

Practical Graph Bipartization with Applications in Near-Term Quantum Computing

Timothy D. Goodrich\textsuperscript{1}, Eric Horton\textsuperscript{1}, and Blair D. Sullivan\textsuperscript{1}

\textsuperscript{1} Department of Computer Science, North Carolina State University, Raleigh, North Carolina 27606
\{tdgoodri, ewhorton, blair_sullivan\}@ncsu.edu

Abstract

We experimentally evaluate the practical state-of-the-art in graph bipartization (Odd Cycle Transversal), motivated by recent advances in near-term quantum computing hardware and the related embedding problems. We assemble a preprocessing suite of fast input reduction routines from the Odd Cycle Transversal (OCT) and Vertex Cover (VC) literature, allowing algorithm implementations to be compared using Quadratic Unconstrained Binary Optimization problems from the quantum literature. These problems represent a wide array of properties, leading to harder OCT instances than in previous benchmarks.

In addition to combinatorial branching algorithms for solving OCT directly, we study various reformulations into other NP-hard problems such as VC and Integer Linear Programming, enabling the use of solvers such as CPLEX. We find that for heuristic solutions with time constraints under a second, iterative compression routines jump started with a heuristic solution perform best, after which it is worth the cost to use a highly tuned solver like CPLEX. Exact solutions are also dominated by ILP formulations for CPLEX, usually by an order of magnitude. These results are then compared to those on a corpus of synthetic graphs, showing that our results are robust and may generalize to other domain data.

Finally, we provide all code and data in an open source suite, including a Python API for accessing reduction routines and branching algorithms, along with scripts for fully replicating our results.

1998 ACM Subject Classification G.2.2 Graph Theory, G.2.3 Applications

Keywords and phrases Graph Bipartization, Odd Cycle Transversal, Near-Term Quantum Computing, Quantum Annealing, Vertex Cover, Integer Linear Programming

Digital Object Identifier 10.4230/LIPIcs...
1 Introduction

ODD CYCLE TRANSVERSAL (OCT), the problem of deleting vertices such that a graph is bipartite, has been well-studied in the theory community over the last two decades. Techniques such as iterative compression [35, 20] and branch-and-reduce [28, 3] 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 [40] (denoted WH), where run times have dropped from over 10 hours [40] to under 3 minutes [20] to under 1 second for multiple state-of-the-art solvers [3].

Recently, a need for practical graph bipartization algorithms has arisen in quantum computing, where physical constraints limit the topology of all near-term quantum hardware to bipartite structure. However, it is unclear whether the existing best practices for OCT generalize to this setting. Specifically, the graphs arising in quantum computing are not necessarily close-to-bipartite, in contrast to WH where non-bipartite vertices represent read-errors in biological data. 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 introduces a run time vs. solution quality trade-off previously not considered. In this paper we experimentally evaluate methods for computing OCT in both the quantum setting and a more generalized setting with synthetic data.

1.1 Related Work

Modern theoretical advances on OCT began with the seminal result of Reed, Smith, and Vetta [35], 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 [20, 29]. 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 [1] where $k'$ denotes the gap between an optimal solution to VERTEX COVER and the solution given by the linear programming (LP) relaxation [28]. 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 [23], and $O^*(4^k)$ [39] and $O(4^k n)$ [24] algorithms for the more general problem of NON-MONOTONIC CYCLE TRANSVERSAL.

Other algorithmic results for OCT include a $O(\log n)$-approximation algorithm [1], a randomized algorithm based on matroids [27], and a subexponential algorithm on planar graphs [30].

On the practical side, the first implementation was a branch-and-bound algorithm by Wernicke in 2003 [40] 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 [20] 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

---

1 $O^*(f(k))$ denotes $O(f(k)n^c)$ for some constant $c$
this algorithm against an ILP formulation using the GNU Linear Programming Kit (GLPK) [15], which had unfavorable run times. More recently, Akiba and Iwata [3] used a VC-solver based on branch-and-reduce to solve OCT using a standard transformation to VC [28]. 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 [21] all finished within three seconds.

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 [40, 20], we provide a new, significantly more difficult benchmark corpus. Motivated by the widespread usage of Quadratic Unconstrained Binary Optimization (QUBO) problems in quantum annealing research [33], we select QUBO instances from a recent survey [12] that would be of interest to practitioners working on near-term quantum annealers. These datasets are selections from Glover, Kochenberger, and Alidaee [14] (denoted GKA) and Beasley [6] (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 [40] and VC literature [3] and applied to all datasets to harden the benchmark corpus. Heuristics for OCT upper bounds [40, 17] 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 [20]. These combinatorial algorithms for solving OCT directly are complemented by VC-based [3] and ILP-based [21] 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 [16] open sourced with the BSD 3-Clause license. For the interested reader, Appendix A contains implementation details and
All modern quantum hardware topologies are natively bipartite (visually denoted by a blue-white 2-coloring). Google’s Bristlecone hardware \cite{GoogleBristlecone} is expected to be the first hardware to achieve quantum supremacy over classical computers. Rigetti Computing \cite{Rigetti} and IBM \cite{IBM} both manufacture circuit-based quantum computers with 20- and 50-qubits, respectively. D-Wave Systems manufactures quantum annealers using ‘Chimera($L, M, N$)’ graphs: An $M \times N$ grid of complete bipartite $K_{L,L}$ cells; the upper-left corner of this fabric is shown here.

Appendix B contains extended results.

2 Background

One motivation for practical graph bipartization implementations stems from advances in near-term quantum computing. While long-term quantum computing focuses on using quantum computers to unequivocally beat classical computing, the theoretical hardware requirements are quite far from modern production models. For example, running Shor’s Algorithm to factor integers in 2048-bit RSA keys is estimated to require three million qubits in a circuit-based model when error correction is taken into account \cite{Shor}. In contrast, near-term quantum computing concentrates on using quantum computers obtainable now to achieve speedups in specific applications. Figure 1 depicts several such hardware models as graphs, where qubits and their couplers are represented as nodes and edges, respectively. Notably, physical constraints force modern hardware to have bipartite topology.

We denote a graph $G = (V, E)$. For a set of vertices $S$, we denote the subgraph induced by deleting $S$ as $G \setminus S$. We now formally define OCT.

**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 $|S|$.

Graph bipartization (OCT) occurs during the compilation step, when a problem graph must be embedded into the hardware graph. The definition of “embedding” depends on the
model of quantum computing, but at a high level it captures the notion of mapping logical qubits needed by the program to physical qubits present in the hardware.

In analog quantum computing (“quantum annealing”), a typical pipeline is for an optimization problem to be formulated as a **Quadratic Unconstrained Binary Optimization (QUBO)** problem, which then must be found as a **graph minor** of the hardware [21]. Researchers have had success running natively bipartite QUBOs (e.g. deep learning models) on D-Wave Systems annealers [36][20][8][34]. Generalizing these tools to non-bipartite QUBOs is currently of interest to enable additional applications (e.g. Karp’s 21 NP-hard problems [32]). The area of automatic embedding tools is under active development (refer to, e.g., [17][18][38]).

In circuit-based quantum computing, the embedding problem is a more complex placement, routing and scheduling problem [37]. Unlike on quantum annealers, adjacent qubits on a circuit-based hardware can be swapped, allowing non-complete hardware topology to execute gates on all combinations of edges through a series of exchanges. However, these swaps should be minimized to decrease run time, highlighting the need for optimal initial placements. Fast graph bipartization routines would enable a better initial placement analysis so that odd cycles in the problem circuit are most easily emulated on the bipartite hardware.

While the theory behind circuit-based models is older, production-level analog computers are more prevalent, so we will restrict our scope to quantum annealing. We will concentrate on two use cases of current relevance. First, a feasible bipartization of a graph is useful for certain embedding algorithms, regardless of minimality [17][18]. An automated 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. In a second use case, OCT can be used to identify when a particular program cannot embed into hardware, as is shown for D-Wave Systems’ Chimera hardware in [17]. 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 5.

