CDMTCS
Research
Report
Series

D-Wave Experimental Results for an Improved QUBO Formulation of the Broadcast Time Problem

Yan Kolezhitskiy
Michael J. Dinneen
André Nies
Department of Computer Science,
University of Auckland,
Auckland, New Zealand

CDMTCS-525
April 2018

Centre for Discrete Mathematics and
Theoretical Computer Science

D-Wave Experimental Results for an Improved QUBO Formulation of the Broadcast Time Problem

Yan Kolezhitskiy, Michael J. Dinneen and André Nies
Department of Cumputer Science, University of Auckland, Auckland, New Zealand

Contents

1 Introduction

Quantum computing has been a popular phenomena in Computer Science over the past few decades. More specifically in recent years, the D-Wave, a commercially available quantum computer, has been receiving significant attention due to the fact that it can take in as input non-trivial NP hard problems and produce results of varying accuracy.

The *broadcast problem* is a popular optimization problem of graph theory, it asks if there is an efficient way to spread a message across a network in a given time frame. The main purpose of our efforts is two-fold; To evaluate the capacity of the D-Wave quantum computer to tackle this type of problem. Also to evaluate the current QUBO formulation (a specific presentation of the problem which the D-Wave can solve) of the broadcast problem and compare to a previous formulation. We present here the results as generated by the D-Wave on the current best-known QUBO formulation. We also compare them to the previous results, concluding that indeed the current QUBO formulation of the broadcast problem is more efficient.

2 Quantum Computing

2.1 What is Quantum Computing

The simplest, and yet the least precise, way of understanding quantum computing, is that it is computing performed by a device (i.e. a computer) that utilizes 'quantum phenomena' such as super-positioning and entanglement. Although indeed this definition captures all current forms of quantum computing, it is still quite vague and also technically encapsulates phenomena which may arguably not be considered as quantum computing. For example if we take a classical computer with an attached *quantum random number generator*, is that a quantum computer? In order to be more precise we give some descriptions as to how a quantum computer is implemented, and mathematically modelled.

An important feature of a quantum computer is that it is probabilistic. So any result it generates is not guaranteed to be accurate, but is only correct with a probability of x percent. Although at first glance this does seem quite problematic, it can be mitigated with multiple runs. This will be realistic, especially if a quantum computer does indeed offer a significant speedup to a given computation.

Conventionally, a quantum computer operates on *qubits*, as opposed to bits that are used by a standard digital computer. A qubit, or quantum bit, is essentially a super-positioned bit that can be anywhere between a 1 and a 0. The idea here is to perform computations on super-positioned qubits in such a way that when the computations are completed, and the super-positioning of the string of qubits is collapsed, it will be probablisitcally likely that what we will get will be the solution to the problem.

2.1.1 The Mathematics

Mathematically, a qubit is represented as an element of $\mathbb{C}^2$, and is written as $|a\rangle$ (Dirac bra-ket notations). The idea of measuring a qubit, and consequently collapsing its super-positioning, is the same as projecting $|a\rangle$ onto the basis $\{0, c\}$. There are a number of different ways to mathematically model a quantum computation. In a later section we will discuss two such methodologies, and give a brief overview of the mathematics behind them.

2.1.2 Physical Implementation

Unlike bits, which are implemented by electrical current, qubits have to be implemented by some entity which exhibits the desired quantum properties. So a good choice, for instance, is a particle which can have a super-positioned 'spin'[1]. That way we can identify the mathematical value 0 with one spin (i.e. positive) and 1 with another spin (i.e. negative). And when the computation is completed, the spin is simply measured by some canonical procedure, and results returned for further computation.

2.2 History of Quantum Computing

Quantum computing borrows heavily from contemporary physics. It was first suggested by Manin in 1980 [10], and shortly thereafter independently introduced by Feynman in 1981 [6].

A mathematical model of a quantum computer followed shortly, with Deutsch introducing the *Quantum Turing Machine* in 1985.

QC did receive interest, but it was not until 1994 when Shor's algorithm [9] was first introduced that people had a direct example of the type of computational power that could be available with quantum computing. To summarize, Shor's algorithm can factor numbers in polynomial time, something that is believed to be hard.

The model of adiabatic quantum computation, which we are interested in, was introduced in 2000. The company D-Wave Systems began experimenting with the implementation, producing a prototype quantum computer in 2007. In 2011 they released the world's first commercially available quantum computer, D-Wave One. Their latest model, the D-Wave 2000Q was released in 2017.

2.3 Different Types of Quantum Computing

There are a few different types of models of a quantum computer. Here we discuss two models; *the quantum gate model* and *the adiabatic model*. It is noteworthy that the results suggest that the two models are polynomially equivalent, and indeed that does make conceptual sense as the adiabatic model can solve NP hard problems. By definition this means that any problem that is in NP, and by extension P, can be translated in polynomial time into the given NP hard problem.

Here we discuss two:

2.3.1 Quantum Gate model

This is perhaps the most popular model of a quantum computer. It is very much analogous to the implementation of a classical computer. Wherein a standard computer is implemented using circuits that are composed of various logic gates (i.e. NOT, OR, XOR etc.), this model of a quantum computer has circuits composed of *quantum gates*. One way to conceptually understand a quantum gate, is that it changes the *quality* of a superposition of a qubit, but perhaps a more sensical notion can be derived from looking at it mathematically. Within the mathematical model, a quantum gate is represented as an invertible matrix that acts on the tuple over the complex numbers, thereby facilitating the evolution of the qubit. Because the matrix is invertible, we know that all quantum gates have inverses. That is to say any operation can be reversed. This is quite different to standard computing, where operators like XOR cannot easily be reversed without more information about the original state.

[1] Spin is the angular momentum of a particle.

Obviously some quantum gates, such as the NOT gate, can be replicated classically, but not others. For instance there is something called the $\sqrt{NOT}$ gate, which in essence has its double application serve as the NOT gate. There is no classical gate that can flip a bit with a double application, and then flip it again with another double application.

A quantum gate computer, then, uses a circuit of quantum gates to evolve qubits in desired ways, after which a 'reading' is taken that collapses the super-positioning and yields a result.

2.3.2 Adiabatic Model

The adiabatic model is significantly different from the quantum gate model. The D-Wave quantum computer falls into this category.

The term *adiabatic* is defined to describe a process in which heat does not leave a given system. Here it is used due to the *adiabatic theorem*, which plays a vital part in any computations performed under this model.

The adiabatic theorem, introduced by Max Born and Vladimir Fock in 1928 [3] reads as such:

"A physical system remains in its instantaneous eigenstate if a given perturbation is acting on it slowly enough and if there is a gap between the eigenvalue and the rest of the Hamiltonian's spectrum."

The idea here, is to use a mixture of processes to minimize the free energy of a system, thereby forcing it to fall into its lowest energy state.

Of course this in-itself would not solve any mathematical problems. What needs to be done happens through a very specific encoding of a problem *into the actual state* of the system in such a way, where the lowest energy state *will* represent an optimum solution to the problem, under the given encoding. This can be done by constructing a specific Hamiltonian[2].

