Universiteit Leiden
The Netherlands

Bachelor Computer Science

Using Binary Decision Diagrams for Exploring Graph State Orbits

S.T.A.N. van Baarsen

First supervisor and second supervisor:
A.W. Laarman & S.O. Brand

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

May 31, 2024

Abstract

Graph state preparation plays an essential role in various quantum computing algorithms, and finding the minimum-edge graph state that is reachable under local complementation (LC) from a target state is often a crucial step in this. This thesis explores the usefulness of binary decision diagrams (BDDs) in the computation of these minimum-edge graphs by using them to compute LC-equivalence orbits: sets of all graphs reachable from a given graph state under LC. We contribute to existing tools by developing a program for calculating and storing LC orbits in BDDs using the `sylvan C++` BDD library. Our approach significantly outperforms a standard brute-force approach (only) for graphs of eight nodes or more. Additionally, we generated a dataset of orbits covering all more than 270 million graphs with up to 8 nodes using our program.

Unlike other methods, our approach did not merge isomorphic graphs, but computed and stored sets of arbitrary graphs. While this made our program more versatile, it added significant complexity to the computation, prompting further research to establish the efficiency of BDDs in orbit computation when isomorphic graphs are merged.

Contents

1 Introduction

Since the 1980s, scientists and engineers have been researching a new form of computation: quantum computing. Unlike conventional computers, quantum computers operate on qubits, making use of the quantum-mechanical properties of particles to perform probabilistic computation. Oftentimes, before a quantum algorithm can be run, it is necessary to set up the quantum state in the computer as a specific target state. *Graph states* form an important subset of these quantum states, as they encompass all quantum states in which the qubits and their entanglement can be represented by an undirected graph (as its nodes and edges, respectively) [13]. Specifically in measurement-based quantum computing, quantum networking, and quantum error correction, this *graph state preparation* is an important process in ensuring the proper execution of any algorithm [13].

Preparation of a graph state from an unentangled empty-graph state can always be performed through combinations of two operations on the graph state: performing local complementation, which corresponds to single-qubit Clifford gates, and applying Controlled-Z (CZ) gates. Whereas CZ is a more flexible operation, allowing for the addition or removal of any given edge in the graph state, LC is a more delicate operation that changes multiple edges simultaneously.

However, the fact that CZ gates operate on two qubits has two critical drawbacks. First, two-qubit operations introduce extra noise to the system, increasing the probability of an error arising during state preparation. Second, due to this extra noise and thus the higher likelihood of failure (and of more retries), they can severely increase the time that the preparation process takes, particularly in quantum networking settings where qubits are separated by a large physical distance.

To minimize these errors, it is best to apply as few (two-qubit) CZ operations as possible and perform as much of the preparation as possible by applying (one-qubit) LCs [7]. All graph states reachable under LCs from a given graph state are called its *orbit* [6]. One strategy for preparing a target graph state would thus be to first find out what graph state in the orbit of the target has the fewest edges, so that this minimal graph can first be reached through CZs, after which the entire remainder of the preparation can be done through less error-prone LCs.

The relevance of accurate and concise graph state preparation gives rise to the question of how to most efficiently compute the graph in a target graph's orbit with the lowest number of edges—a question that remains unanswered. In an effort both to provide an algorithm to find these minimal graphs as well as to gather data on orbit structures, this thesis researches the potential of *binary decision diagrams* (BDDs) for computing and storing the orbits of graph states.

1.1 Orbit computation complexity

Previous research regarding related decision problems has shown that the vertex-minor problem, the problem of deciding whether a graph G' can be reached from a given graph G through local complementations and vertex deletions, is $\mathcal{NP}$-complete [9]. The same was shown by Dahlberg et al. [8] to hold for graph-reachability under local complementations when combined with two other operations on graph states: local Pauli measurements (LPM) and classical communication (CC), i.e., reachability under $\texttt{LC} + \texttt{LPM} + \texttt{CC}$. Lastly, the decision problem of whether a target graph is reachable from another graph through LCs was shown to be in $\mathcal{P}$ by Bouchet [4].

However, the question of what complexity class the problem of finding the minimum-edge graph in a given graph's orbit belongs to has not been answered. The decision problem version of this question, which is what one would have to (dis)prove $\mathcal{NP}$-completeness of, is *Given a target graph* T *and an integer* k*, is there a* $G(E, V) \in \text{orbit}(T)$ *with* $|E| \leq k$*?*

Originally, this thesis' goal was to provide a classification of this decision problem into a complexity class, together with a mathematical proof thereof. However, due to the onerousness of complexity proofs, the lack of proofs for similar problems, and time constraints, we later had to decide to diverge this thesis' research, leaving that problem still open for further research.

1.2 Problem statement

The aim of this thesis is to investigate the effectiveness of using BDDs for finding minimal-edge graph states in an orbit given a source graph. Our first goal is to find out whether this new method is faster and more memory-efficient than performing the calculation by brute force, which is how, for example, *graph state compass* by Adcock et al. implements its orbit exploration [1].

Second, we empirically investigate whether a greedy algorithm provides sufficiently accurate results and how large the difference in computation speed is compared to our BDD approach. Since a greedy algorithm should easily outperform our BDD-based algorithm in terms of speed, it is of interest to know at what accuracy cost this speed benefit comes.

Lastly, we gather as much data as possible on orbits as possible, in order to expand the available data found by Adcock et al. and Cabello et al. ([1] and [6], respectively). Unlike their approaches, ours also regards graphs that are isomorphic to each other as separate graphs in an orbit, making the computation of the minimum graphs in a graph state more intricate and realistic.

In doing so, we aim to establish the usefulness of BDDs in the context of graph state orbit computation by showing their efficiency compared to other approaches.

1.3 Thesis overview

In this paper, we first lay out all the necessary formal definitions of relevant concepts in Section 2. Then, Section 3 elaborates on other related works and their impact on our research's approach, which we detail in Section 4. In Section 5, we present the experiments we ran to benchmark our approach, along with the results and discussion thereof. Lastly, our final conclusions and the insights gained regarding our aforementioned research goals are outlined in Section 6.

2 Background

This section provides readers with an understanding of all the relevant concepts needed for the purpose of this thesis. The remainder of this section assumes familiarity with the fundamentals of graph theory and set theory. For further information on graph theory and set theory, see [18] and [12], respectively.

A central notion in quantum computing is a *quantum state*, which is a mathematical representation of all information about the state of a quantum system at a specific point in time. It encompasses the probabilities of the quantum-mechanical properties in a quantum system, such as a particle's position, momentum, and spin. From this, properties such as superposition or entanglement can be derived, which are essential to quantum algorithms [14]. In this thesis, we focus on a specific category of quantum states: graph states (as defined in Section 2.1).

However, these graph states have the unique property that they can be represented as graphs and therefore, for our purposes, it suffices to reason solely about graphs and not the quantum states they represent, as all relevant operations and computations can also be defined on these graph representations. For a more comprehensive overview of the underlying quantum theory concepts, we refer to [15].

Since graph states and binary decision diagrams are the central concepts for the remainder of this thesis, Sections 2.1 and 2.2 formally define those and closely related concepts.

2.1 Graph theory

Since graph states are quantum systems that can be represented as graphs, this section will outline all the relevant concepts related to graph theory, as well as define the operations on graph states in terms of their graph representations. First, we more formally define the concept that is central to this thesis: a graph state.

Definition 1. A *graph state* is a quantum state in which the entanglement between qubits can be represented as a graph $G = (V, E)$. In this graph, each node $v \in V$ represents a qubit, and the existence of an edge $(u, v) \in E$ indicates that the qubits represented by u and v are *entangled*. Conversely, the absence of an edge $(u, v) \notin E$ indicates that the qubits represented by u and v are not entangled.

It is important to note here that, since entanglement is a mutual property of two qubits, this thesis refers to *undirected* graphs when discussing graph states or graphs in general. Furthermore, we specifically look at undirected graphs without any self-loops (i.e., edges from a node to itself).

A quantum computer can be *prepared* into a given target graph state through various operations. This thesis is centered around two particular operations: the Controlled-Z (CZ) gate and local complementation (LC). First, we define the CZ operation:

Definition 2. Given two nodes $u, v \in V$ in a graph $G = (V, E)$, a *CZ*-gate between the two qubits that these nodes represent is an operation on the graph, $CZ_{(u,\ v)}$, that maps the graph G to a new graph $G' = (V, E')$, where $E' = E \ \Delta \ \{(u,\ v)\}$.

