DHBW
Duale Hochschule
Baden-Württemberg
Karlsruhe

Minor embedding of QUBOs on annealers' hardware topology graphs

Studienarbeit

im Rahmen der Prüfung zum

Bachelor of Science (B.Sc.)

des Studienganges Informatik

an der Dualen Hochschule Baden-Württemberg Karlsruhe

von

Julien Meier

Matrikelnummer, Kurs: 8236502, TINF19B2

und

Dominic Plein

Matrikelnummer, Kurs: 9664381, TINF19B2

Abgabedatum:	16. Mai 2022
Bearbeitungszeitraum:	15.10.2021 - 16.05.2022
Gutachter der Dualen Hochschule:	Prof. Dr. Heinrich Braun

Eidesstattliche Erklärung

Wir versichern hiermit, dass wir unsere Studienarbeit mit dem Thema:

Minor embedding of QUBOs on annealers' hardware topology graphs

gemäß § 5 der "Studien- und Prüfungsordnung DHBW Technik" vom 29. September 2017 selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt haben. Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Wir versichern zudem, dass die eingereichte elektronische Fassung mit der gedruckten Fassung übereinstimmt.

Karlsruhe, den 16. Mai 2022

Gez. Julien Meier, Dominic Plein
Meier, Julien; Plein, Dominic

Abstract

- English -

Due to the emergence of quantum annealers with sparse topology graphs of which only subgraphs can be utilized, the minor embedding problem became a crucial part of solving QUBO problems on these machines. Since [1] proved that the embedding problem on subgraphs of those topology graphs is $\mathcal{NP}$-hard, we present two heuristics to tackle that problem. For that, we formalize two major subproblems, prove their computational intractability and present multiple heuristics for them. Additionally, we develop various mutations increasing the likelihood of a successful embedding.
Apart from that, we present a new reformulation technique for specific constraint types based on the divide-and-conquer paradigm.
Our algorithms are not state-of-the-art and leave room for improvement. The theoretical results are, however, of interest.

Abstract

- Deutsch -

Durch das Aufkommen von Quanten Annealern mit dünn besetzten Topologiegraphen, von denen nur Subgraphen genutzt werden können, wurde das Problem der Minorenfindung ein unabdingbarer Bestandteil des Lösens eines QUBO-Problems auf jenen Maschinen. Aufgrund der durch [1] bewiesenen $\mathcal{NP}$-Schwere des Problems auf Subgraphen der relevanten Topologiegraphen entwickeln wir zwei (suboptimale) Heuristiken. Dabei werden zwei interessante Subprobleme formalisiert, die in Hinblick auf ihre Komplexität analysiert und für die Heuristiken vorgestellt werden. Zudem präsentieren wir verschiedene Mutationen, die die Wahrscheinlichkeit, einen Minoren zu finden, erhöhen sollen.
Abgesehen davon entwickeln wir einen neuen, auf dem Divide&Conquer-Paradigma basierenden Reformulierungsansatz für bestimmte Constraints.
Die vorgestellten Algorithmen sind nicht "state-of-the-art" und folglich verbesserungswürdig. Die theoretischen Resultate sind hingegen teilweise neuartig.

Table of Contents

Mathematical symbols

$\mathbb{B}$	Binary number $\mathbb{B} := \{0, 1\}$
$\bar{x}$	For $x \in \mathbb{B}$, its negation is defined as $\bar{x} := (1 - x)$.
$\mathbb{Z}_n$	Index set $\mathbb{Z}_n := \{ i \in \mathbb{Z} : 1 \leq i \leq n \}$
$2^{\mathcal{U}}$	Given a set $\mathcal{U}$, the power set of $\mathcal{U}$ is defined as the set of all subsets of $\mathcal{U}$, i. e. $2^{\mathcal{U}} := \{ U : U \subseteq \mathcal{U} \}$.
$R(v)$	For two graphs H, G and a mapping $\phi : V(H) \to 2^{V(G)}$, we define the reverse mapping $R : V(G) \to 2^{V(H)}$ of ϕ as $R(v) := \{ u \in V(H) : v \in \phi(u) \}$.
$G_{Chimera}$	Chimera graph
$G_{Pegasus}$	Pegasus graph
G_{King}	King's graph
K_n	Complete graph on n vertices. That is, K_n is isomorphic to the graph $G = (V, E)$ with $V = \mathbb{Z}_n$ and $E = \{ \{ u, v \} : u \neq v, u, v \in V \}$.
S_n	Star graph on n vertices, i. e. S_n is isomorphic to the graph $G = (V, E)$ with $V = \mathbb{Z}_n \cup \{ 0 \}$ and $E = \{ \{ 0, i \} : i \in \mathbb{Z}_n \}$.
$K_{m,n}$	Complete bipartite graph on $m + n$ vertices, i. e. isomorphic to $G = (V, E)$ with $V = A \cup B$ where $A = \{ a_1, ..., a_m \}$ and $B = \{ b_1, ..., b_n \}$ are partite sets and $E = \{ \{ a_i, b_j \} : a_i \in A, b_j \in B \}$.
$V(G)$	Vertex set of a graph G.
$E(G)$	Edge set of a graph G.
$\Delta(G)$	For graph $G = (V, E)$, $\Delta(G) := \max_{v \in V} \{ \deg(v) \}$ where $\deg(v)$ is the degree of vertex $v \in V$.
$G[X]$	The subgraph $G[X]$ of G induced by the vertex set $X \subseteq V(G)$ is defined as $V(G[X]) := X$ and $E(G[X]) := \{ e = \{ u, v \} : u, v \in X, e \in E(G) \}$.
$N_G(v)$	The set of neighbors of $v \in V(G)$ in the graph G, more formally, $N_G(v) := \{ w : vw \in E(G) \}$.
$\mathcal{P}_{u,v}(G, M)$	For some graph G and two vertices $v, u \in V(G)$, $\mathcal{P}_{u,v}(G, M)$ is a path with start vertex u, end vertex v in graph G having internal vertices in $M \subseteq V(G)$, i. e. $\mathcal{P}_{u,v}(G, M) = (u, x_1, ..., x_n, v)$ with $x_i \in M, \forall i \in \mathbb{Z}_n$.

List of Abbreviations

1-In-3-SAT+	Exactly-1-Positive 3-SAT Problem
3PP	3-Partition Problem
3SAT	3-Satisfiability
CMOS	Complementary Metal-Oxide Semiconductor
CNF	Conjunctive Normal Form
CSCP	Connected Set Cover Problem
DFS	Depth-First Search
HCP	Hamiltonian Cycle Problem
ILP	Integer Linear Program
LMRP	Local Minor Repair Problem
LP	Linear Program
MCFP	Minimum-Cost Flow Problem
MVCP	Minimum Vertex Cover Problem
QUBO	Quadratic Unconstrained Binary Optimization
QPU	Quantum Processing Unit
SCP	Set Cover Problem
SVE	Super Vertex Emplacement
SVEP	Super Vertex Emplacement Problem
STP	Steiner Tree Problem
TSP	Travelling Salesman Problem
WCSCP	Weighted Connected Set Cover Problem
WSCP	Weighted Set Cover Problem

List of Figures

List of Tables

List of Algorithms

1. Motivation

Quantum annealing has attracted widespread interest and with the emergence of the new generation of quantum annealers based on the Pegasus topology by D-Wave Systems Inc., the practical use of these machines continues becoming more realistic. Quantum annealers promise to exploit quantum effects to heuristically optimize Quadratic Unconstrained Binary Optimization (QUBO) problems, that is, minimize a quadratic objective consisting of binary variables only. Regardless of whether these quantum effects actually play a role and provide a benefit, a large number of problems could be tackled that way. However, even the new topology is essentially a sparse graph and, therefore, is heavily limited with regard to connectivity, while, in practice, QUBO formulations of $\mathcal{NP}$-hard problems are typically particularly dense [2, 3]. Consequently, the challenge to embed these problems on actual topology graphs arises. This algorithmic problem is in the literature referred to as minor embedding, and we want to develop new heuristic approaches for that problem that might at some point be helpful when designing new algorithms tackling minor embedding. Apart from that, we identify and formalize different subproblems, prove their intractability, present new reformulation approaches and develop and evaluate our own embedding algorithms.

In chapter 2, we present some preliminaries to introduce annealing, the problem formulation as Ising or QUBO, some mathematical definitions for graph minors and different hardware topologies. In chapter 3, we want to motivate viewing problems in terms of their constraints instead of the pure QUBO formulation. Therefore, we discuss known integer encodings, constraint reformulation techniques and the difficulties related to those. To overcome some of these difficulties, we analyze a tiling-based reformulation procedure by [4, 5] and present our own divide-and-conquer based reformulation approach that has, to the best of our knowledge, not been published yet.

Chapter 4 is devoted to formalizing a subproblem that is solved iteratively in a well-known minor embedding algorithm [6]. In the subproblem, we seek for augmenting the mapping by placing a single previously unmapped vertex of the original graph on the hardware graph while connecting to its neighbors. We coin that problem the SVEP. We show that the SVEP is very similar to another optimization problem, prove that it remains $\mathcal{NP}$-hard

on subgraphs of the sparse hardware topology graphs and discuss its approximability to finally develop new heuristics for the SVEP. One of them is based on network flows and creates an initial solution while the other two heuristics aim at improving such an initial solution.

In chapter 5, we formalize a subproblem that arises from a mutation, we use in one of our embedding algorithms. The mutation destroys a connected part of the mapping to then repair it again, which is why, we coin the problem the Local Minor Repair Problem (LMRP). Thus, we formalize it, prove that different variants are $\mathcal{NP}$-hard on subgraphs of the relevant hardware topology graphs and present a heuristic for it.

Chapter 6 introduces our minor embedding heuristics developed and the mutations, we apply during the embedding process.

Finally, in chapter 7, we evaluate the efficacy and embedding performance of our embedding algorithms and give final remarks in chapter 8.

2. Preliminaries

2.1. Quantum Annealing

Quantum annealing is an optimization process leveraging quantum effects in physics in order to find the ground state (the systems' lowest energy state) of an encoded optimization problem, thus yielding optimal or near-optimal solutions for the original problem. In 2000, Farhi et al. proposed a quantum algorithm based on the principle of quantum adiabatic evolution [7]. They construct a time-dependent Hamiltonian $H(t)$ which describes the energy for a particular state of the system. This Hamiltonian is the sum of the initial Hamiltonian H_{init} and the problem Hamiltonian H_{final}. H_{init} describes the state of the system where all qubits are in a superposition state of 0 and 1 at the same time [8]. It is designed so that its ground state is known and we initialize the quantum system to this state in the beginning. We then evolve according to

$$H(t) = \left(1 - \frac{t}{T}\right)H_{init} + \frac{t}{T}H_{final} \tag{2.1}$$

where T is the evolution time or total running time of the algorithm and $t \in [0, T]$. As can be seen, we interpolate between the initial and the problem Hamiltonian.

While H_{final} is also easy to construct, its ground state is usually not known. It describes the solution to the optimization problem, hence we need the system to remain in its ground state, so that at time $t = T$ it is equal to or very close to the ground state of H_{final} encoding the final solution [7, 9]. According to the adiabatic theorem, this behavior is guaranteed in theory for "big enough" T, i. e. the system is evolved slowly enough to ensure $H(t)$ does not jump to a higher, excited state throughout the process [7, 8].[1]

To summarize, the system is initialized according to the Hamiltonian $H_{initial}$ and then adiabatically evolved introducing the problem Hamiltonian H_{final}. In the best case,

[1] Note that the minimum gap is defined as the distance between the two lowest energy levels, i. e. the ground state (lowest energy) and the first excited state of the interpolating Hamiltonian.

the system will end in the ground state of H_{final} encoding the desired solution to the optimization problem.

2.2. Binary Quadratic Models: Ising and QUBO

In order to define our problem Hamiltonian, we use a binary quadratic model that maps directly onto the qubits and couplers of D-Wave's Quantum Processing Unit (QPU). A well-studied Hamiltonian is that of the **Ising model** used in statistical physics. The reader is referred to [10] and [11] for an (advanced) introduction to the Ising model. Its Hamiltonian is given by: [12]

$$H(\sigma) = -\sum_{(ij)\in E(G)} J_{ij}\sigma_i\sigma_j - \mu \sum_{i\in V(G)} h_i\sigma_i \tag{2.2}$$

with $\sigma_i, \sigma_j \in \{-1, +1\}$ indicating the atomic spin of the qubits. Note that this notation reflects the qubits' layout in the QPU: their topology is given by a graph G with edges (couplings) $E(G)$ and vertices (qubits) $V(G)$. **Qubit biases (external influences on the spin) are given by the linear coefficients h_i, whereas the quadratic coefficients J_{ij} correspond to coupling (interaction) strengths.**[2] μ is the strength of the external magnetic field. As Equation 2.2 is our objective function, we can adopt the following simplifications which won't change the position of the optimum x^*:

$$-J_{ij} \to J_{ij}$$
$$-\mu h_i \to h_i$$

Rendering the Hamiltonian into [13, 14]:

$$\boxed{H(\sigma) = \sum_{(ij)\in E(G)} J_{ij}\sigma_i\sigma_j + \sum_{i\in V(G)} h_i\sigma_i} \tag{2.3}$$

While the Ising model draws on spin values "up" and "down" ($\sigma_i \in \{-1, +1\}$), Quadratic Unconstrained Binary Optimization (QUBO) problems stem from computer science and

[2] We assume the Ising model to be ferromagnetic, i. e. $J_{ij} > 0$, but also allow $J_{ij} = 0$ when qubits i and j are not entangled (there is no interaction between them).

use values $x \in \{0, 1\}$ (true, false). In the **QUBO model**, we try to minimize the following energy function:

$$\boxed{\begin{aligned} H(x) &= \sum_{(ij)\in E(G)} q_{ij} x_i x_j + \sum_{i \in V(G)} q_{ii} x_i \\ &= x^T Q x \end{aligned}} \tag{2.4}$$

In the matrix notation Q is an upper-diagonal matrix with entries on the diagonal corresponding to the linear coefficients and nonzero off-diagonal terms corresponding to the quadratic coefficients. Note that we can easily express an Ising problem as QUBO problem using the identity $\sigma \to 2x - 1$. We prove this property in the appendix on page XVIII.

2.3. Solving problems on Quantum Annealers

D-Wave Systems Inc. (hereinafter referred to as *D-Wave*) is a quantum computing company producing commercially available quantum annealers debuting in 2011 [15]. Their QPU is made up of a lattice of tiny metal loops that get cooled down to the point where they become superconductors and exhibit quantum-mechanical effects. Each superconducting loop represents either a quantum bit (referred to as qubit) or a coupler entangling multiple qubits. The energy state of a qubit is biased by a programmable external magnetic field which alters the probability of a qubit "falling" into the classical state 0 or 1 when it collapses from the initial superposition state. We will refer to this quantity as *qubit bias*. Likewise, we can use the magnetic field to control coupling strengths. Qubit biases and coupling strengths together describe the energy landscape of the system and are introduced as problem Hamiltonian H_{final} over time (see section 2.1). Hence, when the user wants to solve a specific optimization problem, they need to encode it into a Hamiltonian function. [8]

2.3.1. Problem Reformulation

In order to set up our problem Hamiltonian (our objective function), we will make use of a binary quadratic model described in detail in section 2.2. This is since D-Wave's quantum annealers are constrained to these models by the way they are built with qubits

obeying the Ising model. To illustrate the steps of solving an optimization problem using D-Wave's quantum annealer, we introduce the Minimum Vertex Cover Problem (MVCP) on graphs.

The MVCP is defined as follows: Given an undirected graph $G = (V, E)$ with vertices V and edges E, a vertex cover is a subset of vertices such that every edge of the graph is incident to at least one vertex in that subset. The MVCP asks for a vertex cover with the smallest number of vertices. Let x_i be a binary variable on each vertex, which is 1 if vertex i is in the cover and 0 otherwise. As we want to use as few vertices as possible for the cover, our objective function is given by:

$$\text{Minimize } H = \sum_{i \in V} x_i \tag{2.5}$$

It is subject to the constraint that every edge is incident to at least one vertex in the cover:

$$\forall (i, j) \in E : x_i + x_j \geq 1 \tag{2.6}$$

However, in the QUBO format, no constraints are allowed, which is why we reformulate our problem, for example, we map it to a quadratic model. There are many different approaches to reformulation, some of which are described in-depth in chapter 3. To reformulate our constraint in Equation 2.6, we introduce a penalty term, which we design to equal zero for feasible solutions and some positive scalar for infeasible solutions [16]. For the MVCP, the penalty term becomes: [2]

$$\sum_{(i,j) \in E} (1 - x_i)(1 - x_j) \tag{2.7}$$

It will be 0 for every edge where at least one end is in the cover (either x_i or x_j is 1) and 1 if both incident vertices are *not* in the cover (x_i and x_j are zero). This allows us to write our objective function as follows:

$$\text{Minimize } H = \sum_{i \in V} x_i \quad + \quad \lambda \sum_{(i,j) \in E} (1 - x_i)(1 - x_j) \tag{2.8}$$

where λ is a positive, scalar *penalty value* used to weigh the penalty term. We are now able to write this equation, so that it bears resemblance to QUBO notation:

$$\begin{aligned}\text{Minimize } H &= \sum_{i \in V} x_i \quad + \quad \lambda \sum_{(i,j) \in E} (1 - x_j - x_i + x_i x_j) \\ &= \sum_{i \in V} x_i + \sum_{(i,j) \in E} \left(-\lambda x_i - \lambda x_j\right) + \sum_{(i,j) \in E} \left(\lambda x_i x_j\right) + m\lambda \\ &= \sum_{i \in V} \left((1 - \deg\, x_i) \cdot x_i\right) + \sum_{(i,j) \in E} \left(\lambda x_i x_j\right) + m\lambda\end{aligned}$$

where $m = |E|$. Note we can drop the additive constant $m\lambda$ as it has no impact on the optimum. This should clarify that Equation 2.8 can easily be written in QUBO notation for a specific instance of the MVCP.

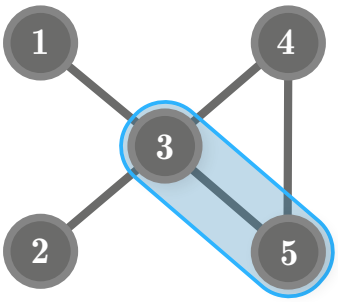


Fig. 2.1.: Sample graph for the MVCP.

As an example, we consider the graph depicted in Figure 2.1. The minimum vertex cover is already marked; it consists of vertices 3 and 5. The MVCP for this graph is modeled

as follows:

$$\begin{aligned}
\text{Minimize } H = {} & x_1 + x_2 + x_3 + x_4 + x_5 \\
& + \lambda(1 - x_1 - x_3 + x_1 x_3) \\
& + \lambda(1 - x_2 - x_3 + x_2 x_3) \\
& + \lambda(1 - x_3 - x_4 + x_3 x_4) \\
& + \lambda(1 - x_3 - x_5 + x_3 x_5) \\
& + \lambda(1 - x_4 - x_5 + x_4 x_5) \\
= {} & (1 - \lambda)x_1 + (1 - \lambda)x_2 + (1 - 4\lambda)x_3 + (1 - 2\lambda)x_4 + (1 - 2\lambda)x_5 \\
& + \lambda x_1 x_3 + \lambda x_2 x_3 + \lambda x_3 x_4 + \lambda x_3 x_5 + \lambda x_4 x_5 + 5\lambda
\end{aligned}$$

Dropping the additive constant 5λ, we can write down our QUBO model $\min x^T Q x$ with the upper-diagonal matrix Q given by:

$$\begin{pmatrix}
1-\lambda & 0 & \lambda & 0 & 0 \\
0 & 1-\lambda & \lambda & 0 & 0 \\
0 & 0 & 1-4\lambda & \lambda & \lambda \\
0 & 0 & 0 & 1-2\lambda & \lambda \\
0 & 0 & 0 & 0 & 1-2\lambda
\end{pmatrix}$$

2.3.2. Minor Embedding

After having formulated the problem in Ising or QUBO notation, we map our objective function to the QPU topology [8]. We will refer to this process as *minor embedding*. Let us represent the QPU topology by a graph $G = (V, E)$ with vertices V and edges E. First, we represent our objective function as graph of *logical* qubits as follows:

- Linear coefficients of the objective function (σ_i or q_{ii}) become nodes of the graph, which in turn stand for logical qubits in the QPU.
- Quadratic coefficients of the objective function (J_{ij} or q_{ij}) become weighted edges of the graph, which in turn stand for couplers and their coupling strengths in the QPU.

For the MVCP example, the graph representing the QUBO problem obtained by following the "mapping rules" will look exactly as our sample graph in Figure 2.1. This is the special case, where our underlying optimization problem itself refers to a graph. In general, however, we do not need to find a separate graph representation of our mathematical problem as long as we can transform it into a binary quadratic model, which we can always map to a graph as described above.

The actual *minor embedding* then refers to the process of mapping this graph of logical qubits to the physical qubits which are arranged in a graph as well representing the topology of the annealer's qubits. For D-Wave systems, this is either the Chimera or Pegasus topology discussed in section 2.5. To ease talking about the different graphs, we define:

Definition 2.1 (Input and hardware graph)**.**

- The input graph H is the undirected graph of the objective function given as a binary quadratic model. Its vertices represent *logical* qubits.
- The hardware graph G is the undirected graph of the quantum annealer's topology of *physical* qubits.

Super vertex placement

Multiple physical qubits in the hardware graph may represent the same logical qubit in the input graph, that is, the same variable of the problem Ising model. This is needed, so that we can map arbitrary binary quadratic models to the hardware graph. For example, notice that we cannot embed the input graph H for our MVCP in the Chimera graph $G_{Chimera}$ using just 5 physical qubits. This stems from the fact that H contains the fully connected graph K_3 which is not native to $G_{Chimera}$ as there is no way in the Chimera graph to connect three qubits in a closed loop. Instead, we "chain together" multiple physical qubits to represent one variable, e. g. x_4. With this approach, we can find a valid embedding shown in Figure 2.2, where the "chain" or *super vertex* representing x_4 is marked in red.

In practice, chains are realized by defining a strong coupling between the chained physical qubits, so that they take the same value at the optimal solution. Mathematically, chains are disjoint, connected subgraphs of G. An edge between two subgraphs indicates that two logical qubits are connected in the input graph H. In terms of graph theory, we can

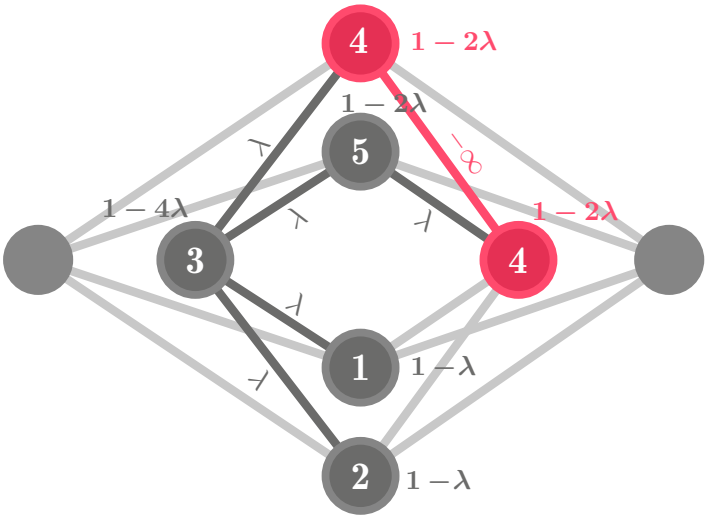


Fig. 2.2.: Embedding of the MVCP example QUBO.

assert these properties if and only if the input graph H (graph of our binary quadratic model) is a minor of the hardware graph G [8], which is equivalent to the requirement that H is embeddable in G (see Lemma 6 in [1]). Minor embedding is discussed formally in section 2.4. Briefly, we define a super vertex placement function ϕ that maps vertices of H to subsets of vertices of G. In D-Wave systems the subsets of vertices in the hardware graph are called "chains", while we will refer to them mainly as *super vertices* in this work.

The problem of finding a minor embedding for arbitrary input graphs and arbitrary hardware graphs is $\mathcal{NP}$-hard as discussed in [1]. While we do consider fixed topologies for the hardware graphs, not all physical qubits might be accessible for the embedding due to engineering limitations, i. e. not all qubits function as desired and are therefore deactivated. Moreover, after each calibration phase, *different* qubits might be inaccessible ("broken"), so the ideal hardware graph (Chimera etc.) differs from the actual hardware graph G presented to the user when embedding a problem. This is why "we are interested in [minor embedding] algorithms in which both G and H are part of the input" [6]. As in [1], we assume the embedding problem to remain relevant in the long term because, first, the number of "broken" qubits will probably decrease, but not reach 0, and second, a fully connected hardware topology is not yet in sight.

2.3.3. Summarized workflow

The following list summarizes the steps needed to solve a real-world problem on a quantum annealer.

1. First, state the problem to be solved, for example, "Which routers in my communication network are central in the sense that with those routers identified I can reach every router in the network?"
2. In the next step, the problem must be represented mathematically as an objective function, e. g. we can pose our question as MVCP.
3. If our objective function is not in the Ising or QUBO format, we have to reformulate it using reformulation techniques, e. g. express constraints with penalty terms. Our objective function is constructed, so that low energy states represent good solutions to our problem.
4. The objective function in Ising or QUBO format can then be represented by an input graph H, where linear coefficients correspond to nodes and quadratic coefficients to edges.
5. In order to solve the problem on the annealer's QPU, we need to map the input graph H to the hardware topology graph G by finding a valid minor embedding ϕ. One logical qubit might get mapped to multiple physical qubits.
6. During the actual annealing where qubits exhibit quantum effects, our problem objective function is introduced over time by altering the magnetic field accordingly, so that the energy landscape represents our problem. Vertices of the embedded graph correspond to qubit biases and edges to coupling strengths.
7. Finally, qubits collapse from a superposition state to a classical state. If the system stayed in the ground state throughout the entire time, this minimum of the resulting energy landscape corresponds to a solution to our problem. The annealing might be repeated multiple times to account for inaccuracies.

2.4. Graph minor

We define minor embedding similar to [6] and proceed in two stages as seen in [14] et al.

Definition 2.2 (Super Vertex Placement)**.** Let H, G be two graphs. A super vertex placement is a function $\phi : V(H) \to 2^{V(G)}$ that assigns each vertex of the input graph $i \in V(H)$ to a subset of vertices of the hardware graph $\phi(i) \subset V(G)$, such that:

(C1) $\forall i \in V(H) : \phi(i) \neq \varnothing$ and the subgraph induced by $\phi(i)$ in G is connected. We denote this induced subgraph by $G[\phi(i)]$.

(C2) $\forall i, j \in V(H)$ with $i \neq j : \phi(i) \cap \phi(j) = \varnothing$, that is, $\phi(i)$ and $\phi(j)$ are disjoint.

In the following, we call $\phi(i)$ the *super vertex* representing vertex i from the input graph H in the hardware graph G. By condition (C1), we ensure that super vertices are non-empty and connected, so that they can encode one spin from the input graph. As a single spin of the hardware cannot represent multiple spins of the input problem, condition (C2) must be met.

To account for couplings of the input problem, that is edges in the input graph, we must ensure respective edges between super vertices in the hardware graph. We therefore define the notion of a minor embedding:

Definition 2.3 (Minor Embedding)**.** Let H, G be two graphs. A super vertex placement ϕ is a minor embedding, if in addition to (C1) and (C2) we ensure:

(C3) $\forall (i, j) \in E(H) : \exists (u, v) \in \phi(i) \times \phi(j)$ with $(u, v) \in E(G)$.

This makes sure that edges $(i, j) \in E(H)$ are represented by at least one edge between the corresponding super vertices $\phi(i)$ and $\phi(j)$ in the hardware graph. In order for a super vertex placement to be a suitable hardware mapping encoding a QUBO problem, it must be a minor embedding, hence all three conditions must be met by our placement or *mapping* ϕ.

2.5. Hardware topologies

Due to the fact that the underlying meta heuristic, simulated annealing, can be implemented on various machines, different hardware topologies have emerged. We describe the Chimera, Pegasus and King's topology as hardware graphs in this section. As discussed previously, the vertices of the graphs represent physical bits of the annealer, while edges denote connections between bits (couplings). On quantum hardware, these bits, obviously, refer to qubits in the QPU. Notice that on quantum hardware, qubits and couplers of insufficient quality might be disabled, i. e. only a subgraph of the actual hardware graph can be utilized in the embedding process. Therefore, we only describe the ideal topologies here. Moreover, while we restrict ourselves to sparse topologies, here, there are machines based on digital bits will full connectivity. For obvious reasons, minor embedding on these machines is trivial.

2.5.1. Chimera graph

The Chimera graph $G_{Chimera}$ was the first topology for quantum annealers introduced and is used in the D-Wave 2000Q and earlier systems. It was first described in 2009 in [17]. A Chimera graph is made up of complete bipartite $K_{4,4}$ unit cells with four horizontal qubits coupled to four vertical qubits. Figure 2.3 visualizes the unit cell of a Chimera graph.

Unit cells are arranged in a grid-like structure. Horizontal vertices, i. e. vertices that lie in the horizontal partition of a unit cell rendered as cross, are connected to the horizontal vertices above and below the current cell. In an analogous manner, this applies to vertical vertices as well. Every vertex is therefore connected to at most 6 other vertices meaning that the graph is sparsely connected. Also note that the total graph remains bipartite as can be seen in figure 2.4. The newest of D-Wave's annealers deploying the Chimera topology (D-Wave 2000Q QPU) features a grid of 16×16 unit cells and therefore $16^2 \cdot 8 = 2048$ qubits.

