization problem that looks like

$$\arg \min_{\sigma} \left( \sum_{i=0}^{N-1} h_i \sigma_i + \sum_{i=0}^{N-2} \sum_{j=i+1}^{N-1} J_{i,j} \sigma_i \sigma_j \right)$$

• Motivation
  – The hardware can provide a heuristic solution in 1µs regardless of $N$
    (For $N = 2048$, an exact, brute-force solution would require an incomprehensibly large $2^{2048} \approx 3.2 \times 10^{616}$ function evaluations)
  – In theory, this implies a faster solution than can be achieved classically
  – Sadly, in practice, this has not yet been demonstrated on any real problem
Goal, Rephrased

• Sample input:

\[ c = (s == 1 \ ? \ a + b : a - b) \]

• Sample output:

\[
\arg \min_{\sigma} \left( -\frac{1}{12} \sigma_0 + \frac{1}{8} \sigma_3 - \frac{1}{12} \sigma_4 - \frac{1}{24} \sigma_6 - \frac{1}{24} \sigma_9 + \frac{1}{16} \sigma_{10} + \frac{1}{4} \sigma_{12} - \frac{1}{24} \sigma_{14} - \frac{1}{12} \sigma_{128} + \frac{1}{8} \sigma_{131} + \frac{1}{16} \sigma_{132} - \frac{1}{8} \sigma_{134} - \frac{1}{8} \sigma_{136} + \frac{1}{8} \sigma_{137} + \frac{1}{16} \sigma_{138} + \frac{1}{16} \sigma_{140} + \frac{1}{8} \sigma_{142} + \frac{1}{8} \sigma_{143} - \frac{1}{2} \sigma_0 \sigma_4 + \frac{1}{8} \sigma_0 \sigma_6 - \frac{1}{2} \sigma_0 \sigma_{128} - \frac{1}{4} \sigma_3 \sigma_4 - \frac{1}{4} \sigma_3 \sigma_5 - \frac{1}{2} \sigma_3 \sigma_{131} - \frac{1}{4} \sigma_4 \sigma_{12} - \frac{1}{8} \sigma_6 \sigma_{14} - \frac{1}{2} \sigma_8 \sigma_{13} - \frac{1}{4} \sigma_8 \sigma_{14} - \frac{1}{4} \sigma_9 \sigma_{12} - \frac{1}{4} \sigma_9 \sigma_{13} - \frac{1}{2} \sigma_9 \sigma_{14} + \frac{1}{4} \sigma_{10} \sigma_{12} - \frac{1}{8} \sigma_{10} \sigma_{14} - \frac{1}{2} \sigma_{10} \sigma_{138} - \frac{1}{8} \sigma_{128} \sigma_{132} + \frac{1}{8} \sigma_{128} \sigma_{134} - \sigma_{128} \sigma_{135} - \frac{1}{4} \sigma_{131} \sigma_{134} - \frac{1}{2} \sigma_{132} \sigma_{140} - \frac{1}{2} \sigma_{134} \sigma_{142} - \frac{1}{8} \sigma_{136} \sigma_{140} - \frac{1}{2} \sigma_{136} \sigma_{141} - \frac{1}{8} \sigma_{136} \sigma_{142} - \frac{1}{4} \sigma_{136} \sigma_{143} + \frac{1}{8} \sigma_{137} \sigma_{140} + \frac{1}{4} \sigma_{137} \sigma_{142} - \frac{1}{2} \sigma_{137} \sigma_{143} - \frac{1}{2} \sigma_{138} \sigma_{140} + \frac{1}{8} \sigma_{138} \sigma_{142} + \frac{1}{8} \sigma_{138} \sigma_{143} - \sigma_{139} \sigma_{140} \right)
\]
Outline

• Background
• Problem statement
• Solution approach
• Enhancements
• Conclusions
Interpreting our One Instruction

• Let’s start by considering only the linear coefficients ($h_i$):

$$\arg \min_{\sigma} \left( \sum_{i=0}^{N-1} h_i \sigma_i + \sum_{i=0}^{N-2} \sum_{j=i+1}^{N-1} J_{i,j} \sigma_i \sigma_j \right)$$

• We arbitrarily call $\sigma_i = +1 \text{ “TRUE” and } \sigma_i = -1 \text{ “FALSE”}$

