CSCI 1515 Final Project: Zero Knowledge Proof for Graph 3-Coloring

Luke Choi[*]
Brown University
Providence, Rhode Island, USA
luke_choi@brown.edu

Brayden Goldstein-Gelb
Brown University
Providence, Rhode Island, USA
brayden_goldstein-gelb@brown.edu

1 OVERVIEW

We created a zero-knowledge-proof for graph 3-coloring. We created a system where a user, Alice, can send a graph to another user, Bob. Then, without revealing information about a specific coloring, Alice can prove to Bob that the graph is 3-colorable.

We created a terminal application with a server and a client. The server has access to several graphs and claims to know their corresponding colorings. The client can then connect to the server. The client tells the server which graph they would like to have proven for them.

2 PROTOCOL

The protocol is as follows:

- Alice begins with a graph, $G = (V, E)$ and a coloring, $\phi : V \to \{0, 1, 2\}$
- Alice randomly samples a permutation $\pi : \{0, 1, 2\} \to \{0, 1, 2\}$ and for each vertex, $v \in V$ a random bitstring $r_v \leftarrow \$\{0, 1\}^\lambda$
- For each vertex, $v \in V$ Alice will then commit

  $$c_v := Comm(\pi(\phi(v)); r_v) = H(r_v || \pi(\phi(v)))$$

  and send $\{c_v\}_{v \in V}$ to Bob.
- Bob can then choose a random edge $(u, v) \in E$ and send it to Alice as a challenge
- To meet the challenge, Alice sends back $\pi(\phi(u)), \pi(\phi(v))$ as well as r_u and r_v
- Bob can then verify that the coloring is valid by checking that $\pi(\phi(u)) \neq \pi(\phi(v))$. He can also ensure that the coloring corresponds to Alice's original commitment by checking that $H(r_u || \pi(\phi(u))) = c_u$ and $H(r_v || \pi(\phi(v))) = c_v$.

The probability of Alice always selecting a coloring in which $\pi(\phi(u)) \neq \pi(\phi(v))$ is very low and decreases in the number of trials (see section 4). Therefore for any fixed p, Bob can perform enough trials with Alice to ensure that the probability that Alice is cheating is less than p.

3 DESIGN CHOICES AND IMPLEMENTATION

The three main components of the codebase are `challenger.py`, `prover.py`, and the graph driver, `graph_driver.py`. The graph driver is where we factor out all logic related to generating the commit, verifying valid colorings, visualization, etc. The challenger and prover act as the client and server, respectively. We use TCP sockets to act as a mean of communication. Each program implements the described protocol in Section 2 directly, with serialization and deserialization handled through the `pickle` library. To hash, we used `hashlib` to use SHA256, and the `secrets` library to generate random bitstrings using the builtin `randbits`. We felt that this codebase structure was the most natural for a protocol of this type, and is consistent with what we have seen in past projects. The main (relevant) difference is the lack of a network driver, but this was handled directly with the `socket` library that implements everything for us.

In addition, we wanted the command line tool to be as user-friendly as possible. To do this, we used the `argparse` library to allow users to add both required and optional arguments.

`prover.py` takes in up to 4 arguments. `graph` and `honest` are required arguments that specify which graph the prover will be using and whether the prover will attempt to cheat the challenger by selecting a valid coloring or not. The current available graphs/colorings are K3 and the Petersen graph but this is easily extendable by adding more graphs to the `example_graphs.py` file. In addition, the user can also choose whether to enable `-show` to visualize the graph and its coloring as well as `-verbose`, which prints out additional information as the protocol is executed.

`challenger.py`, takes in a variable number of arguments. Just as in `prover.py`, the user must specify the graph they want to work with. Then, they can also choose to either specify `-ntrials`, which will execute the protocol n times on random endges, or they can specify `-u` and `-v` to choose a specific edge to run the protocol on. In either case, the challenger will then execute the chosen protocol and tell the user whether the graph's coloring could be verified.

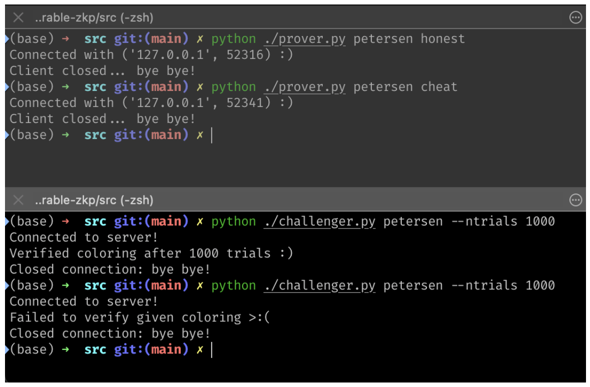
4 RESULTS

The results of this experiment were quite positive, as we got the results that we expected. We made sure to test for false positives as well, which we demonstrate in our presentation recording. Again, the ZKP is unable to leave the challenger 100% confident, but we know that confidence grows exponentially. As we demonstrate in Figure 1, a dishonest prover is unable to convince a challenger with 1000 challenges, so even though we can't guarantee that it safeguards against false positives (for any n trials), it's fairly efficient ot get the result with high probability. After this, it was mainly stress testing by varying the number of vertices and the number of trials, and each experiment was successful.