In other words, $CZ_{(u,\ v)}$ flips the existence of a potential edge $(u,\ v)$: if it exists, it is removed from the graph, and if it does not exist, it is added to it. Given just the CZ operation, one could trivially derive a way of preparing any given target graph state from an empty graph. For a target graph state $G = (V, E)$, one could perform $CZ_{(u,\ v)}$ on all pairs $(u, v) \in E$ to reach G after $|E|$ CZ operations. However, the problem with this, due to the fact that a CZ gate is a two-qubit operation, each application of CZ severely increases the complexity of the quantum circuit as well as the likelihood of introducing errors. This likelihood of error is much smaller with local complementations.

If, instead of solely using CZ gates, we could use CZ gates to reach the graph with the fewest edges from which we can reach our target with local complementations, we could thus significantly reduce the complexity of the graph state preparation. Before we can define what local complementation is, we first need to define the important graph-theoretical concept of a neighborhood:

Definition 3. The (open) *neighborhood* $\mathcal{N}_G(v)$ of a node v in a graph $G = (V, E)$ is the subgraph given by the set of all nodes $u \in V$ for which $(u, v) \in E$, and with the subset of E of all edges between these nodes. Formally, $\mathcal{N}_G(v) = (V', E')$, where $V' = \{u \mid (u, v) \in E\}$ and $E' = \{(u, w) \mid (u, w) \in E \wedge u, w \in E'\}$.

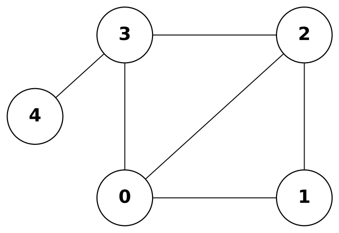
Important to note is that when we refer to neighborhoods in this work, we refer to *open* neighborhoods, which are neighborhoods of a node excluding the node itself (and edges to it). Given this definition of neighborhoods, we can now define the operation this thesis is centered around: local complementation.

Definition 4. Given a node $v \in V$ in a graph $G = (V, E)$, *local complementation* of that node LC_v maps the graph $G = (V, E)$ to a new graph $G' = (V, E')$, where the neighborhood of v is replaced by its complement. Formally, $G' = (V, E')$ where $E' = E \ \Delta \ (V(\mathcal{N}_G(v)) \times V(\mathcal{N}_G(v)))$.

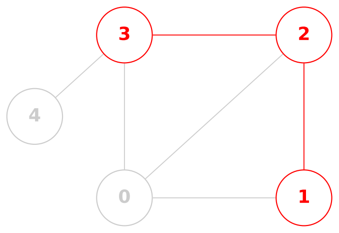
Given this definition of local complementation, example 1 shows an example of how to apply an LC:

Example 1. Suppose a graph $G = (V, E)$ with 5 nodes is given, where $E = \{(0, 1), (0, 2), (0, 3), (1, 2), (2, 3), (3, 4)\}$. To perform local complementation on node 0, i.e., compute $LC_0(G)$, the process (as shown in Figure 1) would look as follows:

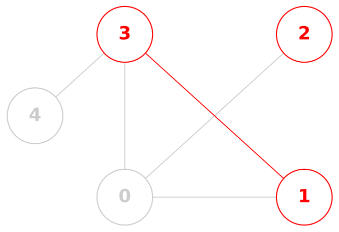
1. First, consider the subgraph given by the neighborhood of 0, which in this example consists of the nodes 1, 2, and 3 and the edges $(1, 2)$ and $(2, 3)$, i.e., $\mathcal{N}_G(0) = (\{1, 2, 3\}, \{(1, 2), (2, 3)\})$.
2. Then, take the complement of this subgraph, which removes the existing edges and adds all the non-existing edges in the subgraph, making its edge set $\{(1, 3)\}$.
3. As a result, the full graph, as shown in Figure 1d, has edges $E' = \{(0, 1), (0, 2), (0, 3), (1, 3), (3, 4)\}$



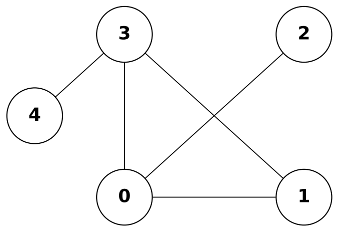
(a) A basic example graph with five nodes.



(b) Outlined in red: the neighborhood of node 0.



(c) Outlined in red: the complement of the neighborhood of node 0.



(d) The result graph: the complement of the neighborhood of node 0 is inserted back into the original graph, and all other edges remain unchanged.

Figure 1: A graphic representation of Example 1, in which we take the local complement of node 0 on a 5-node graph.

This definition then leads to the second core concept of this thesis, that of an orbit:

Definition 5. Given a graph $G = (V, E)$, the *(LC) orbit* of this graph orbit(G) is given by the set of all graphs that are reachable from G through a series of local complementations on its nodes, i.e.:

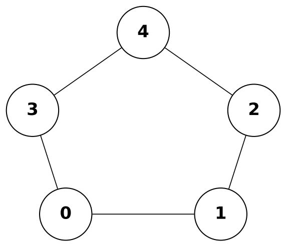
$$\text{orbit}(G) = \{G' \mid G' = (LC_{v_k} \circ LC_{v_{k-1}} \circ \cdots \circ LC_{v_1})(G) \text{ for some } v_1, v_2, \ldots, v_k \in V\}$$

Important to note here is that these local complementations can be applied to the same node multiple times, and that G itself (before any local complementation) is also in orbit(G). Furthermore, there is some terminology we use throughout this thesis regarding graph states and their orbits. When referring to the *size* of a graph $G = (V, E)$, we refer to the number of *edges* ($|E|$) in the graph, and by its *order*, we mean the number of *nodes* ($|V|$) in it. Similarly, with the *size* of an orbit of a graph G, we refer to the number of graphs in the orbit ($|orbit(G)|$), and with the *order* of an orbit, we refer to the number of *nodes* ($|V|$) in the graphs of the orbit.

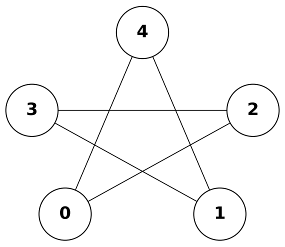
As we will see later, one fundamental difference between our implementation of representing sets of graphs as BDDs and the approaches by other researchers is that we store sets of arbitrary graphs. When graphs are not labeled, however, almost all graphs have numerous graphs that are identical to them, but with their nodes swapped. We call these graphs, which are identical under the exchange of nodes, *isomorphic graphs*. A formal definition is provided in Definition 6, and an example is given in Example 2.

Definition 6. Two graphs G and G' are called *isomorphic* if there exists a one-to-one mapping of their nodes $f : V(G) \to V(G')$, such that $(u, v) \in E(G)$ if and only if $(f(u), f(v)) \in E(G')$.

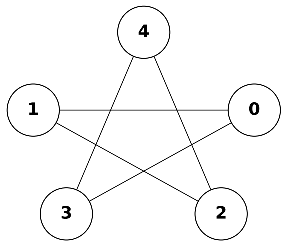
Example 2. An example of two isomorphic graphs would be the two graphs G_a and G_b shown in Figures 2a and 2b. These are isomorphic because of the existence of the mapping $f : V(G_b) \to V(G_a)$ with $f(0) = 3$, $f(1) = 2$, $f(2) = 0$, $f(3) = 1$, and $f(4) = 4$. When mapping the nodes of G_b under this mapping, the result graph (as shown in Figure 2c) has edges $(0, 1)$, $(0, 3)$, $(1, 2)$, $(2, 4)$, and $(3, 4)$, as does G_a.



(a) Graph $G_a = (V_a, E_a)$, isomorphic to the graph in (b)



(b) Graph $G_b = (V_b, E_b)$, isomorphic to the graph in (a)



(c) Another visual representation of the graph in (a), showing more clearly how the nodes of G_b can be mapped to those of G_a.

Figure 2: The graphs that accompany Example 2. In this figure, (a) and (c) are identical graphs with the same nodes and edges but drawn differently. Graph G_b from (b) is isomorphic to that of (a) (and (c)).

Lastly, an important structural property of graphs that is relevant to this thesis, is *connectivity*.

Definition 7. A graph is *connected* if for each pair of nodes $(u, v) \in V \times V$ there exists a path from u to v.

The relevance of this definition for this thesis, is that it is much easier to compute the orbit of unconnected graphs. If in a graph of, for example, ten nodes, only three nodes are connected while there are no other edges between any other nodes in the graph, the computation of this ten-node orbit is as trivial as the computation of the orbit of the three-node subgraph that *is* connected.

2.1.1 Relevant graph theory lemmas

This subsection highlights three lemmas that are important in the computation of the orbit of a graph. We will prove two of these lemmas in this thesis, and a third one will be introduced from an external paper.

Lemma 1. Given a graph $G = (V, E)$ and a node $v \in V$, it holds that $LC_v(LC_v(G)) = G$. In other words, performing local complementation on the same node twice in a row does not affect the graph, regardless of which node it is performed on.