2.5.2. Pegasus graph

A unit cell of a Pegasus graph $G_{Pegasus}$ is depicted in figure 2.3 where, in comparison to a $K_{4,4}$ Chimera unit cell, four edges are added. Thus, the Pegasus cell is no longer bipartite and even contains K_4 as a subgraph. Furthermore, layers are introduced (as third dimension) and more edges are established by connecting unit cells from different layers. This results in a qubit being connected to up to 15 other qubits (compared to a maximum vertex degree of 6 in Chimera graphs). The new D-Wave Advantage QPUs incorporates the Pegasus topology as lattice of 16×16 unit cells (where one unit cell consists of 24 qubits [8]) resulting in 5760 qubits in total [18]. Due to its better connectivity, the Pegasus topology is advantageous for minor embedding, especially for quadratization gadgets resulting in fewer auxiliary qubits [19]. As the visualization of Pegasus graphs is harder than that of the Chimera topology, we refer the reader to [18] for a complete formal description and to [20] for a plethora of different visualizations of this topology including an algorithm and open-source code to construct it. In the complexity proofs, the fact that the Pegasus graph contains the Chimera graph as a subgraph will prove quite useful.

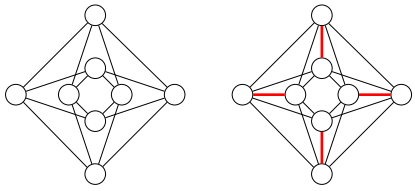


Fig. 2.3.: Unit cells of the Chimera graph (left) and the Pegasus graph (right) visualized. Red edges are the newly introduced one.

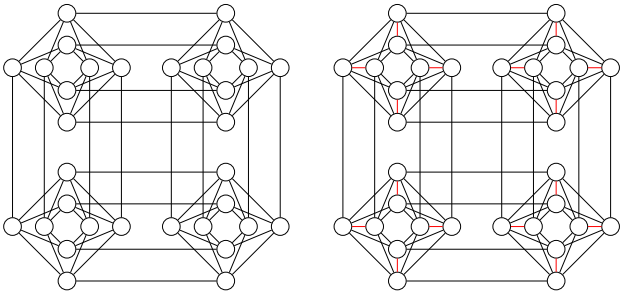


Fig. 2.4.: A 2×2 lattice of Chimera graph (left) and a subgraph of the Pegasus graph (right) with only one layer visualized.

2.5.3. King's graph

The King's graph G_{King} represents all legal moves of the king piece on a chessboard, i. e. each vertex of the graph constitutes a square on the chessboard and each edge is a legal move. For a King's graph of size $n \times m$, there are nm vertices. The King's graph for $m = n = 5$ is presented in Figure 2.5. While D-Wave's annealers currently do not employ the King's graph, it is adopted by Complementary Metal-Oxide Semiconductor (CMOS) annealing processors [21]. Notice that, in practice, CMOS devices usually do not suffer from dysfunctional parts in contrast to quantum hardware. When proving intractability of subproblems in chapters 4 and 5, however, for the sake of completeness, we extend the proofs to subgraphs of the King's graph as well.

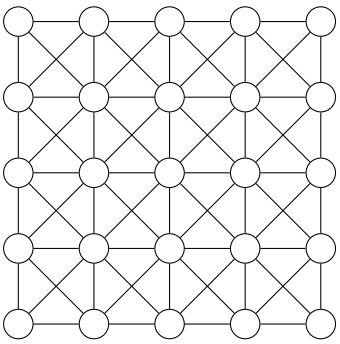


Fig. 2.5.: King's graph for $m = n = 5$

3. ILP and reformulation techniques

Many $\mathcal{NP}$-hard problems can easily be formulated as Integer Linear Programs (ILPs) and in fact, modelling the problem similar to an ILP with linear constraints and a linear objective is advantageous. However, that does not exclude a quadratic objective. In order to motivate this chapter, we want to enumerate some important advantages.

1. MIP presolving routines, as described in [22], can be applied that might reduce constraints' right-hand sides, fix variables, or reduce the range of variables. Note that there are presolving routines for QUBO problems as well that try to identify special properties of variables and quadratic terms, see [23, 24, 25].
2. Validating a solution to a QUBO given a set of constraints is quite simple and can be done generically.
3. Readjusting penalty coefficients in case of iterative solving routines is simpler and can be confined to constraints that are violated. Such a strategy is developed in [26]. Apart from that, there are problem-specific approaches, as [27] for the Travelling Salesman Problem (TSP).
4. Modelling using constraints is usually more intuitive.
5. The special structure of specific constraint types can be exploited to find potentially better embeddings, i. e. more efficient in terms of number of qubits needed. In [4, 5], such special constraint reformulation approaches are developed that essentially rely on encoding the constraint as a tiling problem. Additionally, in section 3.6 we propose a new constraint reformulation approach for multiple constraint types based on the divide and conquer paradigm.

Therefore, developing a reformulation routine to automatically generate a QUBO from a constrained model seems rather appealing. Hence, we will present integer linear programming, integer decomposition methods and generic constraint reformulation approaches. Based on those sections, we assess the difficulties one faces when formulating a QUBO to motivate constraint-specific reformulation techniques, one of which has not been presented in the literature, yet.

3.1. Integer Linear Programming

A generic ILP with $n \in \mathbb{N}$ variables and $m \in \mathbb{N}$ constraints can be formulated as follows

$$\begin{aligned} min\ & c^T x \\ s.t.\ & Ax \leq b \\ & x_i \geq 0, \qquad \forall i \in \mathbb{Z}_n \\ & x \in \mathbb{Z}^n \end{aligned}$$

where x is an n-dimensional decision variable vector of integers, $c \in \mathbb{R}^{n \times 1}$ and $b \in \mathbb{R}^{m \times 1}$ are constant vectors and $A \in \mathbb{R}^{m \times n}$ is a constant matrix. Additionally, we constrain every variable x_i to be bounded, i. e. to have finite-valued bounds. Note that this restriction does not render the problem computationally tractable as there are many $\mathcal{NP}$-hard problems, like the MVCP [28], that are typically encoded using binary variables only. However, when encoding integer variables as a sum of binary variables finite bounds are required.

Obviously, an ILP can have equality and greater-equal constraints as well. Both types can, however, trivially be expressed as lower-equal constraints. While a greater-equal constraint can just be multiplied by (-1), an equality constraint is simply composed of a lower-equal and a greater-equal constraint. This generic formulation is just for the purpose of simplicity. In the following, we will develop methods to reformulate generic equality and inequality constraints in section 3.3, but also some specific constraint types in sections 3.5 and 3.6.

3.2. Integer encodings

Due to QUBO being, by definition, restricted to binary decision variables, integers have to be encoded as a sum of binary decision variables. We will present three options to choose from when decomposing integers [29]. For the following encodings let $x \in \mathbb{Z}$ and $x \in [l, u]$ where $l, u \in \mathbb{Z}$ are constant finite-valued bounds and $u > l \geq 0$. Additionally, denote $r := u - l$ as the range of x. The following encodings have in common that they introduce s variables $y_i \in \mathbb{B}$, $i \in \mathbb{Z}_s$ with suitable coefficients c_i. Then, every occurrence of x is substituted with

$$x := l + \sum_{i=1}^{s} c_i y_i, \qquad c \in \mathbb{R}^s. \tag{3.9}$$

Binary expansion:
Binary expansion relies on the classical encoding of integers using powers of 2. More precisely, $s = \lceil log_2(r) \rceil$ variables are introduced. The coefficients are defined as $c_i := 2^{i-1}$ for $1 \leq i < s$ and $c_s := r - 2^{s-1} + 1$. Note that c_s essentially captures the remaining gap between the upper bound and the largest number representable with monomials $c_i x_i$, $i < s$. Example: For $x \in [2, 8]$, $r = 6$ and thus $s = 3$. The variable is substituted with $x := 2 + 1y_1 + 2y_2 + 3y_3$. Note that large coefficients arise from this encoding while the number of variables introduced is kept rather low.

Unary expansion:
Instead of encoding integers with powers of 2, unary expansion encodes, as the name implies, in unary. That is, $s = r$ and $c_i := 1, \forall i \in \mathbb{Z}_s$. The encoding is quite simple and obvious but creates a lot of variables while being beneficial numerically.

One-hot encoding:
Similarly, to unary expansion, $s = r$ as for every value of x, there is one variable created. The coefficients are, however, defined as $c_i := i$ and therefore grow rather large. Apart from that, a constraint has to be introduced that enforces that at most one variable is chosen to be 1. So the following constraint is introduced:

$$\sum_{i=1}^{s} y_i \leq 1 \tag{3.10}$$

3.3. Generic constraint reformulation

We will shortly introduce the fundamental ideas for reformulating generic constraints. The reader is referred to [16, 30] for further examples on how to formulate a QUBO. Generic constraint reformulation refers to simple reformulation techniques that are usually implicitly applied in QUBO formulations, like those in [2, 3]. These approaches do not exploit any special structure of the constraint which is why we refer to them as "generic." In the following, we assume that all decision variables are binary, i. e. decomposition of integers has already been undertaken.
Equality constraints of the form $\sum_{i=1}^{n} c_i x_i = b$ are quite simply reformulated using a sufficiently large penalty factor $\lambda > 0$, often referred to as Lagrangian multiplier. Essentially, the deviation of the left-hand side from the right-hand side is penalized by adding the following term to the QUBO objective function

$$\lambda \left(b - \sum_{i=1}^{n} c_i x_i \right)^2 . \tag{3.11}$$

The term between the brackets is 0 iff the constraint is not violated and $\neq 0$ otherwise. Squaring that deviation leads to a strictly positive value for a violated term which is penalized by multiplying by the Langrangian multiplier λ.
Inequality constraints are reduced to equality constraints by substituting the right-hand side of the constraint with a slack variable s such that the constraint can be viewed as an equality constraint whose deviation is penalized. More precisely, an inequality constraint $\sum_{i=1}^{n} c_i x_i \leq b$ is adjusted to

$$\sum_{i=1}^{n} c_i x_i = s, \qquad \min\{ \sum_{i=1}^{n} c_i x_i \,:\, x \in \mathbb{B}^n \} \leq s \leq b. \tag{3.12}$$

Note that this reduction might seem trivial at first but encoding s as a sum of binary decision variables can be problematic. Let $\mathcal{C} := \{ c_1, ..., c_n \}$ be the multiset of coefficients in the inequality constraint and denote $\mathcal{R} := \{ x = \sum_{c \in C} c \,:\, x \leq b, C \subseteq \mathcal{C} \}$ as the set of all valid settings of the left-hand side. Notice that, in general, $s := p + \sum_{i=1}^{k} a_i y_i$, ($y \in \mathbb{B}^k$, $a \in \mathbb{R}^k$, $p \in \mathbb{R}$) must be defined such that for every valid left-hand side $r \in \mathcal{R}$, there is a setting for y that satisfies $r = p + a^T y$. More importantly, however, the definition of s

must be constrained to $s \leq b$ as defined in equation 3.12. One approach to define s is to create a one-hot encoding, similar to the definition in section 3.2, that creates a variable for every element in $\mathcal{R}$ with a constraint to enforce that exactly one option is chosen. Unfortunately, the number of variables k created to encode s grows exponentially with n, that is $k \in \mathcal{O}(2^n)$. The following constraint illustrates that given some $n \in \mathbb{N}$

$$\sum_{i=1}^{n} 10^{-i} x_i \leq 1, \qquad x \in \mathbb{B}^n. \tag{3.13}$$

Clearly, every setting of $x \in \mathbb{B}^n$ is valid and sums to a distinct value. Due to the fact that there are 2^n distinct settings for $x \in \mathbb{B}^n$ that each contribute to $\mathcal{R}$ with a distinct element, $k = |R| \in \mathcal{O}(2^n)$. Apart from the growth in variables, the constraint for the one-hot encoding is very large.
An alternative approach would be to relax the requirement that for every $r \in \mathcal{R}$, there must be a variable setting s such that $s = r$ is relaxed to $s \approx_\tau r$, i. e.

$$\forall r \in \mathcal{R}\, \exists y \in \mathbb{B}^k, s = p + a^T y : |s - r| \leq \tau \tag{3.14}$$

for some fixed numeric tolerance $\tau > 0$. This could be achieved through a suitable fixed-point encoding of s. The drawback with this method is clearly the tolerance as slight deviations from constraints will be penalized. For a given Langrangian multiplier $\lambda > 0$ and a numeric tolerance τ that penalty can rise up to $\lambda \cdot \tau$ which can significantly degrade the solution quality.

Constraint	**Reformulation**
$x + y \leq 1$	$\lambda(xy)$
$x + y \geq 1$	$\lambda(1 - x - y + xy)$
$x - y \leq 0$	$\lambda(x - xy)$
$\sum_{i=1}^{k} x_i \leq 1$	$\lambda(\sum_{i=1}^{k} \sum_{j=i+1}^{k} x_i x_j)$

Table 3.1.: Simple constraints with $x, x_i, y \in \mathbb{B}$ that can easily be incorporated into the objective without introducing new slack variables [16].

For some specific constraints, [16] mentions formulations that don't need slack variables. These constraints are presented in table 3.1 and will be of greater interest in sections 3.5 and 3.6.

3.4. Difficulties of generic reformulation

Although, these generic reformulation techniques might be perceived as simple and fast and, thus, advantageous, there are some significant drawbacks that come along with them. Consequently, we will outline some major difficulties one has to consider when designing a QUBO. As implied, these problems do not exclusively play a role in reformulation but in general in the design of QUBOs. However, by explicitly stating the drawbacks, we want to motivate the following sections that try to avoid these classic pitfalls.

When reformulating an equality constraint of the form $\sum_{i=1}^{n} x_i = 1$ with $x_i \in \mathbb{B}$, this involves squaring the terms, as described in section 3.3. Squaring a sum of n variables (and a constant), in general, leads to $\mathcal{O}(n^2)$ quadratic terms. In fact, if we inspect these quadratic terms as an undirected graph, that is, with decision variables as vertex set and for each quadratic term (with non-zero coefficient) $x_i x_j$ as an edge $\{ x_i, x_j \}$ in the edge set, the graph is complete. A complete graph has $m = \frac{n(n-1)}{2}$ (for $n \geq 2$) edges as every vertex is connected to every other vertex. This is visualized in figure 3.1 with $n = 8$.

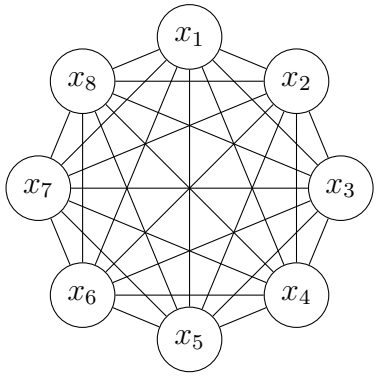


Fig. 3.1.: Complete graph on $n = 8$ vertices that arises from reformulating a simple constraint with $n = 8$ decision variables.

It should be obvious that such a dense graph is relatively difficult to embed as n grows. The sparsity of topology graphs like Chimera with maximum vertex degree $\Delta(G_{Chimera}) \leq 6$ [18], Pegasus with $\Delta(G_{Pegasus}) \leq 15$ [18] and the King's graph $\Delta(G_{King}) \leq 8$ [14], therefore heavily limits the embeddable size of problems with these constraint reformulations. In particular, inequality constraints are problematic as $s \geq 0$ slack variables are added to compensate for the constraint's slack. This, in general, leads to a complete graph on $n + s$ vertices and, thus, $\mathcal{O}((n + s)^2)$ quadratic terms. Note that, while usually not explicitly

stated as constraint reformulation procedures, the aforementioned representations of constraints are very common, see [2, 3] where Hamiltonians often have such structures.

Apart from that, multiplying large coefficients, trivially, leads to even larger coefficients which is problematic when running on quantum annealers as discussed in [31]. Large coefficients partially arise from Lagrangian multipliers which are commonly used to prevent constraints being violated. In general, large coefficients can degrade the performance of a quantum annealer due to its numeric imprecision. There are, however, methods to improve the precision or at least derive better results using raw data, see [31, 32].

3.5. Tiling-based reformulation

As discussed in section 3.4, constraint reformulations that rely on simply squaring the sum of terms, are problematic in terms of the growth of quadratic terms. [4, 5] proposed constructions to exploit the structure of specific constraints.

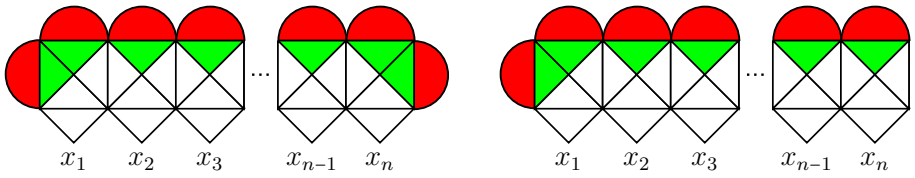


Fig. 3.2.: Construction from [4, 5] to encode specific constraints as colored tiling problems. Constructions for exactly-one constraints (left) and at-most-one constraints (right) each with n variables.

Figure 3.2 illustrates the general structure of the special constraint reformulation. Essentially, exactly-one and at-most-one constraints of the form $\sum_{i=1}^{n} x_i = 1$ and $\sum_{i=1}^{n} x_i \leq 1$, respectively, are encoded as colored tiling problems using n auxiliary variables. In such a colored tiling problem, colors represent assignments. We will define red as 1 and green as 0, and we will refer to a rectangle consisting of 4 triangles as gadgets. The goal is to color the construction from figure 3.2 such that for every side of each gadget the adjacent color is different from the triangle in the gadget that is next to it. This tiling problem can, in turn, be encoded using multiple small constraints that introduce a penalty for an invalid coloring. Note that due to the construction, the coloring of some gadgets is already partially determined and cannot be changed. In general, for the construction to make sense, we restrict a gadget coloring to be one those visualized in figure 3.3.

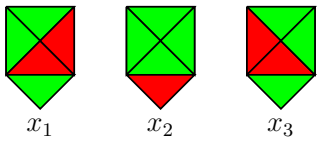


Fig. 3.3.: Gadget colorings are restricted to the three visualized. Variable values x_i are determined implicitly through the gadget coloring. Gadget above x_1 is colored oriented to the right, the gadget above x_2 is fully green and therefore enforces $x_2 = 1$ in a valid coloring and the gadget above x_3 is oriented to the left.

Using these restricted gadget colorings a valid coloring encodes the chosen exactly-one or at-most-one constraint as illustrated in figure 3.4.

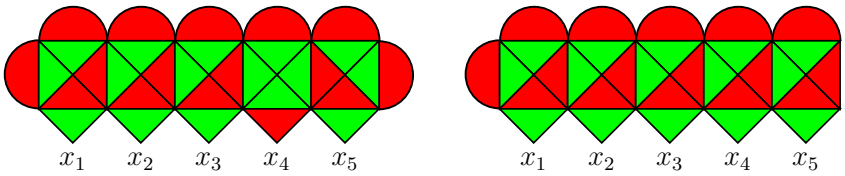


Fig. 3.4.: Valid colorings for constraints $\sum_{i=1}^{5} x_i = 1$ (left) and $\sum_{i=1}^{5} x_i \leq 1$ (right) encoded as colored tiling problems.

Encoding the colored tiling problem

For a constraint with n variables, n additional auxiliary variables $q_i \in \mathbb{B}$ are introduced where for some x_i the variable q_i represents the state of the gadget above. Regardless of the constraint type, the following constraints must be imposed on the variables:

$$q_i + x_i \leq 1, \qquad 2 \leq i \leq n \tag{3.15}$$

$$q_{i+1} + x_{i+1} - q_i = 0, \qquad 1 \leq i \leq n-1 \tag{3.16}$$

$$q_1 + x_1 = 1 \tag{3.17}$$

The semantics of constraints 3.15 - 3.17 follow from the interpretation of the q_i. We define $q_i = 1$ if the gadget is oriented to the right, $x_i = 1$ (and due to constraint 3.15 $q_i = 0$) represents a fully green gadget and $x_i = q_i = 0$ represents a coloring of the gadget to the left. This encoding of the gadget coloring leads to the constraints. Constraints 3.15 prevents $x_i = q_i = 1$ as there are only exactly 3 valid colorings and constraints 3.16 couple adjacent gadgets' colorings: q_i can only ever take the value 1 iff either its neighboring gadget on the right is oriented to the right or fully green, that is $q_{i+1} = 1$ or $x_{i+1} = 1$, respectively. Constraint 3.17 enforces that the leftmost gadget cannot be colored such that it is oriented to the left. Similarly, for exactly-one constraints, the rightmost gadget cannot be colored with an orientation to the right and therefore $q_n = 0$. Obviously, that leads to trivial constraints and q_n as well as q_{n-1} can be spared. Note that, while the reformulation introduces $\mathcal{O}(n)$ auxiliary variables, only $\mathcal{O}(n)$ instead of $\mathcal{O}(n^2)$ quadratic terms are generated. This can be beneficial since current quantum annealers have a sparse topology graph, see [18]. For more information on the construction, the reader is referred to [4, 5].

3.6. Divide-and-conquer based reformulation

In the following, we present new reformulation approaches for some specific constraint types. To the best of our knowledge, these reformulation techniques have not been mentioned anywhere yet. The overall reformulation approach is based on the divide-and-conquer paradigm. That is, a given constraint is decomposed into multiple smaller constraints whose results are captured in helper variables. The following types of constraints can be reformulated using our approach:

- Exactly-one constraints of the form: $\sum_{i=1}^{n} x_i = 1,\ x_i \in \mathbb{B}$
- At-least-one constraints of the form: $\sum_{i=1}^{n} x_i \geq 1,\ x_i \in \mathbb{B}$
- At-most-one constraints of the form: $\sum_{i=1}^{n} x_i \leq 1,\ x_i \in \mathbb{B}$
- Absorption constraints of the form: $\sum_{i=1}^{n} x_i = z,\ x_i, z \in \mathbb{B}$. Note that, in the following, we will call z the "absorption" variable.

Similarly to the construction from [4, 5] in section 3.5, the goal of such a constraint-specific reformulation is to keep the number of needed connections low. We will show that the number of quadratic terms introduced remains linear in the number of variables n involved in the original constraint. Apart from that, these constraint reformulation approaches can be exploited in the embedding algorithm.
Figure 3.5 visualizes the general approach for the first three types of constraints. The idea is to condense the information of two variables into one by introducing a coupling constraint. That is done up to the root. For all these types of constraints, the root does not introduce an extra variable since an additional constraint suffices.

Although, the structure depicted in figure 3.5 can be generalized to trees with internal vertices with degree greater than 2, we will for the sake of simplicity constrain ourselves to binary trees.
Apart from that, in the following, we will assume that n, the number of binary variables x_i in the left-hand side of a constraint (as formulated above), is a power of 2. That is, $n = 2^k,\ k \in \mathbb{N}$. If that is not the case, there are some minor adjustments needed to properly reformulate the constraint.

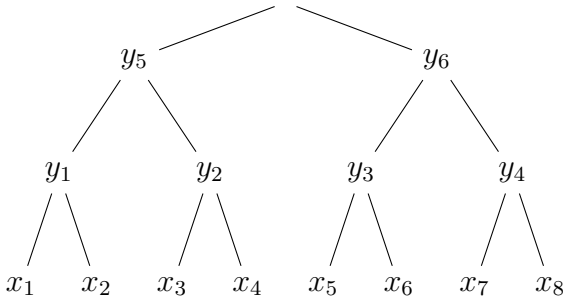


Fig. 3.5.: General approach to divide-and-conquer-based reformulation techniques for specific constraint types. Variable x_i are variables in the original constraint, while y_i are newly introduced for the purpose of reformulating the constraint.

3.6.1. Coupling constraints

Coupling constraints are meant to be constraints that condense the information of child vertices of an internal vertex into that vertex. These different coupling constraints will now be explained.

Except for at-least-one constraints, the coupling constraints are always the same. For a given internal vertex y_j and its two children y_i and y_{i+1}, a constraint of the form

$$y_i + y_{i+1} - y_j = 0, \qquad y_i, y_{i+1}, y_j \in \mathbb{B} \tag{3.18}$$

is introduced. Note that we do not distinguish between the child vertices being internal vertices or leaves. The constraint above can be interpreted as $y_j = 1$ if there is exactly one leaf $x_i = 1$ in the subtree underneath y_j. And $y_j = 0$ if all leaves in the subtree below y_j are 0. Now, let y_i and y_{i+1} be the child vertices of the root and let $s = y_i + y_{i+1}$. If $s = 1$, there is exactly one leaf in the tree that has value 1 and, similarly, if $s = 0$, all leaves of the tree are set to 0. These two cases are of interest. If $s = 2$, both in the left and in the right subtree, there is exactly one leaf with value 1 which does not help in the reformulation of the constraints above.

In an exactly-one constraint an additional constraint is introduced to enforce that $s = 1$:

$$y_i + y_{i+1} = 1 \tag{3.19}$$

Similarly, for at-most-one constraints, the additional constraint is

$$y_i + y_{i+1} \leq 1. \tag{3.20}$$

And for absorption constraints the constraint for the root is the same as for all other internal vertices. Let $z \in \mathbb{B}$ be the absorption variable of the constraint. Then, the following constraint is introduced:

$$y_i + y_{i+1} = z \tag{3.21}$$

At-least-one constraints are special in that the coupling constraints are quite different. For the lowest layer, where y_j is an internal vertex and x_i and x_{i+1} are leaves, we introduce a constraint of the form

$$y_j = \overline{x_i}\overline{x_{i+1}} = (1 - x_i)(1 - x_{i+1}). \tag{3.22}$$

These constraints can be reformulated using Rosenberg polynomials, see B for details. Note that $y_j = 1$ iff both x_i and x_{i+1} are 0 and $y_j = 0$ otherwise. For higher layers, where y_j is an internal vertex and its children y_i and y_{i+1} are internal vertices as well, the constraint is only slightly different:

$$y_j = y_i y_{i+1} \tag{3.23}$$

For any internal vertex y_j of the tree, $y_j = 1$ if all leaves of the subtree below y_j are 0. For an at-most-one constraint, in the root with child vertices y_i and y_{i+1}, the following constraint is introduced

$$y_i + y_{i+1} \geq 1 \tag{3.24}$$

Note that due to the fact that internal vertices are always defined to be the product of their children, in the reformulation of at-least-one constraints, it is not trivial to extend the definition to trees with higher degrees.
Regarding the case that n is not a power of 2, at-least-one constraints are slightly more cumbersome in that in the construction above internal vertices y_i are 1 if there is no leaf with value 1 and $y_i = 0$ otherwise. Apart from that, the reformulation of at-least-one constraints is essentially a special case of the Rosenberg reformulation of higher-order terms.

3.6.2. Complexity

In general, it is obvious that the number of variables introduced remains linear in the size of the underlying constraint. In particular, as figure 3.5 suggests, if n is a power of 2, exactly $n-1$ new variables are introduced. This is similar to the approach discussed in section 3.5. Another similarity is the number of edges introduced as both approaches introduce $\mathcal{O}(n)$ edges instead of $\mathcal{O}(n^2)$. As discussed, for large n, this might be advantageous in the embedding process. The main difference is that a bit flip in a variable x_i takes at most $\mathcal{O}(log\, n)$ flips to propagate compared to the tiling-based reformulation with $\mathcal{O}(n)$ bit flips in the worst case. Note that without the reformulation the "propagation delay" is constant, i. e. in $\mathcal{O}(1)$.

3.6.3. Examples

In [2, 3], for a lot of $\mathcal{NP}$-hard problems QUBO formulations are presented. Among them are the TSP and the exact cover problem that both contain terms of the form

$$\left(1 - \sum_{i \in I} x_i\right)^2 \tag{3.25}$$

which correspond to exactly-one constraints on the variables x_i. The same applies to the Set Cover Problem (SCP) which we will be defined in section 4.2. Apart from that, multiple tree problems such as the Steiner Tree Problem (STP), the undirected feedback vertex/edge set problems and an $\mathcal{NP}$-hard variant of the minimum spanning tree problem contain absorption constraints reformulated to

$$\left(z - \sum_{i \in I} x_i\right)^2. \tag{3.26}$$

Additionally, at-most-one constraints can be needed in an adjusted formulation of the TSP where all edge weights are shifted to ≤ 0 and relaxing the exactly-one constraints to at-most-one constraints. This could be numerically beneficial for instances with similar edge weights.

4. Super Vertex Emplacement Problem

We will refer to the subproblem of iteratively adding a new Super Vertex Emplacement (SVE) for an unmapped vertex as the SVEP, and we shall define it formally. Such a formal definition enables us to argue that the SVEP can quite naturally be encoded as a Connected Set Cover Problem (CSCP), prove that it remains computationally intractable on subgraphs of the hardware topology graphs presented in section 2.5 and develop heuristics for it.

4.1. Formal definition of the SVEP

Definition 4.1 (Super Vertex Emplacement Problem)**.** Let H and G be two graphs and $\phi : V(H) \to 2^{V(G)}$ be a partial SVE, that is, for some $v \in V(H)$, $\phi(v) = \varnothing$. For such a vertex $v \in V(H)$, find a super vertex $\mathcal{M} \subseteq V(G)$ with $\mathcal{M} \neq \varnothing$ such that

($M1$) $G[\mathcal{M}]$ is connected.

($M2$) $\forall u \in V(H) : \phi(u) \cap \mathcal{M} = \varnothing$

($M3$) $\forall u \in N_H(v), \phi(u) \neq \varnothing, \exists e \in E(G) : e \in \phi(u) \times \mathcal{M}$

Given $k \in \mathbb{N}$, the decision version of the SVEP, asks whether there is such an $\mathcal{M}$ with $|\mathcal{M}| \leq k$.

