ABSTRACT

GOODRICH, TIMOTHY DAVID. Practical Graph Algorithms with Applications in Near-Term Quantum Computing. (Under the direction of Dr. Matthias Stallmann.)

The advent of production hardware in quantum computing has introduced a host of engineering issues preventing adoption into the broader high-performance computing machinery. Several of these issues are fundamentally graph theory and optimization problems which require accessible, tuned software solutions. Theoretical solutions often exist, but are typically hampered by non-constructive proofs and hidden constants in run time complexities. Additionally, these solutions rarely utilize the structure inherent to the quantum computing problems. Addressing these issues at a practical level, we develop open source software solutions with (novel or existing) algorithms tuned on relevant data and evaluated in real-world quantum software pipelines.

In the context of quantum annealers, we study the minor embedding problem and develop a solution employing a virtual hardware layer to introduce modularity and simplify the hardware interface. With a bipartite virtual hardware we show competitive performance against all first-generation embedding heuristics. Enabling faster solutions to this embedding method, we experimentally study methods for computing distance-to-bipartite and evaluate these algorithms on quantum annealer program data.

In a quantum-agnostic contribution, we study an approximation algorithm for editing a graph to a specified degeneracy using the local ratio technique; this editing algorithm has broader applications in a new framework for developing approximation algorithms to structured instances in a variety of optimization problems. We use this technique to refine our approximation algorithm for graph bipartization.

Our final application centers on contraction sequence algorithms for tensor networks, a modern framework for scalable quantum simulations. These contraction sequences have direct ties to the treewidth graph invariant, and we provide the software tools needed to use state-of-the-art treewidth solvers as contraction sequences algorithms. With an additional study, we show how cheap reduction routines and choice of preprocessing heuristics lead to speedups on an expanded corpus of treewidth and tensor network instances.

© Copyright 2020 by Timothy David Goodrich

All Rights Reserved

Practical Graph Algorithms with Applications in Near-Term Quantum Computing

by
Timothy David Goodrich

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Computer Science

Raleigh, North Carolina

2020

APPROVED BY:

Dr. Steffen Heber	Dr. Alessandra Scafuro
Dr. Ranga Vatsavai	Dr. Matthias Stallmann Chair of Advisory Committee

DEDICATION

Dedicated to my parents, Susie and David Goodrich,
who instilled in me a love of learning,
and taught me how to succeed through adversity.

Mathematical proofs, like diamonds, are hard as well as clear,
and will be touched with nothing but strict reasoning.
– John Locke (*A Mathematical Journey* by Stanley Gudder)

BIOGRAPHY

Timothy D. Goodrich was born in Nashville, TN. After being homeschooled (Franklin, TN), Timothy enrolled at Valparaiso University (Valparaiso, IN) in 2010. During his fall 2013 semester, Timothy studied in the Budapest Semesters in Mathematics program (Budapest, Hungary). He received a Bachelor of Science with a double major in Mathematics and Computer Science and a minor in Humanities in May 2014. In August 2014, Timothy entered the doctoral program in the Department of Computer Science at NC State University. During this time, he was awarded a three-year National Defense Science & Engineering Graduate fellowship (NDSEG) and a supplemental NC Space Grant graduate fellowship. Timothy earned a Masters of Science under Dr. Blair D. Sullivan (2017) and a Doctor of Philosophy under Dr. Matthias Stallmann (2020).

ACKNOWLEDGEMENTS

First and foremost, I would like to thank Dr. Matthias Stallmann, without whom this work would be incomplete. His time and effort spent in reading, contemplating, and discussing our results has been invaluable, and I will fondly remember our many conversations in his office.

During my time at NC State, I was particularly fortunate to work with two researchers whose influence can be seen in nearly every chapter of this dissertation, Dr. Blair D. Sullivan (University of Utah) and Dr. Travis Humble (Oak Ridge National Laboratory). Dr. Sullivan and her lab provided a wonderful perspective on how structural graph theory and parameterized algorithms can yield custom solutions for computationally difficult problems. Dr. Humble and his lab in the Quantum Institute at ORNL graciously hosted me for several visits, where I learned about the rapidly maturing field of quantum computing.

Additional thanks are owed to my collaborators on individual publications, including Eric Horton (Chapter 4); Dr. Erik Demaine and his lab at MIT (Chapter 5); Allison Fisher, Andrew Wright, and Dr. Eugene Dumitrescu (Chapters 6 & 7).

For their influence on my day-to-day work, I would also like to thank my academic brothers Dr. Michael O'Brien and Dr. Andrew van der Poel, Dr. Kyle Kloster, Dr. Felix Reidl, Steve Reinhardt, and Rev. Kevin 'The Bird' Martin.

I would not have pursued a doctorate without the encouragement of Dr. Rick Gillman, Dr. Lara Pudwell, Dr. Michael Glass, Dr. Mindy Capaldi, Tricia Armstrong, Dr. Mark Bartusch, Dr. Tibor Jordán, Dr. Kurt Nichol, and Dr. Justin Myrick. Surprisingly for its size, this set is minimal; proof is left as an exercise to the reader.

Finally, my most heartfelt thanks goes to my wife, Katherine Goodrich, whose suffering and patience has made the journey much more bearable.

This work has been made possible by the generous funding provided by North Carolina State University, Dr. Blair D. Sullivan's research grants, the Department of Defense through the NDSEG fellowship, and the NC Space Grant consortium.

TABLE OF CONTENTS

LIST OF TABLES viii

LIST OF FIGURES x

Chapter 1 INTRODUCTION 1
- 1.1 Graph algorithm contributions 2
  - 1.1.1 Odd Cycle Transversal 2
  - 1.1.2 Bounded Degeneracy Editing 4
  - 1.1.3 Treewidth 5
- 1.2 Quantum computing contributions 7
  - 1.2.1 A generic quantum pipeline 7
  - 1.2.2 Minor embedding (Chapter 3) 8
  - 1.2.3 Improved bipartization subroutines (Chapter 4) 11
  - 1.2.4 Distance to bounded degeneracy (Chapter 5) 12
  - 1.2.5 Tensor network simulations (Chapter 6) 13
  - 1.2.6 Improved contraction sequence subroutines (Chapter 7) 15
- 1.3 Summary 16

Chapter 2 BACKGROUND 18
- 2.1 Graph Theory 18
  - 2.1.1 General Notation 18
  - 2.1.2 Structures, Orderings, and Editing 18
  - 2.1.3 Synthetic Graph Generators 20
- 2.2 Optimization and Algorithms 21
  - 2.2.1 Optimization Problems 21
  - 2.2.2 Approximation Algorithms 23
  - 2.2.3 Approximation Hardness 23
  - 2.2.4 Parameterized Algorithms and Complexity 24

Chapter 3 MINOR EMBEDDING ALGORITHMS FOR QUANTUM ANNEALING . 25
- 3.1 Introduction 25
- 3.2 Background 27
  - 3.2.1 Minor Embedding for Adiabatic Quantum Programming 28
  - 3.2.2 Related Work 30
- 3.3 Virtual Hardware Framework 30
  - 3.3.1 Biclique Virtual Hardware 32
  - 3.3.2 Biclique Embedding and Reduction Subroutines 33
  - 3.3.3 Emulation and Enhancement 36
  - 3.3.4 Summary 36
- 3.4 Utilizing Bipartite Problem Structure 37
  - 3.4.1 Odd Cycle Transversal 37
  - 3.4.2 OCT and the Chimera Graph 37
  - 3.4.3 Computing OCT and OCT-Embed 39

3.4.4 Approximating OCT and Fast-OCT-Embed . . . 40
3.4.5 Summary . . . 43
3.5 Experimental Results . . . 44
3.5.1 Experimental Results . . . 45
3.6 Conclusion . . . 47
3.7 Appendix: Computing OCT in Series-Parallel Graphs . . . 49

Chapter 4 EDITING TO BIPARTITE . . . 52
4.1 Introduction . . . 52
4.1.1 Related Work . . . 53
4.1.2 Our Contributions . . . 54
4.2 Background . . . 55
4.2.1 Graph bipartization in quantum annealing . . . 56
4.3 Algorithm Overview . . . 57
4.3.1 Reduction Routines . . . 57
4.3.2 Heuristics and Approximations . . . 58
4.3.3 Iterative Compression . . . 58
4.3.4 Modifications to Hüffner's Algorithm . . . 59
4.3.5 Alternative Solvers . . . 60
4.4 Data Benchmark and Code . . . 60
4.4.1 Previous Data . . . 60
4.4.2 Quantum-Inspired Data . . . 61
4.4.3 Synthetic Graph Generators . . . 62
4.4.4 Replicability . . . 63
4.5 Quantum-Specific Results . . . 63
4.5.1 Experimental Setup . . . 63
4.5.2 Preprocessing Effectiveness . . . 63
4.5.3 Use Case: Heuristic Solutions . . . 64
4.5.4 Use Case: Exact Solutions . . . 66
4.6 Generalized Results . . . 66
4.6.1 Reduction Effectiveness . . . 67
4.6.2 Heuristic Solutions . . . 67
4.6.3 Exact Solutions . . . 67
4.7 Conclusion . . . 68
4.8 Appendix: Implementation Details . . . 69
4.8.1 Data Ingestion and Sanitization . . . 69
4.8.2 Reduction Routines . . . 70
4.8.3 Heuristics . . . 70
4.8.4 Hüffner Improvements . . . 71
4.9 Appendix: Extended Results . . . 72
4.9.1 ILP Solver Comparison . . . 72

Chapter 5 EDITING TO BOUNDED DEGENERACY . . . 75
5.1 Introduction . . . 75
5.2 Degeneracy Editing is APX-hard . . . 76

5.2.1 Mapping a VC Instance to a r-DVD Instance . . . 77
5.2.2 Mapping a r-DVD Solution to a VC Solution . . . 79
5.2.3 Proving the strict reduction . . . 80
5.2.4 Extension to r-DED . . . 81
5.3 A Bicriteria Approximation Algorithm for Degeneracy Editing . . . 82
5.3.1 The local ratio technique and analysis overview . . . 82
5.3.2 Proof of approximation factor . . . 84
5.4 A Refined OCT Approximation Algorithm . . . 87

Chapter 6 BENCHMARKING TENSOR NETWORK CONTRACTION SEQUENCE ALGORITHMS . . . 88
6.1 Introduction . . . 88
6.2 Background . . . 90
6.3 ConSequences: An Accessible, Extendable Framework . . . 92
6.4 MERA Applications . . . 94
6.4.1 Initial Comparison on Pfeifer Benchmark . . . 95
6.4.2 Extended Benchmark on Large MERA Networks . . . 96
6.5 Applications with qTorch Simulator . . . 99
6.5.1 Computing Contraction Complexity . . . 100
6.5.2 Simulation Run Times . . . 102
6.6 Conclusion . . . 104

Chapter 7 A TUNED TREEWIDTH SOLVER . . . 106
7.1 Introduction . . . 106
7.2 Background and Data . . . 107
7.2.1 PID algorithm definitions . . . 107
7.2.2 Data . . . 108
7.3 PID Treewidth Algorithm and Improvements . . . 110
7.3.1 A description of Tamaki's PID algorithm . . . 110
7.3.2 Proposed optimizations . . . 114
7.4 Experimental Setup and Results . . . 115
7.4.1 Effectiveness of reduction routines . . . 115
7.4.2 Effectiveness of safe separator heuristic choice . . . 116
7.4.3 Effectiveness of block sieve node width lower bound . . . 117
7.5 Conclusion . . . 118
7.6 Appendix: PID Algorithm Pseudocode . . . 118

BIBLIOGRAPHY . . . 122

LIST OF TABLES

Table 3.1 Definition of density levels for the random input graph generators. 45

Table 4.1 A summary of preprocessing statistics on WH and quantum datasets. Ranges are given for the number of vertices and edge density in both the original and reduced graphs. The statistic $|\widehat{V_r}|$ reports the percentage of vertices removed on average. Likewise, normalized means are reported for edge removals E_r, fixed-OCT vertices V_o, and fixed-bipartite vertices V_b; dashes denote zero changes. The percentage of graphs solved completely by preprocessing routines is also reported. 64

Table 4.2 Observed approximation factors for anytime algorithms: the heuristic ensemble (HE), iterative compression (IC), and integer linear programming (ILP). For each dataset, the worst-case approximation ratio over its instances is reported. Approximation ratios are with respect to OCT on the reduced graph, computed with ILP. A checkmark denotes that exact solutions are found on all instances, if a dataset has no checkmark then the best approximation algorithm is bolded. 65

Table 4.3 Run times (in seconds) of exact solvers on a representative sample of Beasley and GKA data with a 10 minute timeout. Algorithm-data pairings that did not finish within the timeout are denoted with a dash, and the best run time on a dataset is bolded. 65

Table 4.4 A summary of preprocessing statistics over all datasets. Ranges are given for the number of vertices and edge density in both the original and reduced graphs. The normalized statistic $|\widehat{V_r}|$ reports the average percentage of vertices removed. Likewise, normalized means are reported for edge removals E_r, fixed-OCT vertices V_o, and fixed-bipartite vertices V_b; dashes denote zero changes. The percent of graphs per dataset completely solved by preprocessing routines is also noted. Results are reported over 15 seeds per random graph generator, per original dataset. 73

Table 4.5 Observed approximation factors for anytime algorithms and heuristics at various timeouts. For each dataset, the worst-case approximation ratio over its instances is reported. Approximation ratios are with respect to OCT on the reduced graph. A checkmark denotes that exact solutions are found on all instances, if a dataset has no checkmark then the best approximation algorithm is bolded. If OCT could not be found within 10 minutes then the instance is not included; the percent of included data points is denoted in the *represented* column. 74

Table 6.1 Run times for each contraction sequence algorithm when executed on tensor network datasets from Pfeifer et al. [Pfe14b]. For each tensor network, the number of tensors ($|V|$), edges ($|E|$), and optimal contraction complexity (cc) are reported. 95

Table 6.2 Summary of extended MERA benchmark data. (Center) For each lattice type (1D binary or 2D 4-ary) and number of operators, as the number of levels ℓ varies, we report the number of vertices $|V|$ in the resulting networks, the total number of networks $|\mathscr{M}|$, and the number of unique networks up to isomorphism $|\widehat{\mathscr{M}}|$. (Left, Right) These detailed tables each expand a row from the summary table, specifying the number of networks produced (total $|\mathscr{S}|$ and up to isomorphism $|\widehat{\mathscr{S}}|$), for each pair of values for the number of vertices $|V|$ and number of edges $|E|$. Note that sum of the $|\mathscr{S}|$'s in a detailed table sums to the corresponding $|\mathscr{M}|$ in the summary table, and likewise for $|\widehat{\mathscr{S}}|$ and $|\widehat{\mathscr{M}}|$. 97
Table 6.3 Exact contraction complexities found using freetdi and meiji-e on QAOA circuits for computing MaxCut on r-regular graphs. 25 random regular graphs are generated at each $r, |V|$ level using NetworkX, and algorithms were timed out at 15 minutes. Timed out values were dropped from the data, resulting in less than 25 Samples for some parameter values. 100

Table 7.1 Reported quantiles on the percent of vertices removed per instance, clustered by instance corpus. Notably, nothing is removed from at least 25% of both treewidth corpora, but the majority of both vertices and edges are removed from at least one instance in each corpus. Contrastingly, the tensor networks are much more consistent in expected reduction. 116
Table 7.2 Reported quantiles on the percent of edges removed per instance, clustered by instance corpus. Similar to Table 7.1, these reduction routines vary in effectiveness on treewidth instances, but are predictably effective on tensor networks. 116
Table 7.3 Ratio of runtime for the original instance vs. the reduced instance; larger values are better. Similar to the effectiveness of the reduction routines, we find that the treewidth instances vary wildly, whereas the tensor networks are more predictable and positively impacted. 116
Table 7.4 Ratio of runtime for the original execution vs. executing with only the specified safe separator heuristic; larger values are better. Here we find a familiar split: treewidth results vary wildly and tensor networks are less impacted. 117
Table 7.5 The ratio of user runtime vs. real runtime; lower is more serial. 118

LIST OF FIGURES

Figure 1.1 (Left) The Petersen graph [Wei00]; (Right) Removing a minimum OCT set (red, dashed) from the Petersen graph yields a bipartite subgraph (blue and gray). 3

Figure 1.2 An optimal degeneracy ordering on the Peterson graph. Without edits, the graph has degeneracy of at least three, since degeneracy is at least the minimum degree. The degeneracy is also no more than three since each vertex has at most three neighbors to its right side in this *degeneracy ordering*. If vertex 1 or the dashed edge is deleted from the graph then the remaining subgraph has degeneracy two (witness with the same ordering). 4

Figure 1.3 A tree decomposition of the Petersen graph with minimum width of four. 6

Figure 1.4 A generic algorithmic workflow for running a quantum application. The original application must be encoded into a quantum program, such as a QUBO or a circuit using quantum gates. This program is typically preprocessed before execution, including steps such as reduction, pre-optimization, and compilation for the hardware architecture. After this program is executed on the quantum hardware then a postprocessing step is required to extract the solution and map it back to the original application. 7

Figure 1.5 Our minor embedding work concentrates on the pre-execution steps. We assume input is provided as a QUBO program and a hardware graph describing the quantum annealer topology. These two graphs are optionally fed into a litmus testing module to evaluate whether the program's graph properties (such as OCT and treewidth) are compatible with the hardware. Once an OCT set has been computed on the program then the QUBO instance is embedded into a virtual hardware layer, followed by footprint minimization for reducing the number of physical qubits used by this embedding. Once these steps are completed, the QUBO instance has been successfully embedded into the annealer graph and execution may begin. . 9

Figure 1.6 This chapter examines how the NP-hard OCT decomposition subroutine should be solved in a variety of real-world use cases. We assume the subroutine is given a graph instance and a stopping criterion (time- or solution quality-based). This graph is initially reduced using routines from the OCT and VC literature. A heuristic ensemble is then run to construct upper bounds on the OCT solution (which may suffice for the stopping criterion). If a smaller solution is required then an exact solver is executed, based on the structure of the graph. The subroutine is completed by outputting an OCT solution certificate. 10

Figure 1.7 A degeneracy editing algorithm fits into the quantum workflow when used in conjunction with an OCT approximation algorithm parameterized by degeneracy; we introduce such an algorithm in Chapter 3. The graph is first edited to have bounded degeneracy, which leads to a bounded approximation factor when computing OCT. The (bounded size) edit set becomes an additive approximation constant when reintroduced into the graph by patching the full solution together. 12
Figure 1.8 (Left-Right) An example tensor network and its line graph transformation, respectively. Given a graph G, its line graph L is constructed by making a vertex per edge in G, and connecting two vertices if their corresponding edges share an endpoint. 13

Figure 3.1 A Chimera $\mathscr{C}_{3,3,4}$ graph. The Chimera location labels of four qubits are highlighted. 27
Figure 3.2 A typical workflow for running QUBO-formulated optimization problems on an AQC processing unit. Finding efficient and effective embedding algorithms is an area of active research. 28
Figure 3.3 A high-level overview of the virtual hardware framework, including the iterative tuning of the virtual embedding ϕ, virtual hardware template $\mathscr{T}$, and the physical embedding ψ. 31
Figure 3.4 The $K_{12,12}$ biclique virtual hardware for Chimera(4, 3, 3). Thick blue edges show allocations to vertical vertex sets, and dashed gray edges show the horizontal vertex set allocations. 33
Figure 3.5 (Left) The Native-Embed embedding with “+”-shaped vertex sets; (Right) The embedding reduced by Qubit-Reduce, with “L”-shaped vertex sets. . . . 35
Figure 3.6 Embedding an 8-vertex problem into $\mathscr{C}_{4,2,2}$ using the OCT-Embed subroutine. For figure readability, vertex u_i is labeled with i. 41
Figure 3.7 Embedding GNP graphs into Chimera(4, 8, 8); data points are the median over 25 random graphs and 10 random algorithm seeds. Experimentally, we observe that the approximation algorithm performs notably better than its approximation factor guarantees, while additionally achieving highly practical run times. 46
Figure 3.8 Embedding GNP graphs into Chimera(4, 8, 8); data points are the median over 10 random graphs and 10 random algorithm seeds. Reduced Fast-OCT-Embed consistently outperforms CMR in both qubits used and run time. 47
Figure 3.9 Qubits used when embedding into Chimera(4, 8, 8); data points are the median over 25 random graphs and 10 random algorithm seeds. OCT-based algorithms consistently embed larger problem than possible with TRIAD. 48

Figure 4.1 Generating frustrated cluster loop (FLC) instances appropriate for the D-Wave 2000Q hardware. This hardware can embed a 64-clique and a 128-biclique, so we range the clique size for the FCL underlying graph $n \in \{64, 96, 128\}$ and scale the number of sampled cycles n/c to fill this range. . . . 62
Figure 4.2 A scatter plot of VC vs. ILP run times on the FCL corpuses. Points above the line are (up to 10×) faster when run on ILP. . . . 66
Figure 4.3 Relative run times computed by dividing VC run times by those from ILP with one thread; the dashed line indicates equality. Data points reported over a synthetic corpus with five random generator seeds. . . . 68
Figure 4.4 Run times (log scale) of all quantum datasets when sorted in order of fastest to slowest when solved with the OCT → VC → ILP formulation and one thread (VC-1T). . . . 71
Figure 4.5 The run time ratio (log scale) of a 4-threaded solver vs. a single-threaded solver using the OCT → VC → ILP formulation. Easier instances have identical solve times, and all but two of the harder instances benefit from the thread increase. . . . 71
Figure 4.6 The run time ratio (log scale) of the single- and four-threaded solvers using the OCT → ILP formulation compared to the single-threaded OCT → VC → ILP formulation (VC-1T). The majority of instances time out at 10 minutes, and no instance is solved faster than with VC-1T. . . . 72

Figure 5.1 A `path` gadget of width 5. The endpoint vertices p_0, p_5 have degree 4, and internal vertices p_1, p_2, p_3, p_4 have degree 5. Solid vertices are internal to the gadget, whereas dashed vertices are interface points to the rest of the graph. . . . 77
Figure 5.2 A `bomb` gadget, with `path` gadget widths determined by r. Solid vertices are internal to the gadget, whereas the dashed vertex is an interface point to the rest of the graph. . . . 78
Figure 5.3 f maps an instance of Vertex Cover to an instance of r-Degenerate Vertex Deletion. . . . 79

Figure 6.1 (Top) A 1D binary MERA with a 16-site lattice and 3 levels of coarsening; three operator placements are highlighted (red, blue, green). (Bottom) Causal cones and final tensor networks for each of the three highlighted operators. Note that the tensor networks for the left-most (red) and right-most (green) operators are isomorphic to one another, but structurally distinct from the middle (blue) operator's network. . . . 92
Figure 6.2 Visualization of the ConSequences pipeline. Externally-generated domain data is parsed into standardized graph formats with the pre-processing utility. The solver dispatcher then allows the user to compute contraction sequences (or their equivalent, e.g. tree decompositions) using external algorithms in Docker containers, then executes a post-processing utility to output standardized formulations of a contraction sequence. . . . 93

Figure 6.3 Run times for the contraction sequence algorithms on select MERA networks, binned by number of qubits possible (number of $\mathscr{L}_0$ sites). All algorithms are timed out at 20 minutes (horizontal dashed line), and a network that remained unsolved by every algorithm is not included. 2D MERA with one operator had 48 of 131 networks that did not finish, and the two operator networks had 193 of 207 that did not finish. 98
Figure 6.4 Run times for the three algorithms on select MERA networks, binned by optimal contraction complexity. All algorithms are timed out at 20 minutes (horizontal dashed line), and a network that remained unsolved by every algorithm is not included. 2D MERA with one operator had 48 of 131 networks that did not finish, and the two operator networks had 193 of 207 that did not finish. 99
Figure 6.5 Run times for freetdi, meiji-e, and quickbb on QAOA circuits for computing MaxCut on r-regular graphs. 25 random regular graphs are generated at each $r, |V|$ level using NetworkX, and algorithms were timed out at 15 minutes (horizontal line). 101
Figure 6.6 Simulation time is tightly correlated with the contraction complexity of a network. While exact algorithms freetdi and meiji-e may generate different tree decompositions and thus contraction sequences with the same treewidth, the differences have little impact on simulation times. . . 103
Figure 6.7 Simulation times of the qTorch tensor network simulator [Fri18] with contraction sequences produced by exact treewidth algorithms vs. Microsoft's LIQUID solver [Wec14]. Total simulation time includes both computation of the contraction sequence using ConSequences and tensor network simulation time using qTorch. A timeout of 900 seconds is used for computing the contraction sequence (horizontal dashed line), and a simulation is not run unless an optimal contraction sequence is found by at least one contraction sequence algorithm. LIQUID is limited to simulations up to 22 qubits. 105

Figure 7.1 (Left) The instance `Promedas6011` from the Parameterized Algorithms and Computational Experiments (PACE) 2017 challenge. (Right) The instance `miles250` from the DIMACS graph coloring challenge corpus. Unlike the tensor networks, these graphs are more hereogenerous in structure. 109
Figure 7.2 (Left) The line graph of a Google Quantum Supremacy graph, generated on a Bristlecone processor lattice with four rows and 24 clock cycles (i.e. circuit depth). (Right) The line graph of a Unitary Coupled Cluster Singles and Doubles (UCCSD) circuit with one fermion and four qubits. Note the prevalence of degree-1, degree-2, and cliques with four vertices; these are caused from the 1-input, 1-output and 2-input, 2-output quantum gates in the circuit. 109
Figure 7.3 (Above-Left) A graph G with eight vertices. (Above-Right) Sets A-E and their margins are inserted into the block sieve trie. (Below) The populated block sieve trie when $k := 6$. 113

CHAPTER

1 INTRODUCTION

In recent years, production-level quantum computing hardware has made its way to market through companies such as IBM, Intel, Google, and Rigetti Computing. Whereas implementing a pair of qubits in a circuit-based model was a feat a decade ago, these new computers are able to sustain 50 superconducting qubits through the execution of thousands of gates. Alternatives to the circuit-based superconducting-qubit models have appeared on the market as well, including IonQ's trapped ion model (with unrestricted connection topology), and D-Wave's adiabatic quantum computing hardware (which supports up to 2000 qubits in a restricted quantum annealer).

The advent of these various hardware models has introduced a plethora of engineering issues, however. Current implementations of qubits are noisy, may not be held in superposition for the length needed to guarantee optimality, and are configured in a connection topology limited by physical restrictions. Mapping problems onto this hardware is also non-trivial and often requires an extensive software pipeline of preprocessing, data conversions, and methods for breaking down real-world problems into pieces small enough to fit on the quantum hardware.

These pipelines are often bottlenecked by unavoidable and difficult (NP-hard) optimization problems. For example, D-Wave Systems' quantum annealer uses *graph minor embedding* to map a problem instance of arbitrary connectivity to their grid-like, sparse hardware topology. Circuit-based quantum models require solving the complex and intertwined *placement, routing, and scheduling* problems; these problems are similar to modern FPGA placement and routing

problems. Tensor networks, a method for classically simulating quantum systems using high dimensional matrices, require computing the optimal order in which to contract the tensors so that the exponentially-scaling tensors are kept as small as possible.

When theoretical solution do exist to these various optimization problems, they often come with caveats. The graph minor embedding needed for quantum annealing hardware can be computed in polynomial time using Robertson and Seymour's forbidden minor theorem [Rob95]. However, this technique scales as a double exponential in the (fixed) problem instance size and is *impractical* for real-world usage. Optimal contraction sequences for tensor networks can be solved in linear time by computing it with treewidth [Mar05], but only when the contraction sequence cost is a fixed constant [Bod98]. In addition to the theoretical issues, these algorithms were also not developed specifically for quantum applications, resulting in poorly optimized software unsuitable for immediate deployment.

We study these key algorithmic issues with three *practical* graph algorithms (odd cycle transversal, bounded degeneracy editing, and treewidth) used in two quantum computing applications (quantum annealing and tensor networks) and. In this context, we use 'practical' to distinguish our implementations from 'efficient' (polynomial-time) algorithms. We have used or developed algorithms with reasonable constants, and employed heuristics to improve run times or solution quality beyond the theoretical guarantees. Additionally, we provide full open source repositories of our backend code and experiment scripts, providing detailed documentation on how to use and extend our work. After tuning these implementations on both benchmark and synthetic data, we return to the quantum application and evaluate our methods against the state-of-the-art.

In the remainder of this chapter we overview an algorithmic workflow framework common to all our applications, examine each chapter's contribution to the literature and related works, and summarize the overarching themes.

1.1 Graph algorithm contributions

At their core, our key results are algorithms for fundamental graph problems – odd cycle transversal, distance to bounded degeneracy, and treewidth. All three problems are NP-hard in general, requiring the use of *parameterized complexity* and *approximation algorithms* to wrangle this intractability. In this section we overview each graph property, its optimization problem variant, and the algorithmic techniques used to produce practical algorithms.

1.1.1 Odd Cycle Transversal

The first graph property is *odd cycle transversal*, the vertex edit distance to a bipartite graph.

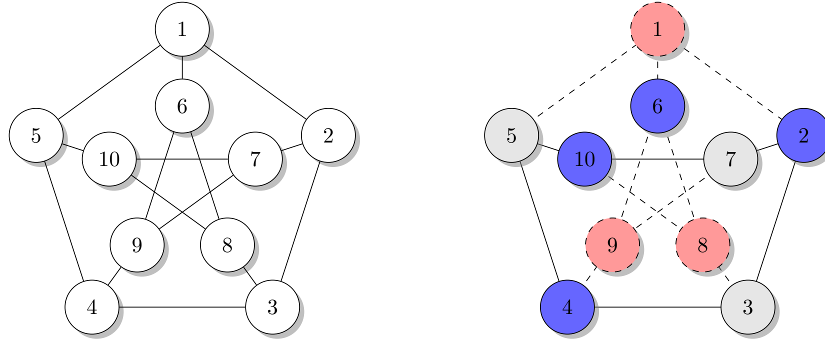


Figure 1.1 (Left) The Petersen graph [Wei00]; (Right) Removing a minimum OCT set (red, dashed) from the Petersen graph yields a bipartite subgraph (blue and gray).

Definition (Odd cycle transversal)**.** *An odd cycle transversal is a set of vertices $S \subseteq V(G)$ such that $G \setminus S$ is bipartite.*

Intuitively, the name comes from the characterization that a graph is bipartite if and only if it contains no odd cycles; a *transversal* will be the minimum set of vertices covering the set of odd cycles. Figure 1.1 shows the Petersen graph and a minimum OCT set (dashed-red vertices) whose removal leaves a bipartite graph.

Computing *an* OCT set S is trivial: let $S = V(G)$. The problem becomes difficult when trying to minimizing the transversal size.

Odd Cycle Transversal (OCT)

Input:	An input graph $G = (V, E)$.
Problem:	Find $S \subseteq V$ such that $G \setminus S$ is bipartite.
Objective:	Minimize $\lvert S \rvert$.

OCT is tightly related to Vertex Cover (VC), the problem of identifying a minimum set of vertices whose removal renders the graph edge-less. This relation implies NP-completeness, as well as APX-hardness, preventing arbitrarily-precise approximation algorithms. In Section 3.4.5, we show a d-approximation algorithm for estimating OCT in linear time on a d-degenerate graph. Empirically, we find that this approximation algorithm outperforms its worst-case ratio and is quite useful in practice (Figure 3.7).

From a parameterized perspective, both OCT and VC are equivalent under the natural parameter. This means that an OCT instance with a solution size of k can be transformed into an VC instance with solution size $O(k)$. In fact, the transformation used in Chapter 4 results in a solution size at most $2k$, making VC-solvers theoretically viable by not introducing large constants.

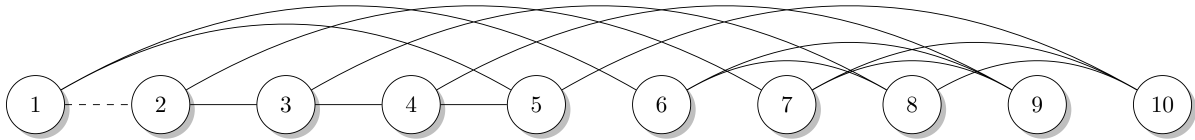


Figure 1.2 An optimal degeneracy ordering on the Peterson graph. Without edits, the graph has degeneracy of at least three, since degeneracy is at least the minimum degree. The degeneracy is also no more than three since each vertex has at most three neighbors to its right side in this *degeneracy ordering*. If vertex 1 or the dashed edge is deleted from the graph then the remaining subgraph has degeneracy two (witness with the same ordering).

Our major contribution to the literature in Chapter 4 is a systematic evaluation of OCT solvers on a corpus of nearly 7000 graphs, both synthetic and quantum (QUBO) problems. This evaluation includes reduction routines from OCT [Wer03] and VC [Aki16], upper-bound heuristics [Wer03; Goo18c] which are also useful for jumpstarting exact algorithms, and exact algorithms solving OCT directly [Hüf09], through a VC-formulation [Aki16], and through an ILP-formulation. We find that a combination of these techniques results in highly-practical solutions, even in limited runtime scenarios (e.g. less than one second). Additionally, we find that the VC-solver and ILP-solver split the exact solution corpus nearly equally, where the optimal choice of solver depends on graph structure. Future work is needed investigating how to predict the best solver to use on real-world datasets based on easy-to-compute graph structure such as degree distribution and degeneracy.

1.1.2 Bounded Degeneracy Editing

The second property we examine is *degeneracy*, a method for decomposing a graph into cores and a measure of the deepest core.

Definition (Degeneracy ordering)**.** *A r-degeneracy ordering of a graph is an ordering of its vertices such that every vertex has at most r neighbors later in the ordering. The smallest r for which there is a r-degeneracy ordering is the degeneracy of the graph.*

A degeneracy ordering of the degeneracy-3 Petersen graph is visualized in Figure 1.2. Unlike OCT, computing the degeneracy of a graph is linear in the number of edges [Bat03]. Instead, we study the problem of computing *how close* a graph is to the class of graphs with a certain (bounded) degeneracy.

r-Degenerate Vertex Deletion (r-DVD)

Input:	An input graph $G = (V, E)$ and positive integer r.
Problem:	Find $S \subseteq V$ such that $G \setminus S$ is r-degenerate.
Objective:	Minimize $\|S\|$.

r-Degenerate Edge Deletion (r-DED)

Input:	An input graph $G = (V, E)$ and positive integer r.
Problem:	Find $S \subseteq E$ such that $G \setminus S$ is r-degenerate.
Objective:	Minimize $\|S\|$.

By standard meta-theorems, vertex deletion to a hereditary (subgraph-closed) property is both NP-complete [Lew80] and APX-hard [Lun93]; however, the difficulty of the edge-deletion variant remained open. It was reasonable to conjecture that r-DED may be easier than r-DVD in general. The special case of $r = 1$ is known as Minimum Spanning Tree and Feedback Vertex Set, respectively, which are polynomial and NP-complete, respectively.

However, in Chapter 5 we show that both r-DED and r-DVD are APX-hard (and trivially NP-complete) with reductions to Vertex Cover. Further, we provide a *bicriteria* approximation algorithm for r-DVD. Whereas a standard approximation has one ratio, α, that bounds the error in solution quality, a bicriteria approximation algorithm introduces a second error bound, β, for another property. More formally, suppose we have a graph G that is OPT vertex removals away from having degeneracy r. Our algorithm will find an edit set S such that $|S| \leq \alpha \cdot OPT$ and $G - S$ will have degeneracy $\beta \cdot r$. In the case of r-DVD, we show a $(4, 4)$-approximation.

Combining the r-approximation for OCT with this $(4, 4)$-approximation for degeneracy editing, in Section 5.4 we show a $(4r')$-approximation algorithm with additive error $4d$ for solving OCT on G, where G has edit distance d to a (user-chosen) target degeneracy r'. By splitting the approximation error between the multiplicative and additive errors, users are able to tune their choice of target degeneracy to minimize total error.

1.1.3 Treewidth

Our final graph structure is *treewidth*, a way of measuring the similarity of a graph to a tree in terms of cut structure. Figure 1.3 visualizes a tree decomposition of the Petersen graph.

Definition (Tree decompositions and treewidth)**.** *A tree decomposition of a graph G is a tree T with a function f mapping nodes in T to bags (sets) of vertices from G, such that the following conditions hold:*

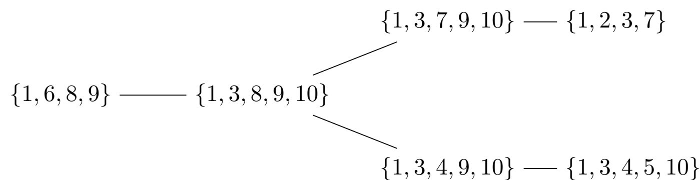


Figure 1.3 A tree decomposition of the Petersen graph with minimum width of four.

1. *All vertices are represented:* $\bigcup_{t \in V(T)} f(t) = V(G)$.
2. *All edges are represented:* $\forall (u, v) \in E(G), \exists t \in V(T)$ *s.t.* $u, v \in f(t)$.
3. *Graph vertices induce a (connected) subtree of* T*: if* $w \in f(r) \cap f(s)$ *for* $r, s \in V(T), w \in V(G)$*, then* $w \in f(t)$ *for all* t *on the path from* r *to* s *in* T*.*

The width of a tree decomposition is $\max_{t \in V(T)} |f(t)| - 1$*, and the treewidth of a graph* G*, denoted* $tw(G)$*, is the minimum width over all valid tree decompositions of* G*.*

Treewidth is commonly used as a data structure for dynamic programming, where a tree decomposition is first computed on the graph instance, which is then traversed inward from the leaves using dynamic programming on the bags. This strategy leads to polynomial algorithms for NP-complete problems if treewidth is assumed to be a constant.

Unfortunately, computing treewidth itself is NP-complete and APX-complete when defined as an optimization problem.

Treewidth ((TW))

Input:	An imput graph G and positive integer w.
Problem:	Construct a tree decomposition T of G.
Objective:	Minimize the width of T.

Due to its importance as a subroutine for larger algorithms, treewidth has been the subject of several practical studies, such as the Parameterized Algorithms and Computational Experiments (PACE) algorithm competition in 2016 and 2017. While these competitions were productive in producing new algorithms to solve competition benchmarks, further work has been needed for adjusting these algorithms to real-world data.

In Chapter 6, we examine how to use treewidth for computing *contraction sequences* on graphs from quantum computing called *tensor networks*, and find that the PACE 2017 algorithms dominate domain-specific algorithms for computing contraction sequences. We further this work in Chapter 7 by tuning the 2017 algorithm by Tamaki [Tam19] for tensor network datasets especially.

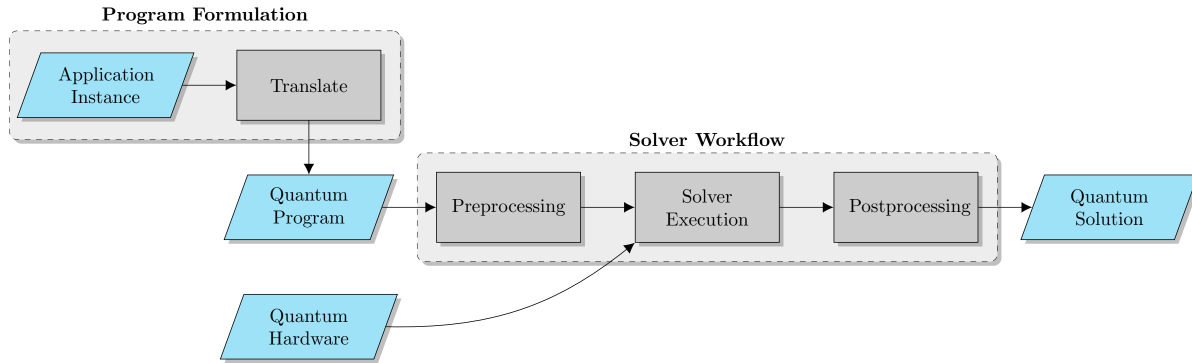


Figure 1.4 A generic algorithmic workflow for running a quantum application. The original application must be encoded into a quantum program, such as a QUBO or a circuit using quantum gates. This program is typically preprocessed before execution, including steps such as reduction, pre-optimization, and compilation for the hardware architecture. After this program is executed on the quantum hardware then a postprocessing step is required to extract the solution and map it back to the original application.

1.2 Quantum computing contributions

1.2.1 A generic quantum pipeline

At a high level, the generic quantum pipeline is straightforward: run a quantum program on quantum hardware to compute a solution (Figure 1.4). Difficulties arise in the details surrounding program formulation and the full solver execution workflow. (Readers unfamiliar with the quantum annealing model or tensor network simulations should consult the detailed overview in Chapter 2).

Program formulation significantly depends on the real-world application at hand and its distance from the program format. For example, quantum annealers naturally solve Quadratic Unconstrained Binary Optimization Problem (QUBO) problems, and any NP-hard problem will innately map to this format through the NP-hardness reduction. A practical issue may arise in this translation, though, where the theoretically-guaranteed no-more-than-polynomial blowup in problem size is actually quite large. In general, this step of formulating a quantum program is similar to that of using a SAT- or ILP-solver, and care should be taken to ensure that execution is tuned for the specific application and hardware available.

Because program formulation is so application specific, it is beyond the scope of our work and we begin the quantum workflow pipeline with a quantum program. For quantum annealing (Chapters 3-5), we assume the quantum program is formulated as QUBO– simply a graph with weights on its vertices and edges. The weights are also irrelevant for our work, which relies on the graph structure alone. For tensor network simulations (Chapter 6-7), the quantum programs

are also graphs, this time representing *tensor networks*. These tensor networks are a simulation of quantum circuits where quantum gates are replaced by tensors, therefore the interactivity of these gates are represented in the tensor network. Again, we are only interested in graph structure, and the tensors maintained in this network are irrelevant for the graph algorithms used. In both applications we can evaluate the graph algorithms on both arbitrary graphs (e.g. generated by common synthetic generators or canonical benchmarks) as well as using data directly from the relevant community (e.g. MERA and QAOA datasets for tensor networks).

Quantum hardware is also described with a graph, where nodes represent qubits and edges represent communication between qubits. For quantum annealing we utilize the Chimera hardware available from D-Wave Systems. For tensor networks no specific hardware topology is needed since we primarily compare results on classical simulators.

Like the program and hardware, the solver workflow heavily depends on the quantum model used. When using a quantum annealer the problem graph must be *embedded as a graph minor* into the hardware graph, its vertex and edge weights adjusted correspondingly. Additional optimizations at this step include adjusting the weights to fit inside a hardware-supported range and minimizing the graph embedding footprint to minimize the number of qubits used. After the solver is run then a distribution of solutions is produced, and postprocessing work includes sampling from this distribution and mapping the final solution back to the original problem.

In tensor network simulations, the primary objective is to contract the network down to a single tensor. Similar to deciding an order in which to multiply matrices, the *contraction sequence* order on this tensor network can effect total execution time exponentially proportional to the largest tensor produced during this process. Once an efficient contraction sequence is identified then the network must be contracted in this order, a straightforward (but computationally intense) matrix multiply. Pre- and post-processing may include identifying the level of granularity needed in the tensors such that the network can be simulated efficiently and the solution quality suffices.

