Programming a Quantum Annealer

some slides originate from
D-Wave and Scott Pakin (LANL)

Outline

- Performance potential of quantum computing
- Quantum annealing
- Case study: D-Wave quantum annealers
- How to program a quantum annealer
- Example: Map coloring

The Quantum Optimization Problem

- We work with only this problem Hamiltonian of qubits σ_i^z :

$$\mathcal{H}_P = \sum_{i=0}^{N-2} \sum_{j=i+1}^{N-1} J_{i,j} \sigma_i^z \sigma_j^z + \sum_{i=0}^{N-1} h_i \sigma_i^z$$

- Objective (what the hardware does)
  - Minimize $\sigma_i^z \in \{0,1\}$ subject to provided $J_{i,j} \in \mathbb{R}$ and $h_i \in \mathbb{R}$
  - i.e., quantum optimization program is a list of $J_{i,j}$ and h_i
- Classical
  - Much easier to reason about than a quantum Hamiltonian
  - Quantum effects used internally to work towards objective
- 2-local
  - Can map >2-local problems to H_p → extra qubits
- Sparsely connected
  - map fully connected problems → D-Wave's Chimera graph, need extra qubits

Interpreting the Problem Hamiltonian

- Consider only external field with its "weights" h_i:

$$\mathcal{H}_P = \sum_{i=0}^{N-2} \sum_{j=i+1}^{N-1} J_{i,j} \sigma_i^z \sigma_j^z + \sum_{i=0}^{N-1} h_i \sigma_i^z$$

- Arbitrarily define "True" (say: +1), "False" (+1)
- Optimal σ_i^z for different h_i:

Negative
(say, h_i =-5)

σ_i^z	$h_i\sigma_i^z$
0	0
1	–5

Zero

σ_i^z	$h_i\sigma_i^z$
0	0
1	0

Positive
(say, h_i = +5)

σ_i^z	$h_i\sigma_i^z$
0	0
1	+5

- Observations
  - A negative h_i means, "I want σ_i^z to be True"
  - A zero h_i means, "I don't care if σ_i^z is True or False"
  - A positive h_i means, "I want σ_i^z to be False"

Interpreting the Problem Hamiltonian (2)

- Consider only "coupler" strengths $J_{i,j}$:

$$\mathcal{H}_P = \sum_{i=0}^{N-2} \sum_{j=i+1}^{N-1} J_{i,j}\sigma_i^z \sigma_j^z + \sum_{i=0}^{N-1} h_i \sigma_i^z$$

- Optimal σ_i^z and σ_j^z for different $J_{i,j}$:

Negative ($J_{i,j}$=-5)

σ_i^z	σ_j^z	$J_{i,j}\sigma_i^z \sigma_j^z$
0	0	0
0	1	0
1	0	0
1	1	–5

Zero

σ_i^z	σ_j^z	$J_{i,j}\sigma_i^z \sigma_j^z$
0	0	0
0	1	0
1	0	0
1	1	0

Positive ($J_{i,j}$=+5)

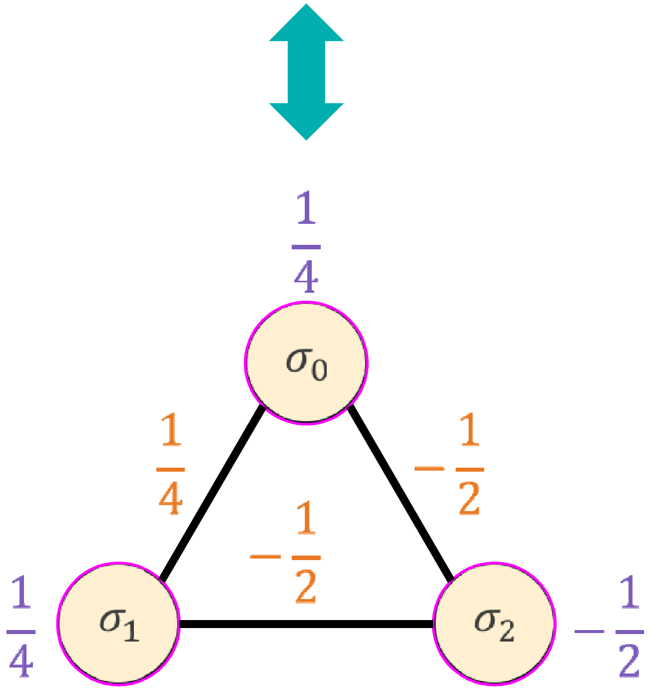
σ_i^z	σ_j^z	$J_{i,j}\sigma_i^z \sigma_j^z$
0	0	0
0	1	0
1	0	0
1	1	+5

- Observations
  - A negative $J_{i,j}$ means, "I want both σ_i^z and σ_j^z to be true"
  - A zero $J_{i,j}$ means, "I don't care how σ_i^z and σ_j^z are related"
  - A positive $J_{i,j}$ means, "I want neither σ_i^z nor σ_j^z to true"

Visualizing a Hamiltonian as a Graph

- Linear terms as vertex weights
- Quadratic terms as edge weights

$$\mathcal{H} = \frac{1}{4}\sigma_0 + \frac{1}{4}\sigma_1 - \frac{1}{2}\sigma_2 + \frac{1}{4}\sigma_0\sigma_1 - \frac{1}{2}\sigma_0\sigma_2 - \frac{1}{2}\sigma_1\sigma_2$$



Alternative Formulation—with Booleans

- **Different names for this appear in the optimization literature**
  - QUBO (quadratic unconstrained binary optimization problem)
  - UBQP (unconstrained binary quadratic optimization problem)
- **Goal**
  - Find $\arg\min_x f(x)$ with

$$f(x) = x^T Q x$$

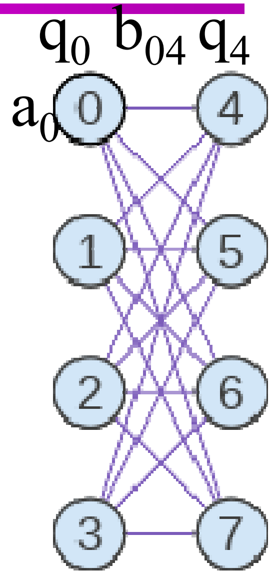
given Q either symmetric or upper-triangular, $Q_{i,j} \in \mathbb{R}$, and solving for $x_i \in \{0,1\}$

- **Can easily map between Ising-model Hamiltonians and QUBOs**
  - Diagonal elements of Q correspond to h_i; off-diagonal elements correspond to $J_{i,j}$
  - Based on a simple linear transformation: $x_i = (\sigma_i + 1)/2$
  - Hint: $x_i^2 = x_i$ when $x_i \in \{0,1\}$
  - Formula: $Q_{i,j} = 4J_{i,j}$ for $i < j$ and $Q_{i,i} = 2(h_i - \sum_{j=0}^{i-1} J_{j,i} - \sum_{j=i+1}^{N-1} J_{i,j})$
  - Example:

$$\mathcal{H} = \frac{1}{4}\sigma_0 + \frac{1}{4}\sigma_1 - \frac{1}{2}\sigma_2 + \frac{1}{4}\sigma_0\sigma_1 - \frac{1}{2}\sigma_0\sigma_2 - \frac{1}{2}\sigma_1\sigma_2 \xleftrightarrow{\mp\frac{3}{4}} f(x) = x^T \begin{pmatrix} 1 & 1 & -2 \\ 0 & 1 & -2 \\ 0 & 0 & 1 \end{pmatrix} x$$

(Use $(Q + Q^T)/2$ if you prefer a symmetric matrix.)

D-Wave's Programming Model (Qubo)



- OBJECTIVE: Obj Real-valued function

  $Obj(a_{i,}\ b_{ij}{:}q_i) = \Sigma_i a_i q_i + \Sigma_{ij} b_{ij} q_i q_j$

  — minimized during annealing cycle

  — System samples from q_i to minimize objective
  → drive into "ground state"
- QUBIT: Quantum bit; participates in annealing cycle

  — settles into one of two possible final states: 0,1
- COUPLER: q_j Physical device

  — allows one qubit to influence another qubit
- WEIGHT: Real-valued constant, 1 per qubit

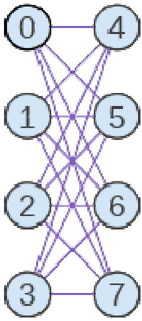
  — Influences qubit's tendency to collapse into its 2 possible final states; controlled by the programmer
- STRENGTH: $_i$ Real-valued constant, 1 per coupler

  — Controls influence exerted by one qubit on another; controlled by the programmer

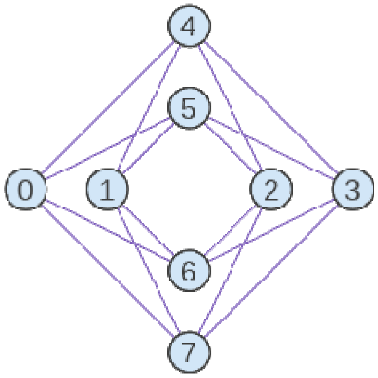
D-Wave 2X Today

— D-Wave2k machine	abstract 3D Chimera graph (L,M,N)
— 2k Qubits:	2LMN
— 6k Intra-cell couplers	$L^2MN+L(M-1)N+LM(N-1)$
— 8k Inter-cell couplers	$2LMN+L^2MN+L(M-1)N+LM(N-1)$

