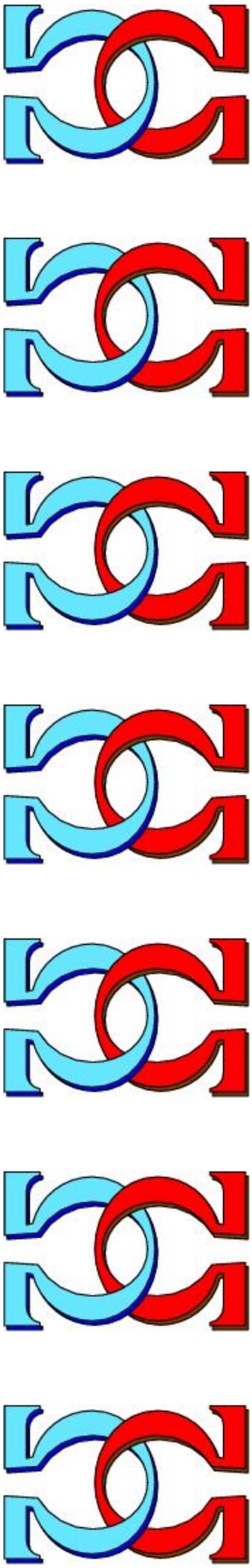

CDMTCS
Research
Report
Series

Efficient Clique Embedding with Faulty Hardware Components

Michael J. Dinneen
School of Computer Science,
University of Auckland, Auckland, New Zealand

Richard Hua
School of Computer Science,
University of Auckland, Auckland, New Zealand

CDMTCS-567
November 2022

Centre for Discrete Mathematics and
Theoretical Computer Science

Efficient Clique Embedding with Faulty Hardware Components

Michael J. Dinneen and Richard Hua

School of Computer Science, The University of Auckland, Auckland, New Zealand.

e-mail: mjd@cs.auckland.ac.nz, rwan074@aucklanduni.ac.nz

Abstract

In this paper, we investigate graph embeddings of large cliques into the existing D-Wave quantum architectures (Chimera, Pegasus) when physical qubits or couplers have faults. The motivation for pre-computing large clique embeddings allows for easier embeddings (without extensive classical computation) of arbitrary logical qubit interaction graphs (e.g. QUBO graphs) when performing quantum annealing on the restricted topologies of the existing D-Wave quantum architectures. To investigate the performance and scalability of existing hardware topology (Pegasus graph), we propose a method for simulating large hardware graphs with random faulty components and compare the performance of different embedding techniques on these graphs.

Keywords: Adiabatic Quantum Computing; Fault Tolerant and Robust Hardware; Quadratic Unconstrained Binary Optimization; Minor Embedding;

1 Introduction

Adiabatic Quantum Computing (AQC) is a relatively new model of quantum computation. Originally proposed in 2001 [18], AQC is based on the process of evolving a ground state of a Hamiltonian representing a problem to a minimum-energy solution state [18, 19]. It has been shown to be equivalent, computability-wise, to the more traditional quantum gate model [2]. Other introductory details about the application of AQC may be found in [5, 29]. Even though AQC can only simulate quantum circuit algorithms with polynomial overhead, it has attracted a lot of attention in recent years. The advantage of AQC is that a particular type of physical device that can be used for AQC known as *quantum annealers* are relatively easier to build.

D-Wave computers are produced by the Canadian company D-Wave Systems Inc. that use quantum annealing as a computation method. Their products include D-Wave One (2011) operating on a 128-qubit chipset; D-Wave Two (2013) with 512 qubits; D-Wave 2X [13] had more than 1000 qubits; D-Wave 2000Q [12] had 2048 qubits. The latest and most advanced model is D-Wave Advantage [25] which 5760 has qubits. D-Wave qubits are

loops of superconducting wire, coupled by magnetic wiring, the machine itself is supercooled close to absolute zero to get quantum effects [5, 24]. The current family of D-Wave computers can solve problems formulated in either *Ising* form or *Quadratic Unconstrained Binary Optimization* (QUBO) form, defined later.

Problem solving with quantum annealers has been shown to be challenging [1, 6, 7, 20, 23]. Although many NP-complete problems are reducible to QUBO [1, 22, 23], it has been an extremely difficult task to demonstrate any kind of practical speedup or advantage when compared with classical algorithms [1]. Many of the research that claim such advantage (see [15] for example) do not consider the entire process of problem solving with quantum annealers.

To solve a computational problem with D-Wave quantum annealers, the problem has to be converted to a QUBO instance (or the equivalent Ising instance). Then, the QUBO instance has to be 'embedded' on the host configuration of the quantum annealer. This embedding process is far from trivial and has become one of the major obstacle that is preventing quantum annealing solution from being useful in practice. After a valid embedding is found, the quantum annealer can be queried (multiple times) to obtain the solution.

This unique process of problem solving has made the analysis of the time complexities of quantum annealing solutions a difficult task. The first question that comes to mind is whether the embedding process should be considered when measuring the computation efficiency of the quantum solution. On one hand, the embeddings are typically computed by classical heuristic algorithms and one may argue that such computation has nothing to do with quantum computing in general. On the other hand, if we completely ignore such these issues, then researchers such as [15] have shown that quantum annealers can outperform classical algorithms to an unbelievable degree. However, such speedups are often artificial in nature and have no practical applications. In the case of [15], the test cases were carefully designed to be able to fit on the host configuration directly (without the need of finding an embedding which is not a realistic situation in practice).

We have proposed a novel framework that aims to address the embedding cost in [1]. The basic principle was to identify families of problems where the same embedding can be re-used for different instances of the problem so that the embedding cost can be 'spread' among the instances and hence improve the time efficiency of the AQC algorithm. We have also presented a proof-of-concept example problem to illustrated the effectiveness of the quassical (quantum-classical hybrid) computing framework. With the MWIS problem, we can only use the same embedding if the problem structure remains the same (i.e. only the vertex weights can change) which somewhat limits its uses. For example, in the communication application mentioned in [1], the quassical computing approach is only applicable if all the nodes in the network remain in the same interference range which is not often the case in practice (e.g. a WiFi network where all the devices stay stationary does not seem a probable situation). In this report, we will study a problem that is very much in the spirit of quassical computing that takes a different approach. Recall that a clique is a complete graph K_n with n vertices where every pair of vertices in the graph is connected by an edge. With any given hardware graph G, if we can compute the embedding of K_n onto G, then this embedding can be (partly) re-used for all guest graphs with order less than or equal to

n since we can just delete unnecessary vertices and edges from the clique until we have the guest graph. Given a specific hardware, its chipset structure should remain mostly the same throughout its life-cycle (e.g. some qubits may occasionally become unavailable for maintenance), so computing the largest embeddable clique on the hardware would be a very beneficial approach. However, this task is not easy in practice due to existence of faulty hardware components represented by missing vertices and edges in the hardware graph. Computing the largest (or at least near-optimal) embeddable cliques on the Chimera and Pegasus graph can be done very efficiently in polynomial-time if the physical graph has no missing components [4, 11]. The paper [4] has also proposed algorithms for finding clique embeddings if the host graph is incomplete. However, the solution qualities of these algorithms seem to be lacking in practice. As we will seen in Section 4, even a relatively small number of faulty hardware components can drastically reduce the order of the largest embeddable clique that these algorithms can find. Furthermore, there does not seem to be much research that focuses on the performance of these clique embedding techniques in a more realistic setting where faulty hardware components are taken into consideration. In this report, we will first review some of these techniques in Section 3. And then, we propose an experimental framework in which the performance of these clique embedding algorithms can be compared and present some experiment results we obtained on simulated Pegasus hardware in Section 4.

2 Preliminaries

In this section we will look at some key concepts and ideas that is necessary for what follows. We have already some of them in Section 1 and we will define them in a slightly more formal fashion here.

Quadratic Unconstrained Binary Optimization, or QUBO for short, is an NP-hard [31] mathematical optimization problem of minimizing a quadratic objective function $F : \mathbb{Z}_2^n \to \mathbb{R}$. The objective function is defined by an upper-triangular $n \times n$ matrix Q and is of the form $F(\mathbf{x}) = \mathbf{x}^T Q \mathbf{x}$, where $\mathbf{x} = (x_0, x_1, \ldots, x_{n-1})$ is a n-vector of binary (Boolean) variables. Formally, QUBO problems are of the form:

$$x^* = \min_{\mathbf{x}} \sum_{i \leq j} x_i Q_{(i,j)} x_j, \text{ where } x_i \in \mathbb{Z}_2. \tag{1}$$

In other words, the goal is to find a binary value assignment of variables $\mathbf{x} = (x_0, x_1, \ldots, x_{n-1})$ such that the value of $F(\mathbf{x})$ is minimum. We typically use x^* to denote the minimum value of $F(\mathbf{x})$ and $\mathbf{x}^* = (x_0^*, x_1^*, \cdots, x_{n-1}^*)$ to denote a value assignment of the n variables that yield x^*.