Because this process *approaches* a global minima, the overall approach is considered heuristic. If the global minima, i.e. ground state of the Hamiltonian, is reached, then we get an exact solution. But due to the probabilistic nature of quantum computing, we may only get the value of a local minima, corresponding to a sub-optimal solution.

In latter sections we give specific examples of this in regards to the D-Wave.

3 D-Wave

3.1 Company, History, and Overview

The D-Wave is the world's first commercially available quantum computer. It is designed and manufactured by the Canadian company D-Wave Systems inc. The company was founded in 1999, and the name refers to their early efforts in using the D-Wave superconductors[3]. It began as an offshoot of the University of British Columbia (UBC), but now has expanded into its own entity, with a number of physical locations across Canada and USA.

In 2011, D-Wave Systems announced the D-Wave One, a quantum computer running on a 128 qubit processor. It is worth mentioning that a research team led my Daniel Lidar concluded that

[2]The operator $H(t)$ that corresponds to the total energy of a system.

[3]A type of high temperature superconductor.

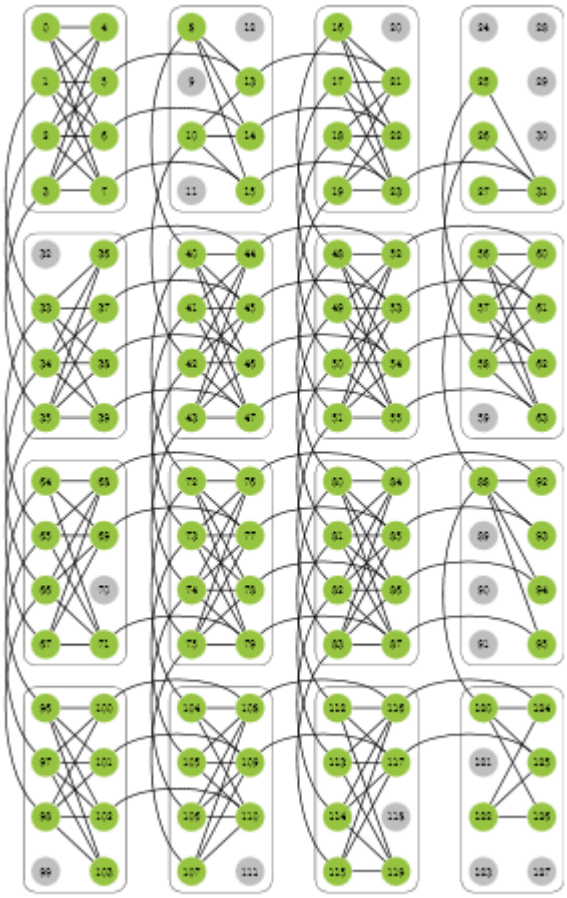
the D-Wave one showed no speed increase in comparison to a classical computer, even though there was evidence that quantum annealing did occur [1].

In 2012, D-Wave Two was released, operating with a QPU (quantum processing unit) of 512 qbits.

Three years lates in 2015, D-Wave 2X was released, operating with 1200 qubits,

In 2017 the D-Wave 2000Q was released, and as the name suggest operated with 2000 qubits.

The D-Wave QPU consists of qubits in a *Chimera graph*. A Chimera graph is a grid of complete four by four bipartite graphs ($K_{4,4}$), with specific connections between these:



3.2 The D-Wave 2X Mathematical Model

The University of Auckland has access to an D-Wave 2X quantum computer, which was used in all our work as discussed in the latter sections.

The D-Wave uses quantum annealing to solve the Ising Model problem, which is an optimization problem. This is very similar to the QUBO problem, which is the one we use. Indeed the two are easily interchangeable using a specific function. In overview, we produce an objective function, and the D-Wave utilizes its processes to find the global minima thereof.

3.2.1 The Ising Model Problem

As previously mentioned, the actual problem that is solved by the D-Wave is the Ising model problem, which is NP hard.

The Ising model problem looks at finding some $s \in \{1, -1\}^n$ such that the following formula is minimized:

$$\sum_{1 \leq i < j \leq n} J_{i,j} s_i s_j + \sum_{1 \leq i \leq n} h_i s_i$$

Where $h_i \in \mathbb{R}$ are biases and $J_{i,j} \in \mathbb{R}$ are couplings.

This is particularly useful here because the physical qubits (ie particles) either have a positive spin (i.e. 1) or a negative spin (i.e. -1).

3.2.2 The QUBO Problem

The QUBO (Quadratic Unconstrained Binary Optimization) problem is a specific instance of *Interger Programming*[4]. Given a binary vector x of length n and an $n \times n$ upper triangular matrix Q with values in the reals, we try to find an $x \in \mathbb{B}^n$ such that $xQx^t = b$ for some b is minimal. When this is a decision problem (i.e. either x is a solution or its not), then usually $xQx^t = 0$.

Specifically we note that it is very easy to transfrom a QUBO problem into an Ising model problem through the following mappings:

$$x_i \mapsto \frac{s_i + 1}{2} \text{ for } i = 1, ..., n$$

$$Q_{i,j} \mapsto J_{i,j} \text{ for } 1 \leq i < j \leq n$$

$$Q_{i,i} \mapsto \frac{1}{2}\Big(h_i - \sum_{i<j} J_{i,j} + \sum_{i>j} J_{i,j}\Big) \text{ for } i = 1, ..., n$$

3.3 QUBO Formulation and Embedding

As discussed, in order to solve any problem using the D-Wave, we first have to transcribe it into a QUBO instance. This is not a straight forward procedure. Although in some cases it can be trivial, for the most part it is an act of ingenuity. The first part to this is to decide what each binary component of the vector $x \in \mathbb{B}^n$ will represent, and then what remains is to come up with a set of equations that equal 0 when all the *right* x_i are 1. The QUBO formulation presented in the following section serves as a clear example of this. The general strategy as utilized by Fowler in [7] was to look at the conditions that ensured any given instance of a certain format was indeed a solution, and transcribe them into specific summations.

Once the summations are given, it is an algorithmic process to construct the QUBO matrix. All the terms of the form cx_ix_j are looked at, and the associated constant c becomes the $ijth$ entry of the matrix Q. With linear terms such as cx_i, we simply treat them as cx_i^2 and apply the same procedure. This is allowed because they are in $\mathbb{B}$, and we note that $x_i = x_i^2$. We also ignore all

[4] An optimiztion problem where some values are restricted to $\mathbb{Z}$.

constants, and add them to the offset. This is enough to convert any appropriate summations into an appropriate matrix Q.