• Here are the optimal values of $\sigma_i$ for different values of $h_i$:

<table>
<thead>
<tr>
<th>$\sigma_i$</th>
<th>$h_i \sigma_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>+5</td>
</tr>
<tr>
<td>+1</td>
<td>-5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$\sigma_i$</th>
<th>$h_i \sigma_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>0</td>
</tr>
<tr>
<td>+1</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$\sigma_i$</th>
<th>$h_i \sigma_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>-5</td>
</tr>
<tr>
<td>+1</td>
<td>+5</td>
</tr>
</tbody>
</table>

• Observations
  – A $\text{negative } h_i$ means, “I want $\sigma_i$ to be TRUE”
  – A $\text{zero } h_i$ means, “I don’t care if $\sigma_i$ is TRUE or FALSE”
  – A $\text{positive } h_i$ means, “I want $\sigma_i$ to be FALSE”
Now let’s consider only the quadratic terms ($J_{i,j}$):

$$\arg\min_{\sigma} \left( \sum_{i=0}^{N-1} h_i \sigma_i + \sum_{i=0}^{N-2} \sum_{j=i+1}^{N-1} J_{i,j} \sigma_i \sigma_j \right)$$

Here are the optimal values of $\sigma_i$ and $\sigma_j$ for different values of $J_{i,j}$:

<table>
<thead>
<tr>
<th>$\sigma_i$</th>
<th>$\sigma_j$</th>
<th>$J_{i,j} \sigma_i \sigma_j$</th>
<th>$\sigma_i$</th>
<th>$\sigma_j$</th>
<th>$J_{i,j} \sigma_i \sigma_j$</th>
<th>$\sigma_i$</th>
<th>$\sigma_j$</th>
<th>$J_{i,j} \sigma_i \sigma_j$</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>-1</td>
<td>-5</td>
<td>-1</td>
<td>-1</td>
<td>0</td>
<td>-1</td>
<td>-1</td>
<td>+5</td>
</tr>
<tr>
<td>-1</td>
<td>+1</td>
<td>+5</td>
<td>-1</td>
<td>+1</td>
<td>0</td>
<td>-1</td>
<td>+1</td>
<td>-5</td>
</tr>
<tr>
<td>+1</td>
<td>-1</td>
<td>+5</td>
<td>+1</td>
<td>-1</td>
<td>0</td>
<td>+1</td>
<td>+1</td>
<td>-5</td>
</tr>
<tr>
<td>+1</td>
<td>+1</td>
<td>-5</td>
<td>+1</td>
<td>+1</td>
<td>0</td>
<td>+1</td>
<td>+1</td>
<td>+5</td>
</tr>
</tbody>
</table>

Observations
- A negative $J_{i,j}$ means, “I want $\sigma_i$ and $\sigma_j$ to be equal”
- A zero $J_{i,j}$ means, “I don’t care how $\sigma_i$ and $\sigma_j$ are related”
- A positive $J_{i,j}$ means, “I want $\sigma_i$ and $\sigma_j$ to be different”
Interpretation

• Look what we can express so far as $\arg\min_{\sigma} (\mathcal{H}(\sigma))$:

<table>
<thead>
<tr>
<th>Component</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>ground</td>
<td>$\mathcal{H}_{\text{GND}} = \sigma_g$</td>
</tr>
<tr>
<td>power</td>
<td>$\mathcal{H}_{\text{VCC}} = -\sigma_v$</td>
</tr>
<tr>
<td>wire</td>
<td>$\mathcal{H}_{\text{wire}} = -\sigma_A \sigma_Y$</td>
</tr>
<tr>
<td>inverter</td>
<td>$\mathcal{H}_{\neg} = \sigma_A \sigma_Y$</td>
</tr>
</tbody>
</table>
Approach to Constructing Other Logic Gates

• Write a *complete* truth table, distinguishing *valid* from *invalid* rows
• Set up a system of inequalities
  – All valid rows must evaluate to the same value
  – All invalid rows must evaluate to a value greater than that of any valid row
• Example: 2-input AND gate \((Y = A \land B)\)

