Universiteit
Leiden
The Netherlands

Bachelor Computer Science

Optimal graph-state preparation
for quantum computers with SAT-Solvers

Melvin Roestenburg

First supervisor and second supervisor:
Alfons Laarman & Sebastiaan Brand

BACHELOR THESIS

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

09/07/2024

Abstract

This thesis examines quantum state preparation and optimizes the preparation by minimizing the amount of two-qubit gates using an SAT-Solver and a cardinality constraint. With this, we investigate if an SAT-Solver is a practically efficient method for finding the minimal quantum state graph using single-qubit Clifford gates, where a quantum state graph represents a quantum state. SAT-Solvers have been used within these problems but yet has anyone looked at a cardinality constraint with SAT-Solver to find this minimum graph. We did this by transforming this problem into a SAT problem, where we experimented which SAT-Solver, search algorithm, and cardinality constraint are most efficient. We did this to optimize the quantum state preparation, where we find the minimum amount of two-qubit gates needed to reach a given quantum state.

1 Introduction

In the last couple of years, there have been a lot of changes and research with quantum computers and quantum networks, before being able to use a quantum network, the quantum state needs to be prepared. This is being done with different types of quantum gates, to create a working system between the qubits.
A problem is the optimization of quantum computers and network preparation. These quantum states are transposable to a graphical representation. Therefore, by optimizing the graph problem, we also optimize the quantum state preparation problem.
There has already been research on such graph problems. One of the operations our graph problem uses is the local complement, this operation has been thoroughly researched by informatics. For example, Bouchet made an algorithm for proving locally equivalent graphs[Bou91]. But other than the combinatorial research, there has also been research with quantum state preparations. This has been done by Brand[BCL23], who looked at transforming graphs to a target graph on local complements, and vertex deletions. Other research has been done to find the complexity of the preparation with only local complements by Kumabe[KMY24].
However, our research will work on the problem of efficiently finding the lowest locally equivalent graph. Which represents finding the quantum state we can create with a minimal amount of two-qubit gates. This is the case because the local equivalent graphs are a substitution for certain single-qubit gates. With this graph problem, we can search for a state with the question, what single-qubit operations must be used to have the minimum amount of two-qubit operations? To answer this question we make a conversion to a SAT encoding to get an efficient solution with the use of SAT-Solvers. With the encoding, we can provide an efficient solution for preparing quantum states. As well as seeing if SAT-Solvers are an efficient tool to use in solving this problem, and which SAT-Solver is the fastest in solving this problem. Finding what kind of SAT encodings will improve the time spent finding the solution. Thereby looking at different ways of using a cardinality constraint.
For our research aim we will focus on the research question: What is a practically efficient method for finding $\min_{|E|} G(E, V) \in \mathit{orbit}(T)$ given a target graph T?

2 Background information

Before we delve deeper into the experiments we explain some key concepts. Within this background section, we give a deeper understanding of the definition of quantum states and how they are represented within graphs. The kind of operation we are performing on these graphs and what kind of representation they have within quantum computers. What the definition of an orbit is within this problem set. Lastly, some key concepts why this is not answerable with a simple or trivial solution.

2.1 Quantum states as graphs

Before the quantum computer can perform tasks we have to prepare its state. The state preparation we are looking at are various ways qubits can connect. This creation of connection of possible gates is defined and explained within Quantum computation and quantum information[NC02]. The quantum state is the way the quantum system has been made and represents the way the qubits are connected with these gates. However, we will only be looking at single-qubit Clifford gates to solve our research question. The advantage of single-qubit Clifford gates are that this problem can be transposed into a graph problem which is explained by Van den Nest et al[VdNDDM04]. With these graphs, we will represent two types of gates, the controlled-Z gates and single-qubit Clifford gates. These controlled-Z gates could lead to noise within a quantum computer, and they take more time to be created than single-qubit Clifford gates. That is why this research aims to reduce the amount of controlled-Z gates we need to use.
We define our graphs with a finite number of qubits where each vertex in our graph represents a qubit. The edges between the qubits grasp the information regarding the quantum state's entanglement. Where there is an entanglement between qubits i and j if and only if there is a path between i and j. Lastly, our graphs are simple graphs meaning we have an undirected graph with a maximum of one edge between two vertexes, and no self-loops.

2.2 Controlled-Z gates as a graph operation

The first operation we can perform on our defined graphs is the CZ operation. The CZ corresponds to a controlled-Z (CZ) gate, which creates an entanglement between two qubits. Although we represent the CZ gate on quantum computers, which could fail to be created or create noise, as said before we do not look at the probabilities of operations or noise created within the quantum system. An example of what a CZ operation does on our graph is provided in Figure 1.

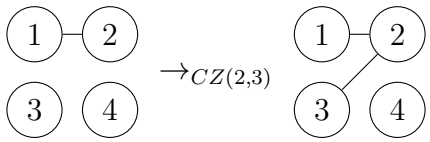

Figure 1: An example of a CZ operation on a four qubit quantum state graph.

2.3 single-qubit Clifford gates

The single-qubit Clifford gates work only on a single qubit. This operation is less noise introducing than the CZ gate because CZ gates work on two qubits while the single-qubit Clifford gate works on a single qubit. Plus the gate only performs on its local qubit, meaning we do not need to search for contact with another qubit. This is a problem when your qubits are in completely different locations making local gates more profitable. For such a network there is an example where there has been a proposition to make a quantum network of 14 qubits where all qubits are in different Dutch cities[RCAW22]. Therefore it is better if we use the minimal amount of CZ gates and replace the most with single-qubit Clifford gates.
These single-qubit Clifford gates are equivalent to local complements (LC) on a vertex within our graph[VdNDDM04]. Such an LC on a vertex will switch all edges of all neighboring vertexes. An example of an LC is in Figure 2.

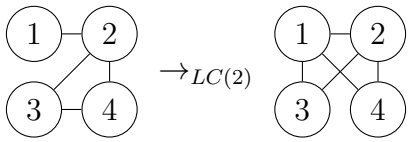

Figure 2: An example of an LC operation on a four qubit quantum state graph.

2.4 Orbit

Within the research question, we had it about searching for the minimum amount of edges within the orbit. We define an orbit of a graph state as all graph states we can create with the LC operations. This means that if we want to know if graphs A and B are in the same orbit. We need to find the LCs on A which will create graph B. If this is discoverable then they are in the same orbit. To get out of this orbit we can perform a CZ operation. This does not necessarily mean we are out of the orbit but CZ operations are the only way we can get to a new orbit within our rule set.