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 [19], which we enhance with heuristics to create an anytime algorithm with increased performance particularly when given small timeouts [16]. Finally, we detail how CPLEX can solve OCT using various reformulations into **Integer Linear Programming**.

3.1 Reduction Routines

Reduction routines for OCT come from two sources. Wernicke’s branch-and-bound algorithm [40] uses nine reductions directly on the OCT graph 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 comes from Akiba and Iwata’s VC solver [3], which uses...
nine reduction routines specific to the VC instance’s graph. Based on the conversion between OCT and VC in Section 3.5, 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 \[40, 3\] 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 \[4\] to output the graph after a single round of reductions.

### 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 \[40\]; 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 \[51\]. Recent work showed that by using the minimum-degree heuristic for independent set, this strategy gives a \( d \)-approximation in \( d \)-degenerate graphs \[17\]. 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.

### 3.3 Iterative Compression

The state-of-the-art implementation for solving OCT combinatorially comes from Hüffner’s simplification of the iterative compression algorithm \[20, 35\]. 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 vertices assigned to the transversal and the vertices removed by the cut are less than \( k + 1 \), then the solution was compressed, and otherwise a proof is found that no solution of size \( k \) exists on this subgraph. Using Edmonds-Karp for the min-cut algorithm, this compression routine costs \( O(3^k \cdot k \cdot |E|) \).

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 \( O(3^k \cdot k \cdot |V| \cdot |E|) \).
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. 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 B.1), 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.

3.5 Vertex Cover

As noted in [28], an instance of OCT can be solved using a related instance of 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 \( G' = (V_1 \cup V_2, E_1 \cup E_2 \cup E') \), where \( V_i = \{v_i \mid v \in V\} \), \( E_i = \{(u_i, v_i) \mid (u, v) \in E \} \) (for \( i = 1, 2 \)), and \( E' = \{(v_1, v_2) \mid v \in V\} \). A solution \( S' \) for VC on \( G' \) can be mapped to a solution \( S \) for OCT on \( G \) with \( S = \{v \mid v_1 \in S \text{ and } v_2 \in S\} \).

This approach currently holds the fastest theoretical run time for OCT and other related problems [28]. Additionally, Akiba and Iwata recently surveyed VC techniques [3] and implemented a VC-solver in Java. We implement a Python wrapper that converts between OCT and VC, providing a common API with the other solvers.
### 3.6 Integer Linear Programming

In addition to solving OCT and VC combinatorially, both OCT and VC can be reformulated as **Integer Linear Programming** instances:

**Odd Cycle Transversal (ILP)** [20]

**Input:** \( G = (V, E) \).

Minimize \( \sum_{v \in V} c_v \)

s.t. \( s_v + s_u + c_v + c_u \geq 1 \) \( \forall (u, v) \in E \)

\( s_v + s_u - c_v - c_u \leq 1 \) \( \forall (u, v) \in E \)

\( s_v \in \{0, 1\} \) \( \forall v \in V \)

\( c_v \in \{0, 1\} \) \( \forall v \in V \)

**Vertex Cover (ILP)** [3]

**Input:** \( G = (V, E) \).

Minimize \( \sum_{v \in V} x_v \)

s.t. \( x_u + x_v \geq 1 \) \( \forall (u, v) \in E \)

\( x_v \in \{0, 1\} \) \( \forall v \in V \)

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 [20] and [3] for details on how these ILP formulations are derived.

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 B.2), 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 [28].

With hardware choice, 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 threads, 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 Data Benchmark and Code

In this section we detail the data used in the experiments, along with the code made available.
Table 1 A summary of preprocessing statistics on WH and quantum datasets. Ranges are given for the number of vertices, edge density, and optimal OCT solution size in both the original and reduced graphs. The statistic $|\hat{V}_r|$ reports the percentage of vertices removed. Likewise, normalized means are reported for edge removals $|\hat{E}_r|$, fixed-OCT vertices $|V_o|$, and fixed-bipartite vertices $|V_b|$.

| Dataset | $|V|$ | $|E|/|V|$ | $OPT$ | $|\hat{V}_r|$ | $|\hat{E}_r|$ | $|\hat{V}_b|$ | $|V'|/|V| \cup |V_b|$ | $OPT'$ |
|---------|------|-----------------|-------|-------------|---------|---------|------------------------------|------|
| WH-aa   | 14–300 | 1.0–5.4 | 0–40 | 13% | - | - | 62% | 0–265 | 2.0–6.1 | 0–40 |
| WH-j    | 14–241 | 0.7–5.9 | 0–19 | 41% | - | - | 10% | 0–74 | 2.0–7.5 | 0–19 |
| b-50    | 50    | 2.0–2.7 | 9–14 | 2% | - | - | 4% | 43–50 | 2.2–2.7 | 9–14 |
| b-100   | 100   | 4.6–5.1 | 41–44 | - | - | - | 100 | 4.6–5.1 | 41–44 |
| gka     | 20–125 | 2.1–61.3 | 10–120 | - | - | 18% | - | 0–100 | 2.0–43.5 | 0–92 |

4.1 Previous Data

As mentioned in Section 1.1, the primary dataset studied in OCT literature originates from Wernicke and is distributed with Höffner’s code. We refer to this data as the Wernicke-Höffner (WH) dataset. These datasets are originally from genetics through the related Minimum Site Removal problem \cite{40} 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). Although files aa-12, j-12, and j-27 are provided in Höffner’s code, they are empty and not included here.

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 \cite{3}. To introduce a new benchmark for OCT solvers, we concentrate on domain data from quantum computing. Specifically, a recent survey \cite{12} collected six datasets from the QUBO literature (see Section 2); of these, only the Beasley \cite{6} and GKA \cite{14} 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 1).

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.

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 Erdős-Rényi model \cite{13}, which takes as input a number of vertices $n$ and a probability $p$. Erdős-Rényi generates a graph by initializing $n$ vertices and adding each edge with probability $p$. 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 Erdős-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 [10]. 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 [5] 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 Replicability

All experiments are fully replicable with our open source code repository [16]. 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 sorted manually 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.

5 Quantum-Specific Results

5.1 Experimental Setup

All experiments were run on a Dell T20 workstation with an Intel Xeon E3-1225 v3 CPU, 16GB ECC DDR3 system memory in a dual-channel configuration, and an Intel S3500 solid state drive as storage. This workstation ran Debian 4.9.82-1 with Linux kernel 4.9.0-6, CPLEX 12.8, and GLPK 4.61. C/C++ code was compiled with gcc/g++ 6.3.0, Java code was compiled and run with OpenJDK 1.8.0_162, and Python code was run with Python 3.5.3 (the newest version supported by CPLEX 12.8).