<table>
<thead>
<tr>
<th>(\sigma_A)</th>
<th>(\sigma_B)</th>
<th>(\sigma_Y)</th>
<th>( \sum_{h=0}^{N-1} h_i \sigma_i + \sum_{i=0}^{N-2} \sum_{j=i+1}^{N-1} J_{i,j} \sigma_i \sigma_j )</th>
<th>Must be</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>(-h_A - h_B - h_Y + J_{A,B} + J_{A,Y} + J_{B,Y})</td>
<td>(= k)</td>
</tr>
<tr>
<td>-1</td>
<td>-1</td>
<td>+1</td>
<td>(-h_A - h_B + h_Y + J_{A,B} - J_{A,Y} - J_{B,Y})</td>
<td>(&gt; k)</td>
</tr>
<tr>
<td>-1</td>
<td>+1</td>
<td>-1</td>
<td>(-h_A + h_B - h_Y - J_{A,B} + J_{A,Y} - J_{B,Y})</td>
<td>(= k)</td>
</tr>
<tr>
<td>-1</td>
<td>+1</td>
<td>+1</td>
<td>(-h_A + h_B + h_Y - J_{A,B} - J_{A,Y} + J_{B,Y})</td>
<td>(&gt; k)</td>
</tr>
<tr>
<td>+1</td>
<td>-1</td>
<td>-1</td>
<td>(+h_A - h_B - h_Y - J_{A,B} - J_{A,Y} + J_{B,Y})</td>
<td>(= k)</td>
</tr>
<tr>
<td>+1</td>
<td>-1</td>
<td>+1</td>
<td>(+h_A - h_B + h_Y - J_{A,B} + J_{A,Y} - J_{B,Y})</td>
<td>(&gt; k)</td>
</tr>
<tr>
<td>+1</td>
<td>+1</td>
<td>-1</td>
<td>(+h_A + h_B - h_Y + J_{A,B} - J_{A,Y} - J_{B,Y})</td>
<td>(&gt; k)</td>
</tr>
<tr>
<td>+1</td>
<td>+1</td>
<td>+1</td>
<td>(+h_A + h_B + h_Y + J_{A,B} + J_{A,Y} + J_{B,Y})</td>
<td>(= k)</td>
</tr>
</tbody>
</table>
Expressing Logic Gates as Hamiltonians (cont.)

• Problem: Not all $N$-input gates can be expressed with $N+1$ qubits
  – System of inequalities may be unsolvable
  – Example: 2-input XOR ($Y = A \oplus B$)

<table>
<thead>
<tr>
<th>$\sigma_A$</th>
<th>$\sigma_B$</th>
<th>$\sigma_Y$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$-1$</td>
<td>$-1$</td>
<td>$-1$</td>
</tr>
<tr>
<td>$-1$</td>
<td>$-1$</td>
<td>$+1$</td>
</tr>
<tr>
<td>$-1$</td>
<td>$+1$</td>
<td>$-1$</td>
</tr>
<tr>
<td>$-1$</td>
<td>$+1$</td>
<td>$+1$</td>
</tr>
<tr>
<td>$+1$</td>
<td>$-1$</td>
<td>$-1$</td>
</tr>
<tr>
<td>$+1$</td>
<td>$-1$</td>
<td>$+1$</td>
</tr>
<tr>
<td>$+1$</td>
<td>$+1$</td>
<td>$-1$</td>
</tr>
<tr>
<td>$+1$</td>
<td>$+1$</td>
<td>$+1$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$\sigma_A$</th>
<th>$\sigma_B$</th>
<th>$\sigma_Y$</th>
<th>$\sigma_a$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$-1$</td>
<td>$-1$</td>
<td>$-1$</td>
<td>$-1$</td>
</tr>
<tr>
<td>$-1$</td>
<td>$-1$</td>
<td>$+1$</td>
<td>$+1$</td>
</tr>
<tr>
<td>$-1$</td>
<td>$+1$</td>
<td>$+1$</td>
<td>$+1$</td>
</tr>
<tr>
<td>$-1$</td>
<td>$+1$</td>
<td>$+1$</td>
<td>$+1$</td>
</tr>
<tr>
<td>$+1$</td>
<td>$-1$</td>
<td>$-1$</td>
<td>$+1$</td>
</tr>
<tr>
<td>$+1$</td>
<td>$-1$</td>
<td>$+1$</td>
<td>$+1$</td>
</tr>
<tr>
<td>$+1$</td>
<td>$+1$</td>
<td>$-1$</td>
<td>$+1$</td>
</tr>
<tr>
<td>$+1$</td>
<td>$+1$</td>
<td>$+1$</td>
<td>$+1$</td>
</tr>
</tbody>
</table>