This Q will then need to be embedded into the Chimera graph architecture of the D-Wave QPU. We treat each i as a qubit, and if the $ijth$ entry of the matrix is of some value $c \neq 0$, we appropriately weigh the entanglement of the qubits corresponding to i and j. The problem here, of course, is that not every such matrix we deal with represents a sub-graph of the Chimera graph, and so we have to compromise. The compromise itself revolves around looking for a graph minor of the graph of Q in the QPU. In a nutshell we look for an embedding of the graph where each logical qubit is mapped to some chain of physical qubits that will then be entangled to act as one qubit. Indeed this is known in literature as *the graph minor problem*, and itself is NP complete.

This raises two issues; first the fact that the embedding is an NP complete problem takes away from the efficiency of the D-Wave. Secondly, the smaller the ratio of logical qubits to physical qubits, the more the chance of an incorrect solution, due to a number of factors. That is, one such issue is when the physical qubits in one chain return different values, in which case it becomes an important question as to how to read the actual returned value from these.

3.4 The Contraversy

It is still debated as to whether or not the D-Wave is a quantum computer. Most definitely it is not a universal quantum computer, as it only solves de-facto one mathematical problem. Further, it is questionable whether or not the D-Wave offers any speedup to classical computation. As of now there are mixed results.

However, the D-Wave is getting significant attention, and more and more is known about its properties as time passes. Thus it is the authors' opinion that this is a noteworthy phenomena regardless of the truth about its ultimate nature, which will no doubt sooner or later become a solved problem.

4 The Broadcast Problem

In his thesis [7], Fowler introduced the QUBO formulation of the broadcast problem, which we base our work on. We present an improved version thereof, which is nonetheless equivalent. The improvement lies in the simplification of H_2.

4.1 Definition

The broadcast time problem asks if, given a graph G and some positive integer t, if its possible to effectively spread a message across a network from some originator vertex v_0 in t steps, such that each node, once in possession of the message, can only transmit to one other node per time. To formalize this, we consider the following:

A *broadcast tree* of depth t for a graph $G = (V, E)$, with originator vertex $v_0 \in V$ is a sequence $V_i \in P(V)$ (where $V_0 = \{v_0\}$) with $0 \leq i \leq t$, and a sequence of arcs A_j with $1 \leq j \leq t$, such that:

(B1) Each arc in A_i is an oriented edge of E.

(B2) $V_i = V_{i-1} \cup \{w \mid (u, w) \in A_i, u \in V_{i-1}, w \notin V_{i-1}\}$.

(B3) for any $(u, v) \in A_i, u \in V_{i-1}$ and $v \notin V_{i-1}$.

(B4) a vertex appears at most once in A_i.

(B5) $V_t = V$.

4.2 QUBO formulation

Task: Given a graph G, a positive integer t, and some starting vertex v_0, does there exist a broadcast tree with these parameters?

We present the following QUBO formulation (H_I, M_I), where $M_I = 0$ is the cut off variable, and H_I is the Hamiltonian.

The Hamiltonian involves the following variables:

- The two variables $e_{uv,i}$ and $e_{vu,i}$ for each edge $\{u, v\} \in E$ and $2 \leq i \leq t$.
- The single variable $e_{v_0u,i}$, for every $\{v_0, u\} \in E$ and $1 \leq i \leq t$.

Essentially a variable $e_{uv,i} \in \mathbb{B}$ represents whether or not the vertex u broadcasts the message to the vertex v in th ith step. Hence a given $e \in \mathbb{B}^{dim(H_I)}$ represents both the sequences $V_{e,i}$ and $A_{e,i}$, where:

$V_{e,i} = \{v \in V \mid e_{uv,j} = 1 \text{ for some } 1 \leq j \leq i\} \cup \{v_0\}$ and
$A_{e,i} = \{(u, v) \mid e_{uv,i} = 1\}$

Note that the variables involving v_0 are defined separately to the others. This is due to the fact that v_0 cannot receive a message, and thus we do not need to bother with variables of the form $e_{u0,i}$.

We now define the Hamiltonian:

Hamiltonian: $H_I(e) = H_1(e) + H_2(e) + H_3(e)$, where:

$$H_1(e) = \sum_{v \in V \setminus \{v_0\}} \Big(1 - \sum_{(u,v) \in E} \sum_i e_{uv,i}\Big)^2$$

$$H_2(e) = \sum_{v \in V, i} \Big(\sum_{(u,v),(w,v) \in E} e_{vu,i} e_{vw,i} \Big)$$

$$H_3(e) = \sum_{v \in V \setminus \{v_0\}} \Big(\sum_{(u,v) \in E, i} e_{uv,i} \Big(\sum_{j \leq i, w \neq v_0} e_{vw,j} \Big) \Big)$$

We interpret these constraints as thus:

- $H_1(e)$ returns 0 if each vertex excluding v_0 has exactly one incoming arc (i.e. receives the message exactly once). It returns a value greater or equal to 1 otherwise.
- $H_2(e)$ returns 0 if no vertex is the source node to two arcs in any A_i (i.e. no vertex broadcasts to two different nodes simultaneously). It returns a value greater or equal to 1 otherwise.
- $H_3(e)$ returns 0 if no vertex broadcasts the message before or simultaneously to receiving it (when there is no arc (u, v) in any A_i with $u \in V_{i-1}$). It returns a value greater or equal to 1 otherwise.

4.3 Proof of Correctness

Claim 1. *When $H_I(e) = 0$, e encodes a broadcast tree of depth t in G.*

Proof. First we recall the following definitions:

$V_{e,i} = \{v \in V \mid e_{uv,j} = 1 \text{ for some } 1 \leq j \leq i\} \cup \{v_0\}$, and
$A_{e,i} = \{(u, v) \mid e_{uv,i} = 1\}$.

The claim is proven by showing that all the 5 criteria for a broadcast tree (introduced above) hold:

(B1) Each arc in $A_{e,i}$ is an oriented edge of E.

If $(u, v) \in A_{e,i}$, then by definition $e_{uv,i} = 1$. But this variable is only defined when $\{u, v\} \in E$. Thus each arc of $A_{e,i}$, for all i is an oriented edge of E.

(B2) $V_{e,i} = V_{e,i-1} \cup \{w \mid (u, w) \in A_{e,i}, u \in V_{e,i-1}, w \notin V_{e,i-1}\}$.

We note that $V_{e,i} = \{v \in V \mid e_{uv,j} = 1 \text{ for some } 1 \leq j \leq i\} \cup \{v_0\}$.
Clearly, $V_{e,i-1} \subseteq V_{e,i}$. If we take some $v \in V_{e,i} \setminus V_{e,i-1}$, then $e_{yv,i} = 1$ for some vertex y, which means that $(y, v) \in A_{e,i}$.
For a contradiction suppose that $y \notin V_{e,i-1}$ or $v \in V_{e,i-1}$. If $y \notin V_{e,i-1}$, then $e_{xy,k} = 0$ for all $k \leq i-1$ and $x \in V$. But by $H_1(e) = 0$, it must be the case that $e_{xy,j} = 1$ for some $j > i-1$. But since $e_{yv,i} = 1$ and $e_{xy,j} = 1$ with $j \geq i$, this means that y broadcasts a message before (or during) receiving it. This contradicts the fact that $H_3(e) = 0$.
Further, suppose that $v \in V_{e,i-1}$, but then $v \notin V_{e,i} \setminus V_{e,i-1}$, which contradicts our assumptions.

