Decoding Quantum Codes

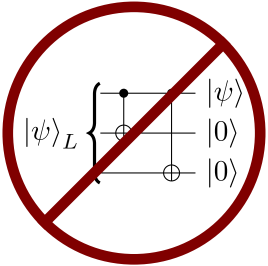
Ben Criger

June 3, 2016

Introduction: What is Decoding?

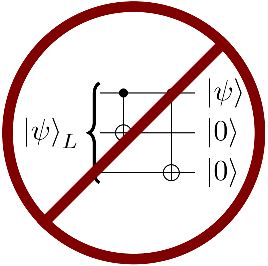
What is decoding?

- Decoding is not the process of getting an unencoded state from an encoded state:



What is decoding?

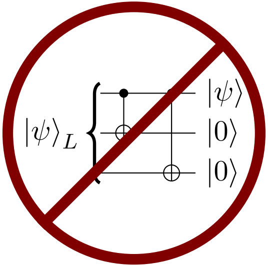
- Decoding is not the process of getting an unencoded state from an encoded state:



- It is the assignment of an error to a set of syndrome measurements.

What is decoding?

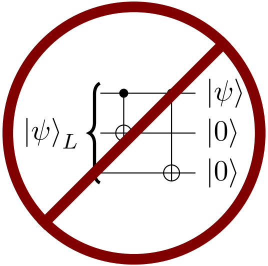
- Decoding is not the process of getting an unencoded state from an encoded state:



- It is the assignment of an error to a set of syndrome measurements.
- The name comes from the decoding of classical codes, which infers a code state from a corrupted state.

What is decoding?

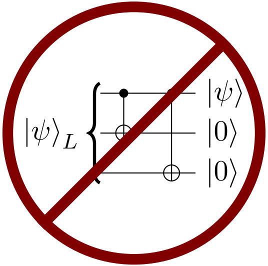
- Decoding is not the process of getting an unencoded state from an encoded state:



- It is the assignment of an error to a set of syndrome measurements.
- The name comes from the decoding of classical codes, which infers a code state from a corrupted state.
- This is the most challenging problem for classical electronics in QC.

What is decoding?

- Decoding is not the process of getting an unencoded state from an encoded state:



- It is the assignment of an error to a set of syndrome measurements.
- The name comes from the decoding of classical codes, which infers a code state from a corrupted state.
- This is the most challenging problem for classical electronics in QC.

To get a feel for decoding, let's focus on classical codes for the moment.

Classical Coding Theory in a Nutshell: Encoding

- A *code* is a set of bitstrings (vectors over $\mathbb{Z}_2^n$)

Classical Coding Theory in a Nutshell: Encoding

- A *code* is a set of bitstrings (vectors over $\mathbb{Z}_2^n$)
- A *linear code* is a code such that, for strings a and b in the code, $a \oplus b$ is also in the code.

Classical Coding Theory in a Nutshell: Encoding

- A *code* is a set of bitstrings (vectors over $\mathbb{Z}_2^n$)
- A *linear code* is a code such that, for strings a and b in the code, $a \oplus b$ is also in the code.
- These strings are in the *kernel* of the *parity-check matrix* H, they can be expressed using the *generator matrix* G.

Classical Coding Theory in a Nutshell: Encoding

- A *code* is a set of bitstrings (vectors over $\mathbb{Z}_2^n$)
- A *linear code* is a code such that, for strings a and b in the code, $a \oplus b$ is also in the code.
- These strings are in the *kernel* of the *parity-check matrix* H, they can be expressed using the *generator matrix* G.

$$G_3 = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}, \; H_3 = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \end{bmatrix} \quad G_7 = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \end{bmatrix}, \; H_7 = \begin{bmatrix} 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 \end{bmatrix}$$

Repetition Code Hamming Code

Classical Coding Theory in a Nutshell: Encoding

- A *code* is a set of bitstrings (vectors over $\mathbb{Z}_2^n$)
- A *linear code* is a code such that, for strings a and b in the code, $a \oplus b$ is also in the code.
- These strings are in the *kernel* of the *parity-check matrix* H, they can be expressed using the *generator matrix* G.

$$G_3 = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}, \; H_3 = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \end{bmatrix} \quad G_7 = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \end{bmatrix}, \; H_7 = \begin{bmatrix} 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 \end{bmatrix}$$

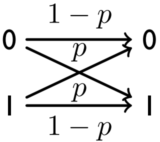
Repetition Code　　　　Hamming Code

A message is multiplied by G to encode, and transmitted over a noisy channel, where it can be randomly flipped.

Decoding Classical Codes

Classical Coding Theory in a Nutshell: Decoding

We typically assume that the errors are *symmetric*, that their output doesn't depend on whether a 0 or 1 was input.

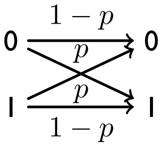


We obtain a *syndrome* by multiplying by the parity matrix:

$$H(\underbrace{e}_{\text{error}} \oplus \underbrace{c}_{\text{codeword}}) = He + \underbrace{Hc}_{Hc=0} = He$$

Classical Coding Theory in a Nutshell: Decoding

We typically assume that the errors are *symmetric*, that their output doesn't depend on whether a 0 or 1 was input.



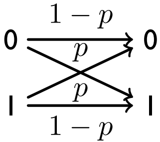
We obtain a *syndrome* by multiplying by the parity matrix:

$$H(\underbrace{e}_{\text{error}} \oplus \underbrace{c}_{\text{codeword}}) = He + \underbrace{Hc}_{Hc=0} = He$$

- A *decoder* is any function with $s = He$ as its input, and an error e' as its output, such that $He' = s$.

Classical Coding Theory in a Nutshell: Decoding

We typically assume that the errors are *symmetric*, that their output doesn't depend on whether a 0 or 1 was input.



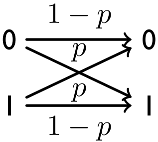
We obtain a *syndrome* by multiplying by the parity matrix:

$$H(\underbrace{e}_{\text{error}} \oplus \underbrace{c}_{\text{codeword}}) = He + \underbrace{Hc}_{Hc=0} = He$$

- A *decoder* is any function with $s = He$ as its input, and an error e' as its output, such that $He' = s$.
- A decoder succeeds when $e \oplus e' = \vec{0}$ and fails when $e \oplus e' = c$ for some codeword c. Prove: these are the only possible cases.

Classical Coding Theory in a Nutshell: Decoding

We typically assume that the errors are *symmetric*, that their output doesn't depend on whether a 0 or 1 was input.



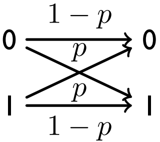
We obtain a *syndrome* by multiplying by the parity matrix:

$$H(\underbrace{e}_{\text{error}} \oplus \underbrace{c}_{\text{codeword}}) = He + \underbrace{Hc}_{Hc=0} = He$$

- A *decoder* is any function with $s = He$ as its input, and an error e' as its output, such that $He' = s$.
- A decoder succeeds when $e \oplus e' = \vec{0}$ and fails when $e \oplus e' = c$ for some codeword c. Prove: these are the only possible cases.
- A decoder usually uses one of two operating principles:

Classical Coding Theory in a Nutshell: Decoding

We typically assume that the errors are *symmetric*, that their output doesn't depend on whether a 0 or 1 was input.



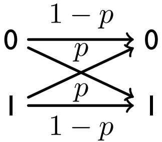
We obtain a *syndrome* by multiplying by the parity matrix:

$$H(\underbrace{e}_{\text{error}} \oplus \underbrace{c}_{\text{codeword}}) = He + \underbrace{Hc}_{Hc=0} = He$$

- A *decoder* is any function with $s = He$ as its input, and an error e' as its output, such that $He' = s$.
- A decoder succeeds when $e \oplus e' = \vec{0}$ and fails when $e \oplus e' = c$ for some codeword c.
  Prove: these are the only possible cases.
- A decoder usually uses one of two operating principles:
  - *Maximum Likelihood* (ML): $e' = \underset{v|Hv=s}{\operatorname{argmax}} [p(v)]$

Classical Coding Theory in a Nutshell: Decoding

We typically assume that the errors are *symmetric*, that their output doesn't depend on whether a 0 or 1 was input.



We obtain a *syndrome* by multiplying by the parity matrix:

$$H(\underbrace{e}_{\text{error}} \oplus \underbrace{c}_{\text{codeword}}) = He + \underbrace{Hc}_{Hc=0} = He$$

- A *decoder* is any function with $s = He$ as its input, and an error e' as its output, such that $He' = s$.
- A decoder succeeds when $e \oplus e' = \vec{0}$ and fails when $e \oplus e' = c$ for some codeword c. Prove: these are the only possible cases.
- A decoder usually uses one of two operating principles:
  - *Maximum Likelihood* (ML): $e' = \underset{v|Hv=s}{\operatorname{argmax}} [p(v)]$
  - *Minimum Weight* (MW): $e' = \underset{v|Hv=s}{\operatorname{argmin}} [|v|]$, equivalent to ML when error rates are small and equal.

Example: Hamming Code Decoding

Example: Hamming Code Decoding

- There are 8 correctable events: $\{$no error$\} \cup \{$error on $j, j = 0, 1, \ldots, 6\}$

Example: Hamming Code Decoding

- There are 8 correctable events: $\{\text{no error}\} \cup \{\text{error on } j, j = 0, 1, \ldots, 6\}$
- There are 8 possible syndromes: $\begin{bmatrix} 0 & 0 & 0 \end{bmatrix}^{\mathsf{T}}$ up to $\begin{bmatrix} 1 & 1 & 1 \end{bmatrix}^{\mathsf{T}}$

Example: Hamming Code Decoding

- There are 8 correctable events: $\{\text{no error}\} \cup \{\text{error on } j, j = 0, 1, \ldots, 6\}$
- There are 8 possible syndromes: $\begin{bmatrix} 0 & 0 & 0 \end{bmatrix}^{\mathsf{T}}$ up to $\begin{bmatrix} 1 & 1 & 1 \end{bmatrix}^{\mathsf{T}}$
- The columns of H are independent.

Example: Hamming Code Decoding

- There are 8 correctable events: $\{$no error$\} \cup \{$error on $j, j = 0, 1, \ldots, 6\}$
- There are 8 possible syndromes: $\begin{bmatrix} 0 & 0 & 0 \end{bmatrix}^{\mathsf{T}}$ up to $\begin{bmatrix} 1 & 1 & 1 \end{bmatrix}^{\mathsf{T}}$
- The columns of H are independent.

This lets us use H as a look-up table for MW decoding:

$$\underbrace{\begin{bmatrix} 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 \end{bmatrix}}_{\text{parity-check matrix}} \underbrace{\begin{bmatrix} 1 \\ 1 \\ 0 \\ 1 \\ 0 \\ 1 \\ 1 \end{bmatrix}}_{\substack{\text{noisy} \\ \text{message}}} = \underbrace{\begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix}}_{\text{syndrome}} \therefore e' = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}, c = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \\ 0 \\ 1 \\ 1 \end{bmatrix} \sim \underbrace{\begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}}_{\substack{\text{original} \\ \text{message}}}$$

Codes like this are called *perfect* or *non-degenerate*.

Example: Repetition Code Decoding

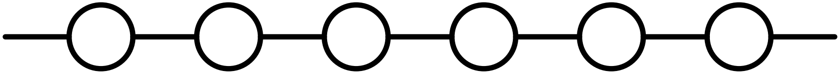
- *Degenerate* codes have multiple errors which are consistent with the syndrome.

Example: Repetition Code Decoding

- *Degenerate* codes have multiple errors which are consistent with the syndrome.
- As long as the number of errors consistent with the syndrome is small, we can simply iterate over the set to determine which is best.