1.2.2 Minor embedding (Chapter 3)

Overview

In our first quantum application we concentrate on improving the minor embedding results into Chimera hardware by utilizing the bipartite aspects of both the program and hardware graphs (Figure 1.5). Given a QUBO instance and an annealer topology, we begin by computing an OCT decomposition on the program graph, splitting the vertices into partite sets and an OCT set. We then use this decomposition to dictate how the QUBO instance is embedded into a complete bipartite minor of the Chimera hardware graph; the bipartite portion is embedded directly, and the OCT set vertices are stretched into chains that occupy both sides. With a full embedding

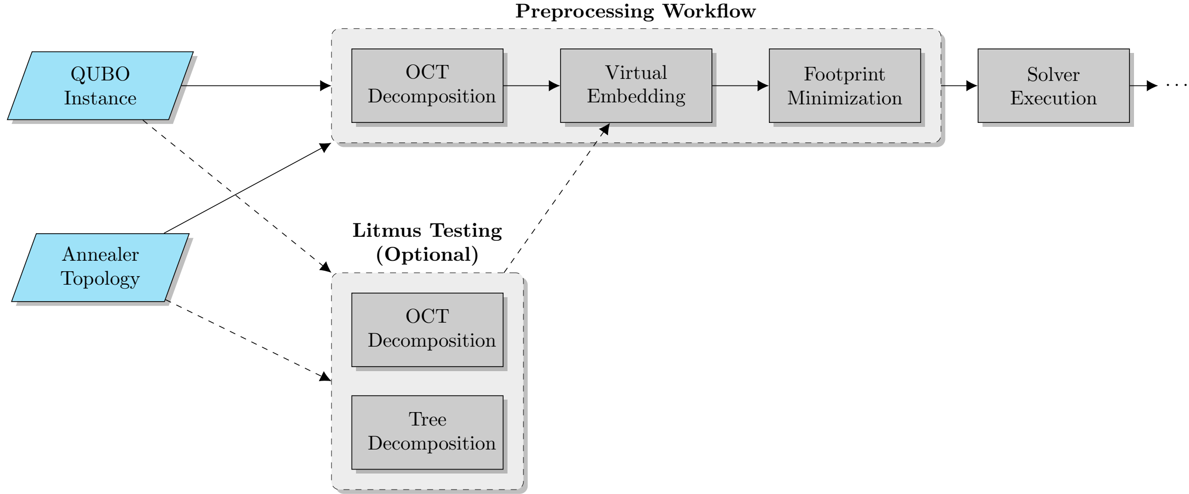


Figure 1.5 Our minor embedding work concentrates on the pre-execution steps. We assume input is provided as a QUBO program and a hardware graph describing the quantum annealer topology. These two graphs are optionally fed into a litmus testing module to evaluate whether the program's graph properties (such as OCT and treewidth) are compatible with the hardware. Once an OCT set has been computed on the program then the QUBO instance is embedded into a virtual hardware layer, followed by footprint minimization for reducing the number of physical qubits used by this embedding. Once these steps are completed, the QUBO instance has been successfully embedded into the annealer graph and execution may begin.

of the QUBO instance into the virtual hardware, we execute local search subroutines to group highly-interactive variables on the qubit assignment, reducing chain length and minimizing the hardware usage.

In experimental results, we find that these embeddings can be computed faster and use less qubits than those found by competitors (TRIAD and the CMR heuristic). These results hold under four synthetic graph generators and three edge density levels.

Our additional theoretical results provide an understanding of when OCT will be easy to solve on a QUBO instance, and when these results may exclude the instance from being embedded into Chimera hardware. We (constructively) show that OCT can be computed in linear time on series-parallel graphs (i.e. graphs with treewidth at most 2). This approach utilizes Eppstein's characterization of series-parallel graphs into nested ear decompositions. These ears can be removed one-by-one such that an optimal OCT set is computed in linear time.

Notably, we also show that OCT can be d-approximated in graphs with degeneracy d. This approximation algorithm provides a way to quickly compute "good enough" OCT solutions while avoiding potential unbounded error. Additionally, the results of Chapter 5 can utilize this algorithm for even tighter approximations for graphs near small degeneracy.

Finally, we show that Chimera graphs are fundamentally limited in their ability to embed

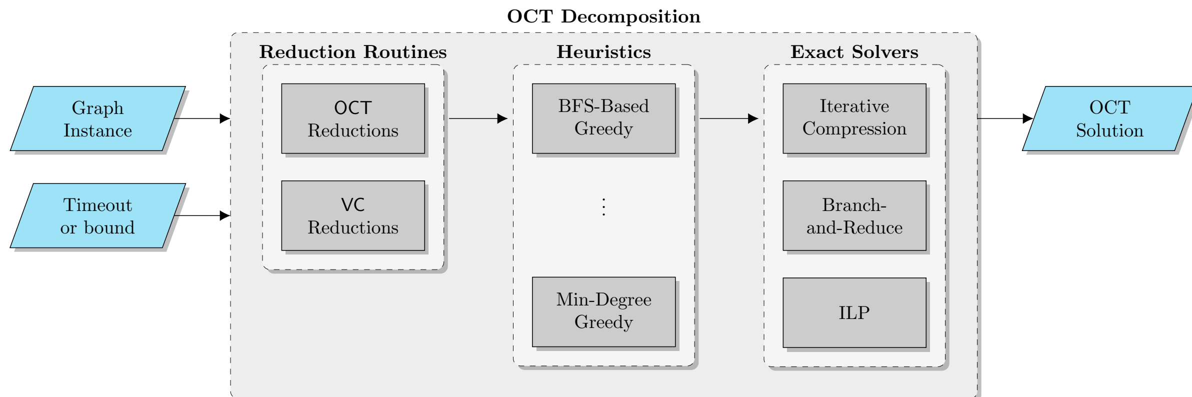


Figure 1.6 This chapter examines how the NP-hard OCT decomposition subroutine should be solved in a variety of real-world use cases. We assume the subroutine is given a graph instance and a stopping criterion (time- or solution quality-based). This graph is initially reduced using routines from the OCT and VC literature. A heuristic ensemble is then run to construct upper bounds on the OCT solution (which may suffice for the stopping criterion). If a smaller solution is required then an exact solver is executed, based on the structure of the graph. The subroutine is completed by outputting an OCT solution certificate.

graphs with large minimum OCT sets. Similar to previous results in treewidth [Kly14], OCT can be used as a *litmus test* to reject programs from embedding before spending much run time attempting to embed.

Related work

Embedding algorithms date back to the original papers on the Chimera hardware by Choi [Cho08; Cho11]. This *TRIAD* embedding mapped a clique of size $LN + 1$ into a corner of the Chimera(L, M, N) hardware. Later work by Boothby, King, and Roy [Boo16] show how to compute the maximal such clique in hardware with hard faults (i.e. missing qubits) present.

Whereas these first two embedding algorithms were exhaustive, a popular stochastic heuristic used in early D-Wave API versions came from Cai, Macready and Roy [Cai14]. This heuristic greedily placed QUBO vertices on the hardware graph and expanded these chains using shortest path and A* algorithms. Future iterations of this API function were kept internal and unpublished.

Building directly off our work in virtual hardware, Hamilton and Humble [Ham17] examine how to compute all minors of a biclique. Future work generalized embedding strategies away from bicliques [Dat19; Oka19].

1.2.3 Improved bipartization subroutines (Chapter 4)

Overview

In the last chapter we saw two distinct needs for a practical OCT subroutine: finding "good enough" decompositions for minor embedding, and finding exact solutions for litmus testing. In this chapter we collect existing methods and systematically evaluate how they perform on a large (7000+ instance) corpus of problem graphs.

As visualized in Figure 1.6, methods for computing OCT are roughly split into reduction rules, heuristics, and exact solvers. We collect reduction rules for OCT directly, as well as those for Vertex Cover (VC), since OCT and VC are similar problems in a parameterized sense [Lok14]. These reduction rules typically apply to sparse areas in graphs and may remove vertices or edges, or add vertices to the solution or bipartite sets. The heuristics come from an existing depth-first search–based approach [Wer03], its breadth-first search variant, a min-degree heuristic (from Chapter 3), and a stochastic heuristic based on degree. These heuristics are combined into an ensemble solver and each constructively provides upper bounds on OCT, which are further useful for jumpstarting exact calculations.

The exact solvers from OCT include an approach based on iterative compression (a technique from parameterized algorithmics), a branch-and-reduce solver from the VC literature, and solving the Integer Linear Programming (ILP) formulation of OCT using CPLEX. We find that iterative compression is not competitive, and the VC and ILP solvers split the datasets fairly evenly. Beyond a corpus of QUBO data, we also generate a synthetic corpus using four random graph generators to mimic the existing graphs.

Related work

Work on exact, combinatorial solvers for OCT date back to Wernicke's branch-and-bound approach [Wer03]. While this exact algorithm is not competitive against the subsequent iterative compression algorithm by Hüffner [Hüf09], the provided reduction routines are still applicable. Theoretical work by Lokshtanov et al [Lok14] provides a basis for suspecting that OCT is actually best solved when reformulated as a VC instance. Akiba and Iwata [Aki16] provide a state-of-the-art VC solver, utilizing reduction routines iteratively with branching to simplify instances after vertices are moved in a branch step. ILP solvers are common in operations research, with IBM's CPLEX being the most well-known commercial software.

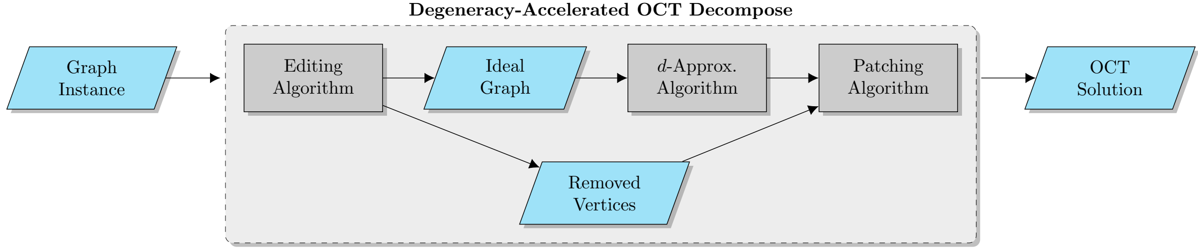


Figure 1.7 A degeneracy editing algorithm fits into the quantum workflow when used in conjunction with an OCT approximation algorithm parameterized by degeneracy; we introduce such an algorithm in Chapter 3. The graph is first edited to have bounded degeneracy, which leads to a bounded approximation factor when computing OCT. The (bounded size) edit set becomes an additive approximation constant when reintroduced into the graph by patching the full solution together.

1.2.4 Distance to bounded degeneracy (Chapter 5)

Overview

In this chapter we examine an unusual technique for approximating optimization problems, where the graph instances close to certain *ideal* properties are approximated based on this distance. This technique is composed of three distinct steps (Figure 1.7).

First, a graph instance is edited to some ideal structure, where *ideal* means that there exists some efficient algorithm for computing the optimization problem on this structure. A common ideal structure from existing literature is treewidth, since many optimization problems on graphs with bounded treewidth can be solved in polynomial time using dynamic programming. We use *bounded degeneracy* as our ideal structure, and provide a (α, β)-approximation algorithm for editing to a (user-provided) target degeneracy. This *bicriterion* approximation algorithm means that if an optimal edit set to degeneracy r has size d, then the algorithm returns an edit set of size at most αd to a graph with degeneracy βr. We show that, for the vertex deletion problem, $\alpha = \beta = 4$.

The second step is to solve the optimization problem on this ideal structure. In our case we utilize the r-approximation algorithm for OCT from Chapter 3; if our graph instance has bounded degeneracy then we have an algorithm for quickly computing an odd cycle transversal with bounded error.

The final step is to patch the removed vertices back into the graph and the optimization solution such that the solution quality still has bounded error. In the case of degeneracy editing, a safe strategy is to add the edit set to the odd cycle transversal, which introduces an additive approximation error. In total, this approach results in an approximation algorithm for OCT with a multiplicative approximation ratio of $4r$ and an additive error of $4d$, when the graph instance G is d edits away from the class of r-degenerate graphs.

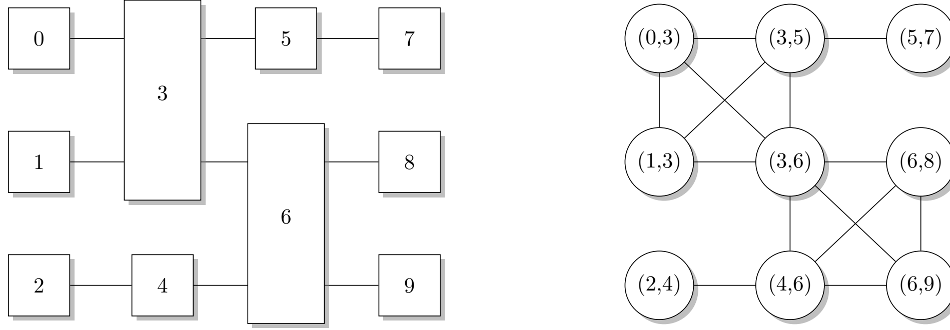


Figure 1.8 (Left-Right) An example tensor network and its line graph transformation, respectively. Given a graph G, its line graph L is constructed by making a vertex per edge in G, and connecting two vertices if their corresponding edges share an endpoint.

In addition to these positive results, we also show that both the vertex and edge editing variations of degeneracy editing are APX-hard, therefore there exists a constant lower bound on the approximation ratio and no polynomial-time approximation scheme (PTAS) exists.

Related work

It was known that the vertex editing variant was APX-hard due to being a monotone (subgraph-closed) property [Lun93]. Several common optimization problems can be phrased as degeneracy editing: Vertex Cover (VC) is vertex-editing to degeneracy-0, spanning tree is edge-editing to degeneracy-0, and Feedback Vertex Set (FVS) is editing to degeneracy-1. Whereas edge-editing to degeneracy-0 is easy compared to the vertex-editing variant, our result shows that for degeneracy-1 and above it is APX-hard.

Mathieson [Mat10] additionally showed that the problem is $W[P]$-hard, therefore no fixed-parameter tractable (FPT) algorithm exists.

1.2.5 Tensor network simulations (Chapter 6)

Overview

In our second quantum application, we evaluate methods for computing *contraction sequences* on tensor networks, a classical simulation method for quantum circuits. Given a quantum circuit, each qubit is replaced with a tensor of sufficient dimension to represent the quantum states; these tensors are then contracted together in order to evaluate the circuit. The order in which these tensors are contracted determines the highest dimension tensor created over the course of evaluating the network. Finding the optimal sequence over all possible orderings, then, is the (NP-hard) optimization problem of interest.

Contraction sequences are related to the structural graph theory notion of *treewidth* through the *line graph* transformation on a network (Figure 1.8). While several domain-specific heuristics exist for computing contraction sequences, this relation to treewidth opens the door for evaluating competition-level treewidth solvers developed by the parameterized algorithms community [Del18]. We provide a full testing hardness for evaluating both treewidth-based methods and domain-specific contraction sequence algorithms. Included in this harness is a full suite of input/output conversion scripts and a standardized interface to all contraction sequence algorithms, which are wrapped in Docker images to provide platform independence.

Evaluating these algorithms in two applications, we identify a treewidth algorithm with faster run times on smaller datasets (freetdi) and another that scales better as treewidth increases (meiji-e).

Previous experiments stipulated that contraction sequences must be computed outside of recorded simulation time, i.e. the time to construct an optimal sequence dominated the full simulation time [Fri18]. However, we show that modern treewidth methods execute in the same order of magnitude as the tensor contraction runtimes.

Related work

Contraction sequence algorithms are a fundamental step in the software pipeline needed for executing tensor network simulations. Early work in tensor networks showed the correspondence between contraction sequences and treewidth [Mar05]. However, applications tended to deploy their own heuristics in order to avoid the NP-hardness of solving treewidth exactly. It remained an open question whether these heuristics were competitive against exact or heuristic solvers developed for treewidth.

Previous work on MERA networks [Vid03] used a netcon algorithm [Pfe14b; Pfe14a], implemented as an external C package driven by MATLAB and fundamentally computing a breadth-first search over the search space of all possible contraction sequences. While this approach excelled at small networks (less than a dozen vertices) due to the low overhead, it quickly became untenable for larger networks (c.f. Table 6.1).

Another application, the qTorch tensor network simulator [Fri18] running on QAOA circuits for computing Max Cut, utilized the quickbb treewidth algorithm on the network line graphs directly. Further, the authors compared their QTorch suite against Microsoft's LIQUID simulator. Unlike tensor network simulators, which minimize memory usage by coordinating the contraction of tensors, LIQUID simulates the full Hamiltonian space. This approach is more natural for dense circuits (networks), whereas tensor networks are able to utilize sparse edge structure.

Treewidth algorithms are historically a topic of theoretical development, but in 2016 and 2017 it became the topic of the inaugural Parameterized Algorithms and Computational Experi-

ments (PACE) competitions [Del17; Del18]. Efficient implementations of dynamic programming algorithms took first prizes each year [Lar17], with alternative (and better scaling) algorithms also developed [Tam19; Oht17].

1.2.6 Improved contraction sequence subroutines (Chapter 7)

Overview

In the previous chapter we demonstrated how treewidth algorithms are not simply an alternative to domain-specific heuristics for computing contraction sequence, but rather clearly dominate the benchmarks. In this chapter we examine methods for tuning the PID algorithm [Tam19] for tensor networks. Specifically, we evaluate three potential optimizations:

1. Reduction routines to remove structure with treewidth 2 or less from the network instance.
2. A judicious choice of which safe separator heuristic routines to employ, based on the network structure.
3. Parameter tweaks to the *blocksieve* data structure to prevent unnecessary resizing.

Expanding our data corpus, we evaluate these improvements on the following corpora:

1. Google “Quantum Supremacy” datasets. Developed specifically to demonstrate quantum supremacy on near term (Google) quantum devices, these graphs are constructed to rapidly increase treewidth as the circuit increases in depth [Boi18].
2. UCCSD instances. Generated using the eXtreme-scale Accelerator programming framework (XACC) [McC20], these circuits are generated using the UCCSD operator.
3. PACE 2017 training, testing, and bonus datasets [Del18]. These graphs were used for the most recent PACE treewidth competition, along with a bonus corpus of more difficult instances.
4. DIMACS graph coloring instances. The canonical corpus for evaluating competition solvers for various optimization problems, these graphs were used in the original report by Tamaki [Tam19].

Experimental results show that reduction routines are effective for removing both vertices and edges from the instances, with consistently (i.e. low standard deviation) smaller instances on both tensor network corpora. These removals led to at most a 20% speedup on both corpora, with the `uccsd` also having a lower bound of 10% speedup. Treewidth instances were similarly reduced, with the addition of up to 40% speedup on some `pace2017` bonus instances.

In the safe separator heuristic experiment we found that the `fill` heuristic was essential for `pace2017` data, but otherwise was not needed. Using the `degree` heuristic on `uccsd` results in a 10-90% speedup with no downside. The `dimacs` and `google-qs` datasets were generally unaffected by these heuristics.

Finally, the blocksieve data structure improvements make no discernible difference.

In future work, this algorithm would benefit most by a reimplementation that moved from heap-based objects to storing them on the stack. We find that real run time is at most 7.6× the user run time, where the only parallelism in this Java implementation is the garbage collector. Shifting to stack-based memory allocations would not only reduce the garbage collector runtime, but provide the memory localization leveraged by modern CPU caches.

Related work

Treewidth solver implementation tuning has proven effective before, with the PACE 2017 winner using a tuned C++ algorithm from the 2016 winning Java implementation. Using reduction routines on treewidth also has precedent [Bod01; Eij07].

1.3 Summary

In summary, we study three graph structures in the context of two modern quantum computing applications. We study all five individually:

1. For quantum annealing, we show how to provide structure to embedding heuristics with graph bipartization. (Chapter 3)

2. Further studying graph bipartization, we identify which methods are best when solving exactly or under various timeouts, on over 8000 graph instances. (Chapter 4)

3. Given an arbitrary graph, we show how to quickly approximate its distance to the class of bounded degeneracy graphs. We also show that a significantly more accurate approximation cannot exist. (Chapter 5)

4. For the simulation of quantum circuits with tensor networks, we extensively study which contraction sequence methods are fastest in two distinct applications. We find that algorithms which address the problem of computing treewidth directly are the most efficient. (Chapter 6)

5. We study how a treewidth solver can be tuned based on graph instance structure, evaluating several options over both tensor network and treewidth benchmarks (Chapter 7)

While these five are presented discretely, in practice there is significant overlap between both graph structure and application:

1. The treewidth graph structure fundamentally underlies the quantum problem of funding a contraction sequence [Mar05]. However, as a minor-closed metric, treewidth is also useful for recognizing which problem graphs cannot be embedded into quantum annealing hardware [Kly14]. (Chapters 3 and 6-7)

2. Similarly, OCT is both a metric for litmus testing embeddability, and a structure that can be algorithmically exploited for faster embedding heuristics. (Chapter 3)

3. Editing to bounded degeneracy is an optimization problem in its own right, but is a useful subroutine for computing graph bipartization instances with close to small degeneracy. (Chapters 3 and 5)

4. Bounded treewidth implies bounded degeneracy, therefore treewidth solvers additionally provide a distance-to-degeneracy estimate. A litmus testing ensemble for minor embedding could use treewidth and OCT as metrics, then proceed with an OCT-based embedding method that utilized the degeneracy editing subroutine for more precise OCT estimations. (Chapters 3-7)

This work makes a case for structural graph theory as a building block in these larger quantum computing software pipelines. Graph structure is present in the input data, whether it is a quantum annealing program or quantum circuit. By examining, litmus testing, and exploiting various structures algorithmically, subroutines in this larger software pipeline can be tuned for specific applications. Future work involving additional structures is of interest.

CHAPTER

2

BACKGROUND

2.1 Graph Theory

Readers are encouraged to reference Diestel [Die05] for detailed definitions.

2.1.1 General Notation

For a *graph* G, we denote its *vertices* with $V(G)$ and *edges* with $E(G)$. An edge (u, v) has *endpoints* u and v. If clear from context, the number of vertices and edges will be denoted n and m, respectively. A *bipartite* graph has the property that its vertex set can be partitioned into *partite sets* such that every edge has one endpoint in each partite set. A *subgraph* with vertices $X \subseteq V(G)$ is denoted $G[X]$. Two vertices u, v are *adjacent* if $(u, v) \in E(G)$, and the set of all vertices adjacent to v is its *neighborhood* $N(v)$. The *degree* of v is $|N(v)|$, and a graph is r-*regular* if every vertex has degree r. An *edge contraction* of (u, v) adds a new vertex uv with neighborhood $(N(u) \cup N(v)) \setminus \{u, v\}$, and then removes vertices u, v.

2.1.2 Structures, Orderings, and Editing

Throughout this work we use graph structures to define tractable subclasses of hard optimization problems (e.g., a linear-time algorithm for OCT in Chapter 3) and to define editing problems to ideal structures (e.g., editing to bounded degeneracy in Chapter 5). In this subsection we define

the common structures, including alternate formulations as vertex and edge orderings. We begin with degeneracy, one of the fastest-to-compute structures.

Definition (Degeneracy)**.** *A graph is r-degenerate if every induced subgraph contains a vertex of degree at most r. The maximum such r is the graph's* degeneracy*.*

The degeneracy of a graph can be computed using the notion of nested k-cores.

Definition (k-cores and -shells)**.** *The k-core of a graph is the maximal subgraph in which all vertices have degree at least k. The k-shell is the set of vertices in the k-core but not the $(k+1)$-core. The degeneracy of a graph is the largest k such that the k-core is non-empty.*

Definition (Degeneracy ordering)**.** *A r-*degeneracy ordering *of a graph is an ordering of its vertices such that every vertex has at most r neighbors later in the ordering.*

A degeneracy ordering can be computed by iteratively removing a vertex of minimum degree, and the k-core and k-shells can be computed during this procedure by noting when the minimum degree increases. The run time of this algorithm is $O(m)$ when lookup tables are used [Bat03]. Figure 1.2 depicts a degeneracy ordering of the Petersen graph.

Another structure for decomposing a graph is a tree decomposition.

Definition (Tree decompositions and treewidth)**.** *A* tree decomposition *of a graph G is a tree T with a function f mapping nodes in T to* bags *(sets) of vertices from G, such that the following conditions hold:*

1. *All vertices are represented: $\bigcup_{t \in V(T)} f(t) = V(G)$.*
2. *All edges are represented: $\forall (u,v) \in E(G), \exists t \in V(T)$ s.t. $u, v \in f(t)$.*
3. *Graph vertices induce a (connected) subtree of T: if $w \in f(r) \cap f(s)$ for $r, s \in V(T), w \in V(G)$, then $w \in f(t)$ for all t on the path from r to s in T.*

The width of a tree decomposition is $\max_{t \in V(T)} |f(t)| - 1$, and the treewidth *of a graph G, denoted $tw(G)$, is the minimum width over all valid tree decompositions of G.*

Unlike degeneracy, computing the treewidth is NP-hard and practical algorithms are of intense interest [Mar05]. However, similar to degeneracy, treewidth can also be expressed with an ordering.

Definition (Elimination ordering)**.** *Given a graph G, an* elimination ordering *is an ordering of its vertices $\pi = v_1, v_2, \ldots, v_n$. The* fill-in graph *$G_\pi$ is constructed by iterating over v_i from $i = 1$ to n and adding edges to make the neighbors of v_i in $G[v_i, \ldots, v_n]$ a clique. The width of an ordering is the largest degree of a vertex witnessed during the fill-in process.*

A graph has a decomposition of width k if and only if there exists an elimination ordering of size k; see [Bod98].

Another structure (from tensor network simulations) is the notion of contraction complexity.

Definition (Contraction Complexity (cc))**.** *A contraction sequence is an ordering of a graph's edges, and the complexity of a contraction sequence S is the largest degree of a merged vertex created by contracting the graph according to S. The contraction complexity (cc) of a graph is the minimum complexity over all possible contraction sequences.*

Surprisingly, treewidth can be viewed as a vertex-centric formulation of the edge-centric contraction complexity, via a transformation of the underlying graph G to its *line graph* $L(G)$. The line graph is constructed with $V(L(G)) = E(G)$ and $E(L(G)) = \{(e_1, e_2) \mid e_1 \neq e_2 \in E(G)$ s.t. e_1, e_2 share a common endpoint$\}$. The treewidth of $L(G)$ therefore captures the same notion.

Theorem (Markov and Shi [Mar05])**.** *The contraction complexity of a graph equals the treewidth of its line graph.*

In addition to structures on graphs and orders of vertices and edges, we also want to consider edits on graphs.

Definition (Vertex cover)**.** *A vertex cover is a set $S \subseteq V(G)$ such that $G \setminus S$ is edgeless.*

Definition (Odd cycle transversal)**.** *An odd cycle transversal is a set of vertices $S \subseteq V(G)$ such that $G \setminus S$ is bipartite.*

Finding a vertex cover (odd cycle transversal) is trivial by letting $S = V(G)$. Finding the smallest vertex cover (odd cycle transversal) is NP-hard and practical algorithms are of interest [Goo18a].

2.1.3 Synthetic Graph Generators

In our experiments we scale beyond real-world benchmark data by generating synthetic graphs from several common generators. The most basic is the first generator from Erdos and Renyi.

Definition (Erdos-Renyi [Erd60])**.** *The Erdos-Renyi (GNP) graph generator takes as input a number of vertices n and probability p. Each edge is included in the generated graph with probability p.*

Whereas the only tunable property to Erdos-Renyi is the edge density, the Chung-Lu model matches a provided degree distribution.

Definition (Chung-Lu [Chu02]). *The Chung-Lu graph generator takes as input a degree distribution* $(d_1, \ldots, d_n)$*, and adds an edge* uv *with probability*

$$P_{uv} = \frac{d_u d_v}{\sum_{i=1}^{n} d_i}.$$

Finally, the preferential attachment model by Barabasi and Albert produces a severely biased degree distribution for a fixed edge density.

Definition (Barabasi-Albert [Alb02]). *The Barabasi-Albert graph generator takes as input an initial set of vertices, a positive integer constant* c*, and a number* n *of additional vertices. Each new vertex is added to the graph with* c *neighbors with probability proportional to the neighbors' current degree.*

2.2 Optimization and Algorithms

2.2.1 Optimization Problems

Previously we defined the editing sets *vertex cover* and *odd cycle transversal*, and the structure *degeneracy*. We formalize minimization problems for finding small edits.

Vertex Cover (VC)

Input: An input graph $G = (V, E)$.
Problem: Find $S \subseteq V$ such that $G \setminus S$ is edgeless.
Objective: Minimize $|S|$.

Odd Cycle Transversal (OCT)

Input: An input graph $G = (V, E)$.
Problem: Find $S \subseteq V$ such that $G \setminus S$ is bipartite.
Objective: Minimize $|S|$.

r-Degenerate Vertex Deletion (r-DVD)

Input: An input graph $G = (V, E)$ and positive integer r.
Problem: Find $S \subseteq V$ such that $G \setminus S$ is r-degenerate.
Objective: Minimize $|S|$.

r-Degenerate Edge Deletion (r-DED)

Input: An input graph $G = (V, E)$ and positive integer r.
Problem: Find $S \subseteq E$ such that $G \setminus S$ is r-degenerate.
Objective: Minimize $|S|$.

As noted in [Lok14], an instance of OCT can be solved as an instance of VC by creating an auxiliary graph $G' = (V_1 \cup V_2, E_1 \cup E_2 \cup E')$, where $V_i = \{v_i \mid v \in V\}$ and $E_i = \{(u_i, v_i) \mid (u, v) \in E\}$ (for $i = 1, 2$), and $E' = \{(v_1, v_2) \mid v \in V\}$. A solution S to the VC instance can be mapped to a solution S' for the OCT instance with $S' = \{v \mid v_1 \in S \text{ and } v_2 \in S\}$.

Additionally, each problem can be reformulated as an Integer Linear Programming instance to be solved with an industrial-grade solver (e.g., CPLEX [IBM17]):

Odd Cycle Transversal (ILP-Formulation) [Hüf09]

Input: $G = (V, E)$.

$$
\begin{aligned}
\text{Minimize} \quad & \sum_{v \in V} c_v & \\
\text{s.t.} \quad & s_v + s_u + c_v + c_u \geq 1 & \forall (u, v) \in E \\
& s_v + s_u - c_v - c_w \leq 1 & \forall (u, v) \in E \\
& s_v \in \{0, 1\} & \forall v \in V \\
& c_v \in \{0, 1\} & \forall v \in V
\end{aligned}
$$

Vertex Cover (ILP-Formulation) [Aki16]

Input: $G = (V, E)$.

$$
\begin{aligned}
\text{Minimize} \quad & \sum_{v \in V} x_v & \\
\text{s.t.} \quad & x_u + x_v \geq 1 & \forall (u, v) \in E \\
& x_v \in \{0, 1\} & \forall v \in V
\end{aligned}
$$

The OCT solution can be recovered from the first formulation with $S = \{v \mid c_v = 1\}$, and a VC solution from the second with $S = \{v \mid x_v = 1\}$. See [Hüf09] and [Aki16] for more details on how these ILP formulations are derived.

2.2.2 Approximation Algorithms

When finding exact solutions is too computationally difficult, one technique for making the problem tractable is to allow *approximate* solutions. These approximations are not simply heuristics with unpredictable solution quality, but are defined to be within some bound of an optimal solution.

Definition (α-approximation algorithm)**.** *For a minimization problem with optimal cost OPT, an α-approximation algorithm is a polynomial time algorithm that returns solutions with cost at most αOPT.*

Note that α can be a constant or function. For editing problems an approximation algorithm with a single factor may not be sufficient; in this case we use bicriteria approximation algorithms.

Definition ((α, β)-approximation algorithm)**.** *For editing problems with an optimal edit cost of OPT to a property parameterized by t, an (α, β)-approximation algorithm is a polynomial time algorithm that returns edit sets of cost αOPT to a property parameterized by βt.*

For example, in Chapter 4 we provide a $(4, 4)$-approximation algorithm for editing to degeneracy r. This algorithm produces edit sets $4\times$ larger than an optimal edit set to degeneracy r, but these edit sets are allowed to leave a graph of degeneracy $4r$.

Problems can have arbitrarily-precise approximation schemes, known as PTAS.

Definition (Polynomial-time approximation scheme (PTAS))**.** *A polynomial-time approximation scheme (PTAS) for a minimization problem takes as input an error factor $\epsilon > 0$ and will return an $(1 + \epsilon)$-approximation algorithm with run time $O(n^\epsilon)$.*

The cost of arbitrarily precision, therefore, is (potentially exponential) growth in run time. However, the existence of a PTAS shows that no lower bound exists when approximating the problem, which may lead to flexibility when creating faster approximation algorithms.

2.2.3 Approximation Hardness

One method for showing that no PTAS exists (up to standard complexity assumptions) is to show that a problem is APX-hard. Similar to NP-hardness, APX-hardness is shown by reducing every problem in APX to the problem at hand; note that by definition it suffices to reduce a single APX-hardness problem to the current problem. Because some problems in APX (e.g., VC) have proven lower bounds, the problem at hand must also have a lower bound. In this work we use strict reductions.

Definition (Strict Reduction)**.** *Suppose X and Y are optimization problems. Let x be an arbitrary instance of X and $f(x)$ an instance of Y for some polynomial-time computable function f.*

Additionally, let y be an arbitrary solution to $f(x)$ and $g(y)$ a solution to x for some polynomial-time computable function g. A strict reduction of X to Y is shown by providing the functions f, g such that

1. $OPT_X(x) = OPT_Y(f(x))$
2. *and* $COST_X(g(y)) \leq COST_Y(y)$,

where $OPT_P(\ell)$ is the optimal solution cost for instance ℓ of problem P, and $COST_P(s)$ is the cost of solution s for problem P.

In addition to manually proving hardness using a reduction, Lund and Yannakakis have shown that node editing problems in general are difficult.

Theorem (Maximum subgraph problems are APX-hard [Lun93])**.** *Let X be an optimization problem for maximizing an induced subgraph with property Π on graph G. If Π is closed under induced subgraphs then X is APX-hard.*

Relevant to our work, the property of being *bipartite* is closed under induced subgraphs, therefore vertex-editing a graph to be bipartite is APX-hard. Editing to bounded degeneracy and bounded treewidth are problem that also fall under this metatheorem.

2.2.4 Parameterized Algorithms and Complexity

Whereas approximation algorithms apply to all instances and are allowed to return a (bounded) non-optimal solution, another approach to develop fast algorithms to NP-hard problems is to select a subclass of instances with particular properties. The field of parameterized algorithms and complexity provides several tools to achieve this effect. For our purposes, it suffices to consider the complexity class FPT.

Definition (Fixed-parameter tractability)**.** *A decision problem is fixed-parameter tractable (FPT) if it can be solved with an algorithm with run time $O(f(k)n^c)$ for some fixed parameter k, instance size n, arbitrary function f and constant c.*

One example from Chapter 5 is that OCT can be solved in time $O(3^k kmn)$ when the OCT instance has solution size k. When k is (expected to be) small then an FPT can very quickly find optimal solutions.

CHAPTER

3

MINOR EMBEDDING ALGORITHMS FOR QUANTUM ANNEALING

3.1 Introduction

Adiabatic quantum computing (AQC) is a model of computation that utilizes quantum mechanics to solve difficult optimization problems. As originally proposed by Farhi et al. [Far00], AQC relies on the dynamical evolution of a quantum state under a Hamiltonian that changes adiabatically from an initial to final form. This computational model uses the final Hamiltonian to express an optimization problem such that adiabatic evolution will recover the corresponding ground state.

In its most general form, the AQC model is equivalent to other universal quantum computing models. However, any limitation on the Hamiltonian forms may reduce the power of the computational model. Recently, an embodiment of the AQC model with a restricted Hamiltonian was developed using superconducting flux qubits by D-Wave Systems Inc. This quantum processor provides a large number of addressable qubits (up to 2048 in the latest D-Wave 2000Q processor) that implement a programmable Ising model over a restricted geometry. While not a universal quantum computer, the D-Wave processor has been shown to produce quantum effects and yield time-to-solution orders of magnitude faster than classical algorithms [Den16; Kin19]. Use of this *quantum annealer* [Kad98] has evolved beyond the design stage to testing and deployment, with recent applications including computational chemistry, NP-hard graph problems, image

recognition, and more [Den16; Kas11; Luc14; Nev08; Rie15; Ven15].

A key step in using current AQC-based processors is compiling the executable program that will run on hardware with restricted connectivity [Hum14; Bri17]. Both the problem and hardware layouts are conventionally represented using graphs with the problem defined by variables connected with dependencies and the hardware layouts defined by qubits connected with couplers. Under this graph-theoretic formulation, the compilation process reduces to the NP-hard problem of *minor embedding* the problem graph into the hardware graph. In practice, this step represents a limitation bottleneck for the end-to-end program performance because existing embedding algorithms take orders of magnitude longer to execute than the quantum annealer itself [Hum16]. Furthermore, no efficient universal embedding algorithm exists, with past algorithms addressing specific classes of problem instance (e.g. complete graphs, very sparse graphs, etc.) and hardware instance (e.g. D-Wave Chimera graph, etc.), along with a myriad of additional assumptions (e.g. fault-free hardware, parameter values, etc.). However, given the disjoint development of algorithms for these specialized instances, the resulting techniques cannot be combined in a common framework.

To address this incompatibility, we introduce a graph-theoretic framework for developing tuned and modularized embedding algorithms. This framework introduces the concept of a *virtual hardware* graph that provides a judiciously simplified representation of the physical hardware graph, greatly reducing the complexity of embedding subroutines. Many existing embedding algorithms are compatible with the virtual hardware layer and we rewrite them as modular subroutines. We then introduce generalized reduction subroutines for minimizing the hardware footprint of a given embedding. We are able to apply these reduction subroutines to the embedding algorithms emulated by our framework, producing notable improvements for reducing hardware footprint.

As a proof of concept, we provide a complete bipartite virtual hardware compatible with the D-Wave Chimera hardware structure. By exploiting bipartite problem structure with an odd cycle transversal decomposition (OCT), we are able to provide embeddings for edge-dense problem graphs. We additionally present a linear-time approximation algorithm for computing OCT decompositions, leading to fast embedding algorithms. Further use cases are provided by Hamilton and Humble [Ham16].

Finally, we provide an efficient implementation of the full virtual hardware framework, including new and existing embedding and reduction subroutines, available at `https://github.com/TheoryInPractice/aqc-virtual-embedding`. Experimentally, we find that this framework is able to unify and expand on existing embedding algorithms, providing baseline tools for future development. Further, we find that OCT-based embedding algorithms perform better – in run time, size of problem graph embedded, and number of qubits used – than the existing TRIAD and CMR algorithms [Cai14; Cho11].

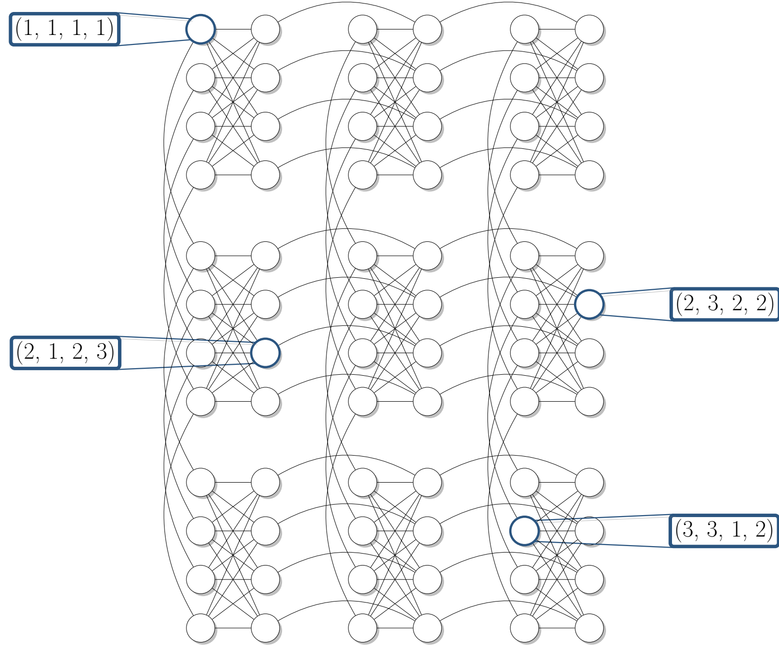


Figure 3.1 A Chimera $\mathscr{C}_{3,3,4}$ graph. The Chimera location labels of four qubits are highlighted.

The manuscript is organized as follows: Section 2 introduces adiabatic quantum computing and the D-Wave hardware, including an overview of related work. Section 3 defines virtual hardware and a stack of baseline subroutines – the graph-theoretic framework – and details the emulation and enhancement of existing embedding algorithms using our framework. Section 4 introduces a new embedding subroutine that exploits bipartite structure in problem and hardware graphs by using an odd cycle transversal decomposition and biclique virtual hardware, respectively; we additionally present a new, fast approximation algorithm for computing an odd cycle transversal. Section 5 contains experimental results of embedding algorithms detailed in previous sections. Finally, we summarize, present our conclusions, and outline future work in Section 6.

3.2 Background

We denote the complete graph on n vertices as K_n and the complete bipartite graph on $n = n_1 + n_2$ vertices with partite sets of order n_1, n_2 as K_{n_1,n_2}. As shorthand, we also refer to complete bipartite graphs as *bicliques*. We also define the contraction of a connected subgraph H as the iterative contraction of its edges (order does not matter due to connectivity). For a set S we denote its power set as $\mathscr{P}(S)$.

Current and prior D-Wave hardware layouts are based on the more general *Chimera graphs*.

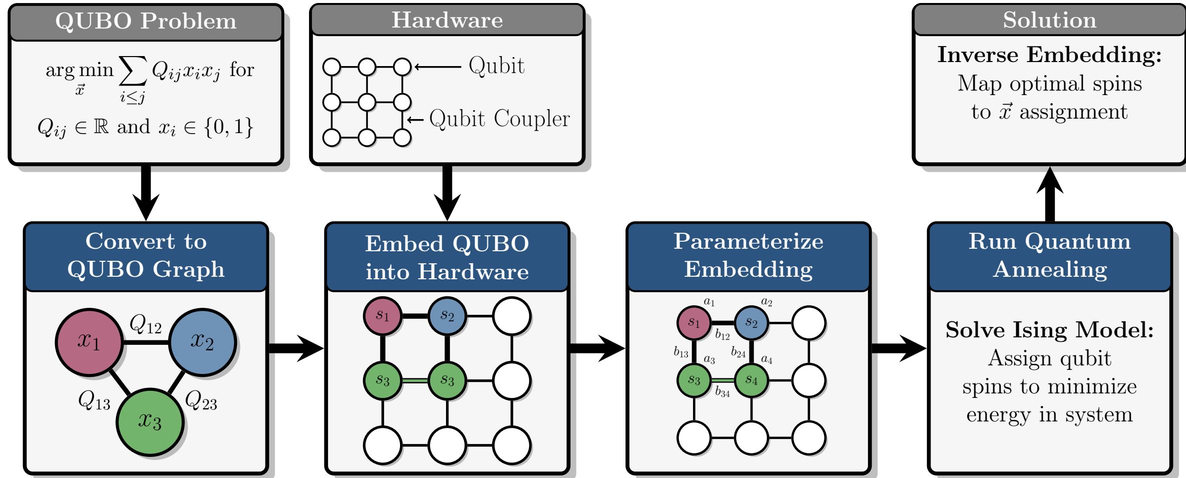


Figure 3.2 A typical workflow for running QUBO-formulated optimization problems on an AQC processing unit. Finding efficient and effective embedding algorithms is an area of active research.

Visualized in Fig. 3.1, a Chimera graph $\mathscr{C}_{L,M,N}$ is an $M \times N$ grid of biclique $K_{L,L}$ cells. For example, the latest D-Wave 2000Q hardware is based on a $\mathscr{C}_{4,16,16}$ graph. In the context of Chimera hardware, we assume that the qubits are labeled by their location in the Chimera layout: $(\ell_r, \ell_c, \ell_p, \ell_h)$ where $1 \leq \ell_r \leq M$ identifies a row, $1 \leq \ell_c \leq N$ identifies a column, $\ell_p \in \{1, 2\}$ identifies a partite set, and $1 \leq \ell_h \leq L$ denotes the in-cell height index.

3.2.1 Minor Embedding for Adiabatic Quantum Programming

Programming a quantum annealer, such as the D-Wave hardware, requires setting the parameters that define the underlying Ising Model. This process includes defining the positive and negative spins as variable assignments such that logical dependencies are maintained within the restricted connectivity of the hardware graph. Recently, several efficient compilation methods have been proposed for managing this process [Cho08; Hum14; Rie15; Ven15].

A generalized compilation pipeline is shown in Fig. 3.2. A common entry point into these compilation frameworks is the quadratic unconstrained binary optimization (QUBO) problem. Given variables $x_1, \ldots, x_n$ where $x_i \in \{0, 1\}$ and constants $c_{ij} \in \mathbb{R}$, the QUBO problem is to compute

$$\arg\min \sum_{i\leq j} c_{ij}x_ix_j.$$

QUBO has become a standard input format for quantum annealers, similar to the linear program format used in efficient classical solvers such as CPLEX. Many constrained optimization

problems can be converted directly to QUBO form [Bor02].