The decision version of the SVEP is in $\mathcal{NP}$ since verifying a solution $\mathcal{M}$ just implies verifying that it satisfies conditions ($M1$) through ($M3$). The first can be achieved using a Depth-First Search (DFS) and the rest is trivial. Apart from that, successfully solving the SVEP $|V(H)|$ times such that every vertex has a non-empty super vertex yields a valid minor embedding as defined in section 2.4. ($C1$) is trivially true since the SVEP ensures that $G[\mathcal{M}]$ is connected and non-empty. ($C2$) and ($C3$) simply follow from ($M2$) and ($M3$), respectively. Note that, obviously, successively solving the SVEP might in general not yield an optimal solution to the minor embedding problem. This approach is, however, an elegant way of decomposing the problem into multiple subproblems to solve and has been successfully applied in [6].

4.2. The SVEP is a CSCP

It is apparent that one can identify multiple subproblems in the embedding problem. One of them, the SVEP, is essentially a Weighted Connected Set Cover Problem (WCSCP) which we will define in this section.

Definition 4.2 (Set Cover Problem)**.** Given a finite set $\mathcal{U}$ and a family $\mathcal{S}$ of subsets of $\mathcal{U}$, i. e. $\mathcal{S} \subseteq 2^{\mathcal{U}}$, find a minimum subset $\mathcal{C} \subseteq \mathcal{S}$ that satisfies

$$\mathcal{U} = \bigcup_{S \in \mathcal{C}} S. \tag{4.27}$$

The decision problem of the SCP is, given some number $k \in \mathbb{N}$, whether there is such a set $\mathcal{C}$ with $|\mathcal{C}| \leq k$.

The SCP is quite well studied [33, 34] and is known to be $\mathcal{NP}$-complete. In fact, it is among Karp's 21 $\mathcal{NP}$-complete problems [35]. The SCP has multiple generalizations among which we will focus on the Weighted Set Cover Problem (WSCP) and the CSCP.

Definition 4.3 (Weighted Set Cover Problem)**.** Similar to the SCP, given sets $\mathcal{U}$ and $\mathcal{S} \subseteq 2^{\mathcal{U}}$ find a set cover $\mathcal{C} \subseteq \mathcal{S}$ with $\mathcal{U} = \bigcup_{S \in \mathcal{C}} S$. Additionally, given a function $c : \mathcal{S} \to \mathbb{R}_+$ out of the valid set covers find a cover with minimum weight. That is,

$$\min \sum_{S \in \mathcal{C}} c(S). \tag{4.28}$$

It is obvious that the WSCP is a generalization of the SCP since defining $c : S \mapsto 1, \forall S \in \mathcal{S}$ and supplying the input of the SCP $(\mathcal{U}, \mathcal{S})$ to the WSCP as $(\mathcal{U}, \mathcal{S}, c)$ yields the optimal value for the SCP. The CSCP is another generalization.

Definition 4.4 (Connected Set Cover Problem)**.** Let $\mathcal{U}$ be a universe of elements, $\mathcal{S} \subseteq 2^{\mathcal{U}}$ and $E \subseteq \mathcal{S} \times \mathcal{S}$ a set of edges. Additionally, let $G = (\mathcal{S}, E)$ be the undirected simple graph with vertices in $\mathcal{S}$. Note that $\mathcal{S}$ is extended to a multiset. Find a minimum set cover $\mathcal{C}$ such that the vertex-induced subgraph $G[\mathcal{C}]$ is connected. Similarly to the SCP, the decision version of the problem is to decide whether there is a cover $\mathcal{C}$ with size $|\mathcal{C}| \leq k$ given $k \in \mathbb{N}$.

Again, it is trivial to show that the CSCP is a generalization of the SCP as the graph G can be defined to be isomorphic to the complete graph $K_{|\mathcal{S}|}$ by setting $E \coloneqq \{\{u, v\} : u \neq v, u, v \in \mathcal{S}\}$. That is valid due to the fact that every non-trivial vertex-induced subgraph of a complete graph is again connected. Finally, let us define the WCSCP.

Definition 4.5 (Weighted Connected Set Cover Problem)**.** Given a tuple $(\mathcal{U}, \mathcal{S}, E, c)$ with $\mathcal{S} \subseteq 2^{\mathcal{U}}$, $E \subseteq \mathcal{S} \times \mathcal{S}$ and $c : \mathcal{S} \to \mathbb{R}_+$ find a connected set cover $\mathcal{C}$ on $(\mathcal{U}, \mathcal{S}, E)$ that minimizes $\sum_{S \in \mathcal{C}} c(S)$.

One might notice that, as mentioned above, the SVEP can quite naturally be encoded as an instance of the WCSCP. Loosely speaking, a super vertex must be connected, some vertices might have higher cost than others due to other super vertices and a super vertex must be adjacent to others.

Claim 4.1. *The SVEP is a WCSCP.*

We will formalize such a mapping from the SVEP to the WCSCP. As defined in section 4.1, given two graphs H and G, a partial placement $\phi : V(H) \to 2^{V(G)}$ and an unmapped vertex $v \in V(H)$, we seek to find a mapping $\mathcal{M}$ satisfying definition 4.1. Additionally, we define $R(w) \coloneqq \{u \in V(H) : w \in \phi(u)\}$ for $w \in V(G)$ to be the reverse mapping. Obviously, for a valid embedding, $|R(w)| \leq 1, \forall w \in V(G)$.

Let us now define the input $(\mathcal{U}, \mathcal{S}, E, c)$ to the WCSCP given an instance (H, G, v, ϕ) to the SVEP. Define $F_u \subseteq V(H)$ with $u \in V(G)$ as the set of all vertices in H that are adjacent to v and are mapped to a vertex in G that is either u itself or adjacent to u. More formally,

$$F_u \coloneqq (\{w \in V(H) : x \in N_G(u), x \in \phi(w)\} \cup R(u)) \cap N_H(v). \tag{4.29}$$

Therefore, we set $\mathcal{S} \coloneqq \{F_u : u \in V(G)\}$ and trivially $\mathcal{U} \coloneqq \bigcup_{F_u \in \mathcal{S}} F_u$. Note that setting $\mathcal{U}$ to the neighbors of v in G is in general not valid since some neighbors of v might not have been embedded yet. The edge set E is defined using the graph H. That is, $E \coloneqq \{\{F_x, F_y\} : xy \in E(G)\}$. And $c(F_u) \coloneqq \lambda \cdot |R(u)| + 1$ with sufficiently large $\lambda > 1$. A solution to the WCSCP would then be a solution to the SVEP iff the cost of the solution $\mathcal{C}$ to the instance of the WCSCP is at most $|V(G)|$ with $\lambda \coloneqq |V(G)|$. □

In practice, however, and in section 4.5 in particular, we seek for optimizing the WCSCP potentially still accepting overlapping super vertices at this stage. Regarding approximability, we want to slightly adjust the reduction and instead reduce to the CSCP, thereby, constraining $\mathcal{S}$ to vertices $u \in V(G)$ that are unmapped, i. e. with $R(u) = \varnothing$.

Thus, we instead define $\mathcal{S} := \{ F_u : u \in V(G), R(u) = \varnothing \}$, and $\mathcal{U} := \{ u \in V(H) : u \in N_H(v), \phi(u) \neq \varnothing \}$. The edge set E must be confined to unmapped vertices in G. Therefore, we now define $E := \{ \{ F_x, F_y \} : R(x) = R(y) = \varnothing, xy \in E(G) \}$. A solution $\mathcal{C}$ to the CSCP would directly correspond to a solution to the instance of the SVEP as for any $F_u \in \mathcal{C} \iff u \in \mathcal{M}$. This reduction will be further analyzed in section 4.4.

Apart from the SVEP, which is solved iteratively, a super vertex replacement problem could be encoded as a WCSCP as well. Such a replacement problem would remove a given SVE for some vertex $v \in H$ and define $\mathcal{S}$ not on all F_u, $u \in G$ but rather for all $u \in \phi(v)$, i. e. such that the goal is to reduce the placement. Alternatively, a sophisticated selection of vertices around $\phi(v)$ (including $\phi(v)$ itself) could be chosen to enable more improvements. Such an approach, i. e. limiting $\mathcal{S}$ to "useful" vertices, could significantly reduce the search space. We will present a heuristic attempting a reduction of an existing SVE in section 4.6. Another heuristic improving an initial solution will be presented in section 4.7 and is based on an evolutionary approach.

4.3. The SVEP is $\mathcal{NP}$-hard

As proven by [1], the embedding problem on Chimera graphs with missing vertices and missing edges is $\mathcal{NP}$-hard. Due to the subgraph relation between Chimera and the Pegasus graph, discussed in section 2.5, it is clear that the embedding problem on Pegasus graphs of arbitrary size remains computationally intractable, that is, $\mathcal{NP}$-hard. We will devote this section to proving that the SVEP on Chimera graphs of arbitrary size with missing vertices and missing edges as defined in section 4.1 is $\mathcal{NP}$-hard as well. And similarly to [1], we will then exploit the subgraph relation between the Chimera and the Pegasus graph to extend the proof to Pegasus graphs.

Claim 4.2. *The SVEP on Chimera graphs of arbitrary size with missing vertices and missing edges is $\mathcal{NP}$-hard.*

In order to prove the claim, we shall reduce the SCP to the SVEP. Let $(\mathcal{U}, \mathcal{S}, k)$ be the input to the SCP where $\mathcal{U}$ is a set of elements, $\mathcal{S} \subseteq 2^{\mathcal{U}}$ are subsets to choose from and $k \in \mathbb{N}$ just as in definition 4.2. Furthermore, define $m := |\mathcal{S}|$, $n := |\mathcal{U}|$. Additionally, let t be a constant introduced when translating the graph construction to the hardware graph and $p \in \mathbb{N}$ be sufficiently large. In the case of Chimera graphs, $t = 1$. And say, the decision version of the SVEP was to decide whether there is a SVE of size $\phi \leq q$. We will now construct a graph for which a SVE of size ϕ corresponds to a set cover of size $\lfloor \frac{\phi}{tp} \rfloor$. The construction is visualized in figure 4.1.

There are multiple aspects to note regarding the construction. Most importantly, the tree construction in 4.1 is only a visualization of the proof idea and serves to explain the resulting construction.
We define red vertices to be "essential" vertices. Such a vertex v must be included in any SVE which can be achieved by choosing an adjacent vertex w that is defined to be part of an adjacent super vertex. Additionally, all edges of w except for vw are removed. These essential vertices will be further defined.
Wavy edges between two vertices u and v are defined to be paths with length p, that is, u and v are connected in any SVE only through a path of p vertices. Below the blue-colored subtrees, a grid, containing for each element $i \in \mathcal{U}$ a super vertex, is placed. We define that the SVE must be connected to each of these super vertices.
The construction essentially enforces that the red-colored tree is chosen as part of the

SVE. The tree has m leaves which are labelled S_1 to S_m indicating that the subtree below S_i encodes the set $S_i \in \mathcal{S}$. For the sake of simplicity, every blue tree has n leaves. Such a blue-colored leaf $v_{i,j}$ is connected to the super vertex representing element $j \in \mathcal{U}$. If $j \notin S_i$, then the edge connecting $v_{i,j}$ and the super vertex for element j is removed.

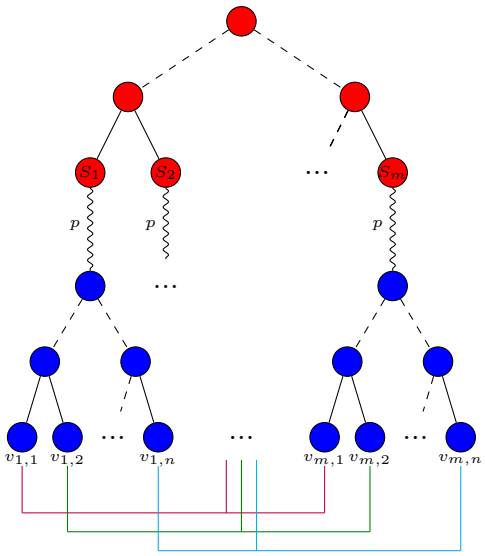


Fig. 4.1.: Graph construction to prove that the SVEP is in general $\mathcal{NP}$-hard. Red vertices are called "essential" vertices, blue vertices will be called leaves. The grid underneath are adjacent super vertices that must be connected to the resulting SVE.

Since the SVE must be connected to each of the super vertices of elements $j \in \mathcal{U}$, for every $j \in \mathcal{U}$ the SVE must contain at least one vertex $v_{i,j}$ with $i \in \mathbb{Z}_m$. That is essentially equivalent to choosing set S_i. However, choosing such a vertex $v_{i,j}$ implies that it must be connected to the red-colored tree, that is, there must be a path from $v_{i,j}$ to the red-colored vertex labelled S_i. This path has, by construction, length at least p. Therefore, if at least one element from a set S_i is chosen, the SVE grows in size by at least p. We defined $p \in \mathbb{N}$ to be "sufficiently large". More precisely, p must be larger than the number of blue and red vertices in the construction. Then, the size of the resulting SVE will be dominated by the wavy paths taken since the size ϕ can be written as

$$\phi = \kappa \cdot t \cdot p + \omega, \quad 0 < \omega < p \tag{4.30}$$

where κ is the number of sets chosen and ω the number of red and blue vertices part of the SVE. Trivially, $\lfloor \frac{\phi}{tp} \rfloor = \kappa$. And therefore, in order to decide the instance of the

SCP $(\mathcal{U}, \mathcal{S}, k)$, it suffices to construct a graph as shown above and decide whether there is some SVE of size $\phi \leq (k+1) \cdot tp - 1$ on that graph. The latter is then equivalent to deciding the original instance of the SCP.

We will now adjust the construction from figure 4.1 to be a grid, and then we will show the different gadgets needed to encode such a grid on a Chimera graph. The adjusted construction is visualized in figure 4.2.

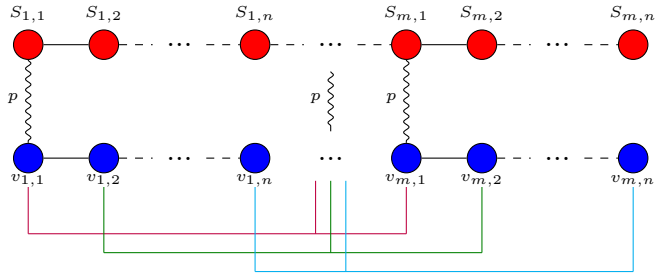


Fig. 4.2.: Adjusted construction that can be encoded on a grid structure.

The proof idea remains the same with only the tree structures being flattened. In figure 4.2, the highest layer consists of essential vertices. This adjusted construction needs $m \cdot n$ of those to enable the lower level to be easily encoded on the grid. Again, each of the vertices $S_{i,1}$ connects to the lower layer through a path of length p. For all $i \in \mathbb{Z}_m$, the $v_{i,j}$ (with $j \in \mathbb{Z}_n$) are only connected to the upper layer through the path of length p from $v_{i,1}$ to $S_{i,1}$. Therefore, again, choosing at least one vertex $v_{i,j}$ with the corresponding element in the set S_i corresponds to a growth of the size of the SVE by at least p. We will now present multiple, trivial gadgets needed to encode the grid structure onto a Chimera graph. All these structures need a constant amount of Chimera cells. Thus, it is clear that the width of the grid in figure 4.2 is in $\mathcal{O}(m \cdot n)$ and the depth is in $\mathcal{O}(n+p)$. Apart from that, the number of adjacent super vertices is $mn + n$ since there are mn essential vertices and n super vertices for the elements in $\mathcal{U}$. Hence, if p is polynomial in m and n, the reduction is indeed a polynomial reduction. In fact, since there are at most $4mn$ red and blue vertices, it would suffice to define $p := 8mn + 2$ which is indeed polynomial in m and n.

We need to define multiple gadgets for the Chimera graph:

(i) "Essential" vertex

(ii) Splitting/Wire gadget

(iii) Crossing gadget

These gadgets are presented in figure 4.3.

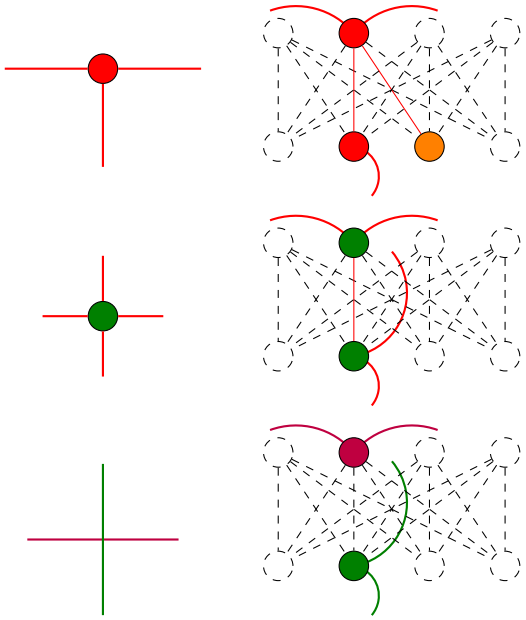


Fig. 4.3.: All three gadget constructions (upmost: "essential" vertex; middle: splitting/wire gadget; lower: crossing gadget) can trivially be encoded in a Chimera graph. Dashed edges and edges not indicated are not part of the graph, neither are dashed vertices. The "splitting" gadget can be reduced to a wire or a T-junction. Thick edges are outgoing edges while thinner ones define gadget-internal edges.

Thus, we have proven that the construction in figure 4.2 can be used to decide an instance of the SCP by solving the SVEP, that the reduction is polynomial and that the construction can be encoded on a Chimera graph. This completes the proof. □

Claim 4.3. *The SVEP on Pegasus graphs of arbitrary size with missing vertices and edges is $\mathcal{NP}$-hard.*

This follows directly from the claim 4.2 and the subgraph relation between Pegasus and Chimera graphs. □

Claim 4.4. *The SVEP on King graphs of arbitrary size with missing vertices and edges is $\mathcal{NP}$-hard.*

It suffices to show that the gadgets used in construction 4.2 can be encoded in a King's graph. Therefore, similar to 4.3, in 4.4 the gadgets are listed.

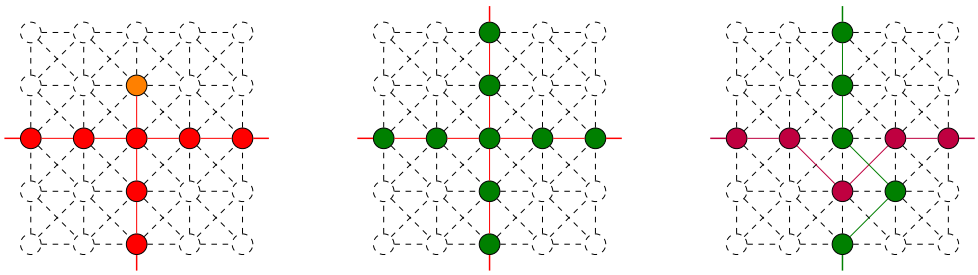


Fig. 4.4.: All three gadget constructions for the King's graph (left: "essential" vertex; middle: splitting/wire gadget; right: crossing gadget) are quite trivial since the King's graph itself is a grid.

Defining p is quite trivial as well with $p := 25 \cdot 2mn + 2$. In this case, as a vertical wire takes exactly 5 vertices due to the definition of the gadgets, $t := 5$. Again, this is polynomial in m and n and thus, the reduction is a valid polynomial-time reduction from the SCP to the SVEP. This completes the proof. □

As argued in section 4.1, the SVEP is in $\mathcal{NP}$ and, thus, the decision version of the SVEP is $\mathcal{NP}$-complete.
It is important to note that while claim 4.4 has no practical relevance since King graphs are only used on CMOS devices and was shown only for the sake completeness, claims 4.2 and 4.3 can be applied to actual quantum devices. Hence, these results motivate the use of heuristics since algorithms solving the SVEP optimally will very likely have a bad runtime.

4.4. Approximability of the SVEP

In [36], a simple algorithm to approximate the CSCP is presented. We want to shortly present that approach. Using an approximation algorithm $\mathcal{A}$ for the SCP and an approximation algorithm $\mathcal{B}$ for the STP, the CSCP can be approximated. The STP asks given an undirected simple graph G and a set $\mathcal{T} \subseteq V(G)$ for a set of edges $\mathcal{X}$ such that the subgraph induced by $\mathcal{X}$ is a spanning tree containing $\mathcal{T}$. The set $\mathcal{T}$ is usually referred to as the set of terminals that have to be connected, and the objective is to minimize $|\mathcal{X}|$. Just as the SCP, the STP is among Karp's 21 $\mathcal{NP}$-complete problems [35]. Let $(\mathcal{U}, \mathcal{S}, E)$ be an instance to the CSCP. The approximation algorithm one can derive from $\mathcal{A}$ and $\mathcal{B}$ simply finds a solution $\mathcal{C}$ to the SCP on $(\mathcal{U}, \mathcal{S})$. Then $\mathcal{B}$ is used to find a Steiner tree $\mathcal{X}$ on the graph $G = (\mathcal{S}, E)$ with terminal set $\mathcal{T} := \mathcal{C}$ minimizing $|\mathcal{X}|$. Note that due to $\mathcal{C} \subseteq V(G[\mathcal{X}])$, $V(G[\mathcal{X}])$ is a set cover and, additionally, it must be connected since it is a Steiner tree. Therefore, $V(G[\mathcal{X}])$ is a solution to the CSCP.

In section 4.2, we presented a reduction from the SVEP to the WCSCP and then adjusted it to the CSCP. The latter shall now be further analyzed.

Definition 4.6 (S-reduction)**.** Let $\mathcal{A}$ and $\mathcal{B}$ be two $\mathcal{NP}$-hard optimization problems with problem spaces $\mathcal{I}_\mathcal{A}, \mathcal{I}_\mathcal{B}$ and solution spaces $\mathcal{S}_\mathcal{A}, \mathcal{S}_\mathcal{B}$ for instances of problems $\mathcal{A}$ and $\mathcal{B}$. Additionally, define two functions f and g where $f : \mathcal{I}_\mathcal{A} \to \mathcal{I}_\mathcal{B}$ reduces an instance of problem $\mathcal{A}$ to an instance of problem $\mathcal{B}$ and $g : \mathcal{S}_\mathcal{B} \times \mathcal{I}_\mathcal{A} \to \mathcal{S}_\mathcal{A}$ translates a solution for problem $\mathcal{B}$ given the original instance of problem $\mathcal{A}$ to a solution for that instance. We define $OPT_\mathcal{A} : \mathcal{I}_\mathcal{A} \to \mathbb{R}$ and $OPT_\mathcal{B} : \mathcal{I}_\mathcal{B} \to \mathbb{R}$ as functions returning optimal solution values to instances of problems $\mathcal{A}$ and $\mathcal{B}$, respectively. And $\mathcal{M}_\mathcal{A} : \mathcal{S}_\mathcal{A} \to \mathbb{R}$ and $\mathcal{M}_\mathcal{B} : \mathcal{S}_\mathcal{B} \to \mathbb{R}$ the cost functions for problems $\mathcal{A}, \mathcal{B}$, respectively. An S-reduction satisfies properties $(S1)$ through $(S3)$ [37].

$(S1)$ f and g are polynomial time computable functions.

$(S2)$ $\forall x \in \mathcal{I}_\mathcal{A} : OPT_\mathcal{A}(x) = OPT_\mathcal{B}(f(x))$

$(S3)$ $\forall x \in \mathcal{I}_\mathcal{A} \forall y \in \mathcal{S}_\mathcal{B}(f(x)) : \mathcal{M}_\mathcal{A}(x, g(x, y)) = \mathcal{M}_\mathcal{B}(f(x), y)$

Claim 4.5. *The reduction from the SVEP to the CSCP satisfies definition 4.6 and is therefore an S-reduction.*

Proof. Clearly, f and g are polynomial time computable and, therefore, $(S1)$ is satisfied. Regarding $(S2)$, suppose there was some instance x to the SVEP for which the optimal value was not equal to the optimal value of the generated instance $f(x)$ to the CSCP. If $OPT_{\mathcal{A}}(x) < OPT_{\mathcal{B}}(f(x))$, this would obviously imply that the mapping $\mathcal{M}$ has size less than the cover $\mathcal{C}$. When introducing the reduction, however, we argued that vertices in the instance x correspond a vertex F_u in the instance $f(x)$ and, therefore, an optimal solution to x could just be translated to an optimal solution of $f(x)$. The same applies for the case $OPT_{\mathcal{A}}(x) > OPT_{\mathcal{B}}(f(x))$. Together, that contradicts the assumption. Thus, $(S2)$ is satisfied.
$(S3)$ is just as straightforward as in no solution to the instance $f(x)$, there is any additional cost introduced when translating the solution because of the direct correspondence of vertices in the instances x and $f(x)$. This completes the proof. □

One property of S-reductions is that they preserve the membership in the approximation hierarchy. Therefore, the SVEP confined to unmapped vertices is at most as hard to approximate as the CSCP. Apart from that, the result implies that an approximation algorithm for the CSCP can be applied to the SVEP while also retaining quality guarantees. Thus, that variant of the SVEP can be approximated as described above. Additionally, [38] analyzes both approximability and inapproximability of the CSCP.

4.5. Flow-based Heuristic

We present a simple heuristic that encodes the SVEP as an instance of the Minimum-Cost Flow Problem (MCFP), the solving of which then, in general, yields a suboptimal solution. Such a solution can then be further improved by another heuristic which will be developed in sections 4.6 and 4.7. A natural definition of the MCFP [39] is through a Linear Program (LP) and can be solved relatively efficiently [40].

Definition 4.7 (Minimum-Cost Flow Problem)**.** Let $G = (V, E)$ be a directed simple graph with a source vertex $s \in V$ and a sink vertex $t \in V$. Additionally, let $c : E \to \mathbb{R}_+$ be a cost function that defines a cost coefficient for a unit flow over an edge and let $\kappa : E \to \mathbb{R}_+$ be a function assigning a maximal flow capacity to each edge. Find a flow of size $d > 0$ from source s to sink t that minimizes the cost, i. e.

$$\min \sum_{(u,v)\in E} c(u,v) \cdot x_{u,v} \tag{4.31}$$

subject to capacity constraints

$$x_{u,v} \le \kappa(u,v), \qquad \forall (u,v) \in E \tag{4.32}$$

flow conservation constraints of the form

$$\sum_{v\in N_G(u)} x_{u,v} = 0, \qquad \forall u \in V \setminus \{ s,t \} \tag{4.33}$$

and outflow and inflow constraints for vertices s and t, respectively,

$$\sum_{v\in N_G(u)} x_{s,v} = d \quad \text{and} \quad \sum_{v\in N_G(u)} x_{v,t} = d. \tag{4.34}$$

We denote an instance of the MCFP as a tuple $(V, E, s, t, \kappa, c, d)$.

For the formulation, let (H, G, v, ϕ) be an instance to the SVEP, i. e. we seek given a partial SVE ϕ from vertices in H to vertices in G for a mapping $\mathcal{M}$ satisfying definition 4.1. Define $N := \{ w \in V(H) : w \in N_H(v), \phi(w) \neq \varnothing \}$ as the set of adjacent vertices in $V(H)$ a SVE of v has to connect to and $n := |N|$ as the number of such neighboring super vertices. We assume that $n \ge 2$ since the case $n = 1$ is trivial as it simplifies for $\mathcal{N} = \{ w \}$

to finding a vertex $u \in V(G)$ that is adjacent to at least one of $\phi(w)$. Similarly to the previous sections, define $R(u) \coloneqq \{ x \in V(H) : w \in \phi(u) \}$ be the reverse mapping of ϕ. Let us quickly sketch the reduction and then formalize it.
We seek for a flow of $n - 1$ units from one vertex a that is adjacent to some super vertex $w \in \mathcal{N}$ adjacent to v to all other adjacent super vertices $\mathcal{N} \setminus \{ w \}$. We define $\mathcal{N}' \coloneqq \mathcal{N} \setminus \{ w \}$ as the set of all other adjacent super vertices. To find such a flow, we define the source s to be the vertex a and create tree structures on all other adjacent super vertices each with capacity 1 such that the flow must go through each adjacent super vertex $\mathcal{N}'$. These additional vertices and edges introduced will be referred to as "artificial". If there is at least one unit of flow through some non-artificial edge $(p, q) \in E$, i. e. $x_{p,q} \geq 1$, then we say p is part of $\mathcal{M}$ and therefore part of the SVE of v. The cost of an edge $(p, q) \in E$ depends on $|R(p)|$, that is, the number other vertices mapped to p. It should be obvious that already mapped vertices must be penalized more than unoccupied ones to prevent them from being added to the placement.

For the sake of clarity, we shall organize the encoding in several steps.

(i) Choose some arbitrary $w \in \mathcal{N}$ and some vertex $a \in N_G(\phi(w))$ with minimal cost which will serve as the source vertex. Additionally, define $\mathcal{N}' \coloneqq \mathcal{N} \setminus \{ w \}$.

(ii) Define non-artificial vertices V_1 and non-artificial edges E_1, the latter comprised of two arcs for each edge in $E(G)$.
$V_1 \coloneqq V(G)$ and $E_1 \coloneqq \{ (p, q), (q, p) : pq \in E(G) \}$

(iii) Define an artificial vertex v_y for each adjacent super vertex $y \in \mathcal{N}'$.

(iv) Let $V_2 \coloneqq \{ v_y : y \in \mathcal{N}' \}$.

(v) Create another artificial vertex t which serves as the sink.