In the quantum annealing model of the QUBO problem, each x_i corresponds to a qubit while Q defines the problem Hamiltonian H_p. Specifically, the non-zero off-diagonal terms $Q_{(i,j)}$, $i < j$, correspond to couplings between qubits x_i and x_j, while the diagonal terms $Q_{(i,i)}$ are related to the local field applied to each qubit. For a given QUBO problem Q, these couplings may be conveniently represented as a graph $G_L = (V_L, E_L)$ representing the interaction between qubits, where $V_L = \{1, \ldots, n\}$ is the set of qubits and $E_L = \{\{i, j\} \mid$

$Q_{(i,j)} \neq 0,\ i < j\}$ are the edges representing the qubit interactions inside the quantum processing unit (QPU). We will refer to such a graph for a given QUBO problem as the *logical graph*, and the set of qubits the QUBO problem is represented over the *logical qubits*.

A quantum annealer has a core processor called its *quantum processing unit* (QPU). A QPU contains a certain number of *physical qubits* used for computation. The physical qubits are coupled together so that they could interact with each other during the computation. In practice, couplings between arbitrary qubits are currently infeasible since it is very difficult to control interactions between qubits that are not physically near to one another. As a result it is often not possible to directly implement an instance of the QUBO problem on the QPU since this would require an arbitrary number of couplings between arbitrary physical qubits. The couplings in a QPU are specified by a graph $G_P = (V_P, E_P)$, where V_P is the set of qubits on the device, and an edge $\{i, j\} \in E_P$ signifies that qubits i and j are physically coupled. The graph G_P is called the *physical graph*, and the qubits V_P are the *physical qubits* [8, 21].

The exact specifications of the physical graph for D-Wave devices have been modified and improved over the years. The older models, such as D-Wave 2X and D-Wave 2000Q, use the *Chimera graph* and the newer[1] D-Wave Advantage model uses the *Pegasus graph* (see [10] for details on all the chipset topologies).

Since the logical graph G_L for a QUBO problem instance Q will not, in general, be a subgraph of the physical graph G_P, the problem instance on G_L must be mapped to an equivalent one on G_P. This process involves two steps: first, G_L must be minor embedded in G_P, and secondly, the weights of the QUBO problem (i.e. the non-zero entries in Q) must be adjusted so those valid solutions on G_P are mapped to valid solutions on G_L. The second stage can be done in polynomial time and so let us focus on the first stage here.

Let $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$. A *minor embedding* of G_1 onto G_2 is a function $f : V_1 \to 2^{V_2}$ such that:

1. For all $v \in V_1$, the set of vertices v maps to under f are disjoint.
2. For all $v \in V_1$, there is a subset of edges $E' \in E_2$ such that $G' = (f(v), E')$ is connected.
3. If $\{u, v\} \in E_1$, then there exist $u', v' \in V_2$ such that $u' \in f(u)$, $v' \in f(v)$ and $\{u', v'\}$ is an edge in E_2.

The embedding stage amounts to finding a minor embedding $f : V_L \to 2^{V_P}$ of $G_L = (V_L, E_L)$ onto $G_P = (V_P, E_P)$ [8]. Typically, this involves mapping each logical qubit to a set (sometimes called 'chains' or 'blocks') of physical qubits.

The problem of finding a minor embedding is itself computationally difficult [8]. Of course, if one has sufficiently many physical qubits to embed K_n then any n-qubit logical graph can trivially be embedded into the physical graph.

[1] As of Jun 2022, D-Wave has announced a new prototype with a new topology called the Zephyr architecture.

2.1 The Chimera graph

The Chimera architecture is used by D-Wave 2X and D-Wave 2000Q. We will provide a more formal definition of the Chimera graph in this subsection. A Chimera graph, denoted by $\chi_{M,N,L}$, consists of M by N blocks of $K_{L,L}$ complete bipartite graphs. Each vertex v in a Chimera graph is connected to four vertices in the same $K_{L,L}$ unit and at most two other vertices in adjacent units. Since all D-Wave models that use the Chimera topology have $n \times n$ blocks of $K_{4,4}$, we will sometimes use χ_n to denote $\chi_{n,n,4}$ for notational convenience.

The D-Wave Ocean SDK [30] provides an indexing function of the vertices of any $\chi_{M,N,L}$. Each vertex v of $\chi_{M,N,L}$ is indexed by a 4-tuple of integers (i, j, u, k) where $0 \leq i < M$, $0 \leq j < N$, $0 \leq u < 2$ and $0 \leq k < L$. The integers i and j specify the location of the $K_{L,L}$ unit that v belongs to. The integer u denotes the which partition of the $K_{L,L}$ the vertex v is in and k indexes the vertices in the partition. What is particularly nice about this indexing is that it can be converted to a linear index by the formula $l((i, j, u, k)) = 1+2nLi+2Lj+Lu+k$. For notational convenience, we will use these two types of indexing interchangeably.

Given two vertices $a = (i, j, u, k)$ and $b = (i', j', u', k')$. There is an edge $\{a, b\} \in E$ if one of the following conditions is met:

1. $i = i'$ and $j = j'$ and $u \neq u'$.
2. $i = i' \pm 1$ and $j = j'$ and $u = u' = 0$ and $k = k'$.
3. $i = i'$ and $j = j' \pm 1$ and $u = u' = 1$ and $k = k'$.

The connections between the adjacent 'blocks' are shown in Figure 1. Specifically, each qubit is coupled with 4 other qubits in the same $K_{4,4}$ block and 2 qubits in adjacent blocks (except for qubits in blocks on the edge of the grid, which are coupled to a single other block). The vertices in Figure 1 follow the indexing scheme.

2.2 The Pegasus architecture

In 2020, D-Wave has released its latest model of quantum annealers. Using a new chipset architecture called the Pegasus graph, the D-Wave Advantage is significantly denser than all the previous model and hence should provide a big advantage in solving larger and denser problems. In this subsection, we will give a formal description of the Pegasus graph. We will follow the definition of the Pegasus graph given in [11]. A Pegasus graph is specified by a single positive integer M. Denoted by ϕ_M, the graph has $24M(M-1)$ vertices. Note that ϕ_M has $8(M-1)$ vertices used exclusively for error-correcting purposes and cannot be used for problem solving and so it effectively has $8(3M-1)(M-1)$ working vertices.

Once again, we will follow the indexing of vertices implemented in the D-Wave Ocean SDK [11, 30]. In this indexing scheme, each vertex in ϕ_M is indexed by a 4-tuple of integers (u, w, k, z) where $0 \leq u \leq 1$, $0 \leq w < M$, $0 \leq k \leq 11$ and $0 \leq z \leq M-2$. The linear index formula is $l((u, w, k, z)) = z + (M-1)(k + 12(w + Mu))$. The edges are slightly harder to define since the Pegasus graph is a lot denser than the Chimera graph. We will follow the method used in [11].

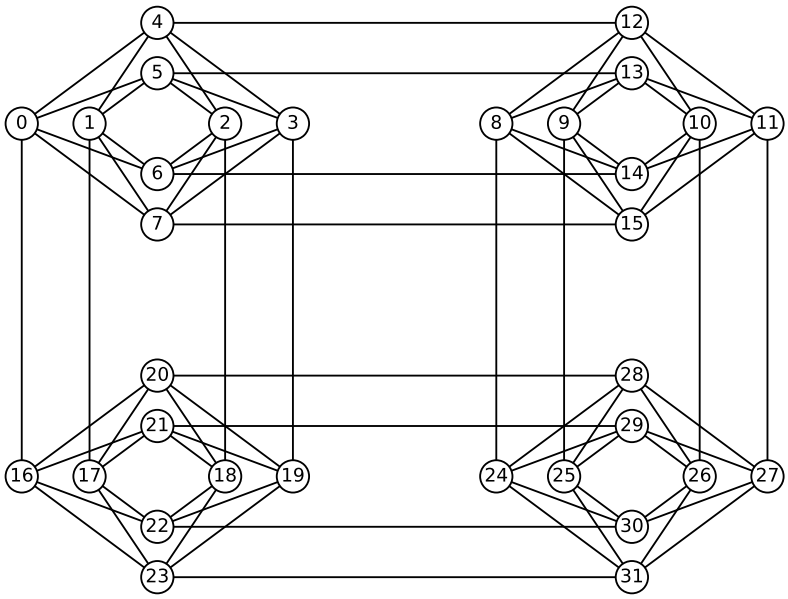

Figure 1: Chimera graph χ_2 consisting of four $K_{4,4}$ blocks. In general, the graph χ_k consists of a $k \times k$ grid of such blocks, with connections between adjacent blocks as shown.

