Zero-Knowledge Interactive Proof Systems for all NP
(Construction for Graph 3-Colorability, G3C)

Content

Abstract:
In this project, we study the result presented in [2] that "*all languages in NP have zero-knowledge interactive proof systems".* In particular, we have studied a general method for constructing ZKP for the well-known NP-complete language, *Graph 3-Colorability (G3C)* which is the basis of the result for all NP. We describe the construction process, provide a sketch implementation (pseudo-code) and analyze the general results of the zero-knowledge proof protocol for G3C.

1. Zero-Knowledge Proof:

A zero-knowledge proof (ZKP) is a proof of true statement which reveals nothing more than its validity. The word "proof" here is not used in the traditional mathematical sense (eg., it is not as we say- prove the Pythagorean theorem). Rather, a "proof" or "proof system", is a **randomized protocol** by which one party (called prover) wants to convince another party (called verifier) that a given statement is true. The notion of zero-knowledge proof seems very strange at first, so let's start with a couple of motivating discussion to illustrate its practical role and see its feasibility.

1.1. Motivation

To illustrate its role, intuitively, assume A(lice) claims that she can predict next week's lottery winning numbers and wants to convince B(ob) of this fact. However A doesn't want B to use this information. In the physical world, A can write the numbers on a piece of paper, put it in a safety box, lock it and give it to B to keep it. Then, on the next week, A will give B the key so that B can open and verify if A was right or not. Observe that if A was right, B will be convinced certainly since he gets the key to open the box and verify. On the other hand, if A was to cheat, B could not be cheated. Why? Because the locked box was with him and he gets the key to open and verify the numbers. To make it explicit, A (the "prover") is capable of convincing a true statement to (the "verifier") B. Equivalently, B is also capable of rejecting a false statement (i,e., B will not be convinced erroneously if A was to cheat him (was dishonest). Zero-knowledge proofs are proofs of this kind which are used as tools (sub-protocols) inside multiparty cryptographic protocols to give us such a guarantee in the digital world.

To motivate and illustrate the notion of zero-knowledge proofs formally, in a greater applied context, consider a scenario in which a party **A**, upon receiving an encrypted message from **B**, wants to send to **C** the least significant bit of the message. Suppose the messages are stored in a public storage media say Gmail. What will be the best solution? Let's propose **A,** to just send only the intended bit of the message, but here there is no way for **C** to believe A and accept the bit message. On the other hand, if **A** provides her secret key along with the message, the A will lose its privacy and **C** could read the entire message which is beyond what was required. The best solution is rather for A to present the bit, and prove that this message is indeed the encrypted one without revealing anything more about its encryption key (or the other encrypted messages). This notion of a proof which reveals nothing more than the validity of the assertion is what we call a zero-knowledge proof.

We remark that the problem in the foregoing proposed solution, namely *for "A" to prove that the message is indeed from the encrypted one…* is of the NP-type problems (i.e., it is a decision problem). It is a well-known fact that proofs of NP problems are efficiently verifiable if we have a true statement of membership in a language. Therefore, existence of zero-knowledge proofs for all NP-problems should imply that the foregoing statement can be proved without revealing anything beyond its validity. This motivating problem leads us further to the notion of zero-knowledge proof systems and to our main discussion.

In this project, we study how zero-knowledge proofs work for all NP problems. In particular, we study a general method presented in [1] (and discussed in detail in [2] from page 244) for constructing Zero-Knowledge Interactive Proof for the NP-complete *Graph 3-Colorability*,

denoted by G3C. Since much of zero-knowledge proof, specifically the result we are interested in, is based on the notion of computational difficulty, we should start from some basic definitions, conventions and results in computational complexity theory and of course we use concepts from probability theory and simulation (real/ideal simulation paradigm) and information theory (knowledge vs information) viewpoints.

