Programmation d'un D-Wave en Logique

Qubits 2018 D-Wave Users Conference

Knoxville, Tennessee

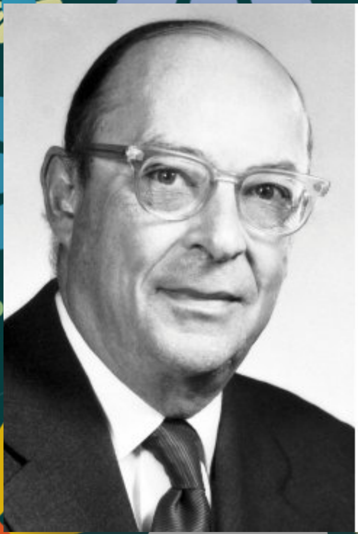
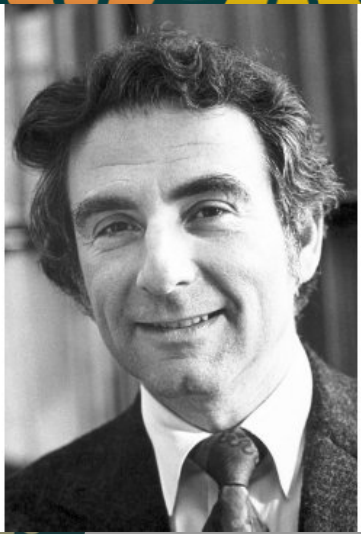
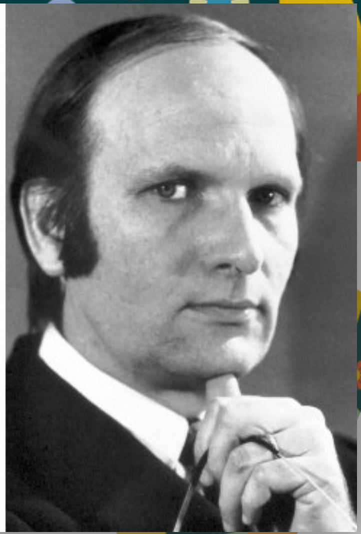
Scott Pakin

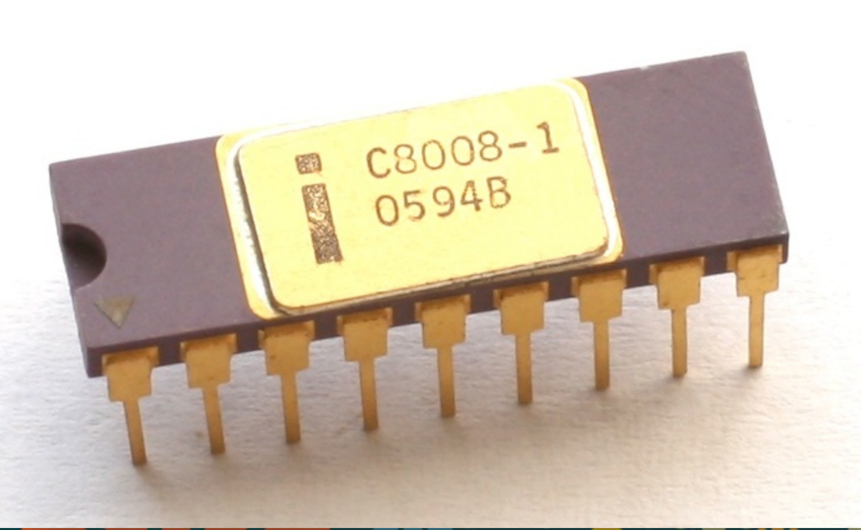
26 September 2018

LA-UR-18-28574

Outline

- A bunch of concepts that seem totally unrelated to each other
- Putting it all together
- Conclusions