For notational convenience, let us define a vector of length 12:

$$s = (s_0^{(v)}, s_1^{(v)}, \cdots, s_5^{(v)}, s_0^{(h)}, s_1^{(h)}, \cdots, s_5^{(h)}).$$

We will also define a function $\delta(a, b)$ such that $\delta(a, b) = 1$ if $a < b$ and 0 otherwise. The D-Wave Advantage uses a Pegasus graph defined by $s = (2, 2, 10, 10, 6, 6, 6, 6, 2, 2, 10, 10)$. Given two vertices $a = (u, w, k, z)$ and $b = (u', w', k', z')$. There is an edge $\{a, b\} \in E$ if one the following conditions is met:

1. $u = u'$, $w = w'$, k = k' and $z = z' - 1$.
2. $u = u'$, $w = w'$, $z = z'$ and $k = 2j$, $k' = 2j + 1$ for some integer $0 \leq j \leq 5$.
3. $u = 0$, $u' = 1$, $w' = z + \delta\left(j, s^{(v)}_{\lfloor k/2 \rfloor}\right)$, $k' = j$ and $z' = w = \delta\left(k, s^{(h)}_{\lfloor j/2 \rfloor}\right)$ for $0 \leq j \leq 11$.

Figure 2 shows the couplers of the Pegasus graph ϕ_3. Note that some vertex labels (e.g. node 68) may seem to be missing. The ϕ_3 topology has 16 debugging qubits that are disconnected from the main structure and are not shown in Figure 2. See [11] for a more detailed description of the Pegasus graph.

2.3 Minor embeddings

Recall the definition of a minor embedding. Given an instance of the QUBO problem, we will have to 'embed' the problem in the QPU unit of the D-Wave quantum annealer before we can solve it. This leads to the following problem:

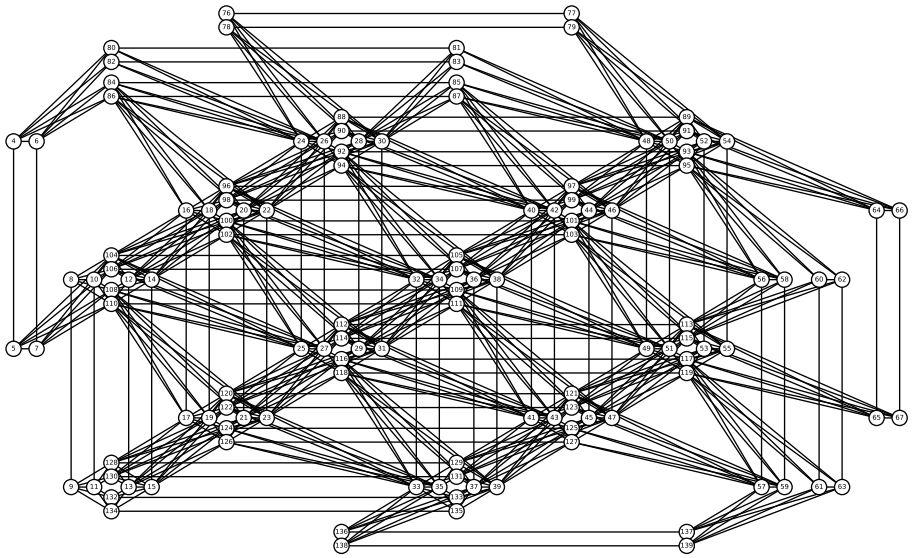

Figure 2: Pegasus graph ϕ_3. In general, the graph ϕ_k is much denser when compared with a Chimera graph with similar order. D-Wave Advantage has a ϕ_{16} chipset.

Minor Embedding Problem:

Instance: Two graphs $H = (V, E)$ and $G = (V', E')$.
Question: Compute a minor embedding function $f : V' \to 2^V$ if one exits.

The mapping f which is used to map the logical qubits of the QUBO instance to sets of connected physical qubits in the hardware graph. The variable interaction graph (G), defined by the QUBO instance, and the hardware connectivity graph (H) are called 'guest' and 'host' graphs respectively in this context.

The decision version of the Minor Embedding Problem (known as The Minor Containment Problem) is known to be NP-complete for arbitrary input graphs and computing the actual function f is NP-hard. The heavy cost of computing embeddings is one of the main obstacles in achieving any practical speedup with quantum annealers. To further complicate matters, much recent research has also shown that the 'quality' of the embedding also affects the performance of quantum annealers [1, 6, 16, 26, 27]. Given a vertex v in the guest graph, say $f(v) = X$ for some $X \subset V$. The cardinality of X is called the *map size*[2] of v. In general, smaller and more uniform map sizes would produce better results in practice [1, 6, 16, 26, 27]. For practical reasons, D-Wave implemented a heuristic-based algorithm in the Ocean SDK [30].

Naturally, with different QUBO instances (with different logical graphs), a different minor embedding would have to be computed. One potential workaround of this issue is to compute

[2]This map size is often called 'chain length' by various authors and it could be somewhat misleading since the vertices in X do not necessarily form a path. For the sake of convenience, we will use the two terms interchangeably.

the minor embedding of a Complete Graph (clique) K_n on the host graph. Since K_n has an edge between every pair of vertices, the minor embedding of K_n on the host graph can be used for any logical graphs with order smaller or equal to n. Ideally, we would want to compute the minor embedding of the largest embeddable clique since it provides the most utility. But this is not a simple task in practice. In theory, since the host graph, say ϕ_M for example, is fixed (or at least is fixed for any given generation of hardware), there is a polynomial-time algorithm to decide whether any given graph K_n is a minor of ϕ_M [28]. However, due to engineering imprecision and difficulties, the manufactured chip is very unlikely to completely match with the Pegasus graphs (e.g. there are faulty qubits and couplers represented by missing vertices and edges in the physical graph). Furthermore, the hardware is under regular maintenance and qubits and couplers often have to be taken offline or become unavailable for various other reasons (e.g. for calibration). And so the largest embeddable clique is likely to vary (at least slightly) over the lifetime of the hardware. The heuristic embedding algorithm implemented in the API does not seem very suitable for this task since the guest graph (K_n) has maximum density and it would be very computation-heavy with the heuristic algorithm if the heuristic algorithm can find a minor embedding at all.

3 Existing techniques for Clique embedding in Chimera and Pegasus graphs

In this section, we will give a brief description of existing techniques used to compute clique embeddings on Chimera graphs which can be generalized to Pegasus graphs as well. Given a Chimera graph $\chi_{M,N,L}$, the algorithm in [4] can be used to compute a minor embedding for K_{LM} with uniform map size in polynomial time. Although it is hard to show that K_{LM} is the largest embeddable clique in $\chi_{M,N,L}$, it is certainly at least very close to the optimal solution since $\chi_{M,N,L}$ does not contain K_{LM+2} minor [4] (i.e. optimal solution is K_{LM+1} if not K_{LM}).

The algorithm in [4] can also be used on induced subgraphs of $\chi_{M,N,L}$. However, it is much harder to adapt (at least from an efficiency point of view) to general subgraphs of $\chi_{M,N,L}$. As mentioned before, quantum annealers manufactured by D-Wave Systems often have inactive qubits as well as additional faulty couplers between active qubits. So the algorithm in [4] cannot be used (at least not directly) since the graph induced by the set of active qubits is not the actual hardware structure that we can use. To solve this issue, a workaround was proposed in [4] which involves enumerating all minimum vertex covers of the subgraph induced by the set of faulty couplers.

Specifically, given a subgraph $\chi'_{M,N,L} = (V', E')$ of Chimera graph $\chi_{M,N,L} = (V, E)$ (note that $\chi'_{M,N,L}$ is not necessarily an induced-subgraph of $\chi_{M,N,L}$). We first compute the edge difference $A = E(\chi_{M,N,L}(V')) - E'$ where $\chi_{M,N,L}(V')$ is the subgraph induced by the set V'. If the set A is empty, then it means $\chi'_{M,N,L}$ is an induced-subgraph of $\chi_{M,N,L}$ (i.e. there are no additional faulty couplers) and so we can use the induced-subgraph embedding algorithm directly. Otherwise, consider the graph $G = (V', A)$ which is the graph that consists of the

additional missing edges. Let $X \subseteq V'$ be a vertex cover for G and consider the induced-subgraph $\chi_{M,N,L}(V' - X) = (V'', E'')$. It is fairly easy to show that $\chi_{M,N,L}(V' - X)$ is a subgraph of $\chi'_{M,N,L}$, since we have $(V' - X) \subset V'$ and for each edge $(u, v) \in E''$ we know that $u \notin X$ and $v \notin X$ which means $(u, v) \in E'$ as otherwise $(u, v) \in A$ and the vertex cover X would need to contain either u or v. Therefore, $E'' \subseteq E'$ which means $\chi_{M,N,L}(V' - X)$ is a subgraph of $\chi'_{M,N,L}$ which is also an induced-subgraph of $\chi_{M,N,L}$ and so the aforementioned clique embedding algorithm can be used on $\chi_{M,N,L}(V' - X)$ [4].