1.2. Computability and Computational complexity

Intuitively, a problem is computable if it is possible to specify a sequence of instructions (steps) which will result in the completion of the task when executed by some machine. Such a set of instructions is called an *effective procedure*, or *algorithm*, for the task. Formally, in computability and computational complexity, a mathematical problem is *computable* if it can be solved in principle by a computing device. We call such a problem is equivalently said to be solvable, decidable, or recursive. **Turing machine** is a well-know abstract machine which is a mathematical model of computation that defines such a computing device. Turing machines are imagined to be a simple computer that reads and writes symbols one at a time on an endless tape by strictly following a set of rules. Here, as will discuss later, we are especially interested with the interactive versions of these machines.

In complexity theory, *decision problems* are computation problems which are arbitrarily yes-or-no questions on an infinite set of inputs which can be natural numbers or strings over some other set of symbols. Traditionally, a decision problem is equivalently defined as: the set of possible inputs together with the set of inputs for which the problem returns yes.

As a result, in complexity theory decision problems are formally defined as (formal) *languages*. That is, a decision problem is a language for which a decision procedure (an algorithm that asks YES/NO questions) produces the answer “yes”.

Generally speaking, a **formal language** is a set of strings of symbols together with a set of rules that are specific to it. Formally it is defined as:

- A formal language L over an alphabet Σ is a subset of Σ^*(the set of all possible strings (words) over the alphabet Σ). That is L is a set of words over the alphabet Σ. Sometimes the sets of words can be regarded as expressions. Besides, rules and constraints could be formulated for the creation of well-formed expressions. Example, the subset of strings for which the problem returns "yes" is a formal language.

1.3. P, NP, and NP-Completeness

Here, we put and briefly discuss the formal definitions of the complexity classes P, NP, and the concept of NP-completeness as in [1].

- **Definition 1.1** (**Complexity Class P**): A language L is solvable in (deterministic) **polynomial time** if there exists a deterministic Turing machine M and a polynomial $p(\cdot)$ such that
    - on input a string x, machine M halts after at most $p(|x|)$ steps, and
    - $M(x) = 1$ if and only if $x \in L$.

Thus, P is the class of languages that can be solved in (deterministic) polynomial time.

That is, P contains all decision problems that can be solved by a deterministic Turing machine using a polynomial amount of computation time, or polynomial time. Problems like, finding the **greatest common divisor** (gcd) and **primality test** are found to be in P.

Whereas, the complexity class NP is defined as the class of languages that cannot be solved in a (deterministic) polynomial-time. Equivalently, NP is defined as the class of languages that can be solvable by a non-deterministic polynomial-time Turing machine. More formally, NP is formalized as follows:

- **Definition 1.2** (**Complexity Class NP**): A language L is in NP if there exists a Boolean relation $R_L \subseteq \{0, 1\}^n \times \{0, 1\}^n$ and a polynomial $p(\cdot)$ such that R_L can be recognized in (deterministic) polynomial time, and $x \in L$ if and only if there exists a y such that $|y| \leq p(|x|)$ and $(x, y) \in R_L$. Such a y is called a **witness for membership of x** $\in$ L.

Thus, NP consists of the set of languages for which there exist short proofs of membership that can be efficiently verified.

In fact, this later statement, the *verifier-based definition,* is found to be more intuitive and practical in many applications, as in the result here with zero-knowledge proofs for all NP. To make it explicit, intuitively and of course equivalently, NP class is defined as the set of all decision problems for which the instances where the answer is "yes" have efficiently verifiable proofs (i.e., proofs verifiable by deterministic polynomial time computations).

It is clear that the complexity class P is contained in NP, but NP contains many important problems, the hardest of which are called NP-complete problems, whose solutions are sufficient to deal with any other NP problem in polynomial time. It is widely believed that $P \neq NP$, but it is the most important open problem in complexity theory. The problem, **the P versus NP ("P=NP")** problem, asks whether polynomial time algorithms actually exist for solving **NP-complete**, and so all NP problems. Here is the formal definition of NP-completeness.