(B3) For every $1 \leq i \leq t$, and any $(u, v) \in A_{e,i}, u \in V_{e,i-1}$ and $v \notin V_{e,i-1}$.

Suppose $(u, v) \in A_{e,i}$, which means that $e_{uv,i} = 1$. If $u = v_0$, then by construction, $v_0 \in V_{i-1}$. If $u \neq v_0$ we observe the following: $H_1(e) = 0$ means that there is some $j \leq t$ such that $e_{wu,j} = 1$, for some $w \in V$. Suppose for a contradiction that $j \geq i$. But as $H_3(e) = 0$, it must be that $e_{uv,i} = 0$, as otherwise u will be broadcasting before it receives, a contradiction. Thus $j < i$ and hence $u \in V_{i-1}$.

For another contradiction, suppose that $v \in V_{i-1}$. This means that $e_{wv,j} = 1$ for some $j \leq t$. But then $H_1(e) > 0$, as v has two incoming arcs, one at stage j and one at stage i, a contradiction.

(B4) For each i, a vertex appears at most once in $A_{e,i}$.

Suppose for a contradiction that for some A_i a vertex v appeared more than once. This is divided into three cases:

Case 1 v is the source node of two or more arcs. Say $(v, u), (v, w) \in A_i$, which means that $e_{vu,i} = 1$ and $e_{vw,i} = 1$. But that would mean that $H_2(e) > 0$, a contradiction.

Case 2 v is the source node of one arc, and the receiving node of another. Say $(v, u), (w, v) \in A_i$, which means that $e_{vu,i} = 1$ and $e_{wv,i} = 1$, i.e. v broadcasts a message simultaneously to receiving it. But this implies that $H_3(e) > 0$, a contradiction.

Case 3 v is the end node of two arcs. Say $(u, v), (w, v) \in A_i$, which means that $e_{uv,i} = 1$ and $e_{wv,i} = 1$. But then $H_1(e) > 0$, so again we have a contradiction.

(B5) $V_t = V$.

Since $H_1(e) = 0$, every vertex $v \in V\{v_0\}$ has an incoming arc, and since $V_{e,t} = \{v \in V \mid e_{uv,i} = 1 \text{ for some } 1 \leq i \leq t\} \cup \{v_0\}$, B5 is true.

□

4.4 Comparing with Previous Methods

Another QUBO formulation of the broadcast problem exists, as presented by Calude and Dinneen in [4]. What they have done was create an *integer programming* (IP) instance of the problem, and then using a number of proven steps, converted it into a QUBO formulation. We present the IP variant below:

Given a graph $G = (V, E)$ with n vertices $V = \{0, 1, ..., n-1\}$, and m edges we present the following set of equations, where t is the time required to build a broadcast tree, $v_i \in \{0, ..., t\}$ is the time when vertex i receives the message, and $b_{i,j} \in \mathbb{B}$ is a decision variable that states whether or not i broadcasts to j. We have the following summations:

$$\sum_{j \neq 0} b_{j,0} = 0$$

$$\sum_{j \neq i} b_{j,i} = 1 \text{ for all } i \in V \setminus \{0\}$$

With the following constraints:

$$b_{i,j}(1 + v_i - v_j) \leq 0 \text{ for all } \{i, j\} \in E$$

$$b_{i,j} + b_{i,j} - (v_j - v_k)^2 \leq 1 \text{ for all } \{i, j\} \in E, \{i, j\} \in E \text{ with } j \neq k.$$

Even though the IP formulation seems relatively straight forward, the process of converting it into a QUBO instance does create significantly more variables. In a latter section we show graph per graph comparison with Fowler's method that highlights this fact. Further, we present data relevant to this formulation in Appendix 6.

4.4.1 Complexity comparison

The previous QUBO formulation requires $O(|V|^3 \log(|V|)^2)$ variables, and $O(|V|^6 \log(|V|)^4)$ interactions between variables. In comparison Fowler's formulation has a total of $2(t-1)(|E| - deg_G(v_0)) + t \cdot deg_G(V_0)$ variables. And in worst case, when $t = |V|$ and $|E| = |V|^2$, the number of variables is $O(|V|^3)$, with density of $O(|V|^5)$. Thus simply in comparison of complexity, Fowler's formulation surpasses the existing results.

5 Results: Running on the D-Wave

We generated QUBO matrices for thirty five instances of the Broadcast Problem. We also utilized the CPLEX QUBO solver in order to check that these were indeed accurate. Then we ran the D-Wave on these problems.

There were three runs. The first two runs were composed of 5000 sub-runs, and had as their settings respectively spin = 1 and spin = 10. The third run had 10,000 sub-runs, with spin = 20. Respectively these are presented in Tables 1, 2, and 3. Here spin denotes the parameter that tells us whether each spin direction of a qubit is interpreted or assigned 1/true or -1/false. This should eliminate hardware bias for multiple samples.

5.1 Comparison to Previous Results

In Appendix 6, we present the table containing the results of Calude and Dinneen's efforts from [4]. In Table 4 we give a table comparing key graphs and the number of qubits needed by each QUBO formulation. Again we note that Fowler's formulation requires significantly less qubits.

5.2 Analysis

It can be seen that the more physical qubits per logical qubit there are, the worse the quality of the solution is. For example with graphs such as K_3, where there are only two more physical qubits than logical, and all results are correct. In contrast with C_9 where there are roughly ten times more physical qubits (with a ratio of 66 to 606) in each of the three cases we have a small fraction of results being accurate, namely around less than 1%.

5.2.1 Chimera Embedding

This leads us to the first point of discussion regarding the matter; the embedding algorithm that is used to embed the logical graph of Q in the physical Chimera graph. The hypothesis is, the better the embedding, the better the results. And indeed there is a lot of truth to this. Firs we note that a longer a chain of entangled qubits is, the bigger the chance of error is for various reasons. The most obvious potential error here is, of course, the scenario where they return different values, but this in itself is caused by more subtle issues. Playing around with different embedding algorithms and quantifying over the differences is a key point to explore, and is worth a lot of attention. Indeed since the embedding algorithms are heuristic, its is likely that they don't provide optimum answers.

5.2.2 Spin Reversals

It is also not clear from a brief analysis as to how the 'spin' setting affects the solution, as there are mixed results. For example for the 'Bull' graph, the results with spin=10 are definitely more favourable than the results with 'spin = 1', but even though there are more correct results for this graph when spin = 20, there are significantly more inaccurate results. A more in-depth study **over a number of problems** should be used to determine the effects of 'spin' on the quality of the result, but the current hypothesis, is there is a 'golden' number of spins, and anything over/under will yield inferior results.