This simple argument leads to the following approach. For each minimum vertex cover X of the graph G, we run the induced-subgraph algorithm on the graph $C_{M,N,L}(V' - X)$ and return the maximum clique found at the end. We only consider minimum vertex covers since we would like to preserve as many active qubits as possible to use in the actual embedding. It is important to note that the maximum clique embedding found using this method is not likely to be the optimal solution for $\chi'_{M,N,L}$ and there is no known efficient method to even estimate the size of the largest embeddable clique in an arbitrary Chimera subgraph, finding the exact solution is at least NP-hard (since it is a generalized version of finding clique embeddings on the complete architecture) while the vertex cover enumeration algorithm is at least fixed-parameter tractable [4]. See Appendix C for a sample implementation of this method that can be used on D-Wave 2X models (which can be modified slightly to be used on other models). Note that the minimum vertex covers are somewhat 'hard-coded' in the source code. There were 14 additional faulty couplers on the D-Wave 2X model we have used. All the missing edges have no common end-points except for two edges, $\{235, 331\}$ and $\{331, 334\}$. So it was relatively straightforward to enumerate over all minimum vertex covers; all minimum vertex covers have to contain the vertex 331 and one end-point of each of the other 12 faulty couplers. Computing minimum vertex covers will not be a simple task if the additional faulty couplers have more common end-points, much more complicated approaches (e.g. an approximation algorithm) have to be considered if it is the case.

Please note that all of the various clique embedding techniques and algorithms for the Chimera graphs also work directly on the Pegasus graphs without the need of any modification since the Chimera graph is a subgraph of the Pegasus graph [11]. One obvious disadvantage of using these Chimera graph embedding algorithms on the Pegasus graph is that they typically do not fully take advantage of the additional couplers (edges) in the Pegasus graph and so the results are in general sub-optimal. A native clique embedding algorithm for the Pegasus graph is also given in [11] which can generate K_{12M-10} embeddings in ϕ_M. Please note that there is an error in the paper [3][3] in the description of the native clique embedding method on page 9. The given set A on the said page does not produce a valid clique embedding, and it is quite easy to verify since the set would give disconnected physical qubits mapping. A correct chain descriptor[4] is $A = \{\{(0, M-2, 4)\}, \{(1, M-2, 1)\}, \{(0, 0, 3), (1, M-2, 3)\}, \{(0, M-2, 3), (1, M-1, 2)\}, \{(0, M-1, 1), (1, M-1, 0)\}, \{(0, M-2, 5), (1, 0, 4)\}, \{(0, M-1, 2), (1, M-1, 1)\}, \{(0, M-1, 0), (1, M-2, 5)\}\}$.

[3]The same error can also be found in the technical report [11] of the same name on D-Wave Systems' official website.

[4]We would like to thank Dr. Kelly Boothby from D-Wave Systems for providing us the correct formula.

4 Simulation of clique embeddings on physical graphs with random faults

As mentioned in previous sections, it is very likely that the actual hardware manufactured by D-Wave Systems will not have a complete structure. And there does not seem to be many research that focus on the performance of these various clique embedding algorithms on more realistic host graphs with inactive qubits and faulty couplers. Before we discuss our experiment setup, we will first define a few new terms in order to have a more rigorous understanding of the setup. There are numerous ways to define faulty qubits and couplers. For example, we could define faulty qubits as those with an incident faulty edge and instead of saying a qubit $v \in V'$ are faulty, we could just say the set of edges $\{\{u, v\} \mid u \in V', \{u, v\} \in E'\}$ is faulty. If we define faulty hardware components this way, we would only need to specify a set of faulty edges. However, this approach does not fully reflect the different engineering aspects of the actual hardware where there are some differences between a faulty edge and a faulty qubit (since they are different hardware components) and so we will need to specify these faulty parts in a slightly more complicated way.

Suppose we have actual hardware, with inactive qubits and/or couplers, which is described by the graph $H = (V, E)$, and suppose the complete structure was supposed to be $H' = (V', E')$. Note that H' can either be a Chimera graph or a Pegasus graph and H is an arbitrary subgraph of H'. The set of inactive qubits is $V' - V$ and the number of inactive qubits is obviously $|V' - V|$. Similarly, we can calculate the probability of faulty qubits as $\frac{|V'-V|}{|V'|}$. Now consider the subgraph induced by the set of active qubits $G = H'(V) = (V'', E'')$. If $E'' = E$, then we say that there are no additional faulty edges. Here the set of faulty edges is calculated as $E'' - E$ and the probability of an edge being faulty is $\frac{|E''-E|}{|E''|}$. A sample implementation is given in Script A to compute the probabilities of faulty qubits and edges in a given hardware structure.

We can also generate more realistic hardware structures with these types of faults. Doing so is relatively straightforward. Given a hardware architecture $G = (V, E)$, suppose that the probability of a qubit being faulty is p and the probability of an additional faulty coupler is q. First, we randomly select a set of faulty qubits, denoted by V', with uniform probabilities where each qubit in the complete architecture has a probability p of being faulty. We then compute the induced subgraph $G(V - V') = (V'', E')$, and select a set of additional faulty couplers, denoted by $E'' \subseteq E'$, in $G(V - V')$ in a similar fashion with uniform probability q. Finally, the hardware structure with random faults can be described by $G' = (V'', E' - E'')$. See Python Script B for a sample implementation.

4.1 Experiment setup

To generate physical graphs with random faults, we need to specify the probabilities of faulty qubits and couplers. Exactly what percentage of qubits (and couplers) on a D-Wave quantum annealer will be active after manufacturing is not public information, and we had to estimate these numbers from the D-Wave 2X we had access to. The D-Wave 2X quantum annealer that we have used for various experiments throughout this work had 1098 active

qubits and 3049 functional couplers[5]. Based on its physical graph, we calculated p and q to be 0.0469 and 0.0046 respectively. See Section 4.2.3 for a brief discussion of the parameters p and q.

For each $16 \leq M \leq 32$, we used the Python program in Appendix B (with p and q set to 0.0469 and 0.0046 respectively) to generate 50 faulty Pegasus graphs ϕ_M. We ran the clique embedding algorithm implemented in the minorminer Python package (part of Ocean SDK) on all generated test cases. See Appendix D for the script[6]. To measure the quality of the embeddings, we also implemented a simple greedy chain-removal algorithm. The greedy algorithm operates as follows: We first use the chain-descriptor method as described in [11][7] to build a clique embedding ignoring faulty qubits and couplers; For each logical qubit v in the clique, the physical qubits map needs to be connected (i.e. $G(f(v))$ has to be a connected graph) and so we remove v from the clique if it is not the case; We now have a set of intact chains (physical qubits maps) which are not necessarily connected pair-wise. And so we sort the intact chains in decreasing order of the number of chains they are disconnected from (i.e. for each logical qubit v, we calculate $|\{u \mid G(f(u) \cup f(v)) \text{ is not connected}\}|$); We then repeatedly remove vertices from the embedding until the remaining chains form a valid clique. A sample implementation of this method is given in Appendix E.

4.2 Experiment results and discussion

We ran both algorithms on all test cases generated. For each $16 \leq M \leq 32$, the average number of vertices in the largest embeddable clique found is presented in Table 1. The 'Max clique order' column in the table shows the largest known embeddable clique for a complete architecture (i.e. if ϕ_M has no faulty components). This number is calculated based on the chain-descriptor method [11]. The amount of time it took for each algorithm to solve all 50 instances of each ϕ_M is presented in Table 2.

As can be seen in Table 1 and Table 2, the minorminer algorithm can find better solutions than the greedy approach but is also much slower. We will discuss several key points here.

4.2.1 Degree of fault-tolerance

Since faulty components seem unavoidable (at least with current technologies), and so the architecture design process should take this into consideration when designing the topology of the chipset. Ideally, faulty components should have a minimum effect on the largest embeddable problem; in this case it would mean that the hardware architecture is very robust. However, this notion of fault-tolerance is quite difficult to be defined precisely. First of all, there is no proof as to the order of the largest embeddable clique on the physical graph ϕ_M. The chain-descriptor method produces cliques of order $12M - 10$, but we do not know

[5]As we have mentioned earlier, qubits and couplers sometimes becomes unavailable for maintenance purposes and are generally restored later. So this specific structure here is the physical graphs that we most often had access to.

[6]The official Ocean SDK API does not have any details on this particular embedding algorithm, but we suspect that it is mainly based on the algorithm described in [4]

[7]Note that the correct chain descriptor mentioned in Section 3 should be used.

