Conventional and Heuristic Solvers for Graph 3-Coloring

Sharath Chandra Vemula and Saikumar Yadugiri

University of California Santa Barbara
{svemula, saikumar}@umail.ucsb.edu

Abstract. Graph coloring is one of the most important problems in graph theory and is used in many real-life applications. It is used in various facets of computer science such as image partitioning, clustering, data mining, networking, etc. With various real-life applications, any improvements/data on the algorithms which solve graph coloring is a huge plus. In this project, we explored two algorithms that aim to solve the graph 3-coloring problem. The first one is an NP-complete approach that uses karp reduction to solve 3-SAT using modern SAT solvers. In the second method, we used a heuristic algorithm that partitions the vertices of the graph using the signs of the eigenvectors of the symmetric adjacency matrix. In particular, we take 2 eigenvectors corresponding to the negative eigenvalues of the adjacency matrix of the graph and partition the vertex set based on the sign patterns of their elements. We compared the execution time for both these approaches and we found that the SAT solver-based approach performs better.

- GitHub Repo for Code - `https://github.com/saikumarysk/cs292f_final`
- Google Drive For Data - `https://drive.google.com/drive/folders/10rdVu-7sS2kvKamdXf5ZgTDpQCrgduxw?usp=sharing`

1 Introduction

In this project, we're trying to solve the graph 3 coloring problem. The problem statement is simple. Given a graph, we need color each vertex of the graph using on three colors, red, green, blue such that no two vertices connected by an edge have the same edge. The coloring problem can be thought of as a partition problem in which we are dividing the graph's vertex set into 3 disjoint partitions such that there are no edges within a partition. While the number and the choice of colors are completely arbitrary, the real-world applications of the graph 3-coloring problem are enormous. Three examples of its applications are -

1. Sudoku, a famous puzzle can be reduced to graph coloring using numbers 1-9 as colors, cells as vertices in the graph, and row and column connections as edges.
2. Assigning variables to minimal CPU registers such that dependent variables are not processed simultaneously to avoid race conditions.
3. Assigning various frequencies to the transmitting towers in a GSM network.

In [2], Lovász proved that the generic graph K-coloring problem is NP-complete and can be reduced to graph 3-coloring. This means there is no polynomial-time algorithm to find a solution for the graph vertex coloring problem and that any solution that solves graph 3-coloring can solve the generic graph K-coloring problem. Because of this, there is a lot of active research going on both in industry and in academia to find good heuristic algorithms which to solve graph 3-coloring in decent time.

In our project, we explore one such spectral heuristic algorithm and compare it to the generic Karp reduction-based approach. As Graph 3-coloring $\leq_p$ 3-SAT([3]), this approach is inherently NP-complete. But with all the improvements in modern SAT solvers, this is an approach worth pursuing. The main motive of this project is to compare the time taken to solve the graph 3-coloring problem using one SAT solvers (like Z3, PySAT etc.) and the heuristic algorithm to check which is better to handle graphs with large sizes. To acheive this, we compare the heuristic approach provided in [5] and SAT solver approach using Z3 and PySAT and compare the execution time on the sparse matrix suite.

2 Structure

The rest of the report is organized as follows. We provide the required definitions required for our approach in section 2. In section 3, we formally define the version of graph vertex coloring that we are

trying to solve and two solutions which we explore in our research. We also compare the execution time for these solutions. In section 4 we provide the details of our implementation and give a brief overview of all the files in our project. In section 5, we provide our experimentation process and the results obtained. We propose a few directions this project can move towards in section 6 and conclude in section 7.

3 Preliminaries

Before proceeding further, we provide a few definitions in this section, which will help the reader to have a clear insight into the approach we used and the trade-offs we made for implementing the project.

3.1 Graphs

A graph in the context of the mathematics and theoretical computer science is an ordered pair of two sets, usually denoted by $G = (V, E)$. The first set, V is the set of all the vertices in the graph. The second set, E is the set of all the edges between two vertices. In our project, we use simple undirected graphs. Hence, we can denote an edge as a set of two vertices and we can define E as,

$$E \subseteq \big\{\{x, y\} | x, y \in V \text{ and } x \neq y\big\}$$

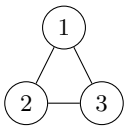


Fig. 1: Triangle Graph

An example of a graph is shown in 1. This is the famous complete graph on three nodes, also known as the triangle graph. For this graph, the vertex set is $V = \{1, 2, 3\}$ and the edge set $E = \{\{1, 2\}, \{2, 3\}, \{1, 3\}\}$. In our project, we represent graphs using the DIMACS edge format.