C8008-1
0594B


Horn Clauses

- **Named after Alfred Horn**
  - Horn, Alfred. “On Sentences Which are True of Direct Unions of Algebras.” *Journal of Symbolic Logic*, 16(1):14–21, 1951, DOI: 10.2307/2268661
- **Disjunction of literals with at most one unnegated literal**
- **Three types of Horn clauses:**

Type	Disjunction form	Implication form	Example
Fact	u	u	“Scott likes the D-Wave.”
Rule	$\neg p \vee \neg q \vee \cdots \vee \neg t \vee u$	$u \leftarrow p \wedge q \wedge \cdots \wedge t$	“Sophia likes *X* if Scott likes *X*.”
Goal	$\neg p \vee \neg q \vee \cdots \vee \neg t$	$\textsc{False} \leftarrow p \wedge q \wedge \cdots \wedge t$	“There is nothing that Sophia likes.”

- **Executable form of logic**
  - Execution consists of the system deriving a contradiction to the goal
  - Equivalent in computational power to a universal Turing machine

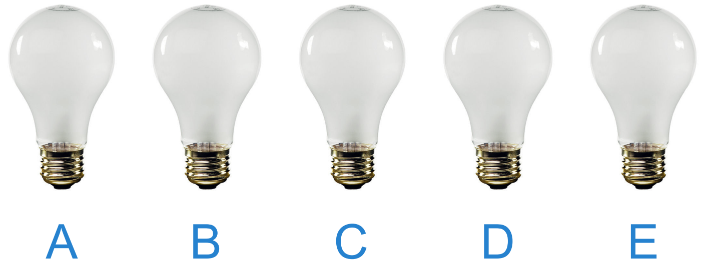
The D-Wave's Native Programming Model

- **What a D-Wave looks like to me:**
  - Minimize $\mathcal{H}(\bar{\sigma}) = \sum_{i=0}^{N-1} h_i \sigma_i + \sum_{i=0}^{N-2} \sum_{j=i+1}^{N-1} J_{i,j} \sigma_i \sigma_j$
    given $h_i \in \mathbb{R}$, $J_{i,j} \in \mathbb{R}$, and solving for $\sigma_i \in \{-1, +1\}$
- **A slightly more realistic formulation:**
  - Minimize $\mathcal{H}(\bar{\sigma}) = \sum_{\langle i \rangle} h_i \sigma_i + \sum_{\langle i,j \rangle} J_{i,j} \sigma_i \sigma_j$
    given $h_i \in [-2, 2]$, $J_{i,j} \in [-1, 1]$, and solving for $\sigma_i \in \{-1, +1\}$
  - That is, only a limited, system-specific subset of coefficients can be nonzero, and those have limited range
- **What the hardware actually does:**
  - Minimize $\mathcal{H}(\bar{\sigma}, s) = \frac{A(s)}{2} \left(\sum_{\langle i \rangle} \sigma_i^x \right) + \frac{B(s)}{2} \left(\sum_{\langle i \rangle} h_i \sigma_i^z + \sum_{\langle i,j \rangle} J_{i,j} \sigma_i^z \sigma_j^z \right)$
    given a hardware-specific annealing schedule ($A(s)$ and $B(s)$) over time $s \in [0,1]$
- **It's in fact slightly more complicated than that**
  - The h_i and $J_{i,j}$ coefficients have a time-dependent Gaussian distribution
  - External noise, crosstalk, manufacturing infidelities, and other unknowns

A D-Wave 2000Q at D-Wave headquarters in Burnaby, British Columbia

Building our Building Blocks

- **Programming a D-Wave involves defining the h_i and $J_{i,j}$ coefficients for the 2-local Ising-model Hamiltonian function from the previous slide**
  - $\mathcal{H}(\bar{\sigma}) = \sum_{i=0}^{N-1} h_i \sigma_i + \sum_{i=0}^{N-2} \sum_{j=i+1}^{N-1} J_{i,j} \sigma_i \sigma_j$
- **One programming approach**
  - Define a set of small Hamiltonians that correspond to repeated subproblems
  - Solve the small Hamiltonians in the *reverse* direction from what the D-Wave does: Given the σ_i variables, solve for the h_i and $J_{i,j}$ coefficients
  - Combine the small Hamiltonians to form a complete problem
  - Solve the complete problem on the D-Wave in the forward direction: Given the h_i and $J_{i,j}$ coefficients, solve for the σ_i variables
- **Sample problem**
  - Configure five lights, labeled A–E, such that exactly one of A, B, and C is on, exactly one of B, C, and D is on, and exactly one of C, D, and E is on



