Quantum Inf Process
DOI 10.1007/s11128-013-0683-9

# Adiabatic quantum programming: minor embedding with hard faults

**Christine Klymko · Blair D. Sullivan · Travis S. Humble**

Received: 8 November 2012 / Accepted: 4 November 2013
© Springer Science+Business Media New York 2013

**Abstract** Adiabatic quantum programming defines the time-dependent mapping of a quantum algorithm into an underlying hardware or logical fabric. An essential step is embedding problem-specific information into the quantum logical fabric. We present algorithms for embedding arbitrary instances of the adiabatic quantum optimization algorithm into a square lattice of specialized unit cells. These methods extend with fabric growth while scaling linearly in time and quadratically in footprint. We also provide methods for handling hard faults in the logical fabric without invoking approximations to the original problem and illustrate their versatility through numerical studies of embeddability versus fault rates in square lattices of complete bipartite unit cells. The studies show that these algorithms are more resilient to faulty fabrics than naive embedding approaches, a feature which should prove useful in benchmarking the adiabatic quantum optimization algorithm on existing faulty hardware.

**Keywords** Quantum computing · Adiabatic quantum optimization · Graph embedding · Fault-tolerant computing

C. Klymko
Department of Mathematics and Computer Science, Emory University,
400 Dowman Drive, Atlanta, GA 30322, USA
e-mail: cklymko@emory.edu

B. D. Sullivan · T. S. Humble (✉)
Oak Ridge National Laboratory, Quantum Computing Institute, 1 Bethel Valley Road,
Oak Ridge, TN 37831, USA
e-mail: humblets@ornl.gov

*Present Address:*
B. D. Sullivan
Department of Computer Science, North Carolina State University, Raleigh, NC 27695, USA
e-mail: blair_sullivan@ncsu.edu

Published online: 20 November 2013

## 1 Introduction

Adiabatic quantum optimization (AQO) applies the principles of quantum computing to solve unconstrained optimization problems. In particular, the AQO algorithm interpolates between two quantum logical Hamiltonians in order to adiabatically transform an initial quantum state to a computational solution state [16]. This specialized application of adiabatic quantum computing has been used to solve a variety of problems including, for example, instances of satisfiability (SAT) [15] and exact cover [16], finding Ramsey numbers [18], classifying binary images [21], training classifiers for machine learning [23] and finding the lowest free-energy configuration in folded proteins [22].

Benchmarking the efficiency of the AQO algorithm is currently of significant interest in quantum computer science. Whereas some studies of optimization problems have uncovered runtimes that scale polynomially in problem size, others suggest worst-case exponential behavior, or even trapping in local minima [2]. Interpreting these analyses are difficult, in part, because of the manner in which instance-specific information alters the implementation of the algorithm, i.e., programming. As emphasized by others [2,11,12,15], choices made in programming the AQO algorithm greatly impact its runtime and, consequently, the observed scaling behavior.

Benchmarking adiabatic algorithms is further complicated when the design of the logical Hamiltonians is constrained. Because the AQO algorithm uses a reduction of the classical optimization problem to a quantum logical representation, i.e., a Hamiltonian, any constraints placed on this underlying logical fabric can only limit performance. Understanding the impact of the logical fabric is especially pertinent since existing AQO hardware supports a specific topology over a relatively modest number of qubits [14,19].

Adiabatic quantum programming has been described previously as requiring two steps: parameter setting [9] and minor embedding [10]. Minor embedding, in particular, uses explicit information about the logical fabric as well as the problem to generate the implementation of the AQO algorithm. Choi has demonstrated how an arbitrary input graph can be minor-embedded within one type of highly regular fabric, a square lattice of $K_{4,4}$s, complete bipartite graphs with eight vertices. In the current paper, we also present algorithms for minor embedding into additional logical fabrics, namely, square lattices of $K_{c,c}$ with $c \geq 1$. We present an attempt at a brute force embedding via graph isomorphism in maximal minors (henceforth called maximal minor embedding) as well as an algorithm for complete-graph embedding. We compare these algorithms in terms of their complexity as well as the scaling of the embedding result. The second step in adiabatic quantum programming, parameter setting, has also been previously addressed [9]. Those results, which show how the Hamiltonian parameters of the embedded program are defined by the original problem, are immediately applicable to the current context.

Notwithstanding algorithms for the unit-cell lattice, an open question in adiabatic quantum programming is how to handle fabrics containing randomized hard faults. Hard faults refer to defects in the logical fabric that compromise its regularity. As their locations are random, the embedding algorithm must handle a variety of target graphs. In the current paper, we present methods for minor embedding that use heuristics

to adapt to random faults in the logical fabric (hardware). We analyze algorithmic performance in terms of the maximum embeddable complete graph obtained using numerical simulations. These studies quantify the impact of faults on the required logical footprint and provide performance expectations for hard fault-tolerant adiabatic quantum programming.

The paper is organized as follows: Sect. 2 defines the role of minor embedding in AQO; Sect. 3 briefly reviews previous work; Sect. 4 defines nomenclature and presents implications of treewidth on graph embeddability; Sect. 5 recounts properties of the unit-cell lattice; Sect. 6 determines treewidth for hardware graphs $F$; Sect. 7 presents embedding of a complete graph in $F$; Sect. 8 presents two algorithms for embedding with hard faults and numerical tests of these algorithms using randomized fault placement; finally, Sect. 9 presents our conclusions.

## 2 Adiabatic quantum optimization

The AQO algorithm is based on the reduction of an unconstrained optimization problem to a quantum logical Hamiltonian that is diagonal in the computational basis [16]. The reduction most naturally begins in terms of binary variables that can then be mapped to the qubits of a logical Hamiltonian $H_F$. For AQO, the problem Hamiltonian takes the form

$$H_F = \sum_{i \in V_F} \alpha_i Z_i + \sum_{(i,j) \in E_F} \beta_{i,j} Z_i Z_j, \tag{1}$$

where $\alpha_i$ is the weight on the $i$-th qubit, $\beta_{i,j}$ is the coupling between qubits $i$ and $j$, and the sets $V_F$ and $E_F$ denote the vertices and edges of the graph $F$ describing the logical fabric; a more formal definition of the hardware graph is found in Sect. 4. In this setting, the Pauli $Z_i$ operator defines the computational basis for the $i$-th qubit.

The 2-local form of Eq. (1) restricts the optimization problems that can be mapped directly into $H_F$. Specifically, any binary optimization problem can be recast to have at most quadratic interactions, i.e., as a quadratic unconstrained binary optimization (QUBO) problem. This reduction can be done by, e.g., substituting the product of two variables with a new one and adding a corresponding penalty term [8]. The AQO program input is therefore defined as the QUBO problem

$$\arg\min_{\mathbf{x} \in \mathbf{B}^n} \mathbf{x}^{\mathbf{T}} \mathbf{P} \mathbf{x}, \tag{2}$$

where $\mathbf{x}$ is a vector of $n$ binary variables and $\mathbf{P}$ is an $n$-by-$n$ symmetric real-valued matrix.

In programming the QUBO problem, the interactions between variables represented by $\mathbf{P}$ must be mapped into the quantum logical fabric. We interpret $\mathbf{P}$ as a weighted version of the adjacency matrix of an input (problem) graph $P$ describing these dependencies. Hence, programming the AQO algorithm requires embedding $P$ in the graph $F$ representing the logical fabric. We defer the formal definition of minor embedding

to Sect. 4, but it suffices to say that this yields a graph $F^* = (V^*, E^*)$ contained within the logical fabric, over which a Hamiltonian $H_{F^*}$ is defined as

$$H_{F^*} = \sum_{i \in V^*} \alpha_i^* Z_i + \sum_{(i,j) \in E^*} \beta_{i,j}^* Z_i Z_j \tag{3}$$

with $\alpha_i^*$ and $\beta_{i,j}^*$ the corresponding weights and couplings. Setting these parameters requires both the matrix **P** and the embedding into the logical fabric specified by $F^*$ [9].