A QUBO can be converted directly into a graph P with vertices $V(P) = \{x_1, \ldots, x_n\}$, edges $E(P) = \{(x_i, x_j) \mid i \neq j, c_{ij} \neq 0\}$, vertex weights c_{ii}, and edge weights c_{ij} for $i \neq j$. Viewing a QUBO as a graph is particularly useful when selecting sets of physical qubits to represent the QUBO variables, since this assignment is known as graph *minor embedding*:

Definition (Minor Embedding). *Given two graphs P and H, a minor embedding of P into H is a function $\phi : V(P) \rightarrow \mathscr{P}(V(H))$ that assigns each vertex in P to a vertex set from $V(H)$ such that the following properties hold:*

1. *Vertex sets cannot overlap: $\phi(u) \cap \phi(v) = \emptyset$ for all distinct $u, v \in V(P)$.*
2. *Vertex sets induce connected subgraphs: $H[\phi(u)]$ is connected for every $u \in V(P)$.*
3. *Edges are represented: $(u, v) \in E(P) \rightarrow (u', v') \in E(H)$ for some $u' \in \phi(u)$ and $v' \in \phi(v)$.*

From a graph-theoretic perspective, this embedding defines the vertex deletions and edge contractions necessary to find P as a minor of H. From the physics perspective, this embedding assigns an appropriate set of *physical qubits* to collectively represent a *logical qubit*, and QUBO weights are adjusted for this embedding by distributing each logical qubit's weight over its vertex sets' physical qubits [Cho08]. Hence, compiling a QUBO into AQC hardware reduces to the problem of finding a minor embedding.

The problem of finding a minor embedding is NP-hard for general graphs, witnessed with a trivial reduction from SubgraphIsomorphism. The most famous minor-embedding result comes from the Robertson-Seymour graph minor theory [Rob95], which implies that there is a polynomial-time algorithm for finding an embedding of a fixed problem graph into any potential hardware graph. However, this algorithm assumes the size of the problem graph is a constant and uses it exponentially, therefore the result is not expected to yield practical embedding algorithms. Choi notes that a similar problem has been previously studied in parallel computing [Lei14] where a job needs to be distributed over a cluster's nodes, but existing results are incompatible with the requirement of a graph minor embedding [Cho11].

In addition to constructing a minor embedding, in practice we want to *tune* the embedding with beneficial graph properties. Finding an embedding with a minimum hardware footprint, measured in qubits, would be preferable to more wasteful embeddings. Experimental evidence also suggests that large vertex sets lead to poor solutions in practice, so minimizing the diameter of each vertex set's induced subgraph is desirable. Thus, in addition to the NP-hard problem of generating a single embedding, we are also interested in searching over the space of embeddings.

Examples of prior application-to-Ising-Model compilations include Lucas's formulation of Karp's 21 NP-hard problems [Luc14], NASA's rover missions [Rie15; Ven15], applications in computational chemistry [Kas11], and computer vision [Nev08].

3.2.2 Related Work

The notion of minor embedding QUBO problems into Chimera $\mathscr{C}_{L,M,N}$ hardware was first introduced by Choi in 2008 [Cho08]. Choi later provided the first general purpose embedding algorithm [Cho11], TRIAD, which embedded K_{LN} (assuming $N \leq M$) into a triangular portion of the D-Wave hardware. This embedding trivially provides embeddings for all graphs of at least LN vertices; however, no tuning mechanism is provided to reduce the hardware footprint for problems with less edges.

Klymko et al. [Kly14] extended this work by providing an alternative embedding algorithm for K_{LN+1}. While TRIAD could be extended for this extra vertex set, it unnaturally used all remaining qubits. The embedding provided by Klymko et al. shifted these qubits around such that all the vertex sets are (roughly) balanced. Klymko et al. showed that this balanced embedding also proved resilient to hardware instances with hard faults (missing qubits). Finally, the authors also introduced the notion of QUBO rejection using structural graph properties. Specifically, Klymko et al. showed that QUBOs with treewidth larger than $L(N+1)$ cannot be embedded in $\mathscr{C}_{L,M,N}$. While treewidth is NP-complete to compute, Wang et al. provided a linear-time approximation for problems based on Ollivier-Ricci curvature [Wan14].

While the algorithms from [Cho11] and [Kly14] ran in constant time and guaranteed an embedding, Cai et al. [Cai14] took a different approach by providing greedy heuristics for embedding arbitrary QUBOs into arbitrary hardware graphs. Experimental results provided by the authors show that, for very sparse graphs such as 3-regular and grid graphs, the algorithm succeeded in embedding larger QUBOs than previous embedding algorithms, while also using less qubits. This so-called *CMR algorithm* is the basis for the embedding algorithm provided in the D-Wave API [D-W16].

Most recently, Boothby et al. [Boo16] generalized the TRIAD embedding into a class of *native clique embeddings* for K_{LN}. They show that, unlike the TRIAD embedding, exponentially-many native clique embeddings exist in a given Chimera graph, making it possible to construct one that avoids hard faults. Additionally, they provide a polynomial-time dynamic programming algorithm for computing the maximum native clique possible in a Chimera graph with faults.

3.3 Virtual Hardware Framework

At the core of our framework is a *virtual hardware layer* created to provide a cleaner interface for finding minor embeddings. The introduction of this intermediary representation splits the minor embedding process into two phases:

1. Find the initial embedding. Starting with a virtual hardware template that allocates physical resources, find a virtual embedding function into the virtual hardware.

2. Iteratively tune the embedding. After obtaining an initial embedding, apply reduction routines to tune both the virtual embedding function and virtual hardware to adjust physical hardware resource usage.

Fig. 3.3 illustrates this iteration. Provided with an initial virtual hardware template $\mathscr{T}$ and its embedding ψ into the physical hardware, a virtual embedding ϕ is sufficient for finding a valid minor embedding of the QUBO into the physical hardware. By iterating reduction subroutines, a sequence of improved embeddings $(\phi, \mathscr{T}, \psi) \to (\phi', \mathscr{T}', \psi') \to (\phi'', \mathscr{T}'', \psi'')$ each produce a full embedding with reduced hardware usage.

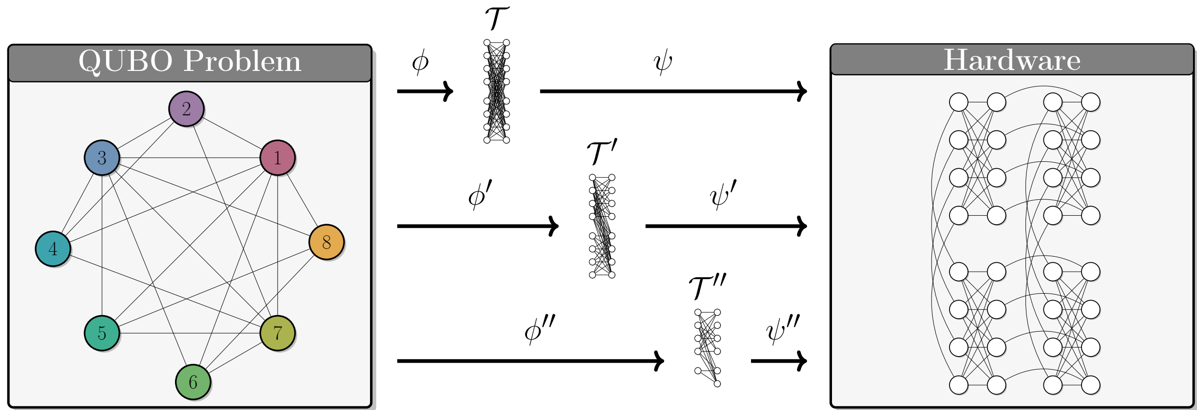


Figure 3.3 A high-level overview of the virtual hardware framework, including the iterative tuning of the virtual embedding ϕ, virtual hardware template $\mathscr{T}$, and the physical embedding ψ.

Formally, we assume the problem is formulated as a graph P and the hardware layout as a graph H. The *virtual hardware template* $\mathscr{T}$ is a graph embeddable into H. This embedding denotes an allocation of qubits in H into virtual qubits in $\mathscr{T}$, encoded with a *physical embedding function* $\psi : V(\mathscr{T}) \to \mathscr{P}(V(P))$. For bookkeeping, we require that each edge in $\mathscr{T}$ represents exactly one edge in H – therefore removing edges in the virtual hardware has a corresponding meaning on the physical hardware footprint. Since we want to define a virtual hardware template that scales with the physical hardware, we define virtual hardware templates in terms of *families*:

Definition (Chimera-Compatible Virtual Hardware Template Family). *A virtual hardware template family $\mathscr{F}$ is a set of virtual hardware graphs defined with a corresponding family of physical hardware embedding functions Ψ, such that for all $L, M, N \in \mathbb{Z}^+$, there exists $\psi \in \Psi$ and $\mathscr{T} \in \mathscr{F}$ such that ψ minor embeds $\mathscr{T}$ into $\mathscr{C}_{L,M,N}$.*

Finding a *virtual embedding function* $\phi : V(P) \to \mathscr{P}(V(\mathscr{T}))$ is sufficient for finding the initial

embedding $\chi : V(P) \to \mathscr{P}(V(H))$, which can be constructed by letting $\chi(u) = \bigcup_{x\in\phi(u)} \psi(x)$. We compute this virtual embedding function with an *embedding subroutine*:

Definition (Embedding Subroutine)**.** *An embedding subroutine takes as input a problem graph P and virtual hardware $\mathscr{T}$, and outputs a virtual embedding $\phi : V(P) \to \mathscr{P}(V(\mathscr{T}))$ or `FAIL`.*

After finding a full minor embedding function, we then apply *reduction subroutines* to produce tuned embeddings:

Definition (Reduction Subroutine)**.** *A reduction subroutine takes as input a problem P, virtual hardware $\mathscr{T}$, and virtual embedding ϕ, then outputs an updated virtual hardware $\mathscr{T}'$ and virtual embedding ϕ' (potentially identical to $\mathscr{T}$ and ϕ).*

After reduction subroutines are applied, an updated physical embedding function ψ' can be recovered from the original ψ and the final virtual hardware $\mathscr{T}'$ by using only the physical qubits needed to represent the edges in $\mathscr{T}'$. Again, we have a full embedding χ' of the problem into the physical hardware by combining ϕ' and ψ'.

3.3.1 Biclique Virtual Hardware

We now present an implementation of this framework using a *biclique virtual hardware*, an embedding subroutine based on both Choi's TRIAD and Klymko et al.'s embedding algorithm, and provide two reduction subroutines for minimizing the total number of qubits used. We start with the virtual hardware:

Definition (Biclique Virtual Hardware Template Family)**.** *A $\mathscr{C}_{L,M,N}$ hardware contains a biclique $K_{LM,LN}$ virtual hardware $\mathscr{T}$ with partite sets $L(\mathscr{T}) = \{v_1, \dots, v_{LM}\}$ and $R(\mathscr{T}) = \{h_1, \dots, h_{LN}\}$; we refer to these as the vertical and horizontal partite sets, respectively. The embedding function defining the minor embedding is given by*

$$\psi(v_i) = \{(j, \lceil i/L \rceil, 1, i \bmod L) \mid 1 \le j \le M\}, \text{ and}$$
$$\psi(h_i) = \{(\lceil i/L \rceil, j, 2, i \bmod L) \mid 1 \le j \le N\}.$$

The intuition behind this allocation is a partitioning of the edges in Chimera graphs (c.f. Fig. 3.4). There are three such edge types – intra-cell, vertical inter-cell, and horizontal inter-cell – and the inter-cell edges provide the highest connectivity increase per minor contraction. Therefore allocating maximal vertical and horizontal paths provides a virtual hardware with relatively large degree per vertex.

The biclique virtual hardware is fairly robust to physical hardware specifications by not requiring a square Chimera grid like previous algorithms, nor depending on the fact that $L = 4$

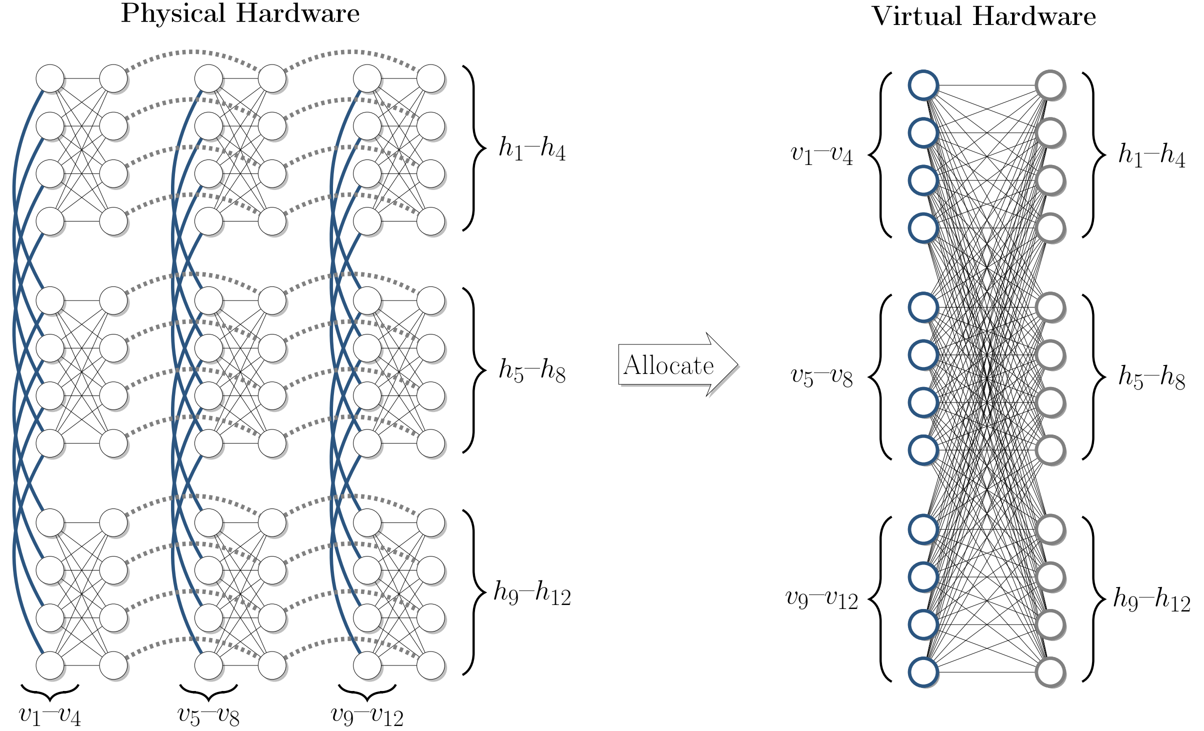


Figure 3.4 The $K_{12,12}$ biclique virtual hardware for Chimera(4, 3, 3). Thick blue edges show allocations to vertical vertex sets, and dashed gray edges show the horizontal vertex set allocations.

in existing hardware implementations. A biclique virtual hardware can also be allocated from a hardware implementation with hard faults; however, in the naive allocation we find that each missing qubit removes a full vertical or horizontal path. Managing hardware implementations with hard faults is less a concern than in prior work, with more mature hardware yields and the introduction of software post-processing methods for emulating missing qubits (e.g. the Full-Yield Chimera Solver provided in D-Wave SAPI 2.4 [D-W16]).

3.3.2 Biclique Embedding and Reduction Subroutines

In this subsection we develop a baseline set of embedding and reduction subroutines utilizing the biclique virtual hardware template. We start by providing an embedding for a complete graph on $\min(LM, LN)$ vertices. At a high level the embedding assignment is straightforward: a single virtual qubit in the vertical partite has edges to every virtual qubit in the horizontal partite, and vice-versa. Therefore, to ensure that every two problem vertices are joined by an edge, we map each problem vertex to a pair of virtual qubits:

Subroutine 1 (Native-Embed)**.** *Given a problem graph P with $V(P) = \{u_1, \ldots, u_n\}$ where*

$n \leq \min(LM, LN)$ and a biclique virtual hardware $\mathscr{T}$ with partites $L(\mathscr{T}) = \{v_1, \ldots, v_{LM}\}$ and $R(\mathscr{T}) = \{h_1, \ldots, h_{LN}\}$, Native-Embed produces an embedding ϕ by mapping $\phi(u_i) = \{v_i, h_i\}$ for $1 \leq i \leq n$.

As defined, Native-Embed redundantly has two edges between every pair of vertex sets $\phi(u_i)$ and $\phi(u_j)$, for $i \neq j$; namely, the edges (v_i, h_j) and (v_j, h_i). Recall that we defined ψ such that each edge in $\mathscr{T}$ represents a unique edge in H, so this redundancy in the virtual hardware represents an actual redundancy in the physical embedding. To gauge the wastefulness, we *score* a virtual hardware and its virtual embedding:

Subroutine 2 (Qubit-Scoring)**.** *Suppose we are given standard input P, $\mathscr{T}$, and ϕ. For each virtual qubit $v_i \in L(\mathscr{T})$, let $I_{v_i} = \{j \mid (v_i, h_j) \in E(\mathscr{T})\}$ be its* index set *– the range of neighbors it has on the virtual hardware. Define the score for each left partite vertex as*

$$\mathsf{score}(v_i) = 1 + \left\lfloor \frac{max(I_{v_i})}{L} \right\rfloor - \left\lfloor \frac{min(I_{v_i})}{L} \right\rfloor .$$

Each h_i is assigned an index set and score analogously. Then the qubit score for ϕ and $\mathscr{T}$ is

$$\sum_{v_i \in L(\mathscr{T})} \mathsf{score}(v_i) + \sum_{h_i \in R(\mathscr{T})} \mathsf{score}(h_i).$$

At a high level, Qubit-Scoring computes the number of physical qubits that must be used with the current virtual hardware and virtual embedding. If removing a redundant edge reduces the score, then we have also reduced physical hardware usage. If removing a redundant edge does *not* affect the score, then we know that this particular edge is not requiring extra hardware usage by itself; however, a sequence of non-score-reducing redundant edge removals could potentially reduce the score. Therefore, it is non-trivial to identify which of the redundant edges should be removed for optimal hardware resource minimization.

Based on this observation, we provide two evaluation methods for computing virtual hardware minimization. First, Qubit-Evaluation computes all possible redundant edge removals and chooses the one with minimum score; this calculation is exponential in the number of redundant edges. A faster evaluation method Fast-Qubit-Evaluation greedily keeps the lexicographically-first edge, providing a *minimal* (but not necessarily minimum) score in linear time.

Subroutine 3 (Qubit-Evaluation)**.** *Suppose we are given standard input P, $\mathscr{T}$, and ϕ. Let S be the set of problem vertices mapped to at least one virtual qubit on each partite, let $\mathscr{E}$ be the set of all edge sets E on the virtual hardware such that for each $u, v \in S$, there is exactly one edge $(u', v') \in E$ with $u' \in \phi(u)$ and $v' \in \phi(v)$. Then Qubit-Evaluation returns $\arg\min_{E \in \mathscr{E}}$ Qubit-Scoring(E).*

Subroutine 4 (Fast-Qubit-Evaluation)**.** *Suppose we are given standard input P, $\mathscr{T}$, and ϕ. Let S be*

the set of problem vertices mapped to at least one virtual qubit on each partite. Then Fast-Qubit-Evaluation returns $E = \{(v_i, h_j) \mid i \leq j,\ v_i \in \phi(x),\ h_j \in \phi(y)$ *for* $x, y \in V(P)$ *and* $y \in N(x)\}$*.*

The last step is to use this reduced edge set to construct a reduced virtual hardware, computed using Qubit-Reduce:

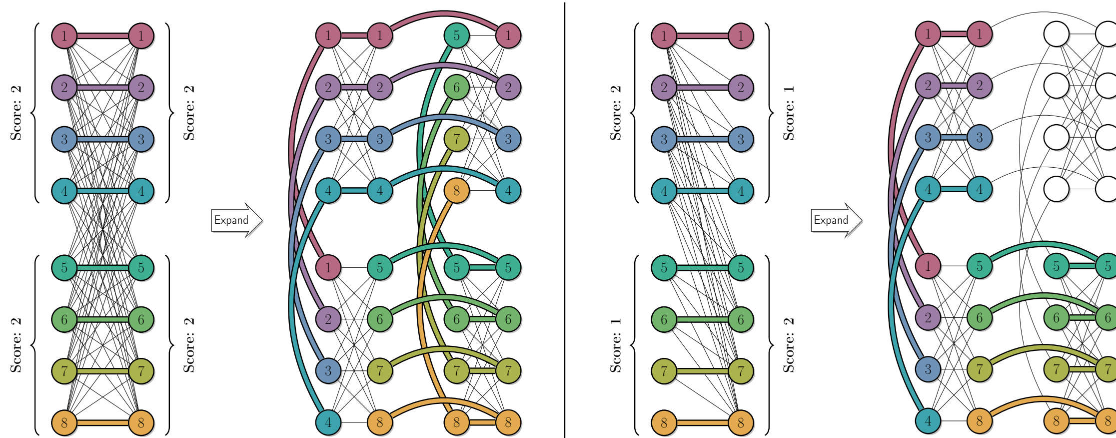


Figure 3.5 (Left) The Native-Embed embedding with "+"-shaped vertex sets; (Right) The embedding reduced by Qubit-Reduce, with "L"-shaped vertex sets.

Subroutine 5 (Qubit-Reduce)**.** *Given standard input* P*,* $\mathscr{T}$*,* ϕ *and an evaluation subroutine, Qubit-Reduce computes a set* E *of redundant edges to be removed, and outputs the current virtual embedding* ϕ *and a new virtual hardware* $\mathscr{T}'$ *with vertices* $V(\mathscr{T})$ *and edges* $E(\mathscr{T}) - E$*.*

While fairly simple, Qubit-Reduce has the potential to reduce qubit usage by 50%. This ratio occurs when Native-Embed's "+"-shaped vertex sets on the physical hardware are reduced to "L"-shaped vertex sets (as described by Boothby et al. [Boo16]). Fig. 3.5 visualizes this reduction.

Up to this point, we have implicitly assumed that the problem graph was complete (i.e. we needed to enforce every edge). However, we can achieve further hardware resource reduction by assuming that the problem is missing edges. Specifically, by shuffling the assignment of vertex sets on the biclique virtual hardware, we can group together those vertices with edges between them, resulting in shorter vertex sets. This computation can be done with a scheme of subroutines, kExchange-Reduce. In local search terminology, we compute a deterministic gradient descent on the k-exchange neighborhood without restarts.

Subroutine 6 (kExchange-Reduce)**.** *Given standard input* $P, \mathscr{T}, \phi$*, and neighborhood exchange parameter* $k \geq 2$*, the subroutine* k*Exchange-Reduce computes a new virtual embedding* ϕ' *with the following steps:*

1. *Let $\phi' = \phi$.*
2. *Starting from ϕ', compute all $\binom{n}{k}k!$ ways to reassign exactly k problem vertices in each partite, and score each qubit reassignment. (For example, if $\phi(u_1) = \{v_1, h_1\}$ and $\phi(u_2) = \{v_2, h_2\}$, then their 2-exchange on the left partite is $\phi(u_1) = \{v_2, h_1\}$ and $\phi(u_2) = \{v_1, h_2\}$).*
3. *Let ϕ' be the reassignment with the lowest score.*
4. *Repeat until no k-exchange leads to a score reduction, and return ϕ' and $\mathscr{T}$.*

For a fixed k, run time for kExchange-Reduce is $\binom{n}{k}k! = O(n^k)$ per iteration with a maximum of $L^2(M+N)$ iterations. With the standard assumptions that L is a constant and $\sqrt{n} = \max(M, N)$, kExchange-Reduce has a run time of $O(n^{k+1/2})$.

3.3.3 Emulation and Enhancement

Applying the tools introduced in the last subsection, we can emulate Choi's K_{LN} TRIAD algorithm [Cho11] with Native-Embed and Qubit-Reduce. Klymko et al.'s K_{LN+1} embedding [Kly14] can be found by tweaking Native-Embed to embed $u_1, \ldots, u_{LN-1}$ as usual, but also setting $\phi(u_{LN}) = \{v_{LN}\}$ and $\phi(u_{LN+1}) = \{h_{LN}\}$. Note that doing so forces the first $LN - 1$ vertex sets to cover both the vertical and horizontal partites in order to be adjacent to the last two vertices, therefore applying Qubit-Reduce is limited and not recommended for general use.

One advantage of emulating existing algorithms in this framework is for the application of virtual hardware-specific reduction subroutines; namely, Qubit-Reduce and kEx-Reduce. In Section 3.5.1 we see that the subroutines do in fact produce smaller embeddings without unreasonably increasing run times.

3.3.4 Summary

In this section we defined a biclique virtual hardware formed naturally from the Chimera graph by exploiting its grid-like structure and high intra-cell connectivity.

We also defined a full baseline stack of embedding and reduction subroutines. As noted in the last subsection, this framework is sufficient for emulating the best existing algorithms for dense problem graphs in hardware layouts without faults. Furthermore, we can apply additional reduction routines to achieve reduced embedding footprints. In total, these results serve as a full proof-of-concept motivating the use of virtual hardware and the development of specialized and modular subroutines. In the next section we take the next step and move beyond existing embedding algorithms by exploiting the bipartite structure in problem graphs to tackle larger, more sparse problems.

3.4 Utilizing Bipartite Problem Structure

In the last section we emphasized the structural properties of the Chimera hardware graph, deriving the biclique virtual hardware and its subroutines. In this section we utilize bipartite structure from the problem graph. Specifically, we use the notion of *odd cycle transversals* to decompose problem graphs and extract a maximal bipartite induced subgraph. We start by defining the odd cycle transversal and its limitations in the Chimera graph, then describe an initial embedding subroutine OCT-Embed, and finally propose a faster heuristic, Fast-OCT-Embed.

3.4.1 Odd Cycle Transversal

One metric for gauging the "bipartite-ness" of a graph G is the smallest set of vertices preventing G from being bipartite, a minimum odd cycle transversal:

Definition (Odd Cycle Transversal (OCT))**.** *An odd cycle transversal of a graph G is a set of vertices S such that $G \setminus S$ is a bipartite graph. We denote the size of a minimum OCT set as the OCT number, $OCT(G)$, and the problem of computing $OCT(G)$ as* MinOCT.

Unfortunately, MinOCT is NP-hard [Lew80] and does not have a constant factor approximation algorithm unless P = NP [Lun93]. However, the problem is fixed-parameter tractable (FPT) when parameterized by the natural parameter (solution size). In other words, graphs with small OCT numbers will also have quickly-computable OCT numbers, regardless of total graph size. Given that the biclique virtual hardware is most efficiently utilized when embedding problem graphs with small OCT numbers, we expect embeddable problem graphs will have an efficiently computable OCT decomposition. As a baseline we use Reed et al.'s $O(3^k kmn)$ algorithm for computing solutions of size k, which is known to have several simplifications and optimizations [Lok09; Hüf05]. Other algorithms for specialized instances also exist [Aga05a; Lok12].

We note that MinOCT and the problem of computing the size of the maximum bipartite induced subgraph (denoted by MaxBipartite) are complements, in the sense that an exact solution to one problem also provides a solution to the other. However, an *approximation* for one problem is not an approximation for the other, so some care must be taken when choosing which problem to approximate.

3.4.2 OCT and the Chimera Graph

In prior work, Klymko et al. showed that the Chimera graph has treewidth bounded by $O(LN)$, assuming $N \leq M$ [Kly14]. In this section we show that the maximum $OCT(G)$ over all Chimera-embeddable graphs G is bounded by all three Chimera parameters. First, we note that treewidth and OCT describe different graph structure:

Proposition 1. *The treewidth of a graph is independent of its OCT number.*

Proof. Consider two families of graphs:

1. The class of grid graphs. These graphs are known to have treewidth proportional to the smallest grid dimension [Die05], but have an OCT number of 0 since they are bipartite.

2. The class of trees with their leaves replaced with triangles. These graphs have treewidth at most three (a tree decomposition exists where each bag contains at most a triangle and its neighbor in the tree), but unbounded OCT number since each (disjoint) triangle contains at least one OCT vertex.

We have shown that one property cannot bound the other, therefore they are independent. □

With this independence established, we proceed to show upper and lower bounds on the maximum OCT number of a Chimera-embeddable graph.

Lemma 1. $OCT(G) \leq \min(|L(B)|, |R(B)|)$ *for all minors G of a bipartite graph B with vertex partite sets $L(B)$ and $R(B)$.*

Proof. Let ϕ be a minor embedding of G into B, and without loss of generality, let $L(B)$ be the smaller of the two partite sets. Let $S = \{x \mid x \in V(G) \text{ and } u \in \phi(x) \text{ for } u \in L(B)\}$, then we know that $|S| \leq |L(B)|$. $V(G) - S$ is necessarily bipartite since $\phi(x)$ is composed of vertices from $R(B)$ for $x \in V(G) - S$, therefore $OCT(G) \leq |S| \leq |L(B)|$.

□

Corollary 1. $OCT(G) \leq LMN$ *for all Chimera-embeddable graphs G.*

Lemma 2. *There exists a Chimera-embeddable graph G such that $OCT(G) \geq (L-1)MN$.*

Proof. We construct G by contracting $L-1$ vertex-disjoint edges in each cell of a Chimera graph. Each cell is now a K_{L+1} clique and $L-1$ of these vertices must be included in an OCT set, therefore $OCT(G) \geq (L-1)MN$.

□

While the treewidth of Chimera graphs only grows in two dimensions (L and $\min(M, N)$), Lemma 2 shows that the minimum odd cycle transversal will increase if L, M, or N is increased. Therefore we recommend using a *minimal* odd cycle transversal as a proxy for estimating how much hardware a problem graph's embedding will require. A minimal OCT set is fast to compute and reflects more of the actual hardware usage than treewidth.

In addition to gauging how much hardware a problem's embedding will require, we can also use the minimum odd cycle transversal to recognize when certain problems are *not* embeddable.

The Klymko et al. [Kly14] result shows that problems with treewidth larger than $(L+1)N$ cannot be embedded, but this bound does not apply to classes of small treewidth, such as series-parallel graphs [Epp92]. However, the OCT rejection criterion provides a characterization of unembeddable graphs in terms of odd cycles, therefore it encompasses a different class of graphs (including series-parallel graphs, which can have an unbounded number of odd cycles).

3.4.3 Computing OCT and OCT-Embed

As mentioned previously, the fastest-known algorithm for computing the OCT number is exponential in the solution size, so we want to prune graphs if possible. One method of doing that is by removing tree-like structure:

Proposition 2. *To compute* MinOCT *on a graph G, it is sufficient to compute* MinOCT *on the maximal 2-edge-connected subgraphs of G.*

Proof. We induct on the number of 2-edge-connected maximal subgraphs. If there is no such subgraph, then every edge is a bridge and the graph is a tree, therefore the claim is true.

Suppose instead that there are k such subgraphs in G and the claim is true for all graphs with $k-1$ such subgraphs. We can decompose the graph into maximal 2-edge-connected subgraphs by computing a chain decomposition [Sch13] on G to identify its *bridges*. Removing these bridges produces each maximal 2-edge-connected subgraph as a connected component. Further, contracting these subgraphs creates a tree with the contracted subgraphs as vertices and the bridges as the edges. Pick a subgraph S that is a leaf on this tree, and let (v_1, v_2) be the bridge separating S from $G \setminus S$. By the induction hypothesis, we can compute $OCT(G \setminus S)$ and $OCT(S)$ on their maximal 2-edge-connected subgraphs, therefore all that remains is to show that these two partial solutions are compatible.

Suppose that the partial solutions are expressed as a coloring: vertices in the left partite set are colored L, the right partite set R, and neither partite set (e.g. in the OCT set) as N. If at least one of v_1, v_2 is colored N, or if one is colored L and the other R, then these partial solutions are compatible as-is. Suppose to the contrary that both are colored with the same partite set color. Then in S we recolor $L \to R$ and $R \to L$. This recoloring does not change $OCT(S)$, and the partial solutions are now compatible.

□

This preprocessing step is fast, costing only an additive $O(m)$ run time when using Schmidt's chain decomposition algorithm [Sch13]. This approach also provides an opportunity for parallelization if the graph has many 2-edge-connected maximal subgraphs. While this technique applies to any graph, we can take advantage of the 2-edge-connectivity in the class of series-parallel graphs by exploiting *nested ear decompositions*:

Proposition 3. *$OCT(G)$ can be computed in linear time for a series-parallel graph G.*

Proof. The proof of Proposition 3 can be found in Appendix 3.7. □

We conclude this subsection by defining an embedding subroutine that uses an OCT-decomposition to embed into the biclique virtual hardware. At a high level, OCT-Embed first computes a minimum OCT set, embeds the OCT vertices as if they were a complete graph, and then embeds the bipartite induced subgraph directly into the biclique virtual hardware (Fig. 3.6).

Subroutine 7 (OCT-Embed)**.** *Let P be a problem graph with $V(P) = S \cup L \cup R$, where $S = \{u_1, \dots, u_i\}$ is a minimum OCT set, and $L = \{u_{i+1}, \dots, u_j\}$ and $R = \{u_{j+1}, \dots, u_n\}$ are a maximum bipartite induced subgraph. Let $\mathscr{T}$ be a biclique virtual hardware with partites $L(\mathscr{T}) = \{v_1, \dots, v_{LM}\}$ and $R(\mathscr{T}) = \{h_1, \dots, h_{LN}\}$. If $j \leq LM$ and $n - i \leq LN$, then OCT-Embed produces an embedding ϕ by mapping:*

$$\phi(u_x) = \begin{cases} \{v_x, h_x\} \text{ if } u_x \in S \\ \{v_x\} \text{ if } u_x \in L \\ \{h_{x-i}\} \text{ if } u_x \in R \end{cases}$$

otherwise it outputs `FAIL`.

3.4.4 Approximating OCT and Fast-OCT-Embed

A downside to OCT-Embed is its exponential run time, restricting the subroutine's real-world applicability. However, an exact solution to MinOCT is not always required for a full embedding – any odd cycle transversal decomposition will work as long as it fits into the biclique virtual hardware. We utilize this fact to develop an approximation algorithm for MaxBipartite, and use this approximation algorithm for two purposes: (1) as an initial solution for the iterative compression in our algorithm for OCT-Embed, and (2) as a standalone embedding subroutine Fast-OCT-Embed.

We approximate MaxBipartite instead of MinOCT for two reasons. First, the Reed et al. algorithm [Ree04] we use to compute the exact OCT number uses a technique called *iterative compression*, where a solution of size $k+1$ is compressed to size k over several subgraph iterations. We can reduce the number of these iterations by providing the algorithm with a large initial subgraph with at most k OCT vertices; therefore we have motivation for estimating a maximal bipartite subgraph. Second, if we approximate MaxBipartite, then our worst approximations (in terms of magnitude) are when the graph has a large bipartite graph. However, this implies

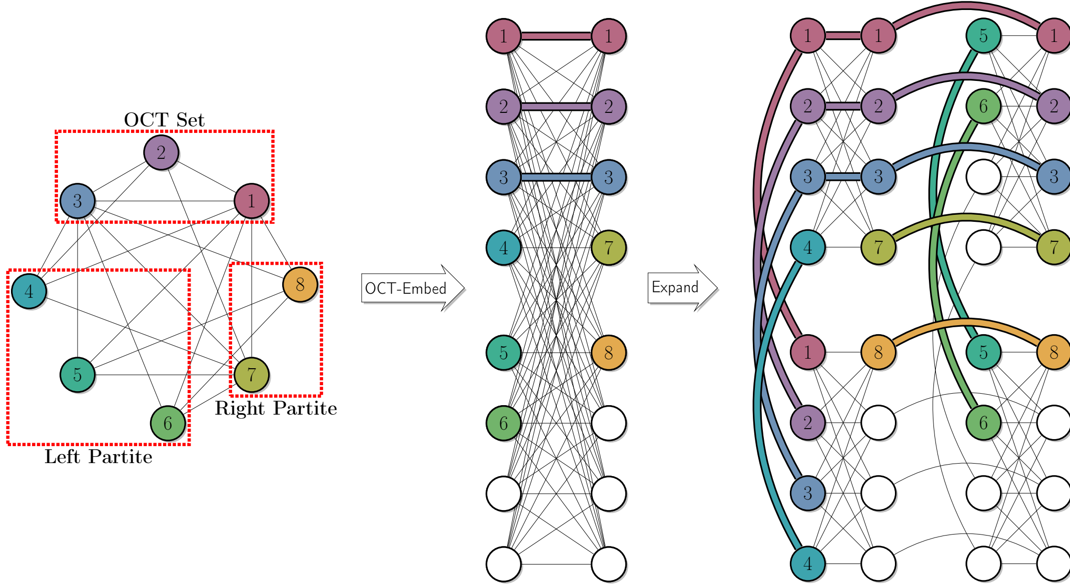


Figure 3.6 Embedding an 8-vertex problem into $\mathscr{C}_{4,2,2}$ using the OCT-Embed subroutine. For figure readability, vertex u_i is labeled with i.

a small OCT set, therefore the exact algorithm will have an exponentially faster run time. Therefore approximating MaxBipartite makes more sense in this context.

Our approximation algorithm is outlined in Algorithm 1. Partially motivated by the success of using a greedy approach to compute exact solutions on series-parallel graphs, we found that a minimum-degree–greedy algorithm also performed well in practice on general graphs (c.f. Section 3.5.1). In total, the algorithm has a run time of $O(m)$ using a modification of Bataglj and Zaveršnik's algorithm for computing k-core decompositions [Bat03].

We begin the approximation factor analysis by noting that an approximation algorithm for maximum independent set translates to MaxBipartite:

Lemma 3. *GreedyBipartite implemented with an α-approximation GreedyIndSet algorithm is an α-approximation algorithm.*

Proof. Let S be a fixed set of vertices such that $G[S]$ is the larger partite of a maximum bipartite induced subgraph. We want to show that for every vertex GreedyBipartite adds to its solution S', at most α vertices from S are not chosen. Let L and R be the first and second independent sets constructed by GreedyIndSet, respectively. First, the set L is chosen without (immediately) disqualifying any vertex in $G \setminus L$ from being in R, so no vertices are disqualified from S' in this step. When constructing R, at most α vertices from S are disqualified for every vertex added to R, by definition of the approximation factor. Therefore R itself is an α-approximation for the partite and a 2α-approximation for MaxBipartite. If $|L| \geq |R|$ then we have shown at least an

Algorithm 1 Greedy Maximal Bipartite Induced Subgraph

```
1:  function GreedyBipartite(G)
2:      L ← GreedyIndSet(G)
3:      R ← GreedyIndSet(G \ L)
4:      return L ∪ R
5:  end function
6:
7:  function GreedyIndSet(G)
8:      S ← ∅
9:      while G not empty do
10:         v ← arg min_{u∈V(G)} d(u)                ▷ Pick any min degree vertex
11:         S ← S ∪ {v}
12:         G ← G \ ({v} ∪ N(v))
13:     end while
14:     return S
15: end function
```

α-approximation. To show the approximation factor still holds when $|L| < |R|$, we want to show that L is a α-approximation for MaxIndSet in $G \setminus R$. But the previous argument still holds, since at most α vertices from S are disqualified from S' for every vertex chosen from $G \setminus R$. Therefore in both cases we have a *alpha*-approximation for the larger partite of a maximum induced bipartite subgraph, therefore we have an α-approximation for MaxBipartite.

□

Corollary 2. *GreedyBipartite is a $\frac{\Delta+2}{3}$-approximation and a $\frac{2\bar{d}+3}{5}$-approximation for graphs with maximum degree Δ and average degree $\bar{d}$.*

Proof. Halldórsson and Radhakrishnan [Hal94] show that GreedyIndSet is a $\frac{\Delta+2}{3}$-approximation and a $\frac{2\bar{d}+3}{5}$-approximation for maximum independent set. By Lemma 3, the same approximation factors hold for GreedyBipartite.

□

Corollary 3. *GreedyBipartite is a d-approximation for d-degenerate graphs.*

Proof. We first want to show that GreedyIndSet is a d-approximation, this proof mirrors that of Lemma 3. Fix a maximum independent set S. In each step of GreedyIndSet, a vertex added to the solution disqualifies at most d vertices from S. Therefore GreedyIndSet is a d-approximation for a maximum independent set, and applying Lemma 3 shows that GreedyBipartite is a d-approximation for MaxBipartite.

□

Up to this point we have not assumed anything about the OCT set when computing an approximation factor. However, as graphs get more dense the OCT set must also grow. We can show this by using *degeneracy* as a metric for density:

Definition (Graph Degeneracy)**.** *The degeneracy of a graph G is the smallest k such that every subgraph of G has a vertex of degree at most k.*

Lemma 4. *GreedyBipartite is a $(n-d)$-approximation for a d-degenerate graph when $d \geq \frac{n}{2}$.*

Proof. When $d \leq \frac{n}{2}$, the desired graph can always be found as a subset of $K_{n/2,n/2}$. However, for larger values of d, vertices must be moved from the bipartite graph into the OCT set, specifically two vertices per additional unit of degeneracy. This fact means that a d-degenerate graph can have at most a bipartite subgraph on $2(n-d)$ vertices. Solving for the approximation factor: $\alpha \cdot 2(n-d) = \frac{2n}{d}$, so $\alpha = \frac{2n}{2d(n-d)} = \frac{n}{n-d} \cdot \frac{1}{d} \geq \frac{n}{n-d} \cdot \frac{1}{n} = \frac{1}{n-d}$.

□

Proposition 4. *Fast-OCT-Embed is a* $\min(d, n-d)$*-approximation for d-degenerate graphs.*

Proof. This result follows directly from Lemmas 3 and 4.

□

In other words, the degeneracy-based approximation factor is best on very sparse and very dense graphs. Swapping the approximation algorithm into our embedding subroutine, we now define Fast-OCT-Embed:

Subroutine 8 (Fast-OCT-Embed)**.** *Let P be a problem graph with $V(P) = S \cup L \cup R$, where $S = \{u_1, \ldots, u_i\}$ is an OCT set, and $L = \{u_{i+1}, \ldots, u_j\}$ and $R = \{u_{j+1}, \ldots, u_n\}$ are a maximum bipartite induced subgraph. Let $\mathscr{T}$ be a biclique virtual hardware with partites $L(\mathscr{T}) = \{v_1, \ldots, v_{LM}\}$ and $R(\mathscr{T}) = \{h_1, \ldots, h_{LN}\}$. If $j \leq LM$ and $n - i \leq LN$, then OCT-Embed produces an embedding ϕ by mapping:*

$$\phi(u_x) = \begin{cases} \{v_x, h_x\} \text{ if } u_x \in S \\ \{v_x\} \text{ if } u_x \in L \\ \{h_{x-i}\} \text{ if } u_x \in R \end{cases}$$

otherwise it outputs `FAIL`.

3.4.5 Summary

In summary, the odd cycle transversal provides a structured method for decomposing problems and embedding them smartly into the Chimera hardware. We showed that OCT is a more flexible property than treewidth in Chimera, increasing flexibility to new generations of hardware, and

also showed how to use OCT to embed into a biclique virtual hardware. In the next section we evaluate these new embedding subroutines against previously studied embedding algorithms.

3.5 Experimental Results

In this section we experimentally evaluate virtual hardware against the existing benchmark algorithms. First, we compare the approximation Fast-OCT-Embed against the exact OCT-Embed, using no reduction routines. We then compare the Reduced Fast-OCT-Embed against Cai et al.'s Dijkstra-based heuristic (denoted here as CMR (Dijkstra)). Finally we conclude with a comparison again Choi's TRIAD algorithm for embedding complete graphs. Against both benchmarks we find that Reduced Fast-OCT-Embed finds embeddings for larger graphs, using less qubits, with fast run times (less than a second).

To minimize bias in the cross-algorithm comparisons, all algorithms and subroutines (e.g. breadth-first search, Dijkstra's algorithm, etc.) were implemented manually in C++ and are available at `https://github.com/TheoryInPractice/aqc-virtual-embedding`.

OCT-Embed is implemented using Lokshtanov et al.'s simplification of Reed et al.'s iterative compression algorithm [Lok09; Ree04]. Fast-OCT-Embed is computed using the smallest OCT number found with $10,000$ runs of GreedyBipartite; run times reported include the total run time to collect this distribution. Reduced Fast-OCT-Embed additionally applies Qubit-Reduce and 2Ex-Reduce using Fast-Qubit-Scoring.