Solving a Subproblem

- **Subproblem to solve**
  - Exactly 1 of 3 lights must be on—will apply to {A, B, C}, {B, C, D}, and {C, D, E}
- **Approach**
  - Set up and solve a system of inequalities
  - Constrain valid truth-table rows to have energy *k* and invalid rows to have energy >*k*

σ_0	σ_1	σ_2	$\sum_{i=0}^{N-1} h_i \sigma_i + \sum_{i=0}^{N-2} \sum_{j=i+1}^{N-1} J_{i,j} \sigma_i \sigma_j$	**Must be**
−1	−1	−1	$-h_0 - h_1 - h_2 + J_{0,1} + J_{0,2} + J_{1,2}$	$> k$
−1	−1	+1	$-h_0 - h_1 + h_2 + J_{0,1} - J_{0,2} - J_{1,2}$	$= k$
−1	+1	−1	$-h_0 + h_1 - h_2 - J_{0,1} + J_{0,2} - J_{1,2}$	$= k$
−1	+1	+1	$-h_0 + h_1 + h_2 - J_{0,1} - J_{0,2} + J_{1,2}$	$> k$
+1	−1	−1	$+h_0 - h_1 - h_2 - J_{0,1} - J_{0,2} + J_{1,2}$	$= k$
+1	−1	+1	$+h_0 - h_1 + h_2 - J_{0,1} + J_{0,2} - J_{1,2}$	$> k$
+1	+1	−1	$+h_0 + h_1 - h_2 + J_{0,1} - J_{0,2} - J_{1,2}$	$> k$
+1	+1	+1	$+h_0 + h_1 + h_2 + J_{0,1} + J_{0,2} + J_{1,2}$	$> k$

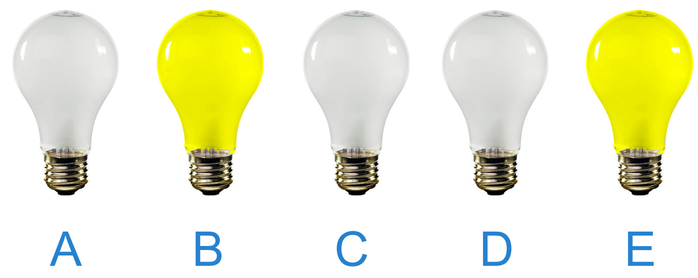
One solution: $\mathcal{H}_{1\text{of}3}(\sigma_0, \sigma_1, \sigma_2) = \sigma_0 + \sigma_1 + \sigma_2 + \sigma_0\sigma_1 + \sigma_0\sigma_2 + \sigma_1\sigma_2$, with $k = -2$

Constructing the Full Problem

- **Given the solution to the subproblem,**
  - $\mathcal{H}_{1\text{of}3}(\sigma_0, \sigma_1, \sigma_2) = \sigma_0 + \sigma_1 + \sigma_2 + \sigma_0\sigma_1 + \sigma_0\sigma_2 + \sigma_1\sigma_2$