3 Method

3.1 Different Solvers

Initially, we wrote two simple methods before working on the optimization of the SAT-Solvers. The first two methods are a bruteforce and a greedy approach written in C++. With these methods, we get some insight into how this problem behaves. If our greedy program will always find the optimal solution we will be done with our research. Also, the greedy solution will be compared to how close it will be to our optimal solution. It could be that a greedy solution is practically a better solution than an optimal solution. But the other side also holds for our bruteforce, if our bruteforce is quick, it is already a practically efficient solution and we will be able to stop there. Otherwise, we have a comparison to see how much faster our optimal is.

3.1.1 Bruteforce solution

The bruteforce algorithm needs to begin with an initial starting graph state. Then it goes through all possible local complements until it has the whole orbit. Every time we visit a new graph we need to count the amount of edges within the graph. When it has collected the whole orbit it looks at which graph has the minimal amount of edges. The drawback of this bruteforce is that it calculates and saves the whole orbit before providing an answer.

3.1.2 Greedy approach

Initially, we made a simple greedy approach. This simple greedy algorithm performs the LC on all vertexes and chooses the smallest amongst them for the next cycle. It will repeat this cycle till the LC does not give a smaller graph. This simple strategy does not give us the smallest graph for our problem. For example, consider a circle with four qubits in Figure 3. Here it is necessary to go to a larger amount of edges before getting to the minimum number of edges within the orbit. The final solution has a minimal amount of edges because it has no loops.

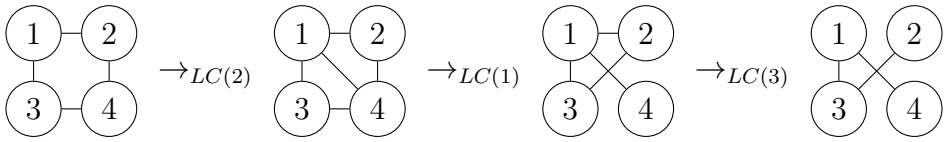

Figure 3: An example of LC operations finding the minimal amount of edges in a four-qubit graph.

This kind of greedy is simple and does not take a lot of memory or calculating power. Also in this kind of greedy we know which vertexes we have performed local complement to get to our minimal graph.
The solution to the previous example had $|V| - 1$ edges. This raises the question if it is always possible to create $|V| - 1$ edges. This is the smallest any graph will be. However, there also is a simple example of a circle with five qubits. The minimum amount of edges within the orbit is drawn at the top of Figure 4. The operations have been simplified for a better overview, all isomorphic graphs have been shown as one graph. A graph state with a circle of five holds the circle of five throughout all LC operations. LCs are only able to add extra edges. This shows that always getting $|V| - 1$ edges is not guaranteed.

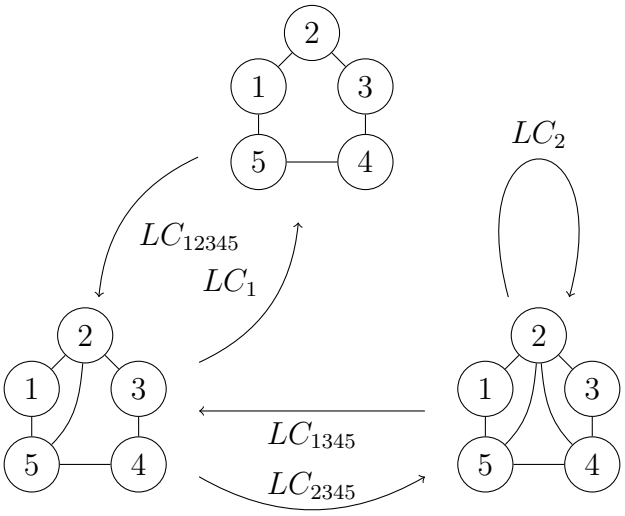

Figure 4: Circle of five-qubit orbit, where all isomorphic graphs are shown as one graph.

The previous graphs depict that simple greedy will not give us correct answers for even simple instances. Therefore it is necessary to look into better solutions. However, the implementation of the greedy can give us insight into how good the accuracy is. Which might practically be better than calculating the correct answer with certainty.

3.1.3 Different SAT-Solvers

To optimize the solution, we want to transition to SAT to be able to solve this with SAT-Solvers. SAT is the question if a boolean formula is satisfiable, in particular, a formula in CNF. SAT is a problem that is NP-complete [Coo71] which means every problem in NP can be transformed into an SAT problem. Where it is possible to transform our research question into an SAT formula. The SAT-Solvers are constantly improved by other people, more than any other NP-complete problem. Although we do not know if our problem is in P, it is likely not possible to solve this problem within a polynomial time.
SAT-Solvers are solvers made for solving the SAT problem as optimally as possible. To do this they are created to solve large problems of CNFs. There are three SAT-Solvers we will measure their performance, Z3 [dMB08], Kissat[BF22], and Glucose4[AS18].
To create our SAT formula we had the help of the code from Brand[BCL23], they had a solution for finding a path between a source graph and a target graph with an extra rule the vertex deletion. This operation is not needed, that is why we will remove this operation from the SAT encoding. Once removed, we have created a transformation from our problem towards an SAT problem.

3.2 The SAT encoding

The SAT encoding for our problem will be formed from the SAT encoding by Brand[BCL23]. Only the vertex deletions need to be removed and the target graph needs to be exchanged before it is able to be used for our problem.

However, we begin by explaining the current SAT encoding. There are in total three different kinds of variables in use. The x variables keep track of all the edges. The y variable holds the vertexes on which an operation will be performed. The z variable which holds the operation is being used. Every variable holds extra information in its name, for all of them the timestep at which the variable holds. There we have the x variable which also holds the vertexes by which the edges are within the graph. Here the name created is $x\_\{i\}\_\{n\}\_\{t\}$ where i,n is in V and t expresses the timestep. With this encoding, it represents the source graph at value zero. For all different possible operations, the formula will add the corresponding edges at the corresponding timesteps. The y variable has the name $y\_\{vertex\}\_\{timestep\}$ and the z variable has the name $z\_\{timestep\}$ both of them together will decide on which vertex at which timestep will receive what for operation. However, our SAT formula needs to exclude the implemented vertex deletions. The operation has been encoded by which the z variable has to be false to represent an LC. We have to add the constraint that the z variables may only be false. With this, the encoding has been transformed. An example of how we can create the encoding in one step:

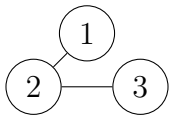

Figure 5: Example three-qubit graph.