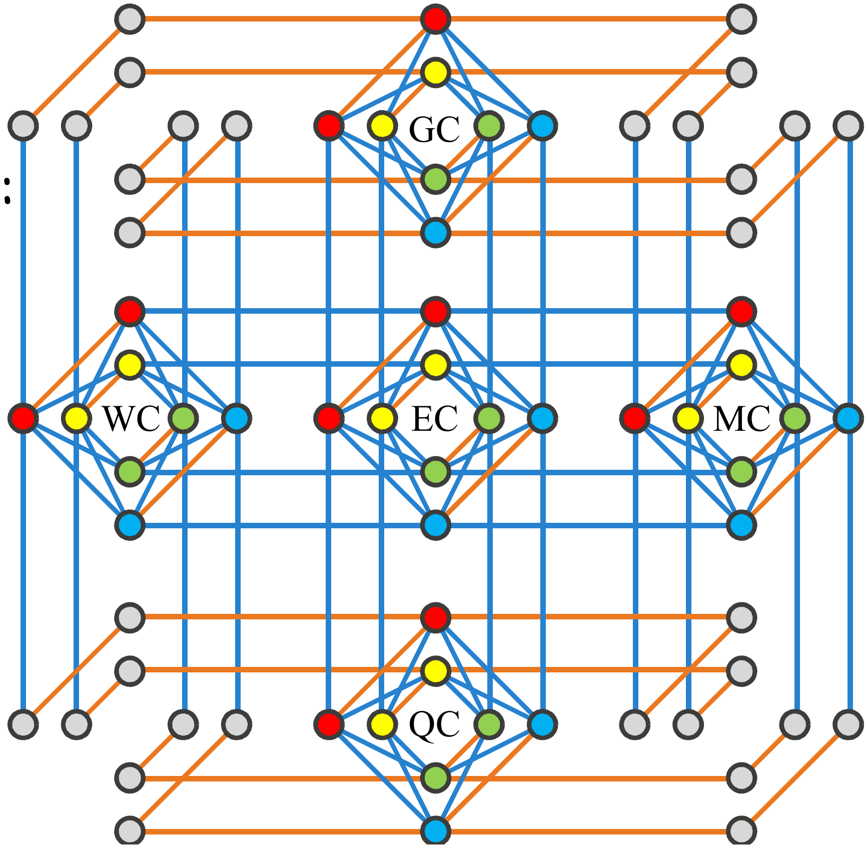
— Intra:



or

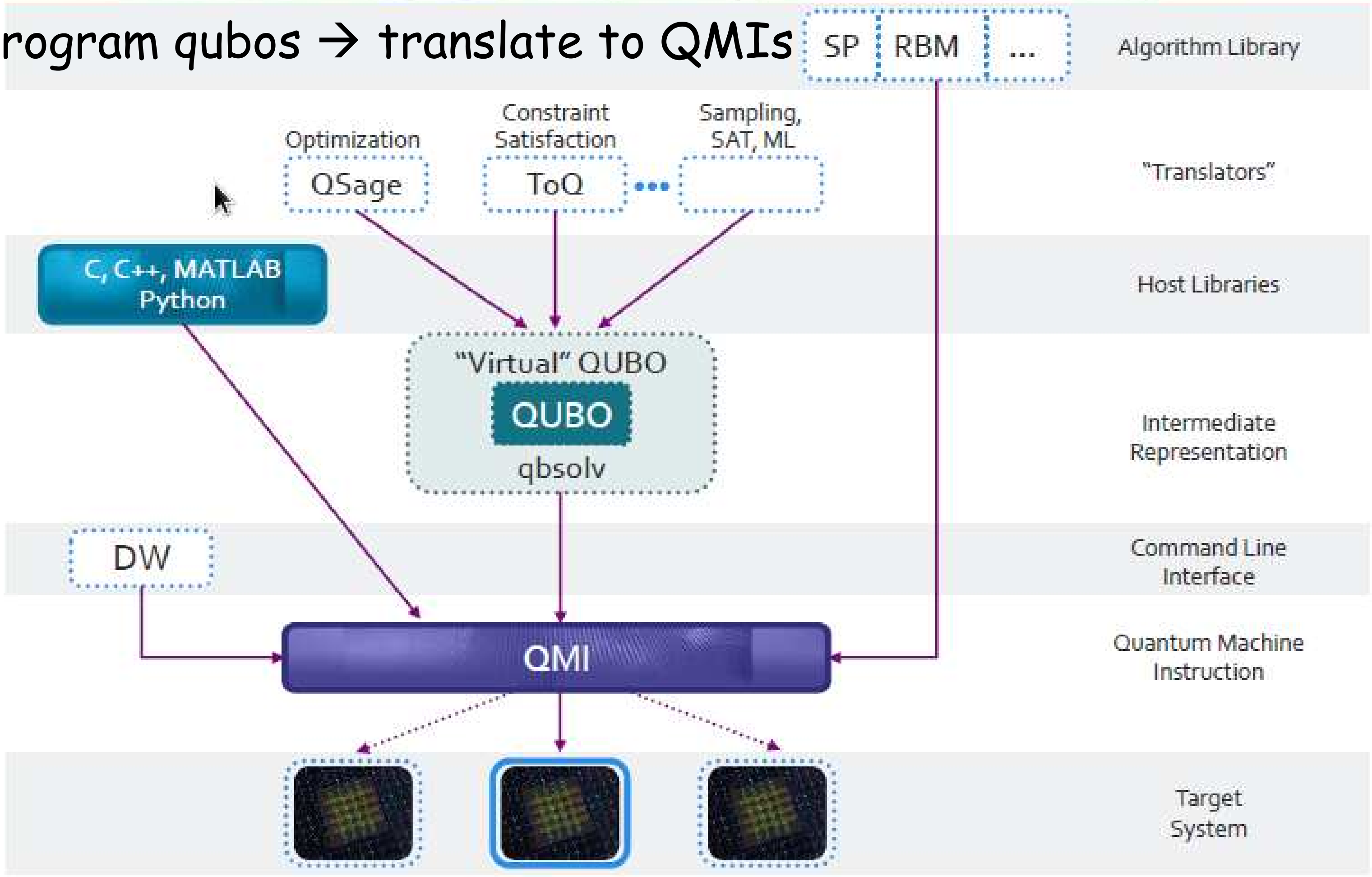


Inter:



Software Environment

- Accelerator (like GPU), runs quantum machine instructions (QMIs)
- program qubos → translate to QMIs

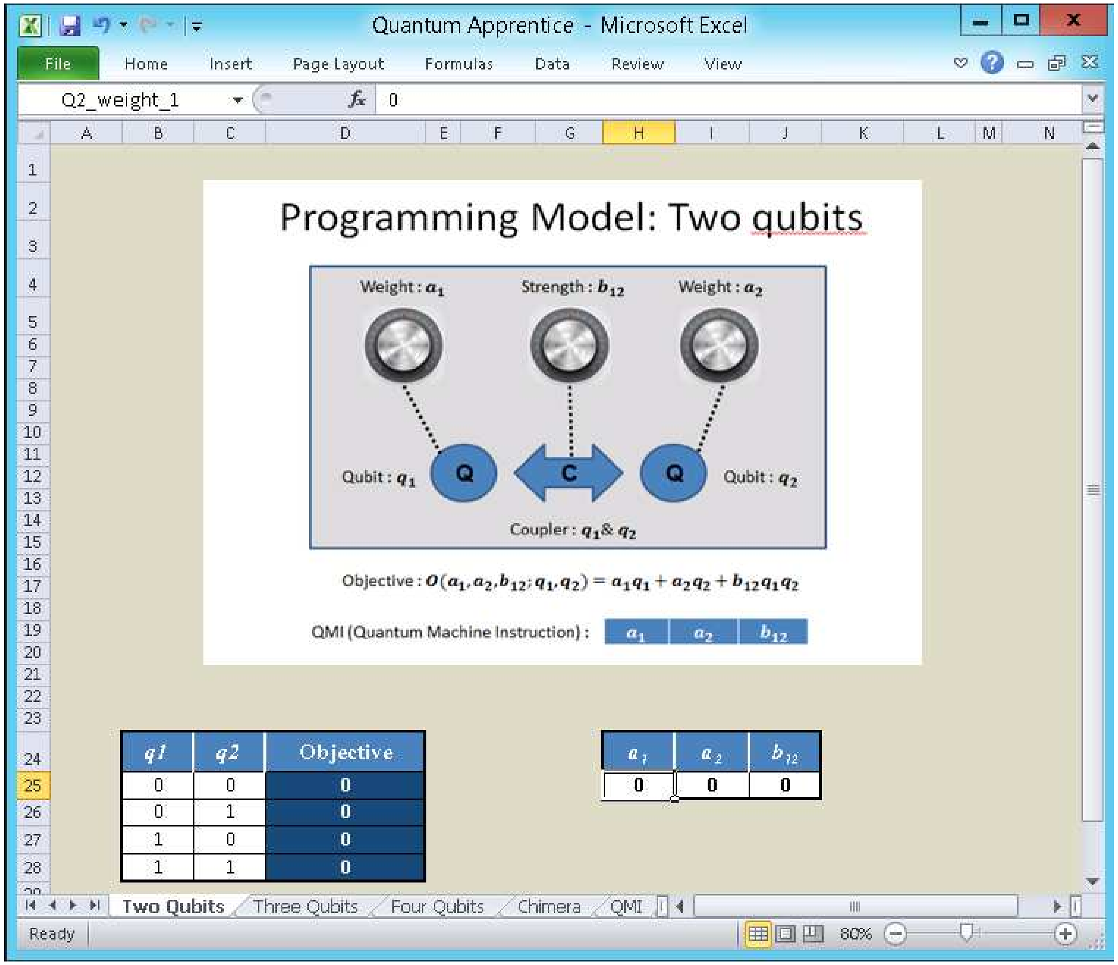


Quantum Apprentice: 2,3,4 qubit simulation

- See web page: http://moss.csc.ncsu.edu/~mueller/qc/qc-tut
- your 1st D-Wave pgm

Later:

- Connect to machine
- Run there
- Receive >1 answer
- Interpret probability
- BUT: h/w access only for IBM, not D-Wave



Objective : $O(a_1, a_2, b_{12}; q_1, q_2) = a_1 q_1 + a_2 q_2 + b_{12} q_1 q_2$

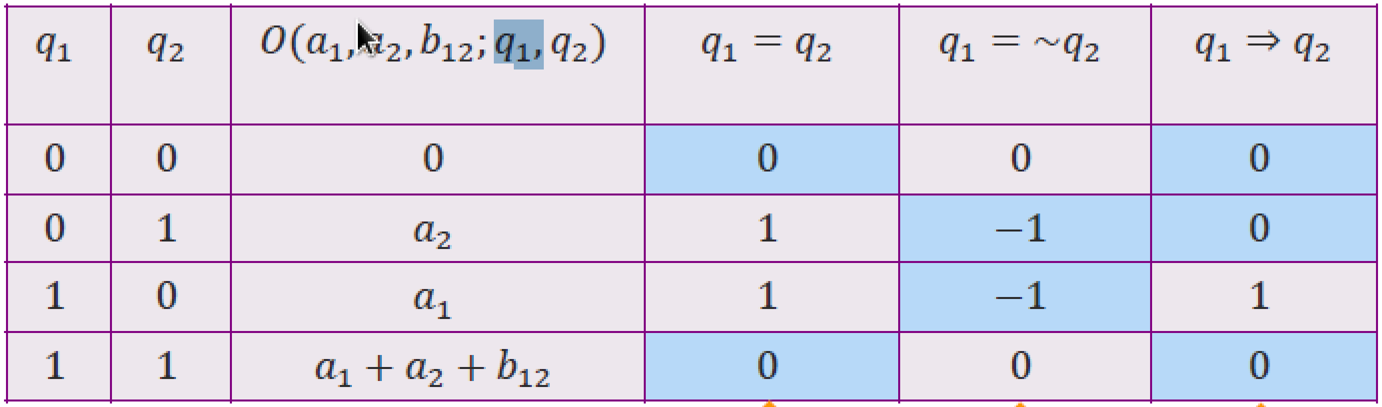
QMI (Quantum Machine Instruction) : a_1 a_2 b_{12}