The program for the AQO algorithm is then expressed by the time-dependent Hamiltonian

$$H(t; T) = A(t; T) H_I + B(t; T) H_{F^*}, \tag{4}$$

where $A(t)$ and $B(t)$ control the time-dependent interpolation between an initial Hamiltonian $H_I$ and the final embedded problem Hamiltonian $H_{F^*}$. The time $T$ represents the quantum annealing time of the algorithm, such that $H(T) = H_{F^*}$. Running the program $H(t)$ requires initializing the quantum register state to be a ground state of $H(0)$. This is followed by annealing to the time $T$ after which the register is measured. Provided the conditions of the adiabatic theorem are met, the state of the register at $T$ will be a ground state of $H_{F^*}$ and a solution to the QUBO problem. In order to meet these conditions, $T$ must scale inversely with the minimum spectral gap of $H(t)$ [16]. The gap, of course, depends on the programmed implementation and we may expect that the choice of embedding plays a role in satisfying this condition.

## 3 Previous related work

In [10], Choi described a hardware graph for minor embedding a large clique, $K_n$, in a limited number of qubits. This layout was called TRIAD. Choi also discussed using the TRIAD scheme on a 128 qubit hardware made up of a $4 \times 4$ grid of $K_{4,4}$ cells to achieve the embedding of $K_{17}$. We note the figure in [10] corresponding to this description in that paper only embeds a $K_{16}$, but it is possible to embed $K_{17}$ using the TRIAD scheme. It is also worth noting that while the text claimed a requirement of only 6 physical vertices for each logical qubit, this is not achievable with the given hardware (and is not realized in the example given). Our work results in the same embedding for $K_{17}$ on the $4 \times 4$ grid, but then extends the algorithm to work on a large family of related logical fabrics. We also provide a straightforward algorithm for extending an embedding from an $n \times n$ grid to an $(n+1) \times (n+1)$ grid of $K_{c,c}$ cells. This paper additionally determines the treewidth of the family of fabric graphs, which enables better screening of QUBOs for feasible embeddability. Perhaps most importantly, prior work did not consider the case of faulty fabric, which we address with two algorithms and a set of simulations to demonstrate performance.

## 4 Graph minors and tree decomposition

A *graph* $G = (V, E)$ is a set of vertices $V$ and a set of edges $E$ formed by unordered pairs of vertices. In this paper, all graphs are finite, simple (no loops or multiple

edges), and undirected. A graph $H = (W, F)$ is a *subgraph* of $G$, denoted $H \subseteq G$, if $W \subseteq V$ and $F \subseteq E$.

A *path* in $G = (V, E)$ is a sequence of vertices $v_1, v_2, \ldots, v_k$ such that for $1 \leq i < k$, $(v_i, v_{i+1}) \in E$. A *cycle* is a path where $v_1 = v_k$. If there are no repeated vertices, the path (cycle) is a *simple path (cycle)*.

A graph is *connected* if there is a path from $u$ to $v$ for every pair of distinct vertices $u, v$ in $V$. A *tree* is a connected graph which does not contain any simple cycles as subgraphs. We say a graph $H$ is a *subtree* of $G$ if $H \subseteq G$ and $H$ is a tree.

Programming adiabatic quantum computing hardware to solve a specific problem requires embedding a *problem graph* $P = (V_P, E_P)$ representing the QUBO problem (elements of $V_P$ correspond to QUBO variables and $E_P = \{(i, j) | \mathbf{P}_{i,j} \neq 0\}$) into a *hardware graph* $F = (V_F, E_F)$ whose vertices representing the qubits and edges are determined by couplings in the logical fabric. In some cases, this can be done in a one-to-one manner through subgraph embedding.

**Definition 1** A *subgraph embedding* of $P$ into $F$ is a mapping $f : V_P \to V_F$ such that:

1. each vertex in $V_P$ is mapped to a unique vertex in $V_F$.
2. if $(u, v) \in E_P$, then $(f(u), f(v)) \in E_F$.

Note that if such an $f$ exists, $P$ is a *subgraph* of $F$, $P \subseteq F$.

However, due to design constraints on the underlying logical fabric, in order to consider a large class of QUBO problems, $P$ will need to be embedded into $F$ as a minor.

**Definition 2** A *minor embedding* of $P$ in $F$ is defined by a mapping $\phi : V_P \to V_F$ such that:

1. each vertex $v$ in $V_P$ is mapped to the vertex set of a connected subtree $T_v$ of $F$.
2. if $(u, v) \in E_P$, then there exist $i_u, i_v \in V_F$ such that $i_u \in T_u$, $i_v \in T_v$, and $(i_u, i_v) \in E_F$.

If such a mapping $\phi$ exists, then $P$ is *minor-embeddable* in $F$ or $P$ is a *minor* of $F$, written $P \leq_m F$.

Equivalently, $P$ is minor-embeddable in $F$ if $P$ can be obtained from $F$ by a series of edge deletions and contractions (see [13] for more information on graph minors). Note that every subgraph embedding is also a minor embedding (since $f(v)$ is a single node subtree of $F$). Furthermore, the property of being a minor is transitive: $G \leq_m F$ and $P \leq_m G$ implies $P \leq_m F$.

Closely related to the idea of a graph minor is the concept of a tree decomposition, a combinatorial way of measuring how "tree-like" a graph is. Many early results on graph minors were first proved for trees [13]. Additionally, certain problems which have exponential complexity on arbitrary graphs have been shown to have polynomial complexity on graphs of bounded treewidth. More importantly, certain properties of tree decompositions, including upper bounds on treewidth (the definition of which can be found below), are closed under the taking of minors. Understanding the tree decomposition of the hardware graph gives us information about the properties of the minors the graph has and thus what sort of QUBO problems can be embedded.

**Definition 3** Given a graph $G = (V, E)$ let $T = (I, D)$ be a tree, and $\mathcal{V} = \{V_i\}_{i \in I}$ be a family of vertex sets (also called *bags*) with $V_i \subseteq V$ indexed by the elements of $I$. The pair $(T, \mathcal{V})$ forms a *tree decomposition* of $G$ if the following hold:

1. $V = \cup_{i \in I} V_i$.
2. if $(u, v) \in E$, then there exists $i \in I$ such that $\{u, v\} \subseteq V_i$.
3. for $i_1, i_2, i_3 \in I$, if $i_3$ lies on the path in $T$ between $i_1$ and $i_2$, then $V_{i_1} \cap V_{i_2} \subseteq V_{i_3}$. Equivalently, for any vertex $v \in V$, $\{i : v \in V_i\}$ forms a connected subtree of $T$.

To avoid confusion, the elements of $V$ are referred to as the *vertices* of $G$ and the elements of $I$ as the *nodes* of $T$. The *width* of a tree decomposition $(T, \mathcal{V})$ is given by $\max_{i \in I}\{|V_i| - 1\}$. The *treewidth* $\tau(G)$ of a graph $G$ is the minimum width over all tree decompositions of $G$. Note that the width of any tree decomposition of $G$ gives an upper bound on $\tau(G)$. The following lemmas are well-known in graph theory and are useful for using treewidth to analyze the quantum hardware graphs described in Sect. 5.

**Lemma 1** *If $H$ is a minor of $G$ (i.e., $H$ is minor-embeddable in $G$), then $\tau(H) \leq \tau(G)$.*

Thus, given the treewidth of a logical fabric $F$, it is possible to automatically narrow down the class of QUBO problems for which it may be possible to find an embedding. The treewidth of several classic families of graphs is known exactly:

**Lemma 2** *Let $K_n$ be the complete graph on $n$ vertices and $K_{n,n}$ the complete bipartite graph on $2n$ vertices.*

1. $\tau(K_n) = n - 1$.
2. $\tau(K_{n,n}) = n$.
3. *The treewidth of an $n \times m$ 2-D planar grid is given by* $\min\{m, n\}$.

For more information on tree decomposition and graph minors (including the proofs of the above lemmas), see [5], chapter 12 of [13], and [17].

## 5 Description of hardware graph

In this section, we review the hardware graph that has been the basis for several proposed or demonstrated experimental studies [4,14,18,22]. The building blocks of this graph are 8-qubit unit cells whose internal couplings form $K_{4,4}$ [19]. Unit cells are tiled together with each qubit on the left half of a $K_{4,4}$ connected to its image in the cells directly above and below, and each qubit on the right half of the $K_{4,4}$ connected to its image in the cells directly to the left and right. A representation of the graph formed by sixteen cells is shown in Fig. 1. Note that due to the way the qubits are physically connected [19], when there is a failure, it will be the failure of a qubit and not an individual coupler. In terms of the hardware graph, this means vertices (and all their adjacent edges) will fail, not individual edges.