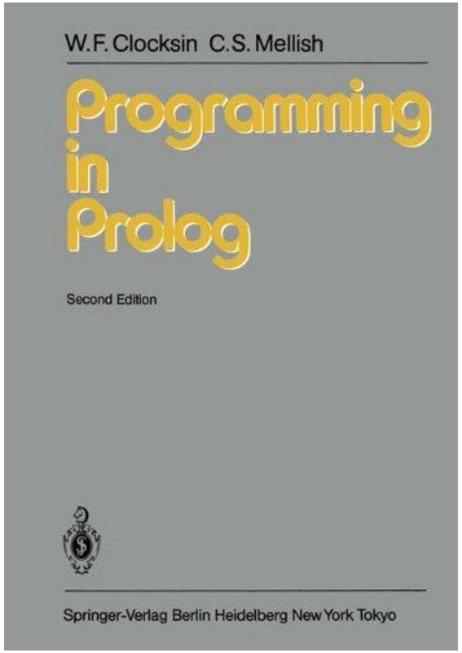
  we can simply add instances of that to define our full problem:
  - $\mathcal{H}(A, B, C, D, E) = \mathcal{H}_{1\text{of}3}(A, B, C) + \mathcal{H}_{1\text{of}3}(B, C, D) + \mathcal{H}_{1\text{of}3}(C, D, E)$

    which expands to
  - $\mathcal{H}(A, B, C, D, E) = A + 2B + 3C + 2D + E + AB + AC + 2BC + BD + 2CD + CE + DE$
- **This can be passed to a D-Wave system for solution**
  - Hint: three valid solutions out of 32 possible configurations of the five lights



The Prolog Programming Language

- **“Programmation en logique”**
  - Or, “Programming in logic”
  - Hence, the title of this talk
- **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



*The first Prolog system was produced by Colmerauer and Roussel in **1972** (Marseille, France). Clocksin and Mellish’s popular textbook came out about ten years later.*

Prolog Code Execution

```
likes(scott, dwave).
likes(sophia, X) :-
   likes(scott, X).
:- likes(sophia, What).
```

- **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**
  - `:- likes(sophia, What).` "I must find a *What* that makes this statement TRUE."
  - `likes(sophia, X) :- likes(scott, X).` "If I can prove that Scott likes *X*, then I can prove that Sophia likes *X*."
  - `likes(scott, dwave).` "I can prove that Scott likes the D-Wave."
  - `likes(sophia, dwave).` "By unifying *X* with dwave, I can prove that Sophia likes the D-Wave."
  - `What = dwave` QED. Proof by contradiction.

Sophia Pakin holding a D-Wave chip and enclosure

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 bo, 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=bo} and {A=bo, B=dianne} as well as {A=bo, B=bo}, {A=dianne, B=dianne}, {A=chad, B=chad}, and {A=talia, B=talia}
  - 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

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

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

Key Prolog Concepts (cont.)

- **Backtracking**
  - If unification fails at any point, Prolog backs up and tries again with alternative facts and rules
  - *Example*: :- knows(A, B), female(A), male(B)
    - Need to satisfy knows(A, B)
    - Possible solution: A=bo, B=bo
    - Need to satisfy female(bo)
    - Fail; backtrack to knows(A, B)
    - Possible solution: A=bo, B=dianne
    - Need to satisfy female(bo)
    - Fail; backtrack to knows(A, B)