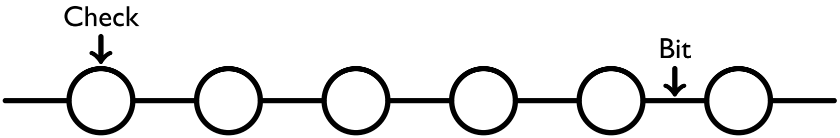
Example: Repetition Code Decoding

- *Degenerate* codes have multiple errors which are consistent with the syndrome.
- As long as the number of errors consistent with the syndrome is small, we can simply iterate over the set to determine which is best.



Example: Repetition Code Decoding

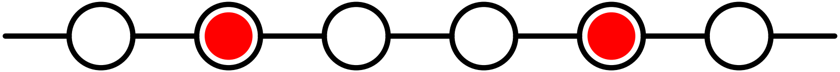
- *Degenerate* codes have multiple errors which are consistent with the syndrome.
- As long as the number of errors consistent with the syndrome is small, we can simply iterate over the set to determine which is best.



Seven-Bit Repetition Code

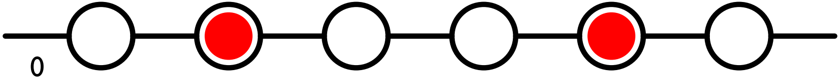
Example: Repetition Code Decoding

- *Degenerate* codes have multiple errors which are consistent with the syndrome.
- As long as the number of errors consistent with the syndrome is small, we can simply iterate over the set to determine which is best.



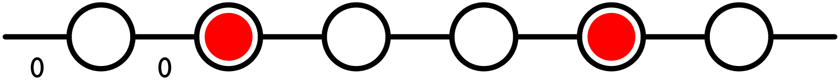
Example: Repetition Code Decoding

- *Degenerate* codes have multiple errors which are consistent with the syndrome.
- As long as the number of errors consistent with the syndrome is small, we can simply iterate over the set to determine which is best.



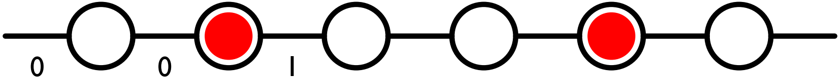
Example: Repetition Code Decoding

- *Degenerate* codes have multiple errors which are consistent with the syndrome.
- As long as the number of errors consistent with the syndrome is small, we can simply iterate over the set to determine which is best.



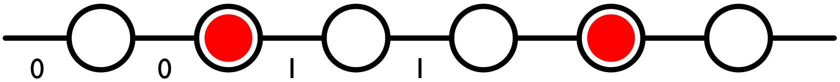
Example: Repetition Code Decoding

- *Degenerate* codes have multiple errors which are consistent with the syndrome.
- As long as the number of errors consistent with the syndrome is small, we can simply iterate over the set to determine which is best.



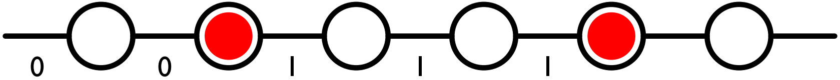
Example: Repetition Code Decoding

- *Degenerate* codes have multiple errors which are consistent with the syndrome.
- As long as the number of errors consistent with the syndrome is small, we can simply iterate over the set to determine which is best.



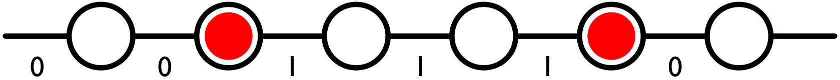
Example: Repetition Code Decoding

- *Degenerate* codes have multiple errors which are consistent with the syndrome.
- As long as the number of errors consistent with the syndrome is small, we can simply iterate over the set to determine which is best.



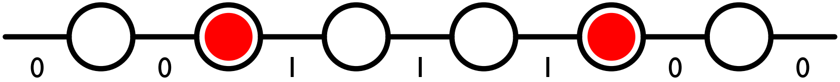
Example: Repetition Code Decoding

- *Degenerate* codes have multiple errors which are consistent with the syndrome.
- As long as the number of errors consistent with the syndrome is small, we can simply iterate over the set to determine which is best.



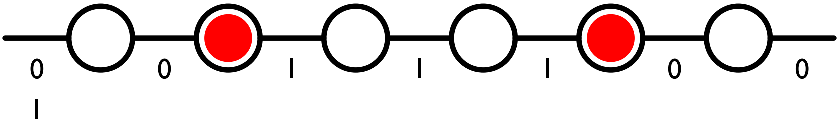
Example: Repetition Code Decoding

- *Degenerate* codes have multiple errors which are consistent with the syndrome.
- As long as the number of errors consistent with the syndrome is small, we can simply iterate over the set to determine which is best.



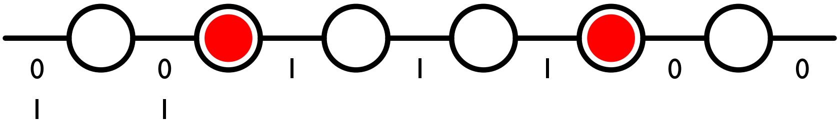
Example: Repetition Code Decoding

- *Degenerate* codes have multiple errors which are consistent with the syndrome.
- As long as the number of errors consistent with the syndrome is small, we can simply iterate over the set to determine which is best.



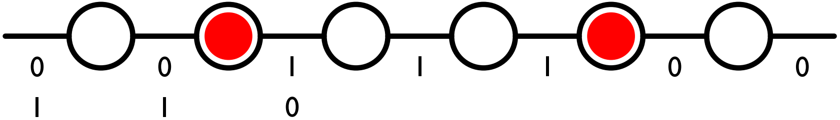
Example: Repetition Code Decoding

- *Degenerate* codes have multiple errors which are consistent with the syndrome.
- As long as the number of errors consistent with the syndrome is small, we can simply iterate over the set to determine which is best.



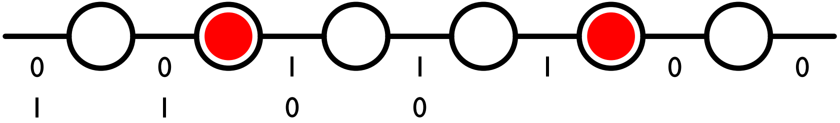
Example: Repetition Code Decoding

- *Degenerate* codes have multiple errors which are consistent with the syndrome.
- As long as the number of errors consistent with the syndrome is small, we can simply iterate over the set to determine which is best.



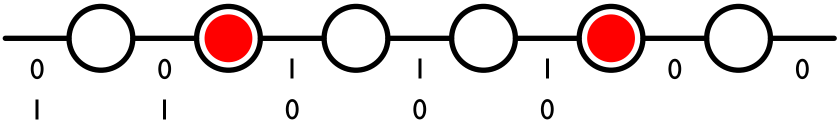
Example: Repetition Code Decoding

- *Degenerate* codes have multiple errors which are consistent with the syndrome.
- As long as the number of errors consistent with the syndrome is small, we can simply iterate over the set to determine which is best.



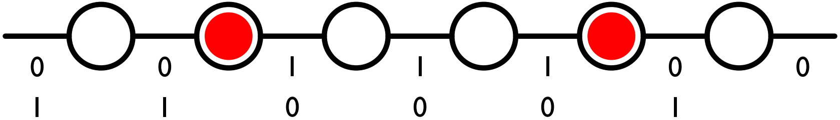
Example: Repetition Code Decoding

- *Degenerate* codes have multiple errors which are consistent with the syndrome.
- As long as the number of errors consistent with the syndrome is small, we can simply iterate over the set to determine which is best.



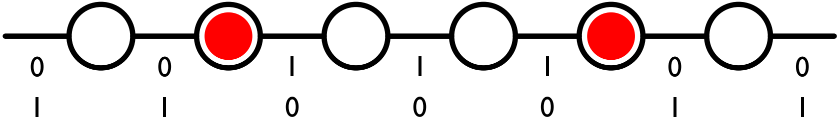
Example: Repetition Code Decoding

- *Degenerate* codes have multiple errors which are consistent with the syndrome.
- As long as the number of errors consistent with the syndrome is small, we can simply iterate over the set to determine which is best.



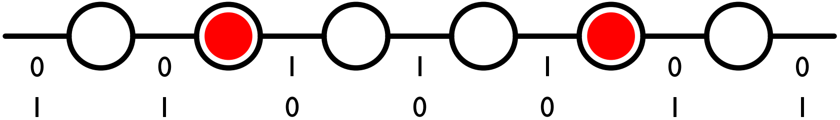
Example: Repetition Code Decoding

- *Degenerate* codes have multiple errors which are consistent with the syndrome.
- As long as the number of errors consistent with the syndrome is small, we can simply iterate over the set to determine which is best.



Example: Repetition Code Decoding

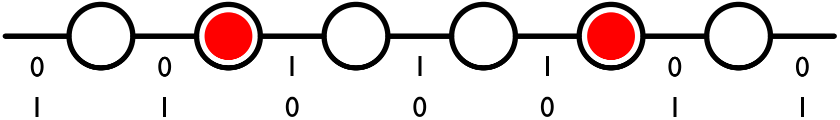
- *Degenerate* codes have multiple errors which are consistent with the syndrome.
- As long as the number of errors consistent with the syndrome is small, we can simply iterate over the set to determine which is best.



This syndrome decoder need only consider two cases, each of which takes time $\propto n$ to evaluate.

Example: Repetition Code Decoding

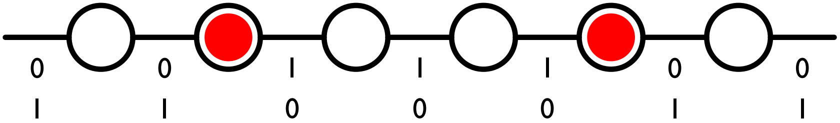
- *Degenerate* codes have multiple errors which are consistent with the syndrome.
- As long as the number of errors consistent with the syndrome is small, we can simply iterate over the set to determine which is best.



This syndrome decoder need only consider two cases, each of which takes time $\propto n$ to evaluate.
The errors in the 1D repetition code "form strings" with syndromes "appearing at the endpoints" (a pattern we will see again).

Example: Repetition Code Decoding

- *Degenerate* codes have multiple errors which are consistent with the syndrome.
- As long as the number of errors consistent with the syndrome is small, we can simply iterate over the set to determine which is best.



This syndrome decoder need only consider two cases, each of which takes time $\propto n$ to evaluate.

The errors in the 1D repetition code "form strings" with syndromes "appearing at the endpoints" (a pattern we will see again).

Place bits in the cells of a lattice (squares of a square tiling, cubes of a cubic tiling) and checks between neighbouring cells. How many errors are consistent with a given syndrome, and what structure do the syndromes have?

The CSS Construction

CSS Codes

Calderbank-Shor-Steane (CSS) codes can be expressed as pairs of classical codes. To see this, we consider the action of two maps on the repetition code checks:

CSS Codes

Calderbank-Shor-Steane (CSS) codes can be expressed as pairs of classical codes. To see this, we consider the action of two maps on the repetition code checks:

$$\{0,\,1\} \begin{matrix} \nearrow \\ \searrow \end{matrix} \begin{matrix} \{I,\,X\} \\ \{I,\,Z\} \end{matrix} \begin{bmatrix} 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 \end{bmatrix} \mapsto \begin{bmatrix} X & X & I & I & I \\ I & X & X & I & I \\ I & I & X & X & I \\ I & I & I & X & X \end{bmatrix} \text{ or } \begin{bmatrix} Z & Z & I & I & I \\ I & Z & Z & I & I \\ I & I & Z & Z & I \\ I & I & I & Z & Z \end{bmatrix}$$

CSS Codes

Calderbank-Shor-Steane (CSS) codes can be expressed as pairs of classical codes. To see this, we consider the action of two maps on the repetition code checks:

$$\{0, 1\} \begin{matrix} \nearrow \\ \searrow \end{matrix} \begin{matrix} \{I, X\} \\ \{I, Z\} \end{matrix} \begin{bmatrix} 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 \end{bmatrix} \mapsto \begin{bmatrix} X & X & I & I & I \\ I & X & X & I & I \\ I & I & X & X & I \\ I & I & I & X & X \end{bmatrix} \text{ or } \begin{bmatrix} Z & Z & I & I & I \\ I & Z & Z & I & I \\ I & I & Z & Z & I \\ I & I & I & Z & Z \end{bmatrix}$$

These stabilisers protect a quantum state against Z or X errors with the same support as correctable errors in the original classical code.