5.2.3 Comparison to Heuristic QUBO solvers

It is also noteworthy that at the moment, the results of the D-Wave are inferior to those produced by standard QUBO solvers in regards to the quality of the result. The QUBO solver CPLEX that was used for testing produced multiple solutions, all accurate. Wherein the D-Wave output was, more or less, riddled with errors.

Table 1: Spin = 1, 5000 runs.

Graph	order	s	t	# of Logical Qubits	# Physical Qubits	Total Solutions	True	False
Bull	5	0	3	18	94	440	64	376
Butterfly	5	0	3	22	123	823	134	689
C4	4	0	2	8	17	2	2	0
C5	5	0	3	18	63	139	72	67
C6	6	0	4	32	190	2941	619	2322
C7	7	0	4	38	241	3742	548	3194
C8	8	0	4	44	285	4873	287	4586
C9	9	0	5	66	606	4995	35	4960
Diamond	4	0	2	10	27	17	6	11
Grid2x3	6	0	3	26	163	1485	215	1270
Grid3x3	9	0	4	68	871	5000	28	4972
Hexahedral	8	0	3	45	394	4672	76	4596
House	5	0	3	22	136	996	239	757
K2,3	5	0	3	21	111	557	234	323
K2x1	3	0	2	4	5	1	1	0
K3,3	6	0	3	33	280	3515	413	3102
K3	3	0	2	6	8	4	4	0
K4	4	0	2	12	46	21	15	6
Octahedral	6	0	3	44	702	4486	159	4327
Q3	8	0	3	45	431	4848	66	4782
S3	4	0	3	9	22	6	6	0
S4	5	0	4	16	64	59	24	35
S5	6	0	5	25	163	632	51	581
S6	7	0	6	36	307	3218	68	3150
S7	8	0	7	49	675	3352	1	3351
Wagner	8	0	4	66	917	4916	11	4905
chromatic+1	8	0	5	44	542	4610	110	4500
chromatic+1	8	2	4	40	373	4681	12	4669
P4	4	0	3	11	30	26	10	16
P4	4	1	2	6	9	1	1	0
P5	5	0	4	22	118	702	166	536
P5	5	1	3	14	39	26	10	16
P6	6	0	5	37	282	4508	340	4168
P6	6	1	4	26	121	952	314	638
P6	6	2	3	18	55	56	21	35

Table 2: Spin = 10, 5000 runs.

Graph	order	s	t	# of Logical Qubits	# Physical Qubits	Total Solutions	True	False
Bull	5	0	3	18	94	283	61	222
Butterfly	5	0	3	22	123	715	143	572
C4	4	0	2	8	17	5	3	2
C5	5	0	3	18	63	174	89	85
C6	6	0	4	32	190	2781	693	2088
C7	7	0	4	38	241	3258	505	2753
C8	8	0	4	44	285	4806	111	4695
C9	9	0	5	66	606	4985	31	4954
Diamond	4	0	2	10	27	7	3	4
Grid2x3	6	0	3	26	163	1738	238	1500
Grid3x3	9	0	4	68	871	4990	10	4980
Hexahedral	8	0	3	45	394	4282	57	4225
House	5	0	3	22	136	750	196	554
K2,3	5	0	3	21	111	434	211	223
K2x1	3	0	2	4	5	1	1	0
K3,3	6	0	3	33	280	2789	243	2546
K3	3	0	2	6	8	4	4	0
K4	4	0	2	12	46	33	18	15
Octahedral	6	0	3	44	702	3494	49	3445
Q3	8	0	3	45	431	4098	35	4063
S3	4	0	3	9	22	6	6	0
S4	5	0	4	16	64	44	24	20
S5	6	0	5	25	163	730	51	679
S6	7	0	6	36	307	2506	71	2435
S7	8	0	7	49	675	3786	0	3786
Wagner	8	0	4	66	917	4677	28	4649
chromatic+1	8	0	5	44	542	4531	59	4472
chromatic+1	8	2	4	40	373	4777	8	4769
P4	4	0	3	11	30	29	13	16
P4	4	1	2	6	9	1	1	0
P5	5	0	4	22	118	818	247	571
P5	5	1	3	14	39	25	10	15
P6	6	0	5	37	282	3790	450	3340
P6	6	1	4	26	121	1360	366	994
P6	6	2	3	18	55	59	23	36

Table 3: Spin = 20, 10000 runs.

Graph	order	s	t	# of Logical Qubits	# Physical Qubits	Total Solutions	True	False
Bull	5	0	3	18	94	686	91	595
Butterfly	5	0	3	22	123	2127	290	1837
C4	4	0	2	8	17	8	4	4
C5	5	0	3	18	63	205	103	102
C6	6	0	4	32	190	6794	1486	5308
C7	7	0	4	38	241	8849	1250	7599
C8	8	0	4	44	285	9902	732	9170
C9	9	0	5	66	606	10000	108	9892
Diamond	4	0	2	10	27	24	9	15
Grid2x3	6	0	3	26	163	4751	633	4118
Grid3x3	9	0	4	68	871	10000	20	9980
Hexahedral	8	0	3	45	394	9914	303	9611
House	5	0	3	22	136	2422	378	2044
K2,3	5	0	3	21	111	1260	312	948
K2x1	3	0	2	4	5	1	1	0
K3,3	6	0	3	33	280	8522	830	7692
K3	3	0	2	6	8	4	4	0
K4	4	0	2	12	46	64	22	42
Octahedral	6	0	3	44	702	9941	191	9750
Q3	8	0	3	45	431	9929	216	9713
S3	4	0	3	9	22	6	6	0
S4	5	0	4	16	64	149	24	125
S5	6	0	5	25	163	2483	118	2365
S6	7	0	6	36	307	8298	224	8074
S7	8	0	7	49	675	9927	10	9917
Wagner	8	0	4	66	917	10000	70	9930
chromatic+1	8	0	5	44	542	9990	141	9849
chromatic+1	8	2	4	40	373	9979	24	9955
P4	4	0	3	11	30	34	15	19
P4	4	1	2	6	9	1	1	0
P5	5	0	4	22	118	1373	331	1042
P5	5	1	3	14	39	67	23	44
P6	6	0	5	37	282	9293	1013	8280
P6	6	1	4	26	121	2746	759	1987
P6	6	2	3	18	55	221	59	162

Table 4: Brief Comparison of the number of qubits needed by each method.

Graph	Order	New Formulation # Logical Qubits	New Formulation # Physical Qubits	Old Formulation # Logical Qubits	Old Formulation # Physical Qubits
C6	6	32	190	735	4164
Grid3x3	9	68	871	2648	
K3x3	6	33	280	915	
S7	8	49	675	1244	
Hexahedral	8	45	394	1446	
Wagner	8	66	917	1446	

We do make a point that no procedure to measure and compare the *speed* of processing was implemented, therefore it would be unfair to conclude that the current version of the D-Wave used is ultimately inferior to CPLEX, but only in terms of the quality of the solution.