• Solution: Introduce ancilla qubits for more degrees of freedom
  – Keep same number of valid rows
  – How many ancillas and which rows should be valid? That’s an open question.
Logic-Gate Examples

- We can express any gate as $\arg \min_{\sigma} \left( \sum_{i=0}^{N-1} h_i \sigma_i + \sum_{i=0}^{N-2} \sum_{j=i+1}^{N-1} J_{i,j} \sigma_i \sigma_j \right)$

- Some examples:

<table>
<thead>
<tr>
<th>Gate</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="AND Gate" /></td>
<td>$\mathcal{H}_\wedge = -\frac{1}{2} \sigma_A - \frac{1}{2} \sigma_B + \sigma_Y + \frac{1}{2} \sigma_A \sigma_B - \sigma_A \sigma_Y - \sigma_B \sigma_Y$</td>
</tr>
<tr>
<td><img src="image" alt="XOR Gate" /></td>
<td>$\mathcal{H}_\oplus = \frac{1}{2} \sigma_A + \frac{1}{2} \sigma_B + \frac{1}{2} \sigma_Y + \sigma_a + \frac{1}{2} \sigma_A \sigma_B + \frac{1}{2} \sigma_A \sigma_Y + \sigma_A \sigma_a + \frac{1}{2} \sigma_B \sigma_Y + \sigma_B \sigma_a + \sigma_Y \sigma_a$</td>
</tr>
<tr>
<td><img src="image" alt="OR Gate" /></td>
<td>$\mathcal{H}_\vee = \frac{1}{2} \sigma_A + \frac{1}{2} \sigma_B - \sigma_Y + \frac{1}{2} \sigma_A \sigma_B - \sigma_A \sigma_Y - \sigma_B \sigma_Y$</td>
</tr>
</tbody>
</table>

- Important feature: expressions can be added
  - That is, $\text{gate} + \text{wire} + \text{gate} = \text{circuit}$
A Standard-Cell Library

- Implement using QMASM, a quantum macro assembler
  - Open-source software, available from https://github.com/lanl/qmasm
- Macros
  - Define reusable components (e.g., gates) that can be instantiated repeatedly
- Symbolic variable manipulation
  - Named variables in place of physical qubit numbers
  - Automatic place-and-route to physical topology
- Include files
  - Multiple files can include the same macro-collection file (e.g., a standard cell library)

\[
\mathcal{H}_\Lambda = -\frac{1}{2} \sigma_A - \frac{1}{2} \sigma_B + \sigma_Y + \frac{1}{2} \sigma_A \sigma_B - \sigma_A \sigma_Y - \sigma_B \sigma_Y
\]

```
# Y = A AND B
!begin_macro AND
A -0.5
B -0.5
Y 1
A B 0.5
A Y -1
B Y -1
!end_macro AND
```

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
</tbody>
</table>

\[a_{12} = 0.5, a_{21} = 0.5, \ldots\]
Converting Circuits to QMASM

• Begin with a digital circuit expressed as an EDIF netlist
  – EDIF = Electronic Data Interchange Format
  – Semi-standard way to represent digital circuits as s-expressions
• Convert from EDIF to QMASM using edif2qmasm
  – Gates: EDIF cell instances → QMASM macro instantiations ("!use_macro")
  – Wires: EDIF nets → QMASM chains ("=")

![Diagram of a digital circuit with gates and wires](image)

```qmasm
!include <stdcell>

!begin_macro example
!use_macro AND $id00005
!use_macro NOT $id00004
!use_macro OR $id00006
$id00004.A = C
$id00005.A = A
$id00005.B = B
$id00006.A = $id00005.Y
$id00006.B = $id00004.Y
$id00006.Y = Y
!end_macro example

!use_macro example example
```
Converting Code to Circuits

• Leverage existing hardware description languages (HDLs)
  – Look more-or-less like an ordinary programming language
  – Multi-bit variables, arithmetic operators, relational operators, conditionals, loops, modules, …