3.2 Vertex Coloring

Vertex coloring for a vertex colorable graph is the mapping $T : V \to C$ which maps the vertices to colors. By solving the graph vertex coloring problem, we are essentially finding the mapping T that lets us color the graph. Moreover, depending on the size of the set $|C| = K$, graph vertex coloring problem is also called as graph K-coloring problem. For instance, one possible three coloring for the triangle graph is shown in 2. Here, C ={red, green, blue} and the mapping T is such that 1 is mapped to red, 2 is mapped to green, and 3 is mapped to blue.

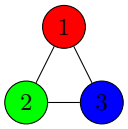


Fig. 2: 3 Colored Triangle Graph

3.3 Adjacency Matrix Representation

Adjacency matrix representation of a graph is one of the ways to represent a graph as a $|V| \times |V|$ matrix in which all the diagonal entries are 0 and the non-diagonal entries are either 0 or 1. Specifically, if there is an edge between vertex i and vertex j, $i \neq j$ then in the adjacency matrix A of the graph G, $A_{ij} = A_{ji} = 1$. For instance, the adjacency matrix of the graph shown in 1 is as follows.

$$A = \begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{bmatrix}$$

3.4 Eigenvalues and Eigenvectors of Adjacency Matrix

As the adjacency matrix is symmetric, all the eigenvectors of the adjacency matrix form a basis of $\mathbb{R}^V$. Moreover, the eigenvalues, in contrast to the eigenvalues of the laplacian matrix are represented in a "reverse" manner i.e, $\lambda_n \leq \ldots \leq \lambda_1$. For instance, the eigenvalues and eigenvectors of the triangle graph are:

$$\lambda_3 = -1, w_3 = \begin{bmatrix} -1 \\ 1 \\ 0 \end{bmatrix}, \lambda_2 = -1, w_2 = \begin{bmatrix} -1 \\ 0 \\ 1 \end{bmatrix}, \lambda_1 = 2, w_1 = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \tag{1}$$

3.5 Boolean Formulae

Boolean formulae are formulae in the predicate logic which use boolean variables and boolean operators. Recall that boolean variables can only take 2 values, True or False. For this project, we dealt with the CNF notation of boolean formulae which uses 3 boolean operators, AND($\wedge$), OR($\vee$), and NOT($\neg$). With this notation, we use two more metadata variables for the boolean formula, which denote the number of variables and the number of clauses in the formula. Consider the following formula:

$$(a \vee b) \wedge (b \vee c) \wedge (c \vee \neg a)$$

which is a boolean formula on three boolean variables a, b, c which contains three clauses. Finding a mapping that assigns the boolean variables to True or False such that the formula evaluates to True is an NP-complete problem, in general denoted using SAT. In our project, we represent boolean formulae using the DIMACS CNF format.

3.6 Karp Reduction

Karp reductions, named after Richard Karp are polynomial-time reductions for transforming the inputs of problem A into the inputs of problem B such that if the instance is solved by problem B, it can be, in polynomial time, transformed back to a solution for problem A. We say such problems are as Karp reducible and denoted using $A \leq_p B$. [3] showed that graph 3-coloring $\leq_p$ SAT.

4 Problem Formulation and Overview of Solution

Using the definitions mentioned in the previous section, we provide a definition for graph 3-coloring, and the karp reduction used to reduce the graph 3-coloring instance to a 3-SAT instance. We also provide an outline of the methodology used for the heuristic algorithm for graph 3-coloring using the eigenvectors of the adjacency matrix for a graph.

4.1 Graph 3-Coloring

Given a graph $G = (V, E)$, we need to find the mapping $T : V \to C$ with C ={red, green, blue}. This is the same as partitioning the vertex set of the graph, V, into 3 parts.

As both these problems are proven to be NP-complete, we can use SAT solvers to find the solution for any general graph. The pseudo-code for the karp reduction which lets us convert the graph instance to a boolean formula is given using algorithm 1.

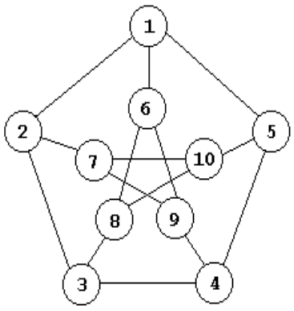
Algorithm 1 Karp Reduction from 3-Coloring to SAT