5 CHALLENGES

The main challenge of this project was getting used to the different libraries we were importing and having them work in sync. For example, having little networking background, we took a lot of time reading and catching up on networking basics and familiarizing ourselves with different protocols and error handling. Aside from this, the most difficult aspect from a technical point of view was

[*]Both authors contributed equally to this research.



```
..rable-zkp/src (-zsh)
(base) ➜  src git:(main) ✗ python ./prover.py petersen honest
Connected with ('127.0.0.1', 52316) :)
Client closed...  bye bye!
(base) ➜  src git:(main) ✗ python ./prover.py petersen cheat
Connected with ('127.0.0.1', 52341) :)
Client closed...  bye bye!
(base) ➜  src git:(main) ✗ |

..rable-zkp/src (-zsh)
(base) ➜  src git:(main) ✗ python ./challenger.py petersen --ntrials 1000
Connected to server!
Verified coloring after 1000 trials :)
Closed connection: bye bye!
(base) ➜  src git:(main) ✗ python ./challenger.py petersen --ntrials 1000
Connected to server!
Failed to verify given coloring >:(
Closed connection: bye bye!
(base) ➜  src git:(main) ✗ |
```

Figure 1: A screenshot of the terminal from the prover and challenger POV.

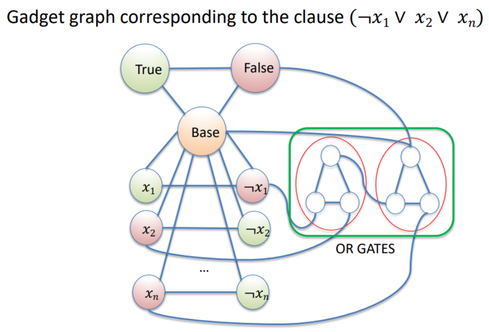

Gadget graph corresponding to the clause $(\neg x_1 \vee x_2 \vee x_n)$
True
False
Base
x_1
$\neg x_1$
x_2
$\neg x_2$
...
x_n
$\neg x_n$
OR GATES

Figure 2: Gadget graph for 3-SAT Problem

implementing the verification logic in the graph API. Other than these two points, the rest of the project went fairly smoothly.

6 FUTURE WORK

We were able to successfully implement our stretch goal for this project as well, which were to consider arbitrary n-colorings (rather than just 3-colorings). On top of this, we spent some time making the interface more user friendly and added a lot of options on the client side for being able to try out the ZKP.

If we had more time, we would have loved to get our other stretch goal to work, which was to add more visualization and support for 3-SAT. The connection here is that it is possible to get a polynomial time reduction from 3-SAT to 3-coloring using gadgets. The idea is that the clause can be represented as a graph, and by augmenting it with gadgets, it is possible to force a satisfying assignment to be equivalent to a valid 3-coloring of the resulting graph. An example of such a graph is below. The idea for future work, then, would be to automate the construction of such a gadget graph and then apply our already implemented ZKP to immediately get a ZKP for the 3SAT problem. This opens up a way to extend this to many other NP-complete problems, in particular those with "nice" reductions.

7 RELEVANT QUESTIONS

The main shortcoming of this algorithm is with the number of trials it takes to reach high confidence levels. As seen in class, the probability of not catching a cheating Prover is

$$\left(1-\frac{1}{|E|}\right)^n$$

where n is the number of trials and $|E|$ is the number of edges in the graph. Using $n = \lambda|E|$ gives

$$\left(1-\frac{1}{|E|}\right)^n \approx (1/e)^\lambda$$

In order to be confident with probability p that the prover is not cheating, then, this requires $\lambda \approx -\ln(1-p)$. Then we require $O(|E| \ln 1/(1-p))$ trials, or $O(|V|^2 \ln 1/(1-p))$ as $|E|$ is roughly $O(|V|)^2$. This quickly grows out of hand as $|V|$ grows large, especially since we need to apply the commitment scheme/random permutation generation for each trial. As a result, there are a few natural questions that come to mind:

(1) Is there a way to optimize the Graph 3-Coloring ZKP in its current form?
(2) Are there other ZKPs with better probabilistic guarantees than the current ZKP?

Of course, the solvability of these questions and their hardness are likely to be connected with the fact that Graph 3-Coloring is NP-hard. We can then consider the same questions in a quantum computation setting as well.

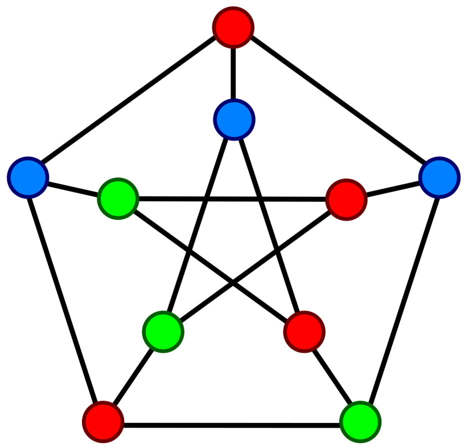


Figure 3: A graph theorist's favorite example of a 3-colorable graph: the Petersen graph.

8 BIBLIOGRAPHY

- Petersen graph image: https://en.wikipedia.org/wiki/Graph_coloring#/media/File:Petersen_graph_3-coloring.svg
- Protocol: CSCI 1515 class slides
- Gadget graph image: CSCI 1570 slides

9 ACKNOWLEDGEMENTS

We'd like to thank our peer Tyler Forman for reviewing this paper and giving us feedback on our practice runs for the presentation. Also, special thanks to all the TAs, our mentor TA Michael for the feedback and advice, and of course Peihan for the great lectures and making this class so much fun :)