$q1$	$q2$	Objective
0	0	0
0	1	0
1	0	0
1	1	0

a_1	a_2	b_{12}
0	0	0

2-qubit objectives

- Blue values → minimum = objective



			equal	not equal	implies
q_1	q_2	$O(a_1, a_2, b_{12}; q_1, q_2)$	$q_1 = q_2$	$q_1 = \sim q_2$	$q_1 \Rightarrow q_2$
0	0	0	0	0	0
0	1	a_2	1	-1	0
1	0	a_1	1	-1	1
1	1	$a_1 + a_2 + b_{12}$	0	0	0

$a_1 = 1$
$a_2 = 1$
$b_{12} = -2$

$a_1 = -1$
$a_2 = -1$
$b_{12} = 2$

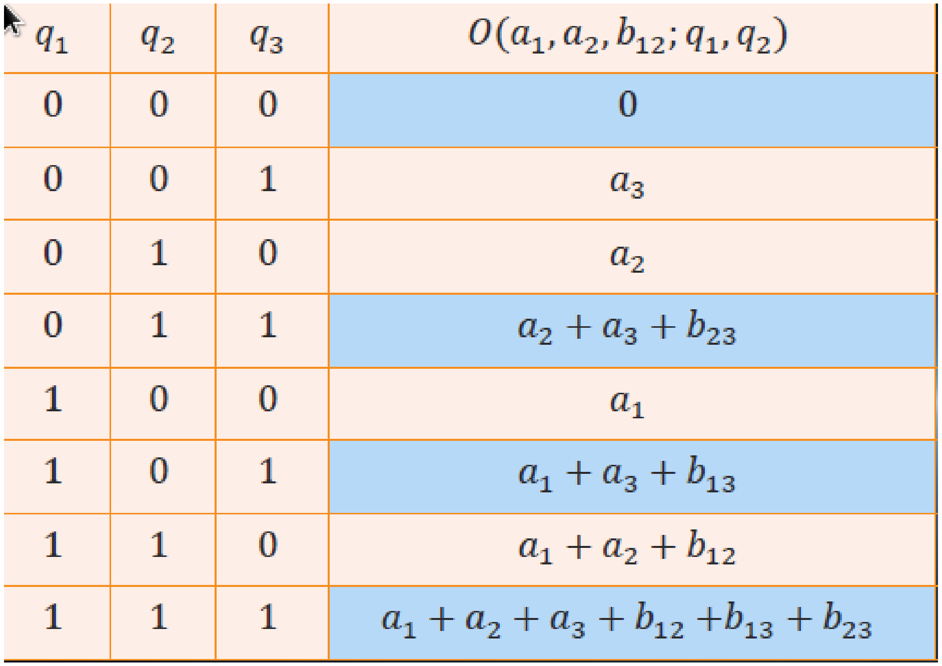
$a_1 = 1$
$a_2 = 0$
$b_{12} = -1$

- $Obj(a_i, b_{ij}{:}q_i) = \Sigma_i a_i q_i + \Sigma_{ij} b_{ij} q_i q_j$

For each of the three problems, transfer the a_1, a_2 and b_{12} values to the Two Qubits tab in Quantum Apprentice and verify that the valid states are correct.

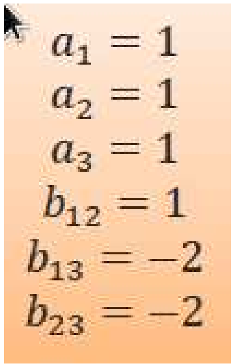
3-qubit objective: OR

- $q_1 \vee q_2 = q_3$

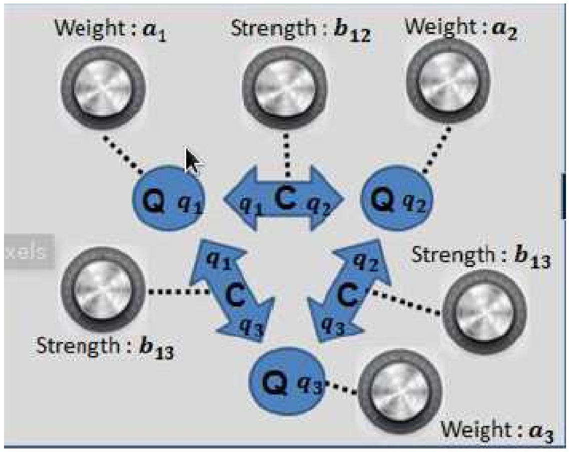


q_1	q_2	q_3	$O(a_1, a_2, b_{12}; q_1, q_2)$
0	0	0	0
0	0	1	a_3
0	1	0	a_2
0	1	1	$a_2 + a_3 + b_{23}$
1	0	0	a_1
1	0	1	$a_1 + a_3 + b_{13}$
1	1	0	$a_1 + a_2 + b_{12}$
1	1	1	$a_1 + a_2 + a_3 + b_{12} + b_{13} + b_{23}$

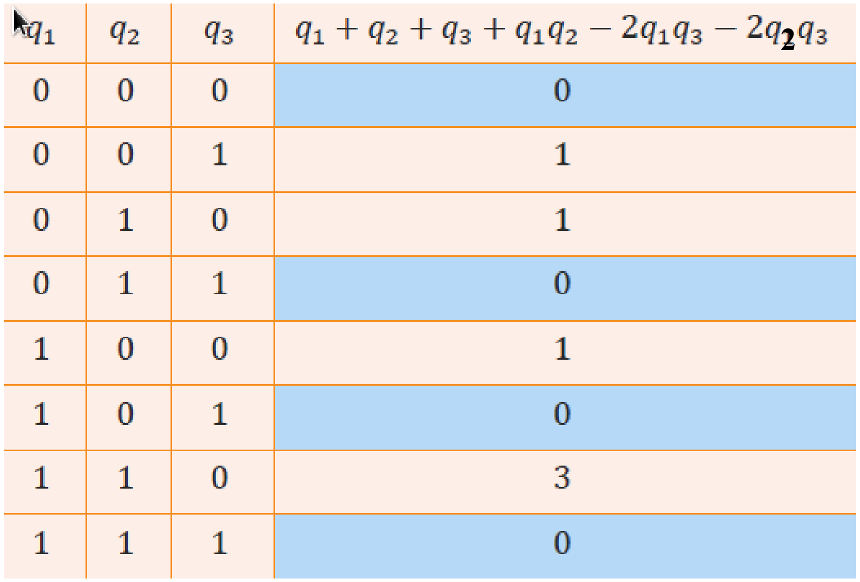
- $Obj(a_i, b_{ij}; q_i) = \Sigma_i a_i q_i + \Sigma_{ij} b_{ij} q_i q_j$



$a_1 = 1$
$a_2 = 1$
$a_3 = 1$
$b_{12} = 1$
$b_{13} = -2$
$b_{23} = -2$



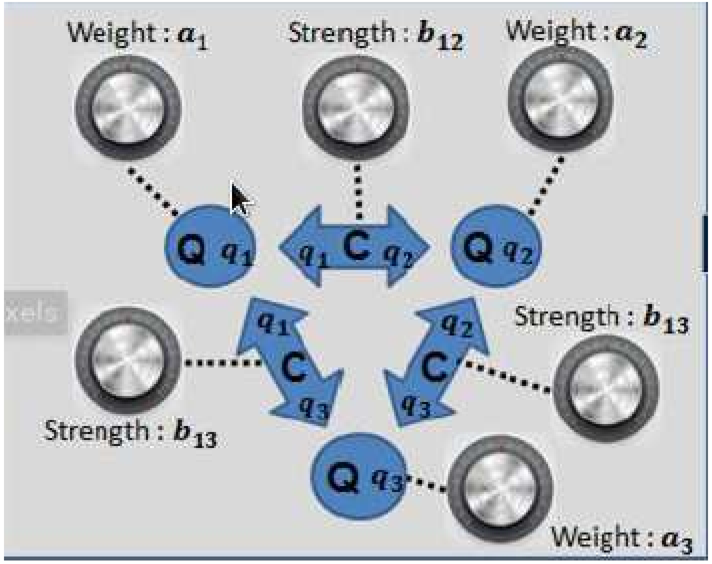
- Values:



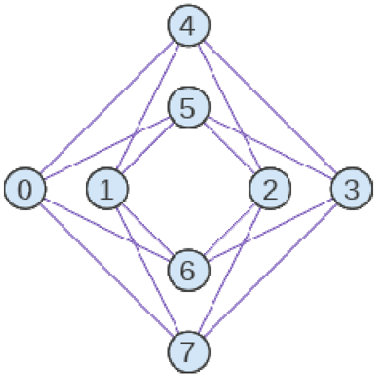
q_1	q_2	q_3	$q_1 + q_2 + q_3 + q_1q_2 - 2q_1q_3 - 2q_2q_3$
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	3
1	1	1	0

3-qubit mapping

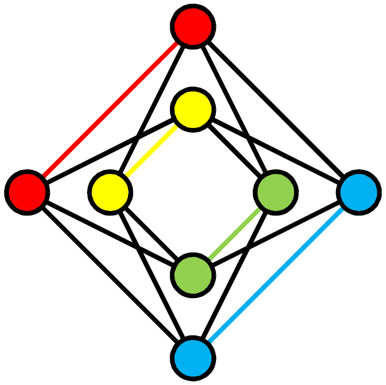
- Problem: cannot map theoretical 3-qubo into bipartite graph