We have three vertexes 5 and can perform our LCs on 1 2 and 3. First, we recreate the source graph with the SAT encoding:

$$x\_1\_2\_0 \wedge x\_1\_3\_0 \wedge -x\_2\_3\_0$$

Then we create the operations and the vertexes belonging to the operation. $z\_\{timestep\}$: z_0, where $z\_\{timestep\}$ = false means an LC. Then the vertexes are $y\_\{node\}\_\{timestep\}$ connecting the operation at certain vertexes to operate on. This whole makes us the formulas to append:

$$\neg z\_0 \wedge y\_1\_0$$
$$\neg z\_0 \wedge y\_2\_0$$
$$\neg z\_0 \wedge y\_3\_0$$

Here every line is a different vertex where we perform an LC on. Lastly, the corresponding graphs need to be added with the x variables. The last part is done by the formulas Brand[BCL23]:

$$LC_k = \bigwedge_{(u,v)\in\mathbb{U}} \begin{cases} x'_{uv} \leftrightarrow \neg((x_{uk} \wedge x_{vk}) \oplus \neg x_{uv}) & \text{if } u \neq k \wedge v \neq k \\ x'_{uv} \leftrightarrow x_{uv} & \text{otherwise} \end{cases}$$

The part we added to form the SAT encoding to our research question is the removal of the vertex deletions and ensuring that still $\neg z\_\{timestep\}$ is being kept for LCs. That is why we will add it to the encoding ensuring that $\neg z\_\{timestep\}$ will always be taken.

3.3 The cardinality constraint

Then for the last step, we need to create our cardinality constraint, with the use of pysat we made the target graph as the pysatcard function. This function can encode the lesser or equal then constraint on the edges of the target graph. The cardinality constraint limits the number of variables that are allowed to be true. Where we can decide in an SAT encoding how many edges are maximally allowed. By using this as our target graph it restricts the SAT-Solvers to solve the problem for a target graph with a certain bound, amount of edges.
There are various cardinality constraints available for SAT formulas. To get these encodings we use the pysat library. In total we use 4 cardinality encodings, the sequential counter by Sinz[Sin05], the sorting network by Batcher[Bat68], the cardinality network by Asin[ANORC09], and the kmtotalizer by Morgando[MIMS15]. There are 2 more totalizers within the pysat, but these are excluded. One totalizer is enough to see if it is an efficient solution. From the totalizers, we chose kmtotalizer on the fact that it had the least amount of help variables created out of all the totalizers. This choice has been made because the totalizers were best at keeping the introduced help variables the lowest. And between all the totalizers the kmtotalizer was the best at it. By doing this we can see which idea or cardinality encoding will be most efficient for our problem.

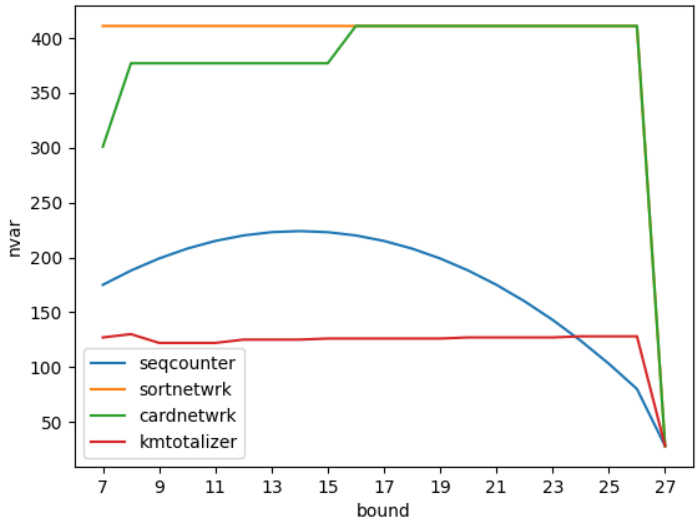

Figure 6: The amount of variables against the bound for different cardinality encodings.

In Figure 6, The different cardinality constraints and the different amount of variables they make are plotted. There are some interesting properties to be seen, the sequential counter has a parabolic function that starts in the middle but on the higher bound, it has the lowest amount of variables. Then the kmtotalizer is mostly constant and low in the creation of its variables, the cardnetwork and sorting network are the highest in the number of variables, although the sortnetwork is also really constant in the number of variables. All graphs converge to the same spot at bound 27 because we have done this test for 8 nodes creating at most 28 edges, if we want less or equal to 27 edges, it is all or's over not all edges.

4 Implementation

4.1 Saving graph

First up we had to save our state in some way in the C++ program. We used a half adjacency matrix saved in a boolean single array. All the edges can be looked up with $index = (from * vertexCount) + to - (from + 1) * (from + 2) * 0.5$. This method is efficient in both speed and memory. For the Python SAT-Solvers, we used the implementation of Brand[BCL23], they made a Python object that saved a graph.

4.2 Bruteforce

Algorithm 1 Bruteforce

```
function getOrbit(graphs, depth)
    set orbit = graphs;
    if depth > MAX then
        return graphs;
    end if
    for graph in graphs do
        for i in graph.Vertexes do
            orbit += graph.LC(i);
        end for
    end for
    if graphs.size() == orbit.size() then
        return orbit;
    end if
    return getOrbit(orbit, depth+1);
end function
function Bruteforce(graph)
    set orbit = getOrbit(graph, 1);
    for g in orbit do
        if g.nEdges < graph.nEdges then
            graph = g;
        end if
    end for
    return graph;
end function
```

The bruteforce is a simple implementation of getting all graphs within an orbit and then choosing the smallest graph. So to calculate the orbit we first have to calculate the number of steps we have to take, and how many LCs we have to look at before getting to the orbit. The amount of steps we have maximally to take is captured within the variable MAX. Then we save our graph in a set and perform LCs on all vertexes. Then repeat it for all new graphs and reduce the amount of LCs we have to take with one. When we have done all trajectories we have created the whole orbit and have to search through the set to find the smallest in orbit.

4.3 Greedy

The greedy algorithm is exactly:

Algorithm 2 Greedy

```
function GREEDY(graph)
    done = false;
    while not done do
        done = true;
        for vertex in graph.vertexes do
            new = graph.localComplement(i);
            if new.nEdges < graph.nEdges then
                graph = new;
                done = false;
                break;
            end if
        end for
    end while
    return graph;
end function
```

We also made a simple greedy, this is done by taking again LCs on all vertexes, but instead of saving them in a set, we will take the smallest one from them. Then we repeat this until there are no smaller graphs than the current graph.