We implemented the CMR (Dijkstra) algorithm from the Dijkstra-based pseudocode provided on page 7 of [Cai14]. Since this heuristic does not necessarily produce an embedding if it exists, we run the heuristic repeatedly until an embedding is found or the time cutoff is reached; this provides the expected time to find an embedding. TRIAD is implemented with Choi's deterministic algorithm, and Reduced TRIAD uses the biclique virtual hardware with Qubit-Reduce and 2Ex-Reduce using Fast-Qubit-Scoring.

To provide a broad spectrum of comparisons, we generated problem graphs using four random graph generators at three density levels (Table 3.1). While previous algorithms such as CMR have been tested on problem graphs with constant vertex degree (e.g. grid and 3-regular graphs), this assumption is unrealistic for real-world QUBOs. Intuitively, the complexity of the problem should scale with the number of variables included. As an example, we note that Beasley's QUBOs [Bea90] have average vertex degree of approximately $\frac{n}{20}$ for n vertex problems.

We define the random graph models as follows. Noisy bipartite graphs were generated by splitting the vertices evenly (up to parity) into two partite sets, including a bipartite edge at probability p, and including a non-bipartite edge at probability $\frac{p}{5}$. The GNP graphs (also known as Erdős-Rényi [Erd60]) are generated by flipping a coin for each possible edge and including it with probability p. The regular graph generator samples from the space of graphs where

each vertex has degree exactly k. Barabási-Albert graphs [Alb02] are generated by iteratively attaching $n - k$ vertices to a subgraph of k vertices using preferential attachment; we generate the initial subgraph using GNP with $p = 0.25$. All graphs are generated using the NetworkX implementations [Hag08], excepting Barabási-Albert, which required a modification to generate the initial subgraph as specified above.

Table 3.1 Definition of density levels for the random input graph generators.

Graph Family	Low Density	Medium Density	High Density
Noisy Bipartite	$p = 0.25$	$p = 0.50$	$p = 0.75$
GNP	$p = 0.25$	$p = 0.50$	$p = 0.75$
Regular	$k = 0.25n$	$k = 0.50n$	$k = 0.75n$
Barabasi-Albert	$k = 0.25n$	$k = 0.50n$	$k = 0.75n$

All experiments were run on a workstation running Fedora 24, and were each allocated a core on an Intel X5675 processor and 1GB of RAM. Run times were limited to 60 minutes using the `timeout -k 10s 60m` command, and no algorithm used more than its allocated memory. The C++ code was compiled with g++ 5.3.1 at the `-O2` optimization level, and controlled with wrapper scripts run with Python 2.7.11. All experiments were seeded using the number of seeds specified in each experiment below. The data points plotted are the median over all problem graph instances and seeded algorithm runs.

3.5.1 Experimental Results

Comparing OCT-Embed and Fast-OCT-Embed on 25 graph instances per n value and 10 seeded algorithm runs, we find that Fast-OCT-Embed practically matches the solution quality of the exact algorithm, while running in under a second. We report a representative sample in Fig. 3.7.

To maintain a reasonable run time while maintaining 10 graph instances per n, we reduced the comparison with CMR to 10 seeded algorithm runs; this reduction did not impact the results since both CMR (Dijkstra) and Reduced Fast-OCT-Embed restart automatically as needed. We found that CMR (Dijkstra) could not find smaller embeddings than Fast-OCT-Embed, in addition to having significantly longer run times. While CMR may be competitive on very sparse graphs (e.g. grid graphs), we found that it was not competitive when the problem graph had a linear density. Fig. 3.8 contains a representative sample using GNP.

Our final comparison is against Choi's TRIAD embedding algorithm, the state-of-the-art for embedding highly dense problem graphs in hardware without hard faults. We do not report run times, given that Choi's algorithm is a deterministic assignment stored in a lookup table and the

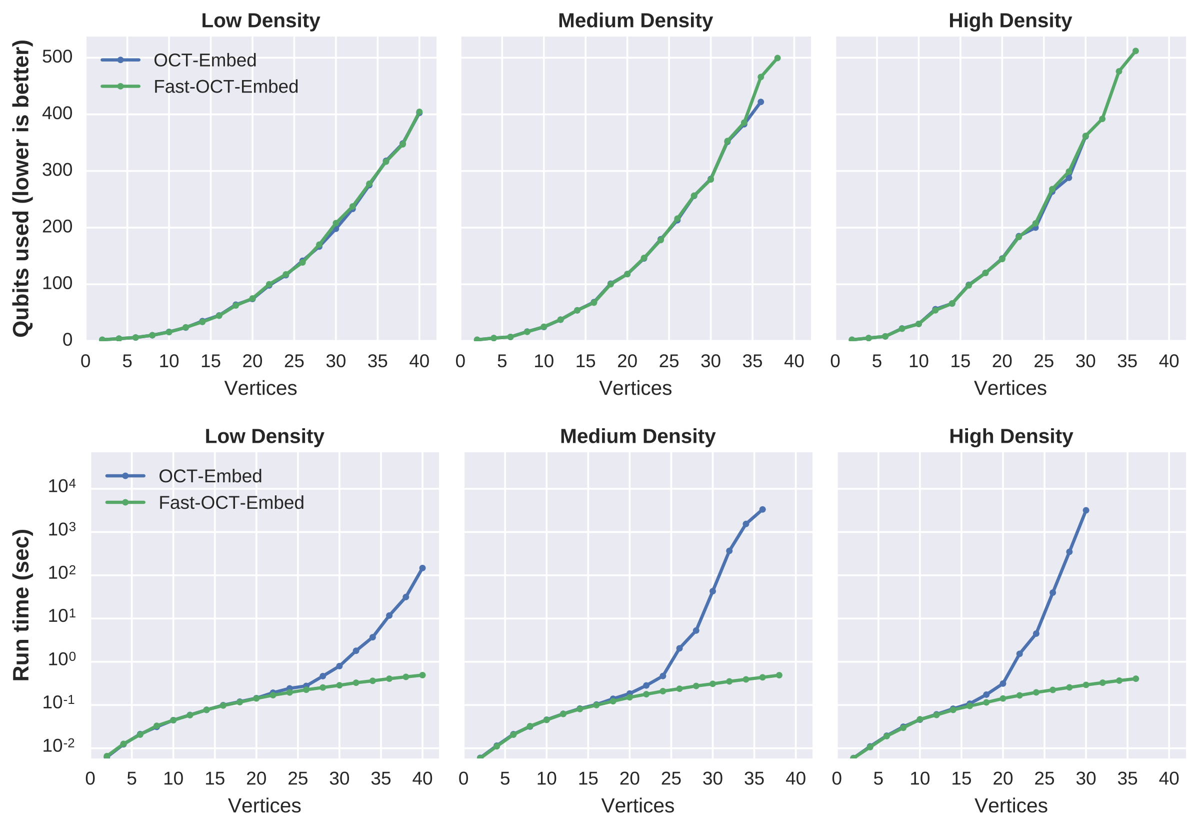


Figure 3.7 Embedding GNP graphs into Chimera(4, 8, 8); data points are the median over 25 random graphs and 10 random algorithm seeds. Experimentally, we observe that the approximation algorithm performs notably better than its approximation factor guarantees, while additionally achieving highly practical run times.

OCT-based algorithm's run times are already reported in the previous plot. For this experiment we again used 25 problem seeds and 10 algorithm seeds. Fig. 3.9 contains a representative sample. Again we find that Reduced Fast-OCT-Embed embeds larger graphs while using less qubits. We also note that Reduced TRIAD was effective compared to stock TRIAD for all low density graphs and some medium density graphs, while only adding less than a second to the run time. Moreover, in several scenarios Reduced TRIAD performed better than vanilla OCT-Embed, given the "L"- vs. "+"-shaped embeddings. However, the flexibility provided with "+"-shaped embeddings made the reduction subroutines much more effective, ultimately producing a better full algorithm. As a best practice, then, we recommend that embedding algorithm designers apply these standard reduction subroutines before evaluating an embedding algorithm's effectiveness.

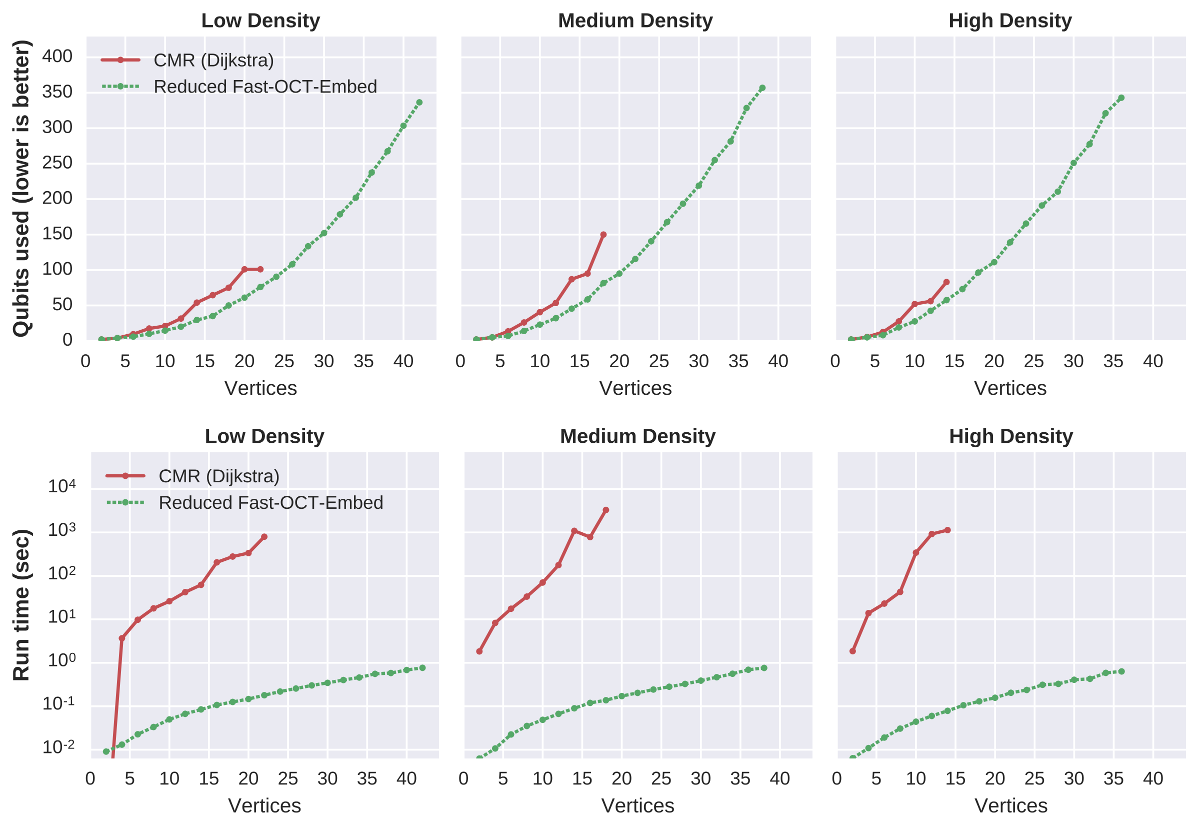


Figure 3.8 Embedding GNP graphs into Chimera(4, 8, 8); data points are the median over 10 random graphs and 10 random algorithm seeds. Reduced Fast-OCT-Embed consistently outperforms CMR in both qubits used and run time.

3.6 Conclusion

We have developed a virtual-hardware–based framework for constructing and deploying optimized techniques for distinct parts of the minor embedding process. By introducing a *biclique virtual hardware*, we provide a cleaner interface for embedding into the Chimera hardware layout and enable modular subroutines for qubit reduction. Exploiting the bipartite structure in problem graphs with odd cycle transversals, we are able to embed problems from from a diverse set of generators and densities. Combining these two methods leads to an embedding algorithm Reduced Fast-OCT-Embed that embeds larger problems, while using less qubits, for reasonably dense problem graphs. Moreover, without any parallelization or system-specific tuning, Reduced Fast-OCT-Embed terminates in the order of seconds. This algorithm sets a baseline for embedding dense problem graphs that should be extended and tuned for the user's application.

Future extensions of this work could include tuned implementation of the reduction methods,

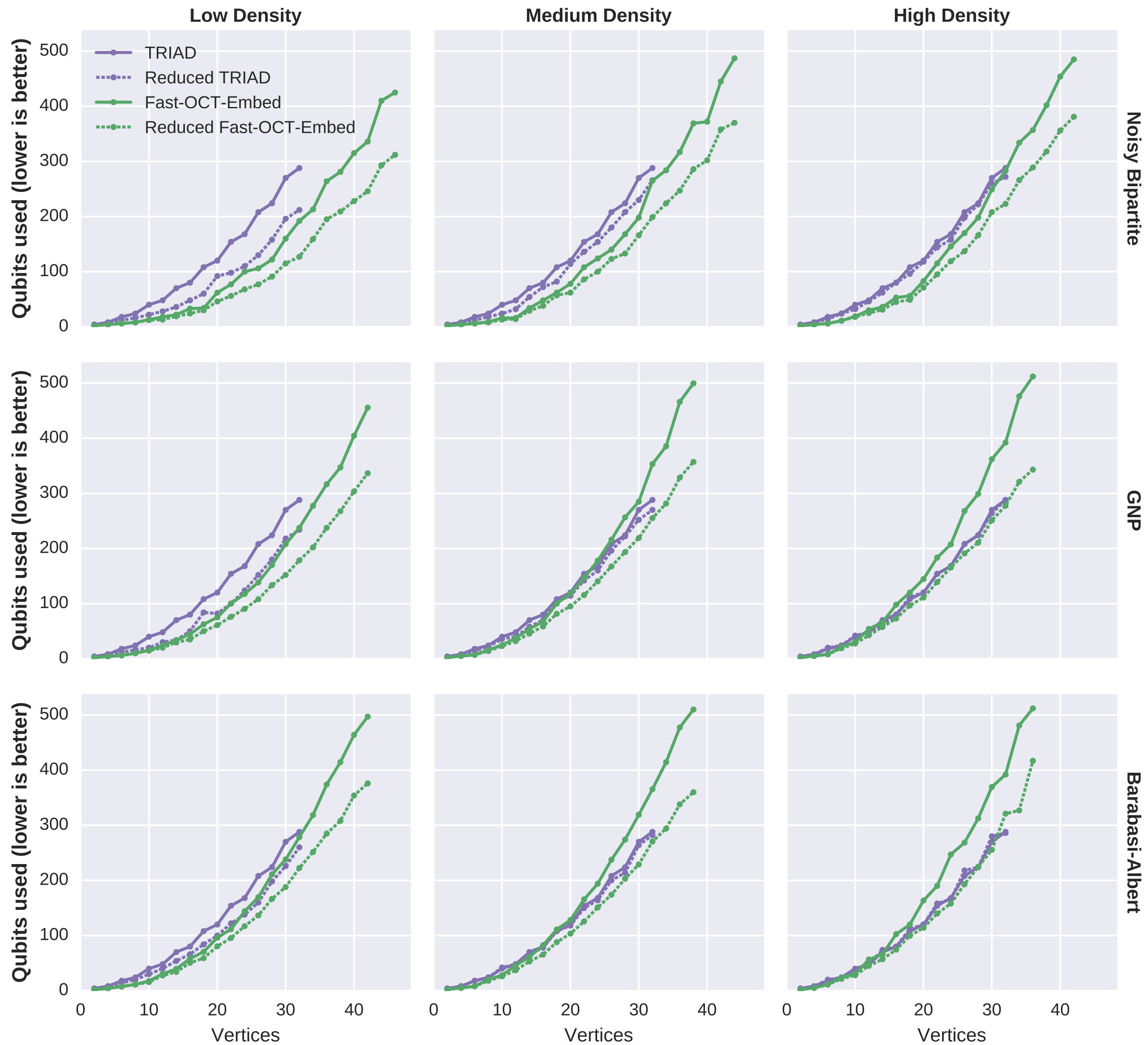


Figure 3.9 Qubits used when embedding into Chimera(4, 8, 8); data points are the median over 25 random graphs and 10 random algorithm seeds. OCT-based algorithms consistently embed larger problem than possible with TRIAD.

which are particularly promising for GPU parallelization. Additionally, as the problem graph becomes highly dense, we see that OCT-Embed (by definition) converges to TRIAD. A more intricate embedding algorithm might not assume the OCT vertices were a clique, allowing even more flexible embeddings. Finally, adapting more intricate embedding algorithms (such as CMR) could provide even better improvements, but would require significant development in the choice of relevant virtual hardware(s).

3.7 Appendix: Computing OCT in Series-Parallel Graphs

In this extended section we prove the following result:

Proposition 5. *$OCT(G)$ can be computed in linear time for a series-parallel graph G.*

The proof is based on the equivalence between series-parallel graphs and graphs with nested ear decompositions. Using this decomposition, we show that a greedy algorithm constructs a minimum OCT set. We start by defining series-parallel graphs and nested ear decompositions:

Definition (Eppstein [Epp92])**.** *A graph G is two-terminal series-parallel with terminals s and t if it can be produced by a sequence of the following operations:*

1. *Base case: Create new graph, consisting of a single edge directed from s to t.*
2. *Parallel composition: Given two-terminal series-parallel graphs X and Y with terminals s_X, t_X, s_Y, and t_Y, form a new graph $G = P(X, Y)$ by identifying $s = s_X = s_Y$ and $t = t_X = t_Y$.*
3. *Series composition: Given two-terminal series-parallel graphs X and Y, with terminals s_X, t_X, s_Y, form a new graph $G = S(X, Y)$ by identifying $s = s_X$, $t_X = s_Y$, and $t = t_Y$.*

Definition (Ear Decomposition (Eppstein [Epp92]))**.** *An ear decomposition of an undirected graph G is defined to be a partition of the edges of G into a sequence of* ears *$(E_1, E_2, \ldots, E_k)$. Each ear is a path in the graph with the following properties:*

1. *If two vertices in the path are the same, they must be two endpoints of the path.*
2. *The two endpoints of each ear E_i, $i > 1$, appear in previous ears E_j and E'_j, with $j < i$ and $j' < i$.*
3. *No interior point of E_i is in E_j for any $j < i$.*

Definition (Nest Intervals (Eppstein [Epp92]))**.** *Given an ear decomposition $(E_1, E_2, \ldots, E_k)$, we say that E_i is* nested *in E_j if both endpoints of E_i are contained in E_j. The* nest interval *of E_i in E_j is the path in E_j between the two endpoints of E_i.*

Definition (Nested Ear Decomposition (Eppstein [Epp92]))**.** *An ear decomposition is* nested *if the following hold:*

1. *For each $i > 1$ there is some $j < i$ such that E_i is nested in E_j.*
2. *If two ears E_i and $E_{i'}$ are both nested in the same ear E_j, then either the nest interval of E_i contains that of $E_{i'}$ or vice versa.*

Eppstein's result shows that these two graph classes are equivalent:

Theorem (Eppstein [Epp92])**.** *Any undirected two-terminal series-parallel graph has a nested ear decomposition starting with a path between the terminals, and any undirected graph with a nested ear decomposition is two-terminal series-parallel with its terminals being the endpoints of the first ear.*

Furthermore, Eppstein shows that these decompositions can be computed in $O(\log^2(n))$ time on a parallel computer, therefore computing the decomposition itself will not bottleneck an OCT-computing algorithm. We now show that given a nested ear decomposition, we can greedily compute a minimum OCT set. First, we define the parity of ears.

Definition (Ear Parity)**.** *We say that an ear E_i is odd if the number of vertices in E_i and its nest interval sum to an odd number. We define an* even ear *analogously.*

Next, we want to show that we can compute the minimum OCT set on a single nest interval correctly.

Lemma 5. *Given an ear decomposition $(E_1, \ldots, E_k)$, let $E = (E_i, \ldots, E_j)$ be an ordered, maximal list of ears contained in a single nest interval. Then the minimum number of OCT vertices contained in these ears is number of maximal in-order sublists of E composed only of odd ears.*

Proof. We proceed by induction on the number of sublists. Suppose that there are zero maximal sublists of odd ears, therefore every ear is even. Then every path from the left-most vertex on the nest interval to the right-most vertex on the nest interval will have the same parity, and we are able to two-color these cycles and the minimum OCT number is zero. Suppose instead that there is one maximal sublist of odd ears, therefore all ears are odd. Removing an endpoint from the inner-most odd cycle renders the remaining edges of this smallest nest interval into bridges that cannot be part of a cycle. This removal also breaks all odd cycles in the maximal nest interval, because any cycle on the remaining ears must use the vertices from two odd cycles of length $2x + 1$ and $2y + 1$, minus the length of the smallest nest interval twice, leaving an odd number of vertices in the cycle. Again we can two-color these and we are done.

Suppose we have an interval with k maximal sublists. If there are any even ears on the outside then we can remove them using the first base case. We now find the inner-most odd ear that is outside of every remaining even ear. Removing an endpoint from this ear renders the outer odd ears bipartite by the second base case. Applying the inductive hypothesis to the earlier ears finds $k - 1$ OCT vertices, therefore we have found a total of k OCT vertices. □

Corollary 4. *We can compute the minimum OCT number of the ears contained in a single maximal nest interval in linear time.*

Proof. In the above proof we visited each ear once. □

Proposition 6. *$OCT(G)$ can be computed in linear time for a series-parallel graph G.*

Proof. We proceed by induction on the number of maximal nest intervals. If there are no nest intervals then we have a single ear and are done, the graph is (by definition) bipartite. Otherwise there is some nest interval. Applying Lemma 5, we can compute a minimum OCT set. Since every nest interval is disjoint, by definition, we can apply the inductive hypothesis to compute the minimum OCT set of the other intervals, visiting each interval exactly once. The number of intervals is bounded by the number of vertices, therefore we compute a minimum OCT set in linear time. □

CHAPTER

4

EDITING TO BIPARTITE

4.1 Introduction

Odd Cycle Transversal (OCT), the problem of deleting vertices to make a graph bipartite, has been well-studied in the theory community over the last two decades. Techniques such as *iterative compression* [Ree04; Hüf09] and *branch-and-reduce* [Lok14; Aki16] have led to significant improvements in both worst-case and experimental run times. These improvements are most drastically seen on the canonical OCT benchmark, Wernicke's Minimum Site Removal dataset [Wer03] (denoted `WH`), where run times have dropped from over 10 hours [Wer03] to under 3 minutes [Hüf09] to under 1 second for multiple state-of-the-art solvers [Aki16]. While these results illustrate the rapid algorithmic advances, they also show that new data is needed for further study.

Recently, a need for practical graph bipartization algorithms has arisen in quantum computing, where the hardware and/or problem structure may naturally have underlying bipartite structure that can be exploited algorithmically[1]. Problem instances from quantum annealing themselves may not be close to bipartite, in contrast to the `WH` data, which was expected to be bipartite barring any read errors. Additionally, "good enough" solutions are of interest in the quantum setting where OCT may be solved as a subroutine in a larger automated compiler, which

[1]For example, the D-Wave Chimera hardware is bipartite, and upcoming Pegasus hardware admits a large complete bipartite graph embedding.

introduces a run time vs. solution quality trade-off previously not considered. Together, we utilize richer data and various timeout scenarios in order to provide a modernized evaluation of OCT methods.

4.1.1 Related Work

Modern theoretical advances on OCT began with the seminal result of Reed, Smith, and Vetta [Ree04], who showed that the problem is fixed-parameter tractable (in terms of the minimum OCT size k) with the technique of *iterative compression*. This algorithm was initially shown to run in time $O(4^k km)$, but improved analyses showed a $O(3^k km)$ run time and simpler algorithms for the compression routines [Hüf09; Lok09]. The next theoretical improvement came from an improved algorithm and analysis for Vertex Cover (VC) and a (straightforward) conversion of an OCT instance to a VC instance. This strategy results in an $O^*(2.3146^{k'})$ algorithm[2], where k' denotes the gap between an optimal solution to Vertex Cover and the solution given by the linear programming (LP) relaxation [Lok14]. Recent work has used the half-integrality of LP-relaxations to reduce the polynomial in n at the cost of a higher parameterized term, resulting in an $O(4^k n)$ algorithm for OCT [Iwa14], and $O^*(4^k)$ [Wah17] and $O(4^k n)$ [Iwa17] algorithms for the more general problem of Non-Monotonic Cycle Transversal. Other algorithmic results for OCT include a $O(\log \sqrt{n})$-approximation algorithm [Aga05b], a randomized algorithm based on matroids [Kra14], and a subexponential algorithm on planar graphs [Lok12].

On the practical side, the first implementation was a branch-and-bound algorithm by Wernicke in 2003 [Wer03] used for studying single nucleotide polymorphisms. A greedy depth-first search heuristic was used to identify upper bounds on OCT, and several sparse graph reduction routines were applied before branching. A Java implementation of this algorithm solved most `WH` instances within a 10 hour timeout. In 2009, Hüffner implemented a refined iterative compression algorithm [Hüf09] with additional pruning for infeasible assignments and symmetry in the compression routine's branching algorithm to achieve experimentally faster run times; all of the `WH` instances could then be solved within three minutes. Hüffner compared this algorithm against an ILP formulation using the GNU Linear Programming Kit (GLPK) [GNU17], which had unfavorable run times. More recently, Akiba and Iwata [Aki16] used a VC-solver based on branch-and-reduce to solve OCT using a standard transformation to VC [Lok14]. The authors reported that their open source Java implementation could solve all `WH` data within a second, while competing implementations based on maximum clique and an ILP formulation solved using CPLEX [IBM17] all finished within three seconds.

[2] $O^*(f(k))$ denotes $O(f(k)n^c)$ for some constant c

4.1.2 Our Contributions

In this work, we collect existing OCT techniques, provide a common Python API for running these algorithms, and evaluate them within a broader experimental envelope by incorporating quantum data and use cases. We also provide a frame for generalizing our conclusions with synthetic data from random graph models.

Whereas OCT can be computed within seconds for all graphs in the previous `WH` dataset [Wer03; Hüf09], we provide a new, significantly more difficult benchmark corpus. Motivated by the widespread usage of Quadratic Unconstrained Binary Optimization Problem (QUBO) problems in quantum annealing research [Nev08], we select QUBO instances from a recent survey [Dun15] that would be of interest to practitioners working on near-term quantum annealers. These datasets are selections from Glover, Kochenberger, and Alidaee [Glo98] (denoted `GKA`) and Beasley [Bea98] (denoted `Beasley`). Not only do these datasets contain significantly harder OCT instances, they also represent a wider array of graph properties such as number of vertices, edge density, degree distribution, etc., than the `WH` benchmark.

Collecting previous code and providing missing implementations, we assemble a unified Python API allowing easy comparison of prior work. Preprocessing routines are taken from the OCT [Wer03] and VC literature [Aki16] and applied to all datasets to harden the benchmark corpus. Heuristics for OCT upper bounds [Wer03; Goo18c] are collected and implemented as a standalone heuristic ensemble solver for stochastically sampling 'good enough' solutions. We use these heuristic solutions, along with a density-first heuristic for the compression ordering, to jump-start the iterative compression algorithm of Hüffner [Hüf09]. These combinatorial algorithms for solving OCT directly are complemented by VC-based [Aki16] and ILP-based [IBM17] solvers.

In quantum-specific experiments, we examine two distinct use cases. First, to represent scenarios where an automated compiler may use OCT as a subroutine and accept heuristic solutions, we evaluate the heuristic ensemble, iterative compression, and the ILP formulation under timeouts of 0.01, 0.1, 1, and 10 seconds. We find that the iterative compression implementation jump-started with heuristic solutions performs best for timeouts less than a second, after which it is worth paying the overhead of using an ILP solver such as CPLEX. In a second use case where an exact solution is required in order to recognize un-embeddable quantum programs, we evaluate iterative compression, VC-based, and ILP-based exact solvers. Here we again find that ILP formulations solved by CPLEX dominate, typically by at least an order of magnitude.

Generalizing these results, we generate synthetic graphs using four random graph generators and evaluate whether "generic" instances matching the density, degree distribution, etc. exhibit similar effectiveness of reduction routines and solver run times. We find that our results on QUBO data are robust, with the same best practice recommendations. This experimental evidence

provides practitioners with a useful reference for developing custom solutions for particular applications.

Our work is fully replicable, with documented code [Goo18b] open sourced under the BSD 3-Clause license. For the interested reader, Appendix A contains implementation details and Appendix B contains extended results.

4.2 Background

We denote a graph $G = (V, E)$. For a set of vertices S, we denote the subgraph induced by deleting S as $G \setminus S$. An edge $(u, v) \in E$ can be *contracted* by adding a new node uv, adding edges from uv to all neighbors of u and v, then deleting u and v. A graph P is a *minor* of a graph H if P can be obtained from H with vertex deletion, edge deletion, and edge contraction. In the context of quantum annealing, we denote the graph P as an *problem graph* and H as a *hardware graph*.

Odd Cycle Transversal (OCT) is formally defined as an optimization problem with natural parameter k.

Odd Cycle Transversal (OCT)

Input:	An input graph $G = (V, E)$.
Problem:	Find $S \subseteq V$ such that $G \setminus S$ is bipartite.
Objective:	Minimize $k := \|S\|$.

OCT is closely related to Vertex Cover (VC).

Vertex Cover (VC)

Input:	An input graph $G = (V, E)$.
Problem:	Find $S \subseteq V$ such that $G \setminus S$ is edgeless.
Objective:	Minimize $\|S\|$.

Specifically, given a graph G, we create an auxiliary graph $G' = (V_L \cup V_R, E_L \cup E_R \cup E')$, where $V_i = \{v_i \mid v \in V\}$ and $E_i = \{(u_i, v_i) \mid (u, v) \in E\}$ for $i \in \{L, R\}$, and $E' = \{(v_L, v_R) \mid v \in V\}$. A solution S' for VC on G' can be mapped to a solution S for OCT on G with $S = \{v \mid v_L \in S \text{ and } v_R \in S\}$; this mapping preserves optimality [Aki16].

Both OCT and VC can also be rewritten as Integer Linear Programming (ILP) instances; as explored further in Section 4.3.5, the choice of OCT $\rightarrow$ ILP formulation has performance implications.

Odd Cycle Transversal (ILP) [Hüf09]

Input: $G = (V, E)$.

$$\begin{aligned}
\text{Minimize} \quad & \sum_{v \in V} c_v \\
\text{s.t.} \quad & s_v + s_u + c_v + c_u \geq 1 && \forall (u,v) \in E \\
& s_v + s_u - c_v - c_u \leq 1 && \forall (u,v) \in E \\
& s_v \in \{0,1\} && \forall v \in V \\
& c_v \in \{0,1\} && \forall v \in V
\end{aligned}$$

Vertex Cover (ILP) [Aki16]

Input: $G = (V, E)$.

$$\begin{aligned}
\text{Minimize} \quad & \sum_{v \in V} x_v \\
\text{s.t.} \quad & x_u + x_v \geq 1 && \forall (u,v) \in E \\
& x_v \in \{0,1\} && \forall v \in V
\end{aligned}$$

An *anytime algorithm* will return the best solution found so far when given a timeout signal. An *exact algorithm* will return a (provably) optimal solution, whereas a *heuristic* is unable to prove optimality. A middle ground between heuristic and exact is an *approximation algorithm*, whose error is bound with a constant. An α-approximation algorithm for a minimization problem has solution size k bounded by $k \leq \alpha \cdot OPT$, where OPT is the optimal solution size; α is the *approximation factor*. We define the approximation factor over a corpus of graph instances as the worst factor over all individual instances in the corpus.

4.2.1 Graph bipartization in quantum annealing

Recent advances in quantum computing has resulted in a class of optimization problem solvers denoted *quantum annealers* (QA). Implementations of QA hardware are based off of the adiabatic quantum annealing (AQC) model, but are typically less powerful (i.e. not universal quantum computers). These QA devices have *hardware topologies*, these graphs define the communication pathways (edges) between qubits (vertices). For example, D-Wave Systems currently produces QA hardware with Chimera hardware, and is preparing for a second generation Pegasus model.

Graph bipartization (OCT) occurs during the compilation step in quantum annealing, when a problem graph[3] must be *embedded* into the hardware graph. This embedding step is equivalent

[3] e.g. a Quadratic Unconstrained Binary Optimization Problem (QUBO)

to finding the problem as a minor of the hardware graph [Cho11]. Researchers have had success running naturally bipartite QUBOs (e.g. deep learning models) on D-Wave Systems annealers [Sch17; Kos88; Boy17; Pot18]. Generalizing these tools to non-bipartite QUBOs is currently of interest to enable additional applications (e.g. Karp's 21 NP-hard problems [Luc14]). The area of automatic embedding tools is under active development (c.f. [Goo18c; Ham17; Ven15]).

We examine two distinct use cases of graph bipartization in quantum program compilation. First, a feasible bipartization of a graph is useful for certain embedding algorithms [Goo18c; Ham17], providing a structure on which to limit the search space of heuristics. In this case, a compiler might require this bipartization in as little as 0.01 seconds, whereas a computer-assisted researcher working in an interactive environment may wait closer to 10 seconds.

Second, OCT can be used to identify when a particular program cannot embed into hardware, as is shown for D-Wave Systems' *Chimera* hardware in [Goo18c]. This scenario requires that the solver return a certified optimal OCT solution, but longer run times are permissible since a hardware owner can compute forbidden configurations once per hardware model. We examine both of these use cases in more detail in Section 4.5.

4.3 Algorithm Overview

In this section, we overview various algorithmic techniques previously applied to OCT. We begin with reduction routines from both the OCT and VC literature, then continue to linear-time heuristics historically used to provide upper bounds for branch-and-bound approaches. The first exact solver we detail is Hüffner's iterative compression solver [Hüf06], which we enhance with heuristics to create an anytime algorithm with increased performance particularly when given small timeouts. Finally, we detail how CPLEX can solve OCT using various reformulations into Integer Linear Programming.

4.3.1 Reduction Routines

We begin with *reduction routines* – rules for simplifying the graph instance such that a solution on the reduced instance is valid for the original instance. Typically these reductions will remove vertices by recognizing configurations that can be simplified deterministically. Reduction routines for OCT come from two sources. Wernicke's branch-and-bound algorithm [Wer03] uses nine reductions directly on the OCT instance. These reductions form roughly three categories: removal of standalone structures (e.g., bipartite components and degree-1 vertices), removal of vertex separators which induce certain bipartite components, and reconfigurations of local structures (e.g., removing a degree-2 vertex in an induced four-cycle). These reductions are most effective on close-to-bipartite graphs, low-connectivity graphs, and sparse graphs, respectively.

A second source of reductions is Akiba and Iwata's VC solver [Aki16], which uses nine reduction routines specific to the VC instance's graph. Based on the conversion between OCT and VC in Section 4.2, a vertex v in the OCT instance must be in the transversal if both v_1 and its mirror v_2 must be in the vertex cover in the VC instance. Similarly, v is labeled bipartite if v_1 or v_2 is excluded from an optimal vertex cover.

We refer the interested reader to the original papers [Wer03; Aki16] for detailed definitions, examples, and complexity analysis. We provide an open source implementation of Wernicke's reductions using Python and NetworkX, and modify a copy of Akiba and Iwata's VC solver [Aki17] to output the graph after a single round of reductions.

4.3.2 Heuristics and Approximations

Heuristics for OCT typically compute a maximal bipartite induced subgraph, then label all remaining vertices as an odd cycle transversal. One strategy for finding a large bipartite subgraph is greedily 2-coloring the vertices using a depth-first search, and adding incompatible vertices to an OCT set as needed [Wer03]; this heuristic has a natural breadth-first search variant. Both of these methods are nondeterministic with respect to the choice of the initial vertex and the order in which neighbors are added to the search queue. Another approach is to find an independent set for the first partite set, then repeat for a second partite set, as in Luby's Algorithm [Lub86]. Recent work showed that by using the minimum-degree heuristic for independent set, this strategy gives a d-approximation in d-degenerate graphs [Goo18c]. Both of these methods are nondeterministic; the former is stochastic by design, and the latter breaks ties between minimum degree vertices. We provide C++ implementations of these four heuristics (DFS, BFS, Luby, and MinDeg), and provide a *heuristic ensemble* solver that runs the heuristics round-robin until a specified timeout is reached.

4.3.3 Iterative Compression

The state-of-the-art implementation for solving OCT combinatorially comes from Hüffner's simplification of the iterative compression algorithm [Hüf09; Ree04]. Broadly, the iterative compression technique starts with a trivial solution on a subgraph of the instance, expands both the solution and subgraph, then applies a compression routine to reduce the solution if possible. By iterating this process, the subgraph eventually encompasses the full graph, and the solution at every step is compressed to remain within some desired bound. The compression routine may have run time exponential in the size of the solution and is applied at most a linear number of times, naturally leading to an FPT algorithm parameterized by the solution size k.

In the specific application to OCT, the compression routine tries all $O(3^k)$ partitions of a $(k+1)$-sized solution into a new transversal and left/right partite sets. For each partition, an

OCT set for the full subgraph is computed by solving a min-cut instance. If the number of vertices assigned to the transversal plus the vertices removed by the cut is less than $k+1$, then the solution was compressed, and otherwise a certificate is found that no solution of size k exists on this subgraph. Using Edmonds-Karp for the min-cut algorithm, this compression routine runs in $O(3^k \cdot k \cdot |E|)$.

The iterating outer loop that expands the solution and subgraph is trivial for OCT: Given an ordering of the vertices, the initial subgraph and solution are the first k vertices, and the subgraph and solution are both expanded by adding the next unused vertex in the ordering to each. In the worst case there are n iterations, resulting in a total run time of $O(3^k \cdot k \cdot |V| \cdot |E|)$.

4.3.4 Modifications to Hüffner's Algorithm

While Hüffner's reformulation of Reed et al.'s algorithm was based on improving the compression routine, we note some straightforward improvements to the outer loop that can also lead to improvements in practice. These improvements are related to the choice of vertex ordering.

First, the number of compressions can be reduced by choosing an initial subgraph larger than the initial solution of size k. Specifically, given a heuristic solution S for OCT, we can construct an initial subgraph of size $\min(|V| - |S| + k, |V|)$ by placing the vertices in $V \setminus S$ first in the ordering, then initializing the subgraph with this bipartite subgraph and up to k of the remaining vertices. This 'bipartite jump-start' has no negative effect on the theoretical run time, but may improve run time by up to a factor of $|V|$ based on the quality of the heuristic solution.

Second, Hüffner's improvements to the compression routine utilize the presence of edges to eliminate possible partitions from consideration. Namely, two vertices cannot be assigned to the same partition if they share an edge, and the number of partitions can be reduced from $O(3^k)$ in the worst case (an independent set) to $O(k^2)$ in the best case (a clique). To exploit this fact, after the ordering is jump-started with a bipartite subgraph, we order the remaining vertices in a reverse degeneracy ordering [Lic70]. This ordering guarantees that vertices added to the subgraph maximize the number of newly introduced edges.

Finally, we note that iterative compression is naturally an anytime algorithm. If, at any point, the iteration stops, the current solution size is a lower bound on the optimal OCT, and the current solution plus the vertices not yet reached in the ordering form an upper bound. By its FPT nature, the iterative compression approach becomes hard when k becomes large. However, iterative compression may fill an important niche between heuristics and exact solvers by offering a structured approach for compressing heuristic solutions as time allows.

We provide a modified copy of Hüffner's implementation with both improvements implemented in C++ and enable anytime functionality when given a termination signal. Based on small-scale experimentation (Appendix **??**), we find that the bipartite jump-start always helps

in expectation, but the degeneracy ordering may not be worth the additional run time when using a small timeout. In our experiments we use only the first improvement on timeouts of 0.01 and 0.1 seconds, and otherwise we use both improvements.

4.3.5 Alternative Solvers

As mentioned in Section 4.2, OCT can be converted into an equivalent Vertex Cover (VC) or Integer Linear Programming (ILP) instance.

Currently, the fastest theoretical run time for OCT and other related problems comes from a VC-formulation [Lok14]. We evaluate this approach with Akiba and Iwata's Java solver, which was previously demonstrated to be faster than Hüffner's iterative compression algorithm [Aki16]. We implement a Python wrapper that converts between OCT and VC, providing a common API with the other solvers.

When solving OCT as an ILP, the instance can be directly converted OCT $\rightarrow$ ILP, or converted with OCT $\rightarrow$ VC $\rightarrow$ ILP. In addition to choosing a formulation, the user must also choose an ILP solver (e.g. CPLEX or GLPK) and consider the effect of additional threads and/or RAM limitations. Testing several configurations (Appendix 4.9.1), we confirmed several best practices from previous work. The biggest factor was choice of solver, where IBM's closed-source CPLEX solver bested the open-source GNU solver GLPK. The next factor with the biggest impact was choice of formulation, where OCT $\rightarrow$ VC $\rightarrow$ ILP performed significantly better than OCT $\rightarrow$ ILP; this performance difference may be explained by a similar result in theoretical analysis [Lok14].

When evaluating the scalability of CPLEX on server hardware, we found that using multiple threads may lead to super-linear speed-up, and that while RAM limitations may increase run time, they do not change the relationship between other factors. In our experiments, we use the OCT $\rightarrow$ VC $\rightarrow$ ILP formulation with the CPLEX solver, a single thread, and unrestricted RAM. Additionally, CPLEX allows recovery of partial solutions, enabling us to use this approach as an anytime algorithm. We again provide a Python wrapper for a common API.

4.4 Data Benchmark and Code

In this section we detail the data used in the experiments, along with the code made available at [Goo18b].

4.4.1 Previous Data

As mentioned in Section 4.1.1, the primary dataset studied in OCT literature originates from Wernicke and is distributed with Hüffner's code. We refer to this data as the `Wernicke-Hüffner`

(WH) dataset. These datasets are originally from genetics through the related Minimum Site Removal problem [Wer03] and are expected to be close to bipartite. This dataset is composed of 45 Afro-American graphs (denoted aa-10, . . . , aa-54) and 29 Japanese graphs (denoted j-10, . . . , j-28). Files aa-12, j-12, and j-27 are provided in Hüffner's code, but are empty and excluded here.

4.4.2 Quantum-Inspired Data

While the WH dataset may have been of historical interest, recent results show that any state-of-the-art solver can solve all these instances within three seconds [Aki16]. To introduce a new benchmark corpus for OCT solvers, we concentrate on domain data from quantum computing. Specifically, a recent survey [Dun15] collected six datasets from the QUBO literature (see Section 6.2); of these, only the Beasley [Bea98] and GKA [Glo98] data have instances small enough to embed in near-term quantum annealing hardware.

In this work we consider the 50-vertex instances (denoted b-50-1, . . . , b-50-10) and the 100-vertex instances (denoted b-100-1, . . . , b-100-10) from the Beasley dataset and the first 35 instances of the GKA dataset (denoted gka-1, . . . , gka-35), which have varying numbers of vertices and densities (c.f., Table 4.1).

Additionally, we examine frustrated cluster loop (FCL) graphs. FCLs were originally introduced as a benchmark for quantum annealing hardware that produced QUBO instances with *rugged* energy landscapes [Kin19]. Rugged landscapes are characterized by having frequent, tall energy barriers, which reduces the effectiveness of classical solvers and makes use of quantum tunneling. The FCL model operates by overlaying cycles from an underlying graph. We use a clique as the underlying graph, meaning that any edge is a valid candidate for each cycle, and we set the number of cycles samples equal to a constant ratio of the order of the clique (c.f. Figure 4.1). Using the D-Wave NetworkX API, we generate a corpus of FCL instances applicable to four hardware models: a D-Wave 2000Q (16×16 Chimera grid), a theoretical D-Wave "8000Q" (32×32 Chimera grid), the upcoming Pegasus-6 (which embeds a $K_{52,52}$) and a Pegasus-12 (which embeds a $K_{124,124}$). For each hardware, we generate FCL instances with more vertices than the clique-embedding can handle, but fewer than the largest embeddable complete bipartite graph (i.e. these are precisely in the regime where OCT-embedding methods would be deployed).

Due to similarity between the graph instances sizes, we further cluster the 2000Q and Pegasus-6 FCL graphs into a fcl-small corpus and the 8000Q and Pegasus-12 into fcl-large.

All QUBO datasets are parsed as undirected, simple graphs with no vertex or edge weights. Vertices, edges with weight zero, and self-loops are excluded.