CSS Codes

Calderbank-Shor-Steane (CSS) codes can be expressed as pairs of classical codes. To see this, we consider the action of two maps on the repetition code checks:

$$\{0,\,1\} \begin{smallmatrix} \nearrow \\ \searrow \end{smallmatrix} \begin{matrix} \{I,\,X\} \\ \{I,\,Z\} \end{matrix} \begin{bmatrix} 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 \end{bmatrix} \mapsto \begin{bmatrix} X & X & I & I & I \\ I & X & X & I & I \\ I & I & X & X & I \\ I & I & I & X & X \end{bmatrix} \text{ or } \begin{bmatrix} Z & Z & I & I & I \\ I & Z & Z & I & I \\ I & I & Z & Z & I \\ I & I & I & Z & Z \end{bmatrix}$$

These stabilisers protect a quantum state against Z or X errors with the same support as correctable errors in the original classical code.

We can protect against both types of error by measuring both X and Z checks, but the checks have to commute.

CSS Codes

Calderbank-Shor-Steane (CSS) codes can be expressed as pairs of classical codes. To see this, we consider the action of two maps on the repetition code checks:

$$\{0, 1\} \begin{array}{l} \nearrow \{I, X\} \\ \searrow \{I, Z\} \end{array} \begin{bmatrix} 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 \end{bmatrix} \mapsto \begin{bmatrix} X & X & I & I & I \\ I & X & X & I & I \\ I & I & X & X & I \\ I & I & I & X & X \end{bmatrix} \text{ or } \begin{bmatrix} Z & Z & I & I & I \\ I & Z & Z & I & I \\ I & I & Z & Z & I \\ I & I & I & Z & Z \end{bmatrix}$$

These stabilisers protect a quantum state against Z or X errors with the same support as correctable errors in the original classical code.

We can protect against both types of error by measuring both X and Z checks, but the checks have to commute.

Prove that a Z operator commutes with X stabilisers (and vice versa) derived from a parity-check matrix H iff it is mapped from a codeword of the code defined by H.

CSS Codes

Not all CSS codes are useful:

CSS Codes

Not all CSS codes are useful:

$$\begin{bmatrix} Z & Z & I & \cdots & I & I \\ I & Z & Z & \cdots & I & I \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ I & I & I & \cdots & Z & Z \\ X & X & X & \cdots & X & X \end{bmatrix} = \frac{1}{\sqrt{2}}\left(|000\cdots00\rangle + |111\cdots11\rangle\right) \leftarrow \text{ No logical qubits}$$

CSS Codes

Not all CSS codes are useful:

$$\begin{bmatrix} Z & Z & I & \cdots & I & I \\ I & Z & Z & \cdots & I & I \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ I & I & I & \cdots & Z & Z \\ X & X & X & \cdots & X & X \end{bmatrix} = \frac{1}{\sqrt{2}}\left(|000\cdots00\rangle + |111\cdots11\rangle\right) \leftarrow \text{No logical qubits}$$

Dual-containing codes (whose parity checks are also codewords) can be used to make *self-dual* CSS codes, whose X and Z stabilizers are identical:

$$S = \begin{bmatrix} X & X & X & X & I & I & I \\ X & X & I & I & X & X & I \\ X & I & X & I & X & I & X \\ Z & Z & Z & Z & I & I & I \\ Z & Z & I & I & Z & Z & I \\ Z & I & Z & I & Z & I & Z \end{bmatrix}$$

This is the Steane code.

$$\begin{matrix} \overline{X} \\ \overline{Z} \end{matrix} = \begin{bmatrix} X & X & X & X & X & X & X \\ Z & Z & Z & Z & Z & Z & Z \end{bmatrix}$$

CSS Codes

Not all CSS codes are useful:

$$\begin{bmatrix} Z & Z & I & \cdots & I & I \\ I & Z & Z & \cdots & I & I \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ I & I & I & \cdots & Z & Z \\ X & X & X & \cdots & X & X \end{bmatrix} = \frac{1}{\sqrt{2}}\left(|000\cdots00\rangle + |111\cdots11\rangle\right) \leftarrow \text{ No logical qubits}$$

Dual-containing codes (whose parity checks are also codewords) can be used to make *self-dual* CSS codes, whose X and Z stabilizers are identical:

$$S = \begin{bmatrix} X & X & X & X & I & I & I \\ X & X & I & I & X & X & I \\ X & I & X & I & X & I & X \\ Z & Z & Z & Z & I & I & I \\ Z & Z & I & I & Z & Z & I \\ Z & I & Z & I & Z & I & Z \end{bmatrix}$$

$$\begin{matrix} \overline{X} \\ \overline{Z} \end{matrix} = \begin{bmatrix} X & X & X & X & X & X & X \\ Z & Z & Z & Z & Z & Z & Z \end{bmatrix}$$

This is the Steane code.

1. How many logical qubits are there?

CSS Codes

Not all CSS codes are useful:

$$\begin{bmatrix} Z & Z & I & \cdots & I & I \\ I & Z & Z & \cdots & I & I \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ I & I & I & \cdots & Z & Z \\ X & X & X & \cdots & X & X \end{bmatrix} = \frac{1}{\sqrt{2}}\left(|000\cdots00\rangle + |111\cdots11\rangle\right) \leftarrow \text{ No logical qubits}$$

Dual-containing codes (whose parity checks are also codewords) can be used to make *self-dual* CSS codes, whose X and Z stabilizers are identical:

$$S = \begin{bmatrix} X & X & X & X & I & I & I \\ X & X & I & I & X & X & I \\ X & I & X & I & X & I & X \\ Z & Z & Z & Z & I & I & I \\ Z & Z & I & I & Z & Z & I \\ Z & I & Z & I & Z & I & Z \end{bmatrix}$$

$$\begin{matrix} \overline{X} \\ \overline{Z} \end{matrix} = \begin{bmatrix} X & X & X & X & X & X & X \\ Z & Z & Z & Z & Z & Z & Z \end{bmatrix}$$

This is the Steane code.

1. How many logical qubits are there?
2. What syndrome is generated by a Y error?

CSS Codes

Not all CSS codes are useful:

$$\begin{bmatrix} Z & Z & I & \cdots & I & I \\ I & Z & Z & \cdots & I & I \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ I & I & I & \cdots & Z & Z \\ X & X & X & \cdots & X & X \end{bmatrix} = \frac{1}{\sqrt{2}}(|000\cdots00\rangle + |111\cdots11\rangle) \leftarrow \text{ No logical qubits}$$

Dual-containing codes (whose parity checks are also codewords) can be used to make *self-dual* CSS codes, whose X and Z stabilizers are identical:

$$S = \begin{bmatrix} X & X & X & X & I & I & I \\ X & X & I & I & X & X & I \\ X & I & X & I & X & I & X \\ Z & Z & Z & Z & I & I & I \\ Z & Z & I & I & Z & Z & I \\ Z & I & Z & I & Z & I & Z \end{bmatrix}$$

$$\begin{matrix} \overline{X} \\ \overline{Z} \end{matrix} = \begin{bmatrix} X & X & X & X & X & X & X \\ Z & Z & Z & Z & Z & Z & Z \end{bmatrix}$$

This is the Steane code.

1. How many logical qubits are there?
2. What syndrome is generated by a Y error?
3. What is the set of correctable errors?

CSS Codes

Not all CSS codes are useful:

$$\begin{bmatrix} Z & Z & I & \cdots & I & I \\ I & Z & Z & \cdots & I & I \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ I & I & I & \cdots & Z & Z \\ X & X & X & \cdots & X & X \end{bmatrix} = \frac{1}{\sqrt{2}}\left(|000\cdots00\rangle + |111\cdots11\rangle\right) \leftarrow \text{ No logical qubits}$$

Dual-containing codes (whose parity checks are also codewords) can be used to make *self-dual* CSS codes, whose X and Z stabilizers are identical:

$$S = \begin{bmatrix} X & X & X & X & I & I & I \\ X & X & I & I & X & X & I \\ X & I & X & I & X & I & X \\ Z & Z & Z & Z & I & I & I \\ Z & Z & I & I & Z & Z & I \\ Z & I & Z & I & Z & I & Z \end{bmatrix}$$

$$\begin{matrix} \overline{X} \\ \overline{Z} \end{matrix} = \begin{bmatrix} X & X & X & X & X & X & X \\ Z & Z & Z & Z & Z & Z & Z \end{bmatrix}$$

This is the Steane code.

1. How many logical qubits are there?
2. What syndrome is generated by a Y error?
3. What is the set of correctable errors?
4. What is the code distance?

Fault-Tolerance and Thresholds

Fault-Tolerance and Thresholds

During syndrome extraction, the qubits in a code must interact with "the outside world".

Fault-Tolerance and Thresholds

During syndrome extraction, the qubits in a code must interact with “the outside world”.
This is the fundamental source of errors in experiment.

Fault-Tolerance and Thresholds

During syndrome extraction, the qubits in a code must interact with “the outside world”.

This is the fundamental source of errors in experiment.

If we have a scalable family of codes, an extractor and a decoder, we can determine a *threshold error rate*, beneath which larger codes do better than smaller ones.

Fault-Tolerance and Thresholds

During syndrome extraction, the qubits in a code must interact with "the outside world".

This is the fundamental source of errors in experiment.

If we have a scalable family of codes, an extractor and a decoder, we can determine a *threshold error rate*, beneath which larger codes do better than smaller ones.

These error rates depend on:

Fault-Tolerance and Thresholds

During syndrome extraction, the qubits in a code must interact with "the outside world".

This is the fundamental source of errors in experiment.

If we have a scalable family of codes, an extractor and a decoder, we can determine a *threshold error rate*, beneath which larger codes do better than smaller ones.

These error rates depend on:

- the syndrome extraction circuit

Fault-Tolerance and Thresholds

During syndrome extraction, the qubits in a code must interact with “the outside world”.
This is the fundamental source of errors in experiment.
If we have a scalable family of codes, an extractor and a decoder, we can determine a *threshold error rate*, beneath which larger codes do better than smaller ones.
These error rates depend on:

- the syndrome extraction circuit
- the error model for gates/circuits

Fault-Tolerance and Thresholds

During syndrome extraction, the qubits in a code must interact with “the outside world”.
This is the fundamental source of errors in experiment.
If we have a scalable family of codes, an extractor and a decoder, we can determine a *threshold error rate*, beneath which larger codes do better than smaller ones.
These error rates depend on:

- the syndrome extraction circuit
- the error model for gates/circuits
- the code family

Fault-Tolerance and Thresholds

During syndrome extraction, the qubits in a code must interact with “the outside world”.

This is the fundamental source of errors in experiment.

If we have a scalable family of codes, an extractor and a decoder, we can determine a *threshold error rate*, beneath which larger codes do better than smaller ones.