In our analysis, we consider extensions of the unit-cell design to include an increase in the number of qubits forming a cell. We also parameterize the hardware fabric to allow for expanding the grid of unit cells. In general, our results are applicable in the

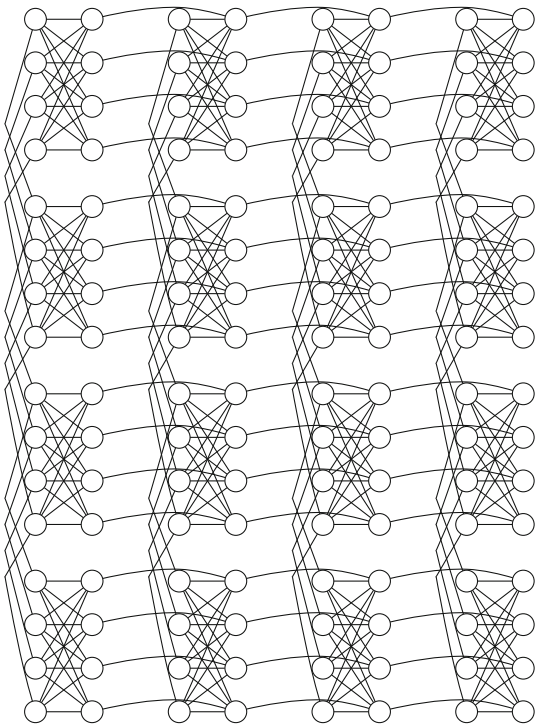

**Fig. 1** A $4 \times 4$ array of $K_{4,4}$ unit cells coupled as in the hardware graph from [19]

setting where cells consist of $2c$ qubits forming a $K_{c,c}$ and are attached to form an $m \times m$ grid in the same manner as described above. We denote a hardware graph of this form as $F(m, c)$. For example, the hardware graph shown in Fig. 1 corresponds to $F(4, 4)$.

For ease of reference, we define a labeling on $V_{F(m,c)}$. First, we number a single cell: the vertices on the left half of the $K_{c,c}$ as $1, 2, \ldots, c$ from top to bottom, and the vertices on the right half of the $K_{c,c}$ as $c + 1, c + 2, \ldots, 2c$, again from top to bottom. See Fig. 5b for an example of this numbering in a $K_{4,4}$ cell. Each vertex in $V_{F(m,c)}$ is then given a label of the form $v^d_{a,b}$ where $(a, b)$ is the (row, column) position of the cell containing the vertex in the $m \times m$ grid—with cell (1,1) in the upper left corner—and $d$ corresponds to the position of the vertex inside the individual cell, as described above.

## 6 Treewidth of the hardware graph

As seen in Lemma 1, if the treewidth of the hardware graph is known, it can be used to a priori rule out the possibility of embedding certain classes of QUBOs.

However, in general, determining the treewidth of an arbitrary graph $G$ is NP-complete [6,7]. In [6], Bodlaender describes a linear time algorithm to determine whether a graph has treewidth at most $k$, for a given fixed $k$. However, the constants for the algorithm are extremely large (and grow exponentially with $k$), making it

impractical for most graphs, including the hardware graphs of interest here. Amir describes a polynomial-time algorithm which finds a factor-$\mathcal{O}(\log(\tau(G)))$ approximation of the treewidth of a graph $G$ [3]; however, we have tighter bounds for the treewidth of the hardware graph $F(m, c)$, as presented below.

**Theorem 1** *Let $F(m, c)$ be a hardware graph made up of an $m \times m$ array of cells, attached as described in Sect. 5, where each cell contains $2c$ qubits connected to form a $K_{c,c}$. Then,*

1. *the treewidth of a single cell ($m = 1$) is $c$.*
2. *$cm \leq \tau(F(m, c)) \leq cm + c - 1$ for $m \geq 2$.*

**Corollary 1** *Any QUBO problem $P$ of treewidth $\tau(P) \geq cm + c$ is not minor embeddable in the hardware graph $F(m, c)$.*

**Corollary 2** *Any QUBO problem which contains a $K_{cm+c+1}$ (either as a subgraph or as a minor) cannot be embedded into the hardware graph $F(m, c)$.*

**Corollary 3** *Any QUBO problem which contains a $c(m + 1) \times c(m + 1)$ grid (either as a subgraph or as a minor) cannot be embedded into the hardware graph $F(m, c)$.*

Thus, even though the hardware graph described in Corollary 2 contains $2\,cm^2$ qubits, a $K_{cm+c+1}$, which would need only $c(m + 1) + 1$ logical qubits (if they were all coupled in the fabric), is shown to not be embeddable, due to its treewidth.

*Proof of Theorem 1* The proof of (1) follows directly from Lemma 2. Furthermore, the lower bound of (2) follows from using the algorithm in Sect. 7.2 to embed a $K_{cm+1}$ into $G$, since by Lemma 2, $\tau(K_{cm+1}) = cm$, and Lemma 1 implies

$$cm = \tau(K_{cm+1}) \leq \tau(F(m, c)).$$

The upper bound is slightly harder to compute. The proof consists of constructing a tree decomposition of $F(m, c)$ with width $cm + c - 1$. Then, since the treewidth of $F(m, c)$ is the minimum width over all tree decompositions, $cm + c - 1$ is an upper bound.

To form a tree decomposition $(T, \mathcal{V})$ of width $cm + c - 1$, we start with $V_1 = \{v_{1,1}^{c+1}, v_{1,1}^{c+2}, \ldots, v_{1,1}^{2c}, v_{1,2}^{c+1}, v_{1,2}^{c+2}, \ldots, v_{1,2}^{2c}, v_{1,m}^{c+1}, v_{1,m}^{c+2}, \ldots, v_{1,m}^{2c}, v_{1,1}^{1}, v_{1,1}^{2}, \ldots, v_{1,1}^{c}\}$. That is, $V_1$ contains the right half of every cell in the first column of the grid plus the left half of the (1, 1) cell.

The idea is to create all other bags of the decomposition by sequentially dropping/adding the left/right halves of individual cells. Each new bag will be formed by removing one of these sets of four vertices from an existing bag, and adding a (different) set of four—specifically one that is not yet contained in any existing bag. The large amount of overlap between the bags is to ensure that the third requirement of Definition 3 is satisfied.

The bags $V_2, \ldots, V_m$ of the decomposition are formed by dropping the right sides of cells in the first column and picking up the left sides, one-by-one. That is, $V_i$ contains the right half of cells $i + 1$ through $m$ in the first column, the left half

of cells 1 through $i-1$, and all of cell $i$. More formally, for $2 \leq i \leq m$, $V_i = \{v_{i,1}^{c+1}, \ldots, v_{i,1}^{2c}, \ldots, v_{m,1}^{c+1}, \ldots, v_{m,1}^{2c}, \ldots, v_{1,1}^{1}, \ldots, v_{1,1}^{c}, \ldots, v_{i,1}^{1}, \ldots, v_{i,1}^{c}\}$. In the tree being formed, $T$, the first $m$ nodes form a path.

The next $m$ bags are formed by (again) starting with $V_1$ but adding the right hand sides of the cells in the second column: for $V_{m+1}$ we drop the remaining four vertices in the left half of the first column and add the top four in the right half of the second; for $V_{m+i}$ with $2 \leq i \leq m$, we add $v_{2,i}^{c+1}, \ldots, v_{2,i}^{2c}$, and remove $v_{1,i-1}^{c+1}, \ldots, v_{1,i-1}^{2c}$. Bag $V_{2m+1}$ is then formed by dropping the last four vertices from the first column and adding the four left vertices of the top cell in the second column. Note that $V_{2m+1}$ is the exact same "shape" as $V_1$, only one column over. There is an edge between node 1 and node $m+1$ in $T$, then nodes $m+2$ through $2m$ continue the path.