6 Conclusion

In conclusion we have effectively demonstrated that the new QUBO formulation of the broadcast problem is superior to the previous one rooted in integer programming. Further, we have performed some very basic testing involving the D-Wave 2X as a QUBO solver. Having examined the results we can conclude that although the quantum computer performs quite well when the number of logical and physical qubits are similar, the performance drops as these numbers diverge.

Suggestions for future work are two-fold: the exploration of the full effects of the various settings of the D-Wave, such as 'spin', and the exploration of better embedding algorithms which will allow to map logical qubits to fewer physical qubits.

References

[1] Sergio Boixo, Troels F Rønnow, Sergei V Isakov, Zhihui Wang, David Wecker, Daniel A Lidar, John M Martinis, and Matthias Troyer. Quantum annealing with more than one hundred qubits. e-print. *arxiv*, 1304, 2013.

[2] Simon Bone and Matias Castro. A brief history of quantum computing. *Imperial College in London, http://www. doc. ic. ac. uk/˜ nd/surprise_97/journa l/vol4/spb3*, 1997.

[3] Max Born and Vladimir Fock. Beweis des adiabatensatzes. *Zeitschrift für Physik*, 51(3-4):165–180, 1928.

[4] Cristian S Calude and Michael J Dinneen. Solving the broadcast time problem using a d-wave quantum computer. In *Advances in Unconventional Computing*, pages 439–453. Springer, 2017.

[5] Michael J Dinneen and Richard Hua. Formulating graph covering problems for adiabatic quantum computers. In *Proceedings of the Australasian Computer Science Week Multiconference*, page 18. ACM, 2017.

[6] Richard P Feynman. Simulating physics with computers. *International journal of theoretical physics*, 21(6-7):467–488, 1982.

[7] Alexander Fowler. Improved qubo formulations for d-wave quantum computing. Master's thesis, University of Auckland, 2017.

[8] Richard Hua. Adiabatic quantum computing with qubo formulations. Master's thesis, ResearchSpace@ Auckland, 2016.

[9] Peter W Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM review*, 41(2):303–332, 1999.

[10] Manin Yu. Computable and uncomputable. *Sovetskoye Radio, Moscow*, 1980.

A QUBO formulation Algorithm

Here we present the Python 3 code that takes as input the triple G, s, t, where G is the adjacency list for some graph, s is the starting vertex, and t is the maximum time. The output is a python dictionary where the entry (i, j) corresponds to the $ijth$ entry in the QUBO matrix.

```
#Based on Richard Hua's work
import sys, math , networkx as nx

# takes input from command line "G S T"

def read_graph():
  file = open(sys.argv[1],"r")
  n=int(file.readline().strip())
  G=nx.Graph()
  for i in range (n):
    G.add_node(i)
  for u in range(n):
    neighbors=file.readline().split()
    for v in neighbors:
      G.add_edge(u,int(v))
  return G

def generateQUBO(G,s,t): #G is the graph, s is the starting vertex, and t is the depth of
     the broadcast tree.
  """NOTE: we establish to dictionaries; Q and Qf. While the former is
  the dictionary that will be used to construct the QUBO matrix,
  the latter is used to calculate the entry in the latter for a variable U_{vi,vj,k},
  given parameters vi, vj, and k. Also fQ is just the reverse of Qf, and is only used in
     the verification process."""
  Q = {} # dictionary with values of the QUBO matrix
  Qf= {} # intermediary dictionary that maps vertices X vertices X time to appropriate
      matrix coordinates:
  fQ ={} # reverse of Qf, used in verification process

  count = 0
  for u in G.neighbors(s): #here we initialize the variables
    for i in range(t): #note here we have 0 \leq i < t, instead of 1 \leq i \leq t
      Qf[(s,u,i)] = count
      fQ[count] = (s,u,i)
      count +=1
  for v in G:
    if v != s:
      for u in G.neighbors(v):
        if u != s:
          for i in range(1,t):
            Qf[(v,u,i)] = count
            fQ[count] = (v,u,i)
            count +=1
  for i in range(count):
    for j in range(count):
      Q[i,j] = 0
```

```
numOfVar = count

#initialize Q, note, constant of 1 was removed.
for v in G:
  #(4) Corresponding to H_2(e), makes sure each vertex outputs only once per round.
  for i in range(t):
    if i != 0 or v==s:
      for u in G.neighbors(v):
        for w in G.neighbors(v):
          if w != u and u != s and w != s:
            a = Qf[(v,u,i)]
            b = Qf[(v,w,i)]
            Q[a,b] +=1

  if v != s:
    for u in G.neighbors(v):
      # Corresponding to H_1(e), ensures each v \neq s has exactly one incoming
          transmission. Broken into 2 parts:
      for i in range(0,t):#(1.1)
        if i != 0 or u==s:
          a = Qf[(u,v,i)]
          b = a
          Q[a,b] -= 2
        for j in range(0,t):
          if i != 0 or u==s:
            for w in G.neighbors(v):#(1.2)
              if j != 0 or w==s:
                a = Qf[(u,v,i)]
                b = Qf[(w,v,j)]
                Q[a,b] +=1
        for k in range(1,i+1): #(3), corresponding to H_3(e), ensures that no vertex
            broadcasts the message before or simultaneously to receiving it.
          for w in G.neighbors(v):
            if w != s:
              a = Qf[(u,v,i)]
              b = Qf[(v,w,k)]
              Q[a,b] +=1
# Moving all entries to upper triangle:
for i in range(numOfVar):
  for j in range(numOfVar):
    if j>i:
      Q[i,j] +=Q[j,i]
      Q[j,i] = 0
print(Qf)
file = open("kmatrix.txt.","w")
file.write(str(numOfVar))
file.write("\n")
var =""
for i in range(numOfVar):
  a = ""
  for j in range(numOfVar):
      a = a + str(Q[i,j]) +"␣"
```

```
    file.write(a)
    file.write("\n")
  file.close

G = read_graph()

s = 0
t = sys.argv[2]
generateQUBO(G,s,t)
```