→

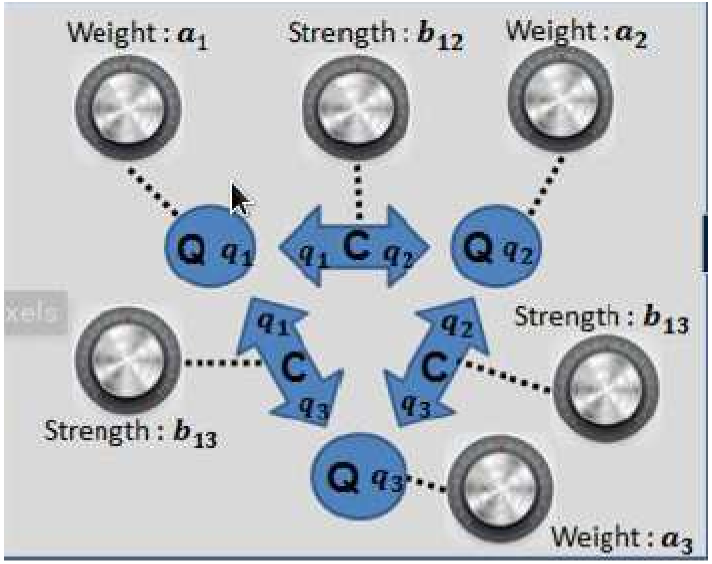


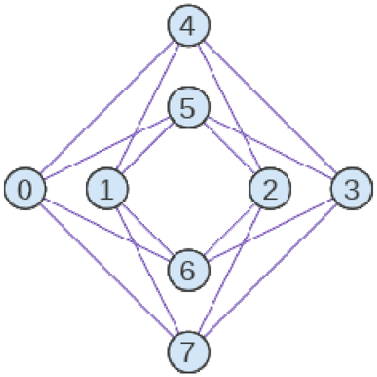
- Use 2 qubit chains
  - Red, yellow, green, blue chains (many adjacent placement options)
  - Let's start w/ 1 chain (red), only



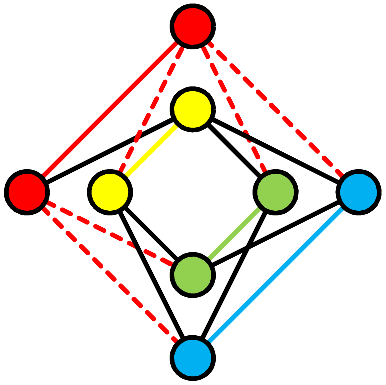
3-qubit mapping

- Problem: cannot map theoretical 3-qubo into bipartite graph



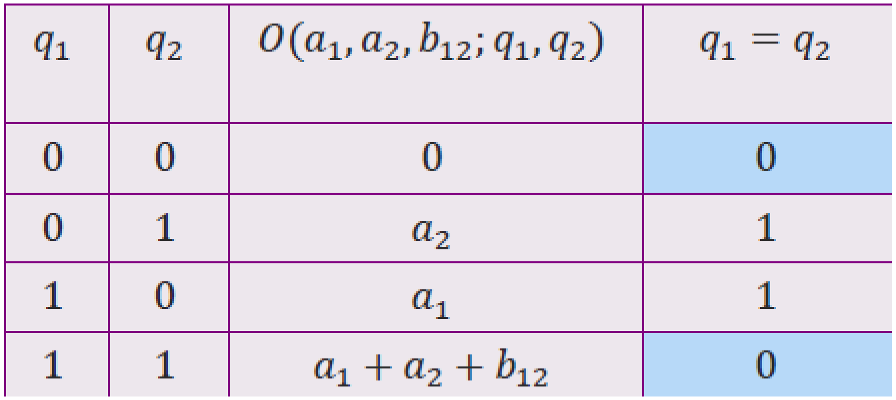


- Use 2 qubit chains
  - Red, yellow, green, blue chains (many adjacent placement options)
  - Let's start w/ 1 chain (red), only
    - 4→6 neighbors

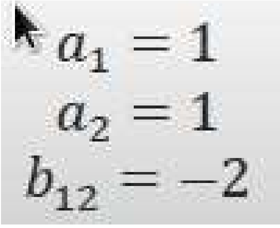


Building Qubit chains

- Use equal qubo b/w 2 or more qubits

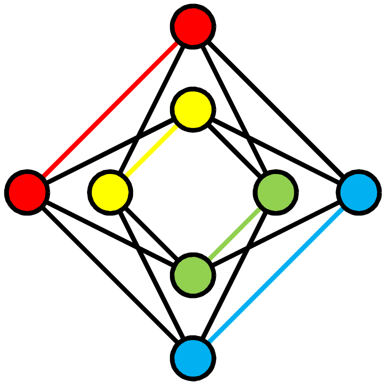


q_1	q_2	$O(a_1, a_2, b_{12}; q_1, q_2)$	$q_1 = q_2$
0	0	0	0
0	1	a_2	1
1	0	a_1	1
1	1	$a_1 + a_2 + b_{12}$	0



$$a_1 = 1$$
$$a_2 = 1$$
$$b_{12} = -2$$

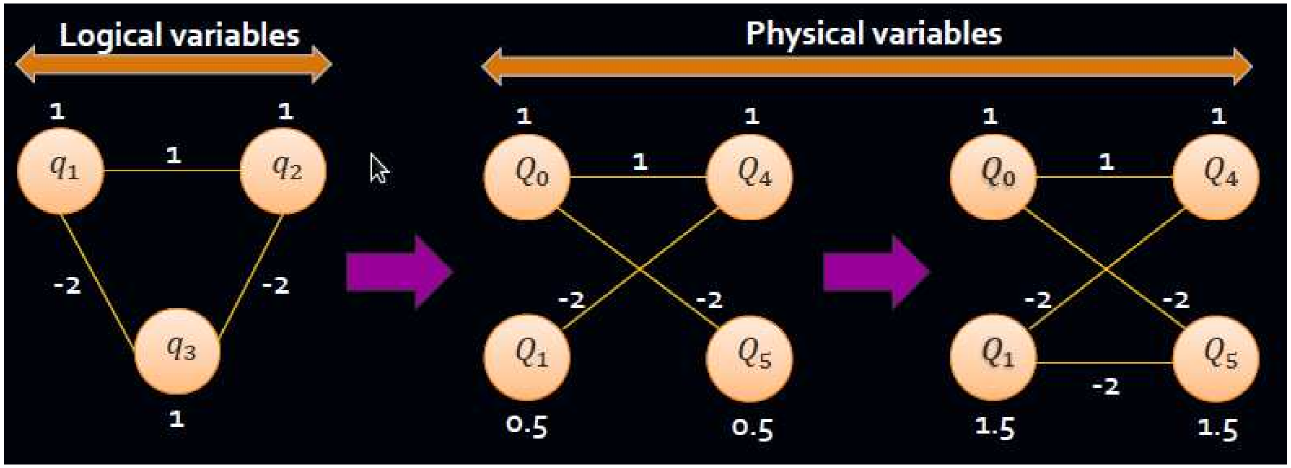
- causes q_1 and $_2$ to be the same
- Idea: if qubits differ,
  we'll pay an objective penalty



Embedding rules

1. A logical qubit can be mapped to N physical qubits as long as physical qubits form a connected set. We call this a chain.
2. For each physical coupler connecting 2 physical qubits in same chain, include QUBO terms to cause the 2 physical qubits to align
3. When a logical qubit is mapped to an N-qubit chain, divide weight for logical qubit by N & apply that weight to each qubit in chain.
4. Split the coupling strength between two logical qubits over all available physical couplers connecting the chains corresponding to the logical qubits.

- $q_1 \rightarrow Q_0$
- $q_2 \rightarrow Q_4$
- $q_3 \rightarrow [Q_1\ Q_5]$



- Why does it work? Logical ground state = physical ground state

Probability Amplitudes

- Simulator: k answers (if your problem has that many answers)
- Quantum computer: n > k answers, each w/ probability
  - Here: 4 answer w/ nearly equally high probaility (rest is low)

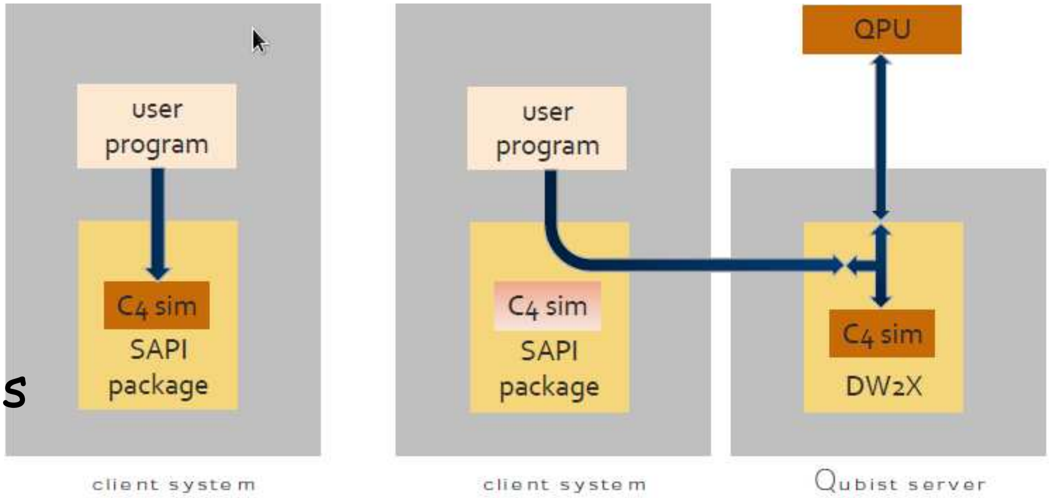
Solution	Energy	Occurrences
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ...] Graph Show	-0.62	260
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ...] Graph Show	-0.62	327
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ...] Graph Show	-0.62	134
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ...] Graph Show	-0.62	277
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ...] Graph Show	-0.37	2

Showing 1 to 5 of 5 entries

Direct Programming of D-Wave

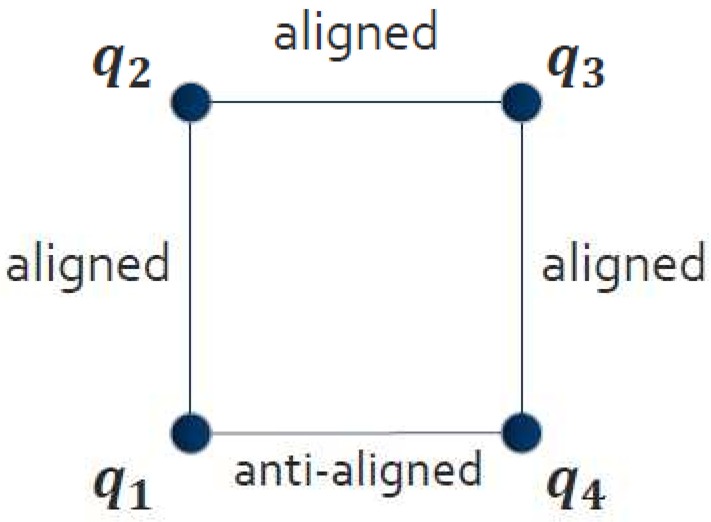
- Solver API (SAPI): for C, Matlab, python (Windows, Linux, iOS)
- Lowest level of programming
- Synchronous/async. QMI instructions