At this point, the tree decomposition branches, with two new bags attached to $V_{2m+1}$ (analogous to $V_1$). The first is $V_{2m+2}$, which starts the branch consisting of $V_{2m+2}, \ldots, V_{3m}$, with $V_{2m+i}$ dropping $v_{2,i-1}^{c+1}, \ldots v_{2,i-1}^{2c}$ and adding $v_{2,i}^{1}, \ldots v_{2,i}^{c}$. Note this is equivalent to how $V_1, \ldots, V_m$ were created. Also attached to $V_{2m+1}$ is $V_{3m+1}$, formed by removing the four righthand vertices from the top cell and adding the top four vertices from the right half of the third column. This branch continues to form $V_{3m+2}, \ldots, V_{4m}$ analogously to $V_{m+2}, \ldots, V_{2m}$, so that $V_{4m}$ has the same shape as $V_{2m}$, only one column over.

The remainder of the tree decomposition is created starting from $V_{4m+1}$ (formed analogously to $V_{2m+1}$), until each column has been covered with a set of bags which are formed like $V_1, \ldots, V_m$. This generates a total of $2m^2 - m$ bags, each containing exactly $cm + c$ vertices of $F(m, c)$. A small example of the beginning of this process on a $3 \times 3$ grid of $K_{4,4}$ cells can be seen in Fig. 2. The tree associated with this tree decomposition can be found in Fig. 3b, along with the trees associated with the tree decompositions of the $2 \times 2$ and the $4 \times 4$ grids in Fig. 3a, c, respectively. Note that these three trees have the same general shape, with only the length of their branches changing, dependent on $m$.

We now show why $(T, \mathcal{V})$ satisfies the three properties of a tree decomposition from Definition 3:

1. every vertex of $F(m, c)$ is in at least one bag.
2. every edge is contained in at least one bag. This can be verified by noticing that every cell is fully contained in exactly one bag, covering all edges within $K_{c,c}$. Additionally, for each column, there is a bag containing all of the left side vertices of the cells in the column, and thus, all the vertical intercell edges in the column. Finally, as the bags move from one column to the next, the right halves of each pair of horizontally adjacent cells are contained in a unique bag, thus covering all horizontal intercell edges.
3. Let $v$ be an arbitrary vertex in $F(m, c)$ and let $V_k$ be the lowest index bag in which $v$ appears. Then, as we walk along $T$ starting at node $k$ and traveling in the direction of increasing node labels, once $v$ is dropped from the bag (on any branch) it is never picked up again. Thus, the nodes of $T$ which correspond to bags that contain $v$ form a connected subtree of $T$.

Since $(T, \mathcal{V})$ is a tree decomposition of $F(m, c)$ where every bag contains $cm + c$ vertices, it has width $cm + c - 1$, so

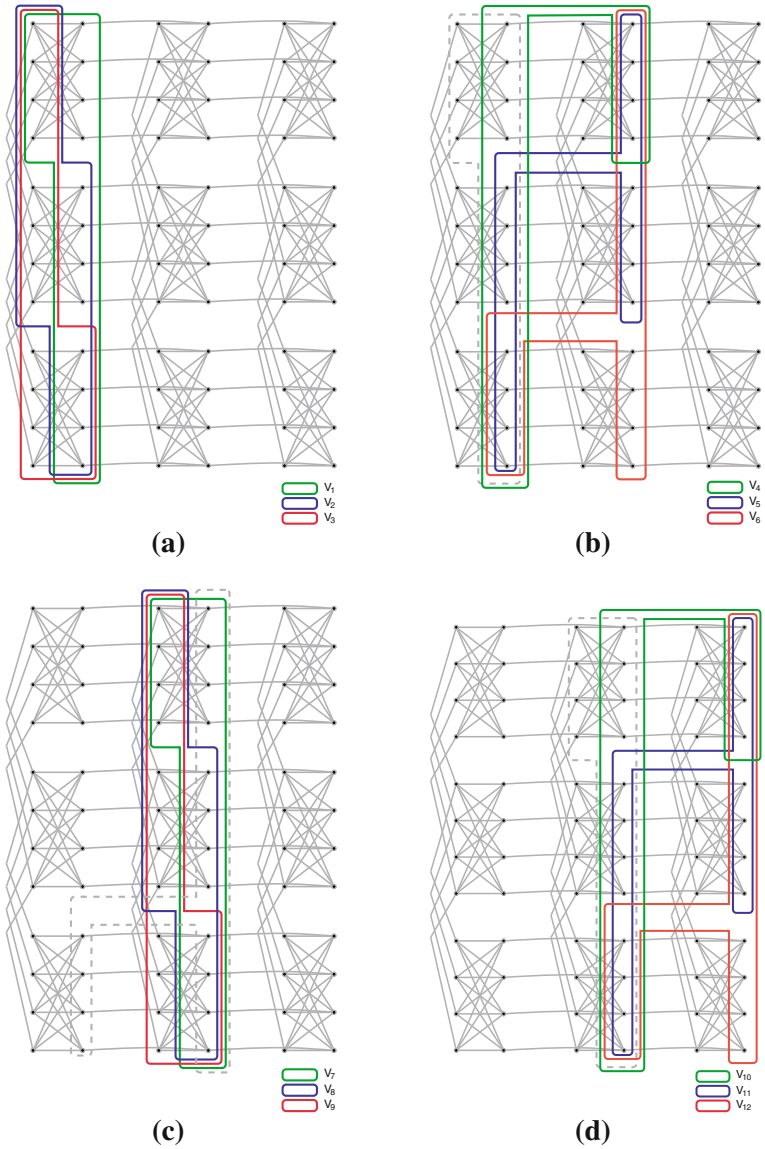


**Fig. 2** The first 12 bags of the tree decomposition of a 3 × 3 cell described in the proof of Theorem 1. The last 3 bags (not shown) have the same layout as the bags in (**c**), moved to the *third column* of cells

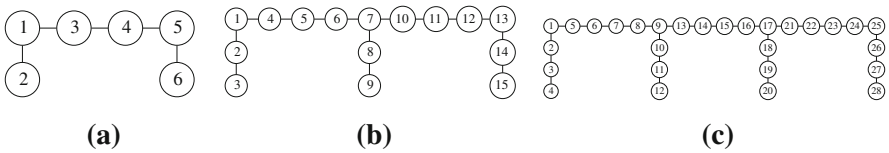


**Fig. 3** Trees of the tree decompositions of the hardware graph with a grid of size, **a** 2 × 2, **b** 3 × 3, and **c** 4 × 4 which satisfy the upper bound from Theorem 1

$$\tau(F(m, c)) \leq cm + c - 1.$$

While these bounds are not tight for all choices of $c$, they are best possible when $c = 1$, as $cm = cm + c - 1 = m$.

Determining bounds on the treewidth of the hardware graph is useful because it allows us to automatically dismiss the possibility of embedding certain classes of QUBO problems, members of which we might otherwise have spent considerable time attempting to embed.

If lower bounds on the treewidth of the QUBO problems are known, these can be combined with the bounds on the treewidth of the hardware graph to rule out even more QUBO problems. There are many graph-theoretic methods for finding lower bounds on treewidth, which use various graph properties including smallest degree, second smallest degree, girth, and spectral radius. Applying lower bounds to classes of QUBO problems is beyond the scope of this paper, but an overview of common lower-bound algorithms can be found in [7].

## 7 Embedding into the hardware graph

In general, determining whether an arbitrary graph $H$ can be minor-embedded into an arbitrary fabric $F$ is NP-complete. The best-known general algorithms assume a fixed input graph $H$ [1], which is the opposite of the situation in the quantum programming problem. Additionally, although there are polynomial- time recognition algorithms for the existence of an embedding, they do not produce the embedding and, in all cases, the hidden constants are prohibitively large [6,24]. Algorithms which allow $H$ to vary, in addition to $F$, are no longer polynomial [1,25] or are limited to specific classes of graphs which do not include the hardware graphs described in Sect. 5 [20].

### 7.1 Maximal minor embedding

Given a fabric $F$ on $n$ vertices, the method for finding and embedding every possible minor-embeddable problem graph $P$ involves solving an NP-complete problem. First, all the minors of $F$ must be found and, second, we must determine whether $P$ is a subgraph of any of them. The first step can be done when fabric is defined but even once all the minors are known, every new problem graph $P$ must be checked against them for subgraph containment, which is still NP-complete on arbitrary inputs.