Table 1: Clique embedding result

Physical graph	Physical qubits	Max clique order	Minorminer avg	Greedy avg
ϕ_{16}	5760	182	85.18	69.12
ϕ_{17}	6528	194	85.92	68.96
ϕ_{18}	7344	206	87.52	69.16
ϕ_{19}	8208	218	88.62	69.3
ϕ_{20}	9120	230	87.8	65.84
ϕ_{21}	10080	242	89.86	70.16
ϕ_{22}	11088	254	89.48	67.38
ϕ_{23}	12144	266	91.22	65.22
ϕ_{24}	13248	278	90.46	66.52
ϕ_{25}	14400	290	92.58	67.24
ϕ_{26}	15600	302	93.54	67.16
ϕ_{27}	16848	314	92.7	67.18
ϕ_{28}	18144	326	94.34	65.28
ϕ_{29}	19488	338	94.18	65.82
ϕ_{30}	20880	350	95.1	64.58
ϕ_{31}	22320	362	96.38	66.66
ϕ_{32}	23808	374	94.76	61.6

how close this value is to the actual optimal solution. Secondly, even if a better estimation is known, there is no guarantee that we will be able to find an embedding algorithm that produces the largest clique embedding (e.g. we might only be able to obtain a better bound via a non-constructive proof). It is important to note that both features (robust architecture topology and clique embedding algorithm) are equally important. Having a perfectly robust architecture is pointless if there is no embedding algorithm that can handle these faulty components effectively. Unfortunately, there does not seem to be much research (at least not publicly available research) on these topics.

While keeping the limitation of the theory mentioned in the previous paragraph in mind, we propose a crude estimation of the degree of fault-tolerance of the current clique embedding method. As mentioned earlier, if the hardware architecture has no faulty component, then the largest known embeddable clique for ϕ_M has order $12M - 10$ which can be produced using the chain-descriptor method. We define the optimal solution ratio, denoted by OPT_r, as the average embeddable clique order found by the two algorithms divided by $12M - 10$. The value OPT_r provides an indication to the degree of fault-tolerance. See Figure 3 for a plot of OPT_r values.

Obviously, one would like to see that the value of OPT_r be as close to 1 as possible. Unfortunately, Figure 3 shows that the current OPT_r values for both algorithms are far from ideal. Not only is the maximum OPT_r value less than 0.5, it also decreases as the hardware gets bigger. The scaling behavior seems linear in terms of M where the actual number of qubits in the hardware is in order of $O(M^2)$ so the scaling behavior is slightly better (decreases slower) if the OPT_r values are plotted against the number of physical qubits. This type of scaling behavior shows that even if the probabilities of hardware components

Table 2: Clique embedding time (seconds)

Physical graph	Minorminer time	Greedy time
ϕ_{16}	7548.672	3255.868
ϕ_{17}	9944.826	3526.758
ϕ_{18}	12854.408	4202.116
ϕ_{19}	16483.624	5010.189
ϕ_{20}	20667.485	4685.019
ϕ_{21}	25579.545	5925.193
ϕ_{22}	31162.816	5845.398
ϕ_{23}	37531.649	6204.689
ϕ_{24}	46321.589	6782.656
ϕ_{25}	55792.984	7656.772
ϕ_{26}	63068.193	8507.113
ϕ_{27}	73958.661	9331.7
ϕ_{28}	85649.297	9340.8
ϕ_{29}	99245.257	9601.456
ϕ_{30}	114409.068	10284.73
ϕ_{31}	131812.174	11300.102
ϕ_{32}	151603.578	11125.429

being faulty remain the same, it will have a much larger negative effect on the qualities of clique embeddings as the hardware increases in scale.

4.2.2 Greedy algorithm comparison

Note that the scaling behavior of both algorithms exhibits a similar scaling behavior in Figure 3. This was quite surprising considering the fact that the greedy algorithm is quite straightforward. As we have seen in Table 2, the greedy algorithm is much faster than the minorminer algorithm. To better understand the time differences, we define the running time ratio, denoted by Time_r, as $\frac{\text{Minorminer time}}{\text{Greedy time}}$. See Figure 4 for a plot of Time_r values.

In Figure 4, we can see that the Time_r values also exhibit a somewhat linear scaling behavior. Together with the fact that the OPT_r values of the greedy algorithm are only lower than the OPT_r values of the minorminer algorithm by a constant factor, it shows that the relatively simple greedy approach could be useful especially in time-constrained situations when the best solution quality is not required.

We could also improve the greedy algorithm in several ways. For instance, we could use a vertex cover approach similar to that described in Section 3. Instead of repeatedly removing chains after the initial step of building the clique embedding using the chain descriptor, we could compute a minimum vertex cover on the graph consisting of all the chains as vertices and broken connection between the chains as edges (i.e. an edge $\{u, v\}$ if $G(f(u) \cup f(v))$ is not connected), and just remove the chains that correspond to the vertices in the minimum vertex cover instead. We are guaranteed to remove a minimum number of chains and therefore have

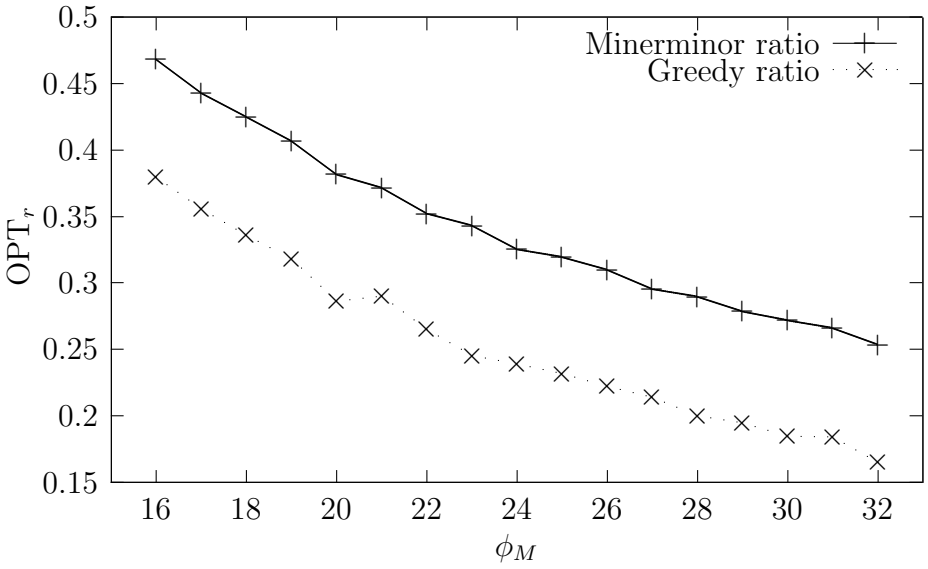

Figure 3: OPT$_r$ plot

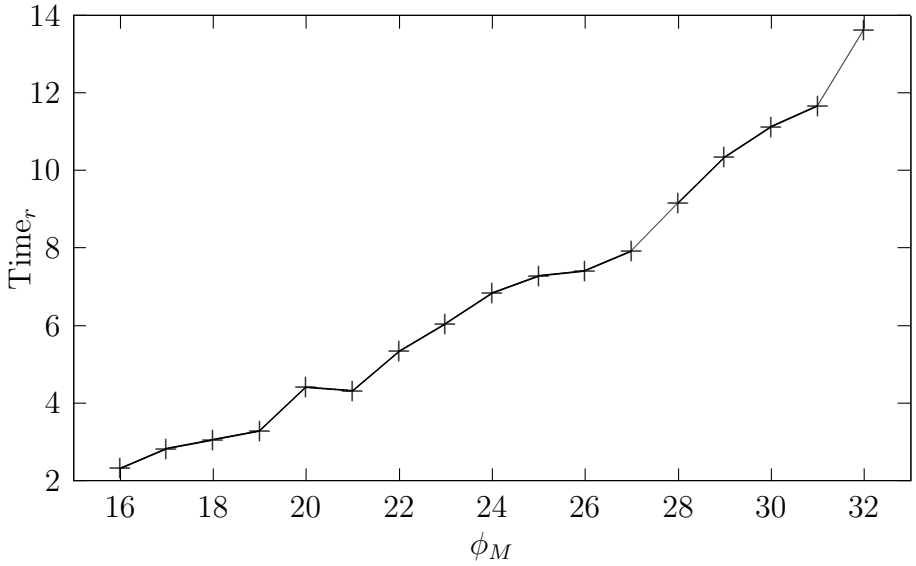

Figure 4: Time$_r$ plot

a maximum number of chains in the clique remaining at the end. The difficult part is that computing a minimum vertex cover for these graphs might not be an easy task. As can be seen in Table 1, we are basically dealing with graphs with hundreds of vertices and computing a minimum vertex cover will add a lot of overhead in terms of computation time. And it has been shown that the approximability of the vertex cover problem is not very good [9, 17] and approximation algorithms might not even be sufficient here. Secondly, we could try to extend the clique after the chain-removal process is done. The greedy algorithm does nothing with the chains it removes and since it removes most of the chains (note that we have less than 40% of chains remaining with ϕ_{16} and the number only gets lower), a lot of physical qubits and couplers are left unused. If we could enumerate the unused qubits in some systematic way, we might be able to add more chains to the clique embedding. This requires a more careful study of the hardware architecture to exploit its properties.