- **Definition 1.3** (**NP-Completeness**): A language is NP-complete if it is in NP and every language in NP is polynomially reducible to it. A language L is **polynomially reducible** to a language G if there exists a polynomial-time computable function *t* such that x □ L if and only if *t* (x) $\in$ G.

Thus, if every L in NP is polynomially reducible to G in NP, G is called NP-complete.

Quadratic residuosity and **Graph Colorability** problems are among the languages known to be in NP. Moreover, *Graph colorability* is known to be NP-complete and so Graph 3-colorability, denoted by G3C, is a typical NP-complete problem in graph colorablility.

1.4. The Notion of a Proof: Mathematical Proof

Classically, unlike in other areas of human activities, mathematical proofs are static object (i.e. they are written sequences of self evident statements or derived from previous such statements). Besides, they are regarded at least as fundamental as the facts (i.e., theorems). In other real-life situations, however, proofs have a dynamic (interactive) nature. For zero-knowledge proofs, a dynamic interpretation is found to be more appropriate [1] than their traditional interpretation as static objects for:

- They are used as tools (i.e., sub-protocols) inside multiparty cryptographic protocols
- The powerful notion of their definition; robustness to convince without revealing any knowledge except the validity of a statement.

A proof system is a two party protocol, which involves – the prover which produce a proof and the verifier which verifies the validity of the proof. The two fundamental properties of a proof system are:

- **Soundness (validity)** – is for the verifier not to be cheated by false statements.
- **Completeness** – is the ability of the prover to convince the verifier of true statements.

These assure that the verifier is convinced if and only if the prover indeed has a true statement.

1.5. Gaining knowledge

We say that zero-knowledge proofs as proofs by which the verifier gains "no knowledge" beyond the validity of the assertion. From information theory viewpoint knowledge is different from information. Knowledge is related to computational ability and it is what we study or gain about public objects. Whereas, information is mainly related to partially know objects [1][3].

Remember the cases in the motivation section; the verifiers in both cases will actually gain pieces of information from the verification process, but no knowledge. That is, the one in the first case (lottery winning numbers prediction scenario), the verifier will actually know the numbers after the box is opened; however, it will not gain any knowledge that enables to predict (compute) future lottery winning numbers. Similarly, C in the second case (assuming the solution is done) could not learn the key or anything about the other encrypted messages of A other than confirming the validity of the intended bit message. Therefore, knowledge is something related to computation capability but not information and so we say, in an interaction, that Bob has gained no knowledge from Alice if whatever Bob can efficiently compute after interacting with Alice can also efficiently compute by himself. That is, Bob gains knowledge from the interaction only if he receives a result of a computation that is infeasible for him.

In computational complexity, the concept of measuring the amount of additional (unnecessary) knowledge contained in a proof system is introduced in [3] and is known to be ***knowledge complexity of a proof system***. It was introduced along with the notion of ***interactive proofs***, which were also studied independently by[4]. In cryptography, we are interested to proof a fact (say for example ones password) at the same time we need to keep it private. However, in doing so we were not aware of the measuring the extra knowledge in the proof we provide. Amazingly, since 1980s, we come to know the computational complexity measure of the additional knowledge in a proof system. This is made possible with the introduction of (probabilistic) interactive proofs. Moreover, the notion of zero-knowledge proof is found to be a paradigm in multiparty cryptographic protocols. That is, its cryptographic role is far from specific.

1.6. Interactive Proof Systems

An interactive proof system refers to the explicit computational tasks of *producing* a proof *(*by the prover) and *verifying* the validity of a proof (by the verifier) by interaction. The interaction between the two parties with a common input (the fact to be proved) is modeled by **a pair of interactive Turing machines.** As introduced earlier, interactive Turing machines are the interactive versions of Turing machines which are used to analyze the complexity (computability) of a computational problem (or an algorithm). In our discussion, we consider the following formal definition of an interactive machine as in [1].