The brute force algorithm for finding all possible minors of $F$ involves finding the maximal minors: a set of minors of $F$ such that every other minor is a subgraph of one of the maximal minors. The first maximal minor is $F$ itself. Subsequent maximal minors are found by contracting an edge in $F$ to form a minor and checking it for subgraph containment against the list of maximal minors. If it is not a subgraph of any of these, it is added to the list. Once every minor of size $n - 1$ is found (i.e., every possible edge contraction of $F$ has been tested), the process is repeated by contracting edges in these minors. The process is completed at step $k$ when no new maximal minors of size $n - k$ are found. An example of a set of maximal minors can be found in Fig. 4, which shows the four distinct maximal minors of $F(4, 4)$.

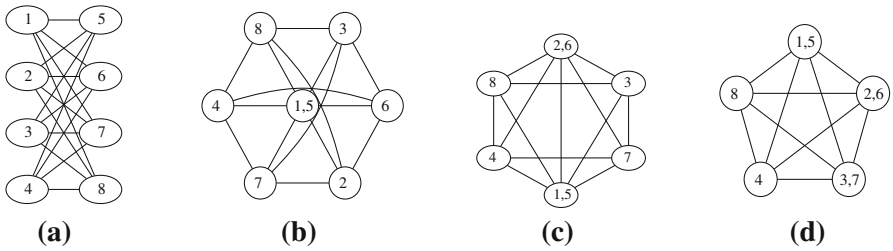


**Fig. 4** All minors of a single $K_{4,4}$ cell are a subgraph of one of these 4 graphs. **a** $K_{4,4}$. **b** 7-wheel plus 3 edges. **c** $K_6$ minus 2 edges. **d** $K_5$

Conceptually, maximal minor embedding is very straightforward. The input graph $P$ is compared with the known list of maximal minors for $F$. However, the comparison requires testing for subgraph containment, which is a combinatorial in the number of checks that must be performed. Consequently, maximal minor embedding suffers from two distinct bottlenecks, i.e., finding the maximal minor and finding the embedding. Nonetheless, this method has the benefit of finding the optimal embedding with respect to the size of the embedded problem. Because smaller embedding sizes may be expected to contribute favorably to the scaling of the energy gap, the effort required must be weighed against its advantages.

### 7.2 Algorithm to embed $K_n$

Instead of trying to find every possible minor of the hardware graph, we can find an embedding of $K_{cm+1}$. Then, for any QUBO problem of size $cm + 1$ or smaller, the embedding problem is solved. The downside of this approach is that it will fail to embed many problems that are indeed embeddable in the hardware. For example, although the graphs in Fig. 4b, c are embeddable in a $K_{4,4}$ cell, they are not embeddable in $K_5$, which is the largest $K_n$ minor in the cell. Because of this, the complete-graph embedding algorithm (as described in Sect. 7.2) requires a $2 \times 2$ array of four cells in order to find an embedding for QUBO problems corresponding to either of those graphs.

Unlike maximal minor embedding, the complete-graph embedding algorithm is computationally simple albeit at the cost of increased usage of the logical fabric. This illustrates that the two methods described here represent a tradeoff between the computational complexity of the embedding algorithm and the potential computational complexity of the quantum program as measured by the area of the computational fabric.

Given a hardware graph as described in Sect. 5, our algorithm to embed $K_{cm+1}$ as a minor in the $m \times m$ grid of $K_{c,c}$ cells is recursive in nature, and constructs the mapping $\phi$ described in Definition 2. For the sake of clarity, in the description of the algorithm, the elements of the $K_{cm+1}$ will be referred to as nodes and the elements of the hardware graph will be referred to as vertices. Let $u_1, u_2, \ldots, u_{cm+1}$ be the nodes of the $K_{cm+1}$ that we are trying to embed.

In the description of the hardware graph in Sect. 5, the vertices were given labels of the form $v^d_{a,b}$. In the Matlab-style pseudocode found below, they are numbered

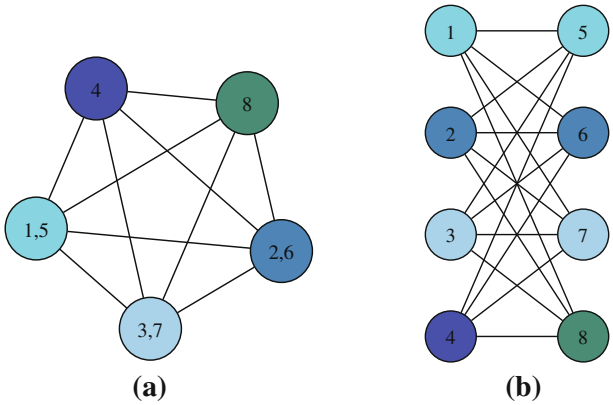


**Fig. 5** The embedding of $K_5$ into a single $K_{4,4}$ unit cell. **a** The map of each logical qubit in $K_5$ to hardware vertices. **b** The explicit embedding of $K_5$ in the unit cell

from 1 to $cm^2$. The numbering starts in the cell in the upper left corner as described in Fig. 5b and is continued across the row, then across subsequent rows. Given a node position in the form $v^d_{a,b}$, the equivalent number in the code below is $n = 2cm(a-1) + 2c(b-1) + d$. Given a node numbered $n$ in the code below, the equivalent label is given by $v^d_{a,b}$ with $a = \lceil \frac{n}{2cm} \rceil$, $b = \lceil \frac{n-2cm(a-1)}{2c} \rceil$, and $d = n \mod 2c$, with $d = 2c$ if $n \mod 2c = 0$.

```
function V = no_failure_embedding(c,m)
% This function takes an mxm hardware graph of K_{c,c} cells and
% outputs a (2m)x(cm+1) matrix V where the non-zero entries
% of V(:,i) are phi(u_i) for u_i in the embedded K_{cm+1}

V = zeros(2*m,c*m+1);
%Almost all the c*m-2 sets are formed similarly
for i = 1:c*m+1
    if i < c
        %First grid row/column, position i in cell
        r = 1; s = i;
    else if i > c+1
        %Calculate row/column of the grid
        r = ceil((i-1)/c);
        %Calculate level within cell
        s = mod((i-1),c);
        if s==0
            s=c;
    else
        continue; %these are handled below

    %fill in the horizontal members of phi(u_i)
    for j=1:m
        V(j,i)=2*c*m*(r-1)+2*c*(j-1)+c+s;
    %fill in the vertical members of phi(u_i)
    for j=1:m
        V(j+m,i)=2*c*(r-1)+2*c*m*(j-1)+s;
end
%At i=c and i=c+1, the sets differ, and have size m
for j=1:m
    V(j,c)= c + (j-1)*2*c*m;
    V(j,c+1)= j*2*c;
```

The algorithm begins by embedding the first $c+1$ nodes (forming a $K_{c+1}$) into the cell in the upper left corner of the hardware. This is done by pairing left and right vertices $c-1$ times.

That is, for $1 \leq j \leq c-1$, $\phi(u_j) = \{v_{1,1}^j, v_{1,1}^{c+j}\}$. The next two nodes are each initially mapped to a set containing a single vertex: $\phi(u_c) = \{v_{1,1}^c\}$, and $\phi(u_{c+1}) = \{v_{1,1}^{2c}\}$. See Fig. 5 for an example of embedding $K_5$ in a $K_{4,4}$ cell of 8 qubits. We provide an instance of this algorithm in a Matlab-style pseudocode for a function which produces an embedding into non-faulty $F(m, c)$ hardware.

After embedding a $K_{c+1}$ into the first cell of the hardware graph, the $m-1$ remaining steps of the algorithm extend the embedding into the subsequent row and column of the $m \times m$ grid. For each step $2 \leq i \leq m$, the embedding forms an *extendable* clique minor in the $i \times i$ grid. We say a minor is extendable if it satisfies two conditions: first for $u_j$, $1 \leq j \leq c(i-1)+1$ the set $\phi(u_j)$ is non-empty. Second, each set has at least one vertex with an edge into the next row or column.