4.2.3 Remarks on faulty hardware component probabilities

Note that the estimated p and q values we used in the experiment are likely out of date. At the time when we conducted this experiment, we only had access to a D-Wave 2X model. We have had (somewhat limited) access to a D-Wave 2000Q and a D-Wave Advantage machine at some point. The probabilities of faulty qubits and couplers on the D-Wave 2000Q are 0.0034 and 0.0003 respectively which are a lot better than the same statistics on the D-Wave 2X. These numbers mean that only 7 of the 2048 physical qubits are faulty and there are only 2 additional faulty couplers. On the D-Wave Advantage, however, these numbers are 0.0362 and 0.0052 which are significantly worse than the D-Wave 2000Q statistics despite the Advantage being the most advanced model at the time. We suspect that using a new chipset architecture (the Pegasus graph) might be one of the reasons that lead to the probability of faulty components being higher. In hindsight, the statistics on the Advantage annealer do somewhat justify our parameter settings for p and q since these probabilities on the Advantage are relatively close to the probabilities we used in our experiment.

5 Conclusion and future work

As we have seen, computing maximum clique embeddings in a given hardware architecture with faulty components is not an easy task. Being able to find the largest embeddable clique in a given physical graph is of huge importance from a practical point of view since it completely mitigates the embedding cost for all logical graphs up to the order of the clique which is a large factor in whether quantum speedup in practice can be observed. Therefore, finding the maximum embeddable clique can be interpreted as a form of quassical computing as discussed in [1]. One may even argue that the maximum clique approach is more flexible since it is not based on any specific problem structure but rather on the physical graph topologies of quantum annealers which can be taken into consideration when designing the hardware. Unfortunately, there does not seem to be much research that focuses on finding these clique embeddings in a more realistic setting where faulty hardware components exist.

In this report, we have proposed a method of simulating hardware graphs with faulty

components as well as a (very crude) method of evaluating the robustness of the hardware structure. Based on our experiment results, we have come to the conclusion that efficient clique embedding algorithms are as equally important as larger and more connected hardware topologies. Increasing the order and the density of the chipset is important since it allows larger problems to be solved on the annealer, but a large number of qubits and couplers will be wasted if the embedding algorithm does not fully exploit them. And this is indeed the case with the current approach, we can see that the majority of the physical qubits in the hardware are left unused by the minorminer algorithm in Table 1 and Figure 3 since the minorminer algorithm produces embeddings with mostly uniform chain length so an OPT_r value of less than 0.5 essentially means that less than half of the physical qubits are unused in the final output. Based on our experiment results we suggest that the scalability of the current setup (hardware topology plus the clique embedding algorithm) is not ideal, especially if the current trend continues as D-Wave computers typically use the same family of hardware topology for a few generations before developing a new structure[8]. Computational efficiency is also an important feature, we have seen in Section 4.2.2 that even a quite unoptimized greedy algorithm can have some merits in this aspect.

We do not have enough experimental data to reach any definitive conclusion at this point due to time constraints. To gather the (somewhat) limited amount of data we had, the experiment we conducted took more than two weeks to run. But we do plan to run more experiment and to investigate further some of the points we discussed in Section 4.2.2. We also plan to study the new Zephyr topology [14] to see if the ideas we proposed in this report are applicable to the new chipset design.

References

[1] A. A. Abbott, C. S. Calude, M. J. Dinneen, and R. Hua. "A hybrid quantum-classical paradigm to mitigate embedding costs in quantum annealing". In: *International Journal of Quantum Information* 17.05 (2019), p. 1950042. DOI: 10.1142/S0219749919500424.

[2] D. Aharonov, W. van Dam, J. Kempe, Z. Landau, S. Lloyd, and O. Regev. "Adiabatic quantum computation is equivalent to standard quantum computation". In: *SIAM Review* 50.4 (2008), pp. 755–787. DOI: 10.1137/080734479. URL: https://doi.org/10.1137/080734479.

[3] K. Boothby, P. Bunyk, J. Raymond, and A. Roy. *Next-generation topology of D-Wave quantum processors*. 2020. DOI: 10.48550/ARXIV.2003.00133. URL: https://arxiv.org/abs/2003.00133.

[4] T. Boothby, A. D. King, and A. Roy. "Fast clique minor generation in Chimera qubit connectivity graphs". In: *Quantum Information Processing* 15.1 (2016), pp. 495–508.

[5] C. S. Calude, E. Calude, and M. J. Dinneen. "Adiabatic Quantum Computing Challenges". In: *ACM SIGACT News* 46.1 (2015), pp. 40–61. DOI: 10.1145/2744447.2744459. URL: http://doi.acm.org/10.1145/2744447.2744459.

[8] This might not be the case with the Pegasus graph since D-Wave has already announced the Zephyr topology.

[6] C. S. Calude, M. J. Dinneen, and R. Hua. "Quantum solutions for Densest k-Subgraph Problems". In: *Journal of Membrane Computing* 2.1 (2020), pp. 26–41. DOI: 10.1007/s41965-019-00030-1.

[7] C. S. Calude, M. J. Dinneen, and R. Hua. "QUBO formulations for the Graph Isomorphism Problem and related problems". In: *Theoretical Computer Science* 701 (2017), pp. 54–69. ISSN: 0304-3975. DOI: https://doi.org/10.1016/j.tcs.2017.04.016. URL: http://www.sciencedirect.com/science/article/pii/%20S0304397517304590.

[8] V. Choi. "Minor-embedding in adiabatic quantum computation: I. The parameter setting problem". In: *Quantum Information Processing* 7.5 (2008), pp. 193–209. ISSN: 1570-0755. DOI: 10.1007/s11128-008-0082-9. URL: http://dx.doi.org/10.1007/s11128-008-0082-9.

[9] A. E. Clementi and L. Trevisan. "Improved non-approximability results for minimum vertex cover with density constraints". In: *Theoretical Computer Science* 225.1-2 (1999), pp. 113–128.

[10] *D-Wave QPU Architecture: Topologies*. https://docs.dwavesys.com/docs/latest/c_gs_4.html. D-Wave Systems. 2021.

[11] D-Wave Systems. *Next-Generation Topology of D-Wave Quantum Processors*. https://www.dwavesys.com/media/jwwj5z3z/14-1026a-c_next-generation-topology-of-dw-quantum-processors.pdf. Feb. 2019.

[12] D-Wave Systems. "The D-Wave 2000Q^{TM} Quantum Computer Technology Overview". In: (2017). URL: https://dwavejapan.com/app/uploads/2019/10/D-Wave-2000Q-Tech-Collateral_1029F.pdf.

[13] D-Wave Systems. "The D-Wave 2X^{TM} Quantum Computer Technology Overview". In: (2016).

[14] D-Wave Systems. *Zephyr Topology of D-Wave Quantum Processors*. https://www.dwavesys.com/media/2uznec4s/14-1056a-a_zephyr_topology_of_d-wave_quantum_processors.pdf. Sept. 2021.

[15] V. S. Denchev, S. Boixo, S. V. Isakov, N. Ding, R. Babbush, V. Smelyanskiy, J. Martinis, and H. Neven. "What is the computational value of finite-range tunneling?" In: *Physical Review X* 6 (2016), p. 031015. DOI: 10.1103/PhysRevX.6.031015.

[16] M. J. Dinneen and R. Hua. "Formulating graph covering problems for adiabatic quantum computers". In: *Proceedings of the Australasian Computer Science Week Multiconference*. Vol. 18. ACSW '17. Geelong, Australia: ACM, 2017, p. 1. ISBN: 978-1-4503-4768-6. DOI: 10.1145/3014812.3014830.

[17] I. Dinur and S. Safra. "On the hardness of approximating minimum vertex cover". In: *Annals of mathematics* (2005), pp. 439–485.

[18] E. Farhi, J. Goldstone, S. Gutmann, J. Lapan, A. Lundgren, and D. Preda. "A quantum adiabatic evolution algorithm applied to random instances of an NP-complete problem". In: *Science* 292.5516 (2001), pp. 472–475. ISSN: 0036-8075. DOI: 10.1126/science.1057726. URL: http://science.sciencemag.org/content/292/5516/472.

[19] E. Farhi, J. Goldstone, S. Gutmann, and M. Sipser. *Quantum computation by adiabatic evolution*. 2000. DOI: 10.48550/ARXIV.QUANT-PH/0001106. URL: https://arxiv.org/abs/quant-ph/0001106.