Proof. First, it is important to observe that applying local complementation to a node v only changes the existence of edges between the neighbors of v, but it does not affect any of the edges from v itself to any other node. This means that local complementation on v does not affect which neighbors v has.
Therefore, performing local complementation on the same node twice complements the same subgraph (in terms of nodes) twice. Since complementing a graph twice yields the original graph, it follows that this subgraph is not changed at all, and thus the entire graph remains unchanged. □

Given this lemma, it becomes obvious that performing local complementation on the same node twice in succession does not help in computing a graph's orbit.

The second lemma is relevant to the generation of random graphs to compute the orbit of, which we do in order to run our speed comparison experiments later:

Lemma 2. Given a complete graph $G = (V, E)$ (meaning $E = V \times V$), $|\text{orbit}(G)| = |V| + 1$.

Proof. Given a complete graph $G = (V, E)$, computing the entire orbit of G can be done by iteratively performing LC_k for each $k \in V$ on all graphs found so far, as with the computation of any orbit. Starting with the set of found graphs $\{G\}$, we get:

1. For each $k \in V$, $LC_k(G)$ removes all edges (u, v) for which $u \neq k$ and $v \neq k$, since in a connected graph, k is adjacent to all other nodes in V. This means that the first iteration adds the following set S of $|V|$ so-called "star graphs" to the set of found graphs: $S = \{(V, E') \mid E' = \{(u, v) \mid u = k \wedge v \neq k,\ (u, v) \in V \times V\},\ k \in V\}$.

2. For each graph $G' = LC_k(G)$ in this set of new graphs, there is now one node k with $|V|-1$ edges to each other node, but without any other edges. Performing local complementation on any other node $j \in V \setminus k$ gives $LC_j(G') = G'$, since the only adjacent node adjacent to j is k. However, $LC_k(G') = LC_k(LC_k(G))$, and thus $LC_k(G') = G$. Therefore, we find no new graphs in this iteration.

We now conclude that $\text{orbit}(G) = S \cup \{G\}$, and thus $|\text{orbit}(G)| = |V| + 1$. □

This lemma is useful for our experiments, as we need to generate random graphs to test the efficiency of our approach there. This lemma, together with later empirical evidence, shows that the orbits of complete or "almost complete" graphs are not good representatives of what an average orbit size for a given graph order N is, as these orbits are very small. As one might expect, we found empirically that the opposite also holds: for graphs with very few edges, the orbit is also likely to be very small. Therefore, for a program to generate random graphs with a reasonable orbit size, choosing a balanced number of edges in these graphs is necessary.

The last useful lemma stems from the work of Brand et al. [5, §3.2], in which they derived an upper bound for the number of local complementations needed to reach any graph in the orbit of a given graph, based on Bouchet's algorithm for determining LC-reachability [4].

Lemma 3. If a given graph $G = (V, E)$ can be transformed into a graph $G' = (V, E')$ using local complementations, this takes at most $M = 1\frac{1}{2}(|V| - s)$ local complementations, with $s = |V| \pmod 2$.

As explained in our methodology in Section 4, our algorithm for orbit computation stops when no new graphs are found. However, given this upper bound M in Lemma 3, the last iteration in which no new graphs are found could be prevented. If the first M iterations all yield new graphs, an implementation not taking this upper bound into account would have performed an extra iteration, only to realize there are no new graphs. Instead, when using this upper bound, it tells us that the $M + 1$th iteration can be skipped, decreasing the likelihood of unnecessary local complementations.

2.2 BDD-related definitions

In this thesis, we compute and store orbits in binary decision diagrams. A binary decision diagram (BDD) is a data structure that represents a Boolean function as follows:

Definition 8. Given an n-variable Boolean function $f : \{0,1\}^n \to \{0,1\}$, a *binary decision diagram* for f is a directed acyclic graph consisting of nodes that represent variables, with outgoing 'high' and 'low' edges denoting true and false assignments to these variables, respectively. Leaf nodes in the BDD represent the output of the Boolean function, i.e., either true or false (or 1 and 0). A walk through the BDD corresponds to a unique variable assignment $(x_1, x_2, \ldots, x_n) \in \{0,1\}^n$, leading to a true-valued leaf node if and only if $f(x_1, x_2, \ldots, x_n) = 1$.

When writing a program to compute orbits of graph states, this program has to store sets of graphs, such as the orbits themselves. Essentially, these graph sets are subsets of the set of all potential graphs of a given order $|V|$. Binary decision diagrams lend themselves particularly well to storing these graph sets, as they can be used to represent *indicator functions* of this graph set.

Definition 9. An *indicator function* (also *characteristic function*) $1_A : X \to \{0,1\}$ for a subset $A \subseteq X$ yields 1 if and only if the input is in the subset, and 0 otherwise. In other words:

$$1_A(x) = \begin{cases} 1 & \text{if } x \in A \\ 0 & \text{if } x \notin A \end{cases}$$

If we want to represent the indicator function for a set of graphs, however, we need to find a way to define the graphs as a combination of binary variables. There are $N(N-1)/2$ potential edges in a graph for $N = |V|$ because each of the N nodes has $N - 1$ potential neighbors, yielding $N(N-1)$ combinations, but, since we are looking at undirected graphs, there are only $N(N-1)/2$ potential edges. If we take $N(N-1)/2$ binary variables $x_1, x_2, \ldots, x_{N(N-1)/2} \in \{0,1\}$ to represent the existence of a potential edge, we can now generate each possible graph as a combination of true or false assignments to these variables. With this, we can now store sets of graphs in binary decision diagrams. An example of a set of graphs stored in a BDD, as well as how this BDD can be pruned to store it more efficiently, is given in Example 3.

Example 3. Suppose a set of graphs S is given, where S contains all graphs of order 3 except for two graphs G_a and G_b. Formally, this means $S = G_3 \setminus \{G_a, G_b\}$, where G_3 is the set of all graphs of order 3, i.e., $G_3 = \{(V, E) \mid V = (0,1,2),\ E \subseteq V \times V\}$. Suppose $G_a,\ G_b \in G_3$ are two graphs with nodes $(0,1,2)$ and edges $E_a = \{(0,1),(1,2)\}$ and $E_b = \{(1,2)\}$, respectively.

To encode this set S as a BDD, we first introduce binary variables $x_0, x_1, x_2 \in \{0,1\}$, indicating the presence of edges $(0,1)$, $(0,2)$, and $(1,2)$, respectively, in a potential graph. Using this representation, we define an indicator function 1_S for S, the truth table of which is given in Table 1. This truth table aids in the construction of the BDD, as shown in Figure 3.

x_0	x_1	x_2	$1_S(x_0, x_1, x_2)$
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

Table 1: A truth table for the indicator function 1_S corresponding to the subset $S \subset G_3$ from Example 3.

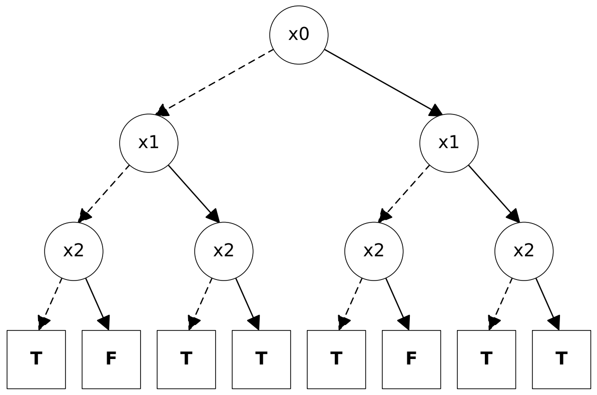
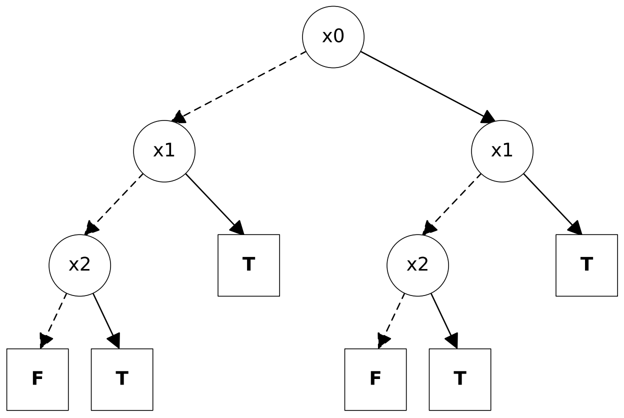
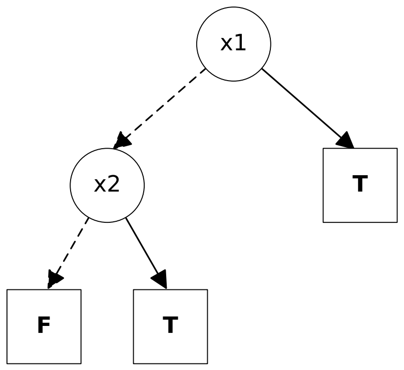


Figure 3: A BDD corresponding to the truth table in Table 1, as another representation of the set of graphs S from Example 3. In this BDD, dashed lines represent low edges, while solid lines represent high edges.