Input $G = (V, E)$
Output CNF formula F

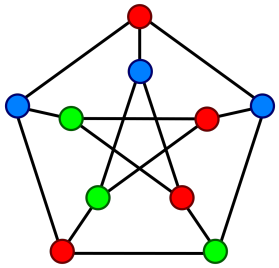
1. For each vertex $u \in$ V, create 3 boolean predicates, u_r, u_g, u_b
2. For each edge $(u, v) \in$ E, create clauses $(\neg u_r \vee \neg v_r) \wedge (\neg u_g \vee \neg v_g) \wedge (\neg u_b \vee \neg v_b)$
3. For each vertex $u \in$ V, create the clause $u_r \vee u_g \vee u_b$
4. Output the formula F with the conjunction of all these clauses

Using this karp reduction, for the generated boolean formula F, the number of variables are $3 * |V|$ and the number of clauses are $3 * |E| + |V|$. Hence, as the size of the graph grows, the formula size also grows linearly. Also, note that F is in the CNF format as required. This way, we can handle the growth of the boolean formula linearly. Compared to other graph problems which might produce a large polynomial number of clauses and variables, this is ideal as we can push the limits of SAT solvers just by adding edges to the graph until we reach a point where the graph is no longer 3 colorable.

To understand the algorithm better, let's use the example of the Petersen graph whose DIMACS edge format file is present here - `petersen.graph`. This is a famous graph that is used to illustrate the graph 3-coloring problem. This graph has 10 vertices and 15 edges. In its pictorial form, it is represented using 3a.



(a) Petersen Graph Numbered



(b) Petersen Graph Colored

Fig. 3: Petersen Graphs

Using this, we generate a CNF boolean formula that has 55 clauses over 30 variables. This CNF formula file is then fed into a SAT solver which provides a valid 3 coloring for the graph. This can be seen in 3b. To check the CNF file, please visit here - `petersen.cnf` The output of the SAT solver can be converted into a valid coloring using the following algorithm.

Algorithm 2 SAT Solution to 3 Coloring

Input V, F's satisying assignment
Output Mapping $T : V \rightarrow \{red, green, blue\}$
for $u \in V$ **do**
  if u_r = True **then**
    Map u to red
  if u_g = True **then**
    Map u to green
  if u_b = True **then**
    Map u to blue
return the mapping T.

4.2 Spectral Coloring Algorithm

The method in the previous section is inherently an NP-complete approach as we are using 3-SAT which is also NP-complete. But they form an objective truth for coloring. It can objectively tell you whether a graph is 3 colorable or not and also provide one valid 3 coloring for colorable graphs. We use this approach as the absolute truth in our project. Even before [3] which showed that graph 3-coloring is NP-complete, several heuristic algorithms were found that compute the chromatic number of the graph. For our project, we use an algorithm is present in [5] which uses the eigenvectors corresponding to the negative eigenvalues of the adjacency matrix.

To understand this heuristic algorithm, consider the following two notions. For a 3-colorable graph, we define the notion of block regular tripartite in which the number of edges outgoing from each vertex from one partition to the other is the same. For instance, in the Petersen graph in 3a, the vertex 1, which is in the red partition, has 3 edges going to the blue partition and 0 edges to the green partition. The vertex 3, which is also in the red partition, has 2 edges going to the green partition and 1 edge to the blue partition. Hence, petersen graph is not block regular tripartite. Using the same logic, we can see that triangle graph is block regular tripartite.

If a 3-colorable graph is block regular tripartite, it's adjacency graph can be re-written as

$$A = \begin{bmatrix} 0 & A_{rg} & A_{rb} \\ A_{gr} & 0 & A_{gb} \\ A_{br} & A_{bg} & 0 \end{bmatrix}$$

where for every $i, j \in \{r, g, b\}$ and $i \neq j$, $A_{ij}\mathbf{1} = b_{ij}\mathbf{1}$ and b_{ij} is the number of edges from each vertex in partition i to partition j.

Consider the notion of eigenvector sign partitioning in which based on the signs of the eigenvector elements, with 0 taken as positive, we partition the vertices of the graph into different color partitions. For instance consider the eigenvectors for the triangle graph, $\lambda_1 = 2, \lambda_2 = -1$ in equation 1. If we convert w_2 and w_1 to its signs, we get $w_2 = \begin{bmatrix} - & + & + \end{bmatrix}^\top$ and $w_1 = \begin{bmatrix} + & + & + \end{bmatrix}^\top$. Then, both the first elements of w_2 and w_1 form a partition, and the second and the third elements form a partition. Thus we form two partitions of the vertex set, $\{\{1\}, \{2, 3\}\}$.

Using these two notions and theorem 6 in [5], we find whether the graph is 3-colorable or not. Note that in this approach we are not converting the resulting partitions to colors as we are just addressing the decision version of the graph 3-coloring problem. The pseudo-code for checking whether the 3-colorable graph is block regular tripartite, partitioning the eigenvector based on the signs, and the major approach in [5] are provided in algorithms 3, 4, and 5 respectively.