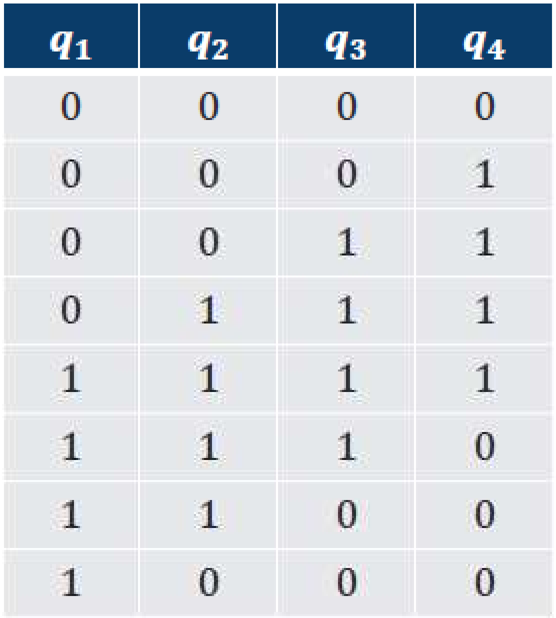
- Via simulation or connect to real machine (if you have access)
- Choose different solvers
- Advanced features
  - Async. Exec
  - Embedding
  - Order reduction
  - Spin reversal transforms
  - Post processing
  - Ising/QUBO translation



SAPI Example: frustrated system

- We know how to make aligned and anti-aligned chains.
- Combine these two chain types to build a frustrated system.





q_1	q_2	q_3	q_4
0	0	0	0
0	0	0	1
0	0	1	1
0	1	1	1
1	1	1	1
1	1	1	0
1	1	0	0
1	0	0	0

QUBOs for individual constraints

aligned

q_1	q_2	$q_1 + q_2 - 2q_1q_2$
0	0	0
0	1	1
1	0	1
1	1	0

aligned

q_3	q_4	$q_3 + q_4 - 2q_3q_4$
0	0	0
0	1	1
1	0	1
1	1	0

aligned

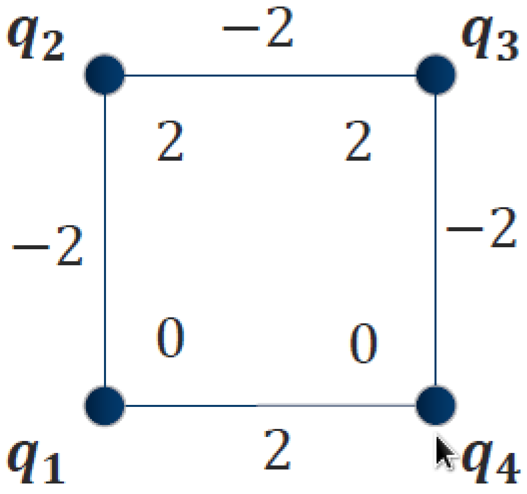
q_2	q_3	$q_2 + q_3 - 2q_2q_3$
0	0	0
0	1	1
1	0	1
1	1	0

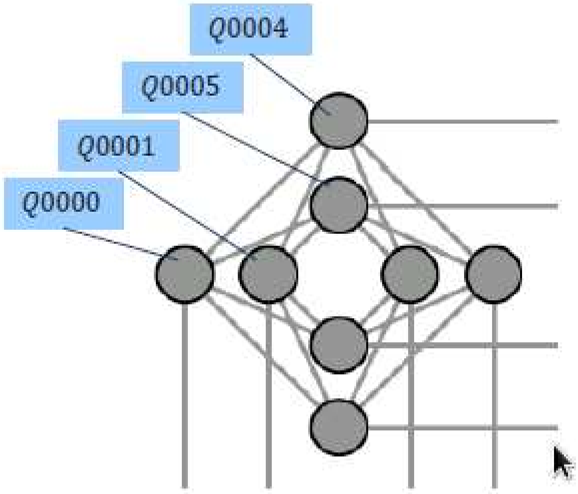
anti-aligned

q_4	q_1	$-q_4 - q_1 + 2q_4q_1$
0	0	0
0	1	-1
1	0	-1
1	1	0

Aggregate QUBO

- Confirm that QUBO represented here is sum of individual QUBOs from last slide.
- Input the QUBO below into Quantum Apprentice on Four Qubits tab.
- Confirm that you get desired set of states.
  - $Obj = 2q_2 + 2q_3 - 2q_1q_2 - 2q_2q_3 - 2q_3q_4 + 2q_4q_1$
- Embed QUBO to unit cell
  - $q_1 \Rightarrow Q0000$
  - $q_2 \Rightarrow Q0004$
  - $q_3 \Rightarrow Q0001$
  - $q_4 \Rightarrow Q0005$
- Compile, run, check output

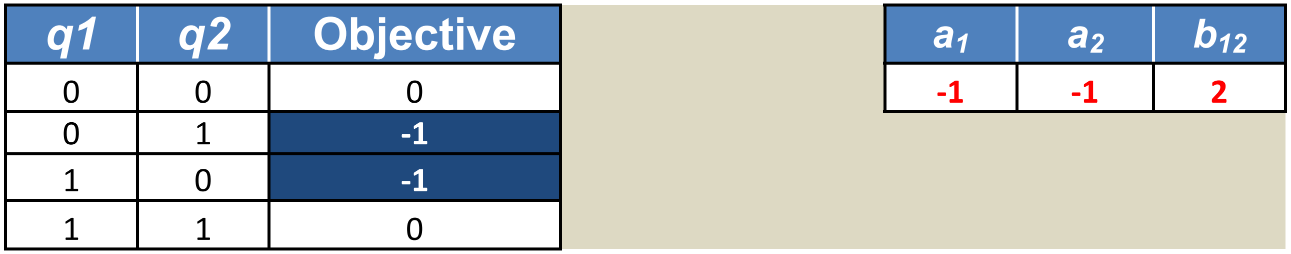




“Not” Derived from QUBO

- Consider ground states
- Find common equation (binary): $z=\neg x$
- Find penalty function: $2xz-x-z+1=0$
  - Already minimized
- Make it a QUBO (add -1): $2xz-x-z$
- Check if minimization is correct:

x	z	Obj
0	0	? → penalize s.t. >T: -1
0	1	T
1	0	T
1	1	? → penalize s.t. >T: xz



$q1$	$q2$	Objective
0	0	0
0	1	-1
1	0	-1
1	1	0

a_1	a_2	b_{12}
-1	-1	2

Simulated “Not”

- Objective function: 2xz-x-z

```
from dimod import ExactSolver
sampler = ExactSolver()
sampler_embedded = EmbeddingComposite(sampler)

Q = {('x', 'x'): -1, ('y', 'y'): -1, ('x', 'y'): 2}

response = sampler_embedded.sample_qubo(Q)
for datum in response.data(['sample', 'energy’]):
  print(datum.sample, "Energy: ", datum.energy)
```

- Output:

```
({'y': 0, 'x': 1}, 'Energy: ', -1.0)
({'y': 1, 'x': 0}, 'Energy: ', -1.0)
({'y': 0, 'x': 0}, 'Energy: ', 0.0)
({'y': 1, 'x': 1}, 'Energy: ', 0.0)
```

Hardware “Not”

- Objective function: $2xz-x-z$

```
from dwave.system.samplers import DWaveSampler
from dwave.system.composites import EmbeddingComposite
sampler = DWaveSampler(endpoint='https://cloud.dwavesys.com/sapi',
    token='DEV-YOUR-TOKEN', solver='DW_2000Q_2_1')
sampler_embedded = EmbeddingComposite(sampler)

Q = {('x', 'x'): -1, ('y', 'y'): -1, ('x', 'y'): 2}

response = sampler_embedded.sample_qubo(Q, num_reads=5000)
for datum in response.data(['sample', 'energy', 'num_occurrences']):
  print(datum.sample, "Energy: ", datum.energy, "Occurrences: ",
    datum.num_occurrences)
```

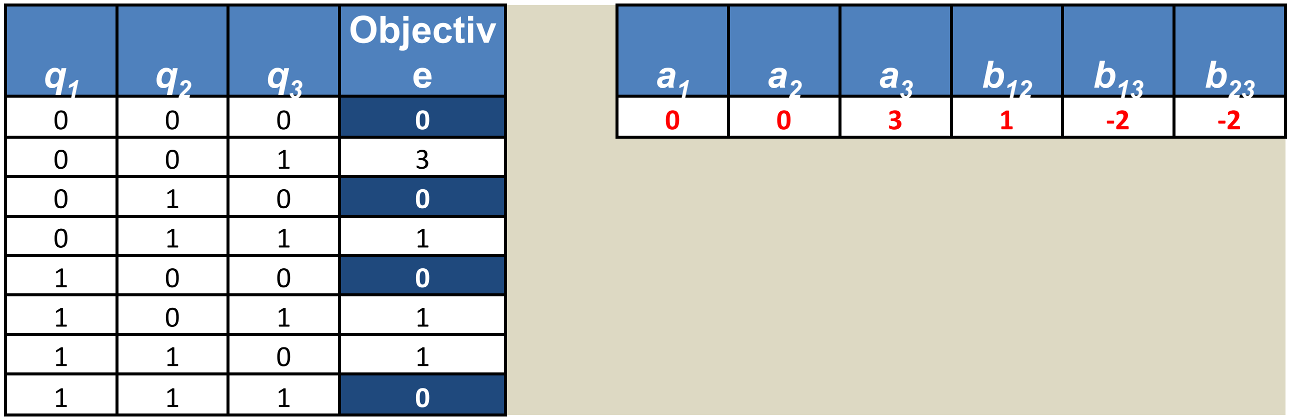
- Output:

```
({'y': 1, 'x': 0}, 'Energy: ', -1.0, 'Occurrences: ', 2617)
({'y': 0, 'x': 1}, 'Energy: ', -1.0, 'Occurrences: ', 2382)
({'y': 0, 'x': 0}, 'Energy: ', 0.0, 'Occurrences: ', 1)
```

"And" Derived from QUBO

- Consider ground states
- Find common equation (binary): $z=x \wedge y=xy$
- Find penalty function: $xy-2(x+y)z-3z=0$
    - Already minimized
- Rewrite as QUBO: $3z+xy-2xz-2yz$
- Check if minimization is correct:

x	y	z	Obj	
0	0	0	T	
0	0	1	?	>T: z
0	1	0	T	
0	1	1	?	>T: yz
1	0	0	T	
1	0	1	?	>T: xz
1	1	0	?	>T: xy
1	1	1	T	



q_1	q_2	q_3	Objective
0	0	0	0
0	0	1	3
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

a_1	a_2	a_3	b_{12}	b_{13}	b_{23}
0	0	3	1	-2	-2

Simulated “And”