5.2 Preprocessing Effectiveness

In this subsection we harden the benchmark by applying the reduction routines detailed in Section 3.1. We denote reductions as a partition of the original vertex set \(V = V_r \cup V_o \cup V_b \cup V'\).
Table 2: 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 $\text{OPT}$ on the reduced graph, computed with ILP. A checkmark denotes that the solver found exact solutions on all instances within the timeout; if a dataset has no checkmark then the best approximation algorithm is bolded.

| Dataset | 0.01(s) | 0.1(s) | 1(s) | 10(s) |
|---------|---------|--------|------|-------|
|       | HE     | IC     | ILP  | HE    | IC    | ILP  |
| WH-aa  | 1.38   | **1.29** | 8.05 | 1.27  | **1.17** | 2.19 |
|        |        |        |      |       |        |      |
|        | **1.19** | 1.12  | ✓    | **1.19** | 1.06  | ✓    |
| WH-j   | 1.11   | 1.09   | 1.60 | 1.11  | 1.07  | 1.21 |
|        |        |        |      |       |        |      |
|        | **1.08** | ✓     | ✓    | **1.08** | ✓     | ✓    |
| b-50   | 1.10   | 1.10   | 1.17 | 1.07  | 1.07  | 1.12 |
|        |        |        |      |       |        |      |
|        | **1.11** | 1.07  | 1.12 | **1.11** | 1.05  | **1.05** |
| b-100  | 1.21   | 1.18   | 1.34 | **1.18** | 1.18  | **1.18** |
|        |        |        |      |       | **1.11** | 1.14  |
|        | **1.11** | 1.14  | **1.07** | **1.11** | 1.14  | **1.07** |

Table 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.

| Graph | Solver | Graph | Solver |
|-------|--------|-------|--------|
| Dataset | OPT | VC | IC | ILP | Dataset | OPT | VC | IC | ILP |
| b-100-1 | 41 | 97.8 | - | 21.3 | gka-21 | 40 | 1.7 | 22.2 | 0.3 |
| b-100-2 | 42 | 185.5 | - | 52.7 | gka-22 | 43 | 5.4 | - | 1.0 |
| b-100-3 | 42 | 242.3 | - | 20.9 | gka-23 | 46 | 42.6 | - | 16.9 |
| b-100-4 | 41 | 199.8 | - | 33.1 | gka-24 | 37 | 74.3 | - | 17.3 |
| b-100-5 | 42 | 208.6 | - | 41.7 | gka-25 | 42 | 122.9 | - | 28.6 |
| b-100-6 | 43 | 143.6 | - | 99.6 | gka-26 | 43 | 160.5 | - | 18.4 |
| b-100-7 | 42 | 182.0 | - | 40.4 | gka-27 | 62 | 535.3 | - | 192.6 |
| b-100-8 | 43 | 356.7 | - | 39.3 | gka-28 | 70 | 284.9 | - | 366.0 |
| b-100-9 | 44 | 321.4 | - | 72.9 | gka-29 | 77 | 64.3 | - | 42.5 |
| b-100-10 | 44 | 188.6 | - | 118.3 | gka-30 | 82 | 32.7 | - | 53.2 |
| gka-3 | 23 | 1.4 | 58.0 | 0.9 | gka-31 | 85 | 11.7 | 107.6 | 17.4 |
| gka-8 | 28 | 7.2 | - | 4.8 | gka-32 | 88 | 5.3 | 17.9 | 19.2 |

The vertices that may be removed without changing $\text{OPT}$ are denoted $V_r$. For some fixed optimal 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 $E = E_r \cup E'$. 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 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 cut from 241 to 74. 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. As seen by the zeroes in the $|V' \cup V_b|$ column, both WH datasets and GKA had some instances that were completely solved by reduction routines alone.
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. 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 2 reports the worst-case approximation factors achieved by an algorithm-data-timeout triple; full results are reported in Appendix B (Tables 10 and 11). Notably, a user could achieve worst-case approximation factors of $1.29$ and $1.18$ for timeouts 0.01(s) and 0.1(s) (respectively) by solving with IC, and approximation factors of $1.10$ and $1.02$ for timeouts 1(s) and 10(s) by solving with ILP.

5.4 Use Case: Exact Solutions

In the second use case, a researcher may want to compute the limits of particular configurations supported on a fixed quantum hardware graph. 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).

In contrast to the results reported in [3], ILP came out the clear winner, sometimes by an order of magnitude. Iterative compression only finished a handful of times within 10 minutes, and only once was competitive to ILP (gka-32). These results naturally extend those in Table 2 where after 10 seconds the overhead imposed by CPLEX reduction routines begins to pay off significantly. Full data is available in Appendix B (Table 9).

6 Generalized Results

In this section, we expand the experimental envelope further by taking each WH or 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 4.3. In these experiments, a single representative graph is generated per model with a fixed seed.

6.1 Reduction Effectiveness

Table 4 depicts the effectiveness of reduction routines on the synthetic data. Across all 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.
| Dataset | | | | | | | |
|---------|---------|---------|---------|---------|---------|---------|---------|
| WH-aa   | 14–300  | 1.0–5.4 | 0–40    | 13%     | 0%      | 0%      | 62%     |
| aa-ER   | 14–300  | 1.1–5.4 | 1–34    | 2%      | -       | -       | 3%      |
| aa-TO   | 14–300  | 0.5–3.4 | 0–39    | 18%     | -       | -       | 13%     |
| aa-CL   | 14–300  | 1.0–5.4 | 2–59    | 50%     | -       | 6%      | 11%     |
| aa-BA   | 14–300  | 0.9–5.9 | 0–40    | 2%      | -       | -       | 3%      |
| WH-j    | 14–241  | 0.7–5.9 | 0–19    | 41%     | 0%      | 0%      | 10%     |
| j-ER    | 14–241  | 1.0–5.8 | 2–44    | 10%     | -       | -       | 4%      |
| j-TO    | 14–241  | 0.6–3.5 | 0–19    | 36%     | -       | -       | 5%      |
| j-CL    | 14–241  | 0.6–6.1 | 1–33    | 34%     | -       | 3%      | 6%      |
| j-BA    | 14–241  | 0.9–5.6 | 0–36    | 11%     | -       | 1%      | 1%      |
| b-50    | 50      | 2.0–2.7 | 9–14    | 2%      | 0%      | 0%      | 4%      |
| b-50-ER | 50      | 1.9–2.7 | 8–14    | 3%      | -       | -       | 4%      |
| b-50-TO | 50      | 1.2–2.1 | 3–10    | 13%     | -       | 6%      | 2%      |
| b-50-CL | 50      | 2.0–2.7 | 9–14    | 6%      | -       | -       | 5%      |
| b-50-BA | 50      | 2.8–2.8 | 10–40   | -       | -       | -       | 2%      |
| b-100   | 100     | 4.6–5.1 | 41–44   | 0%      | 0%      | 0%      | 0%      |
| b-100-ER| 100     | 4.3–5.0 | 40–45   | -       | -       | -       | 1%      |
| b-100-TO| 100     | 3.7–4.2 | 34–35   | -       | -       | 1%      | 1%      |
| b-100-CL| 100     | 4.7–5.1 | 38–40   | 1%      | -       | -       | 1%      |
| b-100-BA| 100     | 4.8–5.6 | 28–37   | -       | -       | -       | 1%      |
| gka     | 20–125  | 2.1–61.3| 10–120  | 0%      | 0%      | 18%     | 0%      |
| gka-ER  | 20–125  | 2.0–61.3| 8–121   | -       | -       | 18%     | -       |
| gka-TO  | 20–125  | 1.5–61.3| 4–120   | 1%      | -       | 19%     | 1%      |
| gka-CL  | 20–125  | 2.1–60.8| 9–119   | 1%      | -       | 15%     | -       |
| gka-BA  | 20–125  | 2.8–31.2| 8–62    | -       | -       | 2%      | -       |

Table 4 A summary of preprocessing statistics over all datasets. Ranges are given for the number of vertices, edge density, and optimal OCT solution size in both the original and reduced graphs. The statistic $|\hat{V}_r|$ 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 $V_b$.

### 6.2 Heuristic Solutions

When extending the heuristic solution comparison to synthetic data, we find the same best practices generally still apply. At 0.01 seconds, the best algorithm is split between HE and IC, depending on the dataset. At 0.1 seconds, IC pulls ahead, having more time to compress solutions but not enough time for ILP to finish its preprocessing. At 1 second and beyond, ILP dominates to the point of finding exact solutions for all of b-50.