4.4 Implementation of the SAT

To implement the SAT encoding in an experiment we used the CNF which was created by the Brand[BCL23]. We used this CNF to correctly implement the SAT encoding which is explained in the method section. The only step it needed was to go from pysat to the CNF form to implement our cardinality constraint. This is where we created a definition that performs this transformation. This is done by creating a mapping from the CNF to pysats CNF, at which we encode all the edges. Then pysat can produce the cardinality constraint in pysat, whereafter we transform the encoding of the edges back to CNF with the mapping and create new variables for the help variables created by the cardinality constraint.

4.5 Creating graphs

The graphs we used for testing our experiments are Erdos-Renyi graphs. An advantage of using these graphs is that we can easily decide what density the graph has. This is done by giving a probability at which every edge has to exist. When we give this a high number like 0.9 we get almost complete graphs, while having a 0.6 probability results in a less dense graph. Hereby Erdos-Renyi graphs have the trait that above the threshold $\frac{\ln n}{n}$, it has a high probability of creating a connected graph[ER84]. Because an unconnected graph is undesirable for our tests we use this property to get all graphs to be connected.

4.6 Implementing the cardinality constraint

Then for the experiments, we need to search for our lowest k, where we have to find a graph with $|E| \leq k$. We start at an upperbound, $|E|$ which is found at zero steps, and zero operations. The upperbound for the amount of steps the calculation takes is the total amount of LC possible on the graph given by Brand[BCL23]. Because we focus on finding the smallest graph in the orbit we do not have to search over the amount of steps. We do this with different search algorithms that every time search for the best low bound and upperbound until the smallest bound is found. Through an identity operation, not all steps need to be filled with LC operations.

4.7 Result gathering

When our SAT-Solvers have calculated the best bound, we also want the amount of time spent on encoding and solving the SATs. To get a practically efficient way of finding our smallest graph we put a timeout of 30 minutes. Hereby we say that when our SAT-Solvers take longer than 30 minutes it is not practically efficient anymore.
The extra programs at which we look are the implementations of a greedy and bruteforce algorithm. Because these will be our baselines. The optimized solution must be faster and as accurate as the bruteforce algorithm to be more efficient. The greedy is likely to have quick solutions but are far from the truth. But in the case greedy is not far off from the truth it would be practically better to use the greedy algorithm.

5 Experiments

The experiments performed are to evaluate the efficiency and accuracy of the different approaches. Here the different greedy approaches, the bruteforce approach, and different SAT-Solvers are being tested. To test this on controlled instances, we made Erdos-Renyi graphs on densities from 0.6 to 0.9 and sizes from 4 to 14 qubits. We have chosen to go to fourteen qubits because of the proposition to create a quantum network with fourteen qubits[RCAW22]. All these parameters let us compare the different approaches between each other and classify the advantages and disadvantages of each approach.

5.1 Greedy and bruteforce

The greedy and bruteforce algorithms used here are explained within the implementation section. The greedy will be used to see how accurate it is. The bruteforce is checked upon the efficiency it performs. Both of these algorithms were programmed within C++ while the implementation of our SAT-Solvers has been done in Python.

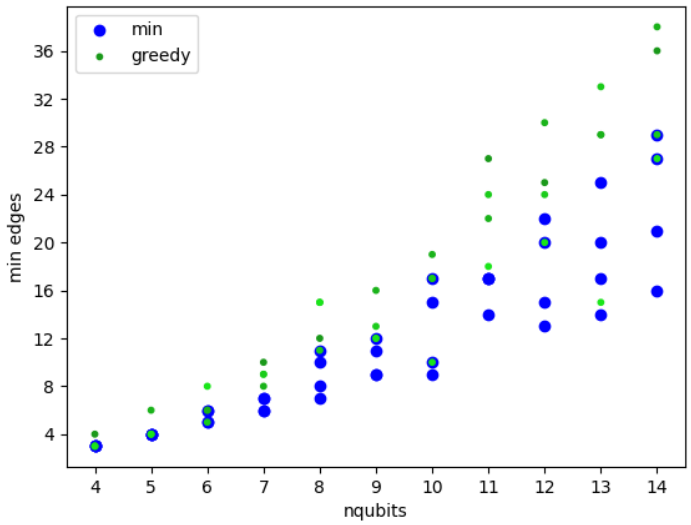

Figure 7: Greedy against bruteforce true minimal

In the above Figure 7 a simple greedy solution is significantly above the minimal solution. Moreover, it seems to have a sharper curve than the minimal. A better greedy algorithm could have a significant help in creating quantum states. However, this simple is not a practically efficient solution.

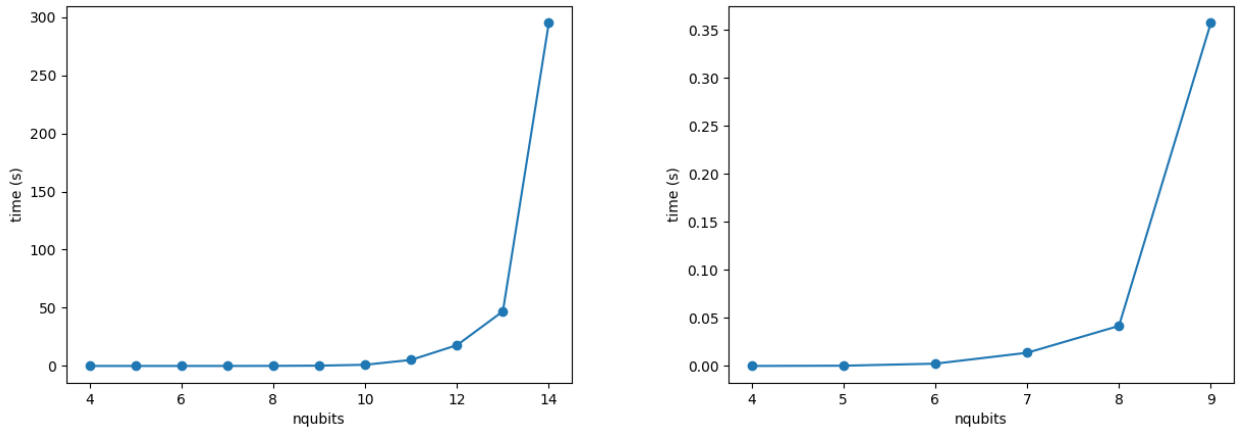

Figure 8: Bruteforce average times, left 0.7 dense Erdos-Renyi graphs, right 0.6 to 0.9 dense Erdos-Renyi graphs.