Algorithm 3 Check if Three Colorable Graph is Block Regular Tripartite

Input Adjacency matrix A, Coloring mapping T
Output True / False
for $u \in$ Red Partition **do**
  $b_{rg} \leftarrow |\{\{u, v\} | v \in$ Green Partition & $\{u, v\} \in E(G)\}|$
  $b_{rb} \leftarrow |\{\{u, v\} | v \in$ Blue Partition & $\{u, v\} \in E(G)\}|$
  if b_{rg} or b_{rb} doesn't match previous values **then**
    return False
for $u \in$ Green Partition **do**
  $b_{gr} \leftarrow |\{\{u, v\} | v \in$ Red Partition & $\{u, v\} \in E(G)\}|$
  $b_{gb} \leftarrow |\{\{u, v\} | v \in$ Blue Partition & $\{u, v\} \in E(G)\}|$
  if b_{gr} or b_{gb} doesn't match previous values **then**
    return False
for $u \in$ Blue Partition **do**
  $b_{br} \leftarrow |\{\{u, v\} | v \in$ Red Partition & $\{u, v\} \in E(G)\}|$
  $b_{bg} \leftarrow |\{\{u, v\} | v \in$ Green Partition & $\{u, v\} \in E(G)\}|$
  if b_{br} or b_{bg} doesn't match previous values **then**
    return False
return True.

One important distinction in our project and theorem 6 in [5] is that we take eigenvectors of all non-positive eigenvalues. This is essential as some graphs might be empty and in that case, the adjacency matrix is an empty matrix which is block regular tripartite. Hence, we need to consider this case as well.

5 Implementation

We wrote all the files required for the execution of our project using the Python language except for one Matlab file. As mentioned, we used PySAT and Z3 as our SAT solvers. Note that PySAT is a suite of

Algorithm 4 Two Sign Vectors to Partitioning

Input Sign vectors w_1, w_2
Output Element Partitions
if $w_1 = w_2$ **then**
  return $\{\{1, |w_1|\}\}$ ▷ Only one partition
for $a \in \{1, |w_1|\}$ **do**
  if $w_1(a) = w_2(a)$ **then**
    Add a to current partition.
  else
    Add current partition to partitions list.
    State new current partition with a.
return Partiton list

Algorithm 5 Theorem 6 Method

Input Graph G, Adjacency matrix A, Coloring mapping T
Output Chromatic number of G
if G is block regular tripartite **then**
  $A^{'} \leftarrow$ Permute A to make make it block regular.
  Find non-positive eigenvalues and corresponding eigenvectors of $A^{'}$.
  Convert these eigenvectors to sign vectors
  for $w_1, w_2 \in$ Eigenvectors **do**
    χ (G) = min(χ(G), |Partitions|)
  return χ (G)
else
  Find non-positive eigenvalues and corresponding eigenvectors of A.
  Convert these eigenvectors to sign vectors
  for $w_1, w_2 \in$ Eigenvectors **do**
    χ (G) = min(χ(G), no. of resulting partitions)
  return χ (G)

theory solvers each with its purpose. For our project, we used PySAT's *Glucose4* solver which is a CDCL-based algorithm mainly used for boolean clauses. For calculating the eigenvalues and eigenvectors, we used the scipy and numpy modules and re-used the information obtained from SAT solvers as the coloring mapping T.

5.1 Core Evaluation Files

KarpReduction.py This file implements the karp reduction algorithm provided in 1 in the function `reduce_to_sat`. It will initially parse the DIMACS edge format file and convert it into an edge list. Based on this, we will create the required boolean CNF formula and write it into `Formulae/<graph_name>.cnf` file. This file is in the DIMACS CNF format. This function will return the CNF file's path as output.

Z3Solver.py This file uses the Z3 API available for Python via `z3-solver` package to solve any CNF formula. We use the CNF file path returned by `reduce_to_sat` and use Z3's `from_file` API to directly convert the CNF file to boolean clauses. We also take the solver's timeout in seconds as input. If the formula is `unsat` or if Z3 is not able to find a satisfying assignment in the stipulated time, we return the time accumulated so far and `None` as output. Otherwise, we will return the satisfying assignment as an integer array following DIMACS notation. In some cases, Z3 doesn't return all the variables and the left-out variables are implicitly False. This file takes care of that case as well.

PySATSolver.py This is similar to `Z3Solver.py` except that we use PySAT's *Glucose4* instead. This file interacts with Python's `python-sat` package. The timeout feature for PySAT is a bit different compared to Z3. PySAT uses it's `solve_limited` API. The documentation states that this API will provide a complete assignment if *Glucose4* can find one in the stipulated time.