We observe that over all data and algorithms, 0.01 seconds is enough to get a 2-approximation, 0.1 seconds for a 1.5-approximation, 1 second for a 1.12-approximation, and 10 seconds for a 1.05-approximation. These tight approximations show that OCT can be considered a practical subroutine when searching for a ‘good enough’ solution.
Table 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 $OPT$ on the reduced graph, computed with ILP. A checkmark denotes that the solver found exact solutions on all instances within the timeout; if a dataset has no checkmark then the best approximation algorithm is bolded.

6.3 Exact Solutions

Similar to Table 3, we again find that the ILP formulation solved using CPLEX dominates the VC-solver in general. Figure 2 shows that VC can be up to two magnitudes of order slower, specifically on GKA datasets. VC does win in certain circumstances, primarily on the Barabási-Albert analogues of instances from wh-aa.
Figure 2 Comparison of VC and ILP run times on synthetic data with a log scale. Data points above (below) the dashed line denote slower (faster) run times for VC in comparison to ILP, respectively.

7 Conclusion

We experimentally evaluated 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. On this significantly expanded corpus, we found that there is no single implementation that dominates in all scenarios. Under extreme time constraints, the low-overhead heuristic ensemble followed by iterative compression performs best. However, even with a 10 second time limit, heuristics quickly become ineffective when compared with modern ILP solvers (which come with the additional benefit of scaling with additional CPU and memory resources). If open source code is required, however, the Akiba-Iwata VC-solver becomes competitive. Our results identify several important directions for future work:

Development of lightweight solvers with efficient implementations. While suites such as CPLEX dominate the exact solver comparisons, there are opportunities to improve in the very short run time scenarios where low overhead is paramount. Efficient implementations of reduction routines, heuristics, and incremental improvement algorithms are needed here.

Increasing scalability of existing algorithms. Akiba-Iwata and Hüffner both leverage combinatorial structure for lightweight, competitive algorithms. However, both current implementations are written for serial processors and use relatively little system memory. CPU-based parallelism would benefit the branch-and-reduce strategy, while GPU-based Single-Instruction-Multiple-Data parallelism fits well with Hüffner’s compression routine.

Exploring formulations of OCT as ILP. We found that formulating OCT as VC led to a more efficient ILP formulation. Additional structures (such as those from biased graphs [39]) may lead to even faster ILP formulations.

Acknowledgments

This work supported in part by the Gordon & Betty Moore Foundations Data-Driven Discovery Initiative through Grant GBMF4560 to Blair D. Sullivan, and with Government support under and awarded by DoD, Air Force Office of Scientific Research, National Defense Science and Engineering Graduate (NDSEG) Fellowship, 32 CFR 168a to Timothy D. Goodrich.
References

1. Amit Agarwal, Moses Charikar, Konstantin Makarychev, and Yury Makarychev. $O(\sqrt{\log n})$ approximation algorithms for min UnCut, min 2CNF deletion, and directed cut problems. In Proceedings of the thirty-seventh annual ACM symposium on Theory of computing, pages 573–581, New York, NY, USA, 2005. ACM, ACM.

2. Muhammad Ahsan, Rodney Van Meter, and Jungsang Kim. Designing a million-qubit quantum computer using a resource performance simulator. ACM Journal on Emerging Technologies in Computing Systems (JETC), 12(4):39, 2016.

3. Takuya Akiba and Yoichi Iwata. Branch-and-reduce exponential/fpt algorithms in practice: A case study of vertex cover. Theoretical Computer Science, 609:211–225, 2016.

4. Takuya Akiba and Yoichi Iwata. Vertex cover solver, 2017. URL: https://github.com/wata-orz/vertex_cover.

5. Réka Albert and Albert-László Barabási. Statistical mechanics of complex networks. Reviews of modern physics, 74(1):47, 2002.

6. John E. Beasley. Heuristic algorithms for the unconstrained binary quadratic programming problem, 1998.

7. John E. Beasley. OR-Library, 2018. URL: http://people.brunel.ac.uk/~mastjjb/jeb/info.html.

8. Edward Boyda, Saikat Basu, Sangram Ganguly, Andrew Michaelis, Supratik Mukhopadhyay, and Ramakrishna R Nemani. Deploying a quantum annealing processor to detect tree cover in aerial imagery of California. PloS one, 12(2):e0172505, 2017.

9. Vicky Choi. Minor-embedding in adiabatic quantum computation: II. Minor-universal graph design. Quantum Information Processing, 10(3):343–353, 2011.

10. Fan Chung and Linyuan Lu. Connected components in random graphs with given expected degree sequences. Annals of combinatorics, 6(2):125–145, 2002.

11. Rigetti Computing. The Rigetti QPU, 2018. URL: http://pyquil.readthedocs.io/en/latest/qpu.html.

12. Iain Dunning, Swati Gupta, and John Silberholz. What works best when? a framework for systematic heuristic evaluation, 2015. URL: http://www.optimization-online.org/DB_FILE/2015/05/4895.pdf.

13. Paul Erdős and Alfréd Rényi. On the evolution of random graphs. Publ. Math. Inst. Hung. Acad. Sci, 5(1):17–60, 1960.

14. Fred Glover, Gary A. Kochenberger, and Bahram Alidaee. Adaptive memory tabu search for binary quadratic programs. Management Science, 44(3):336–345, 1998.

15. GNU. Linear Programming Kit (GLPK), Version 4.61, 2017. URL: http://www.gnu.org/software/glpk/glpk.html.

16. Timothy D. Goodrich, Eric Horton, and Blair D. Sullivan. Practical oct, 2018. URL: https://github.com/TheoryInPractice/practical-oct.

17. Timothy D. Goodrich, Blair D. Sullivan, and Travis S. Humble. Optimizing adiabatic quantum program compilation using a graph-theoretic framework. Quantum Information Processing, 17(5):118, Apr 2018.

18. Kathleen E. Hamilton and Travis S. Humble. Identifying the minor set cover of dense connected bipartite graphs via random matching edge sets. Quantum Information Processing, 16(4):94, 2017.

19. Falk Hüffner. occ, 2006. URL: http://theinf1.informatik.uni-jena.de/occ/.

20. Falk Hüffner. Algorithm engineering for optimal graph bipartization. Journal of Graph Algorithms and Applications, 13(2):77–98, 2009.

21. IBM. CPLEX Optimization Studio 12.8, 2017. URL: https://www.ibm.com/analytics/data-science/prescriptive-analytics/cplex-optimizer.
IBM. IBM announces advances to IBM Quantum Systems & Ecosystem, 2018. URL: http://www-03.ibm.com/press/us/en/pressrelease/53374.wss.

Yoichi Iwata, Keigo Oka, and Yuichi Yoshida. Linear-time fpt algorithms via network flow. In Proceedings of the twenty-fifth annual ACM-SIAM symposium on Discrete algorithms, pages 1749–1761, Philadelphia, PA, USA, 2014. Society for Industrial and Applied Mathematics, Society for Industrial and Applied Mathematics.

Yoichi Iwata, Yutaro Yamaguchi, and Yuichi Yoshida. 0/1/all CSPs, half-integral A-path packing, and linear-time fpt algorithms, 2017.

Julian Kelly. A preview of Bristlecone, Google’s new quantum processor, 2018. URL: https://ai.googleblog.com/2018/03/a-preview-of-bristlecone-googles-new.html.

Bart Kosko. Bidirectional associative memories. IEEE Transactions on Systems, man, and Cybernetics, 18(1):49–60, 1988.

Stefan Kratsch and Magnus Wahlström. Compression via matroids: a randomized polynomial kernel for odd cycle transversal. ACM Transactions on Algorithms (TALG), 10(4):20, 2014.

Daniel Lokshtanov, NS Narayanaswamy, Venkatesh Raman, MS Ramamujan, and Saket Saurabh. Faster parameterized algorithms using linear programming. ACM Transactions on Algorithms (TALG), 11(2):15, 2014.