[20] R. Hua and M. J. Dinneen. "Improved QUBO formulation of the Graph Isomorphism Problem". In: *SN Computer Science* 1.1 (2020), p. 19. DOI: 10.1007/s42979-019-0020-1.

[21] W. Lechner, P. Hauke, and P. Zoller. "A quantum annealing architecture with all-to-all connectivity from local interactions". In: *Science Advances* 1 (2015), e1500838. DOI: 10.1126/sciadv.1500838.

[22] A. Lucas. "Ising formulations of many NP problems". In: *Frontiers in Physics* 2.5 (2014). ISSN: 2296-424X. DOI: 10.3389/fphy.2014.00005. URL: http://www.frontiersin.org/interdisciplinary_physics/10.3389/fphy.2014.00005/abstract.

[23] A. Mahasinghe, R. Hua, M. J. Dinneen, and R. Goyal. "Solving the Hamiltonian Cycle Problem using a quantum computer". In: *Proceedings of the Australasian Computer Science Week Multiconference*. ACSW 2019. Sydney, NSW, Australia: ACM, 2019, 8:1–8:9. ISBN: 978-1-4503-6603-8. DOI: 10.1145/3290688.3290703. URL: http://doi.acm.org/10.1145/3290688.3290703.

[24] C. McGeoch. *Adiabatic Quantum Computation and Quantum Annealing. Theory and Practice*. Morgan & Claypool Publishers, 2014.

[25] C. McGeoch and P. Farré. *Advantage Processor Overview*. https://www.dwavesys.com/media/3xvdipcn/14-1058a-a_advantage_processor_overview.pdf. Jan. 2022.

[26] A. Mishra, T. Albash, and D. A. Lidar. "Finite temperature quantum annealing solving exponentially small gap problem with non-monotonic success probability". In: *Nature communications* 9.1 (2018), pp. 1–8.

[27] K. L. Pudenz, T. Albash, and D. A. Lidar. "Error-corrected quantum annealing with hundreds of qubits". In: *Nature communications* 5 (2014), p. 3243. DOI: https://doi.org/10.1038/ncomms4243.

[28] N. Robertson and P. D. Seymour. "Graph minors. XIII. The disjoint paths problem". In: *Journal of Combinatorial Theory, Series B* 63.1 (1995), pp. 65–110.

[29] G. Rose and W. Macready. *An Introduction to Quantum Annealing*. Tech. rep. Document 0712. D-Wave Systems, Inc., 2007, pp. 1–3.

[30] D.-W. Systems. *D-Wave Ocean Software Documentation*. https://docs.ocean.dwavesys.com/en/stable/index.html. Accessed: 2020-12-19.

[31] D. Wang and R. Kleinberg. "Analyzing Quadratic Unconstrained Binary Optimization problems via multicommodity flows". In: *Discrete Applied Mathematics* 157.18 (2009), pp. 3746–3753.

A Python program to compute faulty hardware component rates

```
#!/usr/bin/env python2
#usage: compute_hardware_stats.py graph_type n < hardware_edge_list
#hardware_edge_list has to contain a list of nodes in the first line
#and a list of 2-tuples representing the edges of the graph in the second line
#graph_type can be either 'C' for Chimera graphs or 'P' for Pegasus graphs

import sys
import networkx as nx
import dwave_networkx as dnx

graph_type = sys.argv[1]
graph_size = int(sys.argv[2])

nodes_list = eval(sys.stdin.readline().split('=')[1].strip())
print len(nodes_list)
sys.stdin.readline()
edge_list = eval(sys.stdin.readline().split('=')[1].strip())

if graph_type == 'C':
    complete_hardware = dnx.chimera_graph(graph_size)
elif graph_type == 'P':
    complete_hardware = dnx.pegasus_graph(graph_size)
else:
    sys.exit('Unrecognized graph type')

number_of_faulty_qubits = complete_hardware.order() - len(nodes_list)

print 'number of faulty qubits =', number_of_faulty_qubits
print 'probability of faulty qubits =', round(float(number_of_faulty_qubits)/
   complete_hardware.order(),4)

active_qubits_induced = complete_hardware.subgraph(nodes_list)

number_of_faulty_edges = active_qubits_induced.size() - len(edge_list)

print 'number of faulty edges =', number_of_faulty_edges
print 'probability of faulty edges =', round(float(number_of_faulty_edges)/
   active_qubits_induced.size(),4)
```

listings/compute_hardware_stats.py

B Python program to generate hardware graph with faulty qubits and couplers

```
#!/usr/bin/env python2
#usage: generate_faulty_hardware_graph.py graph_type n qubit_fault_rate
    edge_fault_rate
#graph_type can be either 'C' for Chimera graphs or 'P' for Pegasus graphs

import dwave_networkx as dnx
import sys
import random

graph_type = sys.argv[1]
M = int(sys.argv[2].strip())
faulty_qubit_prob = float(sys.argv[3].strip())
faulty_edges_prob = float(sys.argv[4].strip())

if graph_type == 'C':
    complete_hardware = dnx.chimera_graph(M)
elif graph_type == 'P':
    complete_hardware = dnx.pegasus_graph(M)
else:
    sys.exit('unrecognized graph type')

faulty_qubits = []
faulty_edges = []
temp_graph = complete_hardware.copy()

# compute set of faulty qubits
for i in range(complete_hardware.order()):
    if random.random() < faulty_qubit_prob:
        faulty_qubits.append(i)

# remove faulty qubits
for i in faulty_qubits:
    if i in temp_graph.nodes():
        for j in temp_graph.neighbors(i):
            faulty_edges.append((min(i,j), (max(i,j))))

temp_graph.remove_nodes_from(faulty_qubits)

for (i, j) in temp_graph.edges():
    if random.random() < faulty_edges_prob:
        faulty_edges.append((i,j))

temp_graph.remove_edges_from(faulty_edges)

print complete_hardware.order()

for i in range(complete_hardware.order()):
    if i in temp_graph.nodes():
        for j in temp_graph.neighbors(i):
```

```
            print j,
    print
```

listings/generate_faulty_hardware_graph.py

C Python program to find largest embeddable clique in Chimera subgraphs

```
#!/usr/bin/env python2
#usage: clique_emb_chimera_subgraph.py < graph_adj_list
#graph_adj_list is the physical graph in standard adj list format
#prints the largest embeddable clique in the hardware structure

from dwave_sapi2.util import get_hardware_adjacency,get_chimera_adjacency
from chimera_embedding import processor
from itertools import product
import networkx as nx
import networkx.algorithms.isolate as isolate
import sys

def read_graph(f=sys.stdin):
    n=int(f.readline().strip())
    G=nx.empty_graph(n,create_using=nx.Graph())
    for u in range(n):
        neighbors=f.readline().split()
        for v in neighbors: G.add_edge(u, int(v))
    return G

def iterate_min_vertex_cover(bin_strs, index):
    string = bin_strs[index]

    vertex_cover = []
    for i in range(len(string)):
        bin_value = int(string[i])
        vertex_cover.append(missing_edges[i][bin_value])
    vertex_cover.append(331)
    return vertex_cover

M = 12
N = 12
L = 4

G = read_graph()
complete_hardware_adj = get_chimera_adjacency(12,12,4)
complete_hardware_graph = nx.Graph()
complete_hardware_graph.add_edges_from(complete_hardware_adj)
faulty_qubits = list(isolate.isolates(G))
active_qubits = list(set(complete_hardware_graph.nodes()) - set(faulty_qubits)
    )
active_qubits_induced = complete_hardware_graph.subgraph(active_qubits)
```

```
missing_edges = [[72, 168], [86, 94], [163, 165], [248, 252], [552, 557],
    [576, 582], [611, 614], [729, 732], [755, 851], [846, 854], [906, 1002],
    [1147, 1149]]
bin_str = [''.join(p) for p in product('10', repeat=len(missing_edges))]

# enumerate all min vertex cover and get clique embeddings
max_clique_size = 0
largest_clique = []

for i in range(len(bin_str)):
    min_vertex_cover = iterate_min_vertex_cover(bin_str, i)
    qubits = list(set(active_qubits_induced.nodes()) - set(min_vertex_cover))

    used_to_embed = active_qubits_induced.subgraph(qubits)
    used_to_embed_adj = used_to_embed.edges()

    embedder = processor(used_to_embed_adj, M=M, N=N, L=L)
    embedding = embedder.largestNativeClique()

    if len(embedding) > max_clique_size:
        max_clique_size = len(embedding)
        largest_clique = embedding
print max_clique_size
print largest_clique
```