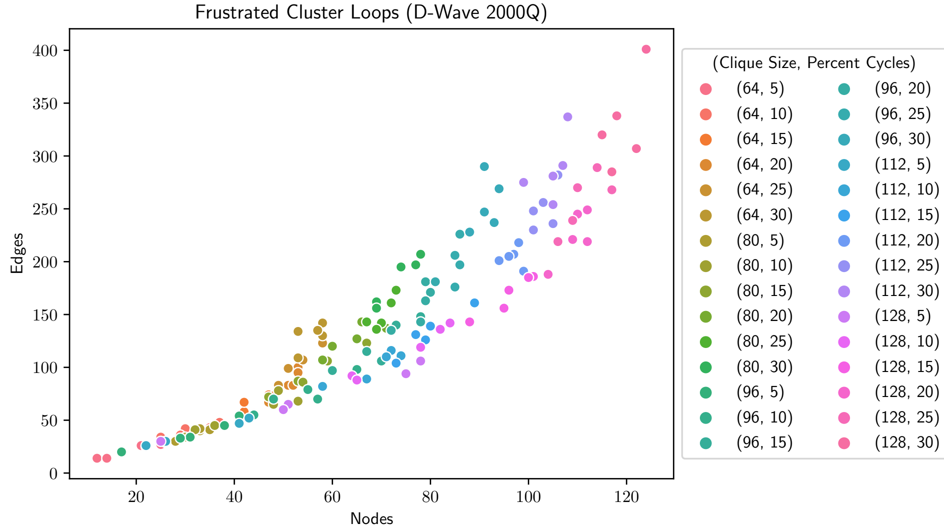


Figure 4.1 Generating frustrated cluster loop (FLC) instances appropriate for the D-Wave 2000Q hardware. This hardware can embed a 64-clique and a 128-biclique, so we range the clique size for the FCL underlying graph $n \in \{64, 96, 128\}$ and scale the number of sampled cycles n/c to fill this range.

4.4.3 Synthetic Graph Generators

To generalize our results and prevent bias that may be present in a difficult QUBO benchmark, we use synthetic graph generators to mimic distinct properties of the quantum graphs.

To match edge density, we use the Erdös-Rényi model [Erd60], which takes as input a number of vertices n and a probability p. Erdös-Rényi generates a graph by initializing n vertices and adding each possible edge with probability p each. By setting $p := |E|/\binom{n}{2}$ we have the same edge density in expectation.

To mimic a dataset's distance-to-bipartite, we provide a Tunable-OCT generator as a modification of Erdös-Rényi. The Tunable-OCT generator requires an upper bound on the optimal OCT solution (denoted n_o) and a bipartite balance parameter $0 \leq b \leq 1$. Tunable-OCT generates a graph by partitioning the vertices into an odd cycle transversal, a left partite set, and a right partite set. The odd cycle transversal has n_o vertices, and the remaining vertices are assigned to the left partite set with probability b. Edges are then generated according to Erdös-Rényi, with the exception that vertices in the same partite set may not share an edge. This construction enforces that $OPT \leq n_o$, highlighting the distinction between arbitrary edge placements and a potentially small optimal OCT solution. In our experimental results we set n_o equal to the optimal solution for the original (non-preprocessed) graphs, and set $b = 0.5$.

For matching degree distribution in addition to density we use the Chung-Lu expected degree model [Chu02]. Given a degree distribution $(d_1, \ldots, d_n)$, the Chung-Lu model adds an edge uv

with probability

$$P_{uv} = \frac{d_u d_v}{\sum_{i=1}^{n} d_i}.$$

We generate these graphs using the original instances' degree distribution.

Finally, we also include the Barabási-Albert preferential attachment model [Alb02] to highlight the effect of a severely biased degree distribution at fixed edge density. Given a set of initial vertices, a constant c, and a number n of additional vertices, each new vertex is added to the graph with c edges attached to the existing nodes with probability proportional to their current degree. We match the original graphs' number of vertices and select c such that the same number of edges are added (up to integer rounding).

4.4.4 Replicability

All experiments are fully replicable with our open source code repository [Goo18b]. After installing the software with the `README` instructions, we direct the interested reader to `REPLICABILITY.md` for detailed instructions.

To sanitize the data, graphs are relabeled with vertices $\{0, \ldots, n-1\}$ and are written to files for each solver's required format. The reduction routines are natively nondeterministic, but data is explicitly sorted in both OCT- and VC-based routines such that a single run will generate identical results on distinct hardware and software environments.

All algorithms are available as standalone solvers using a command-line interface, and Python scripts are provided for reproducing the experiments, including tables and plots.

4.5 Quantum-Specific Results

4.5.1 Experimental Setup

All experiments were run on three identical Dell PowerEdge R430 servers, each with an Intel E5-2623 v2 CPU (3.5GHz single-core turbo, 10MB cache) and 64GB ECC DDR4 RAM. Each server ran Fedora 27 with Linux kernel 4.16.7, CPLEX 12.8, and GLPK 4.61. C and C++ code were compiled with Clang version 5.0.1, Java code was compiled with OpenJDK 1.8.0_181, and Python code was run with Python 3.5.6 (restricted by CPLEX 12.8).

4.5.2 Preprocessing Effectiveness

In this subsection we harden the benchmark by applying the reduction routines detailed in Section 4.3.1. We denote reductions as a partition of the original vertex set $V = V_r \uplus V_o \uplus V_b \uplus V'$. The vertices that may be removed without changing OCT are denoted V_r. For some fixed optimal

Table 4.1 A summary of preprocessing statistics on WH and quantum datasets. Ranges are given for the number of vertices and edge density in both the original and reduced graphs. The statistic $|\widehat{V_r}|$ reports the percentage of vertices removed on average. Likewise, normalized means are reported for edge removals E_r, fixed-OCT vertices V_o, and fixed-bipartite vertices V_b; dashes denote zero changes. The percentage of graphs solved completely by preprocessing routines is also reported.

	Original Graph		**Reductions**					**Reduced Graph**	
Dataset	$\|V\|$	$\|E\|/\|V\|$	$\|\widehat{V_r}\|$	$\|\widehat{E_r}\|$	$\|\widehat{V_o}\|$	$\|\widehat{V_b}\|$	**Solved**	$\|V' \cup V_b\|$	$\|E'\|/\|V' \cup V_b\|$
WH-aa	39–300	1.8–5.4	13%	-	-	70%	11%	27–265	1.9–6.1
WH-j	33–150	1.5–5.9	29%	-	-	13%	23%	8–74	2.0–7.5
b-50	50	2.0–2.7	2%	-	-	4%	-	43–50	2.1–2.7
b-100	100	4.6–5.1	-	-	-	-	-	100	4.6–5.1
gka	20–125	2.1–61.3	-	-	9%	-	11%	6–100	2.0–43.5

solution S, vertices V_o must be in S and vertices V_b cannot be in S. Finally, the remaining vertices are labeled V'. Analogously, the edges are partitioned into $E = E_r \uplus E'$. The sets V_r, E_r, and V_o may be safely removed from the graph, leaving the reduced graph with vertices $V_b \cup V'$ and edges E'.

Table 4.1 summarizes preprocessing results over the non-synthetic datasets. We observe that the reduction routines' effectiveness is dataset-dependent. The WH data is amenable to vertex removals, particularly WH-j, which had its largest graph reduced from 241 to 74 vertices. Perhaps due to a very low edge density, the WH-aa also had a significant number of vertices labeled bipartite. The GKA dataset was only affected by reductions that fixed OCT vertices, which can be important for exact algorithms fixed-parameter tractable in the solution size. Few reductions applied to the Beasley data, with b-100 remaining untouched. Both WH and GKA data contained instances that were completely solved by preprocessing.

4.5.3 Use Case: Heuristic Solutions

For the first quantum use case, an embedding compiler may need a bipartization of an input program (e.g., a QUBO or circuit) in order to prune a search space of embeddings, but has very little time budgeted for this step. Therefore we compare the best (potentially non-optimal) solutions found per solver with fixed timeouts. The algorithms capable of producing heuristic solutions are the heuristic ensemble (HE), the improved iterative compression solver (IC), and the ILP formulation with CPLEX (ILP). To evaluate the heuristics and anytime algorithms in this scenario we choose timeouts of four different orders of magnitude (0.01, 0.1, 1.0, and 10 seconds). At the timeout, each algorithm is given a termination signal and is given time to output the last solution cached.

Table 4.2 Observed approximation factors for anytime algorithms: the heuristic ensemble (HE), iterative compression (IC), and integer linear programming (ILP). For each dataset, the worst-case approximation ratio over its instances is reported. Approximation ratios are with respect to OCT on the reduced graph, computed with ILP. A checkmark denotes that exact solutions are found on all instances, if a dataset has no checkmark then the best approximation algorithm is bolded.

Timeout:	**0.01(s)**			**0.1(s)**			**1(s)**			**10(s)**		
Dataset	HE	IC	ILP	HE	IC	ILP	HE	IC	ILP	HE	IC	ILP
`WH-aa`	**1.54**	1.57	24.25	1.33	**1.24**	2.10	1.29	1.29	**1.25**	1.24	1.08	✓
`WH-j`	1.11	✓	3.92	1.11	✓	✓	✓	✓	✓	✓	✓	✓
`b-50`	1.11	**1.09**	1.67	1.11	✓	1.21	1.08	✓	✓	✓	✓	✓
`b-100`	**1.07**	1.10	2.10	**1.07**	1.10	1.34	**1.05**	1.07	1.15	**1.05**	**1.05**	**1.05**
`gka`	**1.18**	1.21	2.36	**1.18**	1.21	1.36	**1.11**	**1.11**	1.17	1.11	1.11	**1.07**
`fcl-small`	1.50	**1.41**	5.00	1.50	**1.29**	1.57	1.33	1.20	**1.18**	1.23	1.18	**1.09**
`fcl-large`	1.71	**1.50**	5.00	1.53	**1.42**	2.33	1.43	**1.35**	1.48	1.38	1.29	**1.21**

Table 4.3 Run times (in seconds) of exact solvers on a representative sample of `Beasley` and `GKA` data with a 10 minute timeout. Algorithm-data pairings that did not finish within the timeout are denoted with a dash, and the best run time on a dataset is bolded.

Graph		**Solver**			**Graph**		**Solver**		
Dataset	*OPT*	VC	IC	ILP	**Dataset**	*OPT*	VC	IC	ILP
`b-100-1`	41	101.5	-	**79.5**	`gka-21`	40	1.5	31.2	**0.6**
`b-100-2`	42	190.8	-	**159.2**	`gka-22`	43	**5.0**	-	6.5
`b-100-3`	42	252.8	-	**58.0**	`gka-23`	46	**44.5**	-	63.2
`b-100-4`	41	212.7	-	**176.8**	`gka-24`	37	**76.6**	-	95.8
`b-100-5`	42	**217.6**	-	246	`gka-25`	42	**130.6**	-	169.6
`b-100-6`	43	**150.1**	-	223.3	`gka-26`	43	168.2	-	**113.6**
`b-100-7`	42	**189.0**	-	270.1	`gka-27`	62	555.1	-	**477.3**
`b-100-8`	43	368.7	-	**209.6**	`gka-28`	70	**300.8**	-	516.1
`b-100-9`	44	**333.5**	-	470.5	`gka-29`	77	**67.1**	-	423.0
`b-100-10`	44	**195.4**	-	478.4	`gka-30`	82	**33.6**	-	305.2
`gka-3`	23	**1.5**	273.4	4.4	`gka-31`	85	**12.4**	109.6	91.9
`gka-8`	28	**7.2**	-	8.5	`gka-32`	88	**5.4**	18.8	40.3

Table 4.2 reports the worst-case approximation factors achieved by an algorithm-data-timeout triple. Notably, across all data, a user could achieve worst-case approximation factors of 1.57 and 1.24 for timeouts 0.01(s) and 0.1(s) by solving with IC, and approximation factors of 1.25 and 1.07 for timeouts 1(s) and 10(s) by solving with ILP.

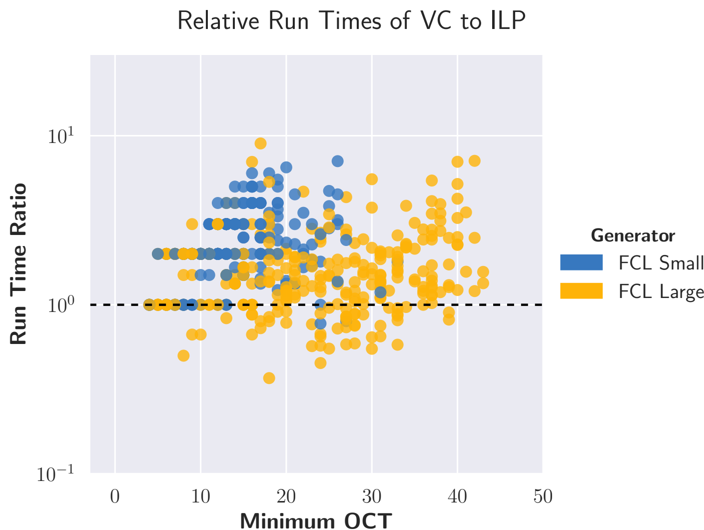


Figure 4.2 A scatter plot of VC vs. ILP run times on the FCL corpuses. Points above the line are (up to 10×) faster when run on ILP.

4.5.4 Use Case: Exact Solutions

In the second use case, a researcher may want a 'litmus test' evaluating whether a program's structure can be represented in a given quantum hardware. By recognizing that the OCT size of a problem graph is too large, we can show that such a configuration is impossible to embed in this hardware topology. We simulate this use case by computing exact solutions with a 10 minute timeout on preprocessed instances using the three solvers that could guarantee optimality: the VC-solver (VC), iterative compression (IC), and ILP (ILP).

We find that the solvers are fairly evenly split on total run time victories (Table 4.3), with ILP performing better on the `Beasley` data and VC performing better on `GKA`. Iterative compression only finished a handful of times within 10 minutes, and only once was competitive (`gka-32`).

4.6 Generalized Results

In this section, we expand the experimental envelope further by taking each `WH` and quantum dataset and creating several "look-a-likes" – synthetic graphs that match the original in different facets. We use the Erdös-Rényi generator to match density, the Tunable-OCT model to match proximity to bipartite, the Chung-Lu model to match degree distribution, and Barabási-Albert to increase degree distribution heterogeneity at a fixed edge density; all models and parameter settings are detailed in Section 4.4.3. A single synthetic instance is generated from a quantum instance, a synthetic graph generator, and a pseudorandom number generator seed. 15

seeds are used for the preprocessing and heuristic experiments, and 5 seeds are used for the (computationally expensive) exact results.

4.6.1 Reduction Effectiveness

Table 4.4 depicts the effectiveness of reduction routines on the synthetic data. Across most datasets, the Tunable-OCT and Chung-Lu synthetic graphs resulted in more vertex removals than the original graphs. In the case of `WH-aa`, this increase in V_r also involved a drastic decrease of vertices labeled bipartite (V_b). Again, the `Beasley` datasets remained relatively unaffected by reduction, excepting the Tunable-OCT analogues for `b-50`.

4.6.2 Heuristic Solutions

When extending the heuristic solution comparison to synthetic data, we find the same best practices generally still apply (Table 4.5). At 0.01s, IC has little time to improve on the initial heuristic solution and so the quick solutions found by HE remain competitive. At 0.1s and 1s, HE begins to have diminishing returns and the compression steps towards an exact solution begin to pay off for IC. At 1s, ILP also begins to produce winning solutions, solving all of `b-50` and splitting the wins otherwise. Finally, with 10s to execute the full CPLEX reduction routines, ILP begins to dominate.

We observe that over all data and algorithms, 0.01 seconds achieves a 2.70-approximation (and 1.42 outside of `WH-aa-to`) with IC; 0.1 seconds yields a 1.79-approximation with IC; 1 second yields a 1.65-approximation with IC, and 10 seconds yields a 1.14-approximation with ILP. From a domain-science standpoint, these relatively small errors show that OCT can be considered a practical subroutine when searching for a ‘good enough’ solution in limited time.

4.6.3 Exact Solutions

Expanding on Table 4.3, we find that the choice between ILP and VC is heavily dataset- and generator-dependent. Notably, in the first facet of Figure 4.3 we find that Tunable-OCT data is difficult for the VC-solver and CPLEX can solve the instances up to 1000× faster. However, the Barabási-Albert instances are similarly easier for a VC-solver. The effectiveness of branch-and-reduce algorithms on instances with heavy-tailed degree distributions is well-known in folklore – the few high-degree vertices in this network result in few branches, and the remaining (sparse) graph can sometimes be solved by reduction routines alone. The results for `WH-j` and `GKA` are also split fairly evenly, although the latter has no clear split by generator. Notably, `GKA` has instances with very large OCT that are best solved with ILP; in the rest of the data VC tends to perform best as minimum OCT increases. The `b-50` and `b-100` corpuses are best solved by ILP and VC, respectively.

Figure 4.3 Relative run times computed by dividing VC run times by those from ILP with one thread; the dashed line indicates equality. Data points reported over a synthetic corpus with five random generator seeds.

Overall, ILP generally does best on graphs with small minimum OCT (≤ 25), otherwise VC returns faster solutions. The results in `WH-aa` and `WH-j` also suggest that Tunable-OCT is best solved with ILP; extended work could evaluate whether these datasets are particularly easy for ILP or particularly hard for VC.

4.7 Conclusion

We experimentally evaluate state-of-the-art approaches to computing Odd Cycle Transversal on the canonical `WH` dataset, a new benchmark dataset from quantum annealing, and synthetic data generated using four common random graph models to emphasize different structural properties. Whereas previous experimental evaluations were limited to 61 graphs, each solved by modern methods within 1s, we utilize 116 classical benchmark instances, 840 frustrated cluster loop instances, and nearly 7000 synthetic instances that push the limits of each solver.

On this significantly expanded corpus, we found that no single implementation dominates

all scenarios. Under the most extreme time constraint of 0.01s, sampling from simple heuristics provides the best solutions. When using 0.1s, 1s, and 10s timeouts we find that a heuristic solution improved with structured work towards an exact solution performs best, including techniques such as iterative compression. After 10s, though, the preprocessing overhead of using CPLEX pays off and an ILP-solver dominates.

For significantly longer runs (a 10m timeout), we find IC uncompetitive and the choice between VC- and ILP-based solvers depends on the instance structure. We expect that the effectiveness of solvers heavily depends on potency of their preprocessing routines – CPLEX applies algebra-based reductions on the ILP before performing branch-and-cut, and Akiba and Iwata's VC-solver applies reductions after each branch in their branch-and-reduce model. More work is needed in identifying (in)effective reduction routines based on instance structure. Results from the 2019 Parameterized Algorithms and Computational Experiments (PACE) challenge[4] for Vertex Cover may be of particular interest towards this goal.

In addition to further study in preprocessing and structure of difficult instances, more flexible implementations are needed on the combinatorial algorithm front. Algorithms such as VC branch-and-reduce can be CPU-parallelized on a shared-memory system to potentially obtain super-linear speed-ups (witnessed by CPLEX in Appendix 4.9). Techniques such as iterative compression may also lend themselves to GPU-parallelization by executing an exponential number of disjoint Minimum Cut subroutines in parallel.

In addition to parallelism, implementations should also be provided as anytime algorithms whenever possible. Many techniques create iteratively better lower bounds which can be completed into upper bounds with a simple rounding strategy. These "good enough" solutions are directly useful for applications such as quantum computing, and the lower bounds may be useful in their own right.

4.8 Appendix: Implementation Details

4.8.1 Data Ingestion and Sanitization

Original data comes from two sources. `Wernicke-Hüffner` data is provided in the Hüffner code download [Hüf06], and `Beasley` and `GKA` data comes from Beasley's repository [Bea18].

When parsing the graphs with Python we read them into a NetworkX graph and remove edges with weight zero (used to denote non-edges in some problems) and self-loops. We then relabel the vertices to $\{0, \dots, n-1\}$. To remove possible non-determinism in how vertices are relabeled, we specify that node labels are relabeled by lexicographical order of the original vertex labels, guaranteeing that each graph is always converted in the same way.

[4] `https://pacechallenge.org/2019/`

See the **Data** section of `REPLICABILITY.md` in our repository for information on how to use our scripts for automating this download and parsing process.

4.8.2 Reduction Routines

Reduction routines for preprocessing come Wernicke [Wer03] and Akiba and Iwata [Aki16].

While Wernicke originally implemented his reductions in Java, the code does not appear to have been open sourced. We implement his reduction routines in Python3 with NetworkX. Some care must be taken that these reductions operate deterministically so the results can be reproduced. Specifically, reduction rules 4 and 6 require vertex cuts, which are returned in arbitrary order when computed by NetworkX; we convert the cuts to tuples and sort them by vertex label. Additionally, reduction rules 7, 8, and 9 remove particular configurations in the graph based on degree 2 and 3 vertices; we sort these sets and the related neighborhoods.

For Akiba and Iwata's reduction routines, we modify their GitHub code [Aki17] so that no branching is done after the first iteration of reduction routines, and the preprocessed graph is output instead. To preprocess a graph, we apply Wernicke reductions first, then Akiba-Iwata reductions, and repeat until the graph does not change. This was done primarily because some of Akiba-Iwata's reductions will not apply after the Wernicke reductions, simplifying the conversion from VC to OCT.

In order to make our experiments replicable, we verified that these reductions are determinstic by performing multiple rounds of preprocessing on different machines and checking that the resulting graphs were isomorphic, if small enough to be feasible, and had matching degree, triangle, and number of cliques sequences using the NetworkX `could_be_isomorphic` method otherwise. To verify that these reductions are safe, we saved and verified a certificate (the odd cycle transversal) from each run of a solver that returned a feasible solution.

See the **Reductions** section of `REPLICABILITY.md` in our repository for information on how to run our scripts for applying these reduction routines.

4.8.3 Heuristics

We implemented the heuristic ensemble in Modern C++14. Given a graph file and a timeout, the ensemble will run greedy independent set (MinDeg), Luby's Algorithm (Luby), DFS 2-coloring (DFS), and BFS 2-coloring (BFS) in a round-robin fashion until the time limit is reached, returning the single best solution found by any heuristic. See [Goo18c] for more on MinDeg, [Lub86] for more on Luby, and [Wer03] for more on DFS.

See the **Utilities/Heuristics Solver** section of `README.md` in our repository for information on how to run the heuristic ensemble solver.

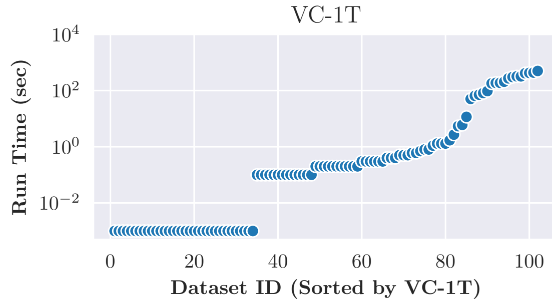


Figure 4.4 Run times (log scale) of all quantum datasets when sorted in order of fastest to slowest when solved with the OCT → VC → ILP formulation and one thread (VC-1T).

4.8.4 Hüffner Improvements

We implemented our improvements to Hüffner's implementation [Hüf06] in Modern C++14, and rewrote the original solver to compile in C11. By default, the `enum2col` solver is run, with the preprocessing level p specified by the user: The default algorithm ($p = 0$), the default algorithm with a heuristic bipartite subgraph starting the ordering ($p = 1$), and the default algorithm with a heuristic bipartite subgraph starting the ordering and the remaining vertices sorted such edges are introduced as quickly as possible ($p = 2$).

See the **Utilities/Iterative Compression Solver** section of `README.md` in our repository for information on how to download, install, and run this improved iterative compression solver.

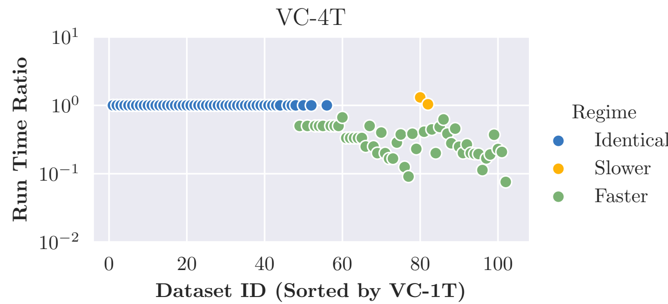


Figure 4.5 The run time ratio (log scale) of a 4-threaded solver vs. a single-threaded solver using the OCT → VC → ILP formulation. Easier instances have identical solve times, and all but two of the harder instances benefit from the thread increase.

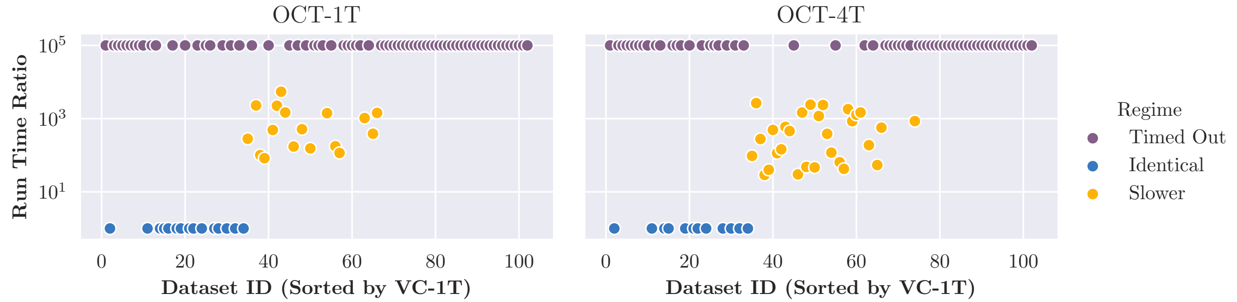


Figure 4.6 The run time ratio (log scale) of the single- and four-threaded solvers using the OCT $\rightarrow$ ILP formulation compared to the single-threaded OCT $\rightarrow$ VC $\rightarrow$ ILP formulation (VC-1T). The majority of instances time out at 10 minutes, and no instance is solved faster than with VC-1T.

4.9 Appendix: Extended Results

4.9.1 ILP Solver Comparison

As noted in the introduction, previous work [Hüf09; Aki16] has conflicting reports on the effectiveness of Integer Linear Programming (ILP) solvers. In this section we identify the best configuration for solving OCT with ILP by using CPLEX with either a formulation from OCT $\rightarrow$ ILP or OCT $\rightarrow$ VC $\rightarrow$ ILP, and evaluate the impact of one thread vs. four. It is well known that GLPK is not competitive with CPLEX, and the 4MB RAM limitations mentioned in [Hüf09] are irrelevant given modern resources, therefore we do not consider these factors.

We find that using the OCT $\rightarrow$ VC $\rightarrow$ ILP formulation alone beats the alternative formulation, and that multithreading with this formulation can result in superlinear speedups. Figure 4.4 visualizes run times for all quantum datasets when sorted in order of fastest to solve with our canonical configuration, the OCT $\rightarrow$ VC $\rightarrow$ ILP formulation with one thread. We find that in general (Figure 4.5), increasing to four threads is worthwhile and can result in a superlinear speedup; we observed speedups over 10× for only a 4× increase in cores. Finally, we observe that the direct OCT $\rightarrow$ ILP formulation severely impacts CPLEX for the worse, regardless of threads used (Figure 4.6).

Table 4.4 A summary of preprocessing statistics over all datasets. Ranges are given for the number of vertices and edge density in both the original and reduced graphs. The normalized statistic $|\widehat{V_r}|$ reports the average percentage of vertices removed. Likewise, normalized means are reported for edge removals E_r, fixed-OCT vertices V_o, and fixed-bipartite vertices V_b; dashes denote zero changes. The percent of graphs per dataset completely solved by preprocessing routines is also noted. Results are reported over 15 seeds per random graph generator, per original dataset.

	Original Graph		Reductions					Reduced Graph	
Dataset	$\lvert V\rvert$	$\lvert E\rvert/\lvert V\rvert$	$\lvert\widehat{V_r}\rvert$	$\lvert\widehat{E_r}\rvert$	$\lvert\widehat{V_o}\rvert$	$\lvert\widehat{V_b}\rvert$	**Solved**	$\lvert V' \cup V_b\rvert$	$\lvert E'\rvert/\lvert V' \cup V_b\rvert$
WH-aa	**39–300**	**1.8–5.4**	**13%**	**-**	**-**	**70%**	**11%**	**27–265**	**1.9–6.1**
WH-aa-ER	15–300	1.2–5.6	1%	-	-	2%	4%	10–300	1.4–5.6
WH-aa-TO	27–300	0.9–4.1	15%	-	-	12%	8%	10–299	1.4–4.1
WH-aa-CL	27–300	1.3–5.5	55%	-	6%	12%	9%	9–155	1.5–6.7
WH-aa-BA	14–300	1.7–5.9	1%	-	-	1%	8%	7–300	1.4–5.9
WH-j	**33–150**	**1.5–5.9**	**29%**	**-**	**-**	**13%**	**23%**	**8–74**	**2.0–7.5**
WH-j-ER	33–241	1.1–6.3	6%	-	-	4%	13%	18–239	1.5–6.3
WH-j-TO	33–150	0.8–4.1	17%	-	-	7%	32%	10–104	1.4–4.1
WH-j-CL	33–241	1.2–6.2	39%	-	4%	7%	22%	10–138	1.5–7.5
WH-j-BA	33–241	1.9–5.6	7%	-	1%	4%	14%	10–241	1.3–5.7
b-50	**50**	**2.0–2.7**	**2%**	**-**	**-**	**4%**	**-**	**43–50**	**2.1–2.7**
b-50-ER	50	1.7–3.3	3%	-	-	3%	-	31–50	1.9–3.4
b-50-TO	50	1.2–2.5	14%	-	-	6%	3%	13–48	1.4–2.7
b-50-CL	50	1.7–3.0	8%	-	1%	4%	-	24–49	1.8–3.2
b-50-BA	50	2.8	4%	-	2%	5%	6%	30–50	1.9–2.8
b-100	**100**	**4.6–5.1**	**-**	**-**	**-**	**-**	**-**	**100**	**4.6–5.1**
b-100-ER	100	4.3–5.5	-	-	-	-	-	98–100	4.3–5.5
b-100-TO	100	3.6–4.8	-	-	-	1%	-	95–100	3.6–4.8
b-100-CL	100	4.1–5.5	1%	-	-	1%	-	95–100	4.2–5.5
b-100-BA	100	4.8–5.6	-	-	-	-	-	98–100	4.3–5.7
gka	**20–125**	**2.1–61.3**	**-**	**-**	**9%**	**-**	**11%**	**6–100**	**2.0–43.5**
gka-ER	20–125	1.8–61.4	-	-	9%	-	13%	6–100	1.9–43.7
gka-TO	20–125	1.2–61.4	1%	-	12%	1%	8%	6–100	1.6–43.6
gka-CL	20–125	1.7–60.9	1%	-	10%	-	11%	6–100	1.7–43.2
gka-BA	20–125	2.8–31.2	-	-	1%	-	1%	9–123	1.6–29.9

Table 4.5 Observed approximation factors for anytime algorithms and heuristics at various timeouts. For each dataset, the worst-case approximation ratio over its instances is reported. Approximation ratios are with respect to OCT on the reduced graph. A checkmark denotes that exact solutions are found on all instances, if a dataset has no checkmark then the best approximation algorithm is bolded. If OCT could not be found within 10 minutes then the instance is not included; the percent of included data points is denoted in the *represented* column.

	Timeout:	**0.01(s)**			**0.1(s)**			**1(s)**			**10(s)**		
Dataset	**Represented**	HE	IC	ILP	HE	IC	ILP	HE	IC	ILP	HE	IC	ILP
WH-aa-er	61%	**1.26**	1.30	3.16	**1.21**	**1.21**	1.52	1.21	**1.16**	1.35	**1.14**	**1.14**	1.15
WH-aa-to	100%	**2.41**	2.70	15.00	1.95	**1.79**	5.75	**1.65**	**1.65**	3.11	1.54	1.56	✓
WH-aa-cl	100%	1.33	**1.24**	3.43	1.25	**1.21**	1.47	1.18	1.19	**1.11**	1.17	1.17	**1.05**
WH-aa-ba	86%	1.50	**1.39**	4.43	1.40	**1.33**	1.73	1.33	**1.30**	1.39	1.30	1.30	**1.14**
WH-j-er	93%	**1.30**	1.31	3.12	**1.21**	**1.21**	1.64	1.19	**1.18**	1.39	1.19	1.18	**1.11**
WH-j-to	100%	1.78	**1.31**	8.33	1.44	**1.18**	1.88	1.31	1.11	✓	1.15	✓	✓
WH-j-cl	100%	**1.29**	**1.29**	3.67	1.25	**1.17**	1.52	1.21	1.17	**1.08**	1.17	1.15	✓
WH-j-ba	100%	1.60	**1.42**	4.58	1.40	**1.33**	1.93	**1.33**	1.36	1.48	1.31	1.36	**1.10**
b-50-er	100%	1.33	**1.17**	2.00	1.18	**1.08**	1.31	1.12	1.07	✓	✓	1.07	✓
b-50-to	100%	1.25	**1.10**	2.25	1.20	**1.14**	1.22	✓	✓	✓	✓	✓	✓
b-50-cl	100%	1.20	**1.17**	1.75	1.17	**1.12**	1.27	1.14	1.08	✓	✓	1.08	✓
b-50-ba	100%	1.25	**1.18**	1.90	1.17	**1.09**	1.18	1.10	1.09	✓	✓	✓	✓
b-100-er	100%	**1.10**	1.19	2.12	**1.10**	**1.10**	1.40	**1.07**	**1.07**	1.21	**1.07**	**1.07**	1.09
b-100-to	100%	**1.16**	1.24	2.70	**1.15**	1.16	1.45	**1.12**	**1.12**	1.31	**1.10**	**1.10**	1.11
b-100-cl	100%	**1.17**	1.18	2.39	**1.14**	**1.14**	1.49	**1.12**	**1.12**	1.20	1.12	1.10	**1.07**
b-100-ba	100%	1.21	**1.18**	3.04	1.14	**1.11**	1.38	**1.11**	**1.11**	**1.11**	1.11	1.11	**1.03**
gka-er	99%	1.25	**1.24**	2.59	**1.20**	1.24	1.44	1.20	1.20	**1.19**	1.16	1.16	**1.10**
gka-to	100%	**1.29**	1.32	2.61	1.25	**1.18**	1.62	**1.18**	**1.18**	1.26	1.12	1.15	**1.08**
gka-cl	100%	**1.22**	1.24	3.14	1.18	**1.15**	1.39	**1.12**	1.17	1.24	1.09	1.12	**1.07**
gka-ba	100%	**1.27**	1.28	3.19	**1.20**	1.24	2.25	1.20	1.16	**1.12**	1.11	1.14	**1.06**

CHAPTER

5

EDITING TO BOUNDED DEGENERACY

5.1 Introduction

In our first graph algorithm we examine the problem of editing a graph to have (user-specified) bounded degeneracy. As previous noted, the graph invariant *degeneracy* has strong ties to the more local structure of *k-cores* and degeneracy *orderings*, which are useful for both algorithm design and real-world network analysis. Bounded degeneracy is a hereditary property (i.e., closed under induced subgraphs), so by the classic meta-theorems by Lewis and Yannakakis [Lew80] and Lund and Yannakakis [Lun93] it is NP-hard and APX-hard to compute a minimum vertex deletion set to bounded degeneracy, respectively. We provide the first full proof that both the vertex- and edge-deletion variants of degeneracy editing are APX-hard with the same lower bounds as Vertex Cover. This result has recently been strengthened to $o(\log n/r)$-inapproximability [Dem19], which implies that no constant factor approximation exists. To avoid this complexity, we introduce a *bicriteria* approximation algorithm that provides approximations in two facets – the edit size and the target degeneracy. By doing so we are able to achieve a $(4, 4)$-approximation for vertex deletion to bounded degeneracy.

This work fits into a larger framework that uses editing algorithms to approximation optimization problems on instances with *noisy* structure [Dem19]. The framework exploits that certain optimization problems may be solved quickly on structured instances. For example, several problems may be solved in time $O(n^{tw})$ using dynamic programming, therefore instances

with bounded treewidth may be solved in polynomial time. Treewidth and bounded degeneracy are a few examples of this *ideal* structure. Graphs with close-to-ideal structure (in terms of edit distance) are denoted as having *noisy* structure. The framework takes a problem instance with a graph having noisy structure, then computes an approximate edit set, solves the optimization problem quickly on the subgraph with ideal structure, then reintroduces the edits in such a way that an approximation for the full instance is constructed. Exact edits to the ideal structure are not required, but an improved editing algorithm results in an improved approximation to the optimization problem solved.

To this end, we are especially interested in experimental work exploring the practicality of these editing algorithms. In proposed additional work, we will evaluate the degeneracy editing algorithm introduced here against an LP-rounding algorithm run on CPLEX. Beyond the serial implementation we will explore shared-memory parallelism, allowing algorithms to scale to very large instances or quickly collect samples of edits. We will then compare the distributions of solutions found for varying target degeneracies.

In the remainder of this chapter, we begin by proving that both the vertex- and edge-deletion variants of r-Degenerate Vertex Deletion and r-Degenerate Edge Deletion (defined in Background) are APX-hard with a strict reduction from Vertex Cover. We then provide an approximation algorithm based on the local ratio theorem, and conclude with proposed experimental work.

5.2 Degeneracy Editing is APX-hard

In this section we prove that both r-DVD and r-DED are APX-hard by reduction from Vertex Cover (VC), which has been shown to be $\frac{7}{6}$-hard [Hås01], 1.36-hard [Din05], and, assuming the Unique Games Conjecture, $(2-\varepsilon)$-hard [Kho08]. Our reduction uses gadgets to encode an arbitrary VC instance into a r-DVD (r-DED) instance. We then show that an arbitrary solution to r-DVD (r-DED) can be re-wired into a canonical solution without increasing solution size; this canonical solution has a straightforward mapping back to a VC solution. The intuition behind the gadgets is that a single (vertex or edge) deletion in a gadget will 'implode' to remove all vertices internal to a gadget from the instance's $(r+1)$-core (Lemmas 6 and 7). Additionally, these deletions can be routed to a single vertex (edge) per bomb to construct a canonical solution for r-DVD (r-DED). In the following subsections we prove this result for r-DVD, then provide the few essential changes required for r-DED.

5.2.1 Mapping a VC Instance to a r-DVD Instance

We begin by building a `bomb` gadget such that vertex deletions to reduce the degeneracy can be interpreted as covering a vertex. A fundamental tool for this `bomb` gadget is a `path` gadget with adjustable width to allow tunable-degeneracy.

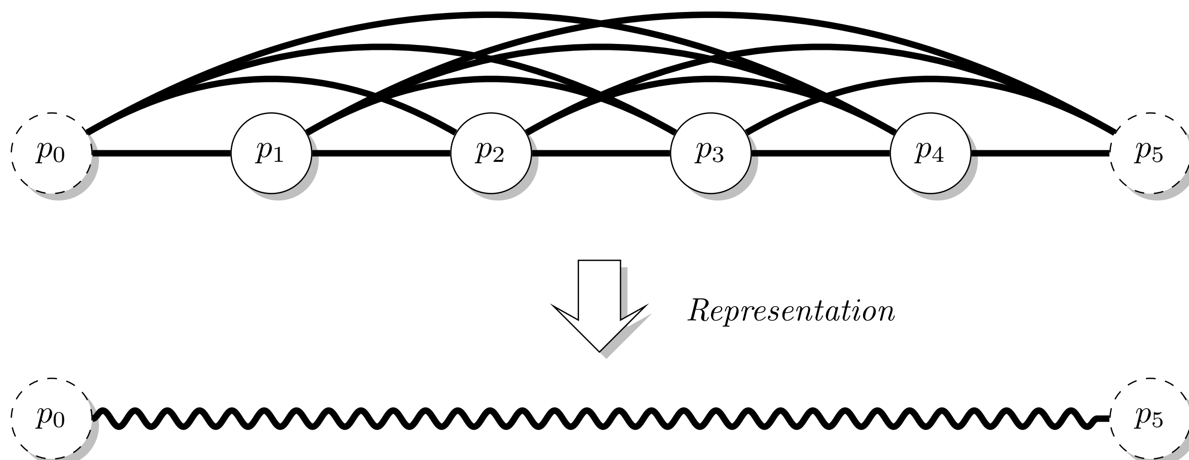


Figure 5.1 A `path` gadget of width 5. The endpoint vertices p_0, p_5 have degree 4, and internal vertices p_1, p_2, p_3, p_4 have degree 5. Solid vertices are internal to the gadget, whereas dashed vertices are interface points to the rest of the graph.

Definition (`path` Gadget)**.** *A `path` gadget of width $w \geq 2$ has two endpoints of degree $w-1$, and $w-1$ internal vertices of degree w. A path gadget is constructed by starting with vertices $p_0, p_1, p_2, \ldots, p_{w-1}, p_w$ and adding edges $\{(p_0, p_i) \mid 1 \leq i \leq w-1\}$ and $\{(p_i, p_j) \mid 1 \leq i \leq w-1, i+1 \leq j \leq w\}$. A path with interface points u, v and width w is denoted $\texttt{path}(u, v, w)$.*

An example `path` gadget is visualized in Figure 5.1. By construction, the `path` will be in the $(r+1)$-core when given width $r+1$ (assuming that its endpoints have edges elsewhere). We now show that this membership in the $(r+1)$-core is *sharp*: a single edit will remove the entire gadget from this core.

Lemma 6. *Given a graph G with vertices u, v and induced subgraph $P = \texttt{path}(u, v, r+1)$, a single vertex or edge deletion in P guarantees no internal vertex of P is in the $(r+1)$-core of G.*

Proof. By construction, every internal vertex of the path gadget P has degree exactly $r+1$, and so deleting any edge or internal vertex from P necessarily reduces the degree of at least one internal vertex $v^* \in P$ by one, to r. This decrease removes v^* from the $(r+1)$-core of G, which

decreases the degree of each vertex in $N(v^*)$ by one when v^* is removed from the graph. By induction, all internal vertices of P are removed from the $(r+1)$-core of G. □

Using these `path` gadgets, we can now create simple `bomb` gadgets where a deletion in one area can propagate through the whole gadget.

Definition (`bomb` Gadget)**.** *A `bomb` gadget of width $w \geq 2$ is composed of an interface vertex v_i, internal vertices a_i, b_i, c_i, edges (a_i, b_i) and (a_i, c_i), and paths `path`(v_i, a_i, w) and `path`(b_i, c_i, w). A `bomb` with interface point v_i and width w is denoted `bomb`(v_i, w).*

An example `bomb` gadget is visualized in Figure 5.2. Similar to the `path` gadget, the `bomb` gadget is also sharp.

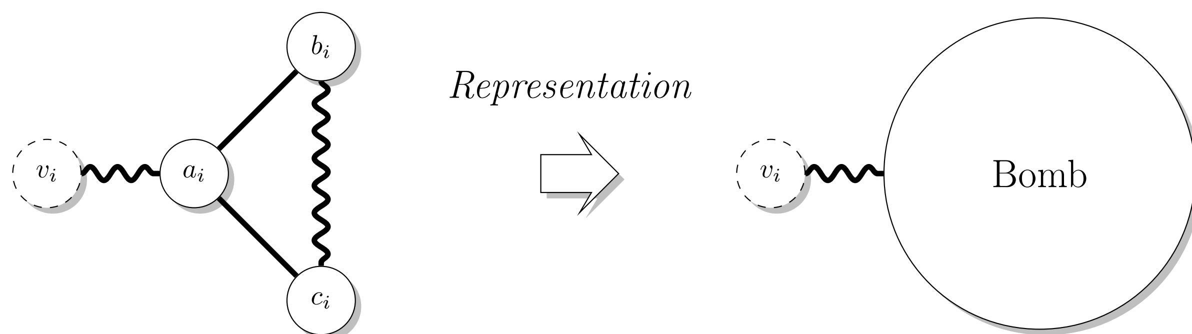


Figure 5.2 A `bomb` gadget, with `path` gadget widths determined by r. Solid vertices are internal to the gadget, whereas the dashed vertex is an interface point to the rest of the graph.

Lemma 7. *Given a graph G containing a bomb gadget `bomb`$(v_1, r+1)$, deleting a single vertex or edge will remove all internal vertices from the $(r+1)$-core of the graph.*

Proof. There are several cases. If b_i, c_i, or any vertex or edge in `path`$(b_i, c_i, r+1)$ is deleted, then by Lemma 6 all of `path`$(b_i, c_i, r+1)$ is removed from the $(r+1)$-core, including the `path`'s endpoints. This removal reduces the degree of a_i to r, and so a_i too is removed from the $(r+1)$-core. By Lemma 6, `path`$(v_i, a_i, r+1)$ is removed from the $(r+1)$-core, proving that all internal vertices of the bomb were removed from the $(r+1)$-core.