PrintValidColoring.py This file implements the algorithm 2 i.e, it takes the metadata and SAT solver's output and produces a valid coloring for the graph as a python dictionary.

evecs_partition.py This file implements the algorithm 4 which takes two sign eigenvectors. We encode sign vectors using 1(+) and 0(-).

Eigenvalues.py The file uses `numpy`'s `eigh` function which evaluates eigenvectors and eigenvalues for hermitian and symmetric matrices. It also extracts the eigenvectors for non-positive eigenvalues and creates sign vectors for them. We did not use Matlab for finding eigenvectors because of two reasons -

1. Because we have implemented the SAT solver approach in Python, it wouldn't be a fair comparison for both approaches. Moreover, we wrote the code for eigenvalue extraction in Matlab as well and we were seeing similar execution times for a lot of sparse suite graphs. The code for Matlab is present here - `final_project.m`.
2. We did not use `scipy`'s sparse eigenvalue evaluation(`eigsh`) as its answers will not be accurate and relatable to Matlab's output. But Matlab output is comparable to `numpy`'s `eigh` and both of them use Lanczo's iteration. For more information about this, please check this `stackoverflow answer`.

Gilbert_Method.py This file implements algorithms 3 and 5 and is the heart of the heuristic graph coloring approach.

5.2 Experimentation Files

SSMat.py This file uses the `ssgetpy`([4]) module to download sparse suite matrices one by one and maul them into various graphs. We maul the matrix by making any diagonal entry to 0 and any non-diagonal entry to 1. We also add the corresponding transpose element of the matrix to make the resulting matrix symmetric. This way, we form an adjacency matrix for a graph from which we extract the edge list, number of vertices, and number of edges.

SSTestDriver.py We use the resulting graph's edge list and metadata information obtained from `SSMat.py` to reduce it to a SAT formula, store it in `Formulae/` directory and then solve it using Z3 or PySAT. This file also uses a default timeout for the solver as 300 seconds. It will then write the results into a csv file.

SSEvaluation.py This file, similar to **SSMat.py** extracts and mauls sparse suite matrices and calls `Gilbert_Method.py` to get the graph coloring heuristic solution. We evaluate both 3-colorable and non-3-colorable graphs to get an idea of the chromatic number of the non-colorable graphs. We store the results in `GA84_Method_colorable.csv` and `GA84_Method.csv` respectively.

5.3 Auxiliary Data Handling Files

Various CSV file We write the results of various experiments into CSV files. For instance, the timing information for solving the sparse suite matrices using Z3 are written into `Z3_Statistics.csv` file.

Data_Statistics.py This file parses the result CSV file and prints out various statistics such as the minimum time taken, the average difference between two solver-based approaches, etc.

plots.py This file uses the result CSV files to generate plots for various approaches.

6 Experiments

Before proceeding to the details of our experiments and our final results, table 1 provides the details of the versions of all the software packages and libraries we used. All the testing was done in Ubuntu 21.10 on a Intel(R) Core(TM) i7-7500U CPU @ 2.70GHz quad core processor with 16 GB memory.

Package	Version
z3	0.2.0
z3-solver	4.8.16.0
pysat	3.0.1
python-sat	0.1.7.dev16
ssgetpy	1.0rc2
numpy	1.22.3
scipy	1.8.0

Table 1: Software Package Version Required for Our Project

6.1 SAT Solver Experimentation

A rough sketch of our experimentation process is given in figure 4.

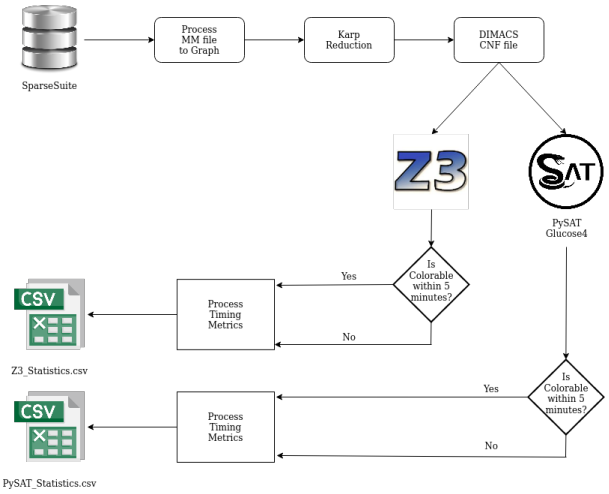


Fig. 4: Overview of Sparse Suite Experiments

We begin by `SSMat.py` which downloads, mauls and extracts the edge list information from a sparse suite matrix. This resulting edge list and metadata information is fed into `SSTestDriver.py` which uses `KarpReduction.py` to convert the sparse suite graph to a CNF formula and solves the formula using `Z3Solver.py` and `PySATSolver.py` using a 300-second timeout. If the solver is able to find a satisfying solution within 300 seconds, we will convert it to a coloring scheme and store the results in `Z3_Statistics.csv` and `PySAT_Statistics.csv` for Z3 and PySAT's *Glucose4* solvers respectively.