(vi) The resulting graph has vertex set $V \coloneqq V_1 \cup V_2 \cup \{ t \}$.

(vii) For the tree structures on different super vertices, define edge set E_y for $y \in \mathcal{N}'$ to be $E_y \coloneqq \{ (p, v_y) : p \in \phi(y) \}$. Therefore, each vertex in $\phi(y)$ has a directed arc towards the artificial vertex v_y.

($viii$) Connect those artificial vertices $v_y \in V_2$ to t, i. e.
$E_2 \coloneqq \{ (v_y, t) : y \in \mathcal{N}' \}$.

(*ix*) The edge set of the resulting graph is then a union of the defined edge sets: $E := E_1 \cup E_2 \cup \bigcup_{y \in \mathcal{N}'} E_y$.

(*x*) The capacity function $\kappa : E \to \mathbb{R}_+$ is defined as follows:

$$\kappa(p,q) := \left\{ \begin{array}{ll} 1, & \text{for } \exists y \in \mathcal{N}' : (p,q) \in E_y \\ 1, & \text{for } (p,q) \in E_2 \\ n-1, & \text{else} \end{array} \right\}$$

(*xi*) And the cost function $c : E \to \mathbb{R}_+$ is defined with only non-artificial edges having a non-zero cost.

$$c(p,q) := \left\{ \begin{array}{ll} 0, & \text{for } \exists y \in \mathcal{N}' : (p,q) \in E_y \\ 0, & \text{for } (p,q) \in E_2 \\ \lambda |R(p)| + 1, & \text{else} \end{array} \right\}$$

(*xii*) The instance to the MCFP is then $(V, E, a, t, \kappa, c, |\mathcal{N}| - 1)$.

Solving the instance of the MCFP, then yields assignments for $x_{p,q}$ and $x_{q,p}$ for each $pq \in E(G)$. As explained above, we say for an assignment $x_{p,q} \geq 1$ for an arc $(p,q) \in E_1$, that $p \in \mathcal{M}$. Figure 4.5 illustrates an example of an encoding of the SVEP as an instance of the MCFP. For clarity, a King's graph was chosen, but the method is not constrained to specific hardware graphs.

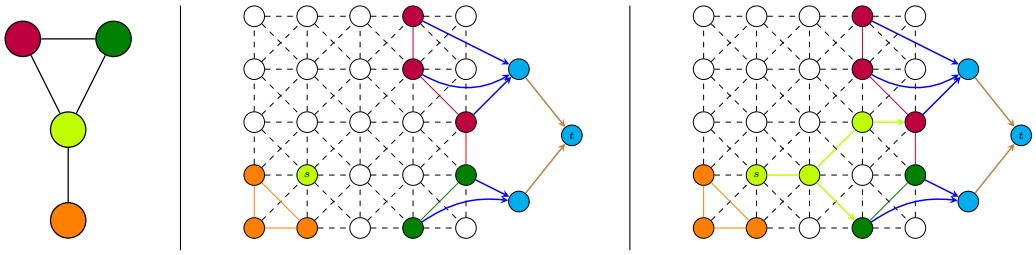


Fig. 4.5.: Example encoding of the SVEP as a MCFP. Orange, purple and green vertices are adjacent super vertices, the placement has to connect to. Vertices colored in cyan are "artificial". Brown arcs are by construction in the set E_2 while blue arcs are in the E_y. Both types are artificial. Find a flow of 2 units from source s to sink t. Dashed edges and white vertices are free to be chosen. Original graph (left), flow construction (middle) and MCFP solution (right).

4.6. Super Vertex Replacement Heuristic

As suggested in 4.2, we developed a heuristic to improve a given SVE. The heuristic randomly adds and removes vertices from the super vertex. If the fitness of the resulting placement is superior to that of the initial super vertex placement, the initial placement is replaced by the newly generated one. Due to the results from section 4.3, unless $\mathcal{P} = \mathcal{NP}$, no polynomial-time algorithm can be expected to solve the SVEP optimally. Hence, a replacement heuristic might improve an initial SVE algorithm.

We call a vertex that is part of two super vertices an "overlap" vertex. The fitness of the SVE depends on the number of overlap vertices contained and the overall size of the SVE. Both is to be minimized in the algorithm with minimizing the number of overlap vertices having a higher priority. The heuristic has a set of candidates that consists of the initial SVE and all neighboring vertices that are not part of any other super vertex. A maximal number of iterations is derived depending on the number of candidates. The first half of all iterations, a random candidate is chosen that might, depending on whether it is part of the current placement, either be removed from or added to the placement. Then, the heuristic tries to greedily remove overlap vertices. In the second half of the number of iterations, the placement can only be reduced.

A vertex of the placement can only be removed if it is neither a cut vertex nor essential. A cut vertex cannot be removed since removing such a vertex would disconnect the subgraph induced by the SVE. The heuristic, however, keeps the current super vertex connected. Additionally, another invariant is the heuristic ensuring that the working super vertex stays incident to all adjacent super vertices. Therefore, before removing a vertex from the super vertex, it is checked whether for each adjacent super vertex, there is some other adjacent vertex in the current placement. This can be efficiently implemented by maintaining a counter for each adjacent super vertex that is set to the number of incident vertices of the placement. A vertex can then only be removed if for every adjacent vertex the counter has value at least 2. Once removed, these counters must be decremented. Obviously, if a vertex is added to the placement, these counters must be incremented. Due to the connectedness invariant, a vertex is only added if at least one of its neighbors is part of the current placement. To prevent a degradation of the placement, overlap vertices are disqualified from reentering the placement and, thus, cannot be added.

Algorithm 4.6.1: Randomized super vertex replacement heuristic

```
1  currentSuperVertex := Set<Vertex> ()
2  candidates := Set<Vertex> ()

   Input: sVertex: Vertex, embedding: EmbeddingState
3  Function SVPReducer:
4      currentSuperVertex ← embedding.getSuperVertex(sVertex)
5      candidates ← embedding.getClosure(currentSuperVertex)
6      maxIters = getMaxIters(candidates.size())
7      for iteration ← 1 to maxIters/2 do
8          v ← getRandom(candidates)
9          if v ∈ currentSuperVertex then
10             RemoveVertex(v)
11         else
12             AddVertex(v)
13     GreedyRemoveOverlapVertices()
14     for iteration ← 1 to maxIters/2 do
15         v ← getRandom(candidates)
16         if v ∈ currentSuperVertex then
17             RemoveVertex(v)

18 Function GreedyRemoveOverlapVertices:
19     foreach v ∈ OverlapVertices(currentSuperVertex) do
20         RemoveVertex(v)

   Input: v: Vertex
21 Function RemoveVertex:
22     if not isCutVertex(v) and not isEssential(v) then
23         currentSuperVertex ← currentSuperVertex ∖ { v }

   Input: v: Vertex
24 Function AddVertex:
25     if not isOverlapVertex(v) and isConnected(v) then
26         currentSuperVertex ← currentSuperVertex ∪ { v }
```

4.7. Evolutionary Reduction Heuristic

Apart from the replacement heuristic presented in section 4.6, we developed another heuristic based on the idea of evolution, i. e. an evolutionary metaheuristic. We will devote this section to presenting its main features, that is, the crossover approach, mutation, local optimization of individuals and the reproduction scheme.

Again, there are two invariants regarding the SVE of an individual. (i) the super vertex remains connected and (ii) the super vertex remains connected to each adjacent super vertex that has already been mapped. These invariants are the same as for the heuristic in section 4.6 and are essentially $(M1)$ and $(M3)$ in definition 4.1.
In the heuristic, each individual holds its own SVE, randomly mutates it, then reduces it for a specified number of iterations and is finally evaluated in terms of its fitness. In each generation, a set of such individuals is generated using the last generation as the parent generation with the first generation having the initial placement as the super vertex at birth.

Reproduction scheme:
The heuristic applies a (μ, λ) selection, that is, out of the parental generation, the best μ are selected to generate λ children. The best μ of those, in turn, form the next parental generation.

Mutation:
For a given placement, each individual randomly chooses a free vertex v that is adjacent to at least one of the vertices in the placement. That vertex v is added to the placement and a DFS is conducted to add more free vertices to the placement. However, at most a constant number of free vertices along the DFS can be added to the placement and once that is done, the DFS stops.

Crossover:
Given two individuals from the parent generation, the individual's initial placement is the union of the two parent's placements. If that initial placement is not connected, i. e. the two parents have disjoint placements that are not connected through an edge, the individual is marked dead and another individual is generated. The crossover takes care of setting up other data structures to improve the performance of reduction steps.

Algorithm 4.7.1: Mutation of an individual's placement

```
placement := Set<Vertex> ()
freeClosure := Set<Vertex> ()
adjacentStack := Stack<Queue<Vertex>> ()
nbAdded: Integer := 0

Input: individual: Individual, embedding: EmbeddingState
Function Mutate:
    placement ← individual.getSuperVertex()
    freeClosure ← embedding.getClosure(placement).filter(isFree)
    start ← chooseRandomElement(freeClosure)
    if |freeClosure| = 0 then return
    AddVertex(start)
    while not adjacentStack.empty() and nbAdded ≤ maxNew do
        if adjacentStack.top().empty() then
            adjacentStack.pop()
        else
            next: Vertex ← adjacentStack.top().pop()
            if embedding.isFree(next) and next ∉ placement then
                AddVertex(next)

Input: v: Vertex
Function AddVertex:
    placement ← placement ∪ { v }
    adjacentStack.push(getAdjacentList(v))
    nbAdded ← nbAdded + 1
```

Local optimization of individuals:

The optimization of an individual tries reducing the placement's size and thereby improving the placement's quality. Similarly to the heuristic developed in section 4.6, the heuristic greedily tries removing overlapping vertices, then for a specific number of iterations depending on the placement's size, random vertices are reduction candidates. If such a reduction is successful, a DFS on the placement is conducted, visiting a vertex iff it can be removed. The motivation behind that is that, in practice, if a vertex can be removed, its neighbors can likely be removed as well. Finally, the algorithm sequentially tries removing each remaining vertex.

Algorithm 4.7.2: Local optimization of an individual's placement

```
1  placement := Set<Vertex> ()
2  stack := Stack<Queue<Vertex>> ()

   Input: individual: Individual, embedding: EmbeddingState
3  Function Optimize:
4      placement ← individual.getSuperVertex()
5      for v ∈ OverlapVertices(placement) do
6          TryRemove(v)
7      for iteration ← 1 to maxIters do
8          v ← getRandom(placement)
9          TryRemoveDFS(v)
10     for v ∈ placement do
11         TryRemove(v)

   Input: v: Vertex
12 Function TryRemoveDFS:
13     if not TryRemove(v) then return
14     stack.push(Queue(N_G(v) ∩ placement))
15     while not stack.empty() do
16         if stack.top().empty() then
17             stack.pop()
18         else
19             if TryRemove(stack.top().pop()) then
20                 stack.push(Queue(N_G(w) ∩ placement))
```

5. Local Minor Repair Problem

One of the mutations, we apply in chapter 6 chooses some vertex of the hardware graph and "destroys" the mapping around that vertex, potentially separating super vertices or even fully erasing them. The mutation then tries reconnecting the remaining ends and previously adjacent super vertices, essentially, repairing that previously "destroyed" spot. We will refer to this problem as the LMRP. Similar to chapter 4, we will first formalize the problem, prove the problem's computational intractability under specific circumstances and finalize the chapter with a heuristic that tries to tackle the problem.

5.1. Formal definition of the LMRP

Definition 5.1 (Local Minor Repair Problem)**.**
Let H and G be two graphs, $\phi : V(H) \to 2^{V(G)}$ be a mapping and let $\mathcal{Y} \subset V(G)$ be some proper subset of vertices of G such that $G[\mathcal{Y}]$ is connected. $\mathcal{Y}$ is a set of vertices for which the mapping is "destroyed". Additionally, define $\mathcal{X} := \{ v \in V(H) : \mathcal{Y} \cap \phi(v) \neq \varnothing \}$ as the affected vertices $\mathcal{X} \subseteq V(H)$. Find a mapping $\psi : \mathcal{X} \to 2^{\mathcal{Y}}$ satisfying conditions $(L1)$ through $(L3)$ while minimizing 5.35. Denote $R(v) := \{ u \in \mathcal{X} : v \in \psi(u) \}$ as the reverse mapping for ψ. Minimize the number of overlapping vertices and the number of vertices mapped to, that is, minimize

$$\min \sum_{v \in \mathcal{Y}} (\lambda \max\{ |R(v)| - 1, 0 \} + sgn(|R(v)|)). \tag{5.35}$$

Let E be the embedded edges related to $\mathcal{Y}$. More formally, for $\mathcal{N} := \{ w \in N_G(v) : v \in \mathcal{Y} \} \cup \mathcal{Y}$ as the closure of $\mathcal{Y}$, define

$$E := \{ pq \in E(H) : r \in \mathcal{N}, s \in \mathcal{Y}, rs \in E(G), r \in \phi(p), s \in \phi(q) \}. \tag{5.36}$$

We define the adjusted mapping as

$$\theta(v) := \left\{ \begin{array}{ll} (\phi(v) \setminus \mathcal{Y}) \cup \psi(v), & \text{for } v \in \mathcal{X} \\ \phi(v), & \text{else} \end{array} \right\}$$

which, essentially, adjusts the mapping which was partially "destroyed" to now be incorporate mapping ψ. For $(L3)$, we define $B(v) := \psi(v) \cap (\mathcal{N} \setminus \mathcal{Y})$ as all mapped vertices adjacent to $\mathcal{Y}$ for a vertex $v \in V(H)$.

$(L1)$ $\forall u, v \in (\mathcal{N} \setminus \mathcal{Y}), u \neq v \exists w \in V(H), u, v \in \phi(w) :$
$\exists \mathcal{P}_{u,v}(G, \phi(w) \cap \mathcal{Y}) \Rightarrow \exists \mathcal{P}_{u,v}(G, \psi(w))$.
That is, for two vertices u, v that are each adjacent to some vertex in $\mathcal{Y}$ and with both u and v being part of some SVE for vertex $w \in V(H)$, the following must be true: If u and v are connected through a path having internal vertices only in $\phi(w) \cap \mathcal{Y}$, then there is again such a path with internal vertices only in $\psi(w)$.

$(L2)$ $\forall pq \in E \exists r \in \mathcal{N} \exists s \in \mathcal{Y} : rs \in E(G), r \in \theta(p), s \in \theta(q)$, i. e. every previously embedded edge of H is again embedded.

$(L3)$ $\forall v \in \mathcal{X}, \psi(v) \neq \varnothing : (B(v) \neq \varnothing) \Rightarrow (\forall a \in \psi(v) \exists b \in B(v) : \exists \mathcal{P}_{a,b}(G, \psi(v)))$, i. e. if $v \in V(H)$ is mapped to some vertex in $B(v)$ and is mapped to at least one vertex in $\mathcal{Y}$, all vertices mapped to in $\mathcal{Y}$ should be connected to at least one on the border $B(v)$.

The decision version for an instance $(H, G, \phi, \mathcal{Y})$ of the LMRP is to decide whether there is such a solution ψ with cost at most $\delta \in \mathbb{R}_+$.

Thus, as the name of the LMRP implies, the problem is defined to only act locally while retaining all local features of the previous embedding. Therefore, instead of reducing the edges to be embedded in the destroyed region to edges not covered by the rest of the embedding, the LMRP just "reorders". Similarly, the existence of paths through $\mathcal{Y}$ remains invariant as $(L1)$ implies. In practice, both of these properties are beneficial when introducing such changes in parallel which is the motivation behind definition 5.1. $(L3)$ is a technical requirement as a super vertex must stay connected. If a solution did not satisfy $(L3)$, the solution would, clearly, disconnect at least one super vertex.

5.2. The LMRP is $\mathcal{NP}$-hard

As suggested, we want to discuss the computational complexity of the LMRP. However, we want to distinguish between two variants of the LMRP for an instance $(H, G, \phi, \mathcal{Y})$: (i) a variant where the size of $\mathcal{Y}$ is not constant and, thus, a complexity parameter and (ii) the case that G has constant size implying the same for $\mathcal{Y}$ as $\mathcal{Y} \subset V(G)$. We will sketch a reduction from the 3-Partition Problem (3PP) to the variant (i) of the LMRP and explain how to extend this proof to Chimera graphs. The result, however, motivates to keep $\mathcal{Y}$ constant, i. e. confining oneself to variant (ii), which is in practice a reasonable restriction. Interestingly, though, as we will prove through this section, the problem remains computationally intractable - even with G having constant size. However, we do not extend the proof to Chimera graphs. Regardless of that, the LMRP likely remains difficult even on Chimera graphs. Interestingly, though, the construction for the reduction to variant (ii) is kept quite simplistic such that it can be found as a subgraph in both the Pegasus and the King's graph and, therefore, on those graphs the problem, indeed, remains $\mathcal{NP}$-hard.

Definition 5.2 (3-Partition Problem)**.**
Given a multiset $\mathcal{I}$ of integers, let $n \coloneqq |\mathcal{I}|$ and without loss of generality assume that n is divisible by 3, decide whether there is some partitioning of elements of $\mathcal{I}$ into $m \coloneqq \frac{n}{3}$ triplets T_i each summing up to the same value $t \coloneqq \frac{1}{m} \sum_{e \in \mathcal{I}} e$ [41].

Let $\mathcal{I}$ be an instance of the 3PP, for the reduction, we observe that there are at most $\binom{n}{3}$ possible triplets to choose from which is in $\mathcal{O}(n^3)$. We will exploit that fact in the reduction.

The reduction from the 3PP to the LMRP is visualized in figure 5.1.

5.2.1. Variant (i) is $\mathcal{NP}$-hard

Claim 5.1. *Variant* (i) *of the LMRP is* $\mathcal{NP}$*-hard.*

Let $\mathcal{I} = \{ x_1, ..., x_n \}$ be an instance of the 3PP with $n := |\mathcal{I}|$. We want to reduce to an instance $(H, G, \phi, \mathcal{Y})$ of the LMRP. For that, define H to be isomorphic to the star graph S_{n+1}. More formally, define a vertex v_i for each element of $x_i \in \mathcal{I}$ and an extra vertex w such that $V(H) := \{ v_1, ..., v_n, w \}$ and $E(H) := \{ \{ v_i, w \} : i \in \mathbb{Z}_n \}$. G is visualized in figure 5.1. Wavy edges are paths of length $q \in \mathbb{N}$. Similar to the proof of claim 4.2 in section 4.3, q will be chosen sufficiently large but polynomial in n. In fact, for general graphs constant q suffices but when adjusting the reduction to grid graphs, this is not true anymore. We want to formalize the construction of graph G. For that, let $T_1, ..., T_r$ be all possible triplets of distinct elements in $\mathcal{I}$ ignoring order. Clearly, $r := \binom{n}{3}$. We define $T_i \in \mathbb{Z}_n^3$ such that for a triplet $T_i = (a, b, c)$, T_i contains elements $x_a, x_b, x_c \in \mathcal{I}$. Additionally, we define for every triplet $T_i = (a, b, c)$ the sum of the elements as $s(T_i) := x_a + x_b + x_c$. For every element $x_i \in \mathcal{I}$, we create a vertex e_i in G and for every triplet T_i, we create a vertex $\mathcal{T}_i$. Let $V_1 := \{ e_1, ..., e_n \}$ and $V_2 := \{ \mathcal{T}_1, ..., \mathcal{T}_r \}$. Furthermore, we create vertices u, t and to connect t to all $\mathcal{T}_i$ through paths of length q, we create q vertices for each $\mathcal{T}_i$, i. e. $W_i := \{ s_{i,1}, ..., s_{i,q} \}, i \in \mathbb{Z}_r$. Thus, the vertex set of G is

$$V(G) := V_1 \cup V_2 \cup \{ u, t \} \cup \bigcup_{i=1}^{r} W_i. \tag{5.37}$$

We connect u and t through an edge and create the paths from the $\mathcal{T}_i$ to t by defining edge sets $E_i := \{ \{ s_i, s_{i+1} \} : i \in \mathbb{Z}_{q-1} \} \cup \{ \{ \mathcal{T}_i, s_1 \}, \{ s_q, t \} \}$. Additionally, we connect a vertex e_l representing an element $x_l \in \mathcal{I}$ to vertex $\mathcal{T}_i$ iff the corresponding triplet T_i contains element x_l, we denote that by $x_l \in T_i$. Thus, the edge set of G is

$$E(G) := \{ \{ u, t \} \} \cup \bigcup_{i=1}^{r} E_i \cup \{ \{ e_l, \mathcal{T}_i \} : x_l \in T_i \}. \tag{5.38}$$

We define $\mathcal{Y} := V(G) \setminus (V_1 \cup \{ u \})$, i. e. essentially all vertices within the rectangle in figure 5.1, and set mapping ϕ to

$$\phi(x) := \left\{ \begin{array}{ll} \mathcal{Y} \cup \{ e_i \}, & \text{for } x = v_i, i \in \mathbb{Z}_n \\ \mathcal{Y} \cup \{ u \}, & \text{else} \end{array} \right\}$$

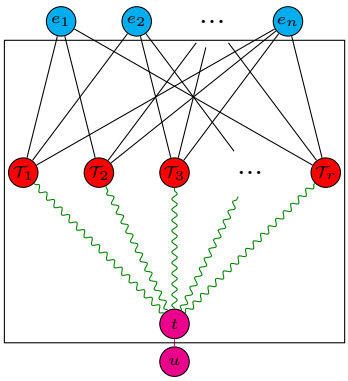


Fig. 5.1.: Reduction from the 3PP to the LMRP.

We set $\lambda := 1$, $\delta := \frac{n}{3}(q+1)+1$ and decide whether there is a solution to the generated instance of the LMRP with cost at most δ. For that, we define $q := 1$. We claim that there is such a solution with cost at most δ iff there is a valid partitioning for the 3PP instance.

"$\Leftarrow$": *If there is a valid partitioning for the 3PP instance, there is a valid solution to the generated instance of the LMRP with cost at most δ.* Given an index set $\mathcal{U} \subset \mathbb{Z}_r$ with $|\mathcal{U}| = \frac{n}{3}$ such that the triplets $T_i, i \in \mathcal{U}$ form a solution to the instance $\mathcal{I}$, θ is a valid solution to the instance of the LMRP

$$\theta(z) := \left\{ \begin{array}{ll} \{e_k\}, & \text{for } z = v_k, k \in \mathbb{Z}_n \\ \{u,t\} \cup \bigcup\limits_{i \in \mathcal{U}} \{\mathcal{T}_i, s_{i,1}, ..., s_{i,q}\}, & \text{for } z = w \end{array} \right\}$$

Clearly, the cost is exactly δ as vertex w is mapped to exactly $\omega = \frac{n}{3}q + \frac{n}{3} + 1$ vertices in $\mathcal{Y}$ introducing cost δ.

"$\Rightarrow$": *If there is a solution to the generated instance of the LMRP with cost at most δ, there is a valid partitioning for the instance of the 3PP.* There are some obvious observations regarding the construction that we want to note. Since H was chosen to be a star graph with the leaves being mapped to the vertices e_i in G and w to vertex u, each v_i must be connected to u. Thus, since every vertex $\mathcal{T}_i$ is connected at exactly 3 distinct e_i, it is clear that at least $\frac{n}{3}$ vertices $\mathcal{T}_i$ must be mapped, introducing cost at least $\frac{n}{3}$. Apart from that, it is obvious that t must somehow be mapped since otherwise not all edges can be embedded. Therefore, the total cost is always at least $\frac{n}{3}+1$. Trivially, at least $\frac{n}{3}$

green paths have to be mapped onto since otherwise w could not possibly be connected to all v_i in a mapping θ. Suppose there was some solution with cost at most δ to the generated instance. This would imply that exactly $\frac{n}{3}$ of those green paths were chosen to be part of the solution which in turn implies the existence of a valid partitioning to the original 3PP instance since every $\mathcal{T}_i$ that is part of the solution corresponds to a triplet T_i in the solution of the 3PP instance.

Thus, deciding the generated instance of the LMRP is equivalent to deciding the original 3PP instance and the reduction is polynomial. This proves claim 5.1. □

As noted, we do not want to extend this reduction to Chimera graphs. In general, however, the graph from figure 5.1 has to be encoded onto a grid graph which in turn can be encoded into the various hardware topology graphs. Similar to the proofs in section 4.3, q must be chosen large enough that it dominates the size of the solution. That is, q must be larger than the number of all other vertices in the construction. Then, q dominates the cost δ and solving variant (i) of the LMRP suffices to decide the original 3PP instance. The construction itself is quite straightforward.

5.2.2. Variant (ii) is $\mathcal{NP}$-hard

For the proof that variant (ii) is $\mathcal{NP}$-hard, we reduce from a variant of the well-known 3-Satisfiability (3SAT) problem. Schaefer's dichotomy theorem states that the Exactly-1-Positive 3-SAT Problem (1-In-3-SAT+) is just $\mathcal{NP}$-hard [42, 43] and due to the fact that there are no negations, this variant is in our case particularly well-suited for a reduction. Therefore, let us formally define the 1-In-3-SAT+.

Definition 5.3 (Exactly-1-Positive 3-SAT Problem)**.** Let $X = (x_1, ..., x_n) \in \{\,\text{true}, \text{false}\,\}^n$, be $n \in \mathbb{N}$ binary literals and $C_j, j \in \mathbb{Z}_m$ be m clauses of a Conjunctive Normal Form (CNF) with each clause consisting of exactly 3 *positive* literals x_i, i. e. no literal is negated. Decide whether there is some assignment for the x_i such that in each clause, there is *exactly* one literal true while the others are false, see [43]. We denote an instance of the 1-In-3-SAT+ as (X, C) with $C \coloneqq (C_1, ..., C_m)$.

Example 5.1. *Let $X = (x_1, x_2, x_3, x_4, x_5) \in \{\,true, false\,\}^5$ be literals. For the CNF*

$$(x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee x_3 \vee x_4) \wedge (x_2 \vee x_3 \vee x_5), \tag{5.39}$$

$x_1 = x_5 = true$ and $x_2 = x_3 = x_4 = false$ is a satisfying assignment such that in every clause, there is exactly one literal set to true.

Let us reduce 1-In-3-SAT+ to variant (ii) of the LMRP.

Claim 5.2. *Variant (ii) of the LMRP, i. e. the LMRP with constant-size G, remains $\mathcal{NP}$-hard.*

Let (X, C) be an instance to the 1-In-3-SAT+ and $(H, G, \phi, \mathcal{Y})$ be an instance of the LMRP where G and $\mathcal{Y}$ must have constant size. G is visualized in figure 5.2 and $\mathcal{Y}$ are all vertices contained in the rectangle. Thus, it is clear that both G and $\mathcal{Y}$ indeed have constant size. We assume for $n \coloneqq |X|$ and $m = |C|$ that $m > 1$ and $n > 1$ and that every literal is part of some clause. We will now construct H and ϕ. Then, we will decide whether there is some solution to the LMRP on that instance with cost at most δ. We will show that this decides the instance of the 1-In-3-SAT+. Let $r, s \in \mathbb{N}$ and $\lambda \in \mathbb{N}$ be sufficiently large. For the reduction, we assume that $C_i[j]$ returns the literal in clause j at position $j \in \mathbb{Z}_3$.

(*i*) Create a vertex v_i in H for every literal x_i:
$V_1 := \{ v_1, ..., v_n \}$

(*ii*) Create r vertices for each clause C_i:
$\forall i \in \mathbb{Z}_m : N_i := \{ c_i^1, ..., c_i^r \}$

(*iii*) Each clause has exactly 3 literals. For each of those positions in a clause and for every clause, create s vertices:
$\forall i \in \mathbb{Z}_m \forall j \in \mathbb{Z}_3 : Q_{i,j} := \{ q_{i,j}^1, ..., q_{i,j}^s \}$

(*iv*) Let w be a vertex of H that is needed to simplify the proof.

(*v*) The vertex set is therefore the union of the defined vertex sets:
$V(H) := V_1 \cup \{ w \} \cup \bigcup_{i=1}^{m} N_i \cup \bigcup_{i=1}^{m} (Q_{i,1} \cup Q_{i,2} \cup Q_{i,3})$

(*vi*) For each clause, connect each vertex representing a position in that clause to the vertices representing the clause, i. e.
$\forall i \in \mathbb{Z}_m : E_i := \{ \{ c_i^k, q_{i,j}^l \} : c_i^k \in N_i, q_{i,j}^l \in Q_{i,j}, j \in \mathbb{Z}_3 \}$

(*vii*) For each clause, connect each pair of vertices representing distinct positions in that clause through an edge, i. e. essentially the edge set of a complete bipartite graph with partite sets $Q_{i,a}$ and $Q_{i,b}$:
$\forall i \in \mathbb{Z}_m \forall j \in \mathbb{Z}_3 : B_{i,j} := \{ \{ q_{i,j}^a, q_{i,k}^b \} : q_{i,j}^a \in Q_{i,j}, q_{i,k}^b \in Q_{i,k}, k = (j \, mod \, 3) + 1 \}$

(*viii*) For each literal x_k, connect the representing vertex v_k to all vertices representing a position of a clause iff x_k is at that position of the clause. That is, if clause 1 was $(x_1 \vee x_2 \vee x_3)$, vertex v_2 would be connected to all $q_{1,2}^l$. More formally,
$\forall k \in \mathbb{Z}_n : X_k := \{ \{ x_k, q_{i,j}^l \} : C_i[j] = x_k, q_{i,j}^l \in Q_{i,j}, i \in \mathbb{Z}_m, j \in \mathbb{Z}_3 \}$

(*ix*) The edge set of H is therefore:
$E(H) := \bigcup_{i=1}^{m} E_i \cup \bigcup_{k=1}^{n} X_k \cup \bigcup_{i=1}^{m} (B_{i,1} \cup B_{i,2} \cup B_{i,3})$

(*x*) For graph G, we refer to the "destroyed" region as $\mathcal{Y}$ and define $V(G) := \mathcal{Y} \cup \{ 1, 2, 3, 4, 5, 6, 7 \}$ as visualized in figure 5.2.