These error rates depend on:

- the syndrome extraction circuit
- the error model for gates/circuits
- the code family
- the decoder

Fault-Tolerance and Thresholds

During syndrome extraction, the qubits in a code must interact with "the outside world".

This is the fundamental source of errors in experiment.

If we have a scalable family of codes, an extractor and a decoder, we can determine a *threshold error rate*, beneath which larger codes do better than smaller ones.

These error rates depend on:

- the syndrome extraction circuit
- the error model for gates/circuits
- the code family
- the decoder
- whether the threshold is a lower bound or an estimate.

Fault-Tolerance and Thresholds

During syndrome extraction, the qubits in a code must interact with "the outside world".
This is the fundamental source of errors in experiment.
If we have a scalable family of codes, an extractor and a decoder, we can determine a *threshold error rate*, beneath which larger codes do better than smaller ones.
These error rates depend on:

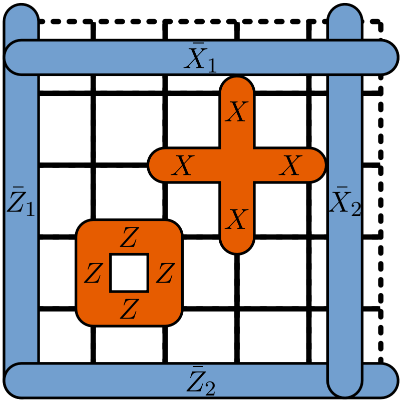
- the syndrome extraction circuit
- the error model for gates/circuits
- the code family
- the decoder
- whether the threshold is a lower bound or an estimate.

We will focus on codes which are inherently planar, so that operations can be performed locally 'on a chip'.

The Toric/Surface/Rotated Surface Code

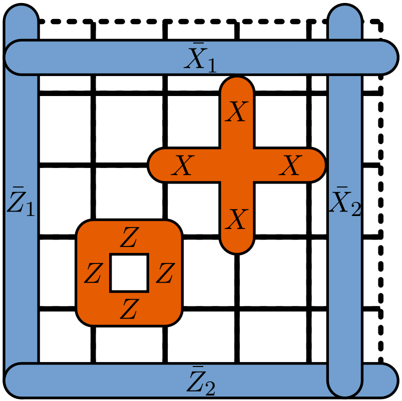
The Toric Code and Friends

Kitaev first put forth the toric code in 1997:



The Toric Code and Friends

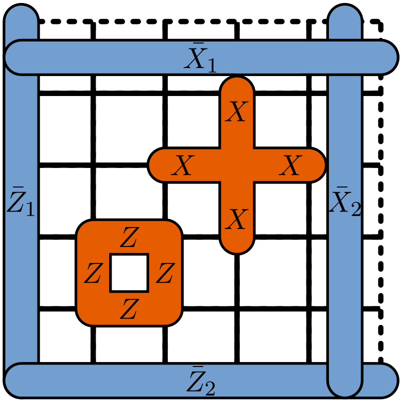
Kitaev first put forth the toric code in 1997:



- qubits placed on edges of a *lattice* (a graph that covers a surface)

The Toric Code and Friends

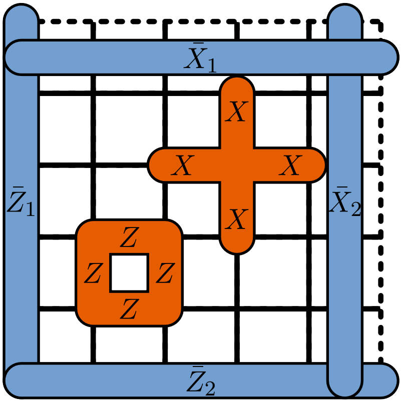
Kitaev first put forth the toric code in 1997:



- qubits placed on edges of a *lattice* (a graph that covers a surface)
- dotted edges 'wrap around' a torus (doughnut)

The Toric Code and Friends

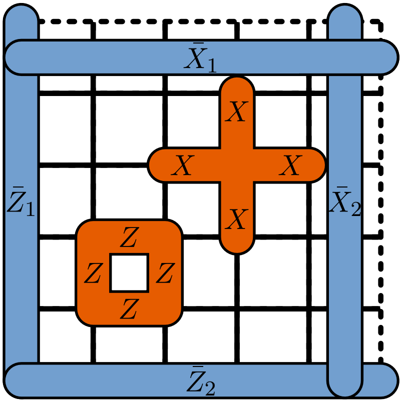
Kitaev first put forth the toric code in 1997:



- qubits placed on edges of a *lattice* (a graph that covers a surface)
- dotted edges 'wrap around' a torus (doughnut)
- $[\![n,\, k,\, d]\!] = [\![2l^2,\, 2,\, l]\!]$ for an l-by-l lattice

The Toric Code and Friends

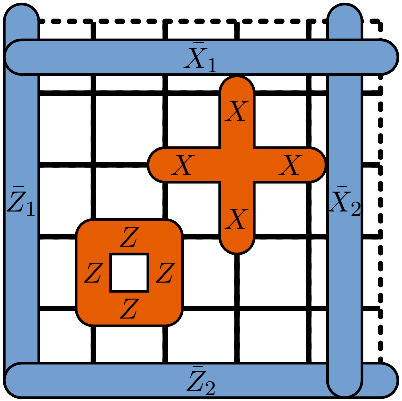
Kitaev first put forth the toric code in 1997:



- qubits placed on edges of a *lattice* (a graph that covers a surface)
- dotted edges 'wrap around' a torus (doughnut)
- $[\![n,\, k,\, d]\!] = [\![2l^2,\, 2,\, l]\!]$ for an l-by-l lattice
- X checks are defined on *stars* (edges neighbouring a vertex)

The Toric Code and Friends

Kitaev first put forth the toric code in 1997:



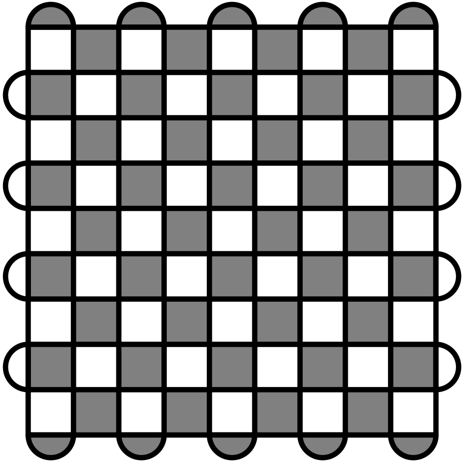
- qubits placed on edges of a *lattice* (a graph that covers a surface)
- dotted edges 'wrap around' a torus (doughnut)
- $[\![n, k, d]\!] = [\![2l^2, 2, l]\!]$ for an l-by-l lattice
- X checks are defined on *stars* (edges neighbouring a vertex)
- Z checks are defined on *plaquettes* (edges neighbouring a face)

The Toric Code and Friends

The toric code is mathematically 2D (position is described by 2 parameters), but not physically 2D (planar). To solve this, “cut” a square out:

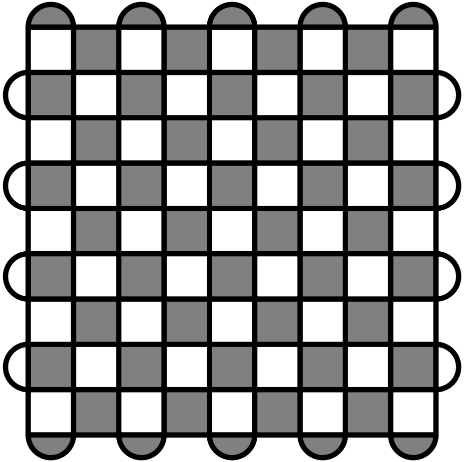
The Toric Code and Friends

The toric code is mathematically 2D (position is described by 2 parameters), but not physically 2D (planar). To solve this, “cut” a square out:



The Toric Code and Friends

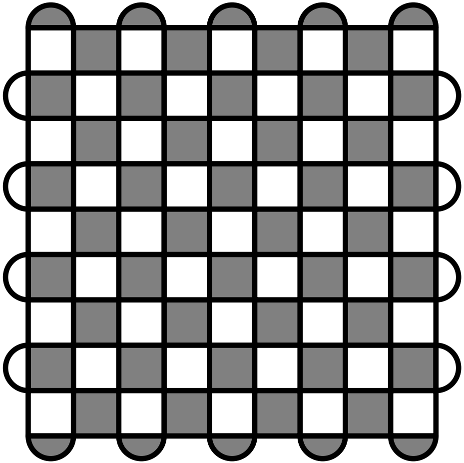
The toric code is mathematically 2D (position is described by 2 parameters), but not physically 2D (planar). To solve this, “cut” a square out:



- Developed in 2011 by Horsman et al.

The Toric Code and Friends

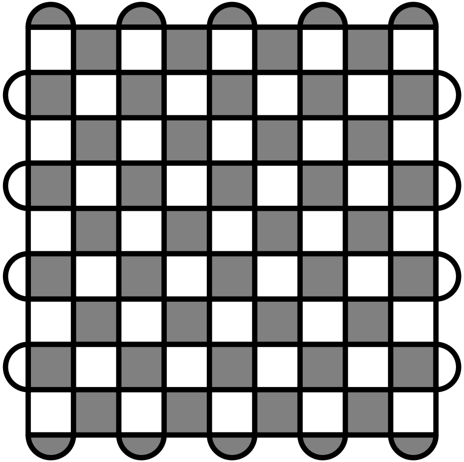
The toric code is mathematically 2D (position is described by 2 parameters), but not physically 2D (planar). To solve this, "cut" a square out:



- Developed in 2011 by Horsman et al.
- $[\![n, k, d]\!] = [\![l^2, 1, l]\!]$ ("half" of a toric code)

The Toric Code and Friends

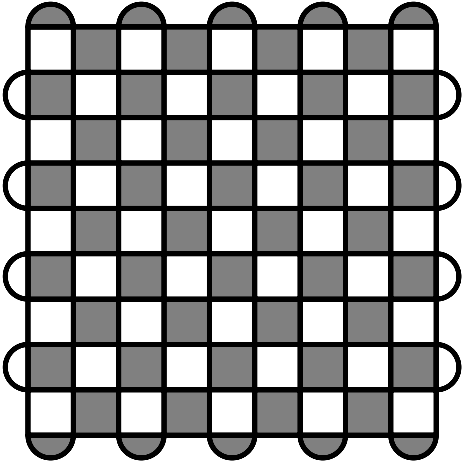
The toric code is mathematically 2D (position is described by 2 parameters), but not physically 2D (planar). To solve this, "cut" a square out:



- Developed in 2011 by Horsman et al.
- $[\![n,\, k,\, d]\!] = [\![l^2,\, 1,\, l]\!]$ ("half" of a toric code)
- Qubits now associated with vertices

The Toric Code and Friends

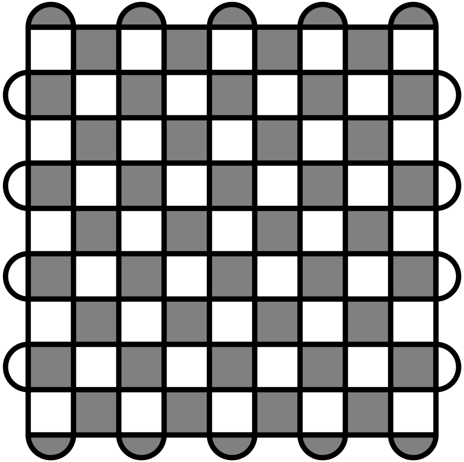
The toric code is mathematically 2D (position is described by 2 parameters), but not physically 2D (planar). To solve this, "cut" a square out:



- Developed in 2011 by Horsman et al.
- $[\![n,\, k,\, d]\!] = [\![l^2,\, 1,\, l]\!]$ ("half" of a toric code)
- Qubits now associated with vertices
- $\square$: weight-4 X check

The Toric Code and Friends

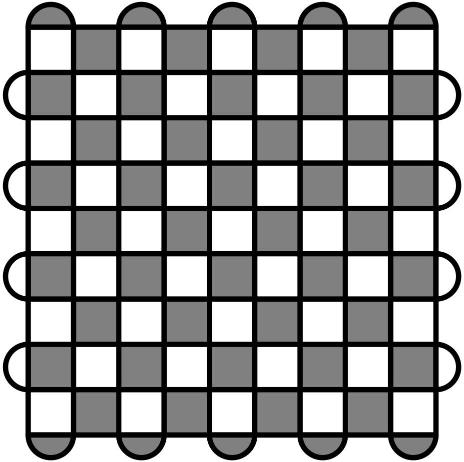
The toric code is mathematically 2D (position is described by 2 parameters), but not physically 2D (planar). To solve this, “cut” a square out:



- Developed in 2011 by Horsman et al.
- $[\![n,\, k,\, d]\!] = [\![l^2,\, 1,\, l]\!]$ (“half” of a toric code)
- Qubits now associated with vertices
-  : weight-4 X check
-  : weight-4 Z check

The Toric Code and Friends

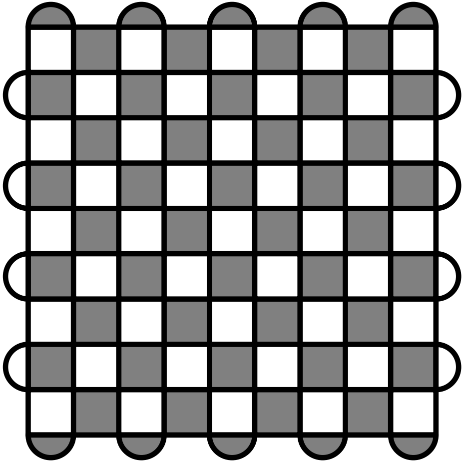
The toric code is mathematically 2D (position is described by 2 parameters), but not physically 2D (planar). To solve this, "cut" a square out:



- Developed in 2011 by Horsman et al.
- $[\![n, k, d]\!] = [\![l^2, 1, l]\!]$ ("half" of a toric code)
- Qubits now associated with vertices
- : weight-4 X check
- : weight-4 Z check
- Weight-2 boundary checks: ,

The Toric Code and Friends

The toric code is mathematically 2D (position is described by 2 parameters), but not physically 2D (planar). To solve this, “cut” a square out:



- Developed in 2011 by Horsman et al.
- $[\![n,\, k,\, d]\!] = [\![l^2,\, 1,\, l]\!]$ (“half” of a toric code)
- Qubits now associated with vertices
-  : weight-4 X check
-  : weight-4 Z check
- Weight-2 boundary checks: , 
- Decoding problem similar to toric code when approximating the threshold (theorists usually use toric code)

Decoding by Minimum-Weight Perfect Matching

Syndrome Structure

Syndrome Structure

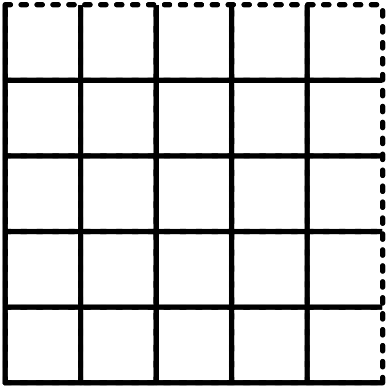
- On the torus, the decoding problem decomposes into two identical problems (consider X errors).

Syndrome Structure

- On the torus, the decoding problem decomposes into two identical problems (consider X errors).
- Syndromes appear in pairs, as with the repetition code:

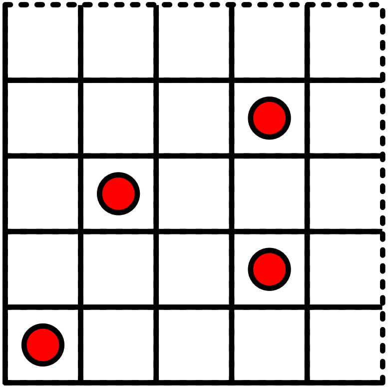
Syndrome Structure

- On the torus, the decoding problem decomposes into two identical problems (consider X errors).
- Syndromes appear in pairs, as with the repetition code:



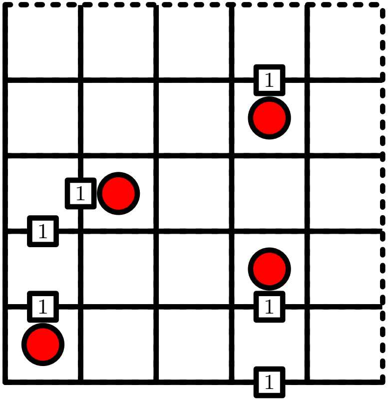
Syndrome Structure

- On the torus, the decoding problem decomposes into two identical problems (consider X errors).
- Syndromes appear in pairs, as with the repetition code:



Syndrome Structure

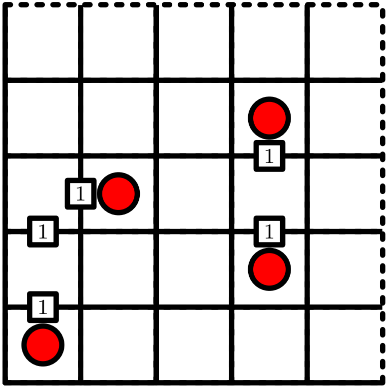
- On the torus, the decoding problem decomposes into two identical problems (consider X errors).
- Syndromes appear in pairs, as with the repetition code:



- Minimum-weight strings may wrap around the torus

Syndrome Structure

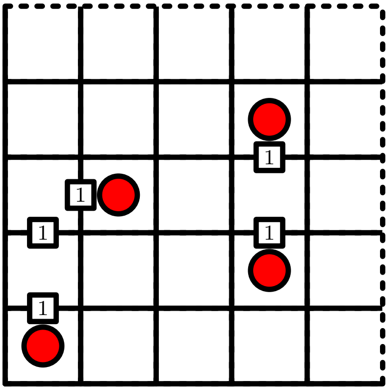
- On the torus, the decoding problem decomposes into two identical problems (consider X errors).
- Syndromes appear in pairs, as with the repetition code:



- Minimum-weight strings may wrap around the torus

Syndrome Structure

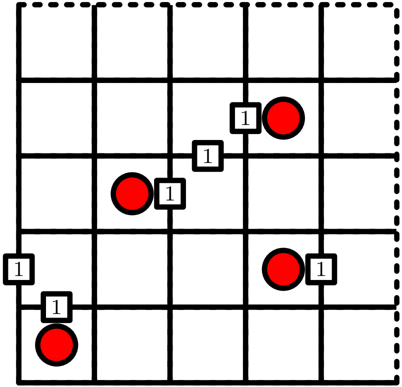
- On the torus, the decoding problem decomposes into two identical problems (consider X errors).
- Syndromes appear in pairs, as with the repetition code:



- Minimum-weight strings may wrap around the torus
- Minimum-weight strings may *end at different syndromes*

Syndrome Structure

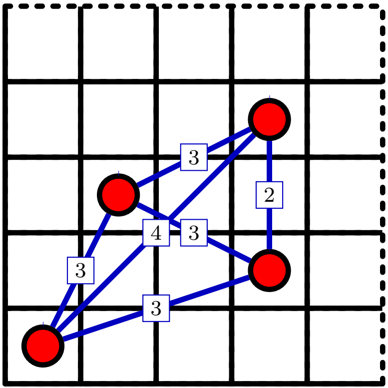
- On the torus, the decoding problem decomposes into two identical problems (consider X errors).
- Syndromes appear in pairs, as with the repetition code:



- Minimum-weight strings may wrap around the torus
- Minimum-weight strings may *end at different syndromes*

Syndrome Structure

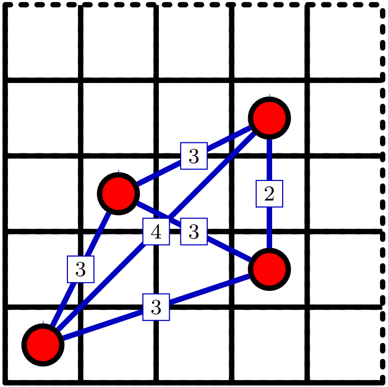
- On the torus, the decoding problem decomposes into two identical problems (consider X errors).
- Syndromes appear in pairs, as with the repetition code:



- Minimum-weight strings may wrap around the torus
- Minimum-weight strings may *end at different syndromes*
- # connections: $\mathcal{O}\left(n^2\right)$

Syndrome Structure

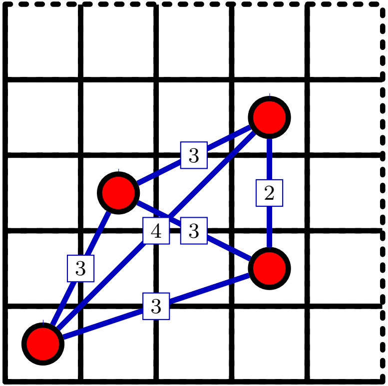
- On the torus, the decoding problem decomposes into two identical problems (consider X errors).
- Syndromes appear in pairs, as with the repetition code:



- Minimum-weight strings may wrap around the torus
- Minimum-weight strings may *end at different syndromes*
- # connections: $\mathcal{O}\left(n^2\right)$
- # potential pairings: $\mathcal{O}\left(\exp(n)\right)$

Syndrome Structure

- On the torus, the decoding problem decomposes into two identical problems (consider X errors).
- Syndromes appear in pairs, as with the repetition code:



- Minimum-weight strings may wrap around the torus
- Minimum-weight strings may *end at different syndromes*
- # connections: $\mathcal{O}\left(n^2\right)$
- # potential pairings: $\mathcal{O}\left(\exp(n)\right)$
- Required: An algorithm to find the minimum weight set of 1D strings, given syndrome minimum lengths between syndrome pairs

The Blossom Algorithm

The Blossom Algorithm

- In 1965, Jack Edmonds produced an algorithm to solve this problem.

The Blossom Algorithm

- In 1965, Jack Edmonds produced an algorithm to solve this problem.
- This algorithm relies on proofs, advanced data structures and subroutines to function.

The Blossom Algorithm

- In 1965, Jack Edmonds produced an algorithm to solve this problem.
- This algorithm relies on proofs, advanced data structures and subroutines to function.

The Blossom Algorithm

- In 1965, Jack Edmonds produced an algorithm to solve this problem.
- This algorithm relies on proofs, advanced data structures and subroutines to function.

Graph: A set of n vertices V, and m edges E consisting of pairs of vertices.

The Blossom Algorithm

- In 1965, Jack Edmonds produced an algorithm to solve this problem.
- This algorithm relies on proofs, advanced data structures and subroutines to function.

Graph: A set of n vertices V, and m edges E consisting of pairs of vertices.

Path: A sequence $v_0, e_0, v_1, e_1, \ldots v_k$ where each v_i is a node, each e_i is an edge, and the ends of e_i are v_i and v_{i+1}.

The Blossom Algorithm

- In 1965, Jack Edmonds produced an algorithm to solve this problem.
- This algorithm relies on proofs, advanced data structures and subroutines to function.

Graph: A set of n vertices V, and m edges E consisting of pairs of vertices.

Path: A sequence v_0, e_0, v_1, e_1, $\ldots v_k$ where each v_i is a node, each e_i is an edge, and the ends of e_i are v_i and v_{i+1}.