Daniel Lokshtanov, Saket Saurabh, and Somnath Sikdar. Simpler parameterized algorithm for oct. In International Workshop on Combinatorial Algorithms, pages 380–384, Hradec nad Moravici, Czech Republic, 2009. Springer, Springer.

Daniel Lokshtanov, Saket Saurabh, and Magnus Wahlström. Subexponential Parameterized Odd Cycle Transversal on Planar Graphs. In Deepak D’Souza, Telikepalli Kavitha, and Jaikumar Radhakrishnan, editors, IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2012), volume 18 of Leibniz International Proceedings in Informatics (LIPIcs), pages 424–434, Dagstuhl, Germany, 2012. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. URL: http://drops.dagstuhl.de/opus/volltexte/2012/3878, doi:10.4230/LIPIcs.FSTTCS.2012.424.

Michael Luby. A simple parallel algorithm for the maximal independent set problem. SIAM journal on computing, 15(4):1036–1053, 1986.

Andrew Lucas. Ising formulations of many NP problems. Frontiers in Physics, 2:5, 2014.

Hartmut Neven, Geordie Rose, and William G. Macready. Image recognition with an adiabatic quantum computer I. mapping to quadratic unconstrained binary optimization, 2008.

Thomas E Potok, Catherine Schuman, Steven Young, Robert Patton, Federico Spedalieri, Jeremy Liu, Ke-Thia Yao, Garrett Rose, and Gangotree Chakma. 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):19, 2018.

Bruce Reed, Kaleigh Smith, and Adrian Vetta. Finding odd cycle transversals. Operations Research Letters, 32(4):299–301, 2004.

Jonathan Schrock, Alex J. McCaskey, Kathleen E. Hamilton, Travis S. Humble, and Neena Imam. Recall performance for content-addressable memory using adiabatic quantum optimization. Entropy, 19(9):500, 2017.

Davide Venturelli, Minh Do, Eleanor Rieffel, and Jeremy Frank. Compiling quantum circuits to realistic hardware architectures using temporal planners. Quantum Science and Technology, 3(2):025004, 2018.

Davide Venturelli, Salvatore Mandra, Sergey Kravsh, Bryan O’Gorman, Rupak Biswas, and Vadim Smelyanskiy. Quantum optimization of fully connected spin glasses. Physical Review X, 5(3):031040, 2015.
Practical Graph Bipartization

39  Magnus Wahlström. LP-branching algorithms based on biased graphs. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1559–1570, Philadelphia, PA, USA, 2017. Society for Industrial and Applied Mathematics, Society for Industrial and Applied Mathematics.

40  Sebastian Wernicke. On the algorithmic tractability of single nucleotide polymorphism (snP) analysis and related problems, 2003.
A Implementation Details

A.1 Data Ingestion and Sanitization

Original data comes from two sources. Wernicke-Hüffner data is provided in the Hüffner code download [19], and Beasley and GKA data comes from Beasley’s repository [7].

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 \( \{0, \ldots, n - 1\} \). 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.

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.

A.2 Reduction Routines

Reduction routines for preprocessing come from two papers: Wernicke [10] and Akiba and Iwata [3].

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 nondeterministically 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 find and remove particular configurations in the graph based on degree 2 and 3 vertices; we sort these sets of vertices and the related neighborhoods.

For Akiba and Iwata’s reduction routines, we modify their GitHub code [4] 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 nondeterministic 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.

A.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 [17] for more on MinDeg, [31] for more on Luby, and [40] for more on DFS.
See the Heuristics section of REPLICABILITY.md in our repository for information on how to run the heuristic ensemble solver.

A.4 Hüffner Improvements

We implemented our improvements to Hüffner’s implementation [19] 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 that the most number of edges appear in the subgraph the earliest (\( p = 2 \)).

See the Iterative Compression section of REPLICABILITY.md in our repository for information on how to download, install, and run this improved iterative compression solver.

B Extended Results

B.1 Iterative Compression Heuristic Evaluation

We implement our heuristic improvements in a version of Hüffner’s code improved for simplicity and compiler compatibility. Recall the preprocessing parameter \( p \) defined in Appendix A.4. To evaluate these preprocessing options, we selected a subset of the data (c.f., Table 7) with a mixture of easy and difficult problems. We then ran all three levels using 50 random seeds on each instance for each of four timeouts \{0.01, 0.1, 1, 10\}, and report mean, standard deviation, and quintiles of the OCT sizes found. Table 6 depicts these results. We find that \( p = 1 \) always dominates \( p = 0 \), especially in max and standard deviation, suggesting that certain orderings (avoided with \( p = 1 \)) are significantly disadvantageous for iterative compression. Second, we find that \( p = 2 \) tends to help on larger timeouts, where the run time cost of computing this ordering disappears. Both observations can be seen in Table 6 where aa-41 and aa-42 benefit slightly from \( p = 2 \), but gka-2 and gka-3 do not.

B.2 ILP Solver Comparison

As noted in the introduction, previous work [20, 3] has conflicting reports on the effectiveness of Integer Linear Programming (ILP) solvers. In this section we identify the best
| Dataset   | p | Mean  | S.D.  | Min  | 25% | 50% | 75% | Max  |
|-----------|---|-------|-------|------|-----|-----|-----|------|
| **Timeout: 0.01 (s)** |     |       |       |      |     |     |     |      |
| aa-43     | 0 | 21.4  | 3.0   | 18   | 19  | 21  | 23  | 29   |
|           | 1 | 18.9  | 0.7   | 18   | 18  | 19  | 19  | 20   |
|           | 2 | 19.5  | 0.6   | 18   | 19  | 20  | 20  | 20   |
| aa-45     | 0 | 29.6  | 4.2   | 22   | 27  | 30  | 32  | 41   |
|           | 1 | 21.8  | 0.9   | 20   | 21  | 22  | 22  | 24   |
|           | 2 | 22.7  | 0.7   | 21   | 22  | 23  | 23  | 24   |
| gka-1     | 0 | 9.0   | 0.0   | 9    | 9   | 9   | 9   | 9    |
|           | 1 | 9.0   | 0.0   | 9    | 9   | 9   | 9   | 9    |
|           | 2 | 9.0   | 0.0   | 9    | 9   | 9   | 9   | 9    |
| **Timeout: 0.1 (s)** |     |       |       |      |     |     |     |      |
| aa-29     | 0 | 35.9  | 16.2  | 21   | 23  | 23  | 23  | 26   |
|           | 1 | 22.2  | 1.3   | 21   | 21  | 22  | 23  | 26   |
|           | 2 | 23.9  | 1.7   | 21   | 23  | 24  | 25  | 28   |
| **Timeout: 1 (s)** |     |       |       |      |     |     |     |      |
| aa-42     | 0 | 51.8  | 13.4  | 33   | 42  | 49  | 59  | 85   |
|           | 1 | 31.0  | 1.2   | 30   | 30  | 31  | 32  | 35   |
|           | 2 | 31.9  | 0.3   | 31   | 32  | 32  | 32  | 32   |
| gka-2     | 0 | 16.8  | 0.9   | 16   | 16  | 17  | 17  | 19   |
|           | 1 | 16.0  | 0.0   | 16   | 16  | 16  | 16  | 16   |
|           | 2 | 16.0  | 0.0   | 16   | 16  | 16  | 16  | 16   |
| **Timeout: 10 (s)** |     |       |       |      |     |     |     |      |
| aa-32     | 0 | 31.6  | 2.4   | 30   | 30  | 30  | 32  | 38   |
|           | 1 | 30.8  | 0.9   | 30   | 30  | 31  | 31  | 33   |
|           | 2 | 30.8  | 0.4   | 30   | 31  | 31  | 31  | 31   |
| aa-41     | 0 | 67.0  | 14.4  | 48   | 55  | 64  | 75  | 110  |
|           | 1 | 41.4  | 1.2   | 40   | 40  | 41  | 42  | 44   |
|           | 2 | 41.2  | 0.7   | 40   | 41  | 41  | 41  | 43   |
| gka-24    | 0 | 47.1  | 1.6   | 44   | 46  | 47  | 48  | 50   |
|           | 1 | 37.9  | 0.3   | 37   | 38  | 38  | 38  | 38   |
|           | 2 | 37.9  | 0.3   | 37   | 38  | 38  | 38  | 38   |
| gka-25    | 0 | 56.2  | 2.6   | 49   | 55  | 56  | 58  | 62   |
|           | 1 | 44.0  | 0.0   | 44   | 44  | 44  | 44  | 44   |
|           | 2 | 44.0  | 0.0   | 44   | 44  | 44  | 44  | 44   |
| gka-26    | 0 | 57.0  | 2.3   | 52   | 55  | 57  | 58  | 62   |
|           | 1 | 43.8  | 0.4   | 43   | 44  | 44  | 44  | 44   |
|           | 2 | 43.8  | 0.4   | 43   | 44  | 44  | 44  | 44   |
| gka-27    | 0 | 72.3  | 1.7   | 68   | 71  | 72  | 74  | 75   |
|           | 1 | 63.0  | 0.1   | 62   | 63  | 63  | 63  | 63   |
|           | 2 | 63.0  | 0.0   | 63   | 63  | 63  | 63  | 63   |
| gka-28    | 0 | 78.0  | 1.2   | 75   | 77  | 78  | 79  | 80   |
|           | 1 | 72.0  | 0.2   | 71   | 72  | 72  | 72  | 73   |
|           | 2 | 72.0  | 0.2   | 72   | 72  | 72  | 72  | 73   |
| gka-29    | 0 | 82.4  | 0.8   | 81   | 82  | 82  | 83  | 84   |
|           | 1 | 77.0  | 0.0   | 77   | 77  | 77  | 77  | 77   |
|           | 2 | 77.0  | 0.0   | 77   | 77  | 77  | 77  | 77   |
| gka-3     | 0 | 25.2  | 1.6   | 23   | 24  | 25  | 26  | 29   |
|           | 1 | 23.3  | 0.5   | 23   | 23  | 23  | 24  | 24   |
|           | 2 | 23.4  | 0.5   | 23   | 23  | 23  | 24  | 24   |