One of the main challenges faced with this experimentation is storage. As we have to download the sparse suite matrices to extract information from them, the local storage shoots up to more than 10 GiB to store these matrices. Later converting them to graphs and then to CNF files will take more than 30 GiB of local storage. Hence, we are sharing our project's data using google drive and GitHub. Moreover, as we are executing Z3 using the command line as well, we are storing the smt2 files generated by the solver to later use them. These smt2 files alone will take another 14 GiB. Hence, the data generated by these experiments will take $\sim$ 50 GiB of local storage.

Another challenge is that of memory exhaustion. Both PySAT's *Glucose4* and Z3 use conflict-driven approach and store information about conflicting clauses that exhausted the memory in our computers. Because of this, we were only able to compute 1622 out of the 2215 square matrices available in the sparse suite. If we had additional memory available, we could've tested a few more. We couldn't move our testing to CSIL VMs or COE Linux VMs as we do not have permission to install the Python packages required for our project. The entire experimentation process took roughly 3 days as we essentially have 10 minutes per graph's execution.

6.2 Sparse Suite Results

Based on the results collected in `Z3_Statistics.csv` and `PySAT_Statistics.csv`, we extracted a few metrics which are shown in table 2 and figure 5. These metrics are for the 1622 square matrices which

we were able to handle in our local machine. The third row in the table corresponds to the difference between time taken to solve the CNF formula for the two solvers. The maximum size of the formula we were able to handle was 1,162,308 clauses over 187,893 boolean variables.

	Average Time Taken(sec)	Median Time Taken(sec)	Minimum Time Taken(sec)
Z3	2.072	0.062	0.004
PySAT	1.313	0.0008	1.6E-5
abs(Z3 - PySAT)	0.759	0.06	0.004

Table 2: Metrics for Sparse Suite Evaluation

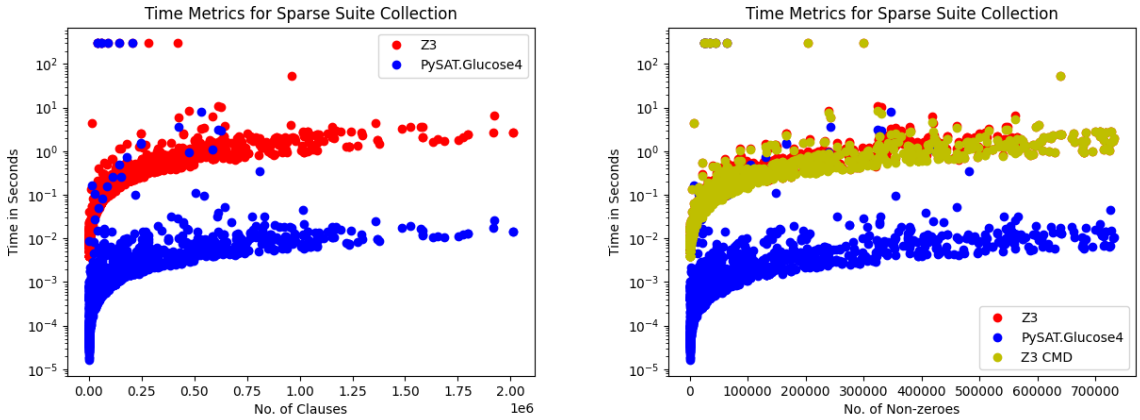


Fig. 5: Semilog Time Comparison for Z3 and PySAT's *Glucose4*

As most of the formulae were solved within a minute, we used a semilog plot in figure 5. The yellow plot on the right side of the figure is the timing information from executing Z3 via command line using `Z3_Sole.py`. As we can see, there is not much difference in Z3 execution via Python or command line. There were a total of 7 graphs out of 1622 which were not solvable by both the SAT solvers and we ran them from 3 times. But they remained unsolved. Hence, we are not providing any metrics for them.