(*xi*) Mapping ϕ is defined such that every vertex $V(H)$ is mapped to every vertex $\mathcal{Y}$ and depending on the type on a subset of the remaining vertices. More precisely, define

$$\phi(u) := \begin{cases} V(G), & \text{for } u = w \\ \mathcal{Y} \cup \{7\}, & \text{for } u \in N_i, i \in \mathbb{Z}_m \\ \mathcal{Y} \cup \{1\}, & \text{for } u \in V_1 \\ \mathcal{Y}, & \text{else} \end{cases}$$

In particular, vertices representing literals are placed on 1, vertices representing clauses are placed on vertex 7, vertices representing positions in clauses are placed only within $\mathcal{Y}$ and vertex w is placed on all vertices.

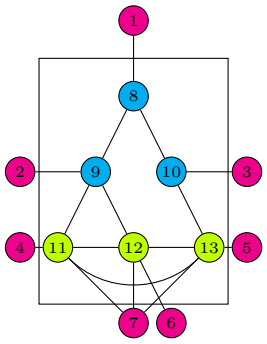


Fig. 5.2.: Graph G for the reduction from 1-In-3-SAT+ to the LMRP with constant size G. Vertices inside the rectangle are in $\mathcal{Y}$ and will have their mapping "destroyed" while vertices outside remain fixed.

We want to explain the reduction and then prove the claim afterwards. Notice that placements on vertices $\{1, 2, 3, 4, 5, 6, 7\}$ are all fixed as they do not belong to $\mathcal{Y}$. Apart from that, both H and ϕ are polynomial in size with respect to m and n, if r and s are. We will show that r and s having polynomial size suffices. In general, $|V(H)| = rm + 3sm + n + 1$. Even if H was isomorphic to the complete graph on $|V(H)|$ vertices, which is in general not the case, $E(H)$ would still be polynomial in size.

Placing multiple vertices with the same semantics on the same vertex is essentially used as a "gadget" in this reduction. More precisely, placing a sufficiently large amount of those, would imply a large cost extending their super vertices. Setting δ accordingly, i. e. such that the cost extending the super vertices implies that the total cost exceeds δ, leads to the placement being fixed. The reduction does exactly that for vertices representing clauses as those will be fixed on vertex 7 and vertices in the sets $Q_{i,j}$ will be fixed on the green vertices in a special manner. For every clause C_i, we introduced for each of the three positions exactly s vertices. Additionally, we added edges between $Q_{i,1}$, $Q_{i,2}$ and

$Q_{i,3}$ for every clause C_i. The LMRP, however, demands that if an edge was previously embedded in ϕ within $\mathcal{Y}$, then it must be embedded again in a solution. Interestingly, this enforces that the positions cannot simply collapse onto one vertex but instead, we enforce a permutation on those three green vertices. Notice that the "permutation" will be overlap-free if there is indeed a solution to the instance constructed with cost at most δ. The idea of the reduction is then that vertices representing literals must choose between extending from vertex 8 to vertex 9 or 10 – but not both since that would exceed δ. Semantically, if the vertex representing literal x_k was mapped to vertex 10, x_k would be true. Conversely, for vertex 9 that would imply that x_k is false. Mapping a vertex representing literal x_k to either vertices 9 or 10, forces vertices representing clauses in which x_k is present to align according to that - or extend. The latter would again imply costs such that the total costs exceed δ. Therefore, mapping an x_k constrains the permutation of mappings for those $Q_{i,j}$ where x_k is in clause C_i. Vertex w helps in that in every solution, w is mapped to all of $\mathcal{Y}$ and therefore any other vertex of H mapped to some vertex in $\mathcal{Y}$ always introduces cost precisely λ.
Obviously, we will have to prove the reduction. Before, let us define r, s and δ. To decide whether there is a valid assignment to the 1-In-3-SAT+ instance, define $\delta \coloneqq 6+2n\lambda+3sm\lambda$ with $\lambda \geq 2$. We define $s \coloneqq 2n+1$ and $r \coloneqq \lceil \delta \rceil + 2$.

In general, we want to show that, if there is a satisfying assignment, there is some solution to the instance generated with cost δ while if there is no satisfying assignment, the optimal solution has cost greater than δ. Suppose there was some satisfying assignment p, then θ is a valid solution as described in definition 5.1:

$$\theta(u) \coloneqq \left\{ \begin{array}{ll} V(G), & \text{for } u = w \\ \{7\}, & \text{for } u \in N_i, i \in \mathbb{Z}_m \\ \{1,8,9\}, & \text{for } u = v_k \in V_1, p(x_k) = \text{false}, k \in \mathbb{Z}_n \\ \{1,8,10\}, & \text{for } u = v_k \in V_1, p(x_k) = \text{true}, k \in \mathbb{Z}_n \\ \{13\}, & \text{for } u \in Q_{i,j}, p(C_i[j]) = \text{true}, i \in \mathbb{Z}_m, j \in \mathbb{Z}_3 \\ \{11\}, & \text{for } u \in Q_{i,j}, p(C_i[(j \bmod 3)+1]) = \text{true}, i \in \mathbb{Z}_m, j \in \mathbb{Z}_3 \\ \{12\}, & \text{else} \end{array} \right\}$$

Clearly, there are paths through $\theta(w)$ connecting vertices 1 through 7. Apart from that, every vertex representing a position in a clause is mapped to a green vertex, all of which

are connected to 7 onto which the vertices representing clauses are mapped. Hence, edges created in (vi) are embedded. Due to the fact that exactly one literal in a clause is true, all the s vertices representing that literal will be mapped on vertex 13. The other two positions are distributed on vertices 11 and 12 such that every position of a clause is placed onto a different vertex. However, since the green vertices form a triangle, all edges of (vii) are embedded. Additionally, if a literal is true in the assignment, there is an edge to vertex 13 which every clause that contains that literal is mapped to and therefore there that edge is again embedded. Conversely, if the literal is false, it shares an edge with both 11 and 12 onto which all vertices representing positions of clauses with that literal are mapped. Thus, all edges from $(viii)$ are embedded. Trivially, all super vertices are properly connected. Consequently, θ is a valid solution encoding a valid assignment to the 1-In-3-SAT+. The cost of θ is

$$c = 3 + 2n\lambda + 3sm\lambda + 3 = \delta \tag{5.40}$$

which is exactly δ. We now have to show that if there is no valid assignment, the cost is higher than that. Suppose for contradiction that for an instance based on an unsatisfiable instance of the 1-In-3-SAT+, there was an *optimal* solution to the generated instance of the LMRP with cost at most δ.

All vertices encoding positions in a clause, i. e. all vertices in some $Q_{i,j}$, must be mapped to *at least* one green vertex. Suppose some vertex $q^l_{i,j} \in Q_{i,j}$ was not mapped to a green vertex. We assumed there was a solution to the instance and therefore this implies that all edges are embedded. In particular, all edges generated in step (vi) must be embedded. By property $(L3)$ from definition 5.1, the mapping must be connected which is why all vertices in N_i must be placed on at least one green vertex. The cost introduced is at least $\lambda(r-1) = \lambda(\lceil\delta\rceil+1) > \delta$ which would contradict the assumption that the solution has cost at most δ. Thus, all vertices encoding positions in a clause are placed on at least one green vertex.

For each clause C_i, every vertex encoding a position in a clause is mapped to at most one vertex. Again, suppose there was at least one vertex $q^l_{i,j}$ that was mapped to more than one vertex in G. Due to the optimality assumption, there cannot be another $q^k_{i,j}$, $k \neq l$ that is mapped to fewer vertices in $\mathcal{Y}$. If there were, setting $\theta(q^l_{i,j}) := \theta(q^k_{i,j})$ would improve the solution since they have the same neighborhood - contradicting optimality. Thus, we can assume that all vertices in $Q_{i,j}$ are mapped to the same number $t \geq 2$ of

vertices. Since $|Q_{i,j}| = s$, the cost would be at least $6 + 3sm\lambda + (t-1)s\lambda$ - if all other $Q_{x,y}$ were mapped to exactly one vertex and not taking into account any vertices representing literals. Due to the definition of s, the cost exceeds δ:

$$6 + 3sm\lambda + \underbrace{(t-1)}_{\geq 1} s\lambda \geq 6 + 3sm\lambda + s\lambda = \underbrace{6 + 3sm\lambda + 2n\lambda}_{\delta} + \lambda > \delta \tag{5.41}$$

Henceforth, we can assume that every vertex representing a clause is mapped to vertex 7 and every vertex representing a position in a clause is mapped to *exactly* one green vertex. Additionally, for every clause C_i, the $Q_{i,1}, Q_{i,2}$ and $Q_{i,3}$ are placed on different (green) vertices, i. e. per clause, positions are mapped onto different vertices. Suppose that was not the case, that is, for clause C_i, there are two vertices $q^l_{i,a}, q^k_{i,b}, a \neq b$ from different $Q_{i,j}$ that are mapped to the same vertex. We proved that, in an optimal solution, all vertices representing positions in clauses must be mapped onto *exactly* one vertex. However, due to edges created in (vii), $q^l_{i,a}$ and $q^k_{i,b}$ must share an edge which they do not. This would contradict the validity or the optimality of the solution.

Clearly, every vertex representing a literal must be mapped to at least two blue vertices to connect to the various $q^l_{i,j}$ introducing a cost of $2n\lambda$. Therefore, the total cost is at least $6 + 3sm\lambda + 2n\lambda = \delta$. We assumed that the instance of the 1-In-3-SAT+ did not have a valid assignment but a solution to the generated instance of the LMRP with cost at most δ. Clearly, since the total cost is at least δ, every vertex representing a literal must be mapped to exactly two blue vertices. Thus, by $(L3)$, every literal must be mapped to either $\{1, 8, 9\}$ or to $\{1, 8, 10\}$. If there was indeed solution with cost at most δ, then we could derive a valid assignment to the original instance of the 1-In-3-SAT+ as follows. We say a literal x_k = true if the corresponding vertex v_k is mapped to vertex 10 and, conversely, x_k = false if v_k is mapped to vertex 9. Trivially, due to the connectivity property of super vertices and the maximal size of a vertex representing a literal, at most one of the literals can be true in an optimal solution with cost at most δ. And due to $(L2)$ from definition 5.1, at least one of those literals must be true to embed edges $(viii)$ - we assumed every literal is part of at least one clause. Since for every clause C_i, the positions $Q_{i,j}$ must be distributed on different green vertices, in each clause one of the positions must be mapped on vertex 13 while the two other positions are mapped on 11 and 12. That in conjunction with literals only being allowed to be mapped to

exactly two blue vertices without contradicting the assumptions made implies that for a valid solution with cost at most δ, exactly one literal in a clause must be true. Thus, every clause must be valid, and therefore we can indeed derive a valid assignment to the original 1-In-3-SAT+ instance. This contradicts the assumption that the instance was not satisfiable. Hence, a solution to the reduced instance of the LMRP cannot have cost at most δ if the original 1-In-3-SAT+ instance did not have a satisfying assignment.

Therefore, the solution of a reduced instance has cost at most δ iff the original 1-In-3-SAT+ instance is satisfiable. Consequently, the reduction is valid since it is polynomial in size and allows to decide the 1-In-3-SAT+ using the LMRP with constant-size G. This proves claim 5.2. □

Obviously, the Chimera graph is bipartite and thus, in particular, it does not contain odd cycles [44]. The graph in figure 5.2 contains a triangle which implies that the Chimera graph does not contain the graph from figure 5.2 as a subgraph. This is different for both Pegasus and King's graphs, however.

Claim 5.3. *Variant* (ii) *of the LMRP, i. e. with constant-size* G*, remains* $\mathcal{NP}$*-hard on both Pegasus and King's graphs with missing vertices and missing edges.*

It suffices to show that both graphs contain the construction from figure 5.2 as a subgraph.

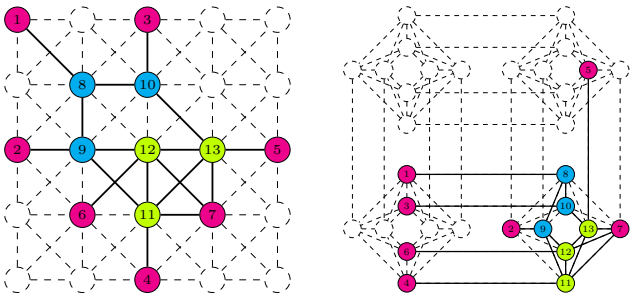


Fig. 5.3.: The construction for the reduction from the 1-In-3-SAT+ to variant (ii) of the LMRP is subgraph of both the King's graph (left) and the Pegasus graph (right). Dashed vertices/edges are "missing" vertices/edges, respectively. Labelling and coloring corresponds to that in figure 5.2.

This proves claim 5.3. □

5.3. Heuristic for the LMRP

A heuristic for the LMRP has to repair the destroyed crater and, therefore, repair previous connections throughout the crater, embed all edges that were previously embedded in that region and preserve connectivity. Our heuristic does exactly that by applying Dijkstra's algorithm iteratively.

Firstly, the algorithm records all connected subgraphs among the crater which we will refer to as components. Thus, for every component, a DFS is conducted, and the border vertices involved are placed in a list for each component. Apart from that, all edges embedded in the initial placement are recorded as well.

The heuristic starts the actual solving process by reconnecting the components. In particular, all components consisting of more than one vertex are recreated using Dijkstra's algorithm and then, at, the end of that, the previously embedded edges are embedded. Finally, those vertices that were completely erased are embedded into the crater. If a vertex has no neighbors in the crater, it is placed randomly onto some vertex. If there is exactly one neighbor, a simple adjacent vertex inside the crater is chosen and if there are multiple neighbors the vertex is mapped using multiple Dijkstra runs. The latter is achieved by adding all vertices from some neighbor to the priority queue and then finding a path of length at least one to some other neighbor. That path is then the initial placement for the previously completely erased vertex and the placement can be successively extended through more Dijkstra runs.

The algorithm must query the subgraph and somehow prevent that anything about the border and the crater are changed in the mapping during the computation to eliminate inconsistencies. Therefore, we use a subgraph manager that keeps track of locks on subgraphs of the hardware graph. Due to the fact that the LMRP relies only on local features, that suffices to ensure that multiple LMRP instances do not interfere with each other.

6. Mutations

Here we present individual heuristics and their mutations which come together to form two embedding heuristics located at the end of this chapter. They include comprehensive, high-level algorithms, whereas the individual strategies each focus on one specific aspect of the heuristics.

6.1. Super Vertex Extension Heuristic

This heuristic comes in two variants that work together. Both try to randomly extend super vertices by one vertex and take into account the local structure of the current embedding, while making sure that conditions (C1) and (C2) (see section 2.4) are met at all times. We intentionally do not incorporate any shortest path calculations (as compared to [6]) since we hope for quicker embeddings making use of only this simple heuristic based on random decisions and optimizing the number of successfully embedded edges.

6.1.1. Extend super vertex to free neighbor

Algorithm 6.1.1 formally depicts the steps of this variant of the heuristic. First, we choose a random node $i \in V(H)$ from our input graph H (line 2). Details of this selection are discussed in subsection 6.1.4. We then extend the subgraph induced by the super vertex $G[\phi(i)]$ by choosing a random node *source* in it (line 3) and trying to embed an edge to a randomly chosen free neighbor *target* (line 4, line 6). This neighbor is therefore included in the set $\phi(i)$ (line 7). A *free* neighbor *target* of *source*, formally $target \in N_G(source)$, is a neighbor node of *source* that is not used in any super vertex placement yet. More formally, the neighbor has to meet the following constraint: $\forall i \in V(H) : target \notin \phi(i)$. If there are no free neighbors, this mutation fails (line 5).

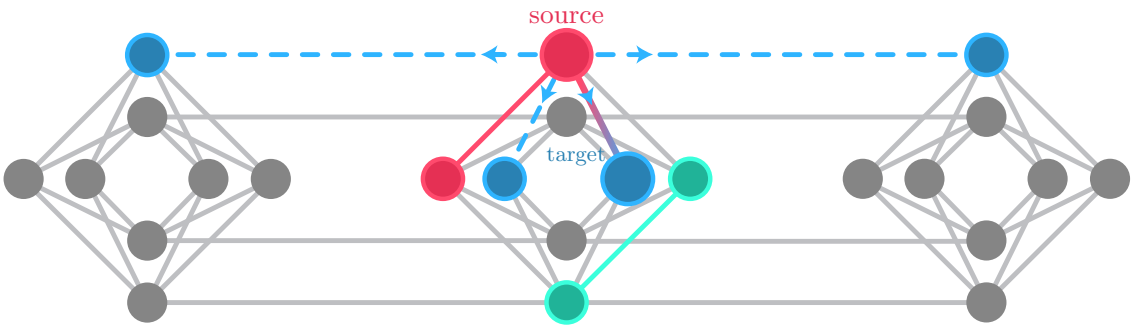


Fig. 6.1.: Extend random super vertex to free neighbor

Figure 6.1 illustrates how the extension works. $G[\phi(i)]$ is marked in red. In this example, *source* is randomly chosen in the red super vertex. Its free neighbors are marked in blue. Note that there is also another embedding drawn in green, thus one neighbor of *source* is not free. We randomly choose vertex *target* for the extension and include it in $\phi(i)$. Using this mutation, the embedding is able to spread out in the graph, i. e. onto multiple chimera cells. This is crucial, so that super vertices which were previously far apart from each other can share an edge eventually.

Algorithm 6.1.1: Extend super vertex to free neighbor

Input: H: Input graph, G: Hardware graph, ϕ: Super vertex placement

1 **Function** *extendToFreeNeighbor*:
2 $i \leftarrow$ chooseRandomBiased($V(H)$)
3 $source \leftarrow$ chooseRandom($\phi(i)$)
4 $targets \leftarrow$ getFreeNeighbors($source$)
5 **if** $targets = \varnothing$ **then return**
6 $target \leftarrow$ chooseRandom($targets$)
7 $\phi(i) \leftarrow \phi(i) \cup \{target\}$

6.1.2. Extend super vertex to embedded neighbor by shifting

While the first variant is useful to let super vertices extend to free neighboring vertices, we also want them to expand onto neighbors that are *already embedded*, i. e. used by another super vertex. This is necessary, so that super vertices "stuck" in the middle of

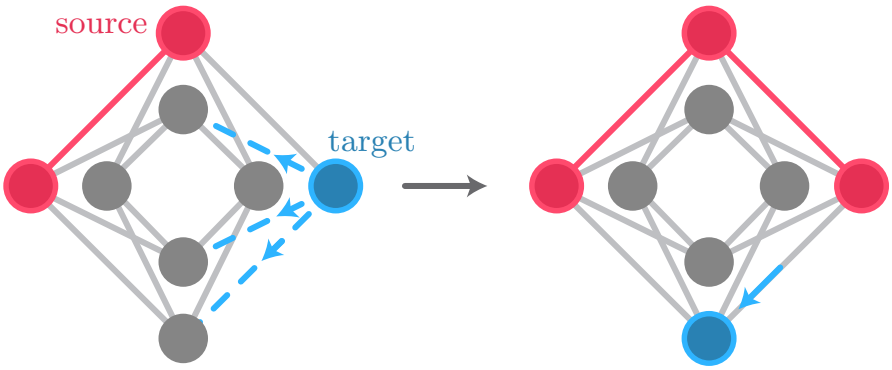


Fig. 6.2.: Extend random super vertex to embedded neighbor by shifting

other super vertices are able to "push aside" neighbors to create space that they can fill themselves.

Figure 6.2 illustrates the result of this approach. The topmost vertex (in the red super vertex) is chosen as source, while the only vertex in the blue super vertex is selected as target. The blue vertex is shifted to a free neighboring vertex, so that the red super vertex can expand onto the vacant position.

The steps for this approach are shown in Algorithm 6.1.2. Similar to the first variant, we choose a random node $i \in V(H)$ from our input graph H as described in detail in subsection 6.1.4. We select a node from the corresponding super vertex $\phi(i)$ as *source* and find a neighboring node as *target* which is already embedded but not in $\phi(i)$, that is, not in the same super vertex as *source*. This is done in the method "chooseSourceAndTarget" which we call in 2 and is essentially equivalent to lines 3 to 6 in Algorithm 6.1.1.

Next, we try to extend super vertex $\phi(i)$ onto the *target* while shifting *target* to one of its free neighbors. Let us denote the resulting position of *target* by $sTarget$ which stands for "shifted target". We try out multiple neighbors as candidates for shifted targets in a loop (line 5) since not every neighbor will be feasible. This is due to edges between *target* and its *embedded* neighbors $targetNeighbors$ (line 4) in G that have to be preserved. However, this constraint can be loosened as *target* itself might reside in a super vertex consisting of more than one node as shown in Figure 6.3. We therefore only need to preserve edges between $superVertexTarget$ (not: *target*) and $targetNeighbors$. Whether a shifted target is suitable or not is checked in the loop starting in line 8. Note

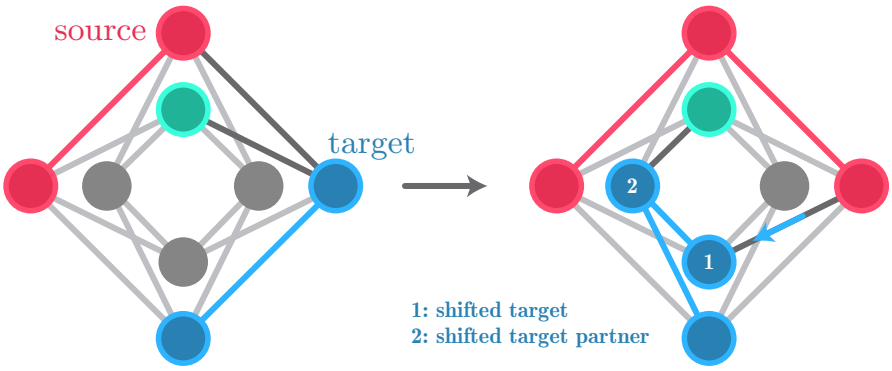


Fig. 6.3.: Extend random super vertex to embedded neighbor by shifting and constructing another super vertex

how we constructed the $tempSuperVertex$ in line 7 by adding the $sTarget$ and removing $target$ from $superVertexTarget$.

In Figure 6.3, we won't find a free neighbor of $target$ allowing us to embed the respective edges as described in the previous paragraph. This means we cannot reach the green node from any shifted target candidate, nor from the lowest blue node. Instead of letting the mutation fail, we try to construct a new super vertex consisting of the $sTarget$ and a free neighbor of $sTarget$, which we refer to as "shifted target partner" (line 10). The $tempSuperVertex$ is updated accordingly in line 11. If we still cannot reach all $targetNeighbors$, we break out of the check (line 13) and try a new node as shifted target candidate.

If we were able to reach all $targetNeighbors$ (line 15), we conclude this mutation by committing the new changes: we include $target$ in the red super vertex (line 16) and overwrite the super vertex where $target$ was previously part of with our $tempSuperVertex$ (line 17). Edges are then embedded in line 18.

6.1.3. Articulation Points (Cut vertices)

The outcome of the mutation described in the previous section has to meet condition (C1) (see section 2.4), that is, all induced graphs by the super vertices have to be connected, including the graph induced by the new blue super vertex (see Figure 6.3). Note how (C1) is violated in the example illustrated in Figure 6.4. We would silently pass the

Algorithm 6.1.2: Extend random super vertex to embedded neighbor by shifting

Input: H: Input graph, G: Hardware graph, ϕ: Super vertex placement

Function *extendToEmbeddedNeighborByShifting*:
  $(source, target) \leftarrow$ chooseSourceAndTarget(H, G, ϕ)
  $superVertexTarget \leftarrow$ getSuperVertexOf$(target)$
  $targetNeighbors \leftarrow$ getEmbeddedNeighbors$(target)$
  foreach $sTarget \in$ getFreeNeighbors$(target)$ **do**
    $reachable \leftarrow []$
    $tempSuperVertex \leftarrow superVertexTarget \cup \{sTarget\} \setminus \{target\}$
    foreach $neighbor \in targetNeighbors$ **do**
      if not canReachFromVertices$(neighbor, tempSuperVertex)$ **then**
        $sTargetPartner \leftarrow$ chooseRandom(getFreeNeighbors$(sTarget)$)
        $tempSuperVertex \leftarrow tempSuperVertex \cup \{sTargetPartner\}$
        if not canReachFromVertices(neighbor, tempSuperVertex) **then**
          break
      $reachable \leftarrow reachable \cup \{neighbor\}$
    if $reachable = targetNeighbors$ **then**
      $\phi(i) \leftarrow \phi(i) \cup target$
      Overwrite super vertex of *target* with $tempSuperVertex$
      embedEdges(from: $superVertexTarget$, to all: $n \in targetNeighbors$)
      return success
  return failure

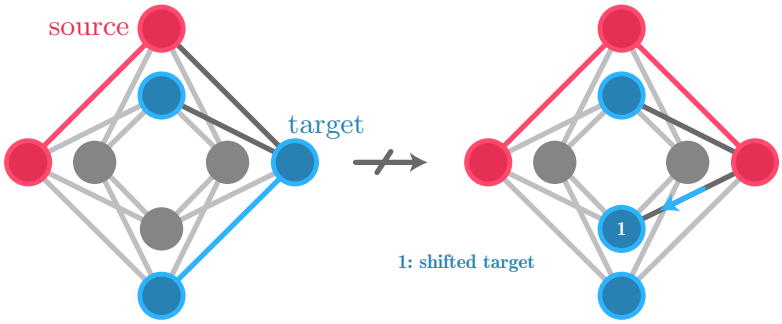


Fig. 6.4.: Violation of (C1) when disregarding articulation points

checks in Algorithm 6.1.2 as there are no embedded target neighbors ($targetNeighbors$ would evaluate to the empty set in line 4).

We mitigate this problem by computing **articulation points** (or *cut vertices*) inside the method "chooseSourceAndTarget" (called in line 2). In our case of a connected undirected graph, an articulation point is a vertex that – once removed (together with its adjacent edges) – disconnects the graph. If *target* is an articulation point, we might try to find a new $(source, target)$ pair or mark the mutation as failed. Note that articulation points can be calculated in linear time using Hopcroft and Tarjan's algorithm [45].

Nevertheless, the induced graph might still be disconnected, even with the computation of articulation points. Such a case occurs if we were to remove the green vertex in Figure 6.3 and apply the mutation again. This is because *target* is *not* an articulation point in this case (since removing it would not result in a disconnected graph). We need to take into account this special case as well. One approach is to check for graph connectivity after every mutation.

6.1.4. Select random super vertices

Both variants start the selection of a random super vertex (a random vertex $i \in V(H)$ from our input graph H). Performing this random selection based on a uniform distribution did not yield good embeddings as we do not factor in the current state of the super vertex placement. Some super vertices might have all edges adjacent to them in H embedded in the hardware graph G, i. e. there are respective edges in G between the graphs induced by the super vertices. It is therefore not reasonable to select these super vertices with

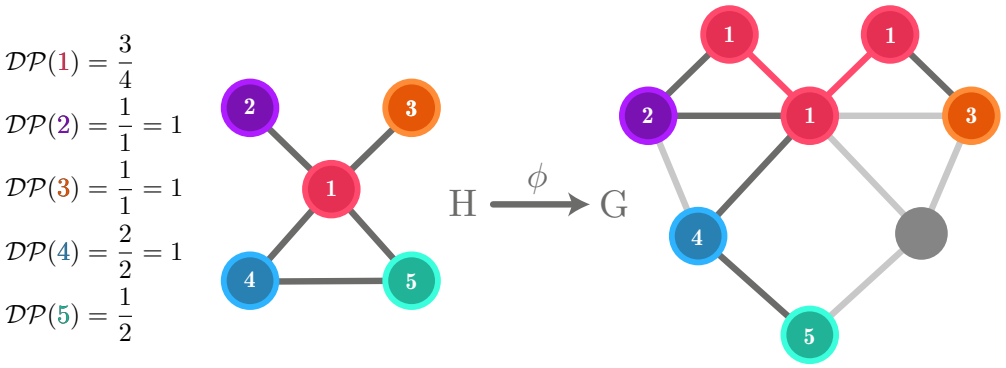


Fig. 6.5.: Example for degree percentage $\mathcal{DP}$ calculations

the same probability as others. Instead, we weight the chances to select super vertices based on how many percent of adjacent edges in H are embedded in G. More formally, we call this metric **embedded degree percentage** $\mathcal{DP}$ and define it as follows

$$\mathcal{DP}(i) = \frac{\left|\left\{\{i,j\} \in E(H) : j \in N_H(i), \{u,v\} \in E(G), u \in \phi(i), v \in \phi(j)\right\}\right|}{\deg(i)} \tag{6.42}$$

where $N_H(i)$ are the neighboring vertices of $i \in V(H)$. In the fraction, we divide the number of *actual* embedded edges by the number of *expected* edges (if all adjacent edges to i were successfully embedded). Note that we find a valid minor embedding and thus satisfy (C3) (see section 2.4) if and only if $\forall i \in V(H) : \mathcal{DP}(i) = 1$.

Figure 6.5 clarifies the calculation of the embedded degree percentages. The red super vertex (number 1) has a degree of 4. However, in the contrived heart hardware graph, $\phi(1)$ is missing an edge to the green vertex (number 5). The degree percentage thus results in 75 %. Note that it does not matter for the calculation that $\phi(1)$ (red nodes in G) shares two edges with $\phi(2)$ (purple node in G).