- **Definition 1.4 (An Interactive Machine):** An **interactive Turing machine** (ITM) is a (deterministic) multi-tape Turing machine. The tapes are a read-only input tape, a read-only random tape, a read-write work tape, a write-only output tape, a pair of communication tapes, and a read-and-write switch tape consisting of a single cell. One communication tape is read-only, and the other is write-only.

Thus, an interactive Turing machine (ITM) is a six-tape deterministic Turing machine with the above mentioned tapes. The common input string on the input tape is called the *input*. Conventionally, the *random tape* is defined to contain infinite strings which are outcomes of an infinite sequence of unbiased coin tosses. The string on the *output tape* when the machine halts is called the *output*. The contents of the write-only communication tape are thought as *messages sent* by the machine; and that of the read-only communication tape are thought as *messages received* by the machine.

An interactive pair of Turing machines is a pair of ITMs that ***share their communication tapes*** so that the read-only communication tape of the first machine coincides with the write-only communication tape of the second machine, and vice versa. The computation of such a pair consists of alternating sequences of computing steps taken by each machine. The alternation occurs when the active machine enters a special idle state. At this time, the other machine passes from idle to the active state. The string written on the communication tape during a single non alternating sequence of steps is called the message sent by the active ITM to the idle one.

Thus, the interaction between the prover and the verifier in an interactive proof system is modeled by a pair an interactive Turing machines combined (linked) together such that, the messages sent by the Prover are received by the verifier which shares its communication tapes (i.e., the read-only communication tape of the Prover coincides with the write-only tape of the Verifier and vice versa). Obviously, these linked machine share the common input tape. However, we note that these linked tapes are (i.e., the random tape, the work tape, and the output tape) are distinct. Therefore, an interactive proof system is formally defined as follows (with slight modification to be more intuitive) [1] (page 214):

- **Definition 1.5 (Interactive Proof System):**
  An interactive proof system for a language L is a pair of ITMs, (*P, V*) such that V is polynomial-time and the fundamental properties of a proof system, namely, **completeness** (of the Prover (P)) and **soundness** (of the Verifier (V)) are satisfied.

Intuitively, an interactive proof system of a statement is require that
- It should be possible to "prove" a true statement, *completeness*
- It should not be possible to "prove" a false statement, *soundness. and*
- Interacting with the "proof" should be efficient, in the sense that regardless of how much time it takes to come up with, its correctness should be efficiently verified in polynomial-time.

Interactive proofs are probabilistic in nature. The zero-knowledge proofs in which we are particularly interested in are interactive proof systems which are called **zero-knowledge interactive proof systems.**

1.6.1. Zero-Knowledge Interactive Proof systems

Briefly, zero-knowledge interactive proofs are special interactive proofs, in the sense that in the interaction process the prover yields nothing (zero knowledge) but the validity a true statement. In other words, for all practical purposes, whatever the verifier could efficiently compute after interacting with a zero-knowledge prover is what it could efficiently compute by just believing that the assertion claimed is indeed valid. Thus, zero-knowledge is the property of the prover: Its robustness against attempts of the verifier to extract knowledge via interaction.

2. Zero-Knowledge Interactive Proof Systems for all NP

The basis of the result that *all languages in NP have Zero-knowledge interactive proofs [2]* is the specific but general result of zero-knowledge interactive proof system of G3C.

For this reason, we should start with discussing a general for constructing zero-knowledge proof for the graph 3-colorability protocol, G3C, and then using the generality of the method and the fact that G3C is NP-complete we could proceed to the general result for all NP. Of course, in this project, we limit ourselves to give only a theoretical justification of the general result as a summary.

In general, construction of zero-knowledge proof systems mainly uses one of the fundamental cryptographic primitive called **commitment schemes**. The construction of zero-knowledge proof protocol for G3C essentially uses these primitives. So, before going to the construction of the protocol we need to understand the notion of commitment schemes.