This BDD is also a good example of how sets can often be represented more efficiently using BDDs, due to their ability to be reduced, in two ways [2]. First of all, nodes that have two equal-valued Boolean leaf nodes as children can be replaced by a leaf node with this Boolean value, eliminating two nodes. In this BDD, this can be applied twice, at two of the occurrences of x_2. Second, if the edges of one node lead to two isomorphic subgraphs, this node can be eliminated as well by replacing it with either of the isomorphic subgraphs. In this example, we can apply this second rule to the root node, which, together with the other pruning of the BDD, is displayed in Figure 4.



(a) The result of replacing two occurences of node x_2 for which both of its edges lead to true leaf nodes, with a true-valued leaf node itself.



(b) The result of merging the two isomorphic subgraphs of the root node of the BDD from (a).

Figure 4: An example of how the BDD from Figure 3 can be pruned into a minimal BDD.

3 Related Work

This section outlines the existing literature concerning graph state orbit exploration, as well as on the `C++` BDD library we use, `sylvan`, and on the encoding of graph state operations through logical variables.

Earlier research on the generation of datasets of LC orbits of graph states was performed by both Cabello et al. [6] and Adcock et al. [1], and they were both able to compute all orbits with orders up to twelve. What separates the two articles is that Adcock et al. expanded on the work of Cabello et al. by not only computing the orbits but also exploring the structure of the orbits themselves. Besides solely investigating the properties of graphs, Adcock et al. also looked at the structure of the transition between graphs in an orbit, and the properties they could obtain thereof. They developed an open-source Python tool called `graph state compass` to enable this orbit exploration.

As is mentioned in Section 4.4, however, both of these papers limit their orbit exploration by merging all isomorphic graphs, making their computation significantly easier. For this, they use a brute-force approach like the one we developed (found in Section 4.1), but instead, our main contribution is a BDD-based approach to storing and computing orbits, making use of the `sylvan` library.

In 2016, the `sylvan` C++ library was released by Van Dijk [17], offering a multi-core parallel programming solution for the storage of and computation on BDDs. They offer high performance not only through this parallelization, but also through caching the results of calculations and by automatically minimizing the BBDs that are in memory using the pruning strategies described in Section 2.2.

Furthermore, the `sylvan` library enables users to extend its functionality, by allowing the development of custom BDD operations that have access to all internal features, including parallelization and caching. They have also implemented transition relationships on BDDs (through their `RelNext` method), making it possible to generate mappings of the set of all elements that the BDD represents to a new set (BDD), while expressing this transition in logical expressions with respect to the BDD's variables. We use this feature to define our LC-relation on entire graph sets at once, instead of on separate graphs.

Brand et al. developed SAT-encodings for graph states and the operations thereon, including local complementations [5]. We can use this encoding of LCs to program logical expressions on our BDD variables (i.e., the potential edges in the graph) for the aforementioned transition relationship for LC. The encoding from Brand et al. to encode the local complementation of a node k can be found in Equation 1. Here, x'_{uv} (for $u < v$ and $u, v \in V$) is a Boolean variable indicating whether the edge (u, v) exists in the graph after performing local complementation, and x_{uv} indicates the same but for the original graph.

$$LC_k = \bigwedge_{\substack{(u,v)\in V\times V\\ u<v}} \begin{cases} x'_{uv} \leftrightarrow \neg((x_{uk} \wedge x_{vk}) \oplus \neg x_{uv}) & \text{if } u \neq k \text{ and } v \neq k \\ x'_{uv} \leftrightarrow x_{uv} & \text{else} \end{cases} \tag{1}$$

4 Algorithm Development and C++ Implementation

For this thesis, we first developed a codebase with three programs for finding minimum graphs in an orbit, after which we ran experiments to determine their effectivities, as described in Section 5. The three programs are the following: one using BDDs, one using a brute-force approach, and one using a greedy algorithm. All three of these programs were written in C++, as it allows for high control over memory usage and (partially therefore) its potential for performing high-performance computation [19]. This section will detail the pseudocode for the programs, but the full code of these programs, as well as of our experiments, can be found in our GitHub repository at [16].

4.1 Brute force implementation

In order to have a baseline to compare our experiments' results to, we first wrote a brute-force approach to the problem (the `C++` program of which is later referred to as `brute_force`). This approach works by storing an `std::set` of "found" graphs and continuously looping over this set, while computing local complementation for every node on every graph in it, until no new graphs are found or the maximum computation depth (from Lemma 3) is reached. The pseudocode for this algorithm is shown in Algorithm 1.

Algorithm 1 Our brute-force algorithm for `ComputeOrbit`, used to compute the orbit of a graph.

Input: *InitialGraph* and *maxDepth*
Output: *Graphs* (a set of all graphs in the orbit of the input graph)

```
1: function ComputeOrbit(InitialGraph, maxDepth)
2:     Graphs, PrevAddedGraphs ← {InitialGraph}
3:     for depth ← 1 to maxDepth do
4:         newGraphs ← empty list
5:         for G in PrevAddedGraphs do
6:             for v in $V(G)$ do
7:                 if $LC_v(G)$ not in Graphs then
8:                     add $LC_v(G)$ to newGraphs

9:         if size of newGraphs = 0 then
10:            break
11:        add every element in newGraphs to Graphs
12:        PrevAddedGraphs ← newGraphs
13:    return Graphs
```

4.2 BDD sylvan implementation

The second program we wrote, `gs_bdd`, uses BDDs instead of `std::sets`, because of the potential benefits offered by a BDD's ability to define transition relations on a set as a whole, instead of having to perform computations on individual elements of the sets. In our brute-force algorithm, for example, we have to compute the local complementation for every node of every graph in the set of found graphs, compared to having to perform the computation only on the BDD itself when storing the sets as BDDs.

For the `C++` implementation of the BDDs, we used `sylvan` to implement storing sets of graphs as BDDs and to compute the outcomes of transition relations on them, and for the storage of the sets of graphs in BDDs, we used the method described in Example 3. We developed a `GSExplorer` class that allows users to store graphs in BDDs, compute the orbits of BDDs, find the smallest graph in a graph set, and find all the orbits of all graphs of a given size. In this class, we implemented local complementations as transition relations, where the logical expressions were implemented as per the approach of Brand et al. [5], as discussed in Section 3. Using this `GetLCRelation(`k`)` function, we implemented a `ComputeOrbit` function, shown in Algorithm 2:

Algorithm 2 Our BDD approach to `ComputeOrbit`, used to compute the orbit of a graph.

Input: *InitialGraph* (a BDD representing only one input graph)
Output: *Graphs* (a BDD with all graphs in the orbit of the input graph)

```
1:  function ComputeOrbit($InitialGraph$)
2:      $Graphs, PrevAddedGraphs \leftarrow InitialGraph$
3:      while $PrevAddedGraphs \neq$ false BDD do
4:          $NewGraphs \leftarrow$ false BDD
5:          for $k \leftarrow 1$ to $N$ do
6:              $NewGraphs \leftarrow NewGraphs \cup$ (GetLCRelation($k$) applied to $PrevAddedGraphs$)
7:          $PrevAddedGraphs \leftarrow Graphs \setminus NewGraphs$
8:          $Graphs \leftarrow Graphs \cup NewGraphs$
9:
10:     return $Graphs$
```

Then, we use this `ComputeOrbit` procedure in `FindAllOrbits`, which is the function we wrote for computing all the orbits of all graphs of a given size N. The pseudocode for this is shown below in Algorithm 3. It works by continuously picking a graph that has not yet been found, computing its orbit, and adding this orbit to the BDD of found graphs, until all graphs of order N have been found.

Algorithm 3 The function `FindAllOrbits`, used to find all orbits of order N.

Input: *InitialGraph* and *maxDepth*

```
1:  function FindAllOrbits($N$)
2:      $G \leftarrow$ empty graph
3:      $DiscoveredGraphs \leftarrow$ false BDD
4:      while true do
5:          $NewOrbit \leftarrow$ ComputeOrbit($G$)
6:          $DiscoveredGraphs \leftarrow AllDiscoveredGraphs \cup NewGraphs$
7:          if $AllDiscoveredGraphs =$ true BDD then
8:              break
9:          $UndiscoveredGraphs \leftarrow$ negation of $DiscoveredGraphs$
10:         $G \leftarrow$ pick one graph from $UndiscoveredGraphs$
```

In practice, one would usually want to save data about the new orbit that was just found between lines 5 and 6 in Algorithm 3. In our implementation, we stored a tuple containing the orbit size, the size of the orbit's smallest graph, and a representative graph in this orbit that has this minimum size. This procedure and the accompanying data collection were implemented for the dataset generation experiment, which is described in Section 5.1.