For all nodes $u_j$, at least one vertex of $\phi(u_j)$ is connected to a cell in the next row and/or column of the grid. These vertices are added to the set $\phi(u_j)$. For nodes $u_c$ and $u_{c+1}$, one vertex is added to $\phi(u_c)$ and $\phi(u_{c+1})$ at each layer $i$. For all other nodes $u_j$, two new vertices are added to $\phi(u_j)$.

The sets $\phi(u_{c(i-1)+2})$ through $\phi(u_{c(i-1)+c+1})$ are formed by picking one of the unclaimed vertices on the right side of cell $(1, i)$. This is extended by following the edges from cell to cell along row $i$. When column $i$ is reached, one edge is taken within the cell, then edges from cell to cell are followed up along column $i$. At the end of this process, each of these sets will contain $2i$ vertices: for $1 \leq s \leq c$, $\phi(u_{c(i-1)+s+1}) = \{v_{1,i}^{c+s}, \ldots v_{i,i}^{c+s}, v_{i,1}^s, \ldots, v_{i,i}^s\}$.

This process is continued until $K_{cm+1}$ is fully embedded in the $m \times m$ grid. See Fig. 6 for an extension of a $K_{13}$ embedding in a $3 \times 3$ grid of $K_{4,4}$ cells to a $K_{17}$ embedding in a $4 \times 4$ grid of cells.

## 8 Embedding with failed qubits

The complete-graph embedding algorithm presented in Sect. 7.2 assumed that there are no failures in the hardware. However, the hardware may exhibit some percentage of failed vertices which prevent a full $K_{cm+1}$ embedding (e.g., in the case of any single qubit failure, the biggest clique embeddable is $K_{cm}$). Instead of losing a node from the $K_{cm+1}$ for each failed qubit, techniques can be employed to embed in a way that attempts to minimize the number of sets $\phi(u)$ which contain any failed qubits.

We present two algorithms below in order to handle the case of fabrics with hard faults. These approaches to embedding test the different starting points available along both diagonals of the $m \times m$ grid and then return the best possible embedding that results. Additionally, if the largest $K_n$ found is smaller than the largest possible in an $(m-1) \times (m-1)$ grid, from each corner, we drop the first row and column and reattempt the embedding. This "dropping down" procedure continues until a large enough clique is found or $(m-1)$ rows and columns have been dropped.

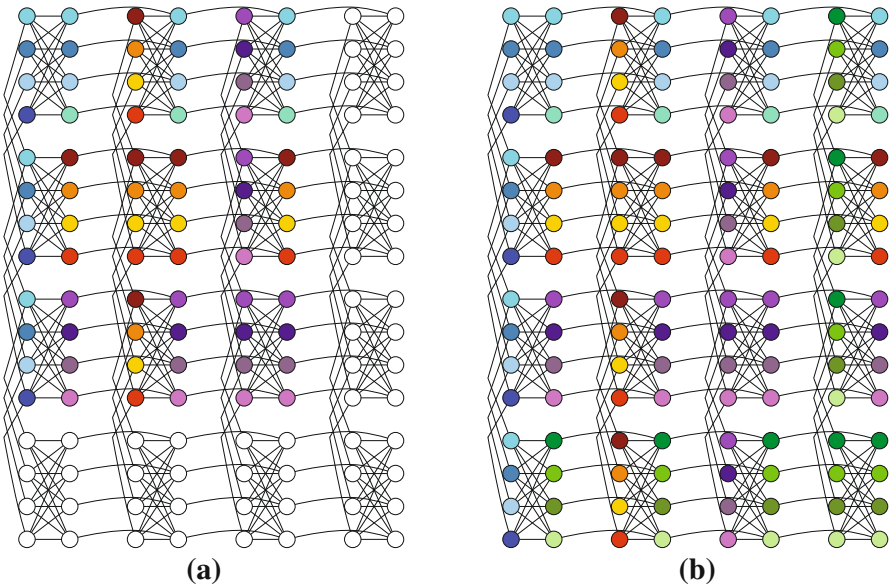


**Fig. 6** A $K_{13}$ embedding in a $3 \times 3$ grid of $K_{4,4}$ cells extended to a $K_{17}$ embedding in a $4 \times 4$ grid. Each *color* represents a single logical qubit. **a** $K_{13}$ embedded in upper $3 \times 3$ sub-grid. **b** $K_{17}$ embedded in the $4 \times 4$ grid (Color figure online)

At the same time, the grid is scanned and the largest $K_n$ embeddable in a single cell ($1 \leq n \leq c + 1$) is found. If a complete cell is found, this is $K_{c+1}$. The reported largest embeddable $K_n$ output by the algorithm is the maximum of the largest clique embeddable inside a single cell and the four cliques found from starting at the four corners.

Combining these two procedures yields a "flip and drop-down" method that we compare with the single, nominal attempt at embedding, i.e., starting in the upper left corner. In all cases, the worst performance possible is to embed a $K_1$, since we assume there is at least one working qubit in the hardware. Note details of the corner selection and drop-down methods are not shown in the pseudocode.

## 8.1 Dropping to a smaller cell-graph

Given an $m \times m$ hardware graph with cells of $K_{c,c}$, one way to deal with failed qubits is to find the largest $c_o$, $c_o \leq c$, such that there is a complete $m \times m$ grid of $K_{c_o,c_o}$'s and use the algorithm described in Sect. 7.2 to embed into this sub-grid. This will lead to an embedding of size $c_o m + 1 \leq cm + 1$. Once the $c_o$ has been determined, the embedding can be found by renumbering the vertices of the hardware graph to reflect the new cell size and running no_failure_embedding($c_o$, $m$).

### 8.2 Greedy failure algorithm

As can be seen in Fig. 6, given a perfect $m \times m$ grid of $K_{c,c}$ cells, for each node $u$ of the embedded $K_{cm+1}$ (other than nodes $u_c$ and $u_{c+1}$ started in the first cell), $\phi(u)$ contains $2m$ vertices. These consist of two sets of $m$ vertices: a connected set consisting of one vertex from the left side of each cell in the a single column in the grid and a connected set consisting of one vertex from the right side of each cell in the row of the same number. Due to the pattern in which cells are connected, within both of these sets every vertex occupies the same position in the cell it comes from.

The greedy failure algorithm works to maximize the size of the complete $K_n$ which can be embedded in the hardware graph with failed vertices, by attempting to pair up sets containing failed vertices with other sets containing failed vertices to create full nodes. These "match-ups" occur in the diagonal cells of the grid. In the case of no failures, each horizontal set (of vertices from the right halves of cells) is matched with a vertical set (of vertices from the left halves of cells) whose vertices occupy the same "height" inside a single cell. When there are errors, however, horizontal sets containing failed vertices attempt to match with vertical sets that also contain failed vertices, regardless of the "heights" at which the vertices sit inside a cell. By matching sets which contain failures, the number of complete nodes (all of which except $u_c$ and $u_{c+1}$ are made up of two sets) containing failures is reduced and, consequently, a larger embedded $K_n$ is achieved.

The Matlab-style pseudocode for a function which produces the nodes of the embedding described above and outputs the number of nodes containing no errors can be found on the next page.

### 8.3 Analysis

A comprehensive set of experiments were run to see how well the fallback and greedy algorithms from Sects. 8.1 and 8.2, respectively, performed under various conditions of vertex failure. These experiments were run using a single attempt at embedding that begins in the upper left corner of the grid of cells as well as a run using the flip and drop-down scheme described at the beginning of Sect. 8. In all cases, the hardware graph was an $m \times m$ grid of $K_{4,4}$ cells. The grid sizes tested were $m = 4, 8, 16,$ and 32. For each of these grid sizes, the algorithms were run with a percentage of failed vertices of $p = 2, 4, 5, 6, 8, 10, 15, 20,$ and 25. The failed vertices were uniformly distributed across the hardware graph. In each of the 148 cases (defined by algorithm, scheme, grid size, and failure rate), 10,000 randomized instances were run to compute statistical averages.

A comparison of the results shown by Figs. 7 and 8 illustrates that the flip and drop-down embedding scheme performs better than a single attempt at embedding from the upper left corner and that the greedy algorithm performs better than the fallback method. In both schemes, the greedy algorithm embeds a $K_n$ with $n$ approximately 85 % of the optimum value at two percent failure rate.