• Hardware synthesis tools compile HDLs to logic primitives
  – AND, OR, NOT, XOR, …
  – Optimizations to reduce # of gates

module example (A, B, C, Y);
input A, B, C;
output Y;
assign Y = (A&B) | ~C;
endmodule

• My toolbox
  – HDL: Verilog (first introduced in 1984)
  – Hardware synthesis tool: Yosys (https://github.com/cliffordwolf/yosys)
    with additional optimizations provided by ABC (https://bitbucket.org/alanmi/abc)
A More Interesting Example

- Given a map and one of four colors for each region, return TRUE if and only if no two adjacent regions share a color.
- Very little Verilog code is required to express this.

```verilog
module australia (NSW, QLD, SA, VIC, WA, NT, ACT, valid);
    input [1:0] NSW, QLD, SA, VIC, WA, NT, ACT;
    output valid;
    assign valid = WA != NT && WA != SA && NT != SA && NT != QLD && SA != QLD && SA != NSW && SA != VIC && QLD != NSW && NSW != VIC && NSW != ACT;
endmodule
```

TRUE
Why Should You Care?

• Our approach actually constructs a relation, not a function
  – \( \arg \min_{\sigma} \mathcal{H}(\sigma) \) corresponds to a valid association of inputs and outputs
• We can bias any subset of inputs and outputs to \text{TRUE} \((h_i < 0)\) or \text{FALSE} \((h_i > 0)\) and solve for the remaining values
  ➔ Effectively, we can run a computation backwards

• Consider problems in the NP complexity class
  – Slow to compute classically
  – Fast to verify a proposed solution
• Proposed approach to solving problems in NP
  – Solve the (easier) inverse problem and run the code backwards from “passes verification” to a set of inputs that pass verification
Solving a “Real” Map-Coloring Problem

• Color a map using four colors such that no two adjacent regions share a color
• NP-complete problem

module australia (NSW, QLD, SA, VIC, WA, NT, ACT, valid);
  input [1:0] NSW, QLD, SA, VIC, WA, NT, ACT;
  output valid;

  assign valid = WA != NT && WA != SA && NT != SA && NT != QLD && SA != QLD && SA != NSW && SA != VIC && QLD != NSW && NSW != VIC && NSW != ACT;
endmodule
• This is one large Hamiltonian function!
• Comparison to hand-coded implementation of map coloring
  – Unary encoding; regions replicated to overcome sparse connectivity and laid out spatially as atomic units
  – Result: 369 ± 26 qubits mechanical vs. 88 qubits hand-coded (4x)
Other Applications

- Any problem in NP can be (heuristically) solved with our approach and tools
- Here are a couple of examples, in addition to map coloring:

```
module circsat (a, b, c, y);
    input a, b, c;
    output y;
    wire [1:10] x;

    assign x[1] = a;
    assign x[2] = b;
    assign x[3] = c;
    assign x[4] = ~x[3];
    assign x[6] = ~x[4];
    assign x[9] = x[6] | x[7];
    assign y = x[10];
endmodule
```

```
module mult (multiplicand, multiplier, product);
    input [3:0] multiplicand;
    input [3:0] multiplier;
    output [7:0] product;

    assign product = multiplicand * multiplier;
endmodule
```

Factoring as inverse multiplication (NP-intermediate)

Circuit satisfiability (NP-complete)
Outline

- Background
- Problem statement
- Solution approach
- Enhancements
- Conclusions
A Higher-Level Language

• Verilog makes for a good proof of concept, but can we compile a high-level language to the D-Wave?

• Main challenge: many desirable features are qubit-hungry
  – Mutable state—requires entire program to be replicated for each state change
  – Data structures or even just addressible memory
  – 64-bit arithmetic—merely storing the inputs and outputs of a single 64-bit addition consumes 9% of a D-Wave 2000Q’s qubits

• Let’s try to implement a language that doesn’t emphasize such things…
The Prolog Programming Language

• “Programmation en logique”
  – Or, “Programming in logic”

• Programming language based on Horn clauses
  – Very different form of programming from, say, Python or C++

• Initially promoted for use in symbolic AI

• Formed the core of Japan’s Fifth-Generation Computer project, 1982–1992
  – Dataflow hardware optimized for running Prolog and targeting AI applications