Important to note here is that this method computes the orbits of *all* graphs of a given order N, including the orbits of *unconnected* graphs. This is unlike the computer programs for orbit computation by Adcock et al. and Cabello et al., in which they only included connected graphs ([6] and [1]). The reason we chose to consider unconnected graphs is that we prioritized completeness in the generation of the dataset of all orbits, in the experiment described in Section 5.3.

Lastly, an important function is `FindSmallestSAT`, which finds the smallest graph in an orbit. We defined this function in `C` as an extension of the `sylvan` library, and it computes the smallest set of variables needed to satisfy the BDD. This is done recursively with a top-down approach, making use of `sylvan`'s caching capabilities, as shown in the pseudocode in Algorithm 4.

Algorithm 4 The function `FindSmallestSAT`, which computes the smallest set of variables needed to satisfy a given BDD.

Input: *BDD*

Output: *result* and *varAssignment* (an example variable assignment array represented by a 0 or 1 entry for each variable in *BDD*).

```
 1: function FindSmallestSAT(BDD, &varAssignment)
 2:     if BDD = true BDD then
 3:         return 0
 4:     if BDD = false BDD then
 5:         return Infinity

 6:     if $|Vars(BDD)| \leq 64$ then
 7:         if BDD is in sylvan cache then
 8:             varAssignment ← decoded result from sylvan cache
 9:             return number of true values in varAssignment

10:     varAssignment_low ← copy of varAssignment
11:     count_low ← FindSmallestSAT(BDD.low, varAssignment_low), computed in a parallel process
12:     result ← FindSmallestSAT(BDD.high, varAssignment)
13:     wait for parallel execution to finish

14:     if count_low < count_high then
15:         varAssignment ← varAssignment_low
16:         result ← count_low
17:     if $|Vars(BDD)| \leq 64$ then
18:         put varAssignment encoded as a 64-bit bitstring in sylvan cache
19:     return result
```

Because `sylvan` requires cache entries to be of type `uint64_t`, we store our `varAssignment` results as bitstrings, where each bit in the cache entry's `uint64_t` represents one variable in the variable assignment (where the rightmost bit is `varAssignment[0]`). This is why the algorithm encodes and decodes the varAssignment before storing it in the cache and retrieving it from the cache, respectively, as well as why it only stores and retrieves cache entries of BDDs with at most 64 variables.

4.3 Greedy implementation

The last program we wrote was a greedy algorithm, used to assess the efficiency and, more importantly, the accuracy of a heuristic approach for finding the minimum graph in an orbit. The greedy algorithm we wrote works by continuously storing only the smallest graph found thus far. In each iteration, the algorithm performs local complementation on each node of this graph and compares the size of the smallest graph found this iteration with the size of the smallest graph from the previous iteration. If a smaller graph was found, the process is repeated, and if not, the smallest graph that was found is returned.

This method may be very fast, as it takes no more than $1\frac{1}{2}|V|(|V|+1)$ local complementations, because the algorithm performs at most $M+1$ local complementations on $|V|$ nodes, given the upper bound in the local complementations M from Lemma 3. However, it is also very unlikely to find the smallest graph in the orbit. This is due to the fact that sometimes it is necessary to first perform a local complementation that does *not* yield any smaller graphs, because it enables later local complementations that *do* lead to smaller graphs. This is why we also allowed a separate version of this algorithm to "look ahead" one layer (later referred to as greedy with *depth* 2). In this version, the algorithm not only considers the graphs reachable from the current-found smallest graph after *one* LC but also considers the smallest reachable graph after *two* LCs, in an effort to make the algorithm more versatile.

Because of its expected high efficiency, it is of particular interest to analyze the difference in the found 'minimum' graph size compared to the true minimum graph in the orbit, as well as the difference in efficiency compared to the BDD algorithm, across different graph orders.

4.4 Our approach to isomorphic graphs

Compared to the work of Adcock et al. [1] and Cabello et al. [6], one fundamental difference between their approach and ours is that we did not merge isomorphic graphs but considered all isomorphic graphs as separate graphs. Table 2 shows the impact this has on the number of graphs that need to be computed and stored. From this, it becomes obvious that the separation of isomorphic graphs vastly increases the computational complexity of storing all graphs of a certain order.

As shown in Sections 5.1 and 5.3, respectively, the largest orbit we were currently able to compute (within a reasonable time) had 10 nodes, and the largest graph order we were able to compute all orbits of was 8. If we had been able to limit our orbit exploration to only non-isomorphic graphs, we likely would have been able to explore all orbits of graphs of up to 10 nodes, as this would have required us to store only 12,005,168 graphs and 1,274,068 orbits [10]. This is less than the number of graphs and orbits we had to store for all 8-node graphs in our current implementation (as shown in Section 5.3).

	N=1	**N=2**	**N=3**	**N=4**	**N=5**	**N=6**	**N=7**	**N=8**
#Non-iso. graphs	1	2	4	11	34	156	1,044	12,346
#Graphs	1	2	8	64	1024	32,768	2,097,152	268,435,456

Table 2: An overview of the number of potential graphs for sizes one through eight. Here, **#Graphs** refers to the total number of arbitrary graphs, whereas **#Non-iso. graphs** refers to the number of graphs after merging isomorphic graphs. Source: [3].

On the other hand, considering the fact that we are trying to generate a dataset of all graphs we are trying to generate in this thesis, it can be argued that looking at arbitrary graphs in an orbit without merging isomorphic graphs is not necessarily a disadvantage for the purpose of this project. This difference in approach necessitates future research to investigate whether our BDD approach can be extended to allow for the merging of isomorphic graphs, to also enable comparison between our approach and others in this scenario.

5 Experiments

To answer our research questions and to establish whether using BDDs is efficient and practically useful, we ran three experiments. In our first two experiments, we compared the speed of our BDD implementation to the brute-force and greedy algorithms in the experiments described in Sections 5.1 and 5.2. In our second experiment, we tried to generate a dataset of the orbits all graphs we could compute, as detailed in Section 5.3.

Both experiments were run on a 2020 M1 Macbook Pro running macOS Sonoma version 14.4.1. All of the code was compiled using `CMake` version 3.14 with the `g++-13` (Homebrew GCC 13.2.0) compiler.

5.1 BDD vs. brute force for minimum graph computation

A practically relevant task is the computation of the smallest graph in the orbit of a given target graph. Therefore, in our first experiment, we wanted to compare the efficiency of our BDD implementation and the brute-force implementation.

The way we compared the efficiencies is twofold. First, we compared the speed of the brute-force and BDD methods for different graph orders, to establish whether either program performed better at smaller or larger numbers of nodes. The other part of this experiment consisted of comparing the brute-force and BDD approaches when computing orbits of different graphs of the *same* order, to determine whether either approach is favorable for the computation of either larger or smaller orbits.

For the first part of this experiment, we generated ten random graphs using the method described below, for graph orders $|V| = 3$ through $|V| = 10$. We chose this range because graphs with under three nodes always have trivial orbits of size 1, and graphs with over ten nodes were too complicated to compute the orbit of for either program. For each of these generated graphs, we computed its orbit and found the smallest graph in it using both programs, while measuring the computation speed and memory usage of both implementations.

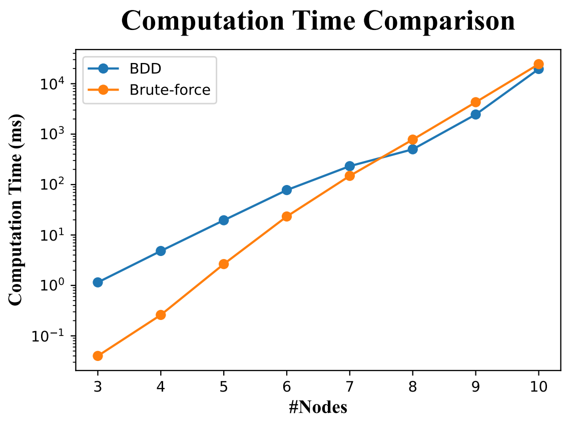
For the second part of this experiment, we generated one hundred random graphs of fixed graph order $|V| = 8$ and computed their orbits using both programs, measuring the time and counting the size of the orbit that we generated. The reason why we opted for $|V| = 8$ as a graph order is that this is the largest order for which we were able to compute all orbits (using the BDD program), and all lower orders had a smaller variety in orbit sizes. Through this part of the experiment, we aimed to determine whether one of the two programs outperforms the other when computing, for example, larger or smaller orbits.