- Objective function: $3z+xy-2xz-2yz$

```
from dimod import ExactSolver
sampler = ExactSolver()
sampler_embedded = EmbeddingComposite(sampler)

Q = {('x', 'y'): 1, ('x', 'z'): -2, ('y', 'z'): -2, ('z', 'z'): 3}

response = sampler_embedded.sample_qubo(Q)
for datum in response.data(['sample', 'energy']):
  print(datum.sample, "Energy: ", datum.energy)
```

- Output:

```
({'y': 0, 'x': 0, 'z': 0}, 'Energy: ', 0.0)
({'y': 0, 'x': 1, 'z': 0}, 'Energy: ', 0.0)
({'y': 1, 'x': 0, 'z': 0}, 'Energy: ', 0.0)
({'y': 1, 'x': 1, 'z': 1}, 'Energy: ', 0.0)
({'y': 1, 'x': 1, 'z': 0}, 'Energy: ', 1.0)
({'y': 1, 'x': 0, 'z': 1}, 'Energy: ', 1.0)
({'y': 0, 'x': 1, 'z': 1}, 'Energy: ', 1.0)
({'y': 0, 'x': 0, 'z': 1}, 'Energy: ', 3.0)
```

Hardware “And”

- Objective function: $3z+xy-2xz-2yz$

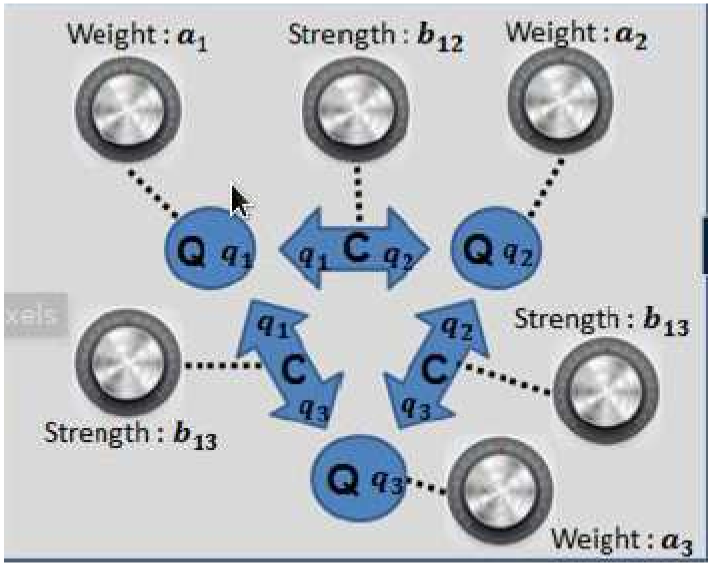
```
from dwave.system.samplers import DWaveSampler
from dwave.system.composites import EmbeddingComposite
sampler = DWaveSampler(endpoint='https://cloud.dwavesys.com/sapi',
    token='DEV-YOUR-TOKEN', solver='DW_2000Q_2_1')
sampler_embedded = EmbeddingComposite(sampler)
Q = {('x', 'y'): 1, ('x', 'z'): -2, ('y', 'z'): -2, ('z', 'z'): 3}
response = sampler_embedded.sample_qubo(Q, num_reads=5000)
for datum in response.data(['sample', 'energy', 'num_occurrences']):
  print(datum.sample, "Energy: ", datum.energy, "Occurrences: ",
    datum.num_occurrences)
```

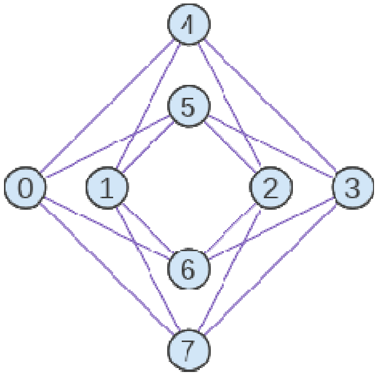
- Output:

```
({'y': 0, 'x': 1, 'z': 0}, 'Energy: ', 0.0, 'Occurrences: ', 1843)
({'y': 1, 'x': 0, 'z': 0}, 'Energy: ', 0.0, 'Occurrences: ', 708)
({'y': 1, 'x': 1, 'z': 1}, 'Energy: ', 0.0, 'Occurrences: ', 766)
({'y': 0, 'x': 0, 'z': 0}, 'Energy: ', 0.0, 'Occurrences: ', 1680)
({'y': 0, 'x': 1, 'z': 0}, 'Energy: ', 0.0, 'Occurrences: ', 2)
({'y': 1, 'x': 1, 'z': 0}, 'Energy: ', 1.0, 'Occurrences: ', 1)
```

“And” Embedding

- D-Wave “unit” graph is a mismatch
- Need ancilla bit in $K_{4,4}$ bi-partite graph
- Ancilla z’=z
- Map
    - x→1
    - y→5
    - z→0 and 4 (actually z and z’)





Hardware "And" Explicitly Embedded

- Duplicated "z" with implicit highest chain strength (weight -2)

```
from dwave.system.samplers import DWaveSampler
from dwave.system.composites import FixedEmbeddingComposite
embedding = {'x': {1}, 'y': {5}, 'z': {0, 4}}
sampler = DWaveSampler(endpoint='https://cloud.dwavesys.com/sapi',
    token='DEV-YOUR-TOKEN', solver='DW_2000Q_2_1')
sampler_embedded = FixedEmbeddingComposite(sampler, embedding)
Q = {('x', 'y'): 1, ('x', 'z'): -2, ('y', 'z'): -2, ('z', 'z'): 3}
response = sampler_embedded.sample_qubo(Q, num_reads=5000)
for datum in response.data(['sample', 'energy', 'num_occurrences']):
  print(datum.sample, "Energy: ", datum.energy, "Occurrences: ",
    datum.num_occurrences)
```

- Output:

```
({'y': 0, 'x': 1, 'z': 0}, 'Energy: ', 0.0, 'Occurrences: ', 1677)
({'y': 1, 'x': 0, 'z': 0}, 'Energy: ', 0.0, 'Occurrences: ', 821)
({'y': 1, 'x': 1, 'z': 1}, 'Energy: ', 0.0, 'Occurrences: ', 1398)
({'y': 0, 'x': 0, 'z': 0}, 'Energy: ', 0.0, 'Occurrences: ', 1101)
({'y': 0, 'x': 1, 'z': 0}, 'Energy: ', 0.0, 'Occurrences: ', 1)
({'y': 1, 'x': 0, 'z': 1}, 'Energy: ', 1.0, 'Occurrences: ', 2)
```

Hardware “And” Explicitly Embedded

- Careful: Lower chain strength → bogus results!!!

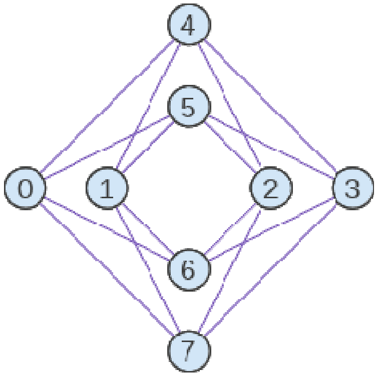
```
from dwave.system.samplers import DWaveSampler
from dwave.system.composites import FixedEmbeddingComposite
embedding = {'x': {1}, 'y': {5}, 'z': {0, 4}}
sampler = DWaveSampler(endpoint='https://cloud.dwavesys.com/sapi',
    token='DEV-YOUR-TOKEN', solver='DW_2000Q_2_1')
sampler_embedded = FixedEmbeddingComposite(sampler, embedding)
Q = {('x', 'y'): 1, ('x', 'z'): -2, ('y', 'z'): -2, ('z', 'z'): 3}
response = sampler_embedded.sample_qubo(Q, num_reads=5000,
    chain_strength=0.1)
for datum in response.data(['sample', 'energy', 'num_occurrences']):
  print(datum.sample, "Energy: ", datum.energy, "Occurrences: ",
    datum.num_occurrences)
```

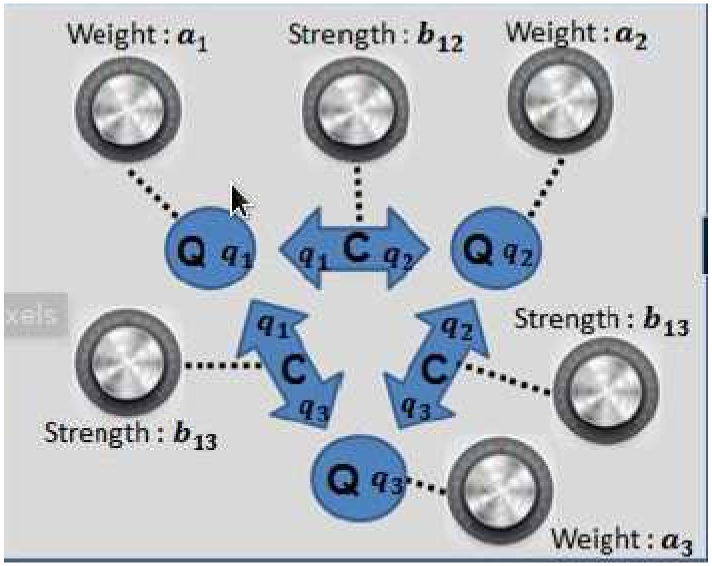
Output:

```
    ({'y': 0, 'x': 1, 'z': 0}, 'Energy: ', 0.0, 'Occurrences: ', 2424)
    ({'y': 1, 'x': 0, 'z': 0}, 'Energy: ', 0.0, 'Occurrences: ', 2509)
    ({'y': 0, 'x': 1, 'z': 0}, 'Energy: ', 0.0, 'Occurrences: ', 14)
    ({'y': 1, 'x': 0, 'z': 0}, 'Energy: ', 0.0, 'Occurrences: ', 12)
    ({'y': 1, 'x': 1, 'z': 1}, 'Energy: ', 0.0, 'Occurrences: ', 30)
    ({'y': 0, 'x': 0, 'z': 0}, 'Energy: ', 0.0, 'Occurrences: ', 11)
```

“And” with Manual Embedding (Option)

- D-Wave “unit” graph is a mismatch
  - 3z+xy-2xz-2yz gets transformed
- Need ancilla bit in $K_{4,4}$ bi-partite graph
- Map w/ ancilla z’
  - x→1
  - y→5
  - z→0 and z’→4