As shown in Figure 8, we have the average times of the bruteforce solution. The bruteforce approach is practically efficient in terms of speed with a solution on fourteen qubits within 300 seconds. Although the speed, this approach does take around 7.8 GB of memory to search the minimal amount of edges on fourteen qubits. Which could only grow if we want to calculate larger amounts of qubits.

5.2 The SAT-Solvers

To test the efficiency of our SAT-Solvers we tested them against the same graphs as the greedy and bruteforce. These Erdos-Renyi graphs were made with different densities and different sizes of qubits. Initially checked how much time the SAT-Solvers took to find the optimal solution for every instance in our test set. After the tests, the best SAT-Solver will be further tested with different parameter settings.

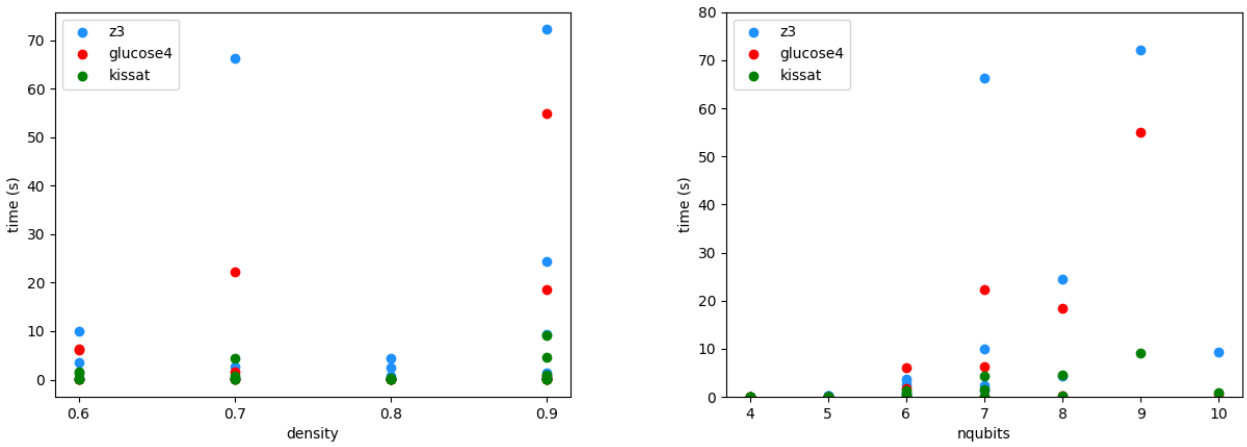

Figure 9: SAT-Solvers times, left different density Erdos-Renyi graphs, right different size qubits.

As shown in Figure 9 the best performing SAT-Solver on density and number of qubits is Kissat. Z3 is the worst in time performance of all three SAT-Solvers. Based on these results further experiments will be done with the Kissat.

5.3 Search algorithms

Using the SAT-Solver Kissat we search for the best algorithm to find the minimal bound. This experiment tests if there is a difference between bound search algorithms. Hereby we have a test set of only 0.7 dense graphs. We chose 0.7 because it makes higher bounds while still getting to a high amount of qubits without failing. The minimal amount of edges for 0.7 dense graphs is not always $|V| - 1$, which is the case for a low and high bound. We will test four kinds of search algorithms on the graphs.
The first one is a simple linear search where we start at the smallest possible graph and try a bound higher until we find a satisfying assignment, which will be the answer to the problem.
Secondly, we have a binary search over the bound. This way we first take the lowerbound as $|V| - 1$ edges and the upperbound as the number of edges within the source graph. The binary search puts the new bound between the lowerbound and upperbound. If it solves an unsat it moves the lowerbound to the new bound. If it is a sat we move the upperbound to the new bound. This way it finds the best bound when the lowerbound and upperbound are the same. Because binary search puts the new bound halfway between the lowerbound and upperbound, it has to solve fewer bounds before getting to the best bound. Hopefully, this approach lessens the search time, especially for larger graphs.

Thirdly, we have an algorithm where we decide how many LC steps may be done before reaching our final target graph. For the upper two, we put this on the maximum of possible LCs there are within the orbit. However, this creates unnecessary clauses which could lead to a higher solving time. With deciding the steps we try to minimize the number of needed clauses. This algorithm starts with one LC step and searches for the minimum bound. This way it has a better upperbound for the upcoming solutions. Then it increases the amount of steps with one and finds again the best bound. We do this till we have had all the steps maximum possible. So because we still have to do the maximum amount of LC it will be slower. But it is great to see the minimum amount of steps it takes to get to our minimum graph.
Fourth is the linear algorithm, however this time from upperbound to lowerbound. This reverse linear search starts at the upperbound and finishes when an unsat has been found. In that case, The previous sat is the minimal bound. The idea with the reverse linear is that we only have to encounter one unsat. There may be a significant difference between sat and unsat solve times. Then we can implement a timeout in which the algorithm has a high probability of finding the minimal bound. Although it did not find the unsat.

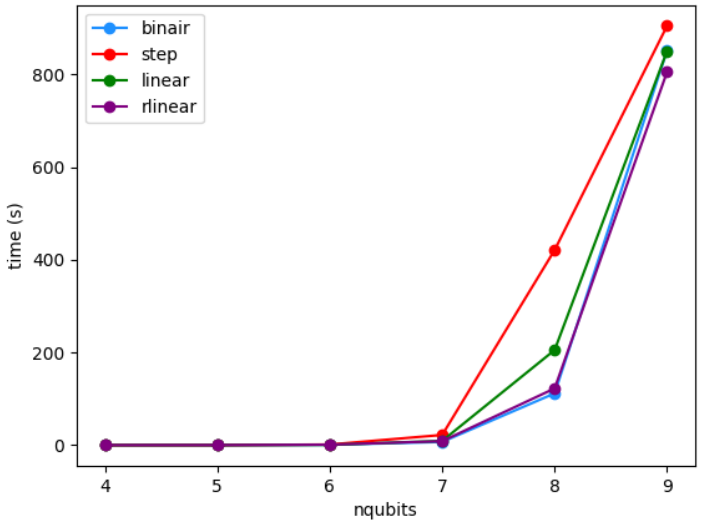

Figure 10: The search algorithms and their time on 0.7 Renyi graphs.