At fixed failure rate, the percent of the maximum embeddable $K_n$ for both algorithms decreases as the grid size $m$ grows.

```
function [V,k] = greedy_embedding(c,m,G)
% This function takes an mxm hardware graph of K_{c,c} cells and a
% list G of failed  vertices. Outputs are a (2m)x(cm+1) matrix V
% with non-zero entries of V(:,i) being phi(u_i) for u_i in the
% embedded K_{cm+1}, and k the number of failure-free sets phi(u_i).

% PAIR(s,t,cv) stores the union of F(:,s) and F(:,t) in V(:,cv)

% First, we form all of the half-sets in a matrix F
F = zeros(m,2*c*m)
for i=1:m
    for pos=1:c
        % determine columns of F to be filled
        Cnum = 2*c*(i-1)+pos
        Rnum = 2*c*(i-1)+pos+c
        for j=1:m
            % half-sets in col i of hardware graph
            F(j,Cnum)=2*c*m*(i-1)+2*c*(j-1)+pos+c
            % half-sets in row i of hardware graph
            F(j,Rnum)=2*c*m*(j-1)+2*c*(i-1)+pos
end % of for i=1:m

% Match row/column half-sets to minimize # of sets with failures.
V=zeros(2*m, c*m+1)
cv = 1; % first open column of V
k=0; % number of failure-free sets created
for i=1:m
    Fi= 2c*(i-1) % offset for column indices in F
    % Pair up sets containing failures
    for s=1:c
        if i==1 and cv==c
            break; % go create size m sets
        if F(:,Fi+s) contains a failure in G
            for t=1:c
                if F(:,Fi+c+t) contains a failure
                    PAIR(s,c+t,cv)
                    cv++
                    break
    end % of for s=1:c
    % Pair remaining half-sets greedily to form c (c-1 if i=1) total
    for s=1:c
        if ((i==1 and cv==c) or (cv==c*i+2))
            break; % create size m sets or next i
        if F(:,Fi+s) unpaired
            for t=1:c
            if F(:,Fi+c+t) unpaired
                PAIR(s,c+t,cv)
                cv++
                if V(:,cv) failure-free
                    k=k+1
    end % of for s=1:c
    % Create two size m sets in row/column 1:
    for s=1:c
        if F(:,s) unpaired
            V(:,cv) <- F(:,s)
            cv++
            if F(:,s) failure-free
                k=k+1
        if(F:,c+s) unpaired
            V(:,cv) <- F(:,c+s)
            cv++
            if F(:,c+s) failure-free
                k=k+1
    end % of for s=1:c
end % of for i=1:m
```

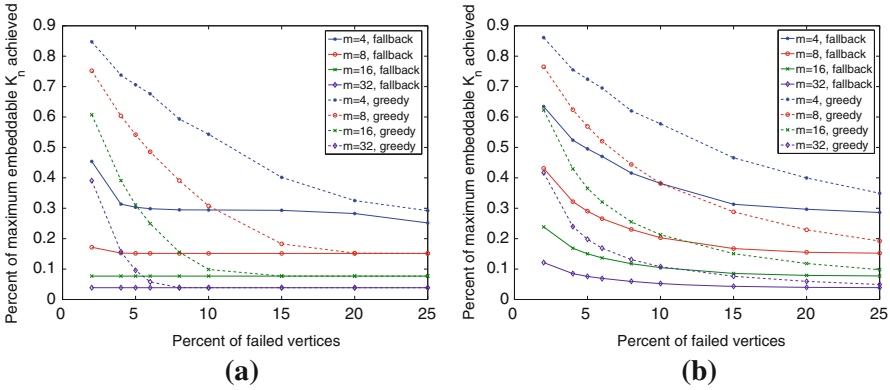


**Fig. 7** Percent of the maximum embeddable $K_n$ achieved for both the fallback and greedy embedding schemes for various percentages of failed vertices, averaged over 10,000 trials. This is calculated for both a single attempt at the embedding (*left*) and multiple attempts at the embedding, starting in all four corners and, if necessary, dropping to a smaller grid (*right*). Both methods also search for a whole cell. **a** Single attempt at embedding. **b** Maximum over flip and drop-down embeddings

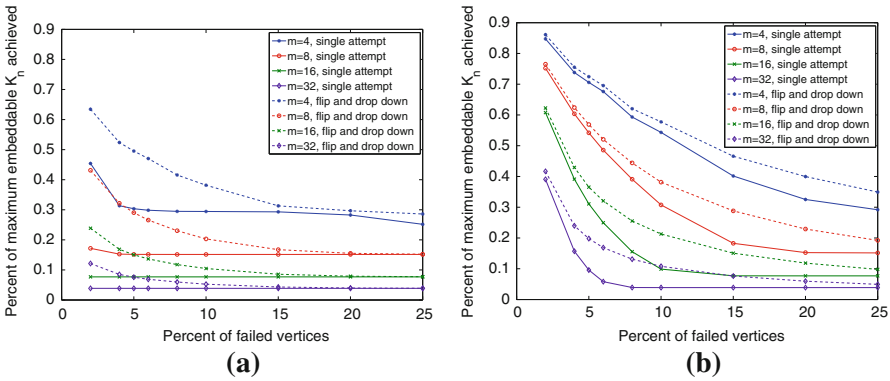


**Fig. 8** Percent of the maximum embeddable $K_n$ achieved in both a single attempt at embedding and multiple attempts (starting in each of the four corners and, if necessary, dropping down) for various percentages of failed vertices, averaged over 10,000 trials. This is calculated both for the fallback method (*left*) and for the greedy method (*right*). **a** Fallback embedding algorithm. **b** Greedy embedding algorithm

Given a 2 % failure rate, this means that any $\phi(u)$ on the $4 \times 4$ grid (with no attempt at a "smart" embedding scheme) has a 16 % chance that the set contains at least one failed vertex (and thus cannot augment the size of the $K_n$ embedded). Similarly, on the $32 \times 32$ grid, each $\phi(u)$ contains 64 vertices, and for 2 % failure having at least one failed vertex per cell is highly likely. This is due to the fact that the number of hardware vertices mapped to a single node of the $K_n$ minor increases linearly with grid size. On the $4 \times 4$ grid, each set $\phi(u)$ is made up of 8 vertices (except for 2 special cases).

At the 2 % failure rate, the greedy embedding scheme with flips and drop-downs achieves embedding of a complete graph of over 40 % the size of the maximum $K_n$ embeddable. For the worst-case scenario, and with no attempt at a "smart" embedding,

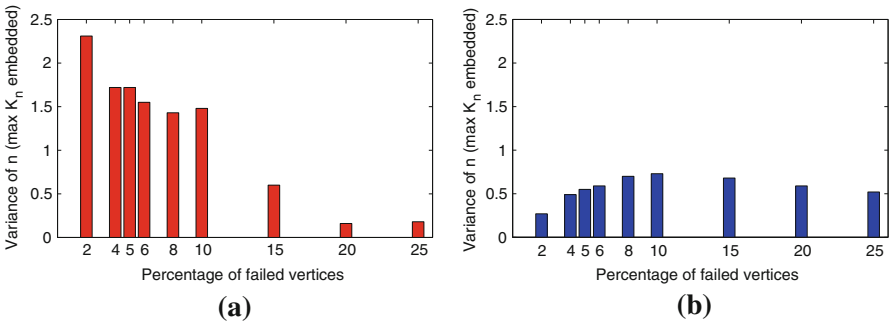


**Fig. 9** Variances of fault-tolerant embedding algorithms (with flip and drop-down) on the $4 \times 4$ grid. **a** Fallback algorithm. **b** Greedy algorithm

it would only take one failed vertex to destroy each logical qubit. Even at only a 2 % failure rate, the $32 \times 32$ grid has on average 163 failed vertices. If the algorithm did not adapt, this high failure density would completely destroy the maximum embeddable clique, which is a $K_{129}$. In the case of a 25 % failure rate, the number of failed vertices jumps to 2,048, yet the greedy failure algorithm is still able to embed a $K_6$ on average.