• Never really caught on
  – Typically relegated to a brief mention in introductory Programming Languages classes
Prolog Code Execution

• Given the code shown to the right, the Prolog system solves for variable What
  – That is, it disproves the claim that there is no value that can be assigned to What

• Effective control flow
  – \( \text{likes(sophia, What)}. \) “I must find a What that makes this statement \text{TRUE}.”
  – \( \text{likes(sophia, X)} :- \text{likes(scott, X)}. \) “If I can prove that Scott likes \( X \), then I can prove that Sophia likes \( X \).”
  – \( \text{likes(scott, dwave)}. \) “I can prove that Scott likes the D-Wave.”
  – \( \text{likes(sophia, dwave)}. \) “By unifying \( X \) with \text{dwave}, I can prove that Sophia likes the D-Wave.”
  – \( \text{What} = \text{dwave} \) QED. Proof by contradiction.
Key Prolog Concepts

• Unification
  – Assigning values to variables to make patterns match
  – *Example 1*: Unification succeeds in :- knows(A, B), female(A), male(B) by binding A to dianne, B to vern, and (internally) C to dwave
  – *Example 2*: Unification fails in :- knows(marcus, W)
• Predicates can complete zero, one, or more times
  – Prolog returns *all* valid variable assignments
  – *Example*: :- knows(A, B) returns both {A=dianne, B=vern} and {A=vern, B=dianne} as well as {A=vern, B=vern}, {A=dianne, B=dianne}, {A=chad, B=chad}, and {A=italia, B=italia}
  – If there are no variables in the goal, Prolog returns TRUE if the goal is a provably true statement or FALSE if it is not provably true
  – *Example*: :- works_at(talia, ibm) returns TRUE, but :- works_at(talia, dwave) returns FALSE

```prolog
male(vern).
male(chad).
female(dianne).
female(talia).
works_at(vern, dwave).
works_at(chad, rigetti).
works_at(dianne, dwave).
works_at(talia, ibm).

knows(P1, P2) :-
  works_at(P1, C),
  works_at(P2, C).
```
Implementation

Prolog → Verilog → EDIF → QMASM → $H$

- Logic programming language
- Hardware description language
- Netlist format
- Quantum macro assembler
- Physical, 2-local Ising-model Hamiltonian function

High-level symbolic and constraint-logic programming constructs
Support for multi-bit arithmetic and relational operators with the ability to compile to simple primitives (logic gates)
Precise specification of inter-gate connectivity
Logical (hardware-independent), symbolic Hamiltonians, macros for representing sub-problems
Ability to run on a D-Wave quantum annealer
Step 0: Prolog

- Let’s use our `knows` example from a couple slides back
  - Large enough to be interesting
  - Small enough to fit on a slide
  - (And generated intermediate files come close to fitting on a slide)

```
male(vern).
male(chad).
female(dianne).
female(talia).
works_at(vern, dwave).
works_at(chad, rigetti).
works_at(dianne, dwave).
works_at(talia, ibm).

knows(P1, P2) :-
    works_at(P1, C),
    works_at(P2, C).

:- knows(A, B),
   female(A),
   male(B).
```
Step 1: Verilog

- Almost a 1:1 mapping from Prolog predicates to Verilog modules

```
// Define all of the symbols
used in this program.
'define vern 3'd0
'define chad 3'd1
'define dianne 3'd2
'define dwave 3'd3
'define ibm 3'd4
'define rigetti 3'd5
'define talia 3'd6

// Define Query(atom, atom).
module Query (A, B, Valid);
  input [2:0] A;
  input [2:0] B;
  output Valid;
  wire [2:0] $v1;
  wire [2:0] $v2;
  assign Valid = &$v1 | &$v2;
endmodule

// Define knows(atom, atom).
module knows/2 (A, B, Valid);
  input [2:0] A;
  input [2:0] B;
  output Valid;
  (* keep *) wire [2:0] C;
  wire [1:0] $v1;
  wire [1:0] $v2;
  assign $v1 = A == `vern;
  assign $v2 = A == `chad;
  assign Valid = &$v1 | &$v2;
endmodule

// Define works_at/2
module works_at/2 (B, C, $v1[1]);
  assign Valid = &$v1;
endmodule

// Define male(atom).
module male/1 (A, Valid);
  input [2:0] A;
  output Valid;
  wire $v1;
  assign $v1 = A == `vern;
endmodule
```
Step 1: Verilog (cont.)