2.1. Commitment Schemes

A commitment schemes are an efficient two-phase two-party protocol. They are used to enable a party, the sender, to commit (*bind*) itself to a value while keeping it secret (*hide*)

from the other party, the receiver, until it is opened later. This is to guarantee that revealing the commitment (committed value) can yield only the value determined in the committing phase and so they are the analogues of non-transparent sealed envelopes in the digital world.

Explicitly the two phases of a commitment scheme are known as ***commit phase*** (the first phase) and ***reveal phase*** (the second phase). The commit phase is required to yield no knowledge to the receiver (at least no knowledge of the commitment (sender's value)), at the same time it is also required to bind (force) the sender to a unique value so that the receiver will only accept this value in the reveal phase. The (perfectly) binding property holds for all-powerful sender, while the (computational) hiding property is only guaranteed with respect to a polynomial-time bounded receiver. Note that these contradicting hiding and binding requirements are satisfied by computational assumptions (i.e., assuming the existence of secure encryption functions).

Construction

Commitment schemes are constructed assuming secure (one-way) encryption functions. With this assumption, any one-way function or one-way permutation with its hard-core predicate is used to construct commitment schemes. The motivation here is that every one-way function has a hard-core predicate or it could be modified trivially to have this property [1] (page 112). However, as the authors stressed to remark on, public-key encryption schemes need not be used for some properties that are claimed not required of commitment schemes [1] (page 246). Intuitively, as we could see that some of the failures are due to the time consuming nature of public-key cryptosystems.

In this project, however, letting the details with the practical implementation be beyond our scope, we could (theoretically) illustrate the construction of commitment schemes using any public-key encryption scheme say RSA with further assumptions; namely, the set of legitimate public-keys should be efficiently recognizable, and an encryption relative to legitimate public keys should have a unique decryption. *(Which actually are the properties claimed not to be satisfied by public key encryption schemes [1] (page 246)).*

General construction: let f(x) is a one-way function and b(x) its hard-core predicate, then a bit commitment scheme is constructed as follows. [1, p 247]

- **Commit phase:** To commit to value $v \in \{0, 1\}$ (using security parameter n), the sender uniformly selects $s \in \{0, 1\}^n$ and sends the pair $(f(s), b(s) \oplus v)$ to the receiver.
- **Reveal phase:** In the reveal phase, the sender reveals the bit v and the string s used in the commit phase. The receiver accepts the value v if $f(s) = \alpha$ and $b(s) \oplus v = \sigma$, where (α, σ) is the receiver's view of the commit phase.

To make it explicit the receiver, getting the revealed values (v, s) in addition to the commitments (α, σ), could verify and accept only if the revealed values are indeed the ones used by the sender in the commit phase. This is possible because the verifier can produce the commitment values (f and b are common) from the revealed values and check if these two values match. So, indeed, the receiver accepts the value v if $f(s) = \alpha$ and $b(s) \oplus v = \sigma$.

For simplicity, without lose of generality, we could consider public-key encryption scheme C (with the assumptions noted above) and construct the scheme as follows:

1. **Commit phase:** To commit to value v Є $\{0, 1\}^n$ (using security parameter n), the sender uniformly selects s Є $\{0, 1\}^n$ and sends the pair C(s, v) = (α, σ) to the receiver.
2. **Reveal phase:** In the reveal phase, the sender reveals the bit v and the string s used in the commit phase. The receiver accepts the value v if C(s, v) = (α, σ).

2.2. Zero-Knowledge Interactive Proof for G3C

In graph theory, graph coloring (in its simplest form) is a way of coloring (assigning colors to) the vertices of a graph such that no two adjacent vertices (vertices sharing the same edge) have the same color. Formally, a simple finite graph is defined to be a pair (V, E), where V is a finite set and E is a set of 2-subsets of V; that is, E ⊆ {e ⊆ V : |e| = 2}. The elements of V are called **vertices**, and the elements of E are called **edges**. Although each edge is an unordered pair of two elements u, v in V, we use the ordered-pair notation (u, v) Є E rather than the notation {u, v} Є E. For e = (u, v) Є E, we say that u and v are the endpoints of e and that u is adjacent to v.