Table 6 Heuristic solution sizes for select datasets at the three levels of preprocessing ($p \in \{0, 1, 2\}$) for Improved Hüffner. The timeout level on each dataset was taken to be the maximum time in \{0.01, 0.1, 1, 10\} less than the time Hüffner at $p = 0$ took to find an exact solution. Mean, standard deviation, and quintiles are computed over 50 samples per timeout, dataset, and preprocessing level.
configuration for solving OCT with ILP with the following options: choosing a solver from CPLEX \cite{21} and GLPK \cite{15}, choosing a formulation from OCT $\rightarrow$ ILP and OCT $\rightarrow$ VC $\rightarrow$ ILP, having one thread or multithreaded (4 physical cores), and having limited system memory (4MB) or plentiful memory (16GB). This memory limit is significant because \cite{20} notes that Hüffner’s implementation only requires 4MB of memory and there may be a use case that requires minimal memory usage. We find that CPLEX and OCT $\rightarrow$ VC $\rightarrow$ ILP result in the fastest solutions, that multithreading can result in superlinear speedup, and that increased system memory can provide up to a $2 \times$ speedup depending on the instance.

To evaluate ILP-based approaches, we use the same subset of data selected for the IC extended experiment and solve with all possible combinations of solver, threads, and OCT formulation described above (denoted $\text{Solver}_{\text{Formulation}}^{\text{Threads}}$). (Note that GLPK is single-threaded).

Comparing solvers, CPLEX dominated GLPK when using the same formulation, regardless of number of threads or system memory levels. Table \ref{table:7} depicts the relative run times when fixing the formulation to OCT $\rightarrow$ VC $\rightarrow$ ILP and system memory to 16GB. Interestingly,

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|c|c|c|}
\hline
Dataset & $|V|$ & $|E|$ & OPT & Time (s) & CPLEX$^{\text{VC}}_4$ & CPLEX$^{\text{ILP}}_4$ & GLPK$^{\text{ILP}}_4$ \\
\hline
\hline
gka-1   & 44  & 90  & 9   & 0.1   & 1.0$\times$ & 1.0$\times$ & 3.0$\times$ \\
\hline
aa-43   & 58  & 303 & 18  & 0.1   & 1.0$\times$ & 1.0$\times$ & 163.0$\times$ \\
\hline
aa-45   & 75  & 381 & 20  & 0.1   & 1.0$\times$ & 1.0$\times$ & 389.0$\times$ \\
\hline
gka-2   & 58  & 149 & 16  & 0.1   & 1.0$\times$ & 4.0$\times$ & 206.0$\times$ \\
\hline
aa-29   & 265 & 1048 & 30 & 0.3   & 1.0$\times$ & 4.0$\times$ & - \\
\hline
aa-32   & 126 & 733 & 20  & 0.4   & 1.0$\times$ & 3.0$\times$ & - \\
\hline
gka-3   & 70  & 223 & 23  & 0.8   & 1.0$\times$ & 5.6$\times$ & - \\
\hline
aa-41   & 261 & 1585 & 40 & 0.7   & 1.0$\times$ & 2.3$\times$ & - \\
\hline
gka-26  & 100 & 494 & 43  & 16.8  & 1.0$\times$ & 3.3$\times$ & - \\
\hline
\hline
\end{tabular}
\caption{Relative run times for solving exactly with the OCT $\rightarrow$ VC $\rightarrow$ ILP formulation and 16GB of memory. Multithreaded CPLEX is used as the baseline time. Times are not reported for configurations that required more than 10 minutes.}
\end{table}
### Preprocessed Graph Solver

| Dataset | $|V|$ | $|E|$ | OPT | Time (s) | CPLEX$_1^{VC}$ | CPLEX$_2^{VC}$ | GLPK$_1^{VC}$ |
|---------|-----|-----|-----|---------|----------------|----------------|--------------|
| gka-1   | 44  | 90  | 9   | 0.1     | 1.0×           | 1.0×           | -            |
| aa-43   | 58  | 303 | 18  | 0.1     | 1.0×           | 1.0×           | -            |
| aa-45   | 75  | 381 | 20  | 0.1     | 1.0×           | 1.0×           | -            |
| gka-2   | 58  | 149 | 16  | 0.1     | 1.0×           | 4.0×           | -            |
| aa-29   | 265 | 1048| 21  | 0.1     | 1.0×           | 4.0×           | -            |
| aa-42   | 226 | 1100| 30  | 0.3     | 1.0×           | 4.0×           | -            |
| aa-32   | 126 | 733 | 30  | 0.4     | 1.0×           | 3.0×           | -            |
| gka-3   | 70  | 223 | 23  | 0.7     | 1.0×           | 6.4×           | -            |
| aa-41   | 261 | 1585| 40  | 0.7     | 1.0×           | 2.3×           | -            |
| gka-26  | 100 | 494 | 43  | 26.1    | 1.0×           | 2.1×           | -            |
| gka-24  | 90  | 400 | 37  | 20.7    | 1.0×           | 4.9×           | -            |
| gka-25  | 100 | 495 | 42  | 35.3    | 1.0×           | 8.5×           | -            |
| gka-29  | 100 | 2000| 77  | 44.1    | 1.0×           | 10.5×          | -            |
| gka-27  | 100 | 1016| 62  | 139.0   | 1.0×           | 3.2×           | -            |
| gka-28  | 100 | 1425| 70  | 317.1   | 1.0×           | 1.4×           | -            |

**Table 8** Relative run times for solving exactly with the OCT → VC → ILP formulation and 4MB of memory. Multithreaded CPLEX is used as the baseline time. Times are not reported for configurations that required more than 10 minutes.