If either edge (a_i, b_i) or (a_i, c_i) is deleted, then the degree of b_i (or c_i) is reduced to r and removed from the $(r+1)$-core. Thus, by the first case the result follows.

Finally, if any vertex or edge in `path`$(v_i, a_i, r+1)$ is deleted, by Lemma 6 we have that all vertices internal to the path gadget are removed from the $(r+1)$-core. Hence, we can delete all the remaining vertices in `path`$(v_i, a_i, r+1)$ without altering the $(r+1)$-core, leaving a_i only

two neighbors (b_i and c_i). Thus, a_i has degree less than $r + 1$, and is also removed from the $(r + 1)$-core. Finally, this reduces the degree of b_i and c_i, which removes $\texttt{path}(b_i, c_i, r + 1)$ from the $(r + 1)$-core, as well. □

With the bomb gadget defined, we can now define the function f for mapping an instance of Vertex Cover to an instance of r-Degenerate Vertex Deletion.

Definition (f: Mapping from VC to r-DVD)**.** *Let G have max degree Δ. Given an instance (G, Δ) of* VC*, we produce an instance (G', r) of* r-DVD *by letting $r = \Delta + 1$ and attaching a $\texttt{bomb}(v_i, r + 1)$ to every vertex $v_i \in V$.*

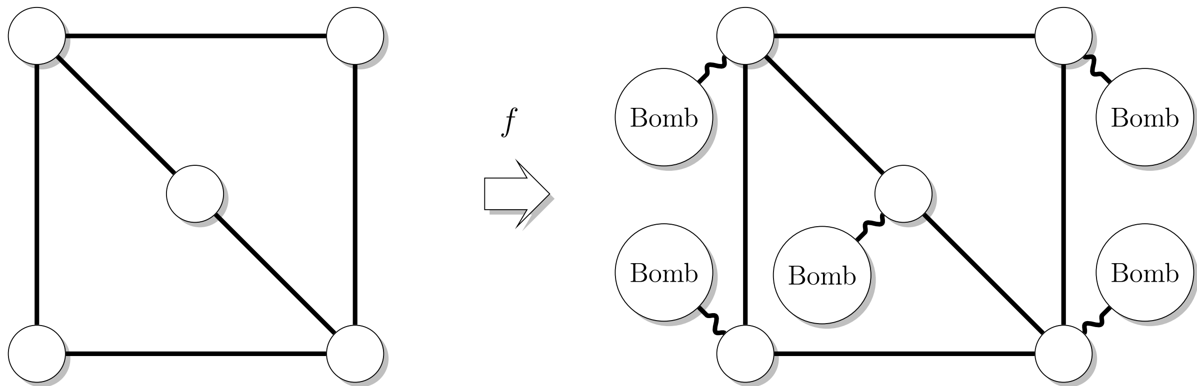


Figure 5.3 f maps an instance of Vertex Cover to an instance of r-Degenerate Vertex Deletion.

An example of this mapping between instances is given in Figure 5.3. Note that r must be greater than Δ to ensure that the degree of a vertex in the VC instance does not interfere with the sharpness results on the interface points in the r-DVD instance. If this guarantee was not required, it would be possible to edit a $\texttt{bomb}(v_i, r + 1)$ gadget and cause an implosion that did not include the vertex v_i itself.

5.2.2 Mapping a r-DVD Solution to a VC Solution

Once a solution y is found for r-DVD, we want to map y to a solution $g(y)$ for VC without increasing the solution size. At a high level, the function will operate in two steps, first transforming an arbitrary solution to r-DVD into a canonical solution, then reinterpreting the canonical solution as a VC solution.

Definition (g: Mapping from r-DVD to VC)**.** *Given a solution y to* r-DVD*, we map it to a solution $g(y)$ for* VC *in two stages:*

1. *Construct a canonical deletion set $\widehat{y}$. For each vertex v_i in G, if v_i or the bomb attached to v_i contains at least one vertex deletion, then add v_i to $\widehat{y}$.*

2. *Let $g(y) = \{v_i \mid v_i \in \widehat{y}\}$.*

We show that refining an arbitrary solution into a canonical solution is a safe process.

Lemma 8. *If y is a feasible solution to r-DVD, then $\widehat{y}$ is also feasible.*

Proof. The conversion to a canonical solution deletes a vertex v_i if and only if a vertex internal to $\texttt{bomb}(v_i, r+1)$ was deleted. By Lemma 7, every vertex in a bomb is removed from the $(r+1)$-core if v_i is removed, therefore the vertices removed from the $(r+1)$-core by $\widehat{y}$ is exactly equal to the vertices removed from $(r+1)$-core by y. Thus $\widehat{y}$ is feasible if y is feasible. □

5.2.3 Proving the strict reduction

By the definition of a strict reduction, we need to show the following properties about f and g:

$$COST_{\mathrm{VC}}(g(y)) \leq COST_{r\text{-DVD}}(y), \tag{5.1}$$

$$OPT_{\mathrm{VC}}(x) \leq OPT_{r\text{-DVD}}(f(x)), \tag{5.2}$$

$$OPT_{\mathrm{VC}}(x) \geq OPT_{r\text{-DVD}}(f(x)). \tag{5.3}$$

We prove each inequality in the following three lemmas.

Lemma 9. *Given the construction provided by g, for every solution y of r-DVD it holds that $COST_{\mathrm{VC}}(g(y)) \leq COST_{r\text{-DVD}}(y)$.*

Proof. Since any r-DVD solution y can be safely converted into a canonical form solution $\widehat{y}$ without increasing the solution cost, and $|\widehat{y}| = |g(y)|$ by definition, it suffices to show that $g(y)$ is feasible.

Suppose not, and the solution $g(y)$ has an uncovered edge $v_i v_j$. By definition of g, the vertices v_i, v_j, and their corresponding bombs have no vertex deletions; denote this subgraph as S. By Lemma 7, S must belong in the $(r+1)$-core, therefore y is not a feasible solution. This conclusion contradicts the assumption that y is feasible. □

Lemma 10. *Given the construction provided by f, for every instance x of* Vertex Cover *it holds that $OPT_{\mathrm{VC}}(x) \leq OPT_{r\text{-DVD}}(f(x))$.*

Proof. Let x be an arbitrary instance to VC and $f(x)$ the constructed instance to r-DVD. Let y be an optimal solution to $f(x)$, and $g(y)$ be the constructed (not necessarily optimal) solution to instance x. By Lemma 9, we have that $COST_{\mathrm{VC}}(g(y)) \leq COST_{r\text{-DVD}}(y) = OPT_{r\text{-DVD}}(f(x))$. By definition, $OPT_{\mathrm{VC}}(x) \leq COST_{\mathrm{VC}}(g(y))$, therefore $OPT_{\mathrm{VC}}(x) \leq OPT_{r\text{-DVD}}(f(x))$. □

Lemma 11. *Given the construction provided by f, for every instance x of* Vertex Cover *it holds that $OPT_{\text{VC}}(x) \geq OPT_{r\text{-DVD}}(f(x))$.*

Proof. Given an instance of VC on graph $G = (V, E)$, let $G' = (V', E')$ be the graph constructed according to f, and let $X \subset V$ be any optimal vertex cover for x. We construct $X' = \{v_i \mid v_i \in V' \text{ and } v_i \in X\}$ (that is, we delete the vertices corresponding to the vertices in the cover). From construction, it is clear that $|X| = |X'|$; it remains to show that X' is in fact a valid deletion set to an r-degenerate graph.

By Lemma 7, every `bomb` attached to a deleted vertex will be removed from the $(r+1)$-core, therefore $G \setminus X'$ may only have vertices from $V' \setminus X'$ (and their corresponding `bomb` gadgets) in its $(r+1)$-core. By definition of VC, each edge in G must be covered by a vertex in X, therefore by the construction of X' it holds that every non-`bomb` edge in G' will be adjacent to a deleted vertex. However, this implies that the only edges remaining are contained in `bomb` gadgets. A `bomb`$(v_i, r+1)$ must be r-degenerate if v_i has no edges outside of the `bomb` gadget, therefore $G' \setminus X'$ is r-degenerate.

Since X' is a valid solution to r-DVD and $|X| = |X'|$, it holds that $OPT_{\text{VC}}(x) \geq COST_{r\text{-DVD}}(X')$ $\geq OPT_{r\text{-DVD}}(f(x))$. □

Theorem. r-Degenerate Vertex Deletion *is APX-hard.*

Proof. Lemmas 9, 10, and 11 show a strict reduction from (APX-hard) Vertex Cover. □

5.2.4 Extension to r-DED

Similar to the vertex-deletion variant, r-DED is also APX-hard.

Theorem. r-Degenerate Edge Deletion *is APX-hard.*

The proof is similar to the vertex-deletion variant where a single edge deletion in a `bomb` gadget will represent a covered vertex. However, some care must be taken when sanitizing the arbitrary solution y because edge deletions may now happen outside of a `bomb` gadget. For simplicity we sketch a strict reduction from r-DVD to r-DED, starting with defining g.

Definition (g: Mapping from r-DED to r-DVD)**.** *Given a solution y to* r-DED*, we map it to a solution $g(y)$ to* r-DVD *in two parts:*

1. *If an edge in v_i's bomb is deleted then add v_i to $g(y)$.*
2. *If an edge (v_i, v_j) is deleted then add v_i to $g(y)$.*

By construction, if y is a solution to r-DED then $g(y)$ must be a solution to r-DVD because the vertex deletions also delete the edges in y. It is clear that $COST_{r\text{-DVD}}(g(y)) \leq COST_{r\text{-DED}}(y)$

because at most one vertex is added to $g(y)$ for each edge in y. For the same reason, we can show $OPT_{r\text{-DVD}}(x) \leq OPT_{r\text{-DED}}(f(x))$. The final inequality $OPT_{r\text{-DVD}}(x) \geq OPT_{r\text{-DED}}(f(x))$ can be shown by taking an optimal solution X to r-DVD and deleting an edge in v_i's bomb for each $v_i \in X$.

5.3 A Bicriteria Approximation Algorithm for Degeneracy Editing

In this section we prove the following:

Theorem. *r-DVD has a $\left(\frac{4m-\beta rn}{m-rn}, \beta\right)$-approximation algorithm.*

Corollary 5. *r-DVD has a $(4,4)$-approximation algorithm.*

Recall that an (α, β)-approximation in this context means that an edit set of cost $\alpha \cdot OPT$ is provided for editing a graph G to a βr-degenerate graph, where OPT is the optimal cost of editing G to have degeneracy r. In other words, we allow error in both the edit set and in the target degeneracy, and this error is bounded multiplicatively by α and β, respectively. The algorithm is defined in Algorithm 2, and the analysis is based on the local ratio theorem from Bar-Yehuda et al. [BY04].

5.3.1 The local ratio technique and analysis overview

Fundamentally, the local ratio theorem [BY04] is machinery for showing that "good enough" local choices accumulate into a global approximation bound. This bookkeeping is done by maintaining *weight vectors* that encode the choices made. The local ratio theorem applies to optimization problems of the following form: given a weight vector $w \in \mathbb{R}^n$ and a set of feasibility constraints $\mathscr{C}$, find a solution vector $x \in \mathbb{R}^n$ satisfying the constraints $\mathscr{C}$ and minimizing $w^T x$ (for maximization problems see [BY04]). We say a solution x to such a problem is *α-approximate with respect to w* if $w^T x \leq \alpha \cdot \min_{z \in \mathscr{C}}(w^T z)$.

Theorem (Local Ratio Theorem [BY04])**.** *Let $\mathscr{C}$ be a set of feasibility constraints on vectors in $\mathbb{R}^n$. Let $w, w_1, w_2 \in \mathbb{R}^n$ be such that $w = w_1 + w_2$. Let $x \in \mathbb{R}^n$ be a feasible solution (with respect to $\mathscr{C}$) that is α-approximate with respect to w_1, and with respect to w_2. Then x is α-approximate with respect to w as well.*

In our case, an instance of (βr)-Degenerate Vertex Deletion (abbreviated (βr)-DVD) is represented with (G, w, r, β), where G is the graph, w is a weight vector on the vertices (where w is the all-ones vector, $\overrightarrow{1}$, when G is unweighted), r is our target degeneracy, and β is a multiplicative error on the target degeneracy. Our bicriteria approximation algorithm will yield an edit set to a (βr)-degenerate graph, using at most $\alpha \cdot \text{OPT}_{(\beta r)\text{-DVD}}(G, w, r, \beta)$ edits. This

(weighted) cost function is encoded as an input vector of vertex weights w, which is evaluated with an indicator function $\mathscr{I}_X$ on a feasible solution X, such that the objective is to minimize $w^T \mathscr{I}_X$. Note that while the local ratio theorem can allow all feasible solutions, we require *minimal* feasible solutions for stronger structural guarantees.

Algorithm 2 Approximation for r-Degenerate Vertex Deletion

```
1: procedure LocalRatioRecursion(Graph G, weights w, target degeneracy r, error β)
2:     if V(G) = ∅ then
3:         return ∅.
4:     else if ∃ v ∈ V(G) where deg_G(v) ≤ βr then
5:         return LocalRatioRecursion(G \ {v}, w, r, β)
6:     else if ∃ v ∈ V(G) where w(v) = 0 then
7:         X ← LocalRatioRecursion(G \ {v}, w, r, β)
8:         if G \ X has degeneracy βr then
9:             return X.
10:        else
11:            return MinimalSolution(G, X ∪ {v}, r, β).
12:        end if
13:    else
14:        Let ϵ := min_{v∈V(G)} w(v)/deg_G(v).
15:        Define w_1(u) := ϵ · deg_G(u) for all u ∈ V.
16:        Define w_2 := w − w_1.
17:        return LocalRatioRecursion(G, w_2, r, β).
18:    end if
19: end procedure
```

To utilize the local ratio theorem, our strategy is to define a recursive function that decomposes the weight vector into $w = w_1 + w_2$ and then recurses on (G, w_2, r, β). By showing that the choices made in this recursive function lead to an (α, β)-approximation for the instances (G, w_1, r, β) and (G, w_2, r, β), by the local ratio theorem, these choices also sum to an (α, β)-approximation for (G, w, r, β).

As outlined in [BY04], the standard algorithm template for this recursive method handles the following cases: if a zero-cost minimal solution can be found, output this optimal solution, else if the problem contains a zero-cost element, do a problem size reduction, and otherwise do a weight decomposition.

Algorithm 2 follows this structure: Lines 2-3 are the first case, Lines 4-5 and 6-12 are the second case, and Lines 13-18 are the third case. The first two cases are typically straightforward, and the crucial step is the weight decomposition of $w = w_1 + w_2$. Note that the first case

Algorithm 3 Subroutine for guaranteeing minimal solutions

```
1: procedure MinimalSolution(Graph G, edit set X, target degeneracy r, error β)
2:     for vertex v ∈ V(G) do
3:         if G \ (X \ {v}) has degeneracy βr then
4:             return MinimalSolution(G, X \ {v}, r, β).
5:         end if
6:     end for
7:     return X.
8: end procedure
```

guarantees that all vertices in G have degree at least $\beta r + 1$ before a weight decomposition is executed, so we may assume without loss of generality that the original input graph also has minimum degree $\beta r + 1$.

In the following subsections we show that Algorithm 2 returns a minimal, feasible solution (Lemma 12), that the algorithm returns an (α, β)-approximate solution with respect to w_1 (Theorem 5.3.2), and finally that the algorithm returns an (α, β)-approximate solution with respect to w (Theorem 5.3).

5.3.2 Proof of approximation factor

Lemma 12. *Algorithm 2 returns minimal, feasible solutions for* (βr)-DVD.

Proof. We proceed by induction on the number of recursive calls. In the base case, only Lines 2-3 will execute, and the empty set is trivially a minimal, feasible solution. In the inductive step, we show feasibility by constructing the degeneracy ordering. We consider each of the three branching cases not covered by the base case:

- Lines 4-5: Given an instance (G, w, r, β), if a vertex v has degree at most βr, add v to the degeneracy ordering and remove it from the graph. By the induction hypothesis, the algorithm will return a minimal, feasible solution $X_{\beta r}$ for $(G - \{v\}, w, r, \beta)$. By definition, v has at most βr neighbors later in the ordering (e.g. neighbors in $G - \{v\}$), so the returned $X_{\beta r}$ is still a feasible, minimal solution.

- Lines 6-12: Given an instance (G, w, r, β), if a vertex v has weight 0, remove v from the graph. By the induction hypothesis, the algorithm will return a minimal, feasible solution $X_{\beta r}$ for $(G - \{v\}, w, r, \beta)$. If $X_{\beta r}$ is a feasible solution on the instance (G, w, r, β), then $X_{\beta r}$ will be returned as the minimal, feasible solution for this instance. Otherwise the solution $X_{\beta r} \cup \{v\}$ is feasible, and can be made minimal with a straightforward greedy subroutine (Algorithm 3).

- Lines 13-18: In this case, no modifications are made to the graph, therefore the recursive call's minimal, feasible solution $X_{\beta r}$ remains both minimal and feasible.

In all cases, a minimal, feasible solution is returned. □

We now show that a minimal, feasible solution is (α, β)-approximate with respect to the instance defined by weight function w_1:

Theorem. *Any minimal, feasible solution $X_{\beta r}$ is a $\left(\frac{4m-\beta rn}{m-rn}, \beta\right)$-approximation to the instance (G, w_1, r, β).*

Given a minimal, feasible solution $X_{\beta r}$, note that $w_1^T \mathscr{I}_{X_{\beta r}} = \epsilon \sum_{v \in X_{\beta r}} \deg_G(v)$, where ϵ is the cost of the vertex removed at each level of the recursive weight decomposition. Therefore it suffices to show that $b \le \sum_{v \in X_r} \deg_G(v)$ and $\sum_{v \in X_{\beta r}} \deg_G(v) \le \alpha b$. In other words, for some bound b, any minimal, feasible edit set X_r to degeneracy r, and any minimal, feasible edit set $X_{\beta r}$ to degeneracy βr. We prove these two bounds for $b = m - rn$ in Lemmas 13 and 15, respectively.

Lemma 13. *For any minimal feasible solution X_r for editing to degeneracy r,*

$$m - rn \le \sum_{v \in X_r} \deg_G(v).$$

Proof. Since $G \backslash X_r$ has degeneracy r, it has at most rn edges, so at least $m - rn$ edges were deleted. Each deleted edge had at least one endpoint in X_r, therefore $m - rn \le \sum_{v \in X_r} \deg_G(v)$. □

Before proving the upper bound, we define some notation. Let $X_{\beta r}$ be a minimal, feasible solution to (βr)-DVD and let $Y = V(G) \setminus X_{\beta r}$ be the vertices in the (βr)-degenerate graph. Denote by m_X, m_Y, and m_{XY} the number of edges with both endpoints in $X_{\beta r}$, both endpoints in Y, and one endpoint in each set, respectively. We begin by bounding m_{XY}:

Lemma 14. *For any $X_{\beta r}$, it holds that $m_{XY} \le 2m_Y + 2m_{XY} - \beta r|Y|$.*

Proof. Recall that we may assume WLOG that every vertex in G has degree at least $\beta r + 1$. Therefore $\beta r|Y| \le \sum_{v \in Y} \deg_G(v) \le 2m_Y + m_{XY}$, and so $m_{XY} \le 2m_Y + 2m_{XY} - \beta r|Y|$. □

Corollary 6. *For any $X_{\beta r}$, it holds that $-\beta r|X_{\beta r}| \ge -2m_Y - 2m_{XY} + \beta r|Y|$.*

Proof. Because $X_{\beta r}$ is minimal, every vertex in $X_{\beta r}$ will induce a $(\beta r+1)$-core with vertices in Y if not removed. Therefore each such vertex has at least $(\beta r+1)$-neighbors in Y, and $\beta r|X_{\beta r}| \le m_{XY}$. Substituting into Lemma 14, we find that $-\beta r|X_{\beta r}| \ge -2m_Y - m_{XY} + \beta r|Y|$. □

We now prove the upper bound:

Lemma 15. *For any minimal, feasible solution* $X_{\beta r}$ *to* (βr)-DVD,

$$\sum_{v \in X_{\beta r}} \deg_G(v) \leq 4m - \beta rn.$$

Proof. By using substitutions from Lemmas 14 and Corollary 6, we know that

$$\begin{aligned}
\sum_{v \in X_{\beta r}} \deg_G(v) &= 2m_X + m_{XY} \\
&\leq 2m_X + 2m_Y + 2m_{XY} - \beta r|Y| \\
&= 2m - \beta r|Y| \\
&= 2m + 2m_Y + 2m_{XY} - 2m_Y - 2m_{XY} + \beta r|Y| - 2\beta r|Y| \\
&\leq 2m + 2m_Y + 2m_{XY} - \beta r|X_{\beta r}| - 2\beta r|Y| \\
&\leq 4m - \beta rn.
\end{aligned}$$

□

Proof of Theorem 5.3.2. Let $X_{\beta r}$ be any minimal, feasible solution for editing to a graph of degeneracy βr. By definition of w_1 in Algorithm 2, it holds that $w_1^T \mathscr{I}_{X_{\beta r}} = \epsilon \sum_{v \in X_{\beta r}} \deg_G(v)$, and because ϵ is a constant computed independently of the optimal solution, it suffices to show that $\sum_{v \in X_{\beta r}} \deg_G(v)$ has an α-approximation.

By Lemma 13, any minimal, feasible edit set to a degeneracy-r graph has a degree sum of at least $m - rn$. If an edit set is allowed to leave a degeneracy-(βr) graph, then by Lemma 15, at most $4m - \beta rn$ degrees are added to the degree sum of $X_{\beta r}$. Therefore $X_{\beta r}$ is $\left(\frac{4m-\beta rn}{m-rn}, \beta\right)$ -approximate with respect to (G, w_1, r, β). □

We now prove the main result stated at the beginning of this section, Theorem 5.3.

Proof. For clarity, let $\alpha := \left(\frac{4m-\beta rn}{m-rn}\right)$; we prove that Algorithm 2 is an (α, β)-approximation. We proceed by induction on the number of recursive calls to Algorithm 2. In the base case (Lines 2-3), the solution returned is the empty set, which is trivially optimal. In the induction step, we examine the three recursive calls:

- Lines 4-5: Given an instance (G, w, r, β), if a vertex v has degree at most βr, add v to the degeneracy ordering and remove it from the graph. By the induction hypothesis, the algorithm will return an (α, β)-approximate solution $X_{\beta r}$ for $(G - \{v\}, w, r, \beta)$. Since v will not be added to $X_{\beta r}$, then $X_{\beta r}$ is also an (α, β)-approximation for (G, w, r, β).
- Line 6-12: Given an instance (G, w, r, β), if a vertex v has weight 0, remove v from the graph. By the induction hypothesis, the algorithm returns an (α, β)-approximate solution

$X_{\beta r}$ for $(G - \{v\}, w, r, \beta)$. Regardless of whether v is added to $X_{\beta r}$ or not, it contributes exactly zero to the cost of the solution, therefore an (α, β)-approximation is returned.

- Line 13-18: In this case, the weight vector is decomposed into w_1 and $w_2 = w - w_1$. By induction, the algorithm will return an (α, β)-approximate solution $X_{\beta r}$ for $(G, w - w_1, r, \beta)$. By Theorem 5.3.2, $w_1^T \mathscr{I}_{X_{\beta r}}$ is also (α, β)-approximate. Therefore, by Theorem 5.3.1, $w^T \mathscr{I}_{X_{\beta r}}$ must be (α, β)-approximate.

□

5.4 A Refined OCT Approximation Algorithm

In Chapter 3, we introduce an r-approximation algorithm for OCT on graphs with degeneracy r. In this section, we further refine this approximation factor by first editing to a graph with smaller degeneracy, applying the OCT approximation algorithm, then using *structural rounding* [Dem19] to round the removed vertices into a solution on the original graph instance.

Theorem. *Let G be a graph with edit distance d to the class of r-degeneracy graphs. There is a $(4r)$-approximation algorithm with additive error $4d$ for solving* OCT *on G.*

Proof. By Corollary 5 in Chapter 5, we can edit G into a graph G' and edit set X, where G' has degeneracy at most $4r$ and $|X| \leq 4d$. By Corollary 3 in Chapter 3, we have an odd cycle transversal S' on G' where S' is a $4r$-approximation for OCT on G'.

Let $S = S' \cup X$, this set is (by definition) a valid odd cycle transversal of G because deleting X from G produces G', and deleting S' from G' produces a bipartite graph. Additionally, since $|X| \leq 4d$ and $MinOCT(G') \leq MinOCT(G)$, the solution S achieves the claimed ratio. □

CHAPTER

6

BENCHMARKING TENSOR NETWORK CONTRACTION SEQUENCE ALGORITHMS

6.1 Introduction

Tensor network factorizations provide a framework for controlled approximation that exponentially reduces the memory required to simulate a variety of quantum many-body systems [Whi92; Vid06] and circuits [Vid03; Mar05]. The tensors comprising the factorization are placed on the vertices of a graph, one appropriate to the geometry under consideration, and are contracted along the edges as needed to compute physical observables [Whi92].

Since their early usage as the density matrix renormalization group description for gapped spin chains [Whi92; Bia17], tensor networks have been adapted and reformulated to also describe 2D area-law states [Eis10; Sch11], critical systems [Vid06], lattice gauge theories [Ric14; Pic16], AdS/CFT duality [Pas15], and open quantum systems [Ver04; Wer16]. In addition to describing the above physical phenomena, tensor networks can also describe satisfiability problems [Bia15; Bia17] and quantum computing simulations [Vid03; Mar05] can be formulated as tensor contraction problems. In the case of the latter, simulations of quantum error correcting codes are leading to important insights into fault tolerant quantum computation [Fer14; Dar17]. Addi-

tionally, given the tremendous interest in validating increasing complex experimental quantum computations [Boi18], a flurry of simulations have recently appeared in which the underlying tensor network graph emerges from the structure of the algorithm being employed [Dum17; Dan19; Fri18; Ped17].

The overall descriptive power and algorithmic computational complexity, as formalized by the *contraction complexity* of a tensor network [Mar05], is determined by the tensor network construction's underlying graph structure. For some tensor network algorithms (e.g. the matrix product state formulation) the contraction complexity is fixed and well understood. However, the task of determining the contraction complexity in general, along with computing an optimal contraction sequence witnessing this complexity, is NP-complete [CC97].

Despite this daunting theoretical complexity, efficient methods exist in practice for obtaining both optimal and 'good enough' contraction sequences. Domain-specific approaches typically search the space of all possible sequences and apply heuristic pruning techniques to reduce the search space [CC97; Pfe14b; Fri18]. Effective algorithms in this area incorporate pruning rules proprietary to the target application's data (e.g., MERA networks [Pfe14b]), which limits their broader applicability. Another standard technique involves transforming the tensor network into a line graph, then computing a perfect elimination ordering and its treewidth, which can be translated into a contraction sequence and complexity for the original network, respectively [Mar05].

In practice, however, engineering issues prevent these methods from being applied effectively. Domain-specific approaches typically suffer from proprietary construction, each assuming a different representation of the tensor networks, and using different code languages, dependencies, and interfaces (often with little-to-no documentation). Additionally, these implementations are typically only tested on the data for which they were designed, providing no expectation for how they might perform and/or scale in different contexts. Treewidth-based approaches further suffer from the graph theory overhead needed to convert their (typical) output of tree decompositions into perfect elimination orderings for the line graph and then contraction sequences for the tensor network.

Our primary contribution to the literature is to provide an open source code framework (available at `github.com/TheoryInPractice/ConSequences`) for integrating all existing contraction sequence algorithms into a common interface designed for extendability and documented for accessibility; further, we have tabulated the performance of several leading contraction sequence algorithms. Our results provide quantum circuit simulation developers an extended benchmark for expected performance on circuits with varying structures and complexities.

We use container-based (Docker [Mer14]) wrappers for each contraction sequence algorithm, completely removing code dependency issues, and provide Python-based utilities for converting various input/output formats into standardized internal formats for consistency. We demon-

strate the utility of this software by reproducing two previous studies based on domain-specific algorithms, and extending them to include treewidth-based solvers in a broader set of experimental results. We find that modern treewidth solvers from the recent PACE 2017 coding challenge [Del18] are both faster and have more consistent run times than the domain-specific algorithms. This speed increase allows us to study larger datasets in both experiments, and provide more competitive comparisons of a tensor network simulator [Fri18] against Microsoft's LIQUi|> Hilbert space simulator [Wec14]. In particular, we show that contraction sequence algorithms are no longer the major bottleneck in tensor network simulations, and there is immediate value in work improving the scalability of downstream contraction code.

The paper is organized as follows. We begin with relevant definitions and an overview of related work in the Background, then describe the functionality of our code framework and considerations for use and extension in ConSequences: An Accessible, Extendable Framework. In the subsequent MERA Applications section, we reproduce a study by Pfeifer et al. [Pfe14b], evaluating their algorithm netcon alongside two treewidth algorithms from the PACE 2017 challenge (freetdi [Lar17] and meiji-e [Oht17]) on a dataset including multi-scale entanglement renormalization ansatz (MERA) networks [Vid06]. We extend this initial comparison on a larger corpus of MERA networks, pushing the limits of these exact contraction sequence solvers on a new benchmark. In the Applications with qTorch Simulator section, we reproduce a study by Fried et al. [Fri18], evaluating another treewidth-based solver (quickbb [Gog04]) against freetdi and meiji-e on quantum circuits formulated with Farhi et al.'s quantum approximate optimization algorithm (QAOA) for MC on r-regular graphs [Far14]. In addition to contraction sequence comparisons, we simulate the tensor network with qTorch [Fri18], noting the correlation between simulation time and contraction complexity, and providing an updated comparison with Microsoft's LIQUi|> simulator [Wec14]. We conclude with a summary and future work.

6.2 Background

This chapter relies on the definitions of contraction complexity and treewidth previous introduced in the Background. Namely, the contraction complexity on a tensor network is equivalent to the treewidth computed on the line graph of the tensor network. All conversions here are constructive, therefore a solution found on one graph can be mapped to the other form and vice-versa. In this paper we use treewidth solvers for computing tensor network contractions, but future work may be to use domain-specific contraction sequence algorithms for computing tree decompositions.

Rapid advances have been made in treewidth solvers in recent years, in large part to the Parameterized Algorithms and Computational Experiments (PACE) Challenge [Del17; Del18]. Previous algorithms with practical implementations (such as quickbb [Gog04]) are based on

searching the space of elimination orderings and given the connection between contraction sequences and elimination orderings [Mar05], share a strong resemblance to typical domain-specific algorithms [CC97; Pfe14b]. However, recent work in separator-based treewidth algorithms has begun to dominate modern benchmarks. The classic Arnborg, Corniel, and Proskurowski dynamic programming algorithm [Arn87], reformulated as a positive-instance dynamic programming (PID) algorithm, has produced the winners of both the PACE 2016 (a Java implementation by Tamaki) and PACE 2017 (a C++ implementation by Larisch and Salfelder, freetdi) challenges [Del17; Del18; Lar17]. The 2017 challenge also saw a better scaling implementation (meiji-e [Oht17]) based off of a PID-reformulation of the improved dynamic programming algorithm by Bouchitté and Todinca [Bou01].

A *tensor network* is a graph where each vertex represents a tensor, and each edge represents a physical bond shared between tensors. The simulation of tensor networks requires contracting each tensor one-by-one until only one remains, the cost of which is determined by the *contraction complexity* previously mentioned in the structures subsection.

One class of tensor networks that we examine is the multi-scale entanglement renormalization ansatz (MERA). Given that contraction sequence algorithms utilize only the structure of the underlying graph in these networks, we restrict our presentation here to the important structural notions (visualized with a 1D binary MERA in Figure 6.1). We direct the interested reader to [Eve09] for a rigorous description beyond the graph structure.

Fundamentally, MERA is a scheme for mapping a lattice of *operator sites* onto a coarser lattice. This mapping is expressed in terms of *coarsening layers*. The lattices on which MERA acts have an inherent dimension, which we denote d; for simplicity we only consider examples in 1- and 2-dimensions in this paper. The most detailed lattice ($\mathscr{L}_0$) contains all sites for operators, and lattice $\mathscr{L}_1$ is produced after one level of coarsening. A coarsening level consists of a layer of unitaries followed by a layer of isometries. To disentangle the sites, in the MERAs we consider, *unitary* tensors take in 2^d wires (edges) and output 2^d wires for a lattice of dimension d. For the coarsening layer, $k:1$ *isometry* tensors take in k wires and output one wire. In total, going from lattice $\mathscr{L}_i$ with s sites to $\mathscr{L}_{i+1}$ requires $\frac{s}{2}$ unitary tensors, $\frac{s}{2}$ isometry tensors, and produces a new lattice with $\frac{s}{k}$ sites. This structure is then reflected for negative lattice levels, and a wire connects the top and bottom level interface tensors. Once this MERA graph is defined, operators are placed on lattice sites in $\mathscr{L}_0$ and the *causal cone* is computed by including the operators and any tensor that lies on an ascending (descending) path to the upper (lower) interface tensor (Figure 6.1). Once the causal cone is computed, every tensor not included in the cone is removed (by unitarity and properties of the isometries), and wires are added from a tensor to its dual mirror such that all tensors have the requisite number of wires.

Another tensor network model comes from the quantum approximation optimization algorithm (QAOA) [Far14], a hybrid classical-quantum algorithm for utilizing near-term (~ 100

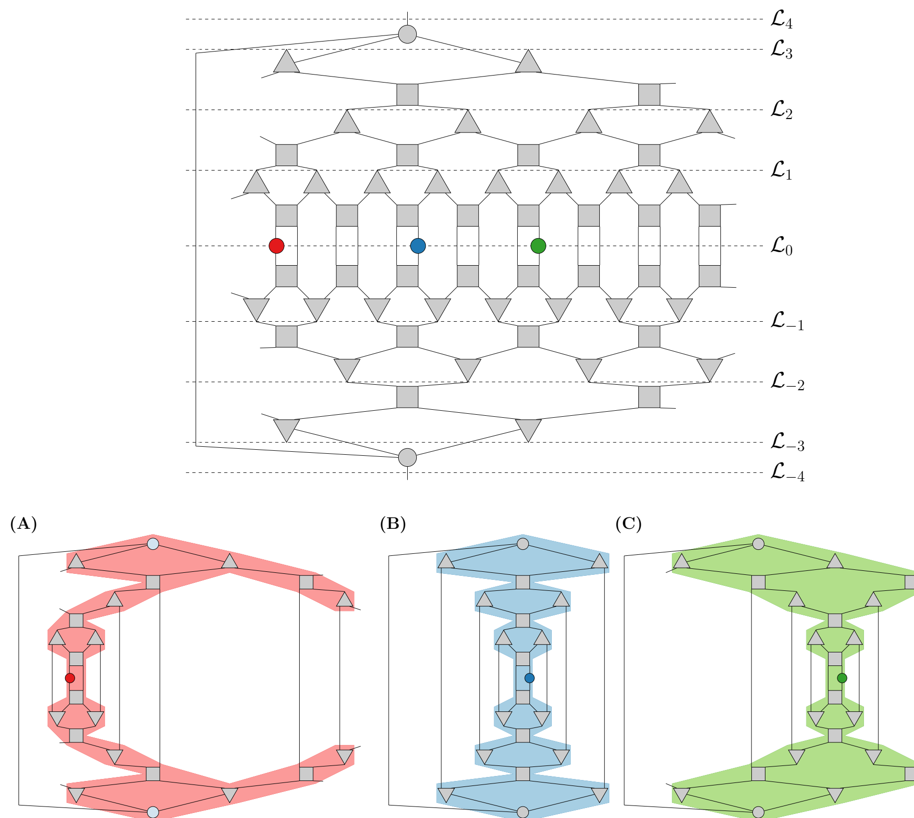


Figure 6.1 (Top) A 1D binary MERA with a 16-site lattice and 3 levels of coarsening; three operator placements are highlighted (red, blue, green). (Bottom) Causal cones and final tensor networks for each of the three highlighted operators. Note that the tensor networks for the left-most (red) and right-most (green) operators are isomorphic to one another, but structurally distinct from the middle (blue) operator's network.

qubit) quantum computers. While applicable to generic satisfiability problems, we restrict ourselves to the MC optimization problem on r-regular graphs. Notably, when QAOA is applied to the MC problem, the structure of the input graph is reflected in the quantum circuit. Interested readers should refer to Farhi et al.'s formulation of QAOA [Far14] for a theoretical treatment, or the qTorch source code [Fri18] for a practical example.

6.3 ConSequences: An Accessible, Extendable Framework

One issue preventing widespread experimentation with (and adoption of) contraction sequence algorithms was the practical problem of installing the software and managing software dependencies. Of the algorithms presented in this paper, one is interfaced with MATLAB and uses C extensions for computationally-difficult sections (netcon), one is written in Java (meiji-e), two

are written in C++ (freetdi, qTorch), and one is *only* distributed as a binary executable for Linux (quickbb). Often these implementations were written as a proof of concept with little-to-no documentation, especially for the library dependencies needed to compile the code.

Additionally, once the code is compiled, each solver has proprietary input and output format. Algorithms from the treewidth literature may require the input graph to have particular vertex labels, and typically output a tree decomposition or an elimination ordering. Algorithms from the contraction sequence literature may require the input as a quantum circuit in Quantum Assembly format or as a tensor network, and the contraction sequence output may be an ordering of edges or a sequence of contractions that automatically removes resulting self-loops.

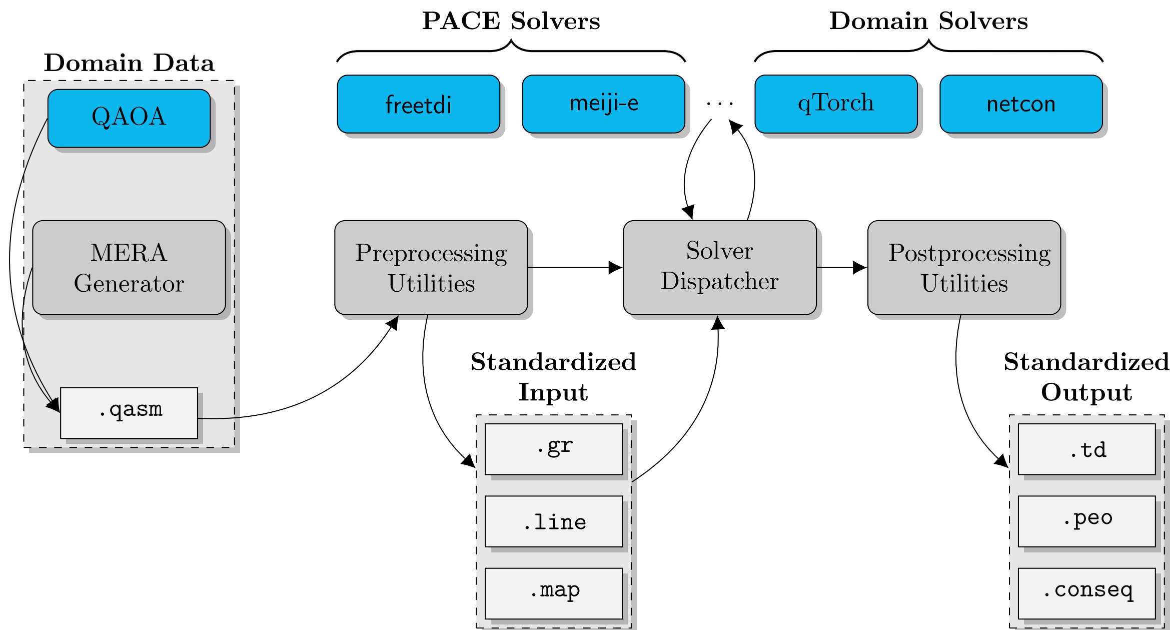


Figure 6.2 Visualization of the ConSequences pipeline. Externally-generated domain data is parsed into standardized graph formats with the pre-processing utility. The solver dispatcher then allows the user to compute contraction sequences (or their equivalent, e.g. tree decompositions) using external algorithms in Docker containers, then executes a post-processing utility to output standardized formulations of a contraction sequence.

Addressing both problems at once, we provide an open source framework ConSequences (Figure 6.2) for running contraction sequence algorithms, designed to be both *accessible* and *extendable*. The code is available at `github.com/TheoryInPractice/ConSequences`, complete with documentation for using existing code and tutorials for extending the functionality. At the core of this framework is a pipeline for pre-processing input, dispatching solvers, and post-

processing output. The pre-processing utilities take tensor networks in formats such as Quantum Assembly and various graph formats, then generate graph files in a standardized format. The solver dispatcher manages calls to contraction sequence algorithms, and includes functionality such as parallelism, managing seeds, and handling timeouts. The post-processing utilities then take output from these solvers and generate a full set of output (including tree decompositions, perfect elimination orderings, and contraction sequences), allowing various aspects of the output to be reported and analyzed.

To make the framework *accessible*, the central pipeline is implemented as lightweight Python files, and individual contraction sequence algorithms are wrapped inside of Docker [Mer14] images. Docker images are similar to virtual machines in that they abstract away all dependency issues, but have the distinct advantage of very little CPU and memory footprint on a native Linux system. Users need only install Python and Docker to run the code on Windows, MacOS or Linux; these steps are straightforward on Linux with instructions for new users.

To make the framework *extendable*, we provide Docker templates for data generators (e.g., MERA networks) and contraction sequence algorithms. These templates are accompanied by tutorials which guide new users through wrapping up their code in Docker and interacting with our existing structure properly.

All experiments were run on three identical workstations, each with a single Xeon E5-2623 v3 processor (8 threads with a 3.0GHz base clock and 10MB cache) and 64GB system memory. Contraction sequence algorithms use a single thread and LIQUi|> and qTorch simulations with all threads. Experiments were run one at a time on the workstations, preventing noise from non-uniform cache usage between competing jobs. These workstations ran Fedora 27 with Docker 18.03.1-ce and Python 3.6.5. Algorithm-dependent software requirements (e.g., gcc, MATLAB) are fixed per algorithm in its Docker image wrapper; see the code repository for details.

6.4 MERA Applications

In this section we compare exact treewidth solvers to Pfeifer et al.'s netcon algorithm [Pfe14b], on data from MERA networks. A small corpus of datasets from [Pfe14b] is initially considered, in which case optimal contraction sequences are found within four seconds by both treewidth solvers. To analyze algorithmic scaling with an extended benchmark, we generate 1- and 2-dimensional MERA networks with all possible placements of 1- and 2-operators, where we additionally observe that the meiji-e treewidth solver scales better than freetdi when networks become dense and increase in contraction complexity.

6.4.1 Initial Comparison on Pfeifer Benchmark

The netcon implementation was developed as the contraction sequence algorithm for a simulation toolset written for MATLAB [Pfe14a; Eve13; Pfe14b]. In the vein of previous approaches [CC97] such as depth-first search and dynamic programming, netcon's core subroutine is a breadth-first search (BFS) over the solution space of all possible contraction sequences. To trim down this exponentially-sized space, the authors introduce two pruning methods for reducing the search space at each step of the BFS: first, if a contraction would cost more than a user-defined threshold, this contraction will not be considered; second, the authors provide a list of criteria for when outer product contractions should not be made. This algorithm is exact (i.e., it finds optimal contraction complexity), but its run time depends heuristically on the effectiveness of the pruning techniques to a particular network's search space. Provided as a MATLAB package, the core subroutines in netcon are implemented in external C code for efficiency. The authors of [Pfe14b] evaluate their pruning heuristics on seven networks, including Tree Tensor Networks (TTN), Time-Evolution Block Decimation (TEBD), and MERA networks ranging from five to 27 tensors. In this previous work, the authors found that the the pruning techniques resulted in faster results on all networks, with the largest network requiring 36 seconds.

Table 6.1 Run times for each contraction sequence algorithm when executed on tensor network datasets from Pfeifer et al. [Pfe14b]. For each tensor network, the number of tensors ($|V|$), edges ($|E|$), and optimal contraction complexity (cc) are reported.