With this new metric, we can subsequently replace the uniform distribution for the selection of random super vertices with our custom distribution that we gain as follows: Choose super vertex $i \in V(H)$ with normalized probability

$$P(i) = \frac{1 - \mathcal{DP}(i)}{\sum_{i \in V(H)} (1 - \mathcal{DP}(i))} \tag{6.43}$$

such that high degree percentages lead to a lower probability that the corresonding super vertex will be chosen and super vertices with a low degree percentage are favored during the selection.

Furthermore, we assign an increased chance of being selected to small super vertices, that is, super vertices i where $|\phi(i)|$ is small compared to other super vertices. The smallest one is allocated an increased chance of 70% of the maximum degree value, linearly going down to an increased chance of 0% for the biggest super vertex. The chances are then normalized resulting in a reduced likelihood of selecting big super vertices. See Figure 7.8 as an example.

6.2. Remove redundant vertices

Some embedded nodes on the hardware graph might be redundant in a sense that they do not contribute to the embedding of any new edges between super vertices (or to be clear: between their respective induced subgraphs on G). This is especially true for the first variant of the *super vertex extension heuristic* presented in section 6.1 where long chains might be constructed. This leads to some embedded super vertices occupying a great portion of the whole hardware graph, which in turn poses a problem as all mutations from section 6.1 might fail in such a case, even when we repeatedly apply them. To prevent failure of the whole heuristic, we remove redundant vertices which results in shorter chains, that is, a smaller size of the induced subgraphs by the super vertices.

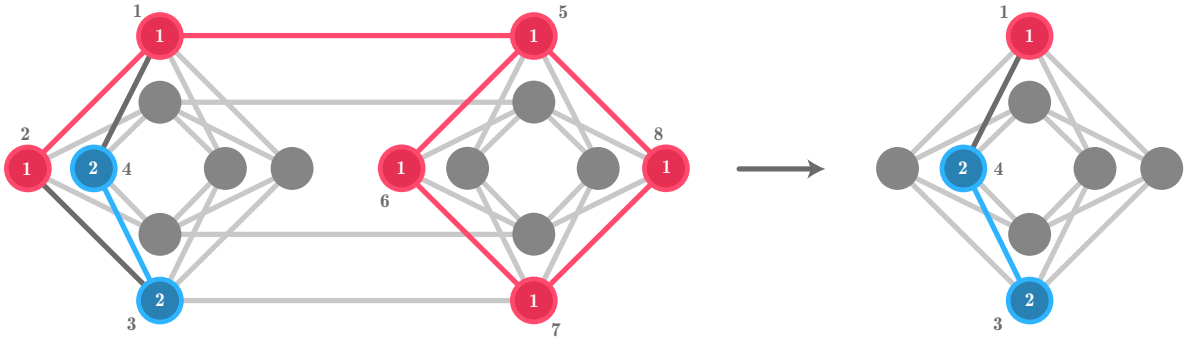


Fig. 6.6.: Remove redundant vertices mutation example

Consider the current embedding in Figure 6.6. The induced graph for vertex 1 is marked in red. It has spread out covering multiple chimera cells. When we apply this mutation, only one red vertex remains, encoding the same information as beforehand, but with less vertices. Observe that apart from vertices 5, 6, 7 and 8, also vertex 2 has been removed from the mapping $\phi(1)$ since there is already an edge between the red and the blue super vertex, namely edge $\{1, 4\}$. As a result of the mutation, we made room for new embeddings that can now use the previously occupied nodes (the right chimera cell is not shown anymore in G since it does not host any embedded nodes anymore).

Algorithm 6.2.1: Remove redundant vertices

Input: H: Input graph, G: Hardware graph, ϕ: Super vertex placement

```
1  Function removeRedundantVertices:
2      for i ∈ V(H) do
3          superVertexNodes ← getEmbeddedNodes(i)
4          if |superVertexNodes| = 1 then return
5          removedNodes ← {}
6          while we removed a vertex in the current iteration do
7              ap ← calcArticulationPoints(superVertexNodes ∖ removedNodes)
8              for nodeToRemove ∈ superVertexNodes do
9                  if nodeToRemove ∈ removedNodes
10                     or nodeToRemove ∈ ap
11                     or not canReachFromVertices(nodeToRemove)
12                 then continue
13                 removeNode(nodeToRemove)
14                 removedNodes ← removedNodes ∪ {nodeToRemove}
15                 if |removedNodes| = |superVertexNodes| − 1 then return
16                 break
```

Algorithm 6.2.1 outlines the steps of this mutation. We iterate through every super vertex (line 2) and try to remove its embedded nodes (line 3) from the hardware graph. We check for every embedded node (line 8) if it is feasible to remove it, i. e. the *nodeToRemove* is no articulation point and we can reach all previously reachable neighbors of *nodeToRemove*. Similar to section 6.1, neighbors stand for their own super vertex. In Figure 6.6, the neighbor of node 2 is reachable from node 1, as there is an edge between the red node 1 and the whole blue super vertex (edge $\{1, 4\}$). If we pass all conditions, we remove *nodeToRemove* and move on with the next node in the current super vertex.

It is crucial to recalculate the articulation points (line 7) every time a node is removed from a super vertex. See Figure 6.6 as example: node 6 and 8 are no articulation points. However, we must not remove both of them before node 7, as this would split the red graph into two partitions. Therefore, remove node 6 and recalculate the articulation points afterwards. We now find that node 8 has indeed become an articulation point.

6.3. Reduction mutation

Apart from extend and shift mutations that try to raise the chance that edges can be embedded by increasing the size of a super vertex, we introduced a reduction mutation based on the heuristic presented in section 4.6. The mutation can be applied once the super vertex of a vertex $v \in V(H)$ is connected to all super vertices of neighbors $N_H(v)$. In fact, after $|V(H)|$ iterations of heuristically solving the SVEP, the reduction mutation can be applied to every super vertex.

Clearly, the mutation aims at improving the fitness of a super vertex by reducing the number of overlappings and the overall size of the super vertex.

6.4. Bomb algorithm mutation

Chapter 5 was devoted to introducing the LMRP – a subproblem that aims at improving local part of the embedding by destroying it to then "repair" that part. The problem was designed to only use local information to enable us to apply the mutation in parallel on different parts of the graph without them interfering with each other.

In section 5.2, we found that the problem is in general $\mathcal{NP}$-hard, for Pegasus and King's graphs, even if the graph has constant size. Hence, we restrict ourselves to constant-size subgraphs and use the heuristic developed in section 5.3.
The actual subgraph can be configured through a functor and might, in general, not be connected which does not pose a problem. For the hardware graphs presented in section 2.5, we assume a labelling of every vertex. On Chimera and Pegasus graphs, by default, a 2×2 lattice is chosen as the subgraph to be destroyed while for King's graphs a 5×5 grid is chosen as the subgraph.

A mutation is applied in the last phase of the embedding process to reduce the number of overlapping super vertices. An overlap vertex is identified, then the mutation on a subgraph containing the vertex is scheduled.

6.5. Embedding Heuristic 1

The embedding algorithm, presented in this section, relies on two invariants that were mentioned in sections 4.6 and 4.7:

(i) Every super vertex remains connected at any time.

(ii) A super vertex is always adjacent to all its embedded neighboring super vertices.

These invariants motivated the formal definition of the SVEP and the algorithm iteratively solves such a subproblem. This algorithmic idea is essentially the same as in [6]. Apart from that, overlapping super vertices are tolerated but the overlapping count is minimized throughout the algorithm. In every iteration, a SVEP is solved to add a new super vertex for some vertex $v \in V(H)$. The solving of the SVE subproblems relies on the flow-based heuristic described in section 4.5 that produces an initial placement which is refined using the heuristic from section 4.6. As discussed in section 4.5, applying the flow-based heuristic really only makes sense if at least two neighbors of v are embedded. If none of the neighbors was embedded, a random vertex in G is chosen as the image of v in ϕ while for the case that exactly one neighbor in H is embedded, an adjacent vertex is chosen.

To increase the chance that a valid embedding is found, after solving the SVE subproblem, mutations are carried out that extend super vertices to free neighbors or shift them to another vertex as presented in section 6.1. Apart from that, in the mutation procedure, every vertex, that has a super vertex and whose neighbors are embedded, is reduced, see 6.3. And, additionally, for overlapping super vertices, we introduce a bomb mutation as described in section 6.4, essentially solving the LMRP.

Finally, after $|V(H)|$ iterations solving the SVEP and mutating ϕ, algorithm 6.5.1 tries to reduce the overlaps by successively applying the heuristic developed in section 4.7. That is done at most k times on all overlapping super vertices.

Note that for algorithm 6.5.1, we define $k \in \mathbb{N}$ to be some constant, and we denote $R(v)$ as the reverse mapping of ϕ, i. e. for some vertex $v \in V(G)$, $R(v)$ is the set of all vertices in H that are mapped to v in ϕ.

Algorithm 6.5.1: Embedding algorithm based on iteratively solving the SVEP.

Input: H, G: Graph
Function *find_embedding*:
  $\phi(v) \leftarrow \varnothing$, $\forall v \in V(H)$
  foreach $v \in V(H), \phi(v) = \varnothing$ **do**
    SolveSVEP(H, G, ϕ, v)
    Mutate(H, G, ϕ, v)
  ReplaceOverlapping(H, G, ϕ)
  return ϕ

Input: H, G: Graph, ϕ: $V(H) \to 2^{V(G)}, v \in V(H)$
Function *SolveSVEP*:
  $neighbors \leftarrow \{ w \in N_H(v) : \phi(w) \neq \varnothing \}$
  if $|neighbors| = 0$ **then**
    $\phi(v) \leftarrow \{ GetRandom(V(G)) \}$
  else if $|neighbors| = 1$ **then**
    $\phi(v) \leftarrow \{ GetBestAdjacent(v) \}$
  else
    $placement \leftarrow$ solve MCFP instance given SVEP instance (H, G, v, ϕ)
    try improving *placement* using heuristic from section 4.6
    $\phi(v) \leftarrow placement$

Input: H, G: Graph, ϕ: $V(H) \to 2^{V(G)}, v \in V(H)$
Function *Mutate*:
  $mutationQueue :=$ Queue< Mutation> ()
  foreach $w \in N_H(v), \phi(w) \neq \varnothing$ **do**
    $mutationQueue.push(ExtendMutation(H, G, \phi, w))$
    $mutationQueue.push(ShiftMutation(H, G, \phi, w))$
  foreach $w \in N_H(v), \phi(w) \neq \varnothing : \phi(u) \neq \varnothing, \forall u \in N_H(w)$ **do**
    $mutationQueue.push(ReductionMutation(H, G, \phi, w))$
  if $\exists p \in \phi(v) : |R(p)| \geq 2$ **then**
    $mutationQueue.push(LMRPMutation(H, G, \phi, p))$
  $mutationQueue.shuffle()$
  foreach $mutation \in mutationQueue$ **do**
    if $mutation.isValid()$ & $mutation.isImproving()$ **then**
      $mutation.apply()$

Input: H, G: Graph, ϕ: $V(H) \to 2^{V(G)}$
Function *ReplaceOverlapping*:
  for $i = 1$ to k **do**
    $overlapping \leftarrow \{ v \in V(H) : |R(u)| \geq 2, u \in \phi(v) \}$
    foreach $v \in overlapping$ **do**
      try improving placement $\phi(v)$ using heuristic from section 4.7

6.6. Embedding Heuristic 2

Similar to the first embedding heuristic (see Algorithm 6.5.1), the algorithm presented in this section also adheres to the invariants mentioned in sections 4.6 and 4.7 and first formalized in section 2.4, i. e. we ensure that conditions (C1) and (C2) are met at all times. In contrast to the first embedding heuristic, we do *not* allow vertices to overlap. This heuristic is based on an evolutionary approach where we generate a new population by mutating the parent genes and selecting the best children of the offspring as next generation. However, we only include the very best mutation of all descendants (children), turning this heuristic into a greedy, probabilistic approach to the embedding problem. We consider the current embedding state, including the current super vertex placement ϕ as gene of an individual.

In Algorithm 6.6.1, an initial embedding (line 2) is found using a depth-first search. We then generate a new population by randomly switching between the variants of the super vertex extension heuristic in section 6.1 (see lines 14, 16). If all mutations failed, we apply the mutation from section 6.2 to remove redundant vertices and then try to find valid mutations again (line 19). We also remove redundant vertices with a certain probability, even if the mutations succeeded (Algorithm 6.6.1) in order to avoid being stuck in a local optimum. The best child is then chosen to form the next "generation", while selecting the one where the most new edges from H are embeddable in G (line 27)[1].

As can be seen from the algorithm, this heuristic is subject to many parameters which need to be adjusted in order to find good embeddings for different input graphs H. This is mainly due to probabilities involved that should help finding good embeddings without being stuck in local optimum for too long. Removing redundant vertices in the case of failure (line 19) can be seen as "last trial" before the whole algorithm fails. Switching between the two variants of the extension heuristic (lines 14, 16) ensures that while long chains can form, super vertices can also expand from "within", that is, they can shift surrounding super vertices away as discussed in subsection 6.1.2. Additionally, the chain lengths cannot become unbalanced since we choose super vertices for the extension mutations according to their degree percentage ($\mathcal{DP}$, see subsection 6.1.4). Apart from parameters concerning probabilities, we can also adjust those that refer to the evolution, e. g. the population size or how many mutations are performed before failing.

[1] new edges: edges that were not yet embedded in G prior to the current mutation

Algorithm 6.6.1: Evolutionary approach to the embedding problem based on super vertex extensions (see section 6.1)

Input: H: Input graph, G: Hardware graph, ϕ: Super vertex placement

Function *evolution*:
- initializeEmbedding()
- **for** $i \in \{1 \dots params.\text{maxGenerations}\}$ **do**
  - $child \leftarrow$ generatePopulationAndSelect()
  - commit($child$)
  - **if** $child$.isValidEmbedding() **then**
    - $child$.removeRedundantVertices()
    - **return** $child$

Function *generatePopulationAndSelect*:
- $population$: Set of embeddings $\leftarrow \varnothing$
- **for** $i \in \{1 \dots params.\text{populationSize}\}$ **do**
  - **for** $i \in \{1 \dots params.\text{maxMutationTrials}\}$ **do**
    - **if** random() $<$ $params$.extendToFreeNeighborProbability **then**
      - $child \leftarrow$ extendToFreeNeighbor (see 6.1.1)
    - **else**
      - $child \leftarrow$ extendToEmbeddedNeighbor (see 6.1.2)
  - **if** all mutations failed **then**
    - $child$.removeRedundantVertices() (see 6.2)
    - Try to do the mutations again
  - $population \leftarrow population \cup \{child\}$
- $bestChild \leftarrow$ selectBest($population$)
- **if** random() $<$ $params$.removeRedundantProbability **then**
  - $bestChild$.removeRedundantVertices() (see 6.2)
- **return** $bestChild$

Input: $population$: Set of embeddings

Function *selectBest*:
- **foreach** $child \in population$ **do**
  - $improvement \leftarrow child$.howManyNewEdgesAreEmbeddable()
- **return** $child$ with best improvement

7. Implementation & Evaluation

In this chapter, we evaluate the results of our implementations of the heuristics presented in the previous chapter. Our source code is publicly available on GitHub[1] including the scripts to reproduce the results shown here.

7.1. Object-oriented architecture

The complexity of the embedding algorithms – as concerns the plethora of data structures that have to be filled and carried around – makes the construction of a reliable and maintainable architecture an imperative task. Different levels of abstraction are needed: Graph data structures form the backbone of our algorithms and are used on higher levels to apply common algorithms like depth-first search on them. We base all of our calculation on these graph structures. Furthermore, the super vertex placement function ϕ, which we referred to extensively in this work, has to be represented in code as a mapping between two different graphs. The architecture should allow for a great reusability of components residing in lower layers, e. g. trying out multiple mutation strategies must not require implementing the necessary data structures to translate ϕ into program code again.

An overview over the architecture used for the Python implementation of the heuristic presented in section 6.6 is depicted in Figure 7.1. We split the code into multiple modules to ease the adoption of the onion model stemming from the concept of a clean architecture: inner layers should not depend on outer layers, that is, in our overview, dependencies should only point downwards *to* the next lower situated (inner) layer.

Graph module. It forms the lowest layer where we define important domain objects like an *UndirectedGraph* making use of a symmetric adjacency list. A *ChimeraGraph is* an *UndirectedGraph* which only additionally implements its own initialization (construction of an $m \times n$ grid of Chimera cells). Further topologies like the Pegasus and King's graph could

[1] Link to our GitHub repository: https://github.com/MinorEmbedding/graph-embedder

be added to this layer as well. We also define various test graphs here. This module offers methods like (among others) "getVertices", "getEdges" and "removeAllEdgesFromNode".

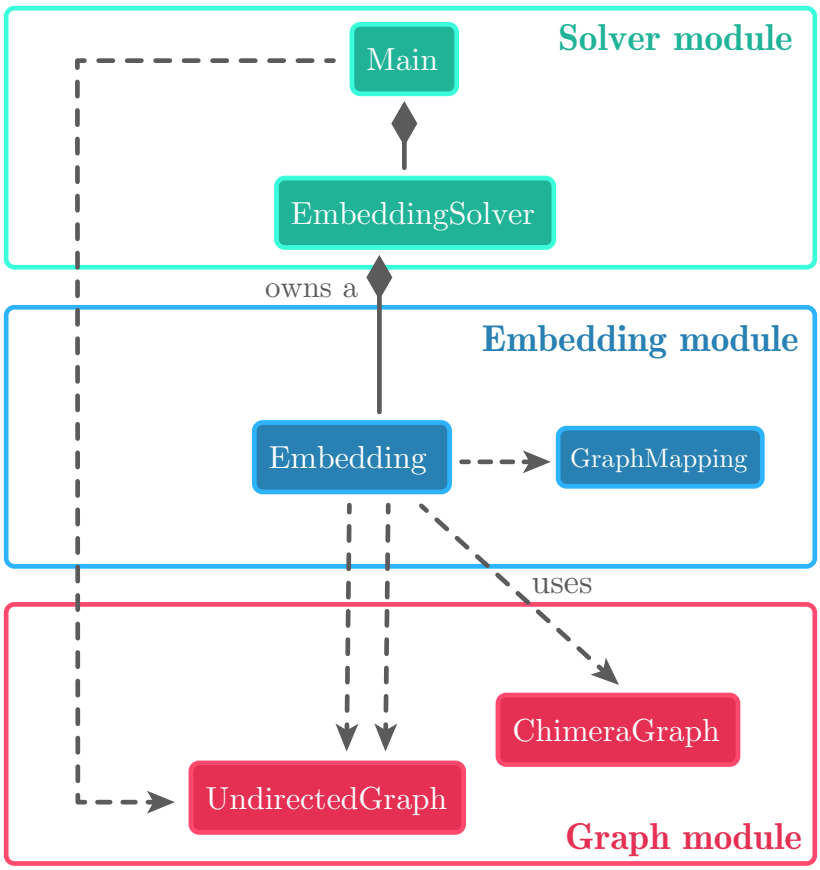


Fig. 7.1.: Object-oriented architecture overview (UML diagram)

Embedding module. This layer makes it easy for the solver module to deal with the super vertex placement mapping ϕ. Therefore, an *Embedding* uses a *GraphMapping* which is a unique dictionary with sets as values to encode the mapping between nodes in the input graph H and the respective embedded nodes in the hardware graph G (offering $\mathcal{O}(1)$ access for both mapping "directions"). Besides this mapping, the *Embedding* stores the layout graph H itself (*ChimeraGraph*), the hardware graph G (*UndirectedGraph*) and a graph similar to H but only with those edges that are already embedded in G. This way, if we keep all these structures in sync, we are able to easily calculate how many new edges from H can be embedded in the current state (which is needed for section 6.6). Note that we make a deep copy of the *Embedding* object when we generate

a new population. This behavior could be improved, e. g. we do not need to copy the whole Chimera layout again every time we do a mutation. This module offers methods like (among others) “embedEdge”, “getFreeNeighbors”, “getEmbeddedNeighbors” and “getVerticesOfSuperVertex”.

Solver module. In this module, we implement the actual heuristics, i. e. the mutations presented in chapter 6. The solver only needs to own an *Embedding* object to access all relevant functions like “embedEdge”. This way, the code in the topmost layer is much better maintainable compared to a solution where the solver had to build up, fill and keep in sync the underlying data structures on its own all the time. This is because the code in the solver module can now focus on the relevant tasks, i. e. implement different mutations and switch between them based on a random number. Furthermore, we also define the main method here. This module offers methods like (among others) “extendRandomSuperVertexToFreeNeighbors”, “calcArticulationPoints”, “initializeEmbedding” and “generateChild”.

Drawing module. This additional module encompasses drawing functionality to output SVG files depicting the current embedding state which is helpful for debugging purposes. Methods like (among others) “drawEmbedding”, “changeBrightnessOfColor”, “drawChimeraGraph” and “drawNode” are offered and used in the main function of the solver module.

Note how some method names suggest different abstraction levels inside the modules as well. Not all methods presented here are actually public and can be accessed by other modules. Abstraction levels on the granularity of modules are realized with different folders, while abstraction inside a module is ensured by splitting up code into multiple files and/or by introducing subfolders.

7.2. Results for Heuristic 1

We tested the embedding heuristic 1 presented in section 6.5 on page 73 on both Chimera and Pegasus graphs with input graphs ranging from complete graphs, TSP graphs to Erdös-Rényi graphs. We will present the results in this section and analyze what can be improved. For each of the test cases, the algorithm was executed exactly 50 times for each configuration due to the fact that the algorithm is randomized.

Figure 7.2 shows the embedding probabilities of complete graphs with different sizes on Chimera graphs consisting of 16×16 lattices using two different cost models: constant and linear. The cost model refers to the cost imposed on an occupied vertex in the flow heuristic. In section 4.5, we defined the cost function $c(p, q)$ for a non-artificial arc from vertex p to vertex q as $\lambda|R(p)|+1$ such that if the vertex is occupied, the cost rises linearly with the number of vertices mapped onto it. We refer to this as the linear cost model while a constant cost $\lambda H(R(p))+1$, with H as the Heaviside function, for an occupied vertex will be referred to as the constant cost model.

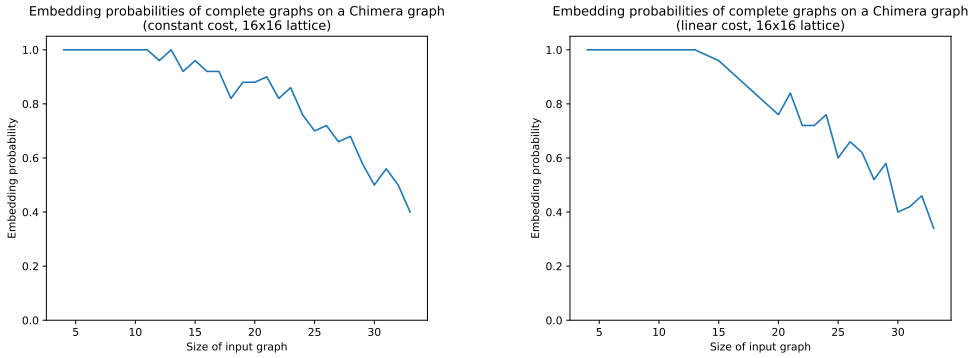


Fig. 7.2.: Embedding probabilities of complete graphs on 16×16 Chimera graphs with different cost models: Constant cost (left), linear cost (right).

Figure 7.2 shows that a constant cost is slightly advantageous but barely significantly better. In general, we observed that the heuristic does not fail due to the size of the Chimera graph but instead does not extend super vertices sufficiently. The limitation of the hardware graph is more severe in figure 7.3 as K_{28} was the largest graph that can be embedded using our first embedding heuristic. However, according to [6], the largest complete graph embeddable on a Chimera graph consisting of 8×8 unit cells is K_{33}. This

shows that the algorithm is not state-of-the-art as it does not surpass the minorminer algorithm [6].

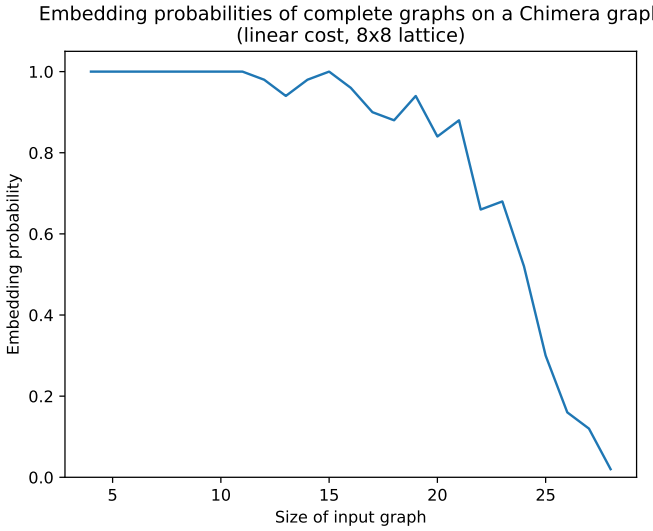


Fig. 7.3.: Embedding probabilities of complete graphs on 8×8 Chimera graphs with constant cost model.

Obviously, embedding complete graphs on a Pegasus graph is much simpler as the topology graph offers substantially better connectivity and vertices. Figure 7.4 visualizes the embedding probability of complete graphs on the Pegasus graph and indeed much larger graphs can be embedded. The issue here is the fact that the algorithm has bad runtime and therefore could not be practically used. There are multiple ways to improve the performance of the algorithm that we have not taken. These are bundling together extend and shift mutations respectively. That is, instead of successively extending or shifting a vertex, a short DFS could be conducted. The same applies to the `reduce` step in the replacement heuristic from section 4.6. Another improvement could be to adjust the graph created in the flow heuristic to encompass only a needed subset of the graph.

Lastly, we generated random Erdös-Rényi graphs to test the first embedding heuristic. For the graphs, the probability that there exists an edge between two distinct vertices was defined to be $p = 0.1$. In figure 7.5, for every n, we generated in each iteration a new random graph given the parameters and embedded them on a Chimera graph consisting of 16×16 unit cells. Starting at around $n = 36$ the chance to successfully embed a graph dramatically decreases.

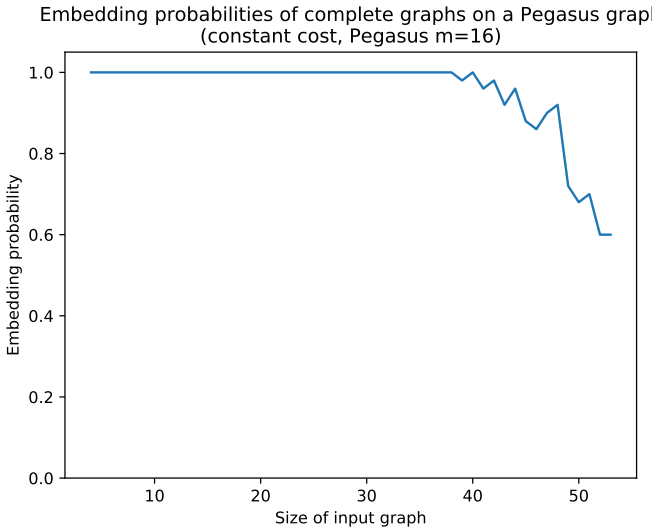


Fig. 7.4.: Complete graphs embedded on a Pegasus graph as found in a D-Wave Advantage system.

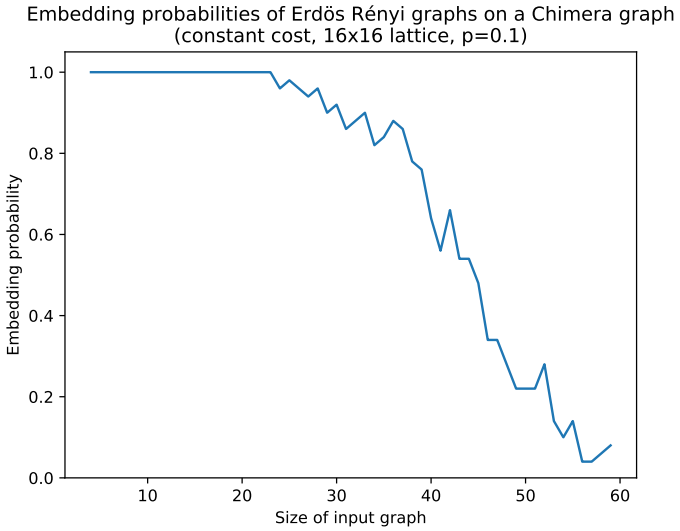


Fig. 7.5.: Random graphs, namely Erdös-Rényi graphs with $p = 0.1$, of different sizes embedded on a 16×16 Chimera graph.

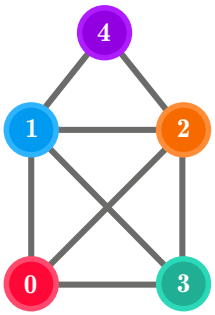
7.3. Results for Heuristic 2

We implemented the second heuristic (shown in section 6.6) in Python as it allows for quick prototyping and enabled us to quickly try out new ideas. section 7.1 explained the overall architecture of the Python implementation. During our tests, the "Main" component in the solver module was altered to test different parameter constellations.