**B.3 Additional Tables**

CPLEX$_1^{VC}$ often performed over four times faster than its single-threaded variant.

Comparing problem formulations, OCT → VC → ILP also dominated OCT → ILP (Figure 3). This result may not be surprising given [28], but it suggests that there may also be alternative, faster formulations of OCT as an ILP that may lead to even better results. In Figure 4 and Table 8, we find that limiting the solvers to 4MB of system memory does not change the relative performance landscape, and only doubles the run time of the slowest dataset gka-28.
| Graph     | Solver | Dataset | OPT | VC | IC | ILP |
|-----------|--------|---------|-----|----|----|-----|
| aa-10     | 6      | 0.2     | 1.0 | 0.0|
| aa-11     | 1      | 1.0     | 0.1 | 0.0|
| aa-13     | 12     | 0.1     | 1.0 | 0.1|
| aa-14     | 19     | 0.2     | 4.7 | 0.1|
| aa-15     | 6      | 0.1     | 1.0 | 0.0|
| aa-16     | 0      | 0.1     | 1.0 | 0.0|
| aa-17     | 25     | 0.4     | 6.4 | 0.1|
| aa-18     | 14     | 0.1     | 1.0 | 0.0|
| aa-19     | 19     | 0.2     | 2.7 | 0.2|
| aa-20     | 19     | 0.3     | 1.1 | 0.1|
| aa-21     | 0      | 0.1     | 1.0 | 0.0|
| aa-22     | 16     | 0.2     | 1.0 | 0.1|
| aa-23     | 18     | 0.2     | 1.0 | 0.1|
| aa-24     | 21     | 0.3     | 1.4 | 0.1|
| aa-25     | 12     | 0.1     | 1.0 | 0.0|
| aa-26     | 12     | 0.1     | 1.0 | 0.0|
| aa-27     | 11     | 0.2     | 1.0 | 0.1|
| aa-28     | 27     | 0.2     | 1.1 | 0.1|
| aa-29     | 21     | 0.3     | 1.1 | 0.1|
| aa-30     | 4      | 0.1     | 1.0 | 0.0|
| aa-31     | 20     | 0.1     | 1.0 | 0.0|
| aa-32     | 30     | 0.6     | 10.2| 0.5|
| aa-33     | 4      | 0.1     | 1.0 | 0.0|
| aa-34     | 13     | 0.2     | 1.0 | 0.1|
| aa-35     | 10     | 0.1     | 1.0 | 0.0|
| aa-36     | 6      | 0.1     | 1.0 | 0.0|
| aa-37     | 4      | 0.1     | 1.0 | 0.0|
| aa-38     | 26     | 0.3     | 1.5 | 0.1|
| aa-39     | 23     | 0.3     | 2.7 | 0.1|
| aa-40     | 22     | 0.3     | 1.5 | 0.1|
| aa-41     | 40     | 0.8     | 48.0| 0.4|
| aa-42     | 30     | 0.4     | 2.1 | 0.3|
| aa-43     | 18     | 0.1     | 1.0 | 0.1|
| aa-44     | 10     | 0.1     | 1.0 | 0.0|
| aa-45     | 20     | 0.2     | 1.1 | 0.1|
| aa-46     | 13     | 0.2     | 1.0 | 0.1|
| aa-47     | 13     | 0.1     | 1.0 | 0.0|
| aa-48     | 17     | 0.1     | 1.0 | 0.1|
| aa-49     | 0      | 0.1     | 1.0 | 0.0|
| aa-50     | 18     | 0.2     | 1.0 | 0.1|
| aa-51     | 11     | 0.1     | 1.0 | 0.0|
| aa-52     | 12     | 0.1     | 1.0 | 0.0|
| aa-53     | 12     | 0.1     | 1.0 | 0.0|
| aa-54     | 12     | 0.2     | 1.0 | 0.1|
| j-10      | 3      | 0.1     | 1.0 | 0.0|
| j-11      | 5      | 0.1     | 1.0 | 0.0|
| j-13      | 4      | 0.1     | 1.0 | 0.0|
| j-14      | 3      | 0.1     | 1.0 | 0.0|
| j-15      | 0      | 0.1     | 1.0 | 0.0|
| j-16      | 0      | 0.1     | 1.0 | 0.0|
| j-17      | 0      | 0.1     | 1.0 | 0.0|
| j-18      | 9      | 0.1     | 1.0 | 0.0|
| j-19      | 3      | 0.1     | 1.0 | 0.0|
| j-20      | 0      | 0.1     | 1.0 | 0.0|
| j-21      | 2      | 0.1     | 1.0 | 0.0|
| j-22      | 9      | 0.1     | 1.0 | 0.0|
| j-23      | 19     | 0.1     | 1.1 | 0.1|
| j-24      | 3      | 0.1     | 1.0 | 0.0|