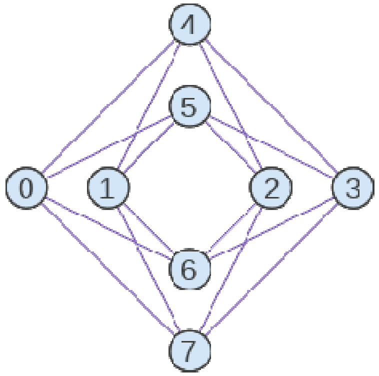




- Ancilla z’=z, add equality as before: -2zz’
  - Split bias/weight of z: 1.5z+1.5z’
  - Add $\frac{1}{2}$ of equality weight: 2.5z+2.5z’
  - -2xz becomes -2xz’
- We get: 2.5z+2.5z’+xy-2yz-2xz’-2zz’

“And” with Manual Embedding

- We get: $2.5z+2.5z'+xy-2yz-2xz'-2zz'$
- Check



a_1	a_2	a_3	a_4	b_{12}	b_{13}	b_{14}	b_{23}	b_{24}	b_{34}
0	0	2.5	2.5	1	0	-2	-2	0	-2

q_1	q_2	q_3	q_4	Objective
0	0	0	0	0
0	0	0	1	2.5
0	0	1	0	2.5
0	0	1	1	3
0	1	0	0	0
0	1	0	1	2.5
0	1	1	0	0.5
0	1	1	1	1
1	0	0	0	0
1	0	0	1	0.5
1	0	1	0	2.5
1	0	1	1	1
1	1	0	0	1
1	1	0	1	1.5
1	1	1	0	1.5
1	1	1	1	0

Hardware “And” Manually Embedded

- Objective function: $2.5z+2.5z'+xy-xz-yz-xz'-yz'-2zz'$

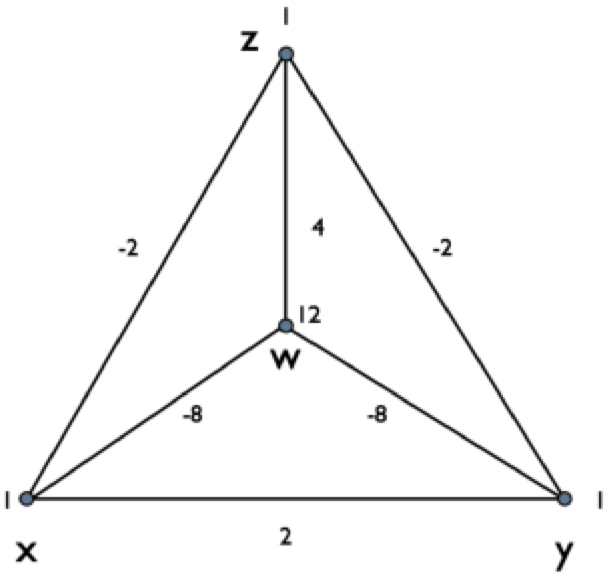
```
from dwave.system.samplers import DWaveSampler
from dwave.system.composites import FixedEmbeddingComposite
embedding = {'x': {0}, 'y': {4}, 'z': {1}, 'zz': {5}}
sampler = DWaveSampler(endpoint='https://cloud.dwavesys.com/sapi',
    token='DEV-YOUR-TOKEN', solver='DW_2000Q_2_1')
sampler_embedded = FixedEmbeddingComposite(sampler, embedding)
Q = {('x‘,'y'):1, ('y‘,'z'):-2, ('x‘,'zz'):-2, ('z‘,'zz'):-2, ('z‘,'z'):2.5, ('zz‘,'zz'):2.5}
response = sampler_embedded.sample_qubo(Q, num_reads=5000)
for datum in response.data(['sample', 'energy', 'num_occurrences']):
  print(datum.sample, "Energy: ", datum.energy, "Occurrences: ",
    datum.num_occurrences)
```

- Output:

```
({'y': 1, 'x': 0, 'z': 0, 'zz': 0}, 'Energy: ', 0.0, 'Occurrences: ', 887)
({'y': 1, 'x': 1, 'z': 1, 'zz': 1}, 'Energy: ', 0.0, 'Occurrences: ', 448)
({'y': 0, 'x': 0, 'z': 0, 'zz': 0}, 'Energy: ', 0.0, 'Occurrences: ', 2303)
({'y': 0, 'x': 1, 'z': 0, 'zz': 0}, 'Energy: ', 0.0, 'Occurrences: ', 1359)
({'y': 0, 'x': 1, 'z': 0, 'zz': 1}, 'Energy: ', 0.5, 'Occurrences: ', 1)
({'y': 1, 'x': 1, 'z': 0, 'zz': 0}, 'Energy: ', 1.0, 'Occurrences: ', 2)
```

Building Block: Half adder QUBO

- Formulate as qubo
- Requires embedding
- Need ancilla bits
- ➢ Need tool support



```
  c    s    y    x   objective
 ---  ---  ---  ---  ---------
  0    0    0    0       0
  0    0    0    1       1
  0    0    1    0       1
  0    0    1    1       4
  0    1    0    0       1
  0    1    0    1       0
  0    1    1    0       0
  0    1    1    1       1
  1    0    0    0       12
  1    0    0    1       5
  1    0    1    0       5
  1    0    1    1       0
  1    1    0    0       17
  1    1    0    1       8
  1    1    1    0       8
  1    1    1    1       1
```

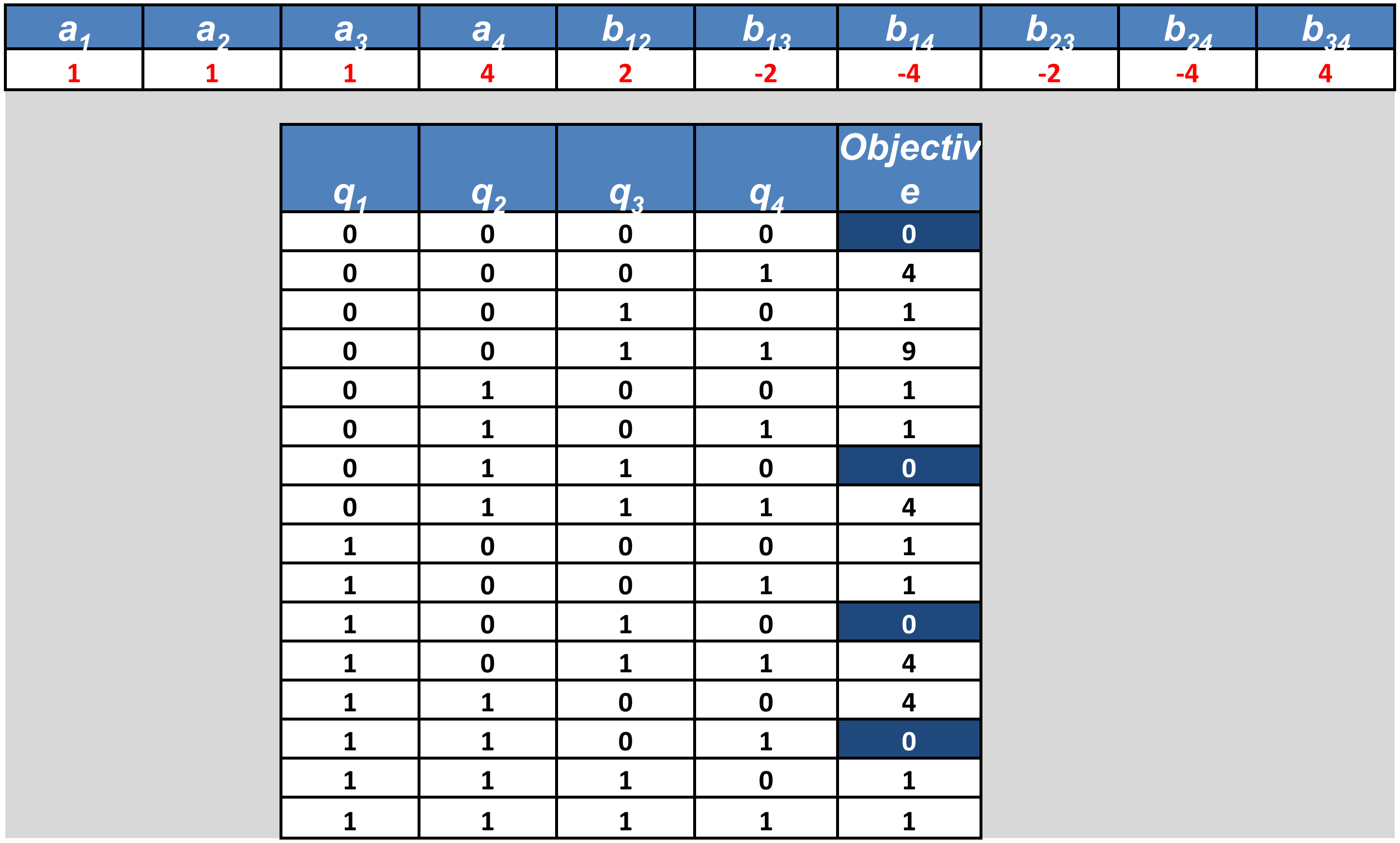
Half Adder Derived from QUBO

- Consider ground states
- Find common equation (binary): $x+y=s+2c$
- Convert into minimization of a QUBO
  - subtract RHS: $x+y-s-2c=0$
- Transform into minimization problem
  - square it: $(x+y-s-2c)^2 = (x+y-s-2c) * (x+y-s-2c) =$
    $x^2+xy-xs-2xc+yx+y^2-ys-2yc-sx-sy+s^2+2sc-2cx-2cy+2cs+4c^2=$
    $x^2+y^2+s^2+4c^2+2xy-2xs-4xc-2ys-4yc+4sc$
  - Note, for a boolean $b \epsilon \{0,1\}$, $b^2=b$
    So we can write
    $x+y+s+4c+2xy-2xs-4xc-2ys-4yc+4sc$

x	y	s	c
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Check Half Adder in Quantum Apprentice

- We have: $x+y+s+4c+2xy-2xs-4xc-2ys-4yc+4sc$, same as:



a_1	a_2	a_3	a_4	b_{12}	b_{13}	b_{14}	b_{23}	b_{24}	b_{34}
1	1	1	4	2	-2	-4	-2	-4	4

q_1	q_2	q_3	q_4	Objective
0	0	0	0	0
0	0	0	1	4
0	0	1	0	1
0	0	1	1	9
0	1	0	0	1
0	1	0	1	1
0	1	1	0	0
0	1	1	1	4
1	0	0	0	1
1	0	0	1	1
1	0	1	0	0
1	0	1	1	4
1	1	0	0	4
1	1	0	1	0
1	1	1	0	1
1	1	1	1	1

- Note: another valid solution was a few slides back

Hardware “Hadder”