…

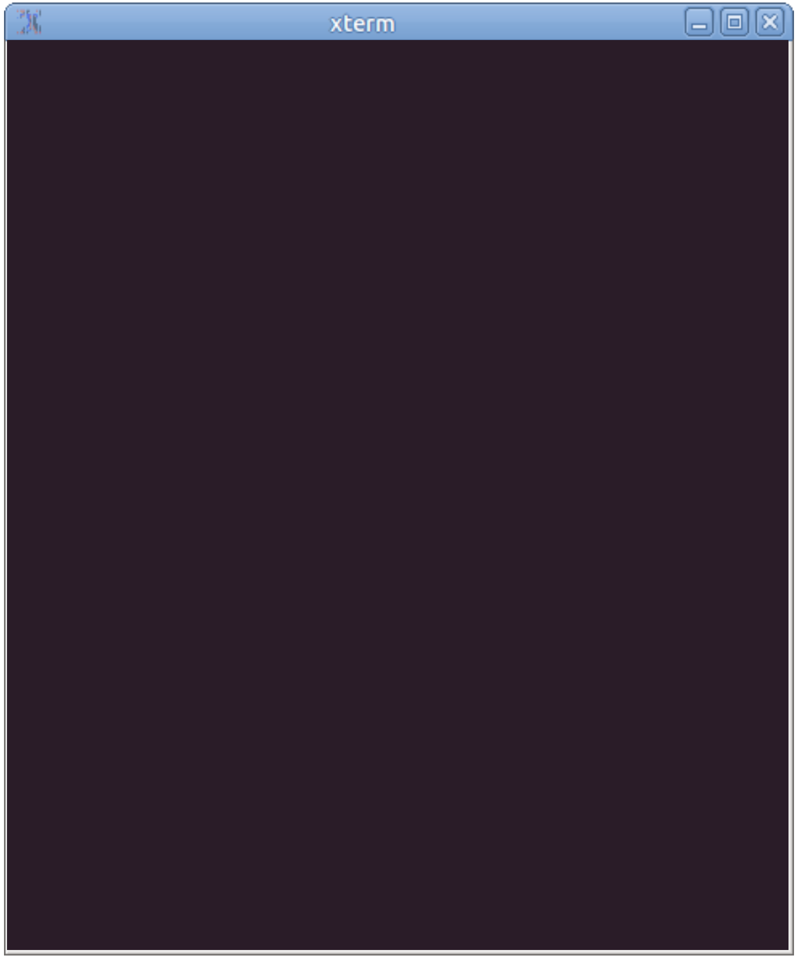
    - Possible solution: A=dianne, B=bo
    - Success; backtrack to knows(A, B) to find more
  - Program order determines the order in which facts and rules are considered
    - Consider: :- female(A), knows(A, B), male(B)

"Sean Spicer, our press secretary, gave alternative facts to that, but the point remains…"
Kellyanne Conway
22 January 2017

Key Prolog Concepts (cont.)

- **Backtracking**
  - If unification fails at any point, Prolog backs up and tries again with alternative facts and rules
  - *Example*: :- knows(A, B), female(A), male(B)
    - Need to satisfy knows(A, B)
    - Possible solution: A=bo, B=bo
    - Need to satisfy female(bo)
    - Fail; backtrack to knows(A, B)
    - Possible solution: A=bo, B=dianne
    - Need to satisfy female(bo)
    - Fail; backtrack to knows(A, B)

      …
    - Possible solution: A=dianne, B=bo
    - Success; backtrack to knows(A, B) to find more
  - Program order determines the order in which facts and rules are considered
    - Consider: :- female(A), knows(A, B), male(B)