Table 9 Run times for solvers VC, iterative compression IC, and CPLEX (ILP). Times are not reported for configurations that required more than 10 minutes.
| Dataset | Wernicke-Hüffner Afro-American Graphs | Wernicke-Hüffner Japanese Graphs |
|---------|--------------------------------------|----------------------------------|
|         | HE IC ILP | HE IC ILP | HE IC ILP | HE IC ILP | HE IC ILP | HE IC ILP |
| aa-10   | 6 7 6 8   | 6 6 6 6   | 6 6 6 6   | 6 6 6 6   | 6 6 6 6   | 6 6 6 6   |
| aa-11   | 11 12 11 15 | 11 12 11 13 | 11 12 11 13 | 11 12 11 11 | 11 12 11 11 |
| aa-13   | 12 15 12 22 | 15 12 12 14 | 12 12 12 13 | 12 12 12 12 | 12 12 12 12 |
| aa-14   | 19 21 19 35 | 19 19 19 19 | 19 19 19 19 | 19 19 19 19 | 19 19 19 19 |
| aa-16   | 6 8 6 6   | 6 6 6 6   | 6 6 6 6   | 6 6 6 6   | 6 6 6 6   | 6 6 6 6   |
| aa-17   | 0 0 0 0   | 0 0 0 0   | 0 0 0 0   | 0 0 0 0   | 0 0 0 0   | 0 0 0 0   |
| aa-18   | 25 28 28 41 | 28 27 34 26 27 25 26 25 25 | 25 25 25 |
| aa-19   | 14 18 14 21 | 17 14 14 14 | 14 14 14 14 | 14 14 14 14 |
|         | 19 22 22 38 | 21 21 20 21 19 19 21 19 19 19 |
| aa-20   | 19 23 24 129 | 20 22 20 20 22 20 19 22 20 19 |
| aa-21   | 0 0 0 0   | 0 0 0 0   | 0 0 0 0   | 0 0 0 0   | 0 0 0 0   | 0 0 0 0   |
| aa-22   | 16 17 37 17 | 17 16 16 17 17 16 17 16 17 16 |
| aa-23   | 18 21 21 35 | 21 18 19 19 18 18 19 18 18 |
| aa-24   | 21 25 27 150 | 25 24 46 24 21 21 24 21 21 |
| aa-25   | 0 0 0 0   | 0 0 0 0   | 0 0 0 0   | 0 0 0 0   | 0 0 0 0   | 0 0 0 0   |
| aa-26   | 12 13 12 14 | 12 12 12 12 12 12 12 12 12 12 |
| aa-27   | 11 14 11 13 | 14 11 11 13 13 11 11 12 11 11 |
| aa-28   | 27 33 32 106 | 31 29 32 30 27 27 30 28 27 |
| aa-29   | 21 26 26 160 | 22 24 24 22 21 21 22 21 21 |
| aa-30   | 4 4 4 4   | 4 4 4 4   | 4 4 4 4   | 4 4 4 4   | 4 4 4 4   |
| aa-31   | 0 0 0 0   | 0 0 0 0   | 0 0 0 0   | 0 0 0 0   | 0 0 0 0   |
| aa-32   | 30 39 33 47 | 32 35 37 32 33 30 32 30 30 |
| aa-33   | 4 4 4 31 4 4 4 4 4 4 4 4 4 4 4 4 |
| aa-34   | 13 16 13 27 | 13 13 13 13 13 13 13 13 13 |
| aa-35   | 10 10 10 10 | 10 10 10 10 10 10 10 10 10 |
| aa-36   | 7 7 7 18 7 7 7 7 7 7 7 7 7 7 7 7 |
| aa-37   | 4 5 4 7   | 4 4 4 4   | 4 4 4 4   | 4 4 4 4   | 4 4 4 4   |
| aa-38   | 26 33 31 102 | 31 27 35 31 26 26 31 26 26 |
| aa-39   | 23 28 27 40 | 26 26 26 24 25 23 24 23 24 23 |
| aa-40   | 22 25 28 33 | 22 23 28 22 22 22 22 22 22 22 22 |
| aa-41   | 40 50 46 172 | 48 45 60 46 45 40 46 41 40 |
| aa-42   | 30 36 35 160 | 36 33 43 34 32 30 34 30 30 30 |
| aa-43   | 18 19 19 20 | 19 19 19 19 19 19 19 19 19 19 19 19 19 19 19 19 19 19 19 |
| aa-44   | 10 10 10 15 | 10 10 12 | 10 10 10 | 10 10 10 | 10 10 10 |
| aa-45   | 20 22 22 20 | 21 22 20 | 21 20 20 | 21 20 20 | 21 20 20 |
| aa-46   | 13 18 13 37 | 16 13 13 | 14 13 13 | 14 13 13 | 14 13 13 |
| aa-47   | 13 14 13 15 | 14 13 13 | 13 13 13 | 13 13 13 | 13 13 13 |
| aa-48   | 17 19 18 21 | 17 17 17 | 17 17 17 | 17 17 17 | 17 17 17 |
| aa-49   | 0 0 0 6   | 0 0 0 0 | 0 0 0 0 | 0 0 0 0 | 0 0 0 0 |
| aa-50   | 18 21 19 27 | 21 18 19 | 20 18 18 | 19 18 18 | 19 18 18 |
| aa-51   | 11 13 11 16 | 11 11 11 | 11 11 11 | 11 11 11 | 11 11 11 |
| aa-52   | 12 13 12 15 | 12 12 12 | 12 12 12 | 12 12 12 | 12 12 12 |
| aa-53   | 12 15 12 15 | 14 12 13 | 13 12 12 | 13 12 12 | 13 12 12 |
| aa-54   | 12 14 12 21 | 14 12 14 | 13 12 12 | 12 12 12 | 12 12 12 |

**Table 10.** Heuristic solution sizes after 0.01, 0.1, 1, and 10 seconds for the heuristic ensemble (HE), iterative compression IC, and CPLEX (ILP) when run on WH datasets. Highlighting indicates instances where the solver found an exact solution before the provided time limit.
| Dataset          | OPT | 0.01(s) | 0.1(s) | 1(s) | 10(s) |
|------------------|-----|---------|--------|------|-------|
| **Beasley 50-Vertex Graphs** |     |         |        |      |       |
| b-50-1           | 11  | 12      | 11     | 11   | 11    |
| b-50-2           | 11  | 11      | 11     | 11   | 11    |
| b-50-3           | 14  | 13      | 13     | 13   | 13    |
| b-50-4           | 11  | 12      | 12     | 11   | 11    |
| b-50-5           | 13  | 13      | 13     | 13   | 13    |
| b-50-6           | 9   | 9       | 9      | 9    | 9     |
| b-50-7           | 9   | 10      |        | 9    | 9     |
| b-50-8           | 14  | 15      | 15     | 14   | 14    |
| b-50-9           | 11  | 11      | 11     | 11   | 11    |
| b-50-10          | 11  | 12      | 12     | 11   | 11    |
| **Beasley 100-Vertex Graphs** |     |         |        |      |       |
| b-100-1          | 41  | 43      | 44     | 43   | 43    |
| b-100-2          | 42  | 44      | 44     | 43   | 43    |
| b-100-3          | 42  | 46      | 45     | 44   | 44    |
| b-100-4          | 41  | 43      | 45     | 42   | 42    |
| b-100-5          | 42  | 45      | 44     | 43   | 43    |
| b-100-6          | 43  | 45      | 46     | 45   | 45    |
| b-100-7          | 42  | 45      | 45     | 44   | 44    |
| b-100-8          | 43  | 47      | 46     | 45   | 45    |
| b-100-9          | 44  | 46      | 47     | 45   | 45    |
| b-100-10         | 44  | 46      | 48     | 45   | 45    |
| **GKA Graphs**   |     |         |        |      |       |
| gka-1            | 9   | 9       | 12     | 9    | 9     |
| gka-2            | 16  | 16      | 16     | 16   | 16    |
| gka-3            | 23  | 24      | 24     | 23   | 23    |
| gka-4            | 28  | 31      | 33     | 30   | 30    |
| gka-5            | 22  | 22      | 22     | 22   | 22    |
| gka-6            | 16  | 16      | 16     | 16   | 16    |
| gka-7            | 18  | 18      | 18     | 18   | 18    |
| gka-8            | 28  | 34      | 33     | 33   | 33    |
| gka-9            | 2   | 2       | 2      | 2    | 2     |
| gka-10           | 8   | 8       | 8      | 8    | 8     |
| gka-11           | 10  | 10      | 10     | 10   | 10    |
| gka-12           | 20  | 20      | 20     | 20   | 20    |
| gka-13           | 16  | 16      | 16     | 16   | 16    |
| gka-14           | 22  | 22      | 22     | 22   | 22    |
| gka-15           | 0   | 0       | 0      | 0    | 0     |
| gka-16           | 0   | 0       | 0      | 0    | 0     |
| gka-17           | 0   | 0       | 0      | 0    | 0     |
| gka-18           | 2   | 2       | 2      | 2    | 2     |
| gka-19           | 31  | 31      | 32     | 31   | 31    |
| gka-20           | 39  | 39      | 39     | 39   | 39    |
| gka-21           | 40  | 41      | 42     | 41   | 41    |
| gka-22           | 43  | 46      | 46     | 46   | 46    |
| gka-23           | 46  | 48      | 48     | 47   | 46    |
| gka-24           | 37  | 39      | 38     | 38   | 38    |
| gka-25           | 42  | 45      | 46     | 44   | 44    |
| gka-26           | 43  | 44      | 44     | 44   | 44    |
| gka-27           | 62  | 65      | 64     | 63   | 63    |
| gka-28           | 70  | 73      | 73     | 72   | 72    |
| gka-29           | 77  | 77      | 77     | 77   | 77    |
| gka-30           | 82  | 83      | 84     | 83   | 83    |
| gka-31           | 85  | 86      | 87     | 86   | 86    |
| gka-32           | 88  | 89      | 90     | 89   | 88    |
| gka-33           | 90  | 91      | 91     | 90   | 90    |
| gka-34           | 92  | 93      | 93     | 92   | 92    |
| gka-35           | 0   | 0       | 0      | 0    | 0     |

Table 11: Heuristic solution sizes after 0.01, 0.1, 1, and 10 seconds for the heuristic ensemble (HE), iterative compression IC, and CPLEX (ILP) when run on Beasley and GKA datasets. Highlighting indicates instances where the solver found an exact solution before the provided time limit.