Instance				**Run Time (sec)**		
Name	$\|V\|$	$\|E\|$	cc	freetdi	meiji-e	netcon
3:1 1D TTN	5	28	6	0.009	0.338	0.002
TEBD	6	24	3	0.002	0.297	0.002
3:1 1D MERA	7	45	8	0.005	0.381	0.007
9:1 2D TTN	9	38	12	0.004	0.459	0.004
2:1 1D MERA	11	34	9	0.005	0.386	0.016
9:1 2D MERA	19	62	16	0.011	0.669	0.051
4:1 2D MERA	27	55	26	2.944	3.534	24.415

We reproduce this experiment on all seven networks (provided in [Pfe14b]). For netcon we use the MATLAB interface with an external C package, using all optimizations as specified in the accompanying code [Pfe14b]. We compare against two exact PACE algorithms, freetdi and meiji-e. Refer to the ConSequences section for workstation specifications. As seen in Table 6.1, both netcon and freetdi require on the order of 0.001 seconds on the first four networks, whereas

meiji-e was two orders of magnitude slower. On the three larger networks, however, freetdi was fastest, with both PACE algorithms finishing within four seconds on the largest network. This last data point is of particular interest because it hints at scalability differences between these algorithms. Whereas meiji-e's run time was 5× slower on the 4:1 2D MERA compared to the 9:1 2D MERA, freetdi increased to over 250× slower and netcon over 450× slower.

6.4.2 Extended Benchmark on Large MERA Networks

To test scalability further, we compared the algorithms' performance on a larger corpus of MERA networks. As seen in Figure 6.1, the iterative layers of isometries and unitaries in MERA networks allow one to easily generate underlying graphs given a unitary and isometry specification. We provide a MERA generator in our code for 1D lattices with binary isometries and 2D lattices with 4:1 isometries. This generator takes as input the number of coarsening levels and whether the top level of isometries should connect to a common tensor in the style of [Eve09]. Once a MERA network is generated from these parameters, the locations of operators to be evaluated are chosen, a causal cone is computed, and the network is reduced down to the final tensor network by simplifying the network outside the causal cone.

Notably, given a fixed set of parameters needed to generator the MERA network, several choices of operator inputs can result in isomorphic (i.e., identical up to relabeling) networks. As seen in Figure. 6.1, placing one operator at various sites results in both isomorphic and non-isomorphic networks, so care must be taken when generating representative graphs.

In our extended comparison, we generate 1D binary MERA and 2D 4:1 MERA with one and two operator placements. For one operator, we generate a MERA network for every possible operator placement, then compute the unique networks up to isomorphism. For two operators, we fix an operator at the first position (0 for 1D and (0,0) for 2D), then range the second operator over all possible positions; again we extract the unique graphs up to isomorphism. Table 6.2 summarizes the number of networks up to isomorphism and details some of their nuances. For example, for a 1D 2:1 MERA with six coarsening levels and two operator placements, the network will have between 40 and 86 vertices based on operator placement (see the dark-blue highlighted row in the summary table). In this case, over all placements, $|\mathscr{M}| = 63$ networks were generated, but only $|\widehat{\mathscr{M}}| = 48$ were unique. Looking into these unique networks even further (left table), we find that only one network had 40 vertices, and over half of the unique graphs had over 80 vertices. While these networks may vary in the extreme cases, on average these networks are fairly predictable.

Figure 6.3 and Figure 6.4 visualize the results of our extended experiment running contraction sequences algorithms on larger MERA networks using ConSequences. In Figure 6.3, networks are binned by the number of qubits they can support for a quantum simulation (which is determined

Table 6.2 Summary of extended MERA benchmark data. (Center) For each lattice type (1D binary or 2D 4-ary) and number of operators, as the number of levels ℓ varies, we report the number of vertices $|V|$ in the resulting networks, the total number of networks $|\mathscr{M}|$, and the number of unique networks up to isomorphism $|\widehat{\mathscr{M}}|$. (Left, Right) These detailed tables each expand a row from the summary table, specifying the number of networks produced (total $|\mathscr{S}|$ and up to isomorphism $|\widehat{\mathscr{S}}|$), for each pair of values for the number of vertices $|V|$ and number of edges $|E|$. Note that sum of the $|\mathscr{S}|$'s in a detailed table sums to the corresponding $|\mathscr{M}|$ in the summary table, and likewise for $|\widehat{\mathscr{S}}|$ and $|\widehat{\mathscr{M}}|$.

1D 2:1 MERA

$\ell = 6$, 2 oper.

$\|V\|$	$\|E\|$	$\|\mathscr{S}\|$	$\|\widehat{\mathscr{S}}\|$
40	65	1	1
62	104	2	1
64	107	2	1
66	110	2	1
68	113	2	1
68	114	2	1
70	116	2	1
70	117	2	1
72	120	2	1
74	123	2	1
74	124	4	3
76	127	4	3
78	130	4	3
80	134	8	7
82	137	8	7
86	144	16	15

ℓ	$\|V\|$	$\|\mathscr{M}\|$	$\|\widehat{\mathscr{M}}\|$
1D 2:1 MERA, 1 oper.			
2	15-17	4	2
3	21-27	8	3
4	27-37	16	8
5	33-47	32	21
6	39-57	64	50
7	45-67	128	111
8	51-77	256	236
9	57-87	512	489
10	63-97	1024	998
1D 2:1 MERA, 2 oper.			
2	16-22	3	2
3	22-38	7	4
4	28-54	15	9
5	34-70	31	21
6	40-86	63	48
7	46-102	127	106
8	52-118	255	227
9	58-134	511	475
10	64-150	1023	978
2D 4:1 MERA, 1 oper.			
2	23-25	12	2
3	29-55	68	7
4	43-81	256	122
2D 4:1 MERA, 2 oper.			
2	29-38	9	3
3	40-84	88	40
4	81-132	223	134

2D 4:1 MERA

$\ell = 4$, 2 oper.

$\|V\|$	$\|E\|$	$\|\mathscr{S}\|$	$\|\widehat{\mathscr{S}}\|$
88	259	4	1
90	264	4	1
104	305	16	9
108	315	8	1
110	320	4	1
110	323	16	9
112	331	8	5
114	333	16	9
116	341	16	13
118	346	4	1
120	351	28	25
120	354	20	13
122	359	8	1
122	362	8	5
124	367	4	1
126	372	40	33
130	385	8	1
132	390	4	1
132	393	6	3

by the number of possible operator sites) and run time is reported in seconds. For the networks that were included, a timeout of 20 minutes was used; if no algorithm could solve an instance within 20 minutes then the network was not included as a datapoint. From this perspective, netcon appears slower than both treewidth-based solvers, and freetdi generally dominates meiji-e in 1D MERAs. However, on 2D MERAs with one operator, all algorithms seem roughly equal for 16-qubits and 64-qubits, with meiji-e pulling slightly ahead on 256-qubits. The improved scaling of meiji-e is reflected in 2D MERAs with two operators, although here we begin to hit the limit of exact algorithms with a 20 minute timeout.

In Figure 6.4, networks are binned based on their contraction complexity, then run times are reported for each algorithm. From this perspective we find a stipulation for freetdi's dominance on 1D MERAs: the contraction complexity never exceeded 9. Indeed, freetdi performed well on 2D MERAs until the optimal contraction complexity reaches 18, at which point meiji-e starts to outperform both algorithms. These experiments depict that finding the optimal contraction sequence within 20 minutes becomes problematic when the contraction complexity reaches the mid-twenties, which may be useful for predicting when heuristics should be used.

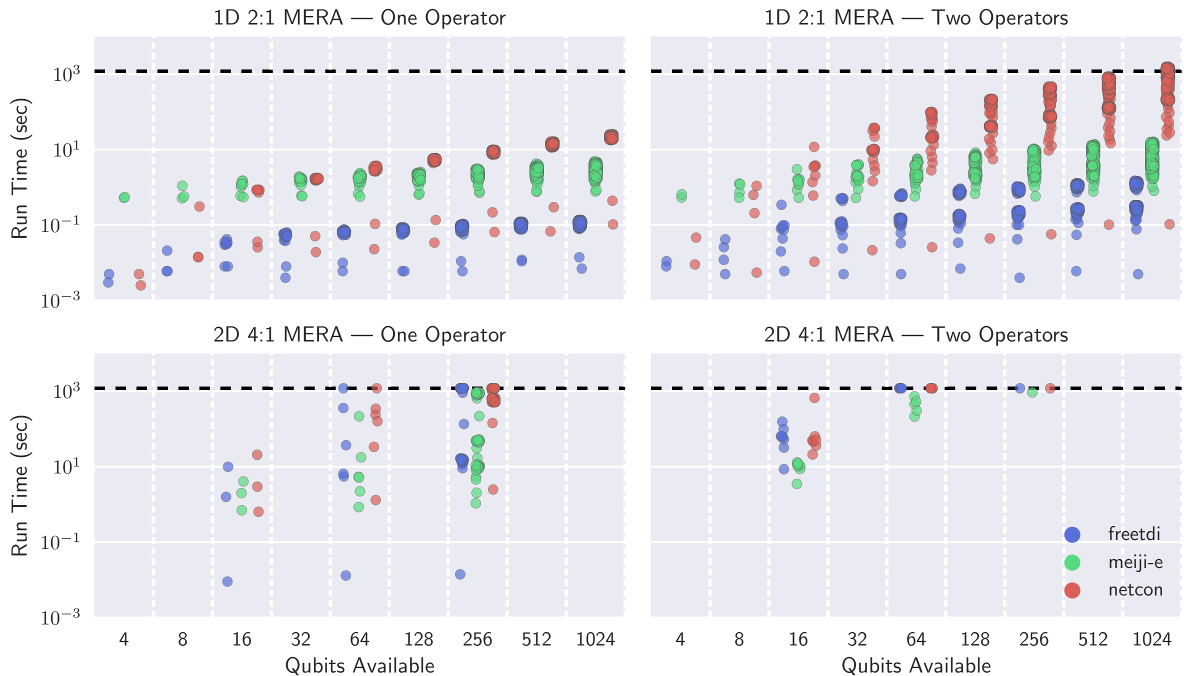


Figure 6.3 Run times for the contraction sequence algorithms on select MERA networks, binned by number of qubits possible (number of $\mathscr{L}_0$ sites). All algorithms are timed out at 20 minutes (horizontal dashed line), and a network that remained unsolved by every algorithm is not included. 2D MERA with one operator had 48 of 131 networks that did not finish, and the two operator networks had 193 of 207 that did not finish.

We note that while the experiment from [Pfe14b] contained a graph with contraction complexity of 26, this network only had 27 tensors and 55 edges, whereas our 2D 4:1 MERA networks with contraction complexity 26 could have as many as 393 edges. This fact implies that search-space–based approaches need pruning techniques that prune exponentially-many sequences in the number of network edges, otherwise the run time will scale without the optimal contraction complexity necessarily increasing.

6.5 Applications with qTorch Simulator

In this section we compare exact treewidth solvers from PACE with the quickbb solver used in Fried et al.'s qTorch tensor network simulator [Fri18]. This comparison is run on quantum circuits constructed with the QAOA method for solving MC on r-regular graphs. We find that exact contraction sequences on these graphs can be computed in less time than needed to execute the tensor contractions, allowing us to discuss the total simulation time without using an untimed preprocessing step. Rerunning the comparison with Microsoft's LIQUi|> simulator

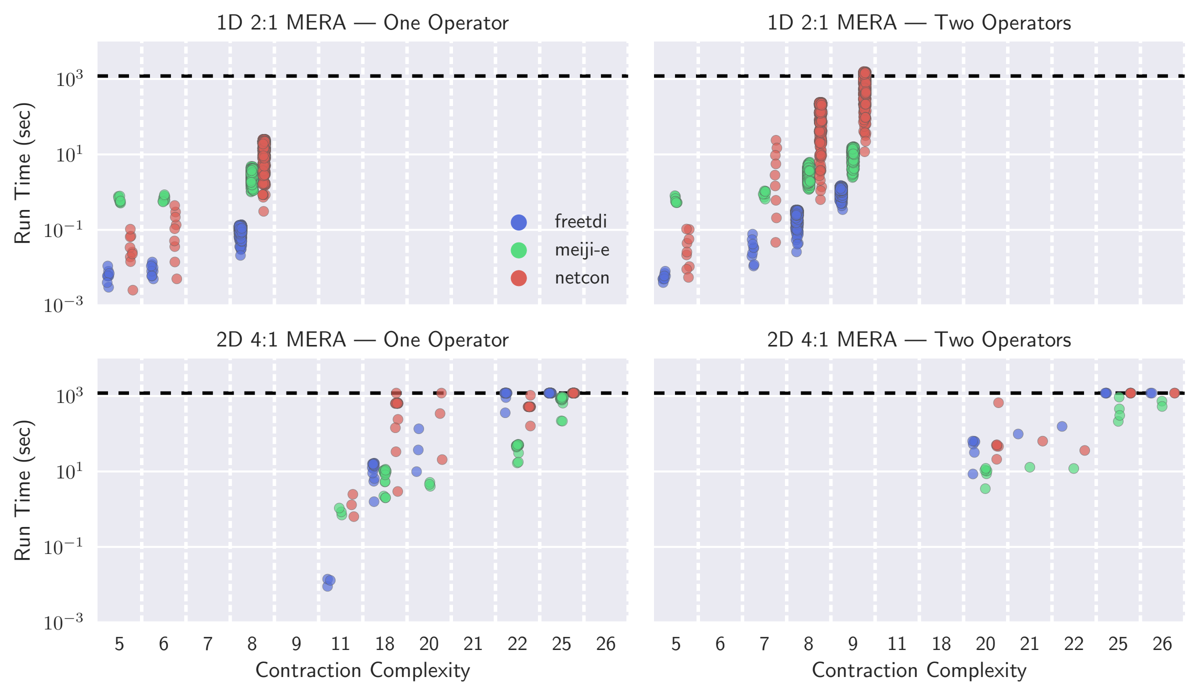


Figure 6.4 Run times for the three algorithms on select MERA networks, binned by optimal contraction complexity. All algorithms are timed out at 20 minutes (horizontal dashed line), and a network that remained unsolved by every algorithm is not included. 2D MERA with one operator had 48 of 131 networks that did not finish, and the two operator networks had 193 of 207 that did not finish.

from [Fri18], we find that tensor networks are competitive against state-of-the-art simulators, allowing speedups when little information is needed to represent the network (i.e., on sparse graphs), and a potential solution for the 22-qubit limit currently built into LIQUi|>.

Table 6.3 Exact contraction complexities found using freetdi and meiji-e on QAOA circuits for computing MaxCut on r-regular graphs. 25 random regular graphs are generated at each $r, |V|$ level using NetworkX, and algorithms were timed out at 15 minutes. Timed out values were dropped from the data, resulting in less than 25 Samples for some parameter values.

Network		Optimal Contraction Complexity					
r	$\|V\|$	**Samples**	**Mean**	**S.D.**	**Min**	**50%**	**Max**
3	10	25	5.0	0.6	4	5	6
	14	25	5.2	0.6	4	5	6
	18	25	6.0	0.8	5	6	7
	22	25	6.3	0.9	5	6	8
	26	25	6.7	0.7	6	7	8
	30	25	7.8	0.6	7	8	9
4	10	25	6.5	0.5	6	7	7
	14	25	7.7	0.8	6	8	9
	18	25	8.7	0.7	8	9	10
	22	25	10.2	0.8	8	10	11
	26	24	11.2	1.2	9	12	13
	30	5	12.0	0.7	11	12	13
5	10	25	7.7	0.6	6	8	9
	14	25	9.6	0.7	8	10	11
	18	25	11.3	0.7	10	11	13
	22	23	12.7	0.8	11	13	14
	26	1	12.0	NaN	12	12	12

6.5.1 Computing Contraction Complexity

Before comparing total simulation times, we evaluate runtimes required to find optimal contraction sequences. Because non-optimal contraction sequences lead to exponentially-slower downstream simulation times, we are especially interested in exploring the limits of exact solvers.

The data for this experiment comes from qTorch's QAOA quantum circuit constructor, which computes MC on a specified (arbitrary) graph. Reproducing circuits similar to the original qTorch experiments, we use r-regular graphs for $r \in \{3, 4, 5\}$, generated with NetworkX and seeded for easy replicability.

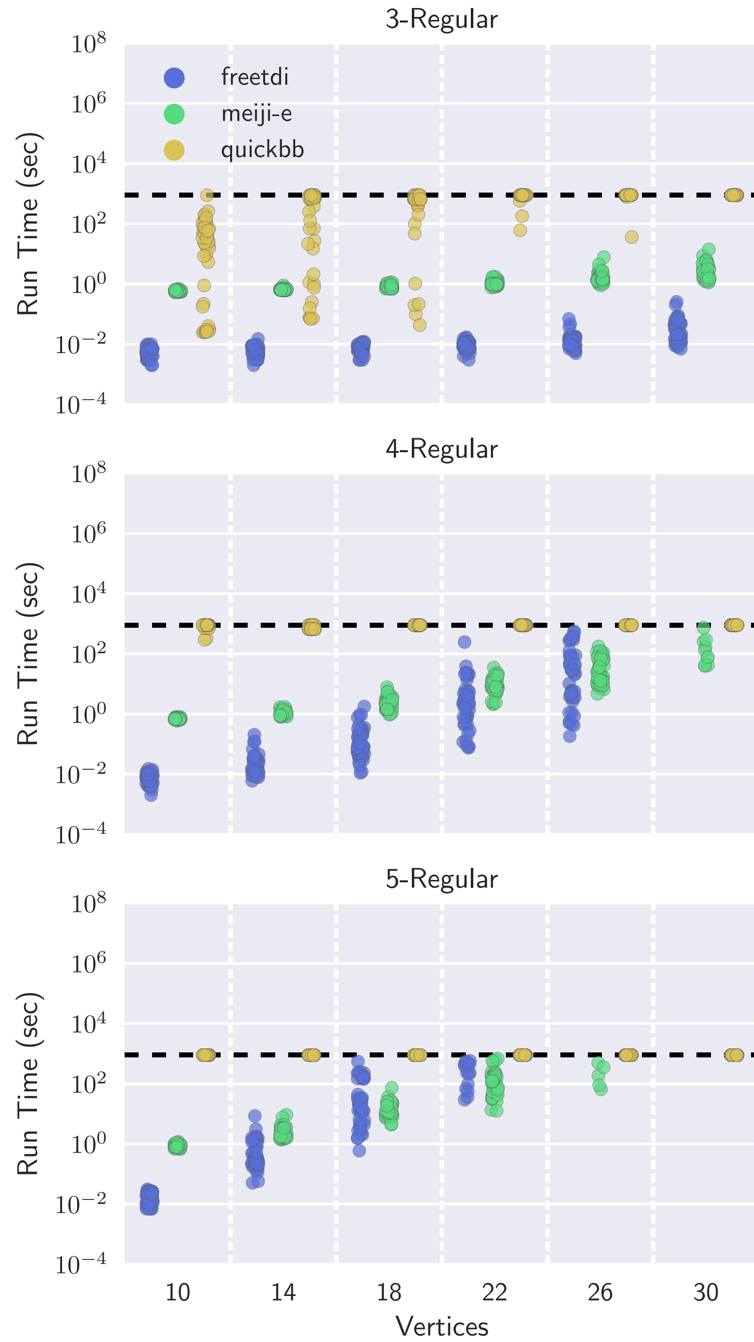


Figure 6.5 Run times for freetdi, meiji-e, and quickbb on QAOA circuits for computing MaxCut on r-regular graphs. 25 random regular graphs are generated at each $r, |V|$ level using NetworkX, and algorithms were timed out at 15 minutes (horizontal line).

Figure 6.5 visualizes the results from using 25 random graphs for each $(r, |V|)$ pair and a 15 minute timeout; Table 6.3 provides further information relating $(r, |V|)$ pairs with their optimal contraction complexity. Similar to the sparse MERA graphs, we find that freetdi dominates run times and is up to five orders of magnitude faster than quickbb on the smallest graphs. meiji-e again scales better than freetdi, allowing it to compute optimal contraction sequences for several networks within the timeout that other algorithms could not finish. Notably, quickbb is an anytime algorithm that finds increasingly better perfect elimination orderings, so it always provides a (potentially non-optimal) solution when given a timeout. Similar behavior may be adapted from heuristic versions of PACE submissions (e.g., meiji-e), but this functionality is left as future work.

6.5.2 Simulation Run Times

Computing downstream simulation times using qTorch, we first evaluate how simulation time correlates with contraction complexity. This comparison enables us to quantify how downstream runtime is impacted by the contraction complexity in practice (theoretical analysis predicts a exponential increase due to the size of the merged tensors, but it is possible that some downstream code mitigates this impact). Further, it provides for a finer-grained comparison of the quality of contraction sequences produced by freetdi and meiji-e, as measured by resulting simulation run time. Although both algorithms produce sequences with the same contraction complexity (and thus have the same leading-order term in the simulation time complexity), this term may occur during the contraction sequence between 1 and $|V|$ times. By examining raw simulation times, we may infer that one algorithm tends towards 'higher quality' contraction sequences than another.

As shown in Figure 6.6, simulation time indeed has an exponential dependence on the contraction complexity. Additionally, the contraction sequences produced by freetdi and meiji-e appear to be of comparable quality over the same corpus of networks. We observe that the range of run times for a given contraction complexity is nearly always an order of magnitude, meaning that the variance in run time scales proportional to the total run time. This observation may be useful for reliably predicting simulation time based on a network's known contraction complexity, which may be useful for optimizing future simulators.

In our final experiment, we compare qTorch paired with optimal contraction sequences to Microsoft's LIQUi|> simulator. Refer to the ConSequences section for workstation specifications. Previous comparisons to LIQUi|> included non-optimal contraction sequences found by quickbb, which may have caused the downstream simulation to be exponentially more expensive. Additionally, the experiments in [Fri18] were structured so that quickbb was run for 3000 seconds per network ahead of time, which resulted in impractical total run times for lower levels of regularity.

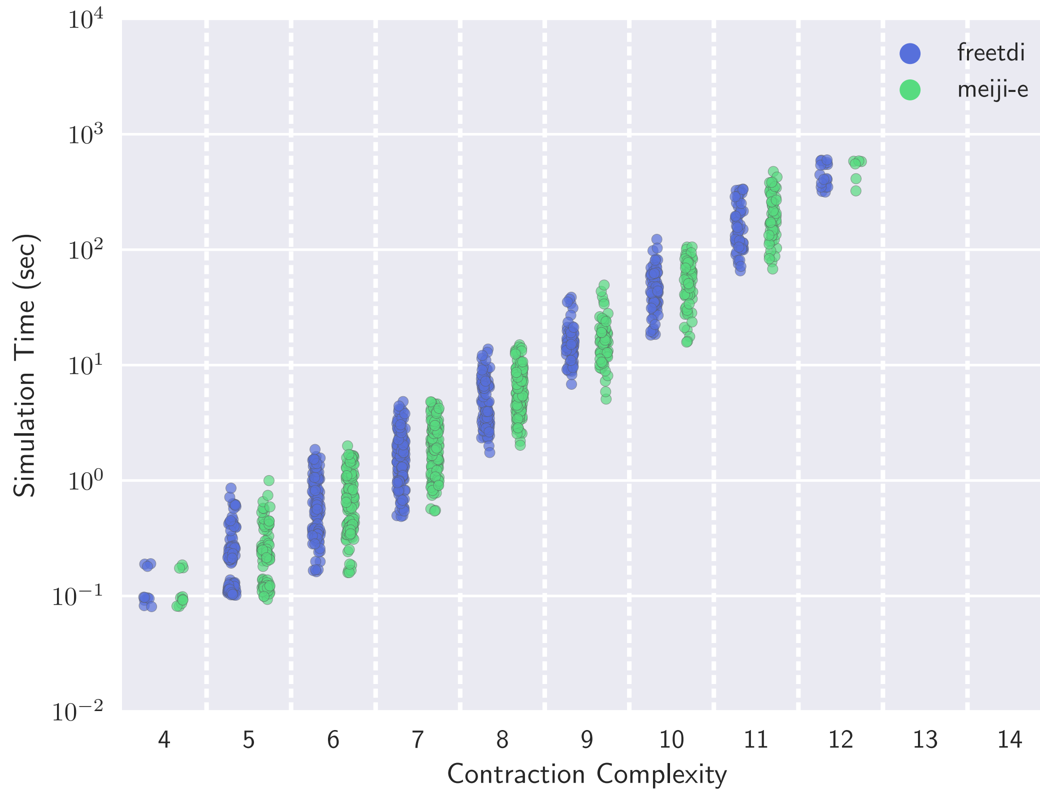


Figure 6.6 Simulation time is tightly correlated with the contraction complexity of a network. While exact algorithms freetdi and meiji-e may generate different tree decompositions and thus contraction sequences with the same treewidth, the differences have little impact on simulation times.

Results for this experiment are visualized in Figure 6.7. Aligning with previous conclusions in [Fri18], both 3-regular and 4-regular graphs are more quickly simulated on tensor networks than LIQUi|>. We additionally find that tensor networks can scale beyond the 22-qubit limit imposed on LIQUi|> (supposedly for exponential system memory usage). Even with 5-regular graphs, qTorch remains competitive as an alternative simulator.

Comparing Figure 6.5 with Figure 6.7, it is also clear that computing the contraction complexity is no longer the primary bottleneck. Instead, efforts should now be directed to improving the contraction simulation times.

6.6 Conclusion

In summary, ConSequences provides an open source, extendable platform for comparing contraction sequence algorithms for tensor networks. By packaging conversion utilities with containerized solvers, we remove both the theoretical and engineering difficulties preventing practitioners from running any contraction sequence solver on any tensor network. Additionally, we demonstrate the framework's applicability by reproducing and significantly extending several prior empirical evaluations. With MERA networks, we introduce a more extensive and difficult benchmark dataset which allows identification of solutions that will scale (e.g., freetdi on 1D MERA networks), subtler performance differences between algorithms (e.g., how meiji-e scales better for larger contraction complexities), and areas where new approaches are needed (e.g, 2D, 2-operator MERA, where 193 of 207 networks timed out). With qTorch on QAOA data, we were able to validate the exponential dependence of run time on optimal contraction complexity, and produce a total simulation time more representative of the full pipeline required for simulation. In doing so, we illuminate the urgent need for improved contraction times once the sequence is finalized.

For contraction sequence algorithms, several avenues offer promising future work. With large MERA networks we found that exact solvers for optimal contraction sequences had prohibitively high run times, a difficulty which may require using non-exact heuristics. Several treewidth-based algorithms have heuristic formulations in a different track of the PACE 2017 challenge [Del18], and our comparison against domain-specific algorithms suggests that these PACE submissions would be the natural starting point for an investigation into heuristics. Additionally, the use of heuristics involves a trade-off between finding a sequence with smaller complexity and the additional search time, which is not a problem for exact solvers. Exploring this trade-off in practical applications may be of interest.

Another consideration is that tensor network simulations are increasingly run on high-performance computing (HPC) systems. Modern HPC platforms scale-up performance with parallelism and heterogenous computing accelerators (such as GPUs and FPGAs). As seen in recent work [VDZ17], adapting treewidth's state-of-the-art dynamic programming algorithms to these platforms is a non-trivial task, but would be of significant potential impact to the quantum computing community.

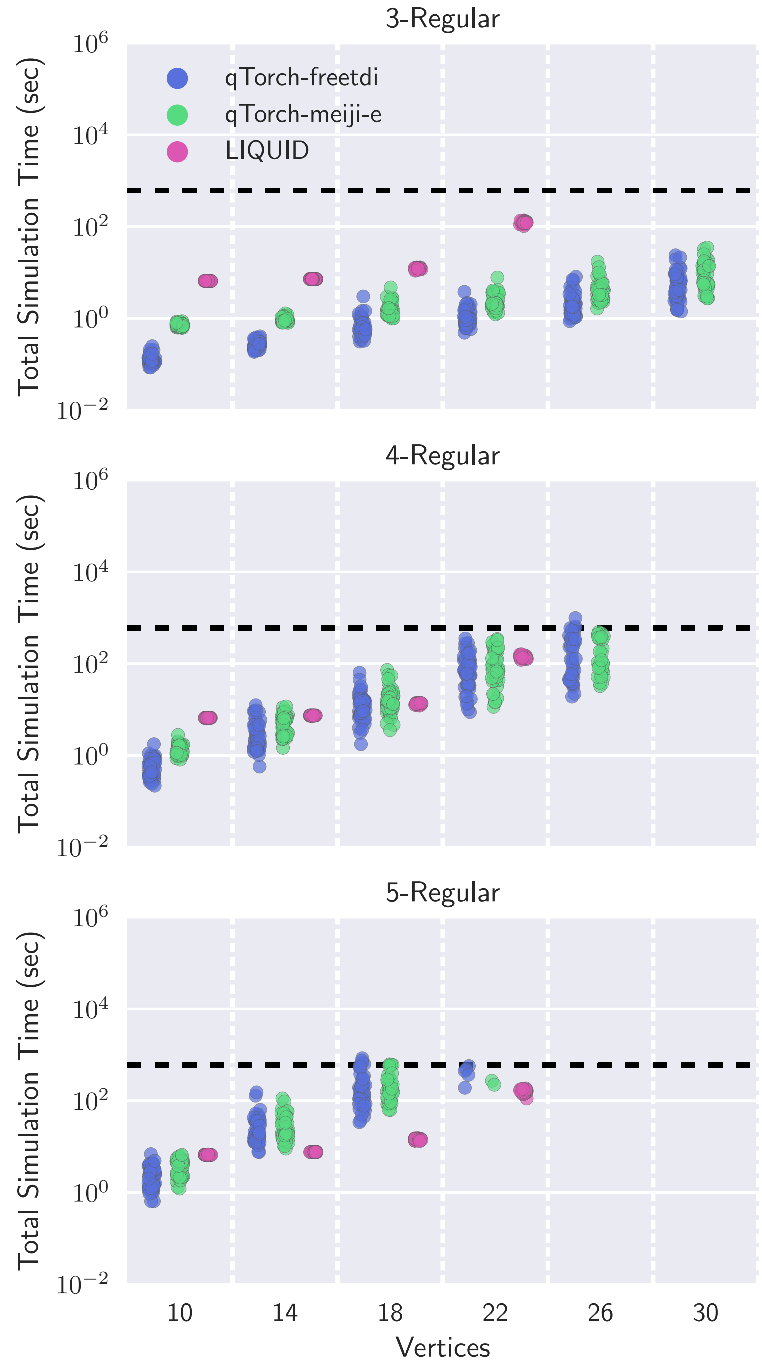


Figure 6.7 Simulation times of the qTorch tensor network simulator [Fri18] with contraction sequences produced by exact treewidth algorithms vs. Microsoft's LIQUID solver [Wec14]. Total simulation time includes both computation of the contraction sequence using ConSequences and tensor network simulation time using qTorch. A timeout of 900 seconds is used for computing the contraction sequence (horizontal dashed line), and a simulation is not run unless an optimal contraction sequence is found by at least one contraction sequence algorithm. LIQUID is limited to simulations up to 22 qubits.

CHAPTER

7

A TUNED TREEWIDTH SOLVER

7.1 Introduction

In the previous chapter we demonstrated how treewidth algorithms are not simply an alternative to domain-specific heuristics for computing contraction sequence, but rather clearly dominate the benchmarks. In this chapter we examine methods for tuning the PID algorithm [Tam19] for tensor networks. Specifically, we evaluate three potential optimizations:

1. Reduction routines to remove structure with treewidth 2 or less from the network instance.
2. A judicious choice of which safe separator heuristic routines to employ, based on the network structure.
3. Parameter tweaks to the *blocksieve* data structure to prevent unnecessary resizing.

Expanding our data corpus, we evaluate these improvements on the following corpora:

1. Google "Quantum Supremacy" datasets. Developed specifically to demonstrate quantum supremacy on near term (Google) quantum devices, these graphs are constructed to rapidly increase treewidth as the circuit increases in depth [Boi18].
2. UCCSD instances. Generated using the eXtreme-scale Accelerator programming framework (XACC) [McC20], these circuits are generated using the UCCSD operator.

3. PACE 2017 training, testing, and bonus datasets [Del18]. These graphs were used in the 2017 PACE treewidth competition, along with a bonus corpus of more difficult instances.
4. DIMACS graph coloring instances. The canonical corpus for evaluating competition solvers for various optimization problems, these graphs were used in the original report by Tamaki [Tam19].

Experimental results show that reduction routines are effective for removing both vertices and edges from the instances, with consistently (i.e. low standard deviation) smaller instances on both tensor network corpora. These removals led to at most a 20% speedup on both corpora, with the `uccsd` also having a lower bound of 10% speedup. Treewidth instances were similarly reduced, with the addition of up to 40% speedup on some `pace2017` bonus instances.

In the safe separator heuristic experiment we found that the `fill` heuristic was essential for `pace2017` data, but otherwise was not needed. Using the `degree` heuristic on `uccsd` results in a 10-90% speedup with no downside. The `dimacs` and `google-qs` datasets were generally unaffected by these heuristics.

Finally, the blocksieve data structure improvements made no discernible difference.

In future work, this algorithm would benefit most by a reimplementation that moved from heap-based objects to storing them on the stack. We find that real run time is at most $7.6\times$ the user run time, where the only parallelism in this Java implementation is the garbage collector. Shifting to stack-based memory allocations would not only reduce the garbage collector runtime, but provide the memory localization leveraged by modern CPU caches.

7.2 Background and Data

In this section we define the concepts needed for the Tamaki's PID algorithm, then overview the expanded tensor network and treewidth data corpus.

7.2.1 PID algorithm definitions

A vertex set C in G is a *component associated with* $S \subseteq G$ if C is a *connected component* of $G[V(G) \setminus S]$. The set of *components associated with* S is denoted $\mathscr{C}_G(S)$. A vertex set $S \subseteq V(G)$ is a *separator* of G if $|\mathscr{C}_G(S)| > |\mathscr{C}_G(\varnothing)|$. For a component C associated with a *separator* S of G, we say that C is a *full component* if $N_G(C) = S$ A separator S is a *minimal separator* if it has at least two *full components* associated with it

A *block* is a pair (S, C) where S is a *separator* and C is a component *associated with* S. A block (S, C) is *full* if C is a full component A graph G is *chordal* if every induced cycle on G has a length of exactly three. A graph H is a *minimal chordal completion* of G if H is chordal, $V(H) = V(G)$, $E(G) \subseteq E(H)$, and $E(H)$ is minimal

A tree-decomposition T of G is *canonical* if every bag of T is a potential maximal clique of G, and for every pair X, Y of adjacent bags in T, $X \cap Y$ is a *minimal separator* of G. A set $S \subseteq V(G)$ is *cliquish* if for every $x, y \in S$, either x and y are adjacent or there is some $C \in \mathscr{C}(S)$ such that $x, y \in N(C)$

A component $C \in \mathscr{C}(K)$ is *confined* to S if $N(C) \subseteq S$. A component C which is *unconfined* on S. Let $\text{unconf}(S, K)$ be the set of components associated with K which are *unconfined* to S

The *crib* of S with respect to K, $\text{crib}(S, K)$, is the union of $K \setminus S$ and all components associated with K that have neighborhoods intersecting $K \setminus S$, i.e. $(K \setminus S) \cup \bigcup_{C \in \text{unconf}(S,K)} C$

Given vertex sets $V, U \subseteq V(G)$, we say V *precedes* U if the minimum element of V is less than the minimum element of U under $<$. We denote this as $V \prec U$

A connected set C is *inbound* if there is some full block associated with $N(C) \prec C$. A full block $(N(C), C)$ is inbound if C is inbound A connected set C is *outbound* if it is not *inbound*. A full block $(N(C), C)$ is outbound if C is outbound

The *outlet* of a cliquish vertex set K, $\text{outlet}(K)$, is $\emptyset$ if there are no non-full components associated with K that are outbound. Otherwise, $\text{outlet}(K) = N(A)$ where A is a non-full component associated with K such that $N(A)$ is maximal. The *support* of a cliquish vertex set K, is defined as $\text{support}(K) = \text{unconf}(\text{outlet}(K), K)$.

A potential maximal clique Ω is *feasible* if $|\Omega| \leq k + 1$ and every $C \in \text{support}(\Omega)$ is feasible. A full block $(N(C), C)$ is an *i-block* if C is inbound and $|N(C)| \leq k$, and an *o-block* if C is outbound and $|N(C)| \leq k$. An i-block $(N(C), C)$ is *feasible* if C is feasible. An o-block $(N(C), C)$ is *feasible* if $N(C) = \bigcup_{A \in \mathscr{C}}$ for some set $\mathscr{C}$ of inbound components.

7.2.2 Data

Complementing the MERA and QAOA corpora studied in Chapter 6, we study two corpora from the treewidth community (Figure 7.1) and two from tensor networks (Figure 7.2).

The first dataset comes from the 2017 Parameterized Algorithms and Computational Experiments (PACE) challenge, which is derived from real-world data as specified in [Del18]. Intentionally constructed for a treewidth competition, we expect that these instances are generally immune to reduction routines and difficult. The data includes 100 graph instances used for training, 100 for competition evaluation, and then 100 more advanced instances for future work. The majority of the first 200 instances are easily solved in seconds, we judiciously sample from the remaining instances and from the 100 advanced instances.

The second treewidth dataset comes from the DIMACS graph coloring instances [DIM20]. Classically used as difficult instance for the graph vertex coloring problem, these instances are historically used as a difficult benchmark in various other graph optimization problems. In fact, the exact treewidth of several of these instances is unknown. Relevant to our current

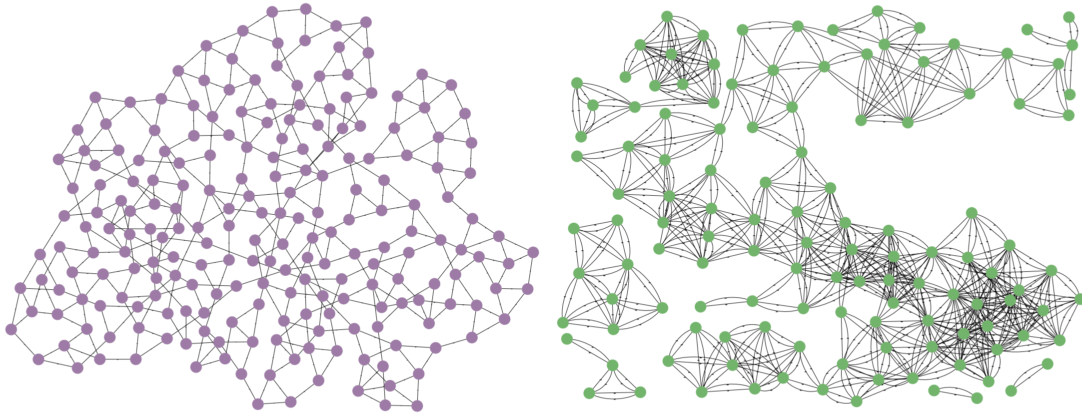


Figure 7.1 (Left) The instance `Promedas6011` from the Parameterized Algorithms and Computational Experiments (PACE) 2017 challenge. (Right) The instance `miles250` from the DIMACS graph coloring challenge corpus. Unlike the tensor networks, these graphs are more hereogenerous in structure.

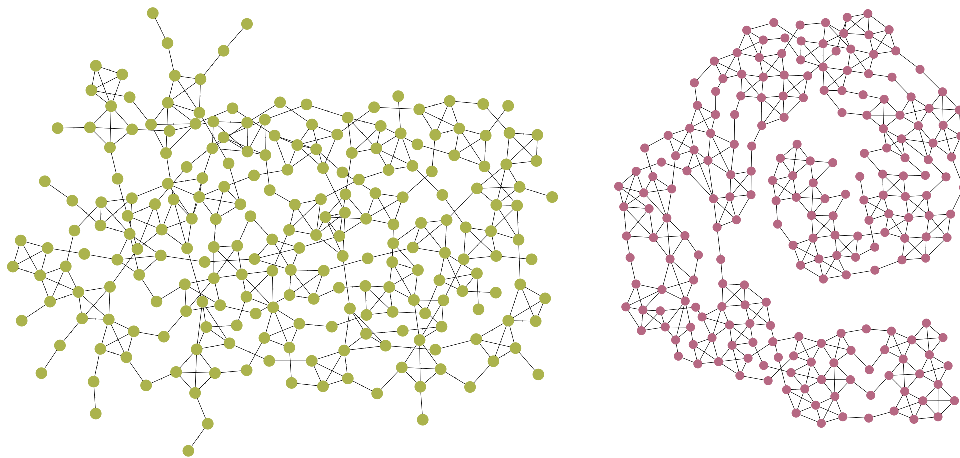


Figure 7.2 (Left) The line graph of a Google Quantum Supremacy graph, generated on a Bristlecone processor lattice with four rows and 24 clock cycles (i.e. circuit depth). (Right) The line graph of a Unitary Coupled Cluster Singles and Doubles (UCCSD) circuit with one fermion and four qubits. Note the prevalence of degree-1, degree-2, and cliques with four vertices; these are caused from the 1-input, 1-output and 2-input, 2-output quantum gates in the circuit.

experiments, these instances were evaluated in the Tamaki paper [Tam19].

The first tensor network dataset comes from Boixo et al. [Boi18], a set of quantum circuits constructed as a potential area where Google can show *quantum supremacy* (solving these problems on quantum hardware and showing that a classical solver is infeasible). While these instances will, by definition, scale towards large treewidth, they are nevertheless of current interest due to tensor networks' appeal as a classical simulation technique.

Finally, the second tensor network dataset comes from generating Unitary Coupled-Cluster Single and Double (UCCSD) circuit with the eXtreme-scale Accelerator programming framework (XACC) [McC20]. These circuits are most closely related to the MERA networks from Chapter 6 in construction and edge density.

As standard practice, the treewidth graph instances are sanitized by removing multiedges and relabeling vertices from 1 to n. The tensor networks are sanitized by computing a line graph transformation and relabeling vertices from 1 to n. The relabeling scripts are deterministic to maintain replicability from the raw, original datasets.

7.3 PID Treewidth Algorithm and Improvements

In this section we present Tamaki's PID algorithm [Tam19], which is included for completeness but not required for understanding the algorithm changes that we evaluate. The full pseudocode for the dynamic programming algorithm using these concepts is provided in Section 7.6. Additionally, we detail how to tune (a) the preprocessing routines, (b) the choice of safe separator heuristics, and (c) the blocksieve data structure.

7.3.1 A description of Tamaki's PID algorithm

Blocks

A concept that underpins the rest of the paper is the idea of *component association*. Simply put, a set of vertices C is associated with a set of vertices S if, when S is removed from the graph, C is a connected component. Related to this association are the concepts of *separators* and *minimal separators*. The intuitive definition suffices for a simple separator; if removing S results in more components than were originally in the graph, then S is a separator. A separator is *minimal* if there is no $v \in S$ such that S is still a separator when v is removed.

Combining these two concepts gives us the basic units on which the algorithm described in this paper operates, *blocks*. A block is defined as a pair (S, C), where S is a separator and C is a component associated with S. In the case where $S = N(C)$, then C is called a full component and the block $(S, C) = (N(C), C)$ is a full block.

Conceptually, a full block (S, C) can be thought of with C as a self contained "chunk" of

the graph, and S describes the minimal "cut" needed to separate C from the parent component.

Potential Maximal Cliques

Crucial to a characterization of treewidth described later in the paper, a set S is *cliquish* if, for every vertex $v, u \in S$, either v is connected to u or there is some path through a component associated with S which connects v and u. A graph G is *chordal* if there are no cycles larger than 3; a minimal chordal completion of a graph is a graph with edges added to remove all cycles larger than 3. A set Ω is a *potential maximal clique* if it is a clique in some minimal chordal completion of the graph. Additionally, if there is no full component associated with a cliquish set S, then S is a potential maximal clique. Ultimately, we will use potential maximal cliques to either (a) provide a counterexample to the claim that the treewidth of G is k, or (b) provide a certificate (e.g. tree decomposition) of width k. Past work shows that if Ω is a potential maximal clique with a minimal separator T, then there is a unique component C_T associated with T such that C_T contains $\Omega \setminus T$. The goal of this algorithm is to find an explicit way of constructing C_T from Ω and T.

Let K be an arbitrary vertex set and S an arbitrary proper subset of K. A component $C \in \mathscr{C}(K)$ is *confined* to S if $N(C) \subseteq S$, otherwise it is unconfined to S. The *crib* of S with respect to K, crib(S, K), is defined as $(K \setminus S) \cup \bigcup_{C \in \text{unconf}(S,K)} C$, or equivalently, the union of $K \setminus S$ and all components associated with K that have neighborhoods intersecting $K \setminus S$. The crib becomes useful in defining the outlet and support functions below. Lemma 3 from [Tam19] shows that if K is a cliquish set and S is a proper subset, then crib(S, K) is a full component associated with S.