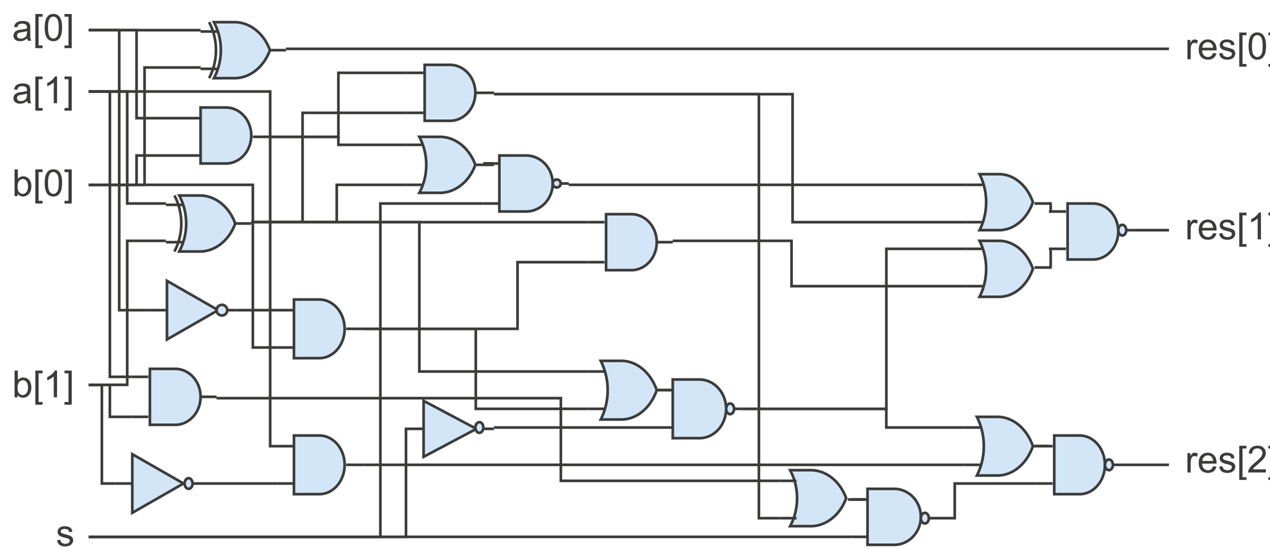


Digital Circuit Design

- **Today, virtually all hardware is created using a hardware description language (HDL)**
  - Lets one *think* gates but *write* textual code
  - Multi-bit variables, arithmetic/relational operators, conditionals, loops, modules, …
- **Hardware synthesis tool compiles code to logic gates (netlist format)**

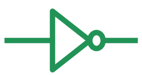
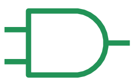
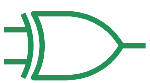
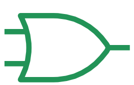
```
module my_prog (s,
a, b, res);
  input s;
  input [1:0] a, b;
  output [2:0] res;

  always @*
   if (s == 1)
     res = a + b;
   else
     res = a - b;
endmodule
```



Something to Consider

- **We can express a logic gate as a Hamiltonian function**
  - Minimized at any valid *relation* of inputs and outputs
  - Force σ_i to TRUE $(+1)$ with $\mathcal{H}_{\mathrm{VCC}}(\sigma_i) = -\sigma_i$ and to FALSE (-1) with $\mathcal{H}_{\mathrm{GND}}(\sigma_i) = \sigma_i$
  - Ergo, $\mathcal{H}_{\wedge}(A, B, Y) + \mathcal{H}_{\mathrm{VCC}}(A) + \mathcal{H}_{\mathrm{GND}}(B)$ anneals to $\{A = +1, B = -1, Y = -1\}$
  - Much cooler: $\mathcal{H}_{\wedge}(A, B, Y) + \mathcal{H}_{\mathrm{VCC}}(Y)$ anneals to $\{A = +1, B = +1, Y = +1\}$

Gate	2-local Ising-model Hamiltonian function
 NOT	$\mathcal{H}_{\neg}(\bar{\sigma}) = \sigma_A \sigma_Y$
 AND	$\mathcal{H}_{\wedge}(\bar{\sigma}) = -\frac{1}{2}\sigma_A - \frac{1}{2}\sigma_B + \sigma_Y + \frac{1}{2}\sigma_A\sigma_B - \sigma_A\sigma_Y - \sigma_B\sigma_Y$
 XOR	$\mathcal{H}_{\oplus}(\bar{\sigma}) = \frac{1}{2}\sigma_A - \frac{1}{2}\sigma_B - \frac{1}{2}\sigma_Y + \sigma_a - \frac{1}{2}\sigma_A\sigma_B - \frac{1}{2}\sigma_A\sigma_Y + \sigma_A\sigma_a + \frac{1}{2}\sigma_B\sigma_Y - \sigma_B\sigma_a - \sigma_Y\sigma_a$
 OR	$\mathcal{H}_{\vee}(\bar{\sigma}) = \frac{1}{2}\sigma_A + \frac{1}{2}\sigma_B - \sigma_Y + \frac{1}{2}\sigma_A\sigma_B - \sigma_A\sigma_Y - \sigma_B\sigma_Y$