Cycle: A path for which $v_k = v_0$.

The Blossom Algorithm

- In 1965, Jack Edmonds produced an algorithm to solve this problem.
- This algorithm relies on proofs, advanced data structures and subroutines to function.

Graph: A set of n vertices V, and m edges E consisting of pairs of vertices.

Path: A sequence v_0, e_0, v_1, e_1, $\ldots v_k$ where each v_i is a node, each e_i is an edge, and the ends of e_i are v_i and v_{i+1}.

Cycle: A path for which $v_k = v_0$.

Tree: A graph with no cycles.

The Blossom Algorithm

- In 1965, Jack Edmonds produced an algorithm to solve this problem.
- This algorithm relies on proofs, advanced data structures and subroutines to function.

Graph: A set of n vertices V, and m edges E consisting of pairs of vertices.

Path: A sequence v_0, e_0, v_1, e_1, $\ldots v_k$ where each v_i is a node, each e_i is an edge, and the ends of e_i are v_i and v_{i+1}.

Cycle: A path for which $v_k = v_0$.

Tree: A graph with no cycles.

Cut: A cut $\delta(\{v_i\})$ around a set of vertices $\{v_i\}$ is the set of edges such that exactly one vertex of each edge is in $\{v_i\}$. An *odd* cut is a cut over an odd number of vertices.

The Blossom Algorithm

- In 1965, Jack Edmonds produced an algorithm to solve this problem.
- This algorithm relies on proofs, advanced data structures and subroutines to function.

Graph: A set of n vertices V, and m edges E consisting of pairs of vertices.

Path: A sequence v_0, e_0, v_1, e_1, $\ldots v_k$ where each v_i is a node, each e_i is an edge, and the ends of e_i are v_i and v_{i+1}.

Cycle: A path for which $v_k = v_0$.

Tree: A graph with no cycles.

Cut: A cut $\delta(\{v_i\})$ around a set of vertices $\{v_i\}$ is the set of edges such that exactly one vertex of each edge is in $\{v_i\}$. An *odd* cut is a cut over an odd number of vertices.

Matching: A set of edges in a graph G, such that each vertex is hit by at most one edge. A matching is *perfect* if it hits all the vertices. A vertex not hit by a matching is called *exposed*, a vertex hit by a matching is *covered*.

The Blossom Algorithm

- In 1965, Jack Edmonds produced an algorithm to solve this problem.
- This algorithm relies on proofs, advanced data structures and subroutines to function.

Graph: A set of n vertices V, and m edges E consisting of pairs of vertices.

Path: A sequence $v_0, e_0, v_1, e_1, \ldots v_k$ where each v_i is a node, each e_i is an edge, and the ends of e_i are v_i and v_{i+1}.

Cycle: A path for which $v_k = v_0$.

Tree: A graph with no cycles.

Cut: A cut $\delta(\{v_i\})$ around a set of vertices $\{v_i\}$ is the set of edges such that exactly one vertex of each edge is in $\{v_i\}$. An *odd* cut is a cut over an odd number of vertices.

Matching: A set of edges in a graph G, such that each vertex is hit by at most one edge. A matching is *perfect* if it hits all the vertices. A vertex not hit by a matching is called *exposed*, a vertex hit by a matching is *covered*.

Symmetric Difference: The symmetric difference of two edge sets A and B is $A \Delta B$, the set of edges that is in A or B, but not both.

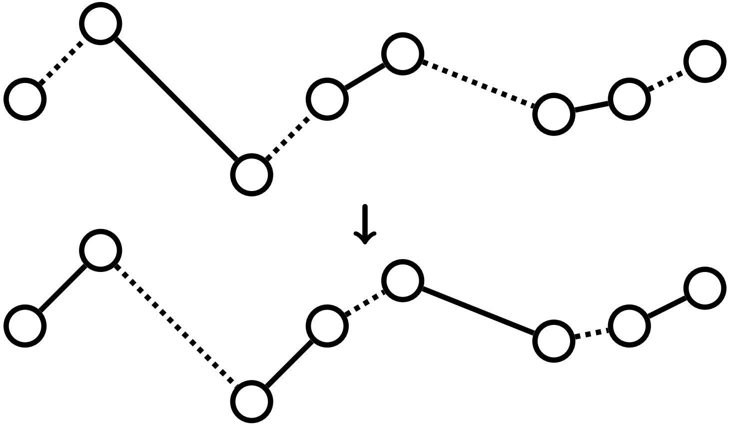
Advanced Definitions

Advanced Definitions

Alternating Path: A path with edges alternately in and out of a matching. Such a path is *augmenting* if both its end vertices are exposed.

Advanced Definitions

Alternating Path: A path with edges alternately in and out of a matching. Such a path is *augmenting* if both its end vertices are exposed.



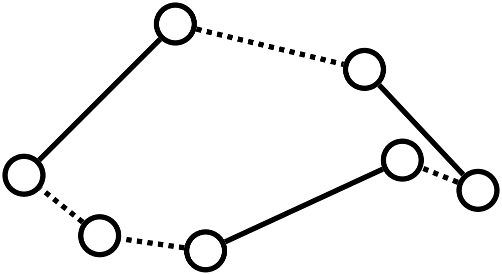
Advanced Definitions

Advanced Definitions

Blossom: an odd-length alternating cycle.

Advanced Definitions

Blossom: an odd-length alternating cycle.



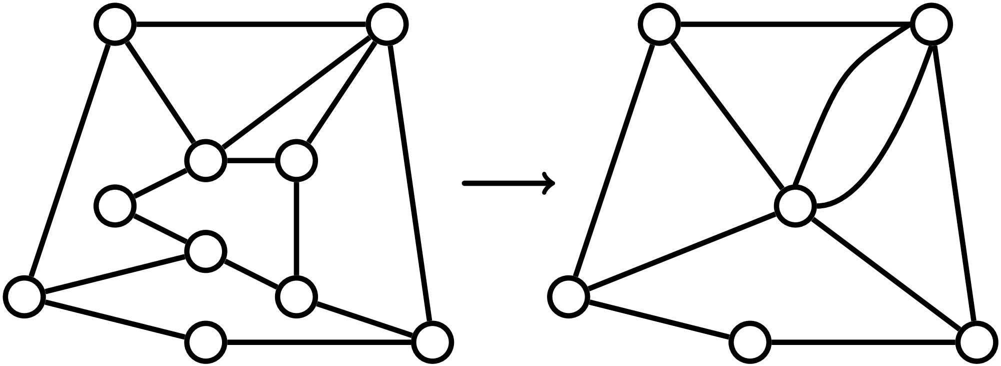
Advanced Definitions

Advanced Definitions

Derived Graph: A graph G' obtained by *contracting* an odd cycle, producing a new vertex.

Advanced Definitions

Derived Graph: A graph G' obtained by *contracting* an odd cycle, producing a new vertex.



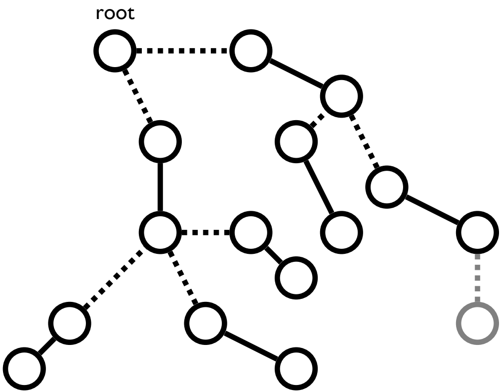
Advanced Definitions

Advanced Definitions

Alternating Tree: A tree with edges alternately in and out of a matching, depending on distance from the *root*. Vertices which are an odd distance from the root r of the tree T are in the set $A(T)$, vertices with an even distance from the root are in $B(T)$.

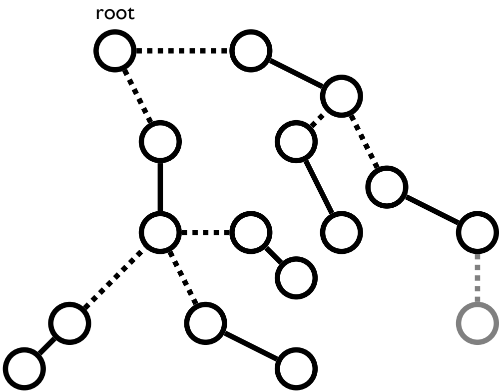
Advanced Definitions

Alternating Tree: A tree with edges alternately in and out of a matching, depending on distance from the *root*. Vertices which are an odd distance from the root r of the tree T are in the set $A(T)$, vertices with an even distance from the root are in $B(T)$.



Advanced Definitions

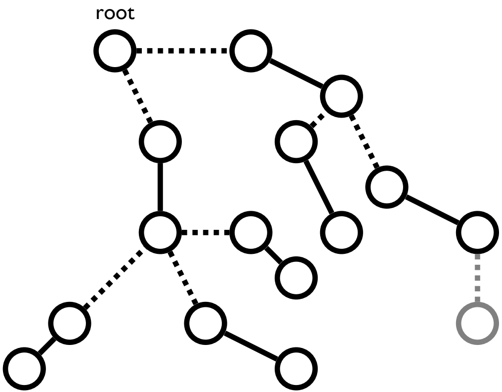
Alternating Tree: A tree with edges alternately in and out of a matching, depending on distance from the *root*. Vertices which are an odd distance from the root r of the tree T are in the set $A(T)$, vertices with an even distance from the root are in $B(T)$.



- if an exposed neighbour (grey) exists, the matching can be extended.

Advanced Definitions

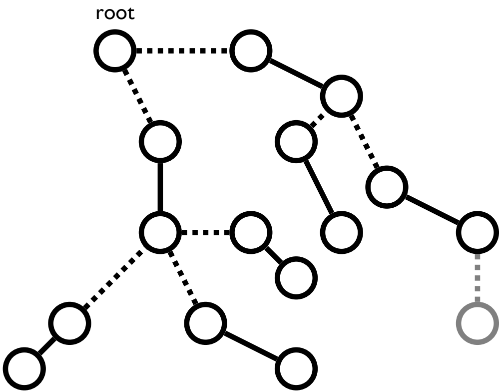
Alternating Tree: A tree with edges alternately in and out of a matching, depending on distance from the *root*. Vertices which are an odd distance from the root r of the tree T are in the set $A(T)$, vertices with an even distance from the root are in $B(T)$.



- if an exposed neighbour (grey) exists, the matching can be extended.
- if no such neighbour exists, the tree is *frustrated*, and no perfect matching exists.

Advanced Definitions

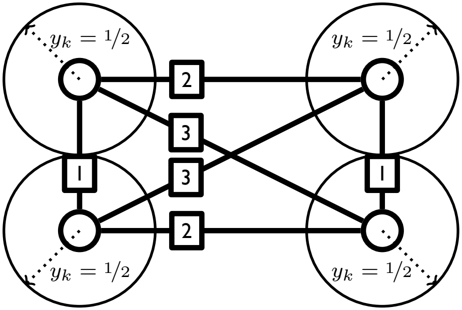
Alternating Tree: A tree with edges alternately in and out of a matching, depending on distance from the *root*. Vertices which are an odd distance from the root r of the tree T are in the set $A(T)$, vertices with an even distance from the root are in $B(T)$.