Let G = (V, E) be a graph. We say that G Є G3C (G is 3-colorable) if there exists a mapping φ : V → {1, 2, 3} such that φ(u) != φ(v) for every (u, v) Є E. The mapping φ is called a coloring of G.

The language **Graph 3-colorability**, denoted by G3C, is a problem of determining whether a graph G is 3-colorable or not. The core idea underlying the interactive zero-knowledge proof system for G3C is to reduce (break) the proof of the *main claim* that *a graph is 3-colorable* into polynomially many *pieces (vertices)* arranged in *edges (pair of pieces)* so that each edge constitutes a claim that *an edge is colorable* so that each edge by itself will yield no knowledge while all the edges together guarantee the validity of the main claim.

Assume a common input, the graph G = (V, E) which is 3-colorable for both parties. To prove that the graph G = (V, E) is 3-colorable the prover generates a typical proper 3-coloring the graph say φ : V → {1, 2, 3} and let n = |V| and m = |E|. Note that the color of each single vertex constitutes a piece of information concerning the 3-coloring and each edge (i.e., each pair (φ(u), φ(v)), where (u, v) Є E, constitutes a single claim of the main claim that G is 3-colorable). A single edge being merely a random pair of distinct elements (colors) in {1, 2, 3} will yield no knowledge. But, if all the edges are found properly colored, then it is guaranteed that graph is 3-colorable. On the other hand, graphs that are not 3-colorable must contain at **least one bad template** and hence will be rejected with noticeable probability. To visualize the interactions involved in proof systems here is the abstract interactive model of the protocol.