B Solution Versifier Algorithm

Here we present the Python 3 code that verifies the solution to this specific QUBO problem. Namely it verifies whether or not the binary vector indeed represents a broadcast tree. Because the QUBO matrix was composed using two dictionaries storing important information, we need to re-compute these dictionaries, and so the input is of the form (G, s, t, x) where G, s, t are the same as in the original algorithm, and x is the solution. We also note that in essence it incorporates the previous algorithm, because of this.

```
#Based on Richard Hua's work
import sys, math , networkx as nx

#INPUT GST x

def read_graph():
  filename = str(sys.argv[1])
  file = open(filename,"r")
  n=int(file.readline().strip())
  G=nx.Graph()
  for i in range (n):
    G.add_node(i)
  for u in range(n):
    neighbors=file.readline().split()
    for v in neighbors:
      G.add_edge(u,int(v))
  return G

def generateQUBO(G,s,t): #G is the graph, s is the starting vertex, and t is the depth of
     the broadcast tree.
  """NOTE: we establish to dictionaries; Q and Qf. While the former is
  the dictionary that will be used to construct the QUBO matrix,
  the latter is used to calculate the entry in the latter for a variable U_{vi,vj,k},
  given parameters vi, vj, and k. Also fQ is just the reverse of Qf, and is only used in
     the verification process."""
  Q = {} # dictionary with values of the QUBO matrix
  Qf= {} # intermediary dictionary that maps vertices X vertices X time to appropriate
      matrix coordinates:
  fQ ={} # reverse of Qf, used in verification process

  count = 0
```

```
for u in G.neighbors(s): #here we initialize the variables
  for i in range(t): #note here we have 0 \leq i < t, instead of 1 \leq i \leq t
    Qf[(s,u,i)] = count
    fQ[count] = (s,u,i)
    count +=1
for v in G:
  if v != s:
    for u in G.neighbors(v):
      if u != s:
        for i in range(1,t):
          Qf[(v,u,i)] = count
          fQ[count] = (v,u,i)
          count +=1
for i in range(count):
  for j in range(count):
    Q[i,j] = 0
numOfVar = count

#initialize Q, note, constant of 1 was removed.
for v in G:
  #(4) Corresponding to H_2(e), makes sure each vertex outputs only once per round.
  for i in range(t):
    if i != 0 or v==s:
      for u in G.neighbors(v):
        for w in G.neighbors(v):
          if w != u and u != s and w != s:
            a = Qf[(v,u,i)]
            b = Qf[(v,w,i)]
            Q[a,b] +=1

  if v != s:
    for u in G.neighbors(v):
      # Corresponding to H_1(e), ensures each v \neq s has exactly one incoming
          transmission. Broken into 2 parts:
      for i in range(0,t):#(1.1)
        if i != 0 or u==s:
          a = Qf[(u,v,i)]
          b = a
          Q[a,b] -= 2
        for j in range(0,t):
          if i != 0 or u==s:
            for w in G.neighbors(v):#(1.2)
              if j != 0 or w==s:
                a = Qf[(u,v,i)]
                b = Qf[(w,v,j)]
                Q[a,b] +=1
        for k in range(1,i+1): #(3), corresponding to H_3(e), ensures that no vertex
            broadcasts the message before or simultaneously to receiving it.
          for w in G.neighbors(v):
            if w != s:
              a = Qf[(u,v,i)]
              b = Qf[(v,w,k)]
```

```
79                Q[a,b] +=1
80    # Moving all entries to upper triangle:
81    for i in range(numOfVar):
82      for j in range(numOfVar):
83        if j>i:
84          Q[i,j] +=Q[j,i]
85          Q[j,i] = 0
86    print(Qf)
87    file = open("kmatrix.txt.","w")
88    file.write(str(numOfVar))
89    file.write("\n")
90    var =""
91    for i in range(numOfVar):
92      a = ""
93      for j in range(numOfVar):
94          a = a + str(Q[i,j]) +"␣"
95      file.write(a)
96      file.write("\n")
97    file.close
98    x = sys.argv[4]
99    vari = test_solution(x,Qf,fQ,G,s,t)
100   print(vari)
101
102 def test_solution(x,Qf,fQ,G,s,t): #takes as input a vector returned by D-Wave and the
      variables dictionary, its reverse, the graph G, number of steps t and initial vertex
      v:
103   count = -1
104   for i in x:
105     count += 1
106     if i == 1:
107       print(fQ[count])
108       v,u,j = fQ[count] #get the corresponding vertices and step
109       #first we check that every '1' actually corresponds to an edge in G
110       if v == s:
111         if not(u in G.neighbors(v)):
112           print(1)
113           return False
114       elif j == 0:
115         print(2)
116         return False
117       else:
118         if not(u in G.neighbors(v)):
119           print(3)
120           return False
121       # This corresponds to H(2): if v,u,i = 1, check that for all x, v,x,i = 0, ie that a
              vertex broadcasts to at most 1 other vertex per step.
122       for a in Qf:
123         v1,u1,j1 = a
124         if v == v1 and j == j1:
125           if u != u1:
126             b = Qf[(v1,u1,j1)]
127             if x[b] == 1:
```

```
            print(a,4)
            return False

    # Corresponds to H(3): if v,u,i = 1, check that for all j \leq i and all x, u,x,j =
        0.
    for k in range(j+1):
      for v2 in G.neighbors(u):
        if v2 != v:
          if v2 == s or k>0:

            var1 = Qf[(v2,u,k)]
            if x[var1] == 1:
              print(5)
              return False

  # This corresponds to H(1): for all v \in G, there are unique u and i s.t. u,v,i = 1.

  for v in G:
    if v !=s:
      varset=0
      for u in G.neighbors(v):
        for i in range(t):
          if i != 0 or u == s:
            if x[Qf[(u,v,i)]] == 1:
              if varset == 0:
                varset = 1
              elif varset == 1:
                print(6)
                return False
      if varset == 0:
        print(7)
        return False
  return True

G = read_graph()

s = int(sys.argv[2])
t = int(sys.argv[3])
generateQUBO(G,s,t)
```

C The D-Wave Run Script

Below we present the program used to submit the generated QUBO onto the D-Wave System using its software API.

```
#!/usr/bin/env python
# Broadcast QUBO (with embedding) -> Ising -> DWave

import sys, time, math, traceback

from dwave_sapi2.remote import RemoteConnection
```

```
7  from dwave_sapi2.util import get_hardware_adjacency
8  from dwave_sapi2.embedding import embed_problem, unembed_answer
9  from dwave_sapi2.util import qubo_to_ising, ising_to_qubo
10 from dwave_sapi2.core import solve_ising
11
12 from sys import exc_info
13
14 # coupler streingth for embedded qubits of same variable
15 s,s2=1.0,1.0
16 if (len(sys.argv)==2): s = float(sys.argv[1])
17 if (len(sys.argv)==3): s,s2 = float(sys.argv[1]),float(sys.argv[2])
18 print 'Embed␣scale=',s,s2
19
20 # read input
21
22 line=sys.stdin.readline().strip().split()
23 n=int(line[0])
24 print('n=', n, '␣graph=', line[1], 's=', line[2], 't=', line[3])
25
26 #Q = defaultdict(int)
27 Q = {}
28 for i in range(n):
29     line=sys.stdin.readline().strip().split()
30     for j in range(n):
31         t = float(line[j])
32         if j>=i and t!=0: Q[(i,j)]=t
33 print('Q=',Q)
34
35 (H,J,ising_offset) = qubo_to_ising(Q)
36
37 # scale by maxV
38 maxH=0.0
39 if len(H): maxH=max(abs(min(H)),abs(max(H)))
40 maxJ=max(abs(min(J.values())),abs(max(J.values())))
41 maxV=max(maxH,maxJ)
42
43 for i in range(n):
44     if len(H)>i:
45         H[i]=s2*H[i]/maxV
46     for j in range(n):
47         if j>=i and (i,j) in J:
48             J[(i,j)]=s2*J[(i,j)]/maxV
49
50 embedding=eval(sys.stdin.readline())
51 print 'embedding=', embedding
52 qubits = sum(len(embed) for embed in embedding)
53 print 'Physical␣qubits␣used=␣%s' % qubits
54
55 # create a remote connection using url and token and connect to solver
56 #
57 url = "dwave␣url"
58 token = "secret"
```

```
solver_name = "DW2X"

print('Attempting to connect to network...')
try:
    remote_connection = RemoteConnection(url, token)
    solver = remote_connection.get_solver(solver_name)
except:
    print('Error: %s %s %s' % sys.exc_info()[0:3])
    traceback.print_exc()

#print('Solver properties:\n%s\n' % solver.properties)
A = get_hardware_adjacency(solver)

# Embed problem into hardware
(h0, j0, jc, new_emb) = embed_problem(H, J, embedding, A)
h1= [val*s for val in h0]
j1 = {}
for (key, val) in j0.iteritems():
    j1[key]=val*s
j1.update(jc)
#print 'new_emb=',new_emb
assert new_emb==embedding
print 'd-wave Ising'
print 'h1=',h1
print 'j1=',j1
(Q,offset) = ising_to_qubo(h1,j1)

# call the solver

annealT,progT,readT=20,100,100
print 'annealT=',annealT,'progT=',progT,'readT=',readT
result = solve_ising(solver, h1, j1, num_reads=10000, annealing_time=annealT,
        programming_thermalization=progT, readout_thermalization=readT,
        postprocess='optimization', num_spin_reversal_transforms=20)
print 'result:', result

#newresult = unembed_answer(result['solutions'], new_emb, broken_chains='discard', h=H, j
    =J)
newresult = unembed_answer(result['solutions'], new_emb, broken_chains='vote', h=H, j=J)
print 'newresult:', newresult
```