Outline

- A bunch of concepts that seem totally unrelated to each other
- **Putting it all together**
- Conclusions

The Talk So Far

D-Wave
(superconducting qubits)

Me (giving this talk)

Logic
(Prolog
programming)

Digital
circuitry

Proposal

- **Run Prolog programs on a D-Wave system**
  - That is, compile Prolog programs to a 2-local Ising-model Hamiltonian function
  - The Hamiltonian's (possibly degenerate) ground state should correspond to all valid variable bindings
- **Insights**
  - Prolog unification can be replaced by equating variables (with a $J_{i,j} < 0$)
  - Prolog's backtracking strategy can be replaced by annealing to valid solutions
  - Prolog's ability to return multiple solutions can be handled by repeated anneals
- **Primary challenge**
  - Huge semantic gap between this:

```
likes(scott, dwave).
likes(sophia, X) :-
   likes(scott, X).
:- likes(sophia, What).
```

and this: $\mathcal{H}(\bar{\sigma}) = \sum_{i=0}^{N-1} h_i \sigma_i + \sum_{i=0}^{N-2} \sum_{j=i+1}^{N-1} J_{i,j} \sigma_i \sigma_j$

Approach

Prolog → Verilog → EDIF → QMASM → $\mathcal{H}$

Prolog	Verilog	EDIF	QMASM	$\mathcal{H}$
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 earlier in the talk**
  - Large enough to be interesting
  - Small enough to fit on a slide
  - (And generated intermediate files come close to fitting on a slide)

```
male(bo).
male(chad).
female(dianne).
female(talia).
works_at(bo, 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**
  - Code excerpt (missing only works_at and female):

```
// Define all of the symbols used
in this program.
`define bo      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;
  \knows/2 \knows_xvLbZ/2 (A,
B, $v1[0]);
  \female/1 \female_GBAIc/1 (A,
$v1[1]);
  \male/1 \male_mraJw/1 (B,
$v1[2]);
  assign Valid = &$v1;
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;
  \works_at/2
\works_at_WHthC/2 (A, C,
$v1[0]);
  \works_at/2
\works_at_TCUaX/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 == `bo;
  wire $v2;
  assign $v2 = A == `chad;
  assign Valid = &$v1 | &$v2;
endmodule
```

Step 2: EDIF

- **Forms a circuit from *cells* (gates) and *nets* (wires)**
  - Excerpt from the generated, 454-line, machine-parsable *s*-expression:

```
(cell (rename id00013 "female/1")
     (cellType GENERIC)
     (view VIEW_NETLIST
       (viewType NETLIST)
       (interface
         (port (array A 3) (direction INPUT))
         (port Valid (direction OUTPUT)))
       (contents
         (instance GND (viewRef VIEW_NETLIST
(cellRef GND (libraryRef LIB))))
         (instance VCC (viewRef VIEW_NETLIST
(cellRef VCC (libraryRef LIB))))
         (instance (rename id00014
"$abc$221$auto$blifparse.cc:286:parse_blif$222")
           (viewRef VIEW_NETLIST (cellRef id00001
(libraryRef LIB))))
         (instance (rename id00015
"$abc$221$auto$blifparse.cc:286:parse_blif$223")
           (viewRef VIEW_NETLIST (cellRef id00002
(libraryRef LIB))))
         (net (rename id00016 "$abc$221$n5_1")
(joined
           (portRef B (instanceRef id00015))
           (portRef Y (instanceRef id00014))))
         (net Valid (joined
           (portRef Valid)
           (portRef Y (instanceRef id00015))))
         (net (rename id00010 "A[0]") (joined
           (portRef (member A 0))
           (portRef A (instanceRef id00014))))
         (net (rename id00011 "A[1]") (joined
           (portRef (member A 1))
           (portRef A (instanceRef id00015))))
         (net (rename id00012 "A[2]") (joined
           (portRef (member A 2)))))))
```

Step 3: QMASM

- **Gates become macro instantiations; wires become QMASM "=" ($J_{i,j} < 0$)**