As expected in Figure 10 the step search is the slowest. However, between linear and binary is not much difference in the solve time in the lower amounts of qubits. Although on average binary is faster than linear. Even if we had more qubits, for example, 14 qubits. In this case, we have a minimal bound on 27 edges and starting on 53 edges. This means that linear will do $27 - 13 = 14$ unsats before getting to the final sat solution. While binary will take 33, 22, 27, 24, 25, and 26 as bounds, giving 6 checks. These checks will be 4 unsats against the 14 we had with linear.
Although binary search is a fast search algorithm, the reversed linear is even faster. This gets the point that the unsat is best to avoid while solving this problem. With the reversed linear it does matter less that it solves more sats. To be exact for the previous example, it has to solve $52 - 27 = 25$ sats and one unsat. Although it has more formulas to solve, it is the quickest out of all search algorithms in getting to the minimal bound.

5.4 Different cardinalities

As shown in Figure 6 the cardinality constraint creates a large amount of help variables. This does not help in easier formulas. It could be possible that different cardinality constraints have different times associated with them. Therefore we experiment with several different cardinality constraints. We have been using the sequential counter, but we will also experiment with the sorting network, cardinality network, and kmtotalizer.

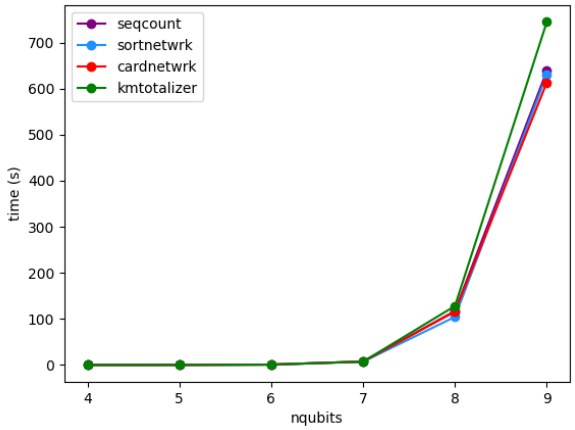

Figure 11: different cardinality encodings on 0.7 Renyi graphs

Above in Figure 11, we can see that kmtotalizer is the worst. But between the others, there is not a great difference. The sorting network is on average the fastest out of the three. However, we did not test further, where the cardinality network could become the faster cardinality constraint. It is also notable that the sorting network was the largest in creating help variables as shown in Figure 6.

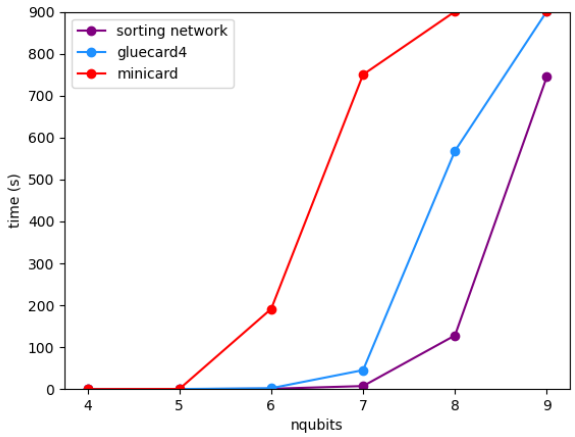

Figure 12: different native cardinalities on 0.7 Renyi graphs and sorting network not native with timeout on 900 seconds.

Aside from the SAT-Solvers mentioned so far, there are also several SAT-solvers with a native

cardinality constraint. These include Minicard[LM12] and Glucose4[AS18]. Pysat calls the native solver of Glucose4, Gluecard4. In Figure 12 we have the average solving time of each of the native solvers in the graph, plus the fastest previous cardinality encoding. However, it is shown that the native solvers are not better than the sorting network encoding.

5.5 Best SAT-Solver approach

In this experiment, we combined all the best approaches. This experiment shows if the approaches complement each other and make a better algorithm. With these results, it shows if it is an efficient algorithm for solving the problem. To determine if the algorithm is efficient, we put a timeout on 30 minutes, because if it takes longer we consider it an inefficient algorithm.

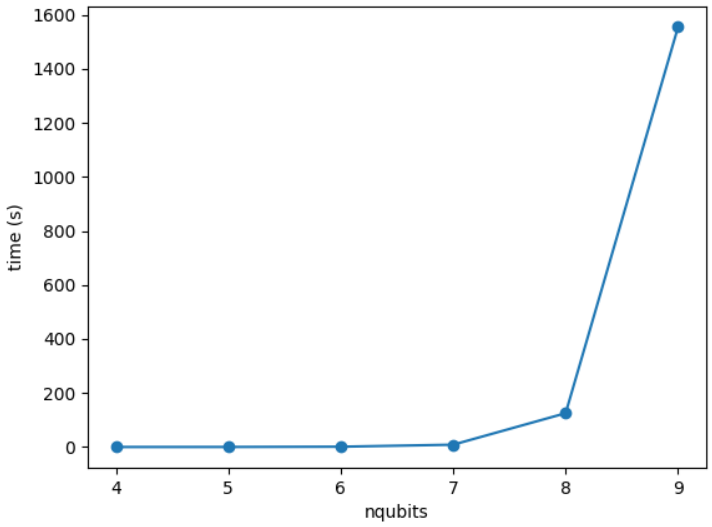

Figure 13: The rlinear algorithm with sorting network cardinality encoding on solving time timeout on half an hour.

in Figure 13 the best solution times are shown. All the previous best solutions, do complement each other, but it cannot get further than 9 qubits within half an hour on 0.7 Erdos-Renyi graphs. Although this solution will indeed find the minimal amount of edges decently fast. The unsat is hindering the SAT-Solver.

5.6 Sat against unsat

We see a difference between the calculation time in an sat against an unsat. Noting these differences gives us insight into where most of the solving time is invested. Showing if it is better to avoid unsats or sats or if the distinction does not matter.