Random graph generation
For reproducibility, we wrote a `Python` program that we chose to employ a standard method for random graph generation. It uses the Erdős-Rényi model [11] with parameter $p = 0.6$, which indicates the probability of an edge being included in a generated graph. As we will see in Section 5.3, the variety in orbit sizes is very large, even when comparing graphs of the same order, making the choice of p a delicate one. The reason why we chose $p = 0.6$ is twofold: firstly, it is important to keep p fixed for reproducibility in future research, and secondly, p needs to be neither too low nor too high.

If p is too high and the graph has too many edges and is almost complete, the orbit has very few graphs in it (see Lemma 2). If the graph has too few edges, the probability of the graph being connected is very low, making it hard to compare to other graphs of that order. Even if a graph with few edges is connected, each node has a small neighborhood, and thus the effect of local complementation is likely small, making the orbit likely very small too. We empirically found that using $p = 0.6$ gives graphs with reasonable orbit sizes, which are usually a suitable average representation for other graphs with that graph order.

Lastly, after this graph is generated, it is checked to see whether it is connected or not, using the `networkx Python` package's `is_connected` method. If it is not, our graph generation program discards this graph and generates a new one.

The results of the first part of this experiment are shown graphically in Figure 5, and a table with all the numerical results corresponding to these graphs can be found in Appendix A.



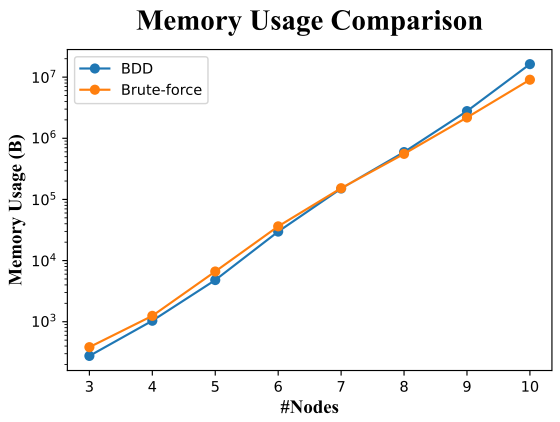


Figure 5: Average computation time in milliseconds and memory usage in bytes for running `ComputeOrbit` on 10 randomly generated graphs for $|V|$ ranging from 3 through 10, for both the `gs_bdd` and `brute_force` programs.

These results show that for graphs of lower order, a simple brute-force approach is significantly faster than our BDD program. This is likely because for graphs with fewer nodes, the performance drawback caused by the overhead of storing the graph sets in BDDs outweighs the benefit of being able to define transition relations on sets as a whole for computing LCs. However, for graphs of higher order, there is a substantial benefit to using BDDs for orbit computation. We observe that using BDDs is consistently more than 20 percent faster than brute-forcing the calculation. This can be explained by the fact that each iteration of local complementations is likely to increase the size of the set of newly found graphs, in particular for graphs of higher order, and thus the speed benefits of using transition relations on entire sets simultaneously become larger too.

Regarding the memory measurements from the first experiment, we observe that the BDDs have a higher memory consumption than the regular `std::sets` of graphs for graphs of higher order. This is likely because for regular graph storage, the program only needs to store the edges that *are* in the graph, whereas in a BDD, this is not necessarily the case. The memory consumption of BDDs does not necessarily grow linearly with the number of graphs or edges in those orbits, but is moreso determined by the complexity of the graph set being stored.

The results of the second part of the experiment, in which we compare the speed of the two programs for various graphs of the *same* order, are shown in Figure 6.

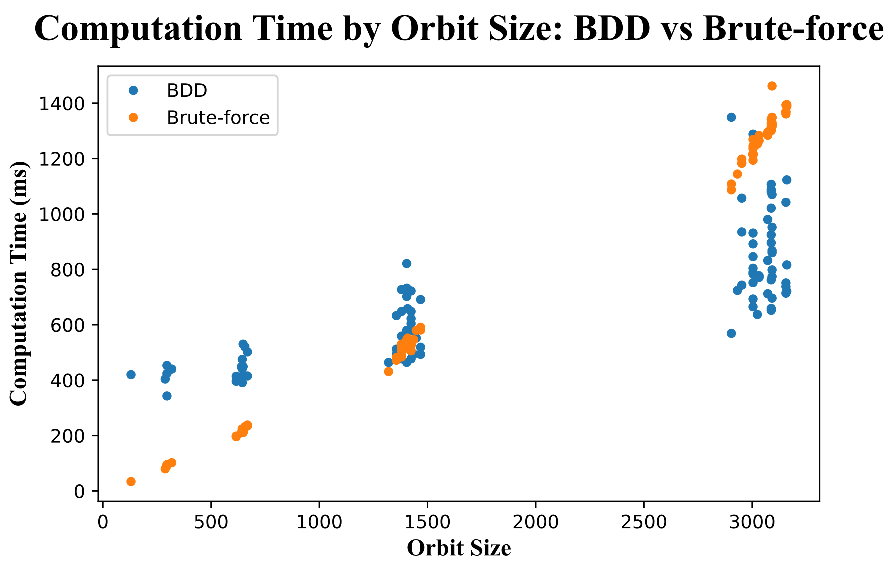


Figure 6: The computation times of the calculation of the orbits of 100 random graphs (with $|V| = 8$) by the BDD program and the brute-force program, plotted against the orbit sizes.

In these results, we observe a very steep, seemingly exponential, increase in the computation time as the orbit size grows when calculating the orbit using `brute_force`, whereas this steep growth is not present with `gs_bdd`. This shows that for graphs with smaller orbits, a brute-force approach is faster than a BDD approach, and vice versa. While the difficulty of predicting the orbit size beforehand might make this information seemingly unhelpful in determining which program to use, there are still two important considerations to derive from these results. First, we established in Section 2.1.1 that very small or large graphs are expected to have small orbits, so in these cases, the brute-force program would be recommended over the BDD program. Second, this performance benefit for larger orbits is yet another supporting fact that shows BDDs' efficiency when computing orbits of graphs of higher order, as these tend to have larger orbits.

5.2 BDD vs. greedy for minimum graph computation

Similar to the way we compared our BDD approach to the brute-force program, we also compared our BDD program to the greedy algorithm (as specified in Section 4.3). Here, we expect the greedy algorithm to be magnitudes faster than the BDD algorithm due to the simplicity of its approach. However, this same simplicity causes the greedy algorithm to fail to find the actual smallest graph in an orbit. Therefore, the more interesting performance metric for `greedy` is the difference between the greedy algorithm's found smallest graph size and the actual smallest graph size.

As a result, we decided for this experiment to again generate 10 graphs for graph orders 3 through 10, and make the BDD and greedy program (with a depth of 1 and a depth of 2) compute the smallest graph in their orbits. We then measure both the computation time and the size of the smallest graph they found, to assess the trade-off between the greedy algorithm's accuracy decrease and speed increase. The results of these measurements are shown in Table 3.

$\|V\|$	**Avg.** $\|E\|$	**BDD**		**Greedy** $(d = 1)$		**Greedy** $(d = 2)$	
		Time	**Graph size**	**Time**	**Graph size**	**Time**	**Graph size**
3	2.4	1.14 ms	2.0	0.01 ms	2.0 (+0.0%)	0.02 ms	2.0 (+0.0%)
4	4.1	4.04 ms	3.0	0.03 ms	3.1 (+3.3%)	0.07 ms	3.1 (+3.3%)
5	6.1	18.4 ms	4.4	0.07 ms	4.6 (+4.5%)	0.21 ms	4.6 (+4.5%)
6	9.0	56.5 ms	5.4	0.15 ms	5.9 (+9.3%)	0.57 ms	5.9 (+9.3%)
7	13.4	266.0 ms	7.3	0.31 ms	9.3 (+27.4%)	1.47 ms	8.3 (+13.7%)
8	17.3	666.0 ms	9.5	0.61 ms	12.5 (+31.6%)	2.82 ms	10.6 (+11.6%)
9	23.2	3947.0 ms	11.4	1.17 ms	15.0 (+31.6%)	5.15 ms	12.7 (+11.4%)
10	28.6	24 297.0 ms	13.9	1.93 ms	18.7 (+34.5%)	10.2 ms	15.9 (+14.4%)

Table 3: This table shows the results of the second experiment. For 10 graphs of orders 3 through 10, we show the average graph size along with the average orbit computation time and the average size of the reduced graph obtained by both the BDD program and the greedy algorithm (with depths 1 and 2). The percentual differences between the greedy algorithms' smallest graph sizes and the optimal solution are shown in parentheses.