• ...and here are the last two modules:

```verilog
// Define female(atom).
module female/1 (A, Valid);
  input [2:0] A;
  output Valid;
  wire $v1;
  assign $v1 = A == `dianne;
  wire $v2;
  assign $v2 = A == `talia;
  assign Valid = &$v1 | &$v2;
endmodule

// Define works_at(atom, atom).
module works_at/2 (A, B, Valid);
  input [2:0] A;
  input [2:0] B;
  output Valid;
  wire [1:0] $v1;
  assign $v1[0] = A == `vern;
  assign $v1[1] = B == `dwave;
  wire [1:0] $v2;
  assign $v2[0] = A == `chad;
  assign $v2[1] = B == `rigetti;
  wire [1:0] $v3;
  assign $v3[0] = A == `dianne;
  assign $v3[1] = B == `dwave;
  wire [1:0] $v4;
  assign $v4[0] = A == `talia;
  assign $v4[1] = B == `ibm;
  assign Valid = &$v1 | &$v2 | &$v3 | &$v4;
endmodule
```

• Key idea: Input all predicate arguments and output a Valid bit
Remaining Steps

• Same as in the case where we started with Verilog:
  – **Step 2**: Synthesize to a digital circuit (EDIF format)
  – **Step 3**: Translate to a logical Hamiltonian function (QMASM format) with gates implemented as macro instantiations and wires implemented as equality constraints
  – **Step 4**: Compile to a physical Hamiltonian function targeting a particular D-Wave instance

• Representative embedding statistics for the Hamiltonian functions:

<table>
<thead>
<tr>
<th>Metric</th>
<th>Type</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear terms ($h_i$)</td>
<td>Logical</td>
<td>108</td>
</tr>
<tr>
<td></td>
<td>Physical</td>
<td>282</td>
</tr>
<tr>
<td>Quadratic terms ($j_{i,j}$)</td>
<td>Logical</td>
<td>185</td>
</tr>
<tr>
<td></td>
<td>Physical</td>
<td>365</td>
</tr>
</tbody>
</table>
$ qa-prolog --verbose --qmasm-args="-O2 -v --postproc=opt" --query="knows(A, B), female(A), male(B)." works_at.pl

qa-prolog: INFO: Parsing works_at.pl as Prolog code
qa-prolog: INFO: Representing symbols with 3 bit(s) and integers with 1 bit(s)
qa-prolog: INFO: Storing intermediate files in works_at
qa-prolog: INFO: Writing Verilog code to works_at.v
qa-prolog: INFO: Writing a Yosys synthesis script to works_at.ys
qa-prolog: INFO: Converting Verilog code to an EDIF netlist
qa-prolog: INFO: Executing yosys -q works_at.v works_at.ys -b edif -o works_at.edif
qa-prolog: INFO: Converting the EDIF netlist to QMASM code
qa-prolog: INFO: Executing edif2qmasm -o works_at.qmasm works_at.edif
qa-prolog: INFO: Executing qmasm --run --values=ints -O2 -v --postproc=opt --pin=Query.Valid := true works_at.qmasm
A = dianne
B = vern

• Open-source software, available from https://github.com/lanl/QA-Prolog
Outline

• Background
• Problem statement
• Solution approach
• Enhancements
• Conclusions
Conclusions

- A quantum annealer heuristically solves in hardware a single, parameterized optimization problem,

\[
\arg \min_\sigma \left( \sum_{i=0}^{N-1} h_i \sigma_i + \sum_{i=0}^{N-2} \sum_{j=i+1}^{N-1} J_{i,j} \sigma_i \sigma_j \right)
\]

- We have shown that it is possible to compile classical code into that form
  - [Constraint logic program →] hardware description language → digital circuit → symbolic quadratic pseudo-Boolean function → physical quadratic pseudo-Boolean function of the form shown above

- Insight
  - Easy but slow: Brute-force solve a computationally expensive problem
  - Difficult but fast: Approximately solve a computationally expensive problem
  - Easy and fast: Use our approach and tools to approximately solve a computationally expensive problem by solving the simpler inverse problem