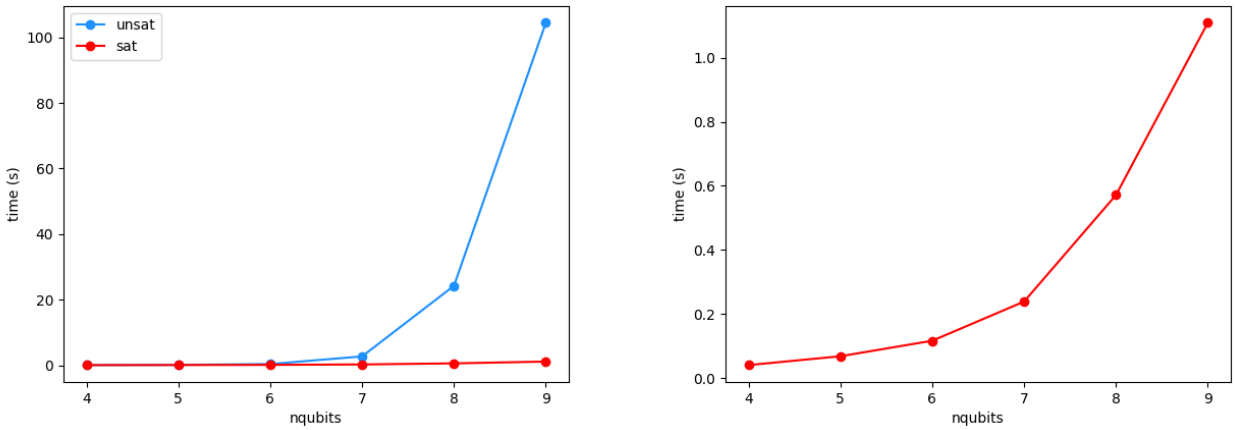

Figure 14: Left sat against unsat on time, right sat on time.

In Figure 14 is shown that our sats take seconds to solve. Do note that sat is not constant what might look like in the left part of Figure 14. But the majority of time is put into our unsat, Unfortunately, a search algorithm has to take at least one unsat before knowing the minimal graph in orbit. Except if the smallest graph has $|V| - 1$ edges. This is why SAT-Solvers are not efficient in solving this problem, although we can optimize a search by taking only one unsat.

5.7 Better greedy solutions

In the previous paragraph, we discussed the great time loss on the unsat statements before reaching certainty over the minimal amount of edges. Although despite the long solving time, the best solution is found faster. Thereby there are several search algorithms better than others in providing a greedy solution. Here the algorithms focussing on keeping solving time low before continuing are the best greedy solutions. The search algorithms are not all that great but with the rlinear and steps implementations they follow the least resistant solve time towards the optimum solution. However, checking if the solution is correct might take longer.

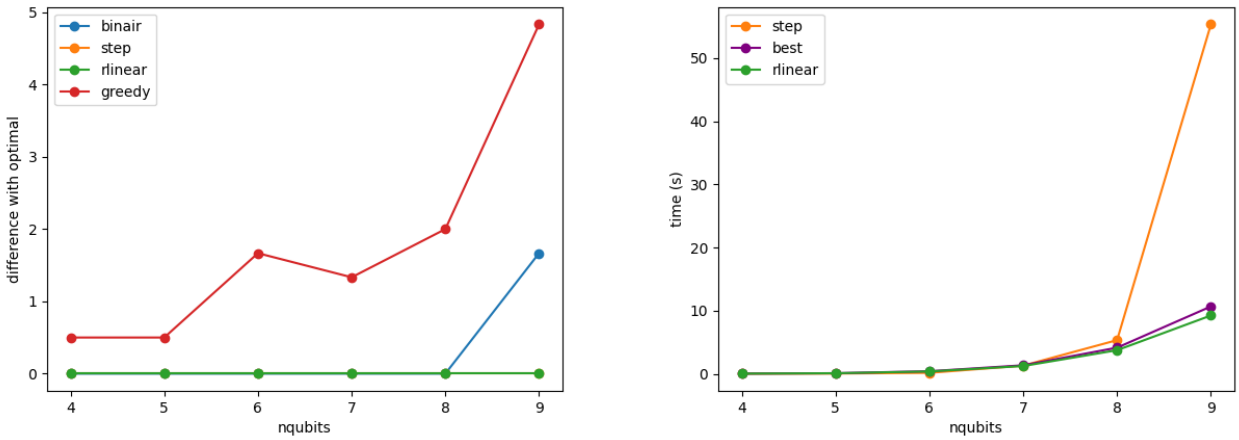

Figure 15: Left: different search algorithms how far off the bound was in 15 minutes. Right: time to find greedy bound.

For step search the idea is to reduce the solve time by looking first at smaller steps, taking less time to find unsat solutions with lesser steps. As shown in figure 15 this certainly is taking less time. The drawback is that for certainty it has to do the maximum amount of steps unless it has seen $|V| - 1$ edges.
The reverse linear is almost the same as linear the only difference is that we start at the upperbound and work our way to the lowerbound. Where uppon finding an unsat the previous one would be the minimal amount of edges in the orbit. As you see for the solution we only have to find one unsat as we know the unsat takes the most time, and this is certainly to our advantage. But what you can notice is that the smallest amount of edges is most of the time on the lower side. This means we have more calculations to perform, but as long as this is faster than going through the unsat, we will probably have the answer the moment we have a long wait and can cut off prematurely.
Lastly, the binary search could also be a greedy approach where we can put a cutoff at a fifteen minute timeout. The drawback of the binary search is that it has a higher chance of encountering an unsat. Therefore the last one rarely is the minimal amount of edges.
On the right graph in Figure 15 we put the fastest time at finding the minimal bound. However, we see that the cardinality of a sequential counter is faster than the sorting network. Although the difference is minimal. It could be because the encoding for the sorting network is better at finding unsats and different encodings are better at finding sats.

6 Discussion

This thesis aimed to find an efficient way of finding the minimal amount of edges within an orbit. We took this and made a bruteforce, SAT-Solver, and several greedy solutions. We optimized the SAT-Solvers with different search algorithms and different cardinality constraints.
Although we explored the different optimizations over the SAT-Solvers, we have shown that bruteforce implementations are faster than the SAT-Solvers, while they do take more memory. It would be worthwhile when trying to solve the research question to look into better bruteforce

algorithms. Despite this, an unsat instance takes most of the solve time. This is why SAT-Solvers are not efficient in solving the minimizing question. Because every solution must take at least one unsat to solve. To counter this factor we made better greedy solutions, where they would only need one unsat to solve the minimal amount of edges within an orbit. This way we can have a timeout where there is a high chance we still found the minimal amount of edges. Although we came to this conclusion, we did not further the research on how the new greedy solution performs until fourteen qubits. The same holds for the amount of graphs we have used. We did the majority of testing on 0.7 Erdos-Renyi graphs. To get more insight we could have tested on more graphs and different densities.
While most of our conclusions to use certain algorithms and encodings depended on the average time. The times itself are not constant on the same graph. Because of the randomness of the SAT-Solver, the solve times can for some graphs differentiate between the 10 seconds.