listings/clique_emb_chimera_subgraph.py

D Python program to find the largest embeddable clique in physical graph

```
#!/usr/bin/env python2
#usage: find_clique_embedding_busclique.py < faulty_edge_list
#faulty_edge_list should contain multiple lines of list of 2-tuples
#representing faulty edges in the hardware, each line is treated as
#a different hardware graph

import dwave_networkx as dnx
import sys
from minorminer import busclique

M = int(sys.stdin.readline().strip())
sys.stdin.readline()

for j in range(50):
    faulty_edges = eval(sys.stdin.readline().strip())
    P_M = dnx.pegasus_graph(M)
    coord = dnx.pegasus_coordinates(M)

    P_M.remove_edges_from(faulty_edges)
```

```
    max_clique_size = 12*(M-1)

    for i in range(max_clique_size, 0, -1):
        result = busclique.find_clique_embedding(i, P_M)

        if len(result) == i:
            embedding = []
            for k in range(i):
                embedding.append(result[k])

            break
```

listings/find_clique_embedding_busclique.py

E Greedy algorithm to find the largest embeddable clique in physical graph

```
#!/usr/bin/env python3
#usage: find_clique_embedding_greedy.py M < faulty_edge_lists
#M is the Pegasus graph spec
#compute the largest embeddable clique using the greddy algorithm

import dwave_networkx as dnx
import networkx as nx
import sys
from dwave.embedding.pegasus import find_clique_embedding

def output_max_clique_embedding(emb):
    n = len(emb)
    print(n)
    for i in range(n):
        for j in range(n):
            if not i == j:
                print(j),
        print
    print(emb)

def embed_with_faults(faulty_edges):
    faulty_chain_edges = []
    faulty_inter_chain_edges = []
    logical_qubits_with_broken_chain = []

    num_logical_qubits_affected = 0
    max_clique = embed_no_faults()
    max_clique_order = len(max_clique)

    max_clique_with_faults = []
    max_clique_with_faults_order = 0

    physical_qubits_with_faults = []
```

```
broken_chains_logical_qubits = []
broken_chains_physical_qubits = []
num_broken_chains = 0

actual_hardware = P_M.copy()
actual_hardware.remove_edges_from(faulty_edges)

# count the number of broken chains
temp = []
for (i,j) in faulty_edges:
    for x in range(max_clique_order):
        qubits = max_clique[x]
        if i in qubits and j in qubits:
            faulty_chain_edges.append((i,j))
            temp.append(x)
            break
broken_chains_logical_qubits = list(set(temp))
num_broken_chains = len(broken_chains_logical_qubits)

for i in broken_chains_logical_qubits:
    broken_chains_physical_qubits.append(max_clique[i])

for (i,j) in faulty_edges:
    if (i,j) not in faulty_chain_edges:
        faulty_inter_chain_edges.append((i,j))

# remove broken chains from embedding
temp = []
chains_removed = []
for i in range(max_clique_order):
    if i not in broken_chains_logical_qubits:
        max_clique_with_faults.append(max_clique[i])
    else:
        chains_removed.append(max_clique[i])

# new set of qubits without broken chains
max_clique_with_faults_order = len(max_clique_with_faults)
physical_qubits_used = [physical_qubit for chain in max_clique_with_faults
 for physical_qubit in chain]

# mapping of physical qubits to logical qubits of clique
physical_to_logical_map = {}
for physical in physical_qubits_used:
    for i in range(max_clique_with_faults_order):
        temp = max_clique_with_faults[i]
        if physical in temp:
            physical_to_logical_map[physical] = i

set_of_disconnected_logical_qubits = []
# verify if new set of qubits is clique
for (i, j) in faulty_inter_chain_edges:
    # check if both end-points are in-use
    if i in physical_qubits_used and j in physical_qubits_used:
```

```
            # check if they logical qubits map are disconnected
            logical_i = physical_to_logical_map[i]
            logical_j = physical_to_logical_map[j]
            temp = list(max_clique_with_faults[logical_i])
            temp.extend(max_clique_with_faults[logical_j])

            if not nx.is_connected(actual_hardware.subgraph(temp)):
                if (logical_i, logical_j) not in
set_of_disconnected_logical_qubits and (logical_j, logical_i) not in
set_of_disconnected_logical_qubits:
                    set_of_disconnected_logical_qubits.append((logical_i,
logical_j))

# greedily remove chains to rebuild clique
unpacked_disconnected_logical_qubits = []
for i in range(len(set_of_disconnected_logical_qubits)):
    unpacked_disconnected_logical_qubits.extend(
set_of_disconnected_logical_qubits[i])

broken_connection_count = {}

# count the number of chains where the connection is broken
for i in unpacked_disconnected_logical_qubits:
    if i not in broken_connection_count:
        broken_connection_count[i] = 1
    else:
        broken_connection_count[i] += 1

# get logical qubits in order of num of broken connections
set_logical_qubits_to_remove = sorted(broken_connection_count, key=
broken_connection_count.get, reverse=True)

set_of_disconnected_logical_qubits_copy = list(
set_of_disconnected_logical_qubits)
removed_logical_qubits = []

# now remove chains greedily
for i in range(len(set_logical_qubits_to_remove)):
    logical_qubit_to_remove = set_logical_qubits_to_remove[i]

    for j in range(len(set_of_disconnected_logical_qubits)):
        broken_connection = set_of_disconnected_logical_qubits[j]
        if logical_qubit_to_remove in broken_connection and
broken_connection in set_of_disconnected_logical_qubits_copy:
            set_of_disconnected_logical_qubits_copy.remove(
broken_connection)
            removed_logical_qubits.append(logical_qubit_to_remove)
    if len(set_of_disconnected_logical_qubits) == 0:
        break

removed_logical_qubits = list(set(removed_logical_qubits))

removed_logical_qubits.sort()
removed_logical_qubits.reverse()
```

```
    for i in removed_logical_qubits:
        chains_removed.append(max_clique_with_faults[i])
        max_clique_with_faults.pop(i)

    if checkclique(actual_hardware, max_clique_with_faults) == False:
        print('Error')

    unused_chains = list(chains_removed)

    post_process_clique = add_qubits(max_clique_with_faults,faulty_edges,
    unused_chains)
    if checkclique(actual_hardware, post_process_clique) == False:
        print('Error')

    return post_process_clique

# verify clique embedding is valid
def checkclique(host, embedding):
    n = len(embedding)
    temp = []
    for qubit in embedding:
        temp.extend(qubit)
    if not len(set(temp)) == len(temp):
        print 'chains not disjoint', emebdding
        return False
    # check each chain is connected
    for chain in embedding:
        if nx.is_connected(host.subgraph(chain)) == False:
            #print chain, 'is not connected'
            return False
    # check each pairs of chain is connected
    for i in range(n):
        for j in range(i+1, n):
            temp = list(embedding[i])
            temp.extend(embedding[j])
            if nx.is_connected(host.subgraph(temp)) == False:
                print embedding[i], embedding[j], 'are not connected'
                return False
    return True

def embed_no_faults():
    p = [4, 0, 2, 1, 3, 5]
    B = []
    for k in range(len(p)):
        for w in range(6*M-9):
            if (4 <= (6*w + k)) and ((6*w+k) < (6*M-9)):
                #changed 1,k
                B.append([[0,w,k], [1, w, p[k]]])

    C = A + B
    clique = []

    for t in range(2):
```

```
            for c in C:
                temp = []
                for z in range(M-1):
                    for (u,w,k) in c:
                        temp.append([u,w,2*k+t,z])
                clique.append(temp)

        clique_linear_index = []

        for v in clique:
            temp = []
            for p in v:
                temp.append(coord.pegasus_to_linear(p))
            clique_linear_index.append(temp)
        emb = []
        for linear in clique_linear_index:
            emb.append(linear)

        return emb

    def add_qubits(max_clique_with_faults, faulty_edges, unused_chains):

        actual_hardware = P_M.copy()
        actual_hardware.remove_edges_from(faulty_edges)

        # enumerate each chain to see if possible to add vertex to clique
        for chain in unused_chains:
            chain_subgraph = actual_hardware.subgraph(chain)
            for component_vertices in nx.connected_components(chain_subgraph):
                potential_clique = list(max_clique_with_faults)
                potential_clique.append(list(component_vertices))

                if checkclique(actual_hardware, potential_clique):
                    max_clique_with_faults = potential_clique

        return max_clique_with_faults

    M = int(sys.stdin.readline().strip())

    A = [[(0, M-2,4)],
        [(1, M-2,1)],
        [(0,0,3), (1, M-2, 3)],
        [(0, M-2, 3), (1, M-1, 2)],
        [(0, M-1, 1), (1, M-1, 0)],
        [(0, M-2, 5), (1, 0, 4)],
        [(0, M-1, 2), (1, M-1, 1)],
        [(0, M-1, 0), (1, M-2, 5)]]

    P_M = dnx.pegasus_graph(M)
    coord = dnx.pegasus_coordinates(M)
    pegasus_2_chimera_max_clique = eval(sys.stdin.readline().strip())

    for i in range(50):
        faulty_edges = eval(sys.stdin.readline().strip())
```

```
    result = embed_with_faults(faulty_edges)
```

listings/find_clique_embedding_greedy.py