Comparing the BDD and greedy programs using this data, the results are as expected: the greedy algorithm is magnitudes faster (even at depth 2) compared to the BDD algorithm, but at the cost of lower accuracy in terms of the smallest graph found. However, the lost accuracy of the greedy approach is relatively small, particularly with the greedy algorithm at depth 2. However, the disparity between the greedy algorithm's result and the true minimum graph size grows larger as the graph's order increases. Therefore, for graphs with more nodes, it appears that the `greedy` program becomes less appropriate to use than, for example, `gs_bdd` or `brute_force`.

5.3 Orbit dataset generation

As one of the goals of this thesis is to form a dataset of as many orbits as possible, in the third experiment, we attempt to compute all orbits of graphs with as many nodes as possible. We again measure the time and memory needed to perform this computation. If, after 24 hours, a computation has not finished yet, we terminate it as it is not likely to succeed within a reasonable time.

From the data garnered from this experiment, we collected the following statistics for each graph order: the number of orbits, the distribution of the orbit sizes, and the distribution of the sizes of the minimum graph in each orbit, i.e.: *how many graphs with N nodes can be reduced to having only k edges?* For this last question, results become more relevant if we filter out disconnected graphs, again by using `networkx`'s `is_connected` method. If we had not excluded disconnected graphs such as, for example, the $8 \cdot (8-1)/2 = 28$ eight-node graphs with only one edge, our data would have suggested that large graphs are much more reducible than they truly are.

For this experiment, we were able to compute all 307,480 orbits of all 270,566,475 graphs of one through eight nodes. The entire dataset of orbits can be found in our GitHub repository at [16]. Table 4 provides an overview of the number of orbits found and the time and memory consumed during this exploration of all orbits of a given order.

Graph order	**#Orbits**	**Time (s)**	**Mem (bytes)**
1	1	0.0002	16
2	2	0.0007	48
3	5	0.006	144
4	18	0.047	720
5	93	0.45	7,888
6	760	8.1	126,336
7	10,773	665	6,481,776
8	295,828	±70,500	408,515,312

Table 4: The results of executing `FindAllOrbits(`N`)` for $N = 1, 2, \ldots, 8$, detailing the number of orbits found, the computation time, and the peak memory consumption for the storage of the orbits' BDDs. The measurement of the computation time of all eight-node orbits is approximated, due to the program needing to be paused frequently to prevent throttling.

It is important to note that the computation of all seven- and eight-node orbits, spread out over multiple cores, was a computationally very intensive task. Particularly with the computation of all eight-node orbits, it was hard to gather concrete results regarding the computation time, as the computation was slowed down significantly due to throttling. Therefore, we decided to implement the possibility to automatically pause and resume the computation (and time measurement) at a given time interval, in order to prevent overheating and get a more accurate result for the computation time.

Our BDD implementation was not able to compute the orbits of all graphs of nine or more nodes within a reasonable time. Even though our first experiment showed that the program is capable of computing orbits of graphs of order nine or ten as well, the computation of *all* graphs of this size would likely take months and require more memory than is available in most contemporary PCs.

Lastly, the graphical representations of the distribution of the graph sizes of the minimum graphs in the orbits, as well as the distribution of orbit sizes, are shown in Figures 7 and 8. The full numerical results corresponding to these graphs are shown in Appendix A.

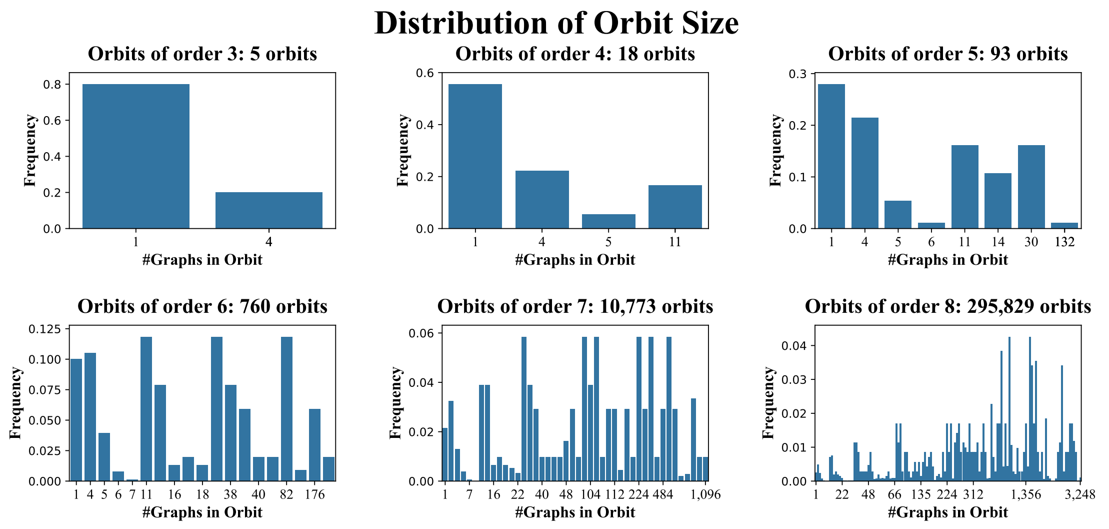


Figure 7: Six histograms showing the frequency (relative to the total number of orbits) of the sizes of the orbits, for all orbits of graph sizes $|V| = 3$ through $|V| = 8$.

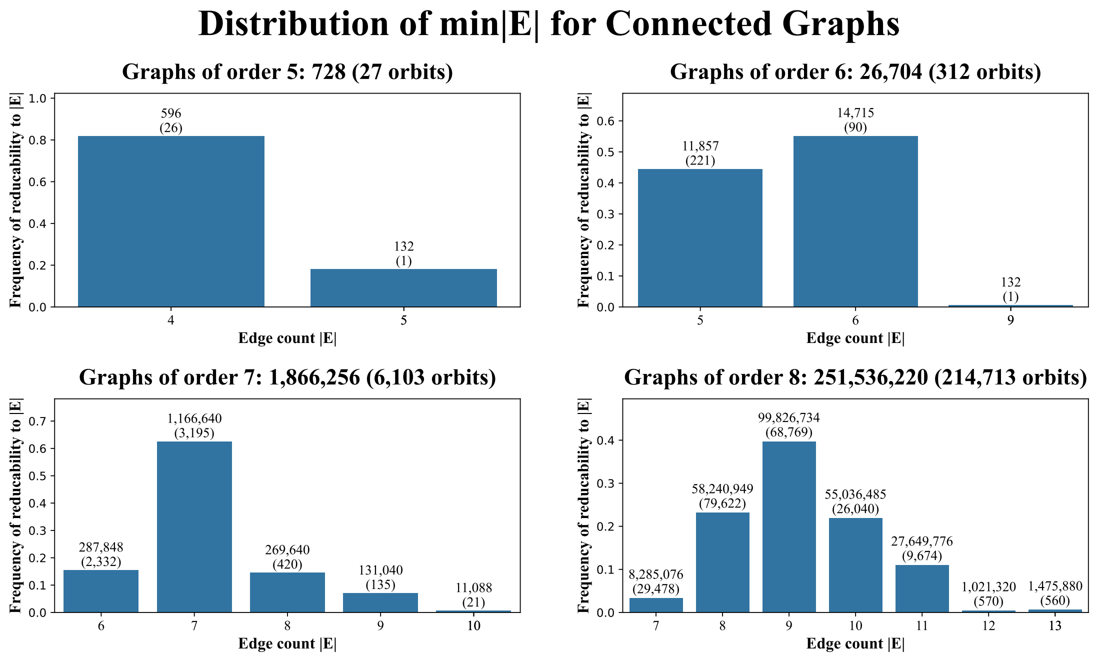


Figure 8: Four histograms showing the frequency of the minimum size a graph is reducible to, for all graphs of order 5 through 8. The bars are annotated with the number of graphs and the number of orbits (in parentheses).

6 Conclusions and Further Research

This thesis explored various methods of computing the minimum graph in an orbit of a given target graph—an important calculation when performing graph state preparation. In particular, we proposed a new method for this computation using binary decision digrams. Our experiments have shown that using BDDs is an efficient manner of computing minimum graphs in an orbit, particularly when computing larger orbits or orbits of graphs of a high order ($|V| \geq 8$).

For graphs of low order, we found a brute-force approach to be faster while also consuming less memory. Furthermore, a greedy approach was found to be multitudes faster than either of the other algorithms, but with the trade-off of significantly decreasing the computation's accuracy. Using the greedy algorithm for an approximation of the smallest graph in an orbit can perform within a reasonable margin, particularly when the graph order is low ($|V| \leq 8$) and the algorithm is allowed to look one LC-layer further ahead (i.e., depth = 2).

Our BDD-based approach considered all isomorphic graphs separately, prompting further research to explore the benefits of merging isomorphic graphs while using BDDs. Although the implementation of this appears complex and has not been studied in this thesis, this would allow for a better comparison to other research on orbit computation and pave the way to further solidify the potential of using BDDs for graph state exploration.