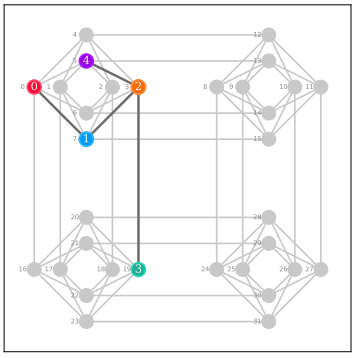
7.3.1. Embedding and degree percentage $\mathcal{DP}$ graphs

The $\mathcal{DP}$ metric was introduced in subsection 6.1.4 and allowed us to evaluate the current embedding state. In a valid embedding, the $\mathcal{DP}$ value has to read 1 for every super vertex, i. e. all edges between super vertices are successfully embedded in the hardware graph G. In the Python implementation, this graph (plotted with the library "matplotlib") is updated live every iteration, so that the user can track the progress. Additionally, SVG files displaying the hardware graph and the embedded nodes and edges are rendered in a user-specified rate, e. g. every 50 generations. The parameter is customizable as this I/O operation degrades the overall performance, yet, these images are vital for debugging purposes.

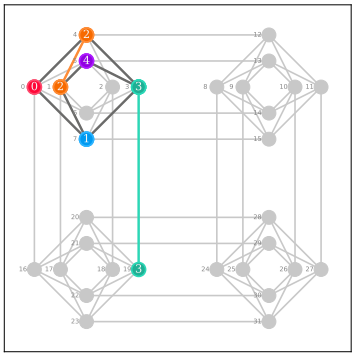
In the following, the crossed house puzzle (known as the "Haus vom Nikolaus" in Germany) serves as an easy-to-embed and easy-to-understand example (see Figure 7.6). The input graph H is depicted in a. The house is composed of a $K4$ graph as base and a $K3$ graph serving as roof. The initial embedding in b is constructed using a depth-first search. In the first generation c, the "extend to embedded neighbor" heuristic from subsection 6.1.2 is employed. The green super vertex 3, previously only on hardware node 19, extends to node 3 in G pushing aside a node from the orange super vertex 2 to node 4 in G. Super vertex 2 was previously connected to the purple super vertex 4, hence we have to construct another chain from node 4 to node 1 in the hardware graph to ensure this edge is preserved. In the second generation d, we apply the "extend to embedded neighbor" heuristic from subsection 6.1.2 and extend the green super vertex 3 to the free hardware node 6. Likewise, we make use of this mutation in the third generation e in order to extend super vertex 4 marked in purple. In f, unnecessary nodes are removed, in this case node 19 in G which corresponded to the green super vertex 3.



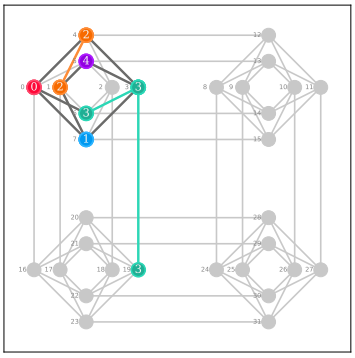
(a) The crossed house graph (Haus vom Nikolaus)



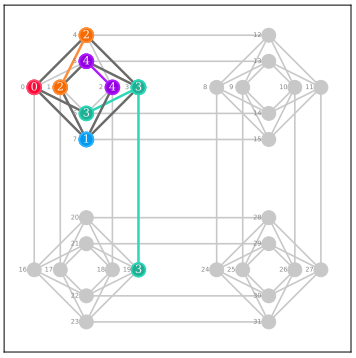
(b) Initial embedding



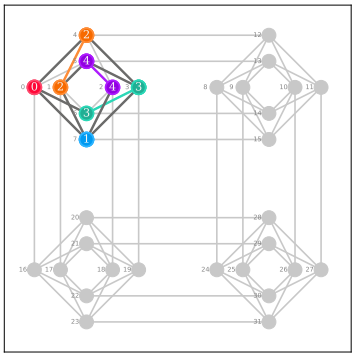
(c) First generation



(d) Second generation



(e) Third generation



(f) Third generation (unnecessary nodes removed)

Fig. 7.6.: Embedding of the crossed house puzzle on a 2x2 Chimera grid

Note how the green super vertex 3 was chosen for the extension mutation in the first generation (c). Figure 7.7 depicts the degree percentages (as discussed in subsection 6.1.4) for all five super vertices. In the initial embedding, super vertex 3 has indeed the lowest $\mathcal{DP}$ value increasing the likelihood of being selected in the first generation. The same reasoning applies to purple super vertex 4 which was selected in the third generation (e) as it had the lowest $\mathcal{DP}$ value in the previous (second) generation.

As discussed in subsection 6.1.4, we assign an increased chance of selection to small super vertices. Figure 7.8 depicts the selection chances of the crossed house embedding in the second generation. Note that super vertex 0 (red) and 3 (green) had the same degree percentage in the first generation, yet super vertex 0 is favored as it only consists of one node in the hardware graph, compared to two nodes for super vertex 3. This results in a decreased chance of selecting super vertex 4 (purple), as it is a small super vertex, yet,

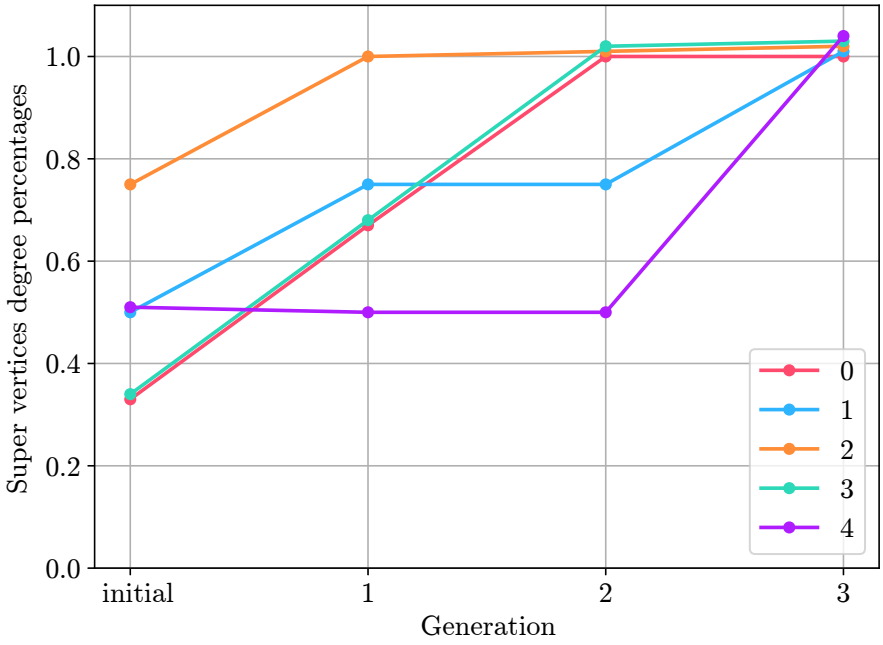


Fig. 7.7.: $\mathcal{DP}$ values for the crossed house embedding

increased chances are assigned in an ascending order of super vertex numbers (0 and 1 come before 4). This could also be randomized which would probably be beneficial for the embeddings, however, we did not implement this feature yet.

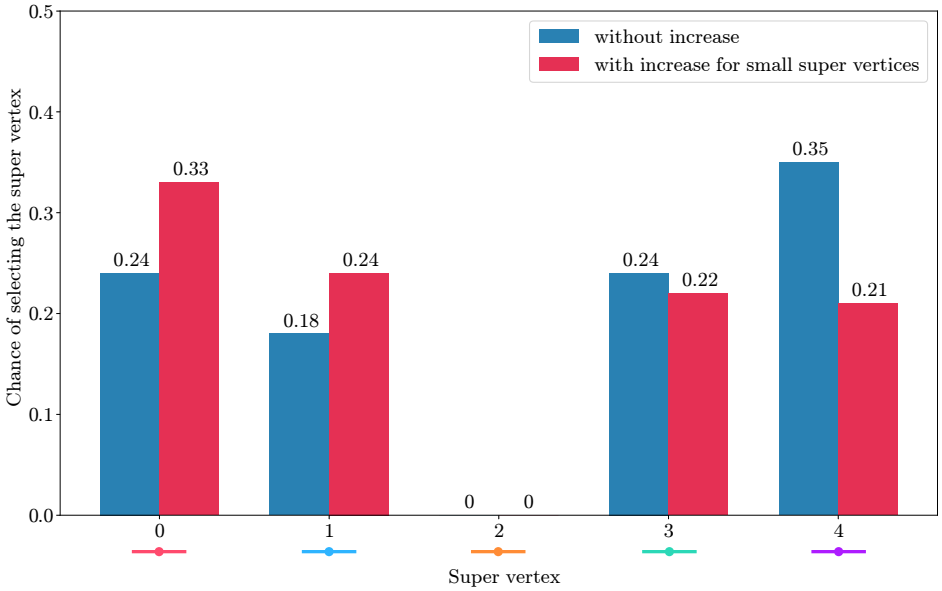


Fig. 7.8.: Selection chances in generation 2 (resulting from $\mathcal{DP}$ values of generation 1)

7.3.2. Embedding of K-graphs on different grid sizes

We tested the heuristic against complete graphs of various sizes on different Chimera grids. Not all parameters found in the captions of the following figures have been thoroughly discussed, however, they are present in the source code and can be freely adjusted. We've included them here to guarantee reproducibility of our measurements.

Figure 7.9 depicts the embedding of K-graphs on a 2x2 Chimera grid. While K_2 to K_6 is successfully embedded in every run (100 % embedding success probability), the rate rapidly drops to 0 % for K_{10} and bigger complete graphs – which was expected considering the small size of the hardware graph. The average number of necessary generations increases but stays below 35. For K_9, this value is not representative, as only six runs (3 %) yielded successful embeddings. Figure C.1 in the appendix gives more detailed insight into the distribution of generations needed to find a valid embedding. To name but a few execution times: using 4 cores and 90 % workload on an i7-6700 CPU (around 3.6 GHz), 200 iterations took around 8 seconds for K_2, 51 s for K_6, 218 s for K_7 and 592 s for K_8. Note that failed embeddings are generally quicker.

In Figure 7.10, we have left all parameters the same except for the grid size and tried to embed K-graphs on a 5x5 Chimera grid. Bigger complete graphs now have a higher probability of being successfully embedded, e. g. 20 % of embeddings are valid for K_{11} on the 5x5 grid, compared to 0 % on the 2x2 grid. Figure C.2 offers more fine-granular data. (Abysmal) Execution times on 4 cores on an i5-8350U CPU (around 2.3 GHz) with 80 % workload (100 runs each, the other 100 runs were carried out on another computer) amount to around 109 s for K_5, 3722 s for K_8 and 3449 s for K_{13}.

A 16x16 Chimera grid was used to collect the data presented in Figure 7.11. Unlike expected, the heuristic performs worse compared to the 5x5 Chimera grid, e. g. K_8 was embedded in 81.0 % of the 200 runs on a 5x5 grid, but only 73.0 % of the 100 runs on a 16x16 grid. One reason could be, that chains expand too broadly to "find back" to other super vertices. A detailed analysis was not carried out yet for this case. Like before, an additional plot is included in the appendix (see Figure C.3). Note that it took 20155 s on the i7 CPU to run the heuristic 100 times for K_{13}.

We've included the actual outcome of the program (SVG files) in the appendix: Figure C.4 and Figure C.5 both depict a valid embedding of the K_8 graph onto a 5x5 Chimera grid.

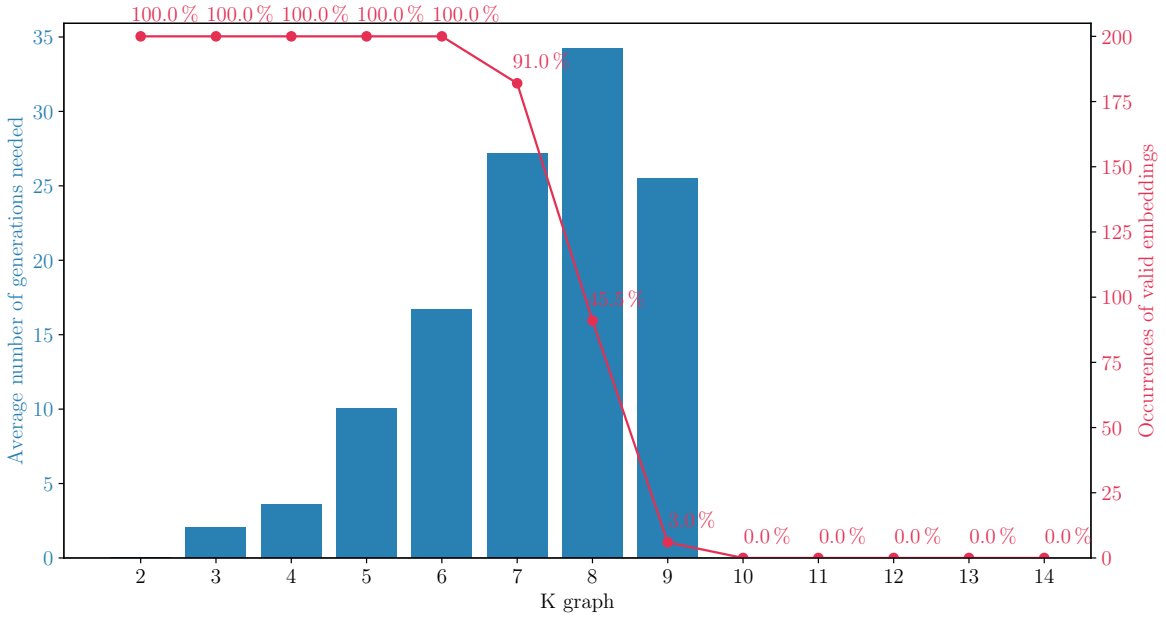


Fig. 7.9.: Embedding complete graphs on a 2x2 Chimera grid. 200 runs for every K_i, population size: 6, max mutation trials: 30, probability to choose mutation "extend to free neigbors": 0.24, max generations before failure: 600, remove redundancy probability: 0.01

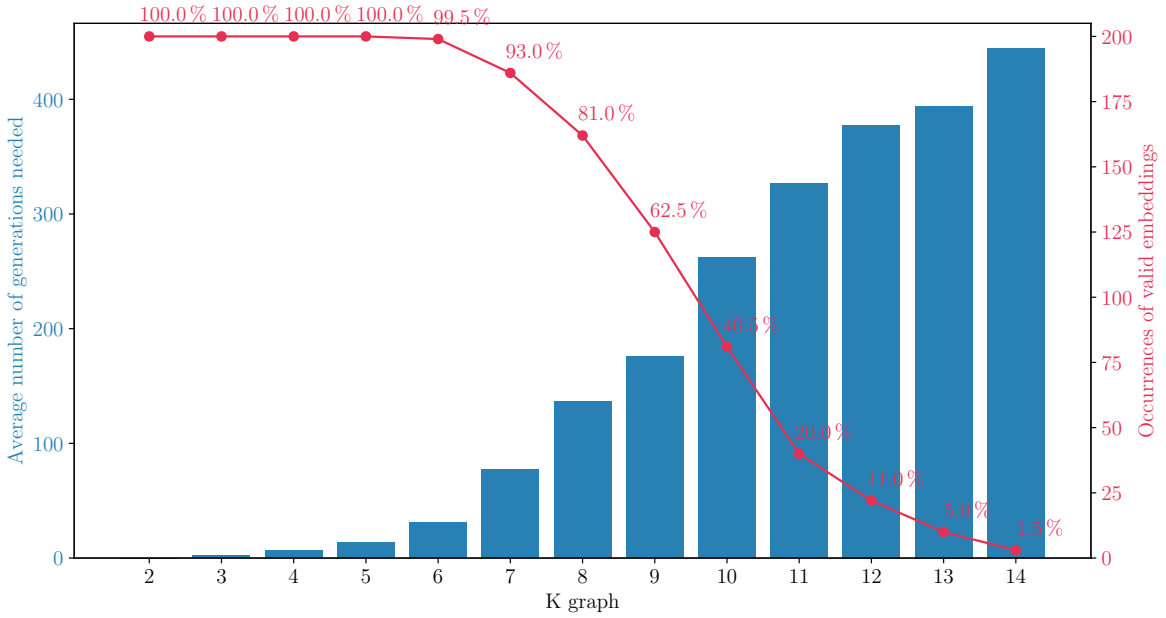


Fig. 7.10.: Embedding complete graphs on a 5x5 Chimera grid. 200 runs for every K_i, population size: 6, max mutation trials: 30, probability to choose mutation "extend to free neigbors": 0.24, max generations before failure: 600, remove redundancy probability: 0.01

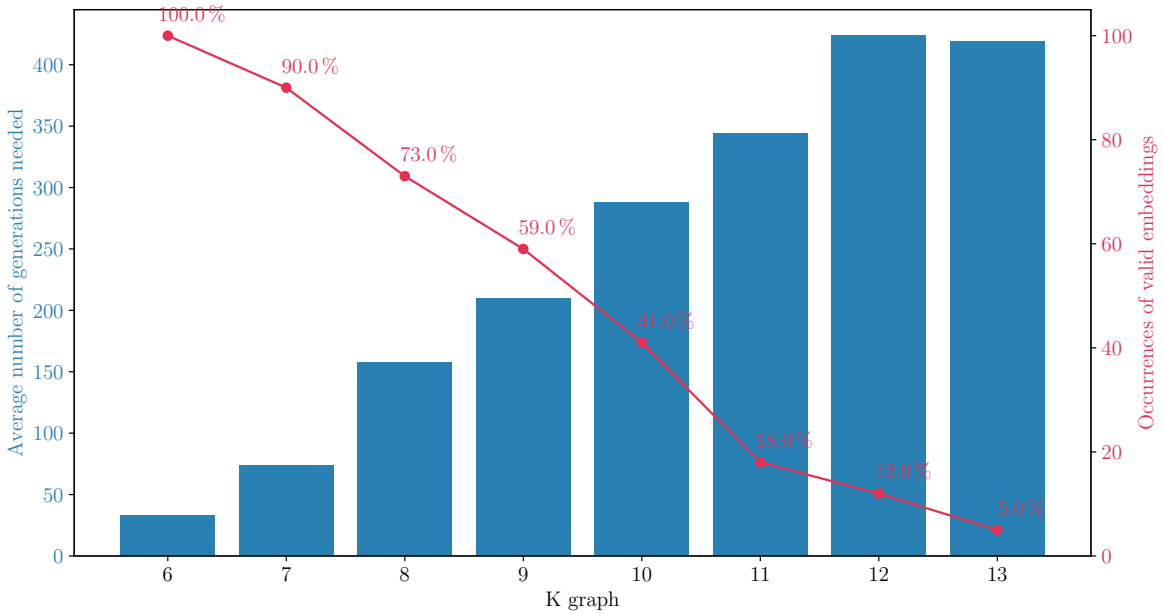


Fig. 7.11.: Embedding complete graphs on a 16x16 Chimera grid. 100 runs for every K_i, population size: 6, max mutation trials: 30, probability to choose mutation "extend to free neigbors": 0.24, max generations before failure: 600, remove redundancy probability: 0.01

7.3.3. Variation of evolutionary parameters

For the fixed graph K_8 and Chimera grid of size 5x5, we have also changed evolutionary parameters and ran the heuristic 300 times for every value. Figure 7.12 shows the results when altering the probability that the "extend to free neighbor" heuristic is used instead of the "extend to embedded neighbor" heuristic (see line 13 in Algorithm 6.6.1). Apparently, changing the probability in the range 11 % to 95 % does not have an overly strong effect on the occurrences of valid embeddings. Very low probabilities lead to a significant loss of quality, even down to 0 % when we only rely on extensions to embedded neighbors. In contrast, we manage to embed 56.67 % out of 300 runs if we only use the "extend to free neighbor" heuristic. A probability of 16 % leads to the highest success of 86.0 %, while we drop only slightly for a probability of 21 % which has a smaller number of average generations needed to find a valid embedding. Note that we performed 10 runs to quickly estimate a good probability to 24 % and used it in the previous section embedding K-graphs.

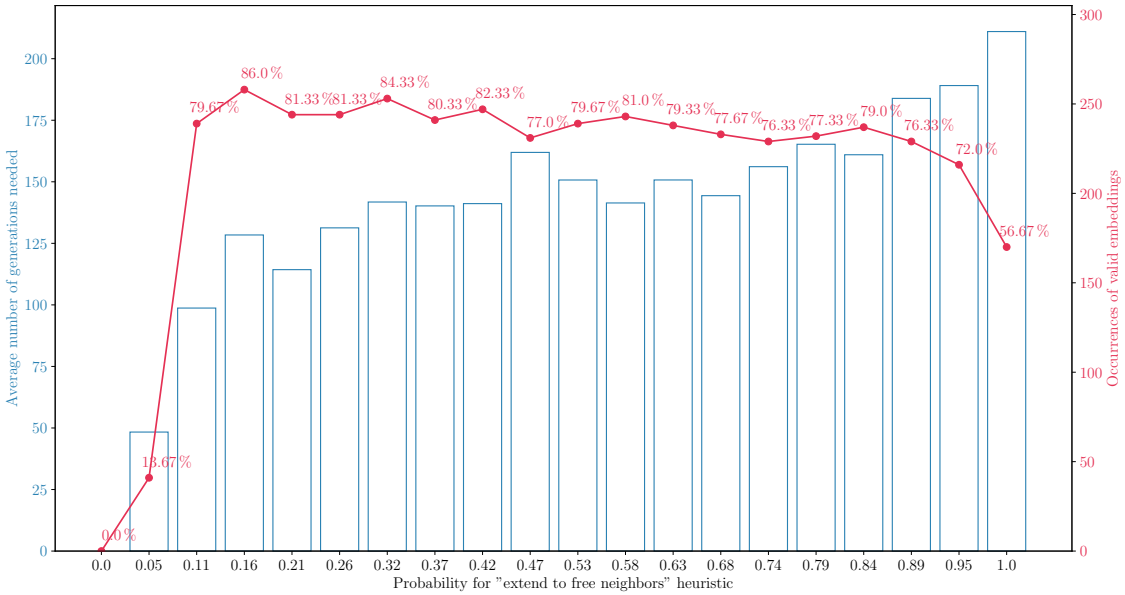


Fig. 7.12.: Embedding K_8 on a 5x5 Chimera grid. 300 runs for every probability, population size: 6, max mutation trials: 30, max generations before failure: 600, remove redundancy probability: 0.01

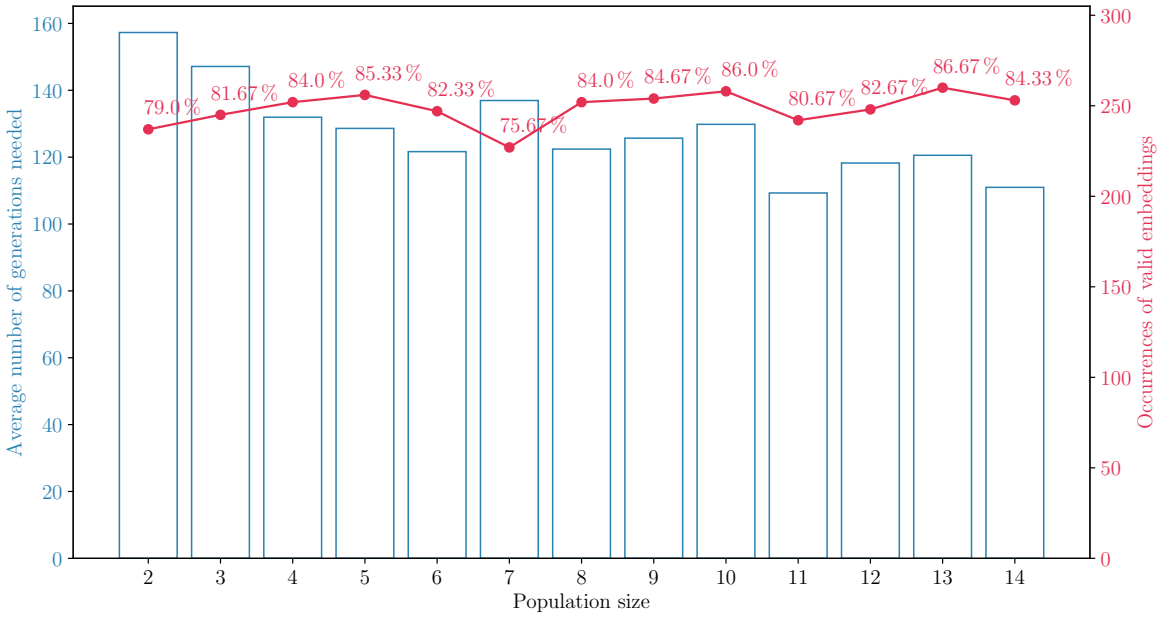


Fig. 7.13.: Embedding K_8 on a 5x5 Chimera grid. 300 runs for every population size, max mutation trials: 30, probability to choose mutation "extend to free neigbors": 0.24, max generations before failure: 600, remove redundancy probability: 0.01

In another experiment, we changed the population size and kept every other parameter fixed (e. g. we used a probability of 24 % to select the "embed to free neighbors" mutation). The execution time increased linearly from 2203 s for a population size of 3, to 7188 s for 7 and 10518 s for 14 children in a population. In a quick experiment, we guessed that 6 was a good population size and used it in the other experiments. Figure 7.13 shows that 5 children would have resulted in slightly better success rates – at least for K_8 graphs on a 5x5 Chimera grid. Overall, the effect of the population size can be considered negligible.

7.3.4. Conclusion of the results for Heuristic 2

The heuristic presented in section 6.6 was deliberately implemented in Python to allow for quick prototyping. This was paid for with horrendously high execution times up to 6 hours for 300 runs on a single parameter value despite using Python's multiprocessing module and a 100 % workload on the CPU. For small, fully connected graphs up to K_7, we obtained a high success rate that dropped quickly afterwards. This shows that our algorithm is not state-of-the-art and can therefore not compete with the minorminer heuristic in [6]. Despite the bad results, the algorithm for the heuristic and the Python implementation might still be valuable for readers as we've developed an easy-to-understand, object-oriented architecture other implementations might benefit from. Furthermore, the algorithm can be improved, e. g. ported to a more "low-level" programming language, so that more insights can be gathered in less time, enabling to identify the bottlenecks of the current implementation.

8. Concluding remarks

The minor embedding problem is a crucial part of the process of solving a QUBO problem on a quantum annealer. We presented two different randomized heuristics tackling the problem. The first one iteratively solves the SVEP subproblem, mutates the temporary solution between these iterations and tries to reduce overlapping super vertices at the end, while the second optimizes the number of embedded edges, therefore not allowing overlaps at any point in time. Additionally, the latter is a population-based algorithm and chooses the best offspring in every iteration.

Unfortunately, as the results imply, the algorithms are by far not state-of-the-art and leave room for improvement. That, however, was to be expected. The heuristics involved might still be of use and the theoretical results, regarding the SVEP in particular, might be of further interest.

Bibliography

[1] Lobe, E./ Lutz, A. *Minor Embedding in Broken Chimera and Pegasus Graphs is NP-complete.* 10/15/2021. URL: https://elib.dlr.de/146749/1/2110.08325.pdf (visited on 03/25/2022).

[2] Lucas, A. "Ising formulations of many NP problems." In: *Frontiers in Physics* 2 (2014). DOI: 10.3389/fphy.2014.00005. URL: http://dx.doi.org/10.3389/fphy.2014.00005.

[3] Lodewijks, B. *Mapping NP-hard and NP-complete optimisation problems to Quadratic Unconstrained Binary Optimisation problems.* 2020. arXiv: 1911.08043 [cs.DS].

[4] Vyskočil, T./ Djidjev, H. "Embedding Equality Constraints of Optimization Problems into a Quantum Annealer." In: *Algorithms* 12.4 (2019). DOI: 10.3390/a12040077. URL: https://www.mdpi.com/1999-4893/12/4/77.

[5] Vyskočil, T./ Pakin, S./ Djidjev, H. N. "Embedding Inequality Constraints for Quantum Annealing Optimization." In: *Quantum Technology and Optimization Problems.* Ed. by Feld, S./ Linnhoff-Popien, C. Cham: Springer International Publishing, 2019, pp. 11–22.

[6] Cai, J./ Macready, W. G./ Roy, A. *A practical heuristic for finding graph minors.* 06/12/2014. eprint: 1406.2741 (quant-ph). URL: https://arxiv.org/pdf/1406.2741v1.pdf (visited on 02/15/2022).

[7] Farhi, E. et al. *Quantum Computation by Adiabatic Evolution.* 2000. arXiv: quant-ph/0001106 [quant-ph]. URL: https://www.physast.uga.edu/~mgeller/quant-ph%200001106.pdf (visited on 03/02/2022).

[8] D-Wave Systems Inc. *Getting Started with D-Wave Solvers: User Manual.* 10/04/2021. URL: https://docs.dwavesys.com/docs/latest/_downloads/ced4bf7d70d137767f74f8bf99173832/09-1076D-C_GettingStarted.pdf (visited on 01/16/2021).

[9] Choi, V. *Minor-Embedding in Adiabatic Quantum Computation: I. The Parameter Setting Problem.* 04/30/2008. URL: https://arxiv.org/pdf/0804.4884.pdf (visited on 02/17/2022).

[10] Cai, W. *Handout 12. Ising Model: ME346A Introduction to Statistical Mechanics*. Stanford University, 02/25/2011. URL: http://micro.stanford.edu/~caiwei/me334/Chap12_Ising_Model_v04.pdf (visited on 03/02/2022).

[11] Mbeng, G. B./ Russomanno, A./ Santoro, G. E. *The quantum Ising chain for beginners*. 09/22/2020. URL: https://arxiv.org/pdf/2009.09208.pdf (visited on 03/02/2022).