For a connected set $C \in\subseteq V(G)$, $G\langle C\rangle$ denotes the graph obtained from $G[N[C]]$ by completing $N(C)$ into a clique. C is called *feasible* if $\text{tw}(G\langle C\rangle) \leq k$. A crucial recurrence to formulating a PID variation of the BT algorithm is that C is feasible if and only if there is some potential maximal clique Ω such that $N(C) \subset \Omega$, $C = \text{crib}(N(C), \Omega)$, and every component $D \in \text{unconf}(N(C), \Omega)$ is feasible.

Support and Outlet

We assume an ordering $<$ on $V(G)$. For vertex sets U, W, we say that U *precedes* W if the minimum element of U is smaller than the minimum element of W under $<$. Using this definition, a connected set C is *inbound* if there is some full block associated with $N(C)$ that precedes C, otherwise it is *outbound*. Notice that if C is inbound, then $N(C)$ is a minimal separator since $N(C)$ has another full component associated with it. On the contrapositive, if $N(C)$ is not a minimal separator then C must be outbound. A full block $(N(C), C)$ is inbound (outbound) if C is inbound (outbound), respectively. Lemma 4 from [Tam19] states that if K is a cliquish set

and A_1, A_2 are two components associated with K, assuming A_1, A_2 are outbound, then either $N(A_1) \subseteq N(A_2)$ or $N(A_2) \subseteq N(A_1)$. This fact is then used in defining the *outlet*.

Let K be a cliquish vertex set, then outlet(K) is $\varnothing$ if no full component associated with K is outbound; otherwise, outlet(K) $= N(A)$ where A is a non-full component associated with K that is outbound chosen so that $N(A)$ is maximal. The *support* of K is unconf(outlet(K), K), the set of components associated with K that are not confined to the outlet of K. According to Lemma 4 [Tam19], every member of support(K) is inbound.

Block Sieve

A fundamental operation in the algorithm is to find all feasible o-blocks associated with a given i-block such that the union of their neighbors is no larger than $k+1$ (the presumed treewidth). This operation is effectively carried out using a *block sieve* data structure, which supports the following two operations:

1. `store`(S): Store the set S in the block sieve.
2. `superSet`(S): Return list of entries W stored in the block sieve such that $S \subseteq W$ and $|N(S) \cup N(W)| \leq k+1$

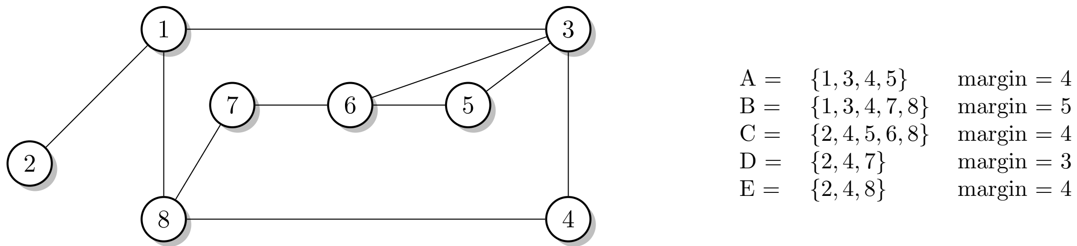
Associated with a block sieve is an integer k, which is used to calculate the *margins* of vertex sets; the margin for a vertex set U is defined as $k + 1 - |N(U)|$. In practice, this margin upper bound allows us to prune the lookup-space of sets stored in the block sieve, and quit if too many new neighbors are introduced.

A trie is used as the starting point for the implementation of the block sieve. Each node of the trie has an interval on $[1, n]$ associated with it such that the terminal element of the parent node's interval is immediately preceding the initial element.

A query (with input set S) can be made against this block sieve by starting at the root node and conducting a depth-first search on edges which are a superset of the input set. If $|N(S)|$ plus the margin of a node is too large, then the DFS returns from this branch: proceeding will only find supersets with too many neighbors.

For example, querying the block sieve in Figure 7.3 with a set $\{4, 8\}$ would return all stored supersets: B, C, and E. The block sieve would run a depth-first search on both edges of the root node in this case, since $\{4\} \subseteq \{1, 3, 4\}$ and $\{4\} \subseteq \{2, 4\}$.

To insert sets into the block sieve, start at the root node and only follow edges that are equal to the particular subset being considered. When no edge is equal, then make a new edge with interval $[t_p + 1, n]$ where t_p is the terminal point of the parent node's interval. If a particular node has too many children, the tree can be rebalanced by reducing the interval of the offending node and reinserting all descendant sets. For example, if the set $\{1, 2\}$ were to be stored in this



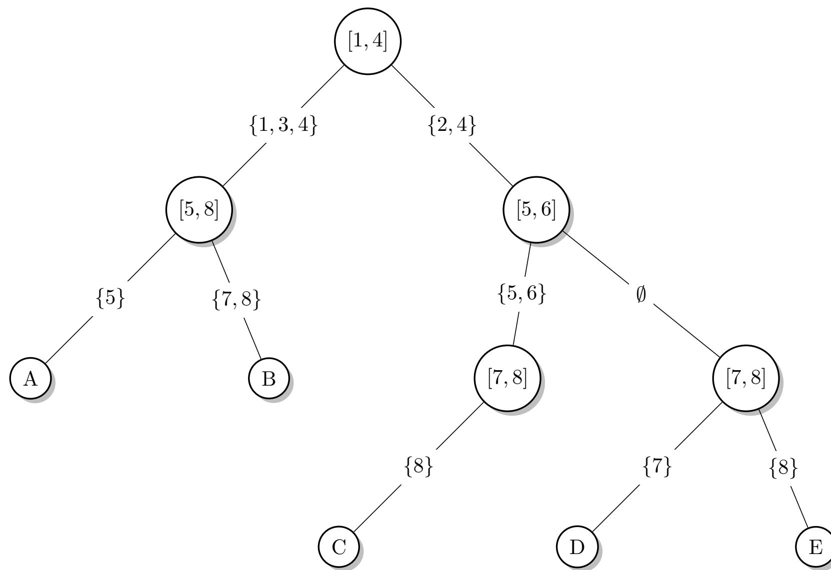


Figure 7.3 (Above-Left) A graph G with eight vertices. (Above-Right) Sets A-E and their margins are inserted into the block sieve trie. (Below) The populated block sieve trie when $k := 6$.

block sieve, a new edge with this label would be added off of the root node, since an edge with this label is not already present.

Layered Block Sieve

For an additional layer of pruning, in practice a list of block sieves will be used; this data structure is a *layered sieve*. Each of the block sieves has an associated maximum margin for sets which are inserted, which limits the number of block sieves that must be queried.

7.3.2 Proposed optimizations

There are three optimizations that we want to evaluate:

Reduction Routines

One characterization of treewidth is with *reduction routines*: apply a set of rules repeatedly to a graph instance until no rule applies; if no vertices remain at the end of this process then the graph has a treewidth specified by the set of reduction rules.

For example, the set of rules for treewidth 1 is simply "*remove a vertex of degree 1*." A graph with treewidth 1 must be a forest, with each component a tree, and each tree always has at least two vertices of degree 1; therefore this reduction rule "unwinds" trees to an empty graph.

Treewidth 2 has an additional reduction routine for simple graphs: "*remove a degree 2 vertex and connect its neighbors*." As a side note: As seen in Chapter 3, the class of treewidth 2 graphs is equal the class of *series-parallel* graphs, which are constructed with the *series* and *parallel* operations. The reduction rule mentioned here is the simple-graph equivalent for destructing series-parallel graphs.

Together, the reduction rules for removing the "tiny treewidth" portion of our graph instances can be run in linear time (each node is visited once). Additionally, given that tensor networks are generated by taking the line graph of a quantum circuit and these circuits often have 1-input, 1-output gates, we expect that a sizable portion of the original graph instance vertices will be degree-1 or degree-2. The expectation is that a good percent of the vertices may be removed with preprocessing, and this graph minimization will aid the dynamic programming tables and block sieves from becoming needlessly large.

Safe separator heuristics

The Tamaki implementation begins by executing three safe separator heuristics: `fill`, `defect`, and `degree`. These heuristics all find a safe separator S, which reduces the problem of computing the treewidth on G to computing the treewidth of $G\langle C\rangle$ for each component C associated with

S. As noted in [Tam19], on some graph instances these heuristics make the problem trivial (e.g. on many `pace2017` instances), but on some graphs none apply (e.g. on many `dimacs` instances). One optimization we want to evaluate is which of these heuristics could be dropped without increasing the dynamic programming runtime.

Block sieve node width

Each node in a block sieve's trie has a *width*, which determines the vertex window on the node. If the node has too many children (defined as 512 in the implementation [Oht17]), then the block sieve must rebalance itself with a larger node size. In the implementation [Oht17], the width options are {`int8`, `int16`, `int32`, `int64`}, all nodes start at `int8` and increase only if the children limit is reached. We want to provide the user with a memory-greedy flag, setting which width is used as default. In theory, choosing the minimum width such that the block sieve never has to resize will balance memory and runtime; but this width must be chosen per instance.

7.4 Experimental Setup and Results

All experiments were executed on a workstation running Ubuntu 18.04 and using Java JDK 8 to compile the Tamaki code with flags `-Xmx30g -Xms30g -Xss10m`, as done in the original repository [Oht17]. The workstation uses a single AMD Threadripper 1950X CPU (16 cores with a base clock of 3.4GHz and 32MB L3 cache), 64GB RAM, and an Intel Optane 900P SSD. The code was run on a single thread except for garbage collection.

7.4.1 Effectiveness of reduction routines

In this first experiment, we evaluate the effectiveness of removing the bits of each graph instance with treewidth at most 2; in Tables 7.1 and 7.2 we report quantiles on the percent of removed vertices and edges, respectively. In terms of portion of the graph instance removed, we find that these heuristics are quite effective *sometimes* on the `pace2017` and `dimacs` instances, sometimes removing all edges. However, more importantly, these heuristics are somewhat effective *always* on the tensor networks, where the lower-bound on removes vertices and edges are both non-zero, and the standard deviation is also small.

While the reduction routines may be effective in removing vertices and edges, this effort is useless without also reducing the PID algorithm run time. In Table 7.3 we find mixed success. For the treewidth corpora we find at most a 1.7× improvement, but risk running 20% slower. This discrepancy may be explained by a different vertex ordering on the reduced instances, leading to faster dynamic programming solutions.

Table 7.1 Reported quantiles on the percent of vertices removed per instance, clustered by instance corpus. Notably, nothing is removed from at least 25% of both treewidth corpora, but the majority of both vertices and edges are removed from at least one instance in each corpus. Contrastingly, the tensor networks are much more consistent in expected reduction.

Corpus	Min	25%	Median	75%	Max
`pace2017`	0.0%	0.0%	8.6%	40.6%	65.9%
`dimacs`	0.0%	0.0%	0.0%	14.8%	80.0%
`google-qs`	23.6%	24.6%	26.0%	27.4%	29.1%
`uccsd`	11.5%	11.9%	12.3%	17.1%	17.4%

Table 7.2 Reported quantiles on the percent of edges removed per instance, clustered by instance corpus. Similar to Table 7.1, these reduction routines vary in effectiveness on treewidth instances, but are predictably effective on tensor networks.

Corpus	Min	25%	Median	75%	Max
`pace2017`	0.0%	0.0%	3.9%	34.2%	59.4%
`dimacs`	0.0%	0.0%	0.0%	4.7%	100.0%
`google-qs`	12.7%	13.4%	14.6%	15.5%	16.7%
`uccsd`	5.0%	5.1%	5.3%	8.1%	8.3%

With the tensor network data we find that the reductions are always helpful, and `uccsd` data is at least 10% faster in all cases.

Table 7.3 Ratio of runtime for the original instance vs. the reduced instance; larger values are better. Similar to the effectiveness of the reduction routines, we find that the treewidth instances vary wildly, whereas the tensor networks are more predictable and positively impacted.

Corpus	Min	25%	Median	75%	Max
`pace2017`	0.8×	1.0×	1.0×	1.1×	1.7×
`dimacs`	0.9×	1.0×	1.0×	1.1×	1.2×
`google-qs`	1.0×	1.0×	1.0×	1.1×	1.2×
`uccsd`	1.1×	1.1×	1.2×	1.2×	1.3×

7.4.2 Effectiveness of safe separator heuristic choice

For evaluating the effectiveness of safe separator heuristics, the code is executed on the reduced graph instance with exactly one heuristic; Table 7.4 reports the observed ratios. Note that at least one heuristic must be chosen because the achieved (non-optimal) tree decomposition is

Table 7.4 Ratio of runtime for the original execution vs. executing with only the specified safe separator heuristic; larger values are better. Here we find a familiar split: treewidth results vary wildly and tensor networks are less impacted.

Heuristic	Corpus	Min	25%	Median	75%	Max
`fill`	`pace2017`	0.9×	1.0×	1.0×	1.1×	3.0×
	`dimacs`	0.9×	1.0×	1.1×	1.3×	3.1×
	`google-qs`	0.8×	0.9×	1.0×	1.0×	1.1×
	`uccsd`	1.1×	1.2×	1.5×	1.5×	1.6×
`defect`	`pace2017`	0.3×	0.8×	1.0×	1.0×	3.0×
	`dimacs`	0.8×	1.0×	1.1×	1.3×	3.0×
	`google-qs`	0.8×	0.9×	1.0×	1.0×	1.1×
	`uccsd`	1.1×	1.1×	1.2×	1.3×	1.4×
`degree`	`pace2017`	0.1×	0.7×	1.0×	1.1×	5.1×
	`dimacs`	0.8×	1.0×	1.1×	1.5×	3.7×
	`google-qs`	0.8×	0.9×	1.0×	1.0×	1.1×
	`uccsd`	1.1×	1.2×	1.6×	1.7×	1.9×

used as an upper bound in the dynamic programming algorithm.

For `pace2017` we find that `degree` achieves the highest speedup (5.1×) at the risk of taking 10× longer. A better middle ground might be the `fill` heuristic, which achieves 3× faster at the risk of 1.1× slower. For `dimacs` we find similar results, although both the risk and reward for running only `degree` is much lower.

For `google-qs`, we find that the heuristics do not play much role in the full runtime, each instance is at most 1.1× faster or slower based on the heuristic chosen.

For `uccsd`, we find that using only one heuristic is a strict improvement over using all heuristics, and that `degree` has the better quantiles of the three options. In total, using only the `degree` heuristic will result in a 1.1× to 1.9× faster total runtime.

7.4.3 Effectiveness of block sieve node width lower bound

In this experiment we found that adding a lower bound to the block sieve node width played absolutely no role in total runtime. In fact, the number of resizes executed per block sieve did not change, meaning that the node ranges played no role in the block sieve balance once 512 children were accumulated.

Anecdotally, adjusting the 512 limit on a node's children did not impact run time either.

Table 7.5 The ratio of user runtime vs. real runtime; lower is more serial.

Corpus	Min	25%	Median	75%	Max
`pace2017`	1.0×	1.4×	2.6×	4.3×	7.6×
`dimacs`	1.0×	1.7×	2.3×	3.1×	6.5×
`google-qs`	1.1×	1.5×	3.0×	4.3×	6.9×
`uccsd`	1.8×	2.4×	2.6×	2.9×	3.7×

7.5 Conclusion

In summary, we evaluated three potential optimizations for Tamaki's PID algorithm on treewidth and tensor network data.

For the reduction routines, we found that removing the treewidth 2 bits of the graph could lead to a 1.7× speedup, but in general, $1.1 - 1.3\times$ is more realistic.

In choosing a safe separator heuristic, we found that using only the most effective one could have a big impact (5.1× speedup), but missing the most effective one had a worse impact (10× slower). A good rule of thumb is to use all heuristics for all data, except `uccsd`, which benefited most from exactly one heuristic; of the heuristics, `degree` was best.

When adding a lower bound to the node width in the block sieve data structure, we observed no difference in runtimes. More exploration is needed to understand the precise implications, we expect that the block sieve data structure itself could be simplified into a bare trie and run faster without losing many advantages.

When executing these experiments, we additionally observed an active garbage collector (Table 7.5). Over all data, the median result shows that the equivalent of three cores are active throughout execution, and at maximum the real runtime is 7.6× the user runtime. This discrepancy suggests that the heap is overly active, especially considering the nature of updating tables in a dynamic programming algorithm. A promising direction for future work is to rewrite the implementation to use stack memory and reuse the table data structures.

7.6 Appendix: PID Algorithm Pseudocode

Algorithm 4 PID Dynamic Treewidth Algorithm

1: $\mathscr{I}$: Feasible I-blocks
2: $\mathscr{O}$: Feasible O-blocks
3: $\mathscr{P}$: Buildable potential maximal cliques
4: $\mathscr{F}$: Feasible potential maximal cliques
5: **procedure** Treewidth(G,k)
6: Initialize $\mathscr{I}_0, \mathscr{O}_0, \mathscr{P}_0, \mathscr{F}_0 = \emptyset$
7: See Excerpt 1 (Assemble table of buildable potential maximal cliques).
8: See Excerpt 2 (Dynamic programming).
9: See Excerpt 3 (Evaluation criteria).
10: **end procedure**

Algorithm 5 Excerpt 1 (Assemble table of buildable potential maximal cliques)

1: Set $j = 0$
2: **for all** $v \in V(G)$ **do**
3: **if** $N[v]$ is a potential maximal clique and $|N[v]| \leq k + 1$ **then**
4: Add $N[v]$ to $\mathscr{P}_0$
5: **if** support($N[v]$) $= \varnothing$ **then**
6: Add $N[v]$ to $\mathscr{F}_0$
7: **if** outlet($N[v]$) $\neq \varnothing$ **then**
8: Let $C =$ crib(outlet($N[v]$), $N[v]$)
9: **if** $C \neq C_h$ for $h \in [1, j]$ **then**
10: Increment j
11: Let $C_j = C$
12: **end if**
13: **end if**
14: **end if**
15: **end if**
16: **end for**

Algorithm 6 Excerpt 2 (Dynamic programming)

1: Set $i = 0$
2: **while** j is not incremented during the iteration step **do**
3:     **while** $i < j$ **do**
4:         Increment i
5:         Let $\mathscr{I}_i = \mathscr{I}_{i-1} \cup \{C_i\}$
6:         Let $\mathscr{O}_i = \mathscr{O}_{i-1}$, $\mathscr{P}_i = \mathscr{P}_{i-1}$, and $\mathscr{F}_i = \mathscr{F}_{i-1}$
7:         **for all** $B \in \mathscr{O}_{i-1}$ such that $C_i \subseteq B$ and $|N(C_i) \cup N(B)| \leq k + 1$ **do**
8:             Let $K = N(C_i) \cup N(B)$
9:             **if** K is a potential maximal clique **then**
10:                 Add K to $\mathscr{P}_i$
11:             **end if**
12:             **if** $|K| \leq k$ and there is a unique full component A associated with K **then**
13:                 Add A to $\mathscr{O}_i$
14:             **end if**
15:         **end for**
16:         Let A be the unique full component associated with $N(C_i)$
17:         Add A to $\mathscr{O}_i$
18:         **for all** $A \in \mathscr{O}_i \setminus \mathscr{O}_{i-1}$ **do**
19:             **for all** $v \in N(A)$ **do**
20:                 Let $K = N(A) \cup (N(v) \cap A)$
21:                 **if** $|K| \leq k + 1$ and K is a potential maximal clique **then**
22:                     Add K to $\mathscr{P}_i$
23:                 **end if**
24:             **end for**
25:         **end for**
26:         **for all** $K \in \mathscr{P}_i \setminus \mathscr{F}_{i-1}$ **do**
27:             **if** support$(K) \subseteq \mathscr{I}_{i-1}$
28:                 Add K to $\mathscr{F}_i$ **then**
29:                 **if** outlet$(K) \neq \varnothing$ **then**
30:                     Let $C = $ crib(outlet$(K), K)$
31:                     **if** $C \neq C_h$ for $h \in [1, j]$ **then**
32:                         Increment j
33:                         Let $C_j = C$
34:                     **end if**
35:                 **end if**
36:             **end if**
37:         **end for**
38:     **end while**
39: **end while**

Algorithm 7 Excerpt 3 (Evaluation criteria)

```
1: if There is some K ∈ 𝒥_j such that outlet(K) = ∅ then
2:     return True
3: end if
4: return False
```

BIBLIOGRAPHY

[Aga05a] Agarwal, A., Charikar, M., Makarychev, K. & Makarychev, Y. "$O(\sqrt{\log n})$ approximation algorithms for min UnCut, min 2CNF deletion, and directed cut problems". *Proceedings of the thirty-seventh annual ACM symposium on Theory of computing*. ACM. 2005, pp. 573–581.

[Aga05b] Agarwal, A., Charikar, M., Makarychev, K. & Makarychev, Y. "$O(\sqrt{\log n})$ approximation algorithms for min UnCut, min 2CNF deletion, and directed cut problems". *Proceedings of the thirty-seventh annual ACM symposium on Theory of computing*. ACM. New York, NY, USA: ACM, 2005, pp. 573–581.

[Aki16] Akiba, T. & Iwata, Y. "Branch-and-reduce exponential/fpt algorithms in practice: A case study of vertex cover". *Theoretical Computer Science* **609** (2016), pp. 211–225.

[Aki17] Akiba, T. & Iwata, Y. *Vertex Cover Solver*. 2017.

[Alb02] Albert, R. & Barabási, A.-L. "Statistical mechanics of complex networks". *Reviews of modern physics* **74**.1 (2002), p. 47.

[Arn87] Arnborg, S., Corneil, D. G. & Proskurowski, A. "Complexity of finding embeddings in ak-tree". *SIAM Journal on Algebraic Discrete Methods* **8**.2 (1987), pp. 277–284.

[BY04] Bar-Yehuda, R., Bendel, K., Freund, A. & Rawitz, D. "Local ratio: A unified framework for approximation algorithms". *ACM Computing Surveys* **36**.4 (2004), pp. 422–463.

[Bat03] Batagelj, V. & Zaversnik, M. "An $O(m)$ algorithm for cores decomposition of networks". *arXiv preprint cs/0310049* (2003).

[Bea90] Beasley, J. E. "OR-Library: distributing test problems by electronic mail". *Journal of the operational research society* (1990), pp. 1069–1072.

[Bea98] Beasley, J. E. *Heuristic algorithms for the unconstrained binary quadratic programming problem*. 1998.

[Bea18] Beasley, J. E. *OR-Library*. 2018.

[Bia17] Biamonte, J. & Bergholm, V. "Tensor Networks in a Nutshell". *arXiv preprint arXiv:1708.00006* (2017).

[Bia15] Biamonte, J. D., Morton, J. & Turner, J. "Tensor Network Contractions for #SAT". *Journal of Statistical Physics* **160**.5 (2015), pp. 1389–1404.

[Bod98] Bodlaender, H. L. "A partial k-arboretum of graphs with bounded treewidth". *Theoretical Computer Science* **209**.1 (1998), pp. 1 –45.

[Bod01] Bodlaender, H. L., Koster, A. M., Eijkhof, F. v. d. & Gaag, L. C. van der. "Preprocessing for triangulation of probabilistic networks" (2001).

[Boi18] Boixo, S., Isakov, S. V., Smelyanskiy, V. N., Babbush, R., Ding, N., Jiang, Z., Bremner, M. J., Martinis, J. M. & Neven, H. "Characterizing quantum supremacy in near-term devices". *Nature Physics* **14**.6 (2018), pp. 1–6. arXiv: `1608.00263`.

[Boo16] Boothby, T., King, A. D. & Roy, A. "Fast clique minor generation in Chimera qubit connectivity graphs". *Quantum Information Processing* **15**.1 (2016), pp. 495–508.

[Bor02] Boros, E. & Hammer, P. L. "Pseudo-boolean optimization". *Discrete applied mathematics* **123**.1 (2002), pp. 155–225.

[Bou01] Bouchitté, V. & Todinca, I. "Treewidth and minimum fill-in: Grouping the minimal separators". *SIAM Journal on Computing* **31**.1 (2001), pp. 212–232.

[Boy17] Boyda, E., Basu, S., Ganguly, S., Michaelis, A., Mukhopadhyay, S. & Nemani, R. R. "Deploying a quantum annealing processor to detect tree cover in aerial imagery of California". *PloS one* **12**.2 (2017), e0172505.

[Bri17] Britt, K. A. & Humble, T. S. "High-performance computing with quantum processing units". *ACM Journal on Emerging Technologies in Computing Systems (JETC)* **13**.3 (2017), pp. 1–13.

[Cai14] Cai, J., Macready, W. G. & Roy, A. "A practical heuristic for finding graph minors". *arXiv preprint arXiv:1406.2741* (2014).

[CC97] Chi-Chung, L., Sadayappan, P & Wenger, R. "On optimizing a class of multidimensional loops with reduction for parallel execution". *Parallel Processing Letters* **7**.02 (1997), pp. 157–168.

[Cho08] Choi, V. "Minor-embedding in adiabatic quantum computation: I. The parameter setting problem". *Quantum Information Processing* **7**.5 (2008), pp. 193–209.

[Cho11] Choi, V. "Minor-embedding in adiabatic quantum computation: II. Minor-universal graph design". *Quantum Information Processing* **10**.3 (2011), pp. 343–353.

[Chu02] Chung, F. & Lu, L. "Connected components in random graphs with given expected degree sequences". *Annals of combinatorics* **6**.2 (2002), pp. 125–145.

[D-W16] D-Wave Systems Inc. *SAPI 2.4.* 2016.

[Dan19] Dang, A., Hill, C. D. & Hollenberg, L. C. "Optimising Matrix Product State Simulations of Shor's Algorithm". *Quantum* **3** (2019), p. 116.

[Dar17] Darmawan, A. S. & Poulin, D. "Tensor-Network Simulations of the Surface Code under Realistic Noise". *Physical Review Letters* **119**.4 (2017), p. 40502. arXiv: 1607.06460.

[Dat19] Date, P., Patton, R., Schuman, C. & Potok, T. "Efficiently embedding QUBO problems on adiabatic quantum computers". *Quantum Information Processing* **18**.4 (2019), p. 117.

[Del17] Dell, H., Husfeldt, T., Jansen, B., Kaski, P., Komusiewicz, C. & Rosamond, F. "The first Parameterized Algorithms and Computational Experiments challenge". *Leibniz International Proceedings in Informatics, LIPIcs*. Ed. by Guo, J. & Hermelin, D. Vol. 63. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017, 30:1–30:9.

[Del18] Dell, H., Komusiewicz, C., Talmon, N. & Weller, M. "The PACE 2017 Parameterized Algorithms and Computational Experiments challenge: The second iteration". *Leibniz International Proceedings in Informatics, LIPIcs*. Ed. by Lokshtanov, D. & Nishimura, N. Vol. 89. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2018, 30:1–30:12.

[Dem19] Demaine, E. D., Goodrich, T. D., Kloster, K., Lavallee, B., Liu, Q. C., Sullivan, B. D., Vakilian, A. & Poel, A. van der. "Structural Rounding: Approximation Algorithms for Graphs Near an Algorithmically Tractable Class". *27th Annual European Symposium on Algorithms (ESA 2019)*. Vol. 144. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. 2019, p. 37.

[Den16] Denchev, V. S., Boixo, S., Isakov, S. V., Ding, N., Babbush, R., Smelyanskiy, V., Martinis, J. & Neven, H. "What is the Computational Value of Finite-Range Tunneling?" *Physical Review X* **6**.3 (2016), p. 031015.

[Die05] Diestel, R. "Graph Theory". *Graduate Texts in Mathematics* **173** (2005).

[DIM20] DIMACS. *Graph Coloring Instances*. 2020.

[Din05] Dinur, I. & Safra, S. "On the hardness of approximating minimum vertex cover". *Annals of Mathematics* (2005), pp. 439–485.

[Dum17] Dumitrescu, E. "Tree tensor network approach to simulating Shor's algorithm". *Phys. Rev. A* **96** (6 2017), p. 062322.

[Dun15] Dunning, I., Gupta, S. & Silberholz, J. *What Works Best When? A Framework for Systematic Heuristic Evaluation*. 2015.

[Eij07] Eijkhof, F. van den, Bodlaender, H. L. & Koster, M. A. "Safe reduction rules for weighted treewidth". *Algorithmica* **47**.2 (2007), pp. 139–158.

[Eis10] Eisert, J., Cramer, M. & Plenio, M. B. "Colloquium: Area laws for the entanglement entropy". *Rev. Mod. Phys.* **82** (1 2010), pp. 277–306.

[Epp92] Eppstein, D. "Parallel recognition of series-parallel graphs". *Information and Computation* **98**.1 (1992), pp. 41–55.

[Erd60] Erdos, P. & Rényi, A. "On the evolution of random graphs". *Publ. Math. Inst. Hung. Acad. Sci* **5**.1 (1960), pp. 17–60.

[Eve09] Evenbly, G & Vidal, G. "Algorithms for entanglement renormalization ver 2". *Physical Review B* **79**.December 2008 (2009), pp. 1–17. arXiv: `arXiv:0707.1454v3`.

[Eve13] Evenbly, G. & Pfeifer, R. N. C. "Improving the efficiency of variational tensor network algorithms". *Physical Review B* **89**.24 (2013), p. 245118. arXiv: `1310.8023`.

[Far14] Farhi, E., Goldstone, J. & Gutmann, S. "A Quantum Approximate Optimization Algorithm". *arXiv preprint arXiv:1411.4028* (2014). arXiv: `1411.4028`.

[Far00] Farhi, E., Goldstone, J., Gutmann, S. & Sipser, M. "Quantum computation by adiabatic evolution". *arXiv preprint quant-ph/0001106* (2000).

[Fer14] Ferris, A. J. & Poulin, D. "Tensor networks and quantum error correction". *Physical Review Letters* **113**.3 (2014), p. 030501. arXiv: `1312.4578`.

[Fri18] Fried, E. S., Sawaya, N. P., Cao, Y., Kivlichan, I. D., Romero, J. & Aspuru-Guzik, A. "qTorch: The quantum tensor contraction handler". *PloS one* **13**.12 (2018).

[Glo98] Glover, F., Kochenberger, G. A. & Alidaee, B. "Adaptive memory tabu search for binary quadratic programs". *Management Science* **44**.3 (1998), pp. 336–345.

[GNU17] GNU. *Linear Programming Kit (GLPK), Version 4.61*. 2017.

[Gog04] Gogate, V. & Dechter, R. "A complete anytime algorithm for treewidth". *Proceedings of the 20th conference on uncertainty in artificial intelligence*. 1. AUAI Press. 2004, pp. 201–208.

[Goo18a] Goodrich, T. D., Horton, E. & Sullivan, B. D. "Practical Graph Bipartization with Applications in Near-Term Quantum Computing". *arXiv preprint arXiv:1805.01041* (2018).

[Goo18b] Goodrich, T. D., Horton, E. & Sullivan, B. D. *Practical OCT*. 2018.

[Goo18c] Goodrich, T. D., Sullivan, B. D. & Humble, T. S. "Optimizing adiabatic quantum program compilation using a graph-theoretic framework". *Quantum Information Processing* **17**.5 (2018), p. 118.

[Hag08] Hagberg, A. A., Schult, D. A. & Swart, P. J. "Exploring network structure, dynamics, and function using NetworkX". *Proceedings of the 7th Python in Science Conference (SciPy2008)*. Pasadena, CA USA, 2008, pp. 11–15.

[Hal94] Halldórsson, M. & Radhakrishnan, J. "Greed is good: Approximating independent sets in sparse and bounded-degree graphs". *Proceedings of the twenty-sixth annual ACM symposium on Theory of computing*. ACM. 1994, pp. 439–448.

[Ham16] Hamilton, K. E. & Humble, T. S. "Identifying the minor set cover of dense connected bipartite graphs via random matching edge sets". *arXiv preprint arXiv:1612.07366* (2016).

[Ham17] Hamilton, K. E. & Humble, T. S. "Identifying the minor set cover of dense connected bipartite graphs via random matching edge sets". *Quantum Information Processing* **16**.4 (2017), p. 94.

[Hås01] Håstad, J. "Some optimal inapproximability results". *Journal of the ACM (JACM)* **48**.4 (2001), pp. 798–859.

[Hüf05] Hüffner, F. "Algorithm engineering for optimal graph bipartization". *International Workshop on Experimental and Efficient Algorithms*. Springer. 2005, pp. 240–252.

[Hüf06] Hüffner, F. *occ*. 2006.

[Hüf09] Hüffner, F. "Algorithm engineering for optimal graph bipartization." *Journal of Graph Algorithms and Applications* **13**.2 (2009), pp. 77–98.

[Hum14] Humble, T. S., McCaskey, A. J., Bennink, R. S., Billings, J. J., D'Azevedo, E., Sullivan, B. D., Klymko, C. F. & Seddiqi, H. "An integrated programming and development environment for adiabatic quantum optimization". *Computational Science & Discovery* **7**.1 (2014), p. 015006.

[Hum16] Humble, T. S., McCaskey, A. J., Schrock, J., Seddiqi, H., Britt, K. A. & Imam, N. "Performance Models for Split-execution Computing Systems". *Parallel and Distributed Processing Symposium Workshops, 2016 IEEE International*. IEEE. 2016, pp. 545–554.

[IBM17] IBM. *CPLEX Optimization Studio 12.8*. 2017.

[Iwa14] Iwata, Y., Oka, K. & Yoshida, Y. "Linear-time FPT algorithms via network flow". *Proceedings of the twenty-fifth annual ACM-SIAM symposium on Discrete algorithms*. Society for Industrial and Applied Mathematics. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2014, pp. 1749–1761.

[Iwa17] Iwata, Y., Yamaguchi, Y. & Yoshida, Y. *0/1/all CSPs, Half-Integral A-path Packing, and Linear-Time FPT Algorithms*. 2017.

[Kad98] Kadowaki, T. & Nishimori, H. “Quantum annealing in the transverse Ising model”. *Physical Review E* **58**.5 (1998), p. 5355.

[Kas11] Kassal, I., Whitfield, J. D., Perdomo-Ortiz, A., Yung, M.-H. & Aspuru-Guzik, A. “Simulating chemistry using quantum computers”. *Annual review of physical chemistry* **62** (2011), pp. 185–207.

[Kho08] Khot, S. & Regev, O. “Vertex cover might be hard to approximate to within $2-\varepsilon$”. *Journal of Computer and System Sciences* **74**.3 (2008), pp. 335–349.

[Kin19] King, J., Yarkoni, S., Raymond, J., Ozfidan, I., King, A. D., Nevisi, M. M., Hilton, J. P. & McGeoch, C. C. “Quantum annealing amid local ruggedness and global frustration”. *Journal of the Physical Society of Japan* **88**.6 (2019), p. 061007.

[Kly14] Klymko, C., Sullivan, B. D. & Humble, T. S. “Adiabatic quantum programming: minor embedding with hard faults”. *Quantum information processing* **13**.3 (2014), pp. 709–729.

[Kos88] Kosko, B. “Bidirectional associative memories”. *IEEE Transactions on Systems, man, and Cybernetics* **18**.1 (1988), pp. 49–60.

[Kra14] Kratsch, S. & Wahlström, M. “Compression via matroids: a randomized polynomial kernel for odd cycle transversal”. *ACM Transactions on Algorithms (TALG)* **10**.4 (2014), p. 20.

[Lar17] Larisch, L & Salfelder, F. *FreeTDI PACE 2017 Submission*. 2017.

[Lei14] Leighton, F. T. *Introduction to parallel algorithms and architectures: Arrays · trees · hypercubes*. Elsevier, 2014.

[Lew80] Lewis, J. M. & Yannakakis, M. “The node-deletion problem for hereditary properties is NP-complete”. *Journal of Computer and System Sciences* **20**.2 (1980), pp. 219–230.

[Lic70] Lick, D. R. & White, A. T. “k-Degenerate graphs”. *Canadian Journal of Mathematics* **22**.5 (1970), pp. 1082–1096.

[Lok14] Lokshtanov, D., Narayanaswamy, N., Raman, V., Ramanujan, M. & Saurabh, S. “Faster parameterized algorithms using linear programming”. *ACM Transactions on Algorithms (TALG)* **11**.2 (2014), p. 15.

[Lok09] Lokshtanov, D., Saurabh, S. & Sikdar, S. “Simpler parameterized algorithm for OCT”. *International Workshop on Combinatorial Algorithms*. Springer. 2009, pp. 380–384.

[Lok12] Lokshtanov, D., Saurabh, S. & Wahlström, M. “Subexponential parameterized odd cycle transversal on planar graphs”. *LIPIcs-Leibniz International Proceedings in Informatics*. Vol. 18. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2012.

[Lub86] Luby, M. “A simple parallel algorithm for the maximal independent set problem”. *SIAM journal on computing* **15**.4 (1986), pp. 1036–1053.

[Luc14] Lucas, A. “Ising formulations of many NP problems”. *Frontiers in Physics* **2** (2014), p. 5.

[Lun93] Lund, C. & Yannakakis, M. “The approximation of maximum subgraph problems”. *Automata, Languages and Programming* (1993), pp. 40–51.

[Mar05] Markov, I. L. & Shi, Y. “Simulating quantum computation by contracting tensor networks”. *SIAM Journal on Computing* **38**.3 (2005), pp. 963–981. arXiv: `0511069 [quant-ph]`.

[Mat10] Mathieson, L. “The parameterized complexity of editing graphs for bounded degeneracy”. *Theoretical Computer Science* **411**.34-36 (2010), pp. 3181–3187.

[McC20] McCaskey, A. J., Lyakh, D. I., Dumitrescu, E. F., Powers, S. S. & Humble, T. S. “XACC: a system-level software infrastructure for heterogeneous quantum–classical computing”. *Quantum Science and Technology* **5**.2 (2020), p. 024002.

[Mer14] Merkel, D. “Docker: lightweight linux containers for consistent development and deployment”. *Linux Journal* **2014**.239 (2014), p. 2.

[Nev08] Neven, H., Rose, G. & Macready, W. G. *Image recognition with an adiabatic quantum computer I. Mapping to quadratic unconstrained binary optimization*. 2008.

[Oht17] Ohtsuka, H & Tamaki, H. *Meiji PACE 2017 Submission*. 2017.

[Oka19] Okada, S., Ohzeki, M., Terabe, M. & Taguchi, S. “Improving solutions by embedding larger subproblems in a D-Wave quantum annealer”. *Scientific reports* **9**.1 (2019), p. 2098.

[Pas15] Pastawski, F., Yoshida, B., Harlow, D. & Preskill, J. “Holographic quantum error-correcting codes: toy models for the bulk/boundary correspondence”. *Journal of High Energy Physics* **2015**.6 (2015), p. 149.

[Ped17] Pednault, E., Gunnels, J. A., Nannicini, G., Horesh, L., Magerlein, T., Solomonik, E. & Wisnieff, R. “Breaking the 49-Qubit Barrier in the Simulation of Quantum Circuits”. *arXiv preprint arXiv:1710.05867* (2017). arXiv: `1710.05867`.

[Pfe14a] Pfeifer, R. N. C., Evenbly, G., Singh, S. & Vidal, G. “NCON: A tensor network contractor for MATLAB”. *arXiv preprint arXiv:1402.0939* (2014). arXiv: `1402.0939`.

[Pfe14b] Pfeifer, R. N., Haegeman, J. & Verstraete, F. “Faster identification of optimal contraction sequences for tensor networks”. *Physical Review E - Statistical, Nonlinear, and Soft Matter Physics* **90**.3 (2014), p. 33315. arXiv: `1304.6112`.

[Pic16] Pichler, T., Dalmonte, M., Rico, E., Zoller, P. & Montangero, S. "Real-Time Dynamics in U(1) Lattice Gauge Theories with Tensor Networks". *Phys. Rev. X* **6** (1 2016), p. 011023.

[Pot18] Potok, T. E., Schuman, C., Young, S., Patton, R., Spedalieri, F., Liu, J., Yao, K.-T., Rose, G. & Chakma, G. "A Study of Complex Deep Learning Networks on High-Performance, Neuromorphic, and Quantum Computers". *ACM Journal on Emerging Technologies in Computing Systems (JETC)* **14**.2 (2018), p. 19.

[Ree04] Reed, B., Smith, K. & Vetta, A. "Finding odd cycle transversals". *Operations Research Letters* **32**.4 (2004), pp. 299–301.

[Ric14] Rico, E., Pichler, T., Dalmonte, M., Zoller, P. & Montangero, S. "Tensor Networks for Lattice Gauge Theories and Atomic Quantum Simulation". *Phys. Rev. Lett.* **112** (20 2014), p. 201601.

[Rie15] Rieffel, E. G., Venturelli, D., O'Gorman, B., Do, M. B., Prystay, E. M. & Smelyanskiy, V. N. "A case study in programming a quantum annealer for hard operational planning problems". *Quantum Information Processing* **14**.1 (2015), pp. 1–36.

[Rob95] Robertson, N. & Seymour, P. D. "Graph minors. XIII. The disjoint paths problem". *Journal of combinatorial theory, Series B* **63**.1 (1995), pp. 65–110.

[Sch13] Schmidt, J. M. "A simple test on 2-vertex-and 2-edge-connectivity". *Information Processing Letters* **113**.7 (2013), pp. 241–244.

[Sch17] Schrock, J., McCaskey, A. J., Hamilton, K. E., Humble, T. S. & Imam, N. "Recall Performance for Content-Addressable Memory Using Adiabatic Quantum Optimization". *Entropy* **19**.9 (2017), p. 500.

[Sch11] Schuch, N., Pérez-García, D. & Cirac, I. "Classifying quantum phases using matrix product states and projected entangled pair states". *Phys. Rev. B* **84** (16 2011), p. 165139.

[Tam19] Tamaki, H. "Positive-instance driven dynamic programming for treewidth". *Journal of Combinatorial Optimization* **37**.4 (2019), pp. 1283–1311.

[VDZ17] Van Der Zanden, T. C. & Bodlaender, H. L. "Computing Treewidth on the GPU". *arXiv preprint arXiv:1709.09990* (2017).

[Ven15] Venturelli, D., Mandra, S., Knysh, S., O'Gorman, B., Biswas, R. & Smelyanskiy, V. "Quantum optimization of fully connected spin glasses". *Physical Review X* **5**.3 (2015), p. 031040.

[Ver04] Verstraete, F., García-Ripoll, J. J. & Cirac, J. I. "Matrix Product Density Operators: Simulation of Finite-Temperature and Dissipative Systems". *Phys. Rev. Lett.* **93** (20 2004), p. 207204.

[Vid06] Vidal, G. “A class of quantum many-body states that can be efficiently simulated”. *Phys. Rev. Lett.* **101**.11 (2006), p. 110501. arXiv: `0610099 [quant-ph]`.

[Vid03] Vidal, G. “Efficient classical simulation of slightly entangled quantum computations”. *Physical Review Letters* **91**.14 (2003), p. 147902. arXiv: `0301063 [quant-ph]`.

[Wah17] Wahlström, M. “LP-branching algorithms based on biased graphs”. *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*. Society for Industrial and Applied Mathematics. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2017, pp. 1559–1570.

[Wan14] Wang, C., Jonckheere, E. & Brun, T. “Ollivier-Ricci curvature and fast approximation to tree-width in embeddability of QUBO problems”. *Communications, Control and Signal Processing (ISCCSP), 2014 6th International Symposium on*. IEEE. 2014, pp. 598–601.

[Wec14] Wecker, D. & Svore, K. M. “LIQUi|>: A software design architecture and domain-specific language for quantum computing”. *arXiv preprint arXiv:1402.4467* (2014).

[Wei00] Weisstein, E. W. “Petersen graph” (2000).

[Wer16] Werner, A. H., Jaschke, D., Silvi, P., Kliesch, M., Calarco, T., Eisert, J. & Montangero, S. “Positive Tensor Network Approach for Simulating Open Quantum Many-Body Systems”. *Phys. Rev. Lett.* **116** (23 2016), p. 237201.

[Wer03] Wernicke, S. *On the algorithmic tractability of single nucleotide polymorphism (SNP) analysis and related problems*. 2003.

[Whi92] White, S. R. “Density matrix formulation for quantum renormalization groups”. *Physical Review Letters* **69**.19 (1992), pp. 2863–2866.