We have also analyzed the variances in embeddability from these experiments. In the case of a single attempt at embedding, the distribution of embeddable graphs tends to be narrower than when using the drop-down scheme. For larger grid sizes and for higher percentages of failure, the variance of the single attempt falls to zero. This is caused by the fact that the algorithm never does better than embedding a $K_5$ into a single, complete cell. However, this happens less often for the drop-down embedding scheme, yielding larger average $K_n$ with higher variances. An example of this behavior is shown in Fig. 9 for the case of $F(4, 4)$ when varying the percent failure rate. It is notable that while the variance of the fallback method is relatively large for small error rates, the greedy algorithm maintains a near constant, much lower variance across all failure rates. In Fig. 10, the distribution of achieved embeddings over 10,000 trials using the flip and drop-down scheme on $F(4, 4)$, with the percentage of failed vertices at $p = 2, 4, 8,$ and 10, is shown. The embeddings achieved by the greedy algorithm are both more clustered and larger than those achieved by the fallback algorithm. With the added evidence of panel (b) in Figs. 7 and 8, this demonstrates the greedy approach is more robust in the presence of hard faults.

## 9 Conclusions

We have presented methods for adiabatic quantum programming that embed problem-specific information into an underlying quantum logical fabric. Our methods include an embeddability analysis based on the treewidth of an $m$-by-$m$ lattice of $K_{c,c}$ unit cells, which is a generalization of existing adiabatic quantum hardware. This has provided bounds on the graphs that can be embedded in a predefined logical fabric and should be useful for guiding adiabatic quantum-programmed implementations.

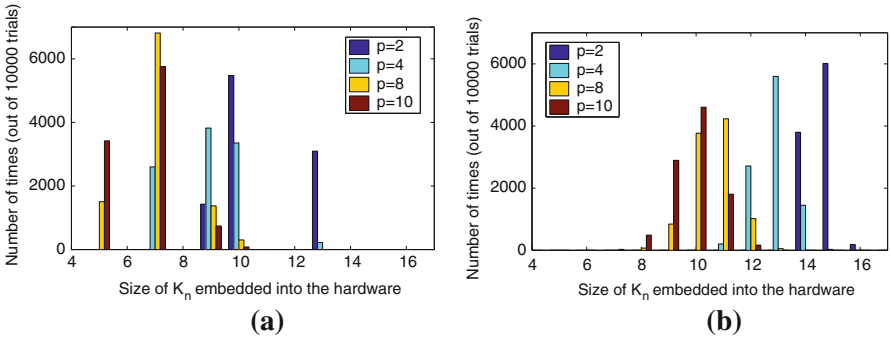


**Fig. 10** Histograms for 10,000 trials of the fallback (*left*) and greedy (*right*) embeddings with flipping and drop-down on the $4 \times 4$ grid at $p = 2, 4, 8,$ and 10 percent node failures. **a** Fallback embedding algorithm. **b** Greedy embedding algorithm

In addition, we have presented two new methods for finding an embedding of a complete graph in faulty fabric. The first method handles failures by falling back to a set of smaller available unit cells, while the second searches for embeddings that minimize the number of affected logical qubits using matching within cells on the diagonals. The latter was shown to have greater power for programming implementations of arbitrary QUBO instances. Our approach is heuristic, however, and it is unclear whether similar ideas can be applied to alternative embedding algorithms. Nevertheless, numerical studies of embeddability run against randomized failures further have shown the relative robustness of the second algorithm and the remarkably smaller variance in embeddable graphs.

In our study of embedding for adiabatic quantum programming, we have neglected any question regarding the subsequent computational complexity. The question of how a particular embedding algorithm impacts the complexity of the resulting AQO program is a point for future research. The current work, however, is expected to support uncovering the dependency of the computational complexity on both the embedding and parameter setting methods used. We believe that the embedding algorithms explored here, which provide a constructive approach to programming, will be useful for providing a consistent means of comparing the AQO algorithm across different problem sizes and hardware.

**Acknowledgments** This work was supported by the Lockheed Martin Corporation under Contract No. NFE-11-03394. The authors thank Greg Tallant (Lockheed) for technical interchange and Daniel Pack (ORNL) for help preparing Fig. 2. This manuscript has been authored by a contractor of the U.S. Government under Contract No. DE-AC05-00OR22725. Accordingly, the U.S. Government retains a non-exclusive, royalty-free license to publish or reproduce the published form of this contribution, or allow others to do so, for U.S. Government purposes.

## References

1. Adler, I., et al.: Faster parameterized algorithms for minor containment. Theor. Comput. Sci. **412**, 7018–7028 (2011)
2. Altshuler, B., Karvi, H., Roland, J.: Anderson localization makes adiabatic quantum optimization fail. Proc. Natl. Acad. Sci. USA **108**, E19–E20 (2011)

3. Amir, E.: Approximation algorithms for treewidth. Algorithmica **56**, 448–479 (2010)
4. Bian, Z., Chudak, F., Macready, W.G., Clark, L., Gaitan, F.: Experimental determination of Ramsey numbers. Phys. Rev. Lett. **111**, 130505 (2013)
5. Bodlaender, H.L.: A tourist guide through treewidth. Acta Cybern. **11**, 1–23 (1993)
6. Bodlaender, H.L.: A linear-time algorithm for finding tree decompositions of small treewidth. SIAM J. Comput. **25**, 1035–1317 (1996)
7. Bodlaender, H.L., Koster, A.M.C.A.: Treewidth Computations II. Lower Bounds, Technical Report UU-CS-2010-022. Department of Information and Computing Sciences, Utrecht University (2010)
8. Boros, E., Hammer, P.L.: Pseudo-boolean optimization. Discret. Appl. Math. **123**, 155–225 (2002)
9. Choi, V.: Minor-embedding in adiabatic quantum computation: I. The parameter setting problem. Quantum Inf. Process. **7**, 193–209 (2008)
10. Choi, V.: Minor-embedding in adiabatic quantum computation: II. Minor-universal graph design. Quantum Inf. Process. **10**, 343–353 (2011)
11. Dickson, N.G., Amin, M.H.S.: Does adiabatic quantum optimization fail for NP-complete problems? Phys. Rev. Lett. **106**, 050502 (2011)
12. Dickson, N.G., Amin, M.H.S.: Algorithmic approach to adiabatic quantum optimization. Phys. Rev. A **85**, 032303 (2012)
13. Diestel, R.: Graph Theory. Springer, Heidelberg (2005)
14. D-Wave Systems Inc., 100–4401 Still Creek Drive, Burnaby V5C 6G9, BC, Canada. http://www.dwavesys.com/
15. Farhi, E., Goldstone, J., Gutmann, S., Sipser, M.: Quantum computation by adiabatic evolution, arxiv:quant-ph/0001106 (2000)
16. Farhi, E., et al.: A quantum adiabatic evolution algorithm applied to random instances of an NP-complete problem. Science **292**, 472–476 (2001)
17. Fomin, F.V., Thilikos, D.M.: Dominating sets and local treewidth. LNCS **2832**, 221–229 (2003)
18. Gaitan, F., Clark, L.: Ramsey numbers and adiabatic quantum computing. Phys. Rev. Lett. **108**, 010501 (2012)
19. Harris, R., et al.: Experimental investigation of an eight qubit unit cell in a superconducting optimization processor. Phys. Rev. B **82**, 024511–024526 (2010)
20. Kleinberg, J., Rubinfeld, R.: Short paths in expander graphs. In: Proceedings of the 37th Annual Symposium on Foundations of Computer Science, pp. 86–95 (1996)
21. Neven, H., Rose, G., Macready, W.G.: Image recognition with an adiabatic quantum computer I. Mapping to quadratic unconstrained binary optimization, arXiv:0804.4457v1 [quant-ph] (2008)
22. Perdomo-Ortiz, A., Dickson, N., Drew-Brook, M., Rose, G., Aspuru-Guzik, A.: Finding low-energy conformations of lattice protein models by quantum annealing. Sci. Rep. **2**, 571 (2012)
23. Pudenz, K.L., Lidar, D.A.: Quantum adiabatic machine learning. Quantum Inf. Process. **12**, 2027–2070 (2012)
24. Robertson, N., Seymour, P.D.: Graph minors. XIII: the disjoint paths problem. J. Comb. Theory Ser. B **63**, 65–110 (1995)
25. Xiong, L., Dinneen, M.J.: The Feasibility and Use of a Minor Containment Algorithm, Computer Science Technical Reports 171, University of Auckland (2000)