Interactive abstract model of G3C

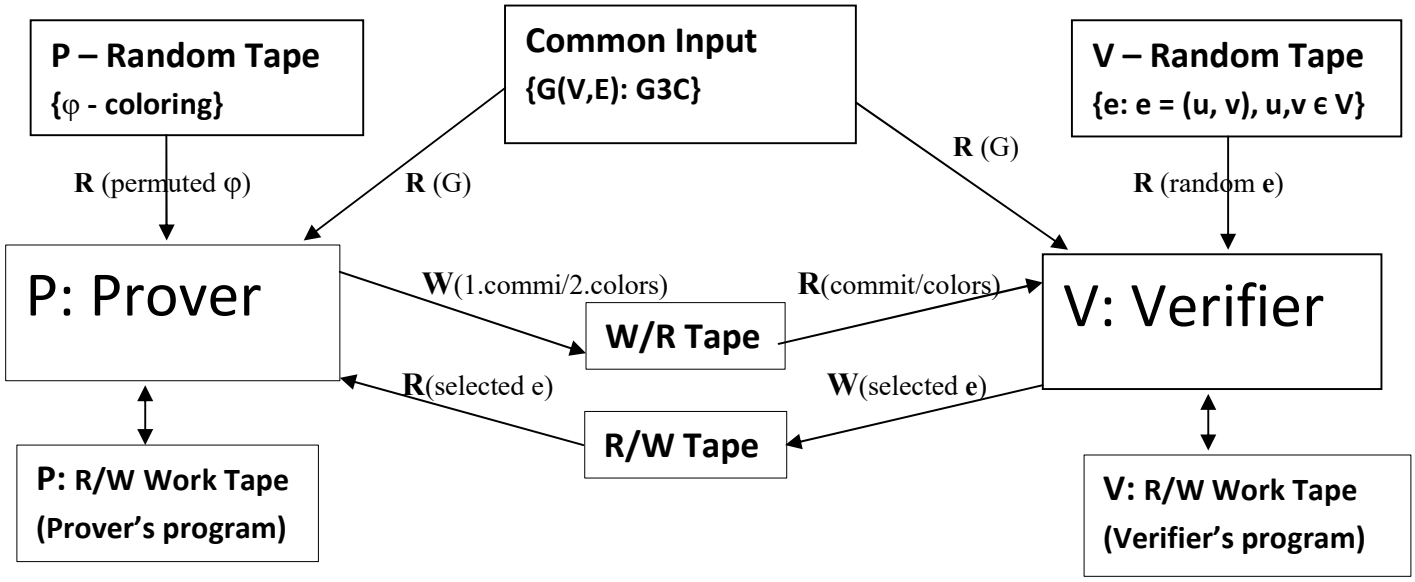

Figure 1, Interactive G3C protocol model: ←→ denotes a read/write, R a read-only, W a write-only head.

Roughly, in this model, the prover first commits to a randomly permutated coloring of the graph and then writes (sends) it to its write-only communication tape which is also a read-only communication tape of the verifier and so the verifier reads (receives) the committed values. The verifier, in turn, writes (sends) a randomly selected edge of the graph to its write-only communication tape as a challenge to see if the edge is properly colored. Finally, the prover reading (receiving) the challenge from its read-only communication tape sends the colors of the vertices of the edge so that the verifier can see and decide to accept and continue with other random selection of edges or else reject and stop if it gets bad colored edge. In our sketch implementation we have used this model.

Now, we could understand the abstract implementation of the protocol below and we provided a detailed sketch implementation (pseudo-code) of the protocol in following section.

Common Input: A graph G (V, E), and let n = |V| and m = |E|
Let f is a public-key encryption which is a common function; and
Assuming the prover has a secret coloring Ψ of G, a random permutation T and the three colors are {0, 1, 2}.

1. **(Prover) P-1: commit phase:** (assume) Prover produce permutation a new coloring Φ using the permutation T. Then P relabels (renames) the permuted 3-coloring Φ, encrypts it, and sends the encryption to the verifier. Specifically, the prover computes C[v] = f(Φ(v), v) (for every v ϵ V), and sends the sequence $C[v_1], \ldots, C[v_n]$F. to the verifier.
2. **(Verifier) V-**1: The verifier chooses at random an edge e ϵ E and sends it to the prover.
3. **(Prover) P-2:** If e = (u, v), e ϵ E, then the prover reveals the coloring of u and v. That is, the prover sends (Φ(u), u) and (Φ(v), v) to the verifier. If e ≠ E, then the prover resets e to an arbitrary fixed edge.

4. **(Verifier) V-2:** The verifier verifies the revealed values received in (step P-2). It computes the commitment using the common f and the respective public-keys say k[u] and k[v] and check if C[u] = f(Φ(u), C[v] = f(Φ(v), v) and if the edge is properly colored that is Φ(u) # Φ(v), with Φ(u), Φ(v) ∈ {0, 1, 2}. The verifier rejects and stops if either condition does not hold. Otherwise, the verifier continues to the next phase.

Assume this protocol is executed $\mathbf{m^2}$ times. If the graph is not 3-colorable and the verifier follows the protocol then, no matter how the prover plays, at each round the verifier will reject with probability at least 1/m. Whereas, the probability that the verifier will accept without detecting that bad colored edge is bounded above by $(1- m^{-1})^{m\hat{}2}$. A general analysis is given in the discussion section below.

2.3. Implementation ketch (Pseudo-code)

Note that, we are considering public-key encryption for the commitment scheme (just for theoretical purpose). As we have noted above in the real implementation we need to use one-functions or one-way permutations along with their respective hard-core predicates instead. So, in our case we can think of the ***encryption and decryption modules*** of the public-key encryption scheme (say RSA) as the prover's program in the commit phase and the verifier's program in the reveal phase respectively. Moreover, we also note that the prover commits to each vertex color independently (i.e., each vertex color is encrypted using distinct public keys and so the respective public keys are part of the commitment, so that they are used for verification by the verifier in the reveal phase).