7 Conclusion

In this thesis, we searched for an optimal way to prepare a quantum state. This was done by minimizing the amount of two-qubit gates needed. To do this we optimized a SAT-encoding which searched the minimum amount of edges within an orbit. We optimized by testing different SAT-Solvers, search algorithms, and cardinality constraints. Additionally, we checked the influence an unsat and sat instances had on the total solving time.
To conclude, we have shown that Kissat is the fastest SAT-Solver to use for this problem. Out of the different cardinality constraints sorting network is the fastest. Although the sorting network makes the most amount of help variables. Lastly, we had different kinds of search algorithms to search for the bound that represents the number of edges a graph maximally has. The fastest search algorithm is a linear search starting from the upperbound. This is the case because it avoids the most unsats. Combining every optimization lets us consistently solve up to 8 qubits within half an hour.
The bruteforce algorithm could solve these problems to 8 qubits within seconds. This shows that a bruteforce algorithm calculating the whole orbit is faster in solve time. However, a drawback for the brute is that it takes a significant amount of storage to calculate the minimal graph in orbit.
Lastly, we have shown that an unsat instance takes significantly more time than a sat instance. With this, we found that the best SAT-Solver approach can find the minimal amount of edges with an average of 10 seconds for problems on 9 qubits.
This demonstrates that a SAT solution can be a practically efficient solution to finding the minimal amount of edges within the orbit.

8 Future work

8.1 Different SAT encoding

There are a few parts that could be done better, first, we take a look at our SAT encoding. Because we reformed another SAT formula for our problem our SAT formula is not optimal. This could be made better by truly getting the vertex deletions out of there and thereby the removal of the z variable. Because we will always perform the local complement operation on a vertex.

8.2 Faster SAT greedy solutions

We have seen the importance of different search strategies within this problem. With almost certainty, we can use the speed of these SAT-Solvers at which they find sat formulas. We have sorted most of our work on the finding of the minimal edges. It might be that the other cardinality constraints are faster in finding sat than the one we have chosen, because of the speed overall. For example, this is seen in the greedy solution comparison with the best overall, and reverse linear in Figure 15. If this is the case we could make faster greedy solvers.

8.3 Better Upperbound

Because the reverse linear did so well in finding the solution the fastest. It is important and could enhance performance by finding a better upperbound. Because we have seen that in most of the graphs, the optimal solution is on the lower side of the scale. Never on the high side, so it would not be surprising if we could find a better upperbound for this problem.

8.4 Better upperbound for steps

The amount of steps helps in finding the unsat as we have noticed within the step search algorithm. We know the amount of LC steps we have to take to find the whole orbit. But we do not need the whole orbit only the minimal amount of edges, and as seen the amount of steps needed to take was decently low despite what the max steps were. This part shows that there is a way we can safely lower the amount of maximum steps needed to find the minimal amount of edges. This would help immensely with the solving time needed for calculating the minimal edges. Also, We found that binary search is reliable and quick in these problems. It could be that we can create better greedy algorithms with the binary search and with steps, leading to faster finding the bound but still reducing the number of unsats. This could lead to a better performance in finding the best solution.

8.5 Different approach

In this thesis, we looked at what the smallest graph is within an orbit. However, we want a certain state with the least amount of two-qubit gates. To do this we could use the same SAT formula but replace the target graph with the graph we are searching for. Then replace the source graph with the current state(which could be empty). Lastly, we need a cardinality constraint over the amount of CZ gates. This way we need to add edge flips into the SAT formula. The answer to this question will also lead to the minimal amount of CZ gates used and might be a better formula to solve for an SAT-Solver.

References

[ANORC09] Roberto Asín, Robert Nieuwenhuis, Albert Oliveras, and Enric Rodríguez-Carbonell. Cardinality networks and their applications. In Oliver Kullmann, editor, *Theory and Applications of Satisfiability Testing - SAT 2009*, pages 167–180, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[AS18] Gilles Audemard and Laurent Simon. On the glucose sat solver. *International Journal on Artificial Intelligence Tools*, 27(01):1840001, 2018.

[Bat68] Kenneth Batcher. Sorting networks and their applications. volume 32, pages 307–314, 01 1968.

[BCL23] S. Brand, T. Coopmans, and A. Laarman. Quantum graph-state synthesis with sat. 2023.

[BF22] Armin Biere and Mathias Fleury. Gimsatul, IsaSAT and Kissat entering the SAT Competition 2022. In Tomas Balyo, Marijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda, editors, *Proc. of SAT Competition 2022 – Solver and Benchmark Descriptions*, volume B-2022-1 of *Department of Computer Science Series of Publications B*, pages 10–11. University of Helsinki, 2022.

[Bou91] A. Bouchet. An efficient algorithm to recognize locally equivalent graphs. *Combinatorica*, 11:315–329, 1991.

[Coo71] S. A. Cook. The complexity of theorem proving procedures. In *Proceedings of the Third Annual ACM Symposium*, pages 151–158, New York, 1971. ACM.

[dMB08] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[ER84] Paul L. Erdos and Alfréd Rényi. On the evolution of random graphs. *Transactions of the American Mathematical Society*, 286:257–257, 1984.

[KMY24] S. Kumabe, R. Mori, and Y. Yoshimura. Complexity of graph-state preparation by clifford circuits, 2024.

[LM12] Mark H. Liffiton and Jordyn C. Maglalang. A cardinality solver: More expressive constraints for free. In Alessandro Cimatti and Roberto Sebastiani, editors, *Theory and Applications of Satisfiability Testing – SAT 2012*, pages 485–486, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[MIMS15] António Morgado, Alexey Ignatiev, and Joao Marques-Silva. Mscg: Robust core-guided maxsat solving. *J. Satisf. Boolean Model. Comput.*, 9:129–134, 2015.

[NC02] M. A. Nielsen and I. Chuang. *Quantum computation and quantum information*. American Association of Physics Teachers, 2002.

[RCAW22] Julian Rabbie, Kaushik Chakraborty, Guus Avis, and Stephanie Wehner. Designing quantum networks using preexisting infrastructure. *npj Quantum Information*, 8(1), January 2022.

[Sin05] Carsten Sinz. Towards an optimal cnf encoding of boolean cardinality constraints. In Peter van Beek, editor, *Principles and Practice of Constraint Programming - CP 2005*, pages 827–831, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[VdNDDM04] M. Van den Nest, J. Dehaene, and B. De Moor. Graphical description of the action of local clifford transformations on graph states. *Physical Review A*, 69(2), February 2004.