As both table 2 and figure 5 show, PySAT's *Glucose4* performs vastly better compared to Z3. In fact, we saw on an average 98.3% drop in solving time when we used PySAT's *Glucose4* compared to Z3. The maximum drop in solving time is 99.6% and the median drop is 98.9%. Moreover, PySAT's *Glucose4* was able to extract 2 graph colorings while Z3 was not able to finish in 300 seconds for these graphs. There was no graph that Z3 was able to solve when PySAT's *Glucose4* timed out. From these inferences, we conclude that PySAT's *Glucose4* is better at solving boolean constraints compared to Z3.

6.3 Heuristic Graph Coloring

We start with `SSEvaluation.py` which downloads and mauls the sparse suite matrix and checks if it is satisfactory using `PySATSolver.py`(as this is the winner in the previous approach). We then subtract the SAT solving time from 600 seconds and feed the coloring mapping T, and the edge list information extracted from previous SAT solver's run to `Gilbert_Method.py`. This file then checks if the graph is block regular tripartite or not and finds the eigenvalues of the adjacency matrix and stores the timing results and chromatic number information in respective CSV files.

We faced similar challenges like in the previous approach. Our computer's memory was getting exhausted as we needed to deal with large number of vectors. This was worse than the SAT solving approach as we need to store eigenvectors and their sign vectors as well. In essence, we need to store $\Omega(|V|^2)$ vectors per graph. We thought of executing our implementation in CSIL VMs or COE Linux VMs but we

faced a similar hurdle to SAT approach as we are using `python-sat` package which we can't install in those VMs. However, we did limit the precision of eigenvectors and eigenvalues to 6 digits to save some memory. We planned to calculate only half of the total number of eigenvectors but we could not find a proof that all the negative eigenvalues will be half of the total amount of eigenvalues. Furthermore, we couldn't find an algorithm that directly provides the signs for eigenvectors as opposed to real values. Hence, we expected significant performance drops with this approach.

One major advantage in terms of the execution time was that we stored the information of all the sparse suite graphs from the SAT solving approach. This helped us in reducing the download time and time it takes to maul the matrix into a graph and then into an edge list. Out of the 1622 matrices that the SAT solving approach was able to handle, there were 500 3-colorable matrices. Out of the 500 matrices, only 49 are block regular tripartite. Out of the 49, only 23 complied with the heuristic-based approach. A majority of these 23 were empty graphs. There were a few graphs that did not produce desired results and the rest exceeded the time limit of 10 minutes(600 seconds).

The entire testing for colorable graphs approach took 3 days as there was a 10 minute time out and a few times our computer crashed. We also tested non-colorable graphs to the point that our computer was able to handle. We were hoping to find some interesting counter-examples or contradictions. We will summarize the results in the next subsection.

6.4 Heuristic Based Approach Results

Based on the results collected in `GA84_Method.csv` and `GA84_Method_colorable.csv`, we extracted a few metrics which are shown in table 3 and figure 6. The largest graph which was colorable that our implementation was able to handle is `AG-Monien/ukerbe1` with matrix ID 2422. It contains 5,981 vertices and 7,852 edges. The largest graph which was not colorable that our implementation was able to handle is `Grund/poli_large` with matrix ID 468. It has 15,575 vertices and 17,427 edges.

	# of Graphs Handled in						
	10 sec	30 sec	60 sec	120 sec	300 sec	450 sec	600 sec
Z3	1610	1612	1613	1613	1622	1622	1622
PySAT	1615	1615	1615	1615	1622	1622	1622
Heuristic(Colorable only)	180	254	268	278	286	295	300

Table 3: Time Boundaries for Sparse Suite Evaluation

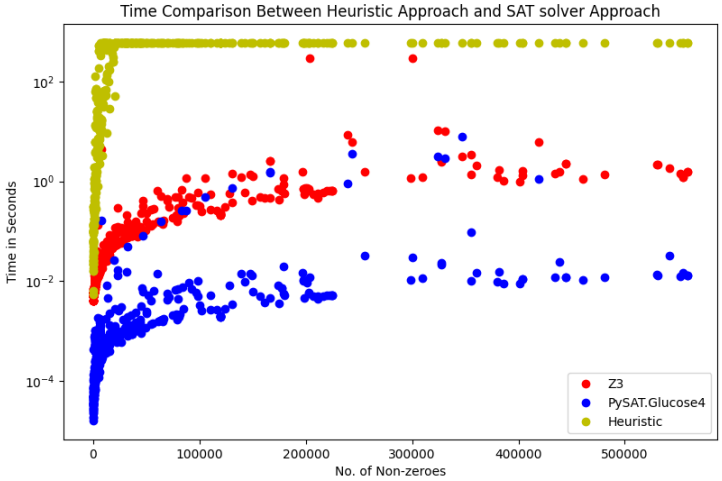


Fig. 6: Timing Comparison between Z3, PySAT and Heuristic Approach for Colorable Graphs

As there is a huge disparity between the execution time, the y-axis on figure 6 is plotted using `semilogy`. We can clearly see that PySAT's *Glucose4* is still the most efficient. The average time to check

if a graph is colorable using 3 colors in the heuristic approach is 255.16 sec which is roughly a 19000% increase. Similarly the median time also took at giant leap of 18.627 seconds compared to PySAT's *Glucose4* median of 0.0008 seconds.