D Information on Previous QUBO Broadcast Formulation

For comparison, here is the information regarding the QUBO formulation of the broadcast problem as done in [4].

Table 5: Number of qubits required for some small graphs families.

Graph	Order	Size	Integer Variables	Quadratic Constraints	Binary Variables	Binary Constraints	Slack Variables	Logical Qubits	Chimera Qubits
C3	3	3	10	16	50	86	96	146	394
C4	4	4	13	21	74	131	146	220	662
C5	5	5	16	26	178	324	366	544	3258
C6	6	6	19	31	240	443	495	735	4164
C7	7	7	22	36	311	580	642	953	
C8	8	8	25	41	391	735	807	1198	
C9	9	9	28	46	778	1484	1608	2386	
C10	10	10	31	51	944	1809	1948	2892	
C11	11	11	34	56	1126	2166	2320	3446	
C12	12	12	37	61	1324	2555	2724	4048	
Grid2x3	6	7	21	37	254	472	543	797	4306
Grid3x3	9	12	34	65	832	1597	1816	2648	
Grid3x4	12	17	47	93	1414	2745	3084	4498	
Grid4x4	16	24	65	133	2420	4737	5252	7672	
Grid4x5	20	31	83	173	5537	10909	11815	17352	
K2	2	1	5	7	9	15	13	22	47
K3	3	3	10	16	50	86	96	146	394
K4	4	6	17	33	94	171	202	296	1378
K5	5	10	26	61	248	469	606	854	7973
K6	6	15	37	103	366	713	981	1347	
K7	7	21	50	162	507	1014	1482	1989	
K8	8	28	65	241	671	1375	2127	2798	
K9	9	36	82	343	1264	2591	4200	5464	
K10	10	45	101	471	1574	3279	5588	7162	
K2x1=P2	3	2	8	12	36	59	64	100	170
K1x2=S2	3	2	8	12	40	68	76	116	238
K2x2=C4	4	4	13	21	74	131	146	220	662
K2x3	5	6	18	32	192	353	414	606	4823
K3x3	6	9	25	49	282	529	633	915	
K3x4	7	12	32	69	381	727	894	1275	
K4x4	8	16	41	97	503	973	1227	1730	
K4x5	9	20	50	129	976	1906	2432	3408	
K5x5	10	25	61	171	1214	2391	3124	4338	
K5x6	11	30	72	118	1468	2914	3896	5364	
K6x6	12	36	85	277	1756	3511	4804	6560	
Q3	8	12	33	65	447	851	999	1446	
Q4	16	32	81	193	2564	5045	5860	8424	

Table 6: Number of qubits required for hypercubes and some other small known graphs.

Graph	Order	Size	Integer Variables	Quadratic Constraints	Binary Variables	Binary Constraints	Slack Variables	Logical Qubits	Chimera Qubits
S2=K1x2	3	2	8	12	40	68	76	116	238
S3	4	3	11	18	64	114	130	194	505
S4	5	4	14	25	164	301	354	518	3711
S5	6	5	17	33	226	423	501	727	5120
S6	7	6	20	42	297	564	672	969	
S7	8	7	23	52	377	724	867	1244	
S8	9	8	26	63	760	1471	1736	2496	
S9	10	9	29	75	926	1803	2132	3058	
S10	11	10	32	88	1108	2168	2568	3676	
BidiakisCube	12	18	49	97	1432	2779	3124	4556	
Bull	5	5	16	28	178	324	366	544	3523
Butterfly	5	6	18	33	192	353	414	606	5927
Chvatal	12	24	61	145	1540	3013	3604	5144	
Clebsch	16	40	97	273	2708	5373	6628	9336	
Diamond	4	5	15	27	84	151	174	258	742
Dinneen	9	21	52	142	994	1950	2552	3546	
Dodecahedral	20	30	81	161	5515	10855	11645	17160	
Durer	12	18	49	97	1432	2779	3124	4556	
Errera	17	45	108	320	4480	8900	10890	15370	
Frucht	12	18	49	97	1432	2779	3124	4556	
GoldnerHarary	11	27	66	209	1414	2814	3792	5206	
Grotzsch	11	20	52	118	1288	2508	2968	4256	
Heawood	14	21	57	113	1894	3691	4100	5994	
Herschel	11	18	48	101	1252	2429	2800	4052	
Hexahedral	8	12	33	65	447	851	999	1446	
Hoffman	16	32	81	193	2564	5045	5860	8424	
House	5	6	18	32	192	353	414	606	4176
Icosahedral	12	30	73	205	1648	3257	4164	5812	
Krackhardt	10	18	47	114	1088	2116	2548	3636	
Octahedral	6	12	31	73	324	619	795	1119	
Pappus	18	27	73	145	4514	8869	9575	14089	
Petersen	10	15	41	81	1034	1995	2276	3310	
Poussin	15	39	94	276	2446	4863	6152	8598	
Robertson	19	38	96	229	5211	10287	11570	16781	
Shrikhande	16	48	113	369	2852	5715	7508	10360	
Sousselier	16	27	71	154	2474	4849	5452	7926	
Tietze	12	18	49	97	1432	2779	3124	4556	
Wagner	8	12	33	65	447	851	999	1446	