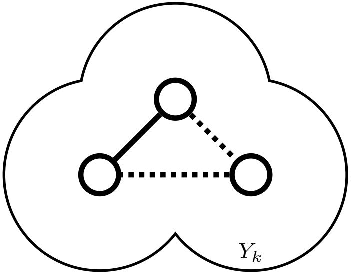


- if an exposed neighbour (grey) exists, the matching can be extended.
- if no such neighbour exists, the tree is *frustrated*, and no perfect matching exists.
- For un-weighted perfect matchings: repeatedly grow these trees until a perfect matching or frustrated tree is found

The Blossom Algorithm

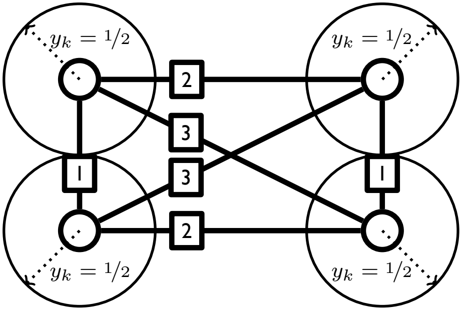
There exists an edge set on a weighted graph such that a perfect matching on that set is also minimum-weight, called the *tight edges*.

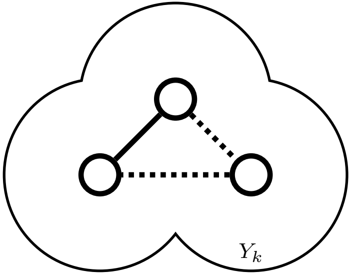




The Blossom Algorithm

There exists an edge set on a weighted graph such that a perfect matching on that set is also minimum-weight, called the *tight edges*.

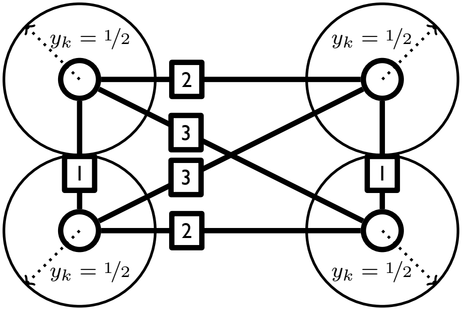


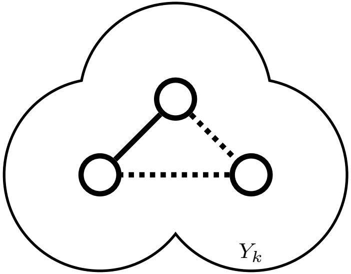


- Blossom alternates between changing the tight edge set (by increasing/decreasing $\vec{y}$) and searching for augmenting paths.

The Blossom Algorithm

There exists an edge set on a weighted graph such that a perfect matching on that set is also minimum-weight, called the *tight edges*.

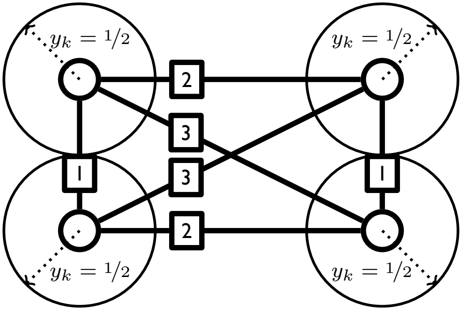


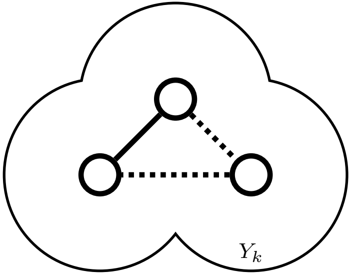


- Blossom alternates between changing the tight edge set (by increasing/decreasing $\vec{y}$) and searching for augmenting paths.
- This takes time $\mathcal{O}(n^{2.5-4})$ for a complete graph with n vertices, depending on whether the edge weights are integers, and how trees are stored.

The Blossom Algorithm

There exists an edge set on a weighted graph such that a perfect matching on that set is also minimum-weight, called the *tight edges*.



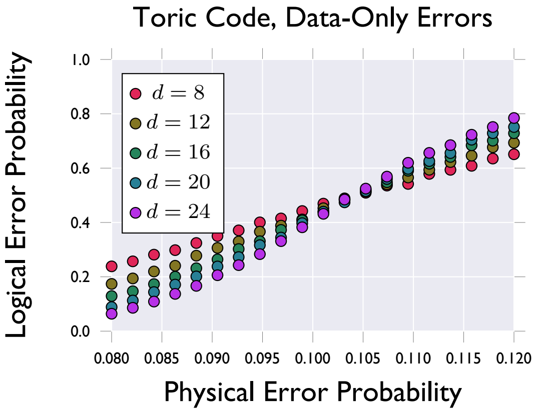


- Blossom alternates between changing the tight edge set (by increasing/decreasing $\vec{y}$) and searching for augmenting paths.
- This takes time $\mathcal{O}(n^{2.5-4})$ for a complete graph with n vertices, depending on whether the edge weights are integers, and how trees are stored.

Why do we tolerate this complexity?

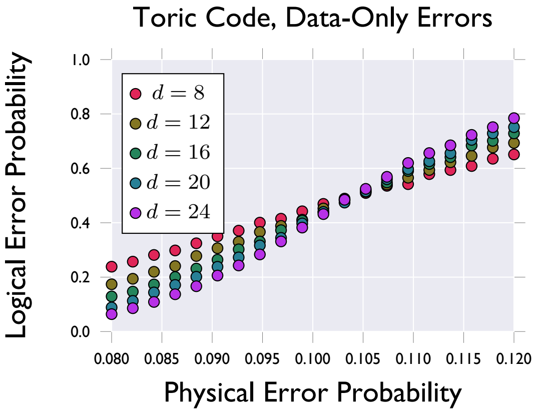
Performance

When subjected to independent bit- and phase-flip errors, a threshold of $\sim 10.3\%$ can be obtained with the toric code:



Performance

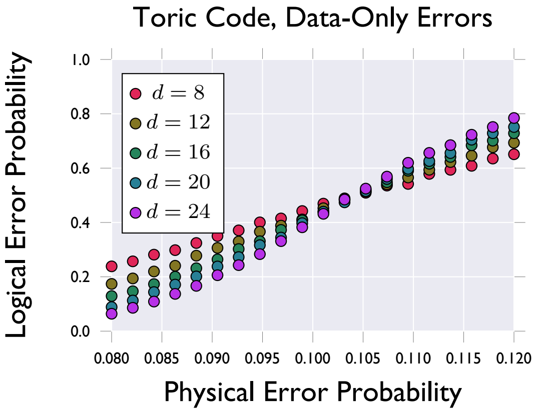
When subjected to independent bit- and phase-flip errors, a threshold of $\sim 10.3\%$ can be obtained with the toric code:



- Maximum possible threshold: $\sim 10.9\%$

Performance

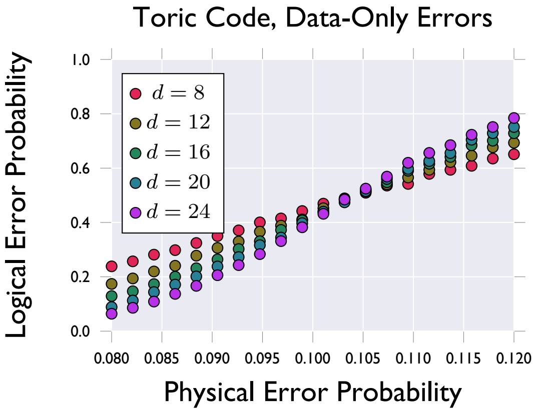
When subjected to independent bit- and phase-flip errors, a threshold of $\sim 10.3\%$ can be obtained with the toric code:



- Maximum possible threshold: $\sim 10.9\%$
- Tight, poly-time approximation first achieved in 2002.

Performance

When subjected to independent bit- and phase-flip errors, a threshold of $\sim 10.3\%$ can be obtained with the toric code:



- Maximum possible threshold: $\sim 10.9\%$
- Tight, poly-time approximation first achieved in 2002.
- Does not take into account measurement errors.

Fault-Tolerance

If measurements have error probability q, and qubits are X/Z-flipped with probability p:

Fault-Tolerance

If measurements have error probability q, and qubits are X/Z-flipped with probability p:

- we measure the stabilisers repeatedly, layering 2D sheets of measurements into a 3D object.

Fault-Tolerance

If measurements have error probability q, and qubits are X/Z-flipped with probability p:

- we measure the stabilisers repeatedly, layering 2D sheets of measurements into a 3D object.
- *syndrome changes* appear at the ends of 1D chains

Fault-Tolerance

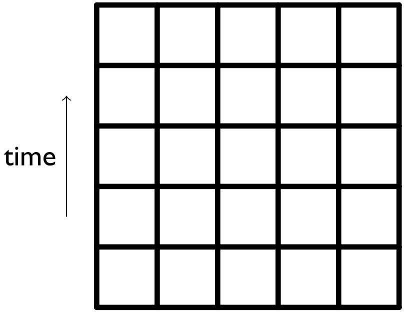
If measurements have error probability q, and qubits are X/Z-flipped with probability p:

- we measure the stabilisers repeatedly, layering 2D sheets of measurements into a 3D object.
- *syndrome changes* appear at the ends of 1D chains
- the "length" of an error chain consisting of d data and m measurement errors can be determined from a likelihood model:

Fault-Tolerance

If measurements have error probability q, and qubits are X/Z-flipped with probability p:

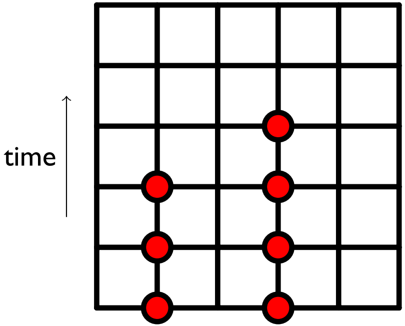
- we measure the stabilisers repeatedly, layering 2D sheets of measurements into a 3D object.
- *syndrome changes* appear at the ends of 1D chains
- the "length" of an error chain consisting of d data and m measurement errors can be determined from a likelihood model:



Fault-Tolerance

If measurements have error probability q, and qubits are X/Z-flipped with probability p:

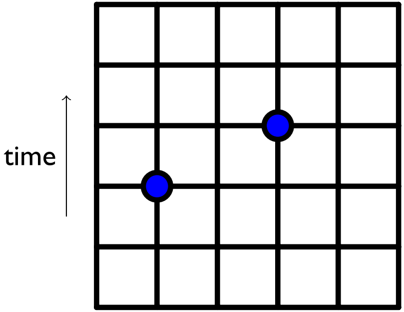
- we measure the stabilisers repeatedly, layering 2D sheets of measurements into a 3D object.
- *syndrome changes* appear at the ends of 1D chains
- the "length" of an error chain consisting of d data and m measurement errors can be determined from a likelihood model:



Fault-Tolerance

If measurements have error probability q, and qubits are X/Z-flipped with probability p:

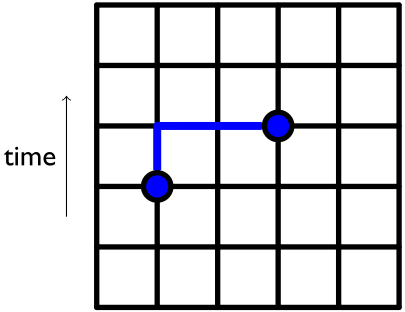
- we measure the stabilisers repeatedly, layering 2D sheets of measurements into a 3D object.
- *syndrome changes* appear at the ends of 1D chains
- the "length" of an error chain consisting of d data and m measurement errors can be determined from a likelihood model:



Fault-Tolerance

If measurements have error probability q, and qubits are X/Z-flipped with probability p:

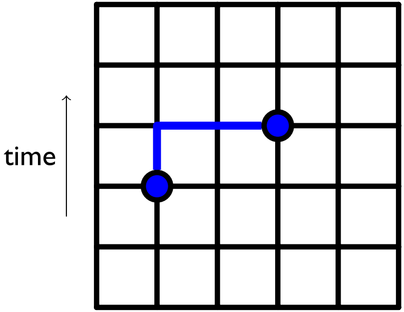
- we measure the stabilisers repeatedly, layering 2D sheets of measurements into a 3D object.
- *syndrome changes* appear at the ends of 1D chains
- the "length" of an error chain consisting of d data and m measurement errors can be determined from a likelihood model:



Fault-Tolerance

If measurements have error probability q, and qubits are X/Z-flipped with probability p:

- we measure the stabilisers repeatedly, layering 2D sheets of measurements into a 3D object.
- *syndrome changes* appear at the ends of 1D chains
- the "length" of an error chain consisting of d data and m measurement errors can be determined from a likelihood model:



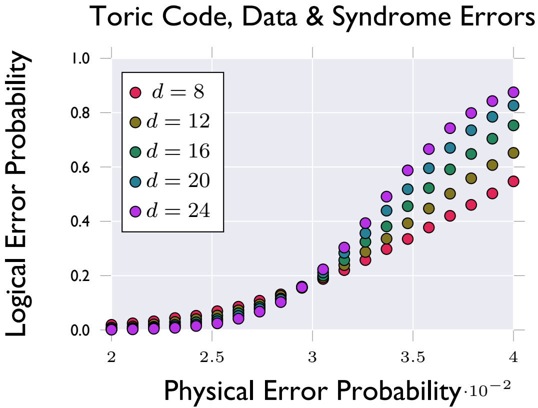
$$p_{\text{chain}} = p^d (1-p)^{n-d} \times q^m (1-q)^{n_s - m}$$

$$\sim \left(\frac{p}{1-p}\right)^d \left(\frac{q}{1-q}\right)^m$$

$$\log(p_{\text{chain}}) = d \log\left(\frac{p}{1-p}\right) + m \log\left(\frac{q}{1-q}\right)$$

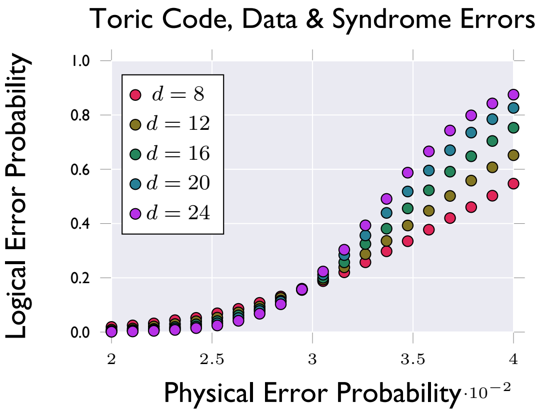
Performance

This toy model results in a reduced threshold of $\sim 2.9\%$:



Performance

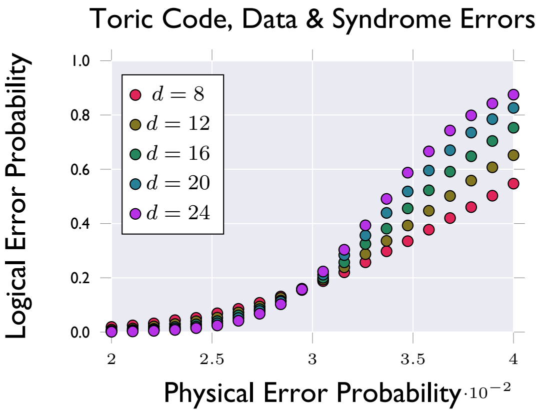
This toy model results in a reduced threshold of $\sim 2.9\%$:



- More advanced likelihood models based on circuit noise can result in thresholds $0.6\% - 1.4\%$.

Performance

This toy model results in a reduced threshold of $\sim 2.9\%$:



- More advanced likelihood models based on circuit noise can result in thresholds $0.6\% - 1.4\%$.
- This result implies that real fault tolerance is possible if classical computation is "free".

Alternate Codes and Decoders

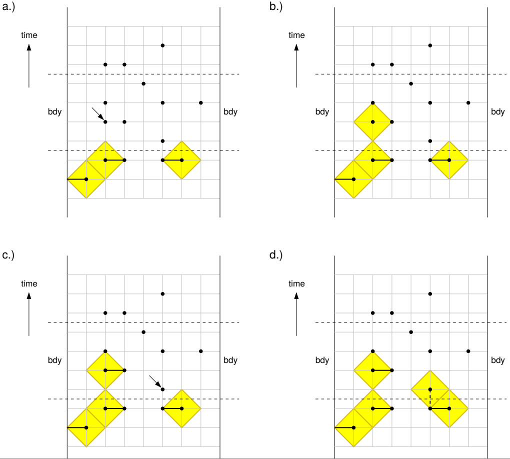
The Backlog Problem

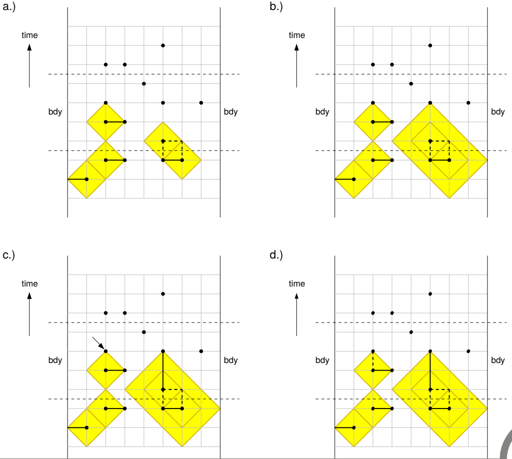
If errors take time $\mathcal{O}(n^k)$ to correct, and errors are time-dependent, then *error strength increases with* n.

The Backlog Problem

If errors take time $\mathcal{O}(n^k)$ to correct, and errors are time-dependent, then *error strength increases with* n.

We must interleave decoding with faulty measurement.





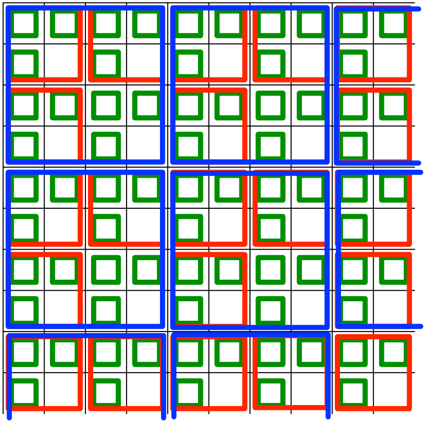
Locality

For an easier hardware implementation, it would help if the algorithm were local/parallelizable.

Locality

For an easier hardware implementation, it would help if the algorithm were local/parallelizable.

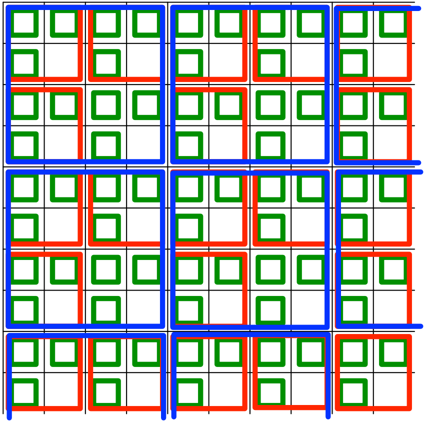
Renormalization Group decoders use a local lookup table and modified belief propagation to achieve a parallelizable algorithm requiring $\log(n)$ steps:



Locality

For an easier hardware implementation, it would help if the algorithm were local/parallelizable.

Renormalization Group decoders use a local lookup table and modified belief propagation to achieve a parallelizable algorithm requiring $\log(n)$ steps:

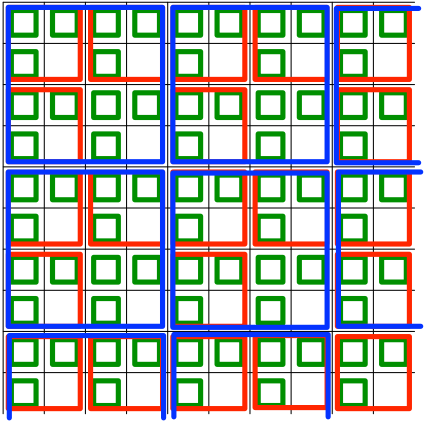


- data-only thresholds: $\sim 8\%$

Locality

For an easier hardware implementation, it would help if the algorithm were local/parallelizable.

Renormalization Group decoders use a local lookup table and modified belief propagation to achieve a parallelizable algorithm requiring $\log(n)$ steps:

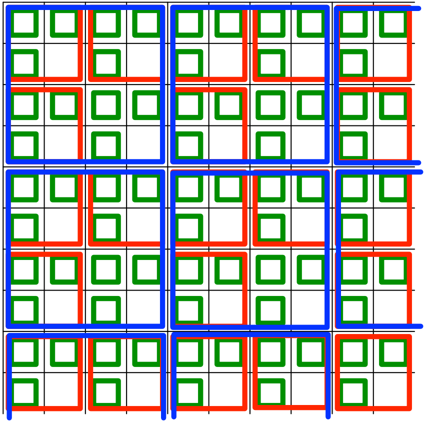


- data-only thresholds: $\sim 8\%$
- data & syndrome thresholds: $\sim 1.9\%$

Locality

For an easier hardware implementation, it would help if the algorithm were local/parallelizable.

Renormalization Group decoders use a local lookup table and modified belief propagation to achieve a parallelizable algorithm requiring $\log(n)$ steps:

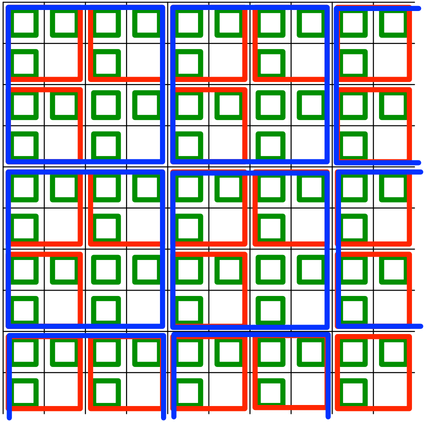


- data-only thresholds: $\sim 8\%$
- data & syndrome thresholds: $\sim 1.9\%$
- circuit-based thresholds: unknown

Locality

For an easier hardware implementation, it would help if the algorithm were local/parallelizable.

Renormalization Group decoders use a local lookup table and modified belief propagation to achieve a parallelizable algorithm requiring $\log(n)$ steps:

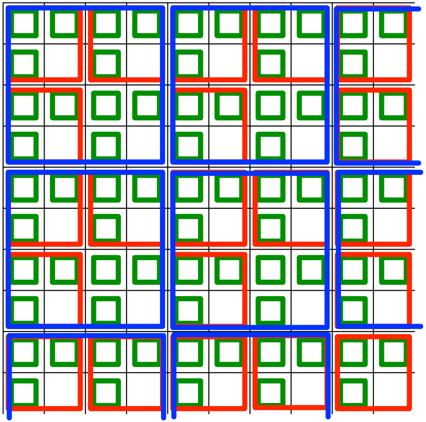


- data-only thresholds: $\sim 8\%$
- data & syndrome thresholds: $\sim 1.9\%$
- circuit-based thresholds: unknown
- effect of accurate error modelling: unknown

Locality

For an easier hardware implementation, it would help if the algorithm were local/parallelizable.

Renormalization Group decoders use a local lookup table and modified belief propagation to achieve a parallelizable algorithm requiring $\log(n)$ steps:



- data-only thresholds: $\sim 8\%$
- data & syndrome thresholds: $\sim 1.9\%$
- circuit-based thresholds: unknown
- effect of accurate error modelling: unknown
- ability to interleave: unknown

Questions?