Some interesting graphs we encountered in our experimentation are:

1. `Newman/dolphins`: This graph is not colorable using 3 colors according to SAT solvers which is the absolute truth in our experiments. But the heuristic-based approach says that it is colorable using 3 colors.
2. `Pajek/GD01_Acap`: Similar to `dolphins` matrix, this is also not colorable with 3 colors but the heuristic-based approach shows that it is colorable with 1 color. However, the graph is not empty. There are a lot of graphs such as this one in `GA84_Method.csv` file
3. `Gset/G11`: This is a 3-colorable, block regular tripartite graph which is refuting the heuristic-based approach. Moreover, it did not exceed 600 seconds as well. This might be due to tradeoffs in generating the eigenvalues and eigenvectors which produce errors in signing scheme.

7 Future Improvements

In our project, we evaluated two techniques (one using SAT solvers, and the other using a heuristic algorithm based on eigenvalue decomposition) that are used to solve the problem of graph vertex coloring. The programs we wrote for this project include the core structure required to solve graph vertex coloring. We provide a list of improvements that can be implemented within a reasonable amount of time.

1. **Comparing algorithms based on different Heuristics:** Currently, we only evaluated one heuristic algorithm which is based on Eigenvectors. But, there are some probabilistic algorithms which finds solutions for graph vertex coloring [6]. We can evaluate this technique too and compare the efficiency and accuracy of both algorithms.
2. **Solving coloring of larger Graphs:** As we did project on our own laptop, we couldn't solve graph coloring problems for graphs which are huge due to limitation of computation and time. If we use Google's GPU cloud services, we can automate the solving of graph coloring of any size.
3. **Generating Graphs:** In [6], the authors use a probabilistic generation that agree with their theory. We can also focus on algorithms that can be used to generate block regular tripartite graphs.

The following improvements are related to SAT solver-based approach:

1. **Automating Solver Choice:** We can also automate the suggestion of the solver which is best suitable according to the input problem using several heuristics.
2. **Parallelization:** One other thing we could also focus is on the parallelization of the execution to find clauses and solutions quicker.

This list by no means is exhaustive but a few things to consider. Having the right hardware, we can combine modern-day techniques like Machine Learning and apply some models to detect any hidden patterns or relations between graphs and the algorithm's runtime.

8 Conclusion

In this project, we solved the problem of graph 3 colorings using two methods. In one, we take the graph in DIMACS edge format and transform into a CNF formula using Karp reduction and feed it to two SAT solvers to find a solution. In the other approach, we used a heuristic algorithm based on eigenvalue decomposition and solved the 3 coloring problem.

For benchmarking our implementation, we used Texas A& M's sparse matrix suite to maul the matrices into adjacency matrices and used them as graphs. We compared the execution times for both approaches to find a reasonable way to solve the graph 3-coloring problem in minimal time. From our results, we conclude that using PySAT's *Glucose4* is currently the most efficient way to solve graph 3-coloring problem.

References

1. Timothy A. Davis and Yifan Hu. 2011. SuiteSparse Matrix Collection - ACM Transactions on Mathematical Software (TOMS) 38, 1 (2011), 1–25. https://sparse.tamu.edu
2. László Lovász. 1973. Coverings and colorings of hypergraphs. https://web.cs.elte.hu/ lovasz/scans/covercolor.pdf
3. Richard M. Karp. 1972. Reducibility Among Combinatorial Problems - Complexity of Computer Computations. New York: Plenum. pp. 85–103. https://link.springer.com/chapter/10.1007/978-1-4684-2001-2_9
4. Sudarshan Raghunathan. 2020. Search and download sparse matrices from the SuiteSparse Matrix Collection. https://github.com/drdarshan/ssgetpy
5. Aspvall, Bengt, and John R. Gilbert. "Graph coloring using eigenvalue decomposition." SIAM Journal on Algebraic Discrete Methods 5.4 (1984): 526-538. https://epubs.siam.org/doi/abs/10.1137/0605051
6. Alon, Noga, and Nabil Kahale. "A spectral technique for coloring random 3-colorable graphs." SIAM Journal on Computing 26.6 (1997): 1733-1748. https://epubs.siam.org/doi/abs/10.1137/S0097539794270248