With this consideration, the detail pseudo-code is included at the end of the document as an appendix.

2.4. Discussion: Results and Analysis

Consider a graph with m number of edges. The probability, at the first phase, that a dishonest prover could succeed cheating a verifier is (m-1)/m which seems bit higher than the probability it be caught (which is 1/m). But, the good thing is, this scenario will automatically be reversed exponentially when the number of verifications get larger. Consider the verifier repeats $k = m$ times to verify the assertion claimed by the prover. Then the probability that the prover could succeed cheating will be ((m-1)/m)^k which will be negligible.

The general could be shown as follows:

Verification phase (k)	**Probability (Prover could cheat)**
1	$\leq 15/16$
..	..
16	$\leq (15/16)^{16} = 0.35607 < 0.5$
..	..
100	$\leq (0.5)^{100} = 7.889 * 10^{-31}$
..	..
1000	$\leq (0.5)^{1000} = 9.332 * 10^{-302}$
..	..
k	$\leq (m\text{-}1/m)^k = (m-1)^k(m^{-k})$

This general analysis implies that, while introducing the concept of probability to the proof we are also enjoying the very notion of classical proofs. Specifically, a dishonest prover is much less likely to cheat a verifier while an honest prover is capable of convincing a verifier without revealing any knowledge beyond the validity the assertion with very high probability, which tends to $1 - m^{-k} \approx 1$, for a large k, the security parameter. On the other hand, we observe that if the graph is not 3-colorable and the verifier follows the protocol then, no matter how the prover plays, at each round the verifier will reject with probability at least 1/m.

2.5. Conclusion

Using the generality of the method we described and NP-completeness of the language G3C we could (at least theoretically) say that *all languages in NP have zero-knowledge interactive proofs systems*. That is, using the concept of NP-completeness, any language in NP can be reduced to G3C applying a reduction method called *Cook reduction* [1] to obtain an instance of 3-coloring. Of course, it has some subtle parts which are really beyond the scope our study here. But, the point here is that, using the NP-Completeness of graph 3-colorability, we have at least a theoretical justification that every NP-language has zero-knowledge interactive proofs assuming secure encryption functions.

- ✓ This result is claimed to be the first "*positive*" use of NP-completeness in the sense that rather than using it to show something cannot be done it is used to show something positive; the existence of zero-knowledge proofs for all NP languages [2].

When we come to the applications of zero-knowledge proofs, one can obviously think of authentication systems where one party wants to prove its identity to a second party via some secret information (say a password) which clearly doesn't want the second party to learn anything about it. Moreover, the generality of the result we discussed here implies many applications that are very far from specific. The concept of zero-knowledge proofs is found to be the underlying paradigm in the construction of multiparty cryptographic protocols [1]. That is, the construction of zero-knowledge proofs for all NP provides a tool for forcing ("bind and hide") parties to properly execute any given protocol. Specifically, almost all problems one may need to prove in practice can be taken as claims concerning membership in languages in NP.

Reference:

1. O. Goldreich. *Foundations of Cryptography: Volume 1 – Basic Tools.* Cambridge University Press, 2001.
2. O. Goldreich, S. Micali and A. Wigderson, *Proofs that yield nothing but their validity or all languages in NP have zero-knowledge proof systems,* Journal of the ACM, Vol. 38, No. 1 (1991) pp. 691–729.
3. S. Goldwasser, S. Micali, and C. Rackoff. *The Knowledge Complexity of Interactive Proof Systems.* SIAM Journal on Computing, Vol. 18, pages 186–208, 1989. (preliminary 1985)
4. L. Babai. M. Shlomo. *Arthur-Merlin Games: a randomized proof system, and a Hierarchy of Complexity Classes. 17th ACM Symposium on