```
!include <stdcell>

# works_at/2
!begin_macro id00017
  !use_macro AOI3 $id00023
  !use_macro AOI3 $id00024
  !use_macro NAND $id00029
  !use_macro NOR $id00022
  !use_macro NOT $id00018
  !use_macro NOT $id00020
  !use_macro NOT $id00021
  !use_macro NOT $id00025
  !use_macro OAI4 $id00031
  !use_macro OR $id00019
  !use_macro OR $id00026
  !use_macro OR $id00027
  !use_macro OR $id00028
  !use_macro OR $id00030
  A[0] <-> $id00021.A
  A[1] <-> $id00020.A
  A[2] <-> $id00029.A
  B[0] <-> $id00025.A
  B[1] <-> $id00019.B
  $id00019.A = $id00018.Y
  $id00019.A = $id00022.A
  $id00019.A = $id00024.A
  $id00019.B = $id00022.B
  $id00019.B = $id00024.B
  $id00020.A = $id00029.B
  $id00021.A = $id00024.C
  $id00021.A = $id00028.B
  $id00022.A = $id00018.Y
  $id00022.A = $id00024.A
  $id00022.B = $id00024.B
  $id00023.A = $id00022.Y
  $id00023.B = $id00020.Y
  $id00023.C = $id00021.Y
  $id00024.A = $id00018.Y
  $id00025.A = $id00028.A
  $id00026.A = $id00025.Y
  $id00027.A = $id00026.Y
  $id00027.B = $id00024.Y
  $id00028.B = $id00024.C
  $id00029.A = $id00026.B
  $id00030.A = $id00029.Y
  $id00030.B = $id00028.Y
  $id00031.A = $id00030.Y
  $id00031.B = $id00019.Y
  $id00031.C = $id00027.Y
  $id00031.D = $id00023.Y
  A[0] = $id00024.C
  A[0] = $id00028.B
  A[1] = $id00029.B
  A[2] = $id00026.B
  B[0] = $id00028.A
  B[1] = $id00022.B
  B[1] = $id00024.B
  B[2] = $id00018.A
  Valid = $id00031.Y
!end_macro id00017
```

Step 4: The Final Hamiltonian

- **Targets a specific D-Wave device**
  - Uses the SAPI library's minor-embedder
- **Representative embedding statistics for this problem:**

Metric	Type	Count
Linear terms (h_i)	Logical	108
	Physical	282
Quadratic terms ($J_{i,j}$)	Logical	185
	Physical	365

It Really Works!

```
$ 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 = bo
```

Outline

- A bunch of concepts that seem totally unrelated to each other
- Putting it all together
- **Conclusions**

Conclusions

- **There exists a huge semantic gap between programming with logic (Horn clauses) and programming an Ising-model Hamiltonian function**
- **It turns out it is indeed possible to bridge this gap**
- **Insights**
  - Analogy between variable unification and impact of negative quadratic coefficients
  - Serial backtracking can be replaced by constraining all valid solutions to lie in a degenerate ground state
  - Transformation from one classical problem to another; no need to explicitly reason about quantum effects
- **Approach: successive lowering of the level of abstraction**
  - Logic program → hardware program → circuit specification → symbolic Hamiltonian → physical Hamiltonian
- **It is now possible to program a quantum annealer with an existing, classical, logic-programming language**

For More Information

- Pakin, Scott. "Performing Fully Parallel Constraint Logic Programming on a Quantum Annealer." *Theory and Practice of Logic Programming*, vol. 18, no. 5–6, 2018, pp. 928–949, September 2018. Eds.: Ferdinando Fioretto and Enrico Pontelli. Cambridge University Press. ISSN: 1475-3081, DOI: 10.1017/S1471068418000066.

- Try the code yourself:

https://github.com/lanl/QA-Prolog