Since this thesis was not able to find a proof for a classification of the complexity of finding the smallest graph state in an LC orbit (as discussed in Section 1.1), further research is needed to (dis)prove the $\mathcal{NP}$-hardness of the problem. If it is shown to be in $\mathcal{P}$, a polynomial algorithm could then be used to compute the smallest graph state in an orbit. Until that time, in situations where isomorphic graphs are not excluded, BDDs have now been shown to be an efficient manner of computing a target graph's local complementation orbit during graph state preparation.

References

[1] Jeremy C Adcock, Sam Morley-Short, Axel Dahlberg, and Joshua W Silverstone. Mapping graph state orbits under local complementation. *Quantum*, 4:305, 2020.

[2] Anna Bernasconi, Valentina Ciriani, Fabrizio Luccio, and Linda Pagli. Exploiting regularities for Boolean function synthesis. *Theory of Computing Systems*, 39:485–501, 2006.

[3] Nicolas Borie. Generating tuples of integers modulo the action of a permutation group and applications. *Discrete Mathematics and Theoretical Computer Science*, 11 2012.

[4] André Bouchet. An efficient algorithm to recognize locally equivalent graphs. *Combinatorica*, 11:315–329, 1991.

[5] S.O. Brand, T. Coopmans, and A.W. Laarman. Quantum graph-state synthesis with SAT. *arXiv preprint*, 2023.

[6] Adán Cabello, Lars Eirik Danielsen, Antonio J López-Tarrida, and José R Portillo. Optimal preparation of graph states. *Physical Review A*, 83(4):042314, 2011.

[7] A. Dahlberg and Wehner S. Transforming graph states using single-qubit operations. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 376, 2018.

[8] Axel Dahlberg, Jonas Helsen, and Stephanie Wehner. How to transform graph states using single-qubit operations: computational complexity and algorithms. *Quantum Science and Technology*, 5(4):045016, 2020.

[9] Axel Dahlberg, Jonas Helsen, and Stephanie Wehner. The complexity of the vertex-minor problem. *Information Processing Letters*, 175:106222, 2022.

[10] Lars Eirik Danielsen and Matthew G Parker. On the classification of all self-dual additive codes over GF(4) of length up to 12. *Journal of Combinatorial Theory, Series A*, 113(7):1351–1367, 2006.

[11] P. Erdős and A. Rényi. On random graphs. I. *Publicationes mathematicae*, 6(3-4):290–297, 7 1959.

[12] Felix Hausdorff. *Set theory*, volume 119. American Mathematical Soc., 2021.

[13] M. Hein, J. Eisert, and H. J. Briegel. Multiparty entanglement in graph states. *Physical Review A*, 69, 2004.

[14] Michael A Nielsen and Isaac L Chuang. *Quantum computation and quantum information*. Cambridge University Press, 2001.

[15] Andrew Steane. Quantum computing. *Reports on Progress in Physics*, 61(2):117, 1998.

[16] Stan van Baarsen. GitHub: StanvBaarsen/graph-state-bdds. https://github.com/stanvbaarsen/graph-state-bdds. GitHub repository for the C++ code and orbit datasets accompanying this thesis.

[17] Tom van Dijk. Sylvan: Multi-core decision diagrams. *Tools and Algorithms for the Construction and Analysis of Systems*, 2016.

[18] D.B. West. *Introduction to graph theory.* Featured Titles for Graph Theory. Prentice Hall, 2001.

[19] Farzeen Zehra, Maha Javed, Darakhshan Khan, and Maria Pasha. Comparative analysis of C++ and Python in terms of memory and time. 2020.

A Full results of experiments

The below table shows the full (non-logarithmically scaled) numerical results of the first part of the first experiment, in which we compare the computation time and memory consumption of our BDD and brute-force programs. This is the data that is graphically represented in Figure 5.

Graph order	Avg. orbit size	BDD		Brute-force	
		Time	Mem. usage	Time	Mem. usage
3	4	1.15 ms	276	0.04 ms	384
4	11	4.82 ms	1038	0.26 ms	1248
5	45	19.59 ms	4803	2.65 ms	6628
6	197	77.82 ms	29 782	23.3 ms	36 278
7	659	232 ms	150 616	149 ms	152 630
8	1980	500 ms	594 512	779 ms	555 945
9	6307	2452 ms	2 781 694	4302 ms	2 187 004
10	40434	19 586 ms	16 311 721	24 441 ms	9 069 840

Table 5: Average computation time and memory usage (in bytes) for running `ComputeOrbit` on 10 randomly generated graphs for $|V|$ ranging from 3 through 10, for both gs_bdd and brute_force. For reference and reproducability, we also included the average orbit size for the 10 random graphs.

In the following table, the entire distribution of orbit size frequencies for all graph orders, resulting from our third experiment in which we generated orbit datasets, as detailed in Section 5.3, is shown.

Table 6: The frequency of all orbit sizes for graph orders 3 through 8.

#Nodes	Orbit size	Freq. (#orbits)	#Nodes	Orbit size	Freq. (#orbits)
3	1	4	8	52	840
3	4	1	8	54	168
4	1	10	8	55	210
4	4	4	8	56	560
4	5	1	8	57	210
4	11	3	8	58	280
5	1	26	8	60	210
5	4	20	8	61	420
5	5	5	8	62	840
5	6	1	8	63	210
5	11	15	8	64	280
5	14	10	8	66	280
5	30	15	8	82	5040

Continued on next page

Table 6: Full results of the third experiment (continued)

#Nodes	Orbit size	Freq. (#orbits)	#Nodes	Orbit size	Freq. (#orbits)
5	132	1	8	104	3360
6	1	76	8	106	5040
6	4	80	8	108	840
6	5	30	8	110	2520
6	6	6	8	112	2520
6	7	1	8	120	840
6	11	90	8	121	315
6	14	60	8	126	840
6	16	10	8	130	1680
6	17	15	8	132	840
6	18	10	8	134	1680
6	30	90	8	135	420
6	38	60	8	136	1680
6	39	45	8	138	2520
6	40	15	8	139	1260
6	41	15	8	140	2100
6	82	90	8	142	2520
6	132	7	8	145	1260
6	176	45	8	148	105
6	372	15	8	149	420
7	1	232	8	151	630
7	4	350	8	152	315
7	5	140	8	176	2520
7	6	42	8	220	840
7	7	7	8	224	5040
7	8	1	8	232	2520
7	11	420	8	236	5040
7	14	420	8	264	210
7	16	70	8	284	3360
7	17	105	8	288	4200
7	18	70	8	290	5040
7	20	56	8	294	2520
7	22	35	8	296	3360
7	30	630	8	300	2940
7	38	420	8	302	2520
7	39	315	8	306	5040
7	40	105	8	308	2520
7	41	105	8	312	2520
7	44	105	8	314	2520

Continued on next page

Table 6: Full results of the third experiment (continued)

#Nodes	Orbit size	Freq. (#orbits)
7	46	105
7	48	175
7	50	315
7	52	105
7	82	630
7	104	420
7	106	630
7	108	105
7	110	315
7	112	315
7	132	49
7	176	315
7	220	105
7	224	630
7	232	315
7	236	630
7	372	105
7	484	315
7	492	630
7	504	315
7	528	21
7	532	30
7	1052	360
7	1056	105
7	1096	105
8	1	764
8	4	1456
8	5	700
8	6	224
8	7	56
8	8	8
8	9	1
8	11	2100
8	14	2240
8	16	560
8	17	840
8	18	560
8	20	448
8	22	280
8	318	3360
8	372	840
8	484	2520
8	492	5040
8	504	2520
8	528	224
8	532	270
8	612	6720
8	616	2100
8	636	840
8	640	5040
8	644	5040
8	648	11 339
8	652	1260
8	656	5040
8	660	1316
8	668	12 600
8	680	3150
8	692	840
8	704	630
8	1052	2880
8	1056	840
8	1096	840
8	1320	2520
8	1356	5038
8	1368	1260
8	1380	12 599
8	1404	10 079
8	1408	5040
8	1424	10 496
8	1428	2520
8	1436	840
8	1448	2520
8	1452	168
8	1468	5460
8	1492	420
8	1628	210
8	2784	1
8	2848	2

Continued on next page

Table 6: Full results of the third experiment (continued)

#Nodes	Orbit size	Freq. (#orbits)
8	23	28
8	24	56
8	25	35
8	26	56
8	27	35
8	30	3360
8	38	3360
8	39	2520
8	40	840
8	41	840
8	44	840
8	46	840
8	48	1400
8	50	2520
8	2904	210
8	2932	2520
8	2952	3360
8	3004	10 077
8	3024	840
8	3032	2520
8	3072	2520
8	3088	5040
8	3092	5040
8	3156	3480
8	3160	2520
8	3168	35
8	3248	315