- Objective function: $x+y+s+4c+2xy-2xs-4xc-2ys-4yc+4sc$

```
from dwave.system.samplers import DWaveSampler
from dwave.system.composites import EmbeddingComposite
sampler = DWaveSampler(endpoint='https://cloud.dwavesys.com/sapi',
    token='DEV-YOUR-TOKEN', solver='DW_2000Q_2_1')
sampler_embedded = EmbeddingComposite(sampler)
Q = {('x', 'x'): 1, ('y', 'y'): 1, ('s', 's'): 1, ('c', 'c'): 4,
    ('x', 'y'): 2, ('x', 's'):-2, ('x', 'c'):-4, ('y', 's'):-2, ('y', 'c'):-4, ('s', 'c'): 4 }
response = sampler_embedded.sample_qubo(Q, num_reads=5000)
for datum in response.data(['sample', 'energy', 'num_occurrences']):
  print(datum.sample, "Energy: ", datum.energy, "Occurrences: ",
    datum.num_occurrences)
```

- Output:

```
({'y': 1, 'x': 0, 'c': 0, 's': 1}, 'Energy: ', 0.0, 'Occurrences: ', 396)
({'y': 0, 'x': 1, 'c': 0, 's': 1}, 'Energy: ', 0.0, 'Occurrences: ', 650)
({'y': 0, 'x': 0, 'c': 0, 's': 0}, 'Energy: ', 0.0, 'Occurrences: ', 363)
({'y': 1, 'x': 1, 'c': 1, 's': 0}, 'Energy: ', 0.0, 'Occurrences: ', 3558)
({'y': 0, 'x': 1, 'c': 1, 's': 0}, 'Energy: ', 1.0, 'Occurrences: ', 14)
({'y': 1, 'x': 0, 'c': 1, 's': 0}, 'Energy: ', 1.0, 'Occurrences: ', 16)
({'y': 1, 'x': 1, 'c': 1, 's': 1}, 'Energy: ', 1.0, 'Occurrences: ', 2)
({'y': 0, 'x': 1, 'c': 1, 's': 0}, 'Energy: ', 1.0, 'Occurrences: ', 1)
```

Explicitly Embed Half Adder (Option)

- Let's embed it: x+y+s+4c+2xy-2xs-4xc-2ys-4yc+4sc

a_1	a_2	a_3	a_4	b_{12}	b_{13}	b_{14}	b_{23}	b_{24}	b_{34}
1	1	1	4	2	-2	-4	-2	-4	4

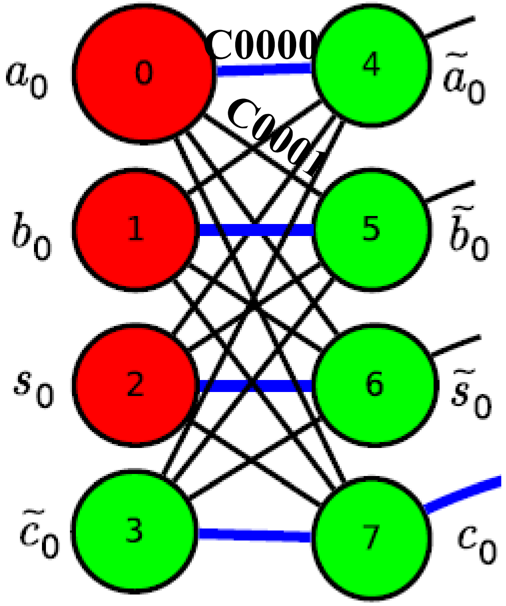
  - x → x, x' etc.
- Make sure it's still an integer (why?)
  - multiply weights/strength (here: by 2)

a_1	a_2	a_3	a_4	b_{12}	b_{13}	b_{14}	b_{23}	b_{24}	b_{34}
2	2	2	8	4	-4	-8	-4	-8	8

- Add "equal" relation with a maximal negative strength per embedding
  - So x, x' need a connecting coupler with max. strength, say, -200
  - x,x' get proportional weight of 2/2+200/2=101
    - Half the original wait + half the absolute value of the coupler
  - Same for x, y' (optionally also s,s' and c,c')

QMI for Half Adder

- Resulting embedding (for bit 0 of x,y,s,c embedded):
  - x=Q0, x'=Q4, A_0=101, A_4=101, B_{14}=-200, etc. for y,s,c
    - -b_{12}→C0001 etc.
  - Written as QMI:



Quantum Machine Assembly (QMI)

```
Q0000 <== 101
Q0001 <== 101
Q0002 <== 101
…
C0000 <== -200
C0001 <== 4
…
```

Specify Half Adder directly as QUBO

- Objective function: 101x+101y+101s+104c+101xx+101yy+101ss+104cc+4xyy-4xss-4xcc-4yss-8ycc+8scc, also -200x*xx, -200y*yy, -200s*ss, -200c*cc

```
from dwave.system.samplers import DWaveSampler
from dwave.system.composites import FixedEmbeddingComposite
embedding = {        'x': {0}, 'xx': {4},
                     'y': {1}, 'yy': {5},
                     's': {2}, 'ss': {6},
                     'c': {3}, 'cc': {7} }
sampler = DWaveSampler(endpoint='https://cloud.dwavesys.com/sapi', token='DEV-YOUR-
   TOKEN', solver='DW_2000Q_2_1')
sampler_embedded = FixedEmbeddingComposite(sampler, embedding)
Q = {('x', 'x'): 101, ('y', 'y'): 101, ('s', 's'): 101, ('c', 'c'): 104,
   ('xx','xx'):101, ('yy','yy'):101, ('ss','ss'):101, ('cc','cc'):104,
   ('x', 'yy'): 4, ('x', 'ss'):-4, ('x', 'cc'):-8,
   ('y', 'ss'):-4, ('y', 'cc'):-8,
   ('s', 'cc'): 8,
   ('x', 'xx'): -200, ('y', 'yy'): -200, ('s', 'ss'): -200, ('c', 'cc'): -200 }
response = sampler_embedded.sample_qubo(Q, num_reads=5000)
for datum in response.data(['sample', 'energy', 'num_occurrences']):
 print(datum.sample, "Energy: ", datum.energy, "Occurrences: ", datum.num_occurrences)
```

Specify Half Adder directly as QUBO

- Objective function: $2.5z+2.5z'+xy-xz-yz-xz'-yz'-2zz'$

…

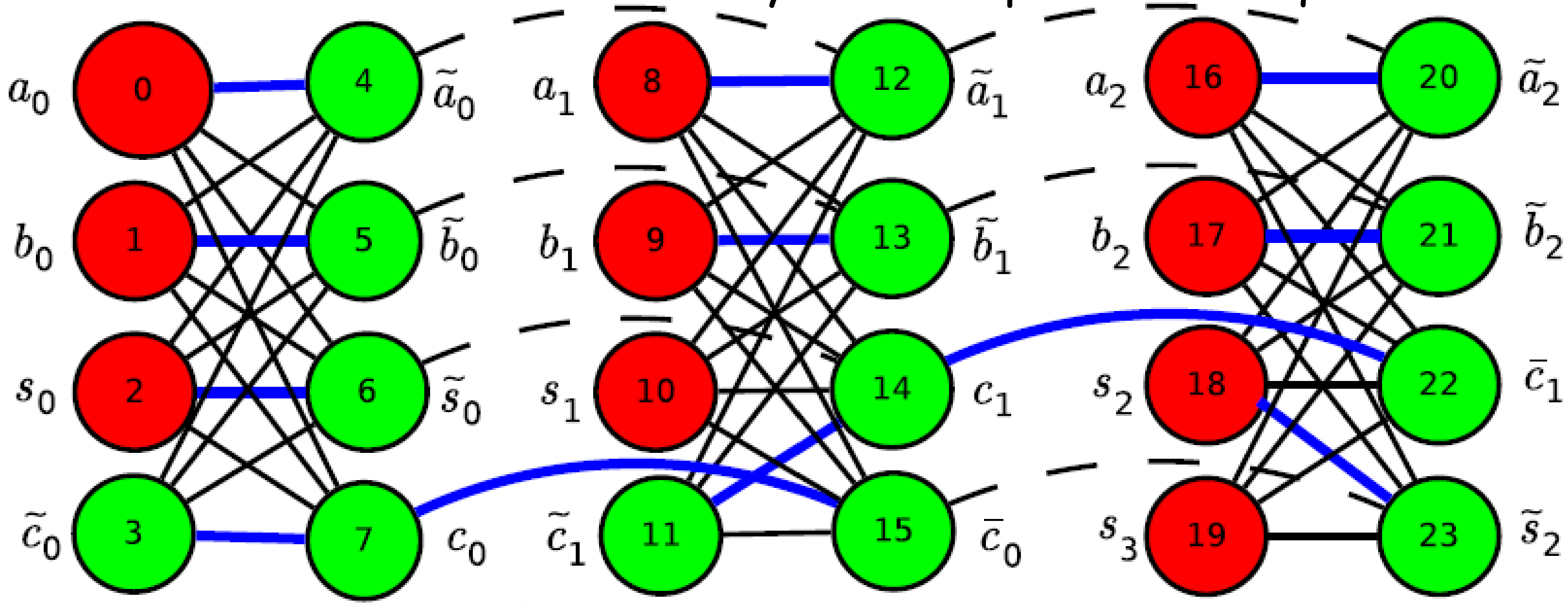
- Output:

```
({'c': 1, 'cc': 1, 'xx': 1, 'yy': 1, 'ss': 0, 's': 0, 'y': 1, 'x': 1}, 'Energy: ', 0.0, 'Occurrences: ‘ , 716)
({'c': 0, 'cc': 0, 'xx': 0, 'yy': 0, 'ss': 0, 's': 0, 'y': 0, 'x': 0}, 'Energy: ', 0.0, 'Occurrences: ‘ , 583)
({'c': 0, 'cc': 0, 'xx': 1, 'yy': 0, 'ss': 1, 's': 1, 'y': 0, 'x': 1}, 'Energy: ', 0.0, 'Occurrences: ‘ , 212)
({'c': 0, 'cc': 0, 'xx': 0, 'yy': 1, 'ss': 1, 's': 1, 'y': 1, 'x': 0}, 'Energy: ', 0.0, 'Occurrences: ‘ , 283)
```

…

More complex example: 3-bit Adder

- Add two 3-bit numbers a,b in range [0-7]
  - Returns 4-bit sum s in [0-14] range and carry bits c (14 qbits)
  - 10 ancilla bits indicated by blue couplers → 24 qubits total



  - Notice: Embed $\tilde{c}_0$,c_0,$\bar{c}_0$ by adding their original weights in half+full adder and then dividing by chain length: (4+1)/3
    - better: find LCM(5,3), multiply orig. weights/strength by 3&5