[12] Kadowaki, T./ Nishimori, H. "Quantum annealing in the transverse Ising model." In: *Physical Review E* 58.5 (1998), pp. 5355–5363. DOI: 10.1103/physreve.58.5355. URL: http://dx.doi.org/10.1103/PhysRevE.58.5355 (visited on 03/16/2022).

[13] Bian, Z. et al. "Discrete optimization using quantum annealing on sparse Ising models." In: *Frontiers in Physics* 2 (2014). DOI: 10.3389/fphy.2014.00056. URL: https://www.frontiersin.org/articles/10.3389/fphy.2014.00056/full (visited on 03/01/2022).

[14] Sugie, Y. et al. "Minor-embedding heuristics for large-scale annealing processors with sparse hardware graphs of up to 102,400 nodes." In: *Soft Computing* 25.3 (2021), pp. 1731–1749. DOI: 10.1007/s00500-020-05502-6. (Visited on 02/17/2022).

[15] Zyga, L. *D-Wave sells first commercial quantum computer*. PhysOrg, 06/01/2011. URL: https://phys.org/news/2011-06-d-wave-commercial-quantum.html (visited on 03/16/2022).

[16] Glover, F./ Kochenberger, G./ Du, Y. *A Tutorial on Formulating and Using QUBO Models*. 2019. arXiv: 1811.11538 [cs.DS].

[17] Neven, H. et al. "NIPS 2009 Demonstration: Binary Classification using Hardware Implementation of Quantum Annealing." In: (12/07/2009). URL: https://static.googleusercontent.com/media/www.google.com/en//googleblogs/pdfs/nips_demoreport_120709_research.pdf (visited on 03/25/2022).

[18] Boothby, K. et al. *Next-Generation Topology of D-Wave Quantum Processors*. 2020. DOI: 10.48550/ARXIV.2003.00133. URL: https://arxiv.org/abs/2003.00133 (visited on 03/25/2022).

[19] Dattani, N./ Chancellor, N. *Embedding quadratization gadgets on Chimera and Pegasus graphs*. 2019. arXiv: 1901.07676 [quant-ph].

[20] Dattani, N./ Szalay, S./ Chancellor, N. *Pegasus: The second connectivity graph for large-scale quantum annealing hardware*. 2019. arXiv: 1901.07636 [quant-ph].

[21] Okuyama, T./ Hayashi, M./ Yamaoka, M. “An Ising Computer Based on Simulated Quantum Annealing by Path Integral Monte Carlo Method.” In: *2017 IEEE International Conference on Rebooting Computing (ICRC)*. IEEE, 11/8/2017 - 11/9/2017, pp. 1–6. DOI: 10.1109/ICRC.2017.8123652. (Visited on 02/16/2022).

[22] Achterberg, T. et al. “Presolve Reductions in Mixed Integer Programming.” In: *INFORMS Journal on Computing* 32 (11/2019). DOI: 10.1287/ijoc.2018.0857.

[23] Boros, E. et al. “A max-flow approach to improved lower bounds for quadratic unconstrained binary optimization (QUBO).” In: *Discrete Optimization* 5.2 (2008). In Memory of George B. Dantzig, pp. 501–529. DOI: https://doi.org/10.1016/j.disopt.2007.02.001. URL: https://www.sciencedirect.com/science/article/pii/S1572528607000400.

[24] Boros, E./ Hammer, P./ Tavares, G. “Preprocessing of unconstrained quadratic binary optimization.” In: *RUTCOR Research Report* (01/2006).

[25] Lewis, M./ Glover, F. “Quadratic unconstrained binary optimization problem preprocessing: Theory and empirical analysis.” In: *Networks* 70.2 (2017), pp. 79–97. DOI: 10.1002/net.21751. URL: https://arxiv.org/ftp/arxiv/papers/1705/1705.09844.pdf (visited on 03/02/2022).

[26] Yonaga, K./ Miyama, M. J./ Ohzeki, M. *Solving Inequality-Constrained Binary Optimization Problems on Quantum Annealer*. 2020. arXiv: 2012.06119 [quant-ph].

[27] Huang, T. et al. “QROSS: QUBO Relaxation Parameter optimisation via Learning Solver Surrogates.” In: *2021 IEEE 41st International Conference on Distributed Computing Systems Workshops (ICDCSW)*. IEEE. 2021, pp. 35–40.

[28] Xu, H./ Kumar, T./ Koenig, S. “A new solver for the minimum weighted vertex cover problem.” In: *International Conference on AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*. Springer. 2016, pp. 392–405.

[29] Tamura, K. et al. “Performance Comparison of Typical Binary-Integer Encodings in an Ising Machine.” In: *IEEE Access* 9 (2021), pp. 81032–81039. DOI: 10.1109/ACCESS.2021.3081685.

[30] Calude, C. S./ Dinneen, M. J./ Hua, R. "QUBO formulations for the graph isomorphism problem and related problems." In: *Theoretical Computer Science* 701 (2017), pp. 54–69.

[31] Dorband, J. E. *Extending the D-Wave with support for Higher Precision Coefficients*. 2018. arXiv: 1807.05244 [cs.ET].

[32] Dorband, J. E. *Improving the Accuracy of an Adiabatic Quantum Computer*. 2017. arXiv: 1705.01942 [quant-ph].

[33] Williamson, D. P./ Shmoys, D. B. *The Design of Approximation Algorithms*. 1st. USA: Cambridge University Press, 2011.

[34] Vazirani, V. V. *Approximation algorithms*. Vol. 1. Springer, 2001, pp. 15–26.

[35] Karp, R. M. "Reducibility among Combinatorial Problems." In: *Complexity of Computer Computations: Proceedings of a symposium on the Complexity of Computer Computations, held March 20–22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York, and sponsored by the Office of Naval Research, Mathematics Program, IBM World Trade Corporation, and the IBM Research Mathematical Sciences Department*. Ed. by Miller, R. E./ Thatcher, J. W./ Bohlinger, J. D. Boston, MA: Springer US, 1972, pp. 85–103. DOI: 10.1007/978-1-4684-2001-2_9. URL: https://doi.org/10.1007/978-1-4684-2001-2_9.

[36] Zhang, Z./ Gao, X./ Wu, W. "Algorithms for connected set cover problem and fault-tolerant connected set cover problem." In: *Theoretical Computer Science* 410 (03/2009), pp. 812–817. DOI: 10.1016/j.tcs.2008.11.005.

[37] Ausiello, G./ Paschos, V. "Approximability preserving reduction." In: (2005).

[38] Zhang, W. et al. "Complexity and approximation of the connected set-cover problem." In: *Journal of Global Optimization* 53.3 (2012), pp. 563–572.

[39] Waissi, G. R. *Network Flows: Theory, Algorithms, and Applications*. 1994.

[40] Orlin, J. "A polynomial time primal network simplex algorithm for minimum cost flows." In: *Math. Prog.* 78 (01/1996), pp. 109–129. DOI: 10.1007/BF02614365.

[41] Joosten, S. J. "Relaxations of the 3-partition problem." MA thesis. University of Twente, 2011, pp. 7–11.

[42] Schaefer, T. J. "The Complexity of Satisfiability Problems." In: *Proceedings of the Tenth Annual ACM Symposium on Theory of Computing*. STOC '78. San Diego, California, USA: Association for Computing Machinery, 1978, 216–226. DOI: 10.1145/800133.804350. URL: https://doi.org/10.1145/800133.804350.

[43] Bura, V. "Positive 1-in-3-SAT admits a non-trivial Kernel." In: *arXiv preprint arXiv:1808.02821* (2018), pp. 8–10.

[44] Diestel, R. *Graph Theory (Graduate Texts in Mathematics)*. Springer, 2005, p. 15.

[45] Hopcroft, J./ Tarjan, R. "Algorithm 447: Efficient algorithms for graph manipulation." In: *Communications of the ACM* 16.6 (1973), pp. 372–378. URL: https://irem.univ-reunion.fr/IMG/pdf/hopcroft73.pdf (visited on 03/14/2022).

[46] Rosenberg, I. G. "Reduction of bivalent maximization to the quadratic case." In: (1975).

A. QUBO and Ising Equivalence

We show that the Ising Model and QUBO formulation are isomorphic. Consider the Hamiltonian of the Ising Model (see section 2.3 for an explanation of the parameters):

$$H(\sigma) = \sum_{(ij)\in E(G)} J_{ij}\sigma_i\sigma_j + \sum_{i\in V(G)} h_i\sigma_i \tag{A.44}$$

Using the **spin-binary bijection** $\sigma \to 2x - 1$ (1 is a vector whose components are all one), we are able to transform Equation A.44 into the equivalent QUBO problem as described in reverse in [9]:

$$\begin{aligned}
H(\sigma) &= \sum_{(ij)\in E(G)} J_{ij}(2x_i - 1)(2x_j - 1) + \sum_{i\in V(G)} h_i(2x_i - 1) \\
&= \sum_{(ij)\in E(G)} J_{ij}(4x_ix_j - 2x_i - 2x_j + 1) + \sum_{i\in V(G)} (2h_ix_i - h_i) \\
&= \sum_{(ij)\in E(G)} 4J_{ij}x_ix_j - \underbrace{\sum_{(ij)\in E(G)} (2J_{ij}x_i + 2J_{ij}x_j) + \sum_{i\in V(G)} 2h_ix_i}_{\sum_{i\in V(G)} x_i\left(2h_i - \sum_{\mathrm{nbr}(i)}(2J_{ij}+2J_{ji})\right)} \\
&\quad + \underbrace{\sum_{(ij)\in E(G)} J_{ij} - \sum_{i\in V(G)} h_i}_{c} \\
&= \boxed{\sum_{(ij)\in E(G)} J'_{ij}x_ix_j + \sum_{i\in V(G)} h'_ix_i + c}
\end{aligned}$$

where we define in the last step:

$$\begin{aligned}
J'_{ij} &= 4J_{ij} \\
h'_i &= 2h_i - \sum_{\mathrm{nbr}(i)} (2J_{ij} + 2J_{ji})
\end{aligned}$$

This way, we obtained the QUBO representation out of the Ising model, hence they are equivalent. Note that we can drop the constant c as it does not change the position of the optimum x^*.

B. Rosenberg polynomials

In [46], Rosenberg developed a reformulation for constraints of the form

$$y = ab, \qquad a, b, y \in \mathbb{B} \tag{B.45}$$

The usual reformulation would create higher order terms through squaring. Instead, Rosenberg proposed to introduce the following terms in a QUBO in order to enforce y to be the AND of a and b:

$$\lambda \cdot (3y - 2x_1y - 2x_2y + 1x_1x_2) \tag{B.46}$$

thereby penalizing variable assignments that do not satisfy constraint B.45. Table B.1 proves that.

x_1	x_2	y	contribution
0	0	0	0
0	0	1	3λ
0	1	0	0
0	1	1	1λ
1	0	0	0
1	0	1	1λ
1	1	0	1λ
1	1	1	0

Table B.1.: Invalid assignments are penalized in Rosenberg's reformulation.

Clearly, every valid assignment evaluates to 0 while invalid assignments evaluate to a positive penalty (given $\lambda > 0$). Similarly, to the reformulation of B.46, table B.2 shows reformulation for similar quadratic relations.
The reformulations follow trivially by substituting $\overline{x_i}$ with $(1 - x_i)$.

Constraint	**Reformulation**
$y = x_1 x_2$	$\lambda \cdot (3y - 2x_1 y - 2x_2 y + 1x_1 x_2)$
$y = x_1 \overline{x_2}$	$\lambda \cdot (y - 2x_1 y + 2x_2 y - 1x_1 x_2 + x_1)$
$y = \overline{x_1} x_2$	$\lambda \cdot (y + 2x_1 y - 2x_2 y - 1x_1 x_2 + x_2)$
$y = \overline{x_1}\,\overline{x_2}$	$\lambda \cdot (-y + 2x_1 y + 2x_2 y + 1x_1 x_2 - x_1 - x_2 + 1)$

Table B.2.: Rosenberg reformulation for negated variables.

C. Additional results of heuristic 2

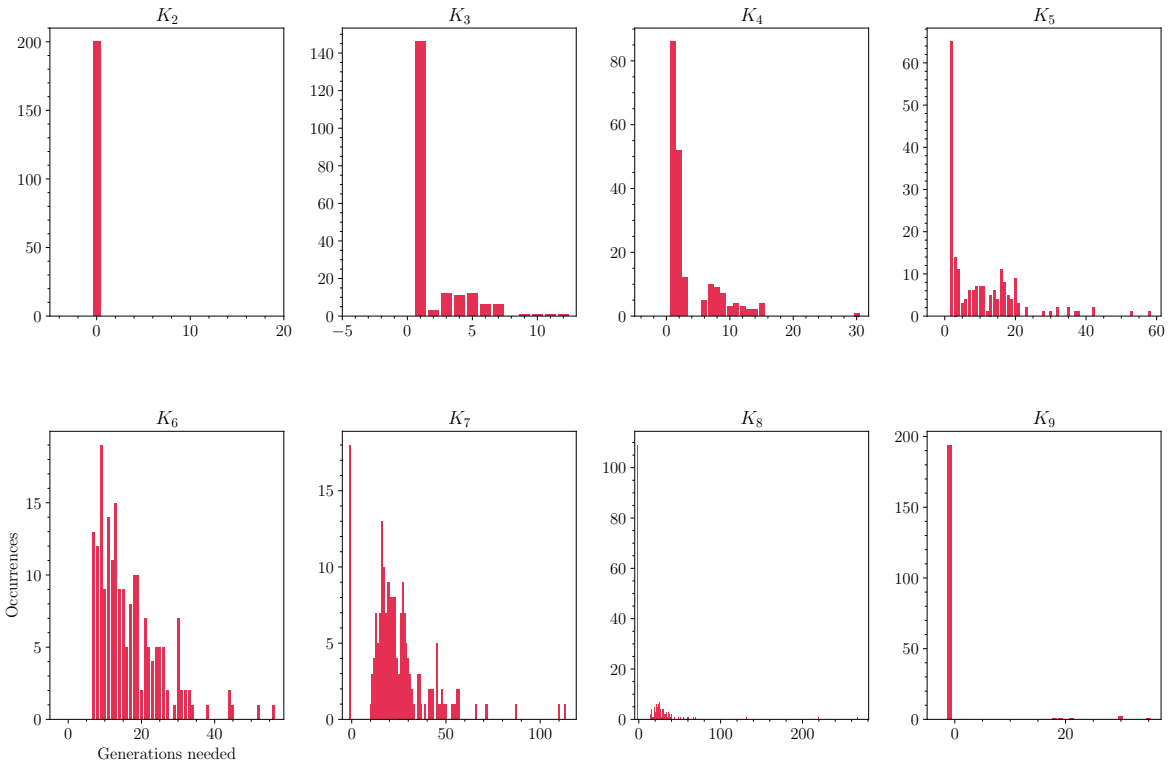


Fig. C.1.: Embedding complete graphs on a 2x2 Chimera grid. A value of −1 represents an invalid/failed embedding. 200 runs for every K_i, population size: 6, max mutation trials: 30, probability to choose mutation "extend to free neigbors": 0.24, max generations before failure: 600, remove redundancy probability: 0.01

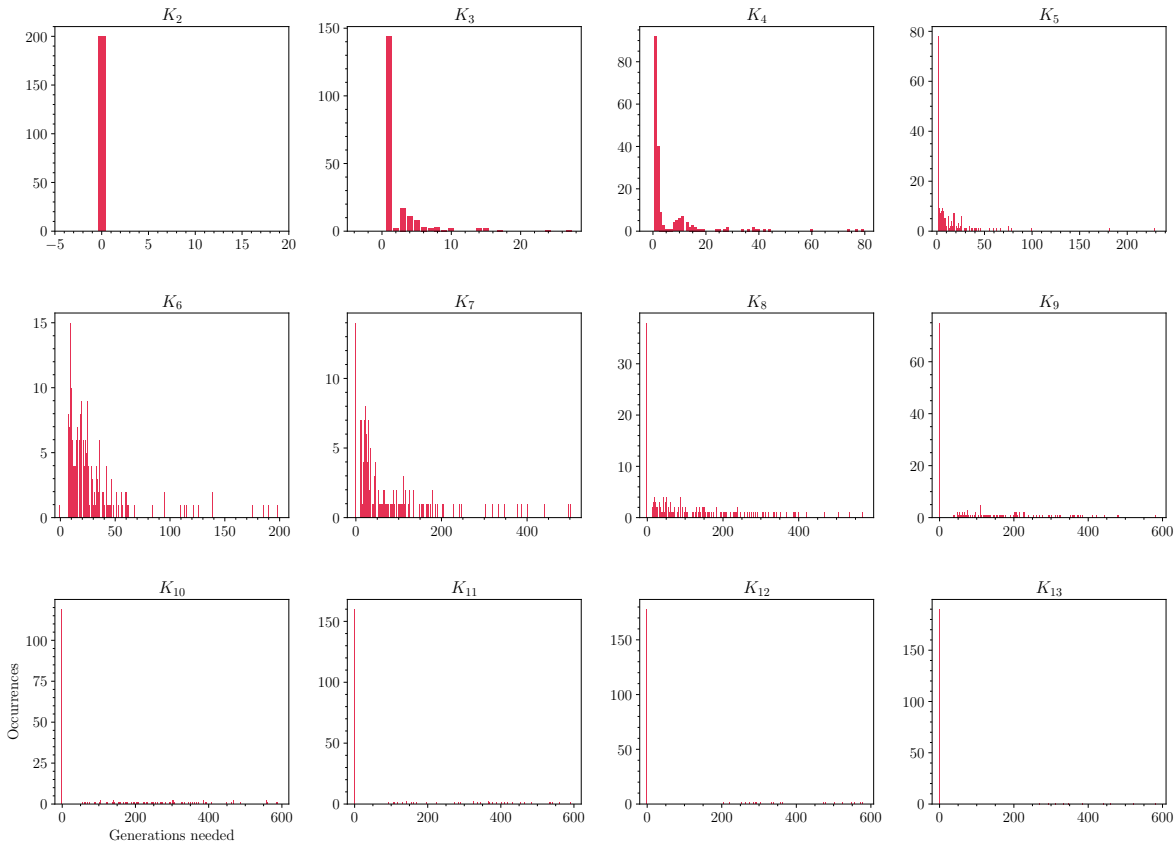


Fig. C.2.: Embedding complete graphs on a 5x5 Chimera grid. A value of −1 represents an invalid/failed embedding. 200 runs for every K_i, population size: 6, max mutation trials: 30, probability to choose mutation "extend to free neigbors": 0.24, max generations before failure: 600, remove redundancy probability: 0.01

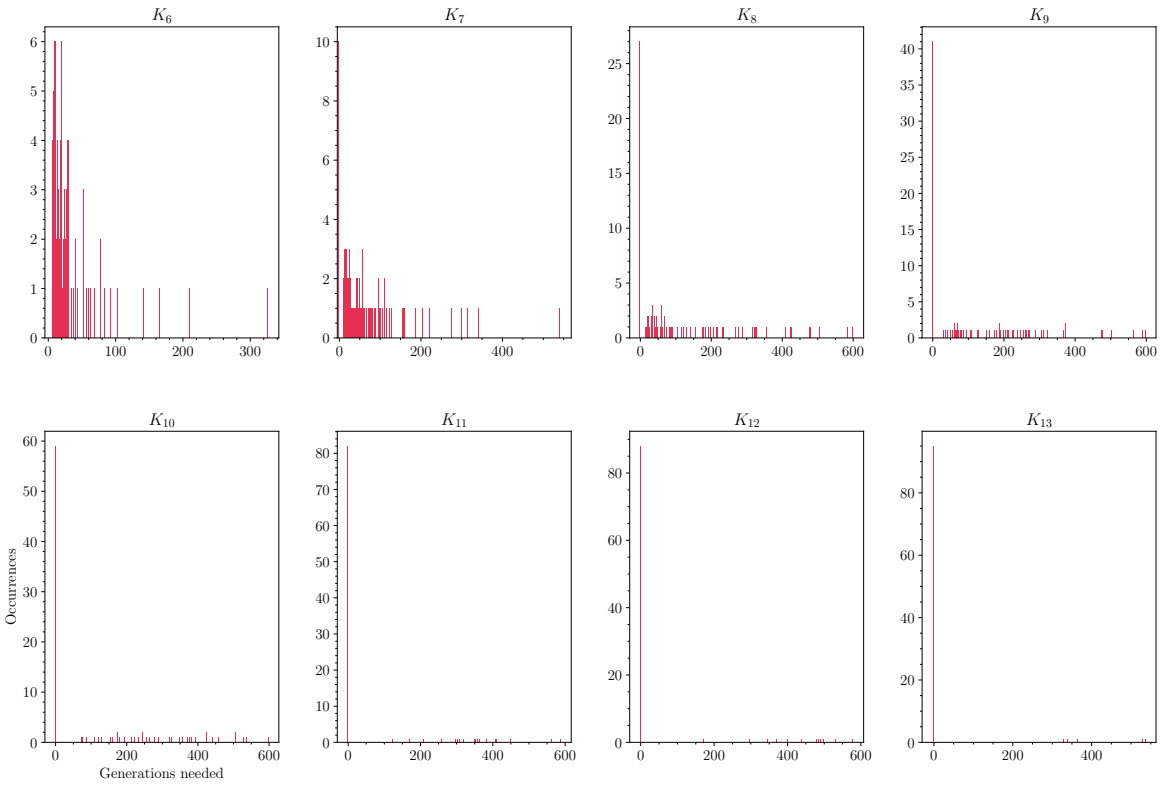


Fig. C.3.: Embedding complete graphs on a 16x16 Chimera grid. A value of −1 represents an invalid/failed embedding. 100 runs for every K_i, population size: 6, max mutation trials: 30, probability to choose mutation "extend to free neigbors": 0.24, max generations before failure: 600, remove redundancy probability: 0.01

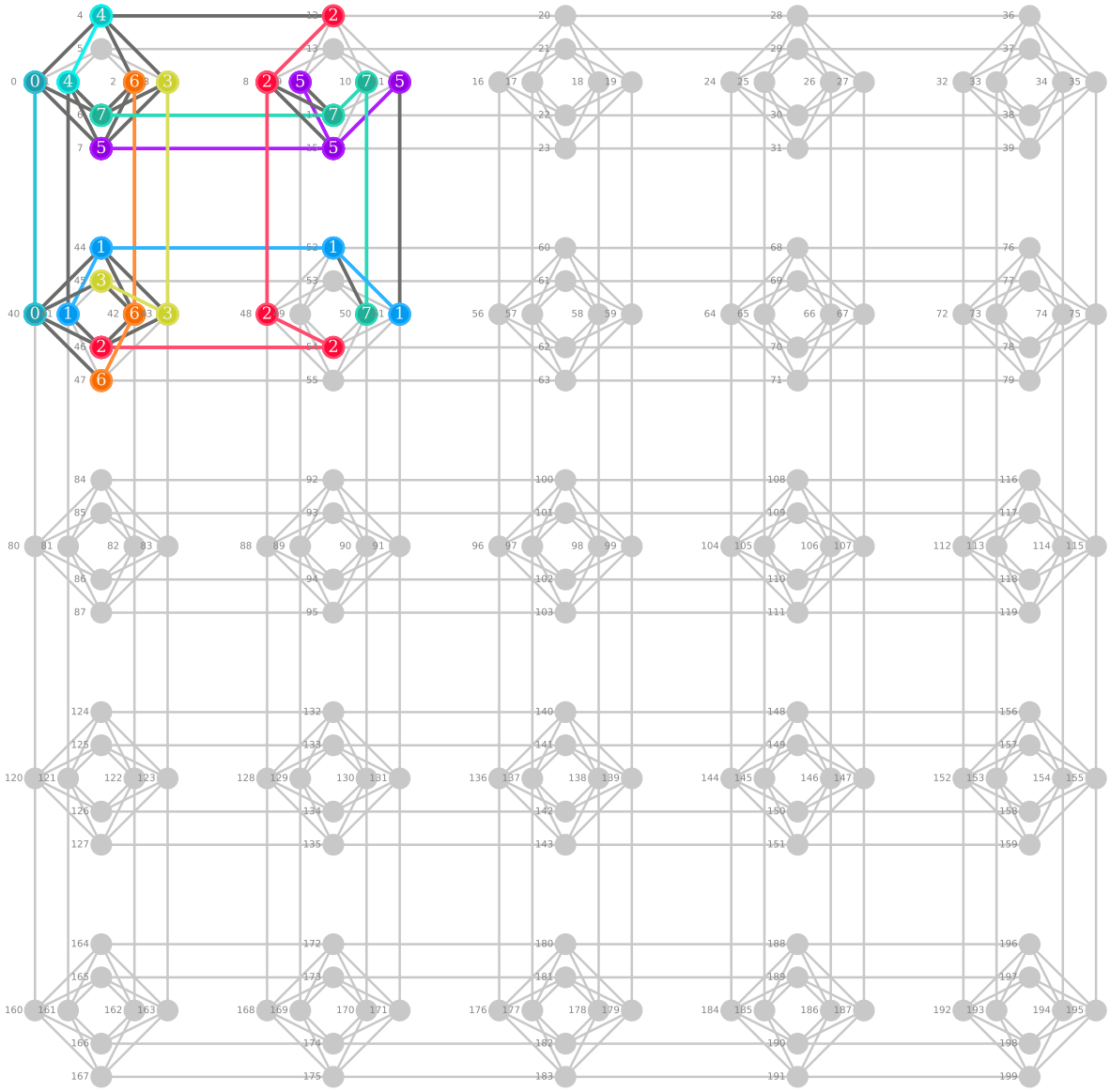


Fig. C.4.: Good Embedding of K8 on a 5x5 Chimera grid (17 Generations needed)

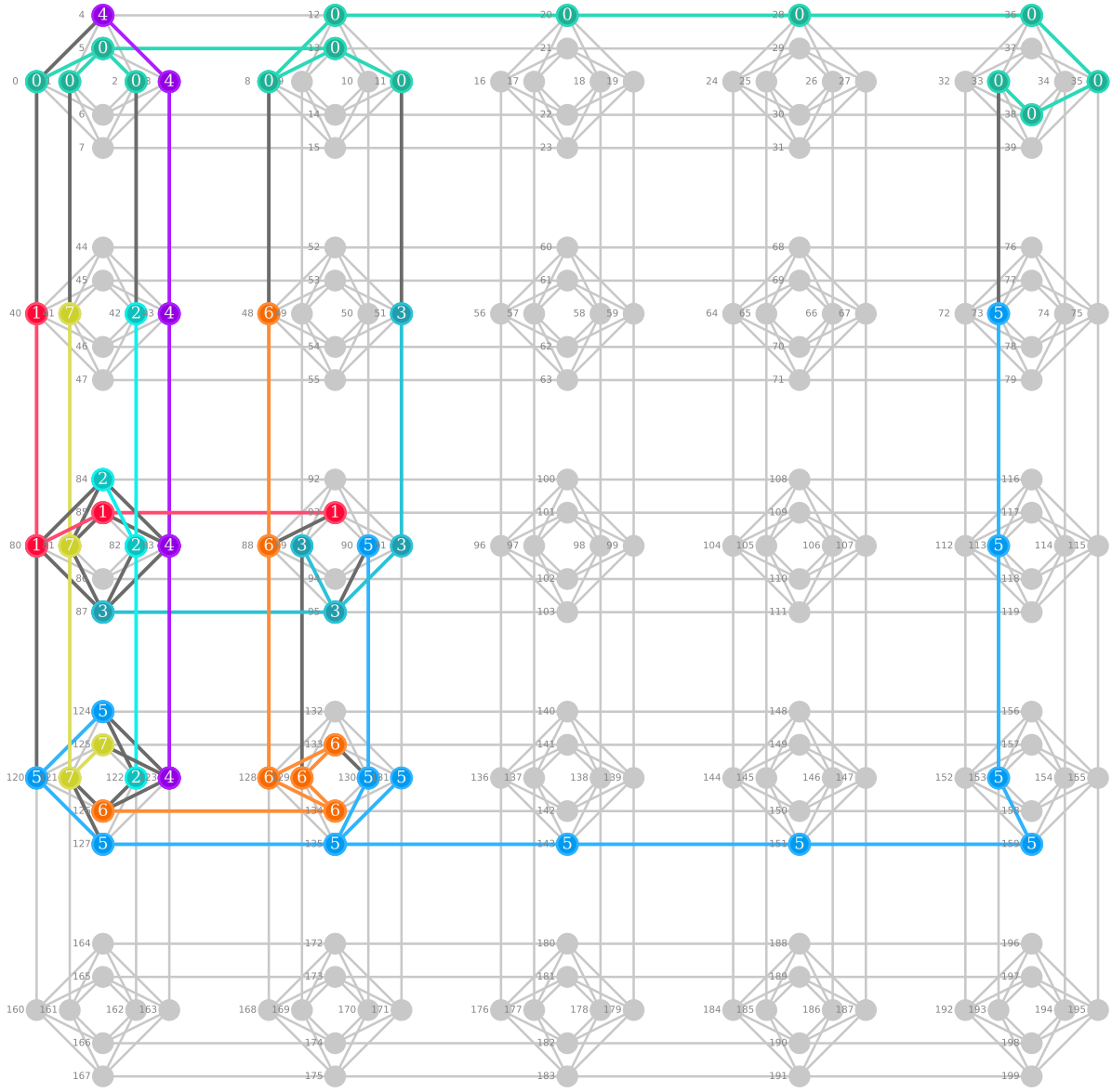


Fig. C.5.: Not ideal embedding of K8 on a 5x5 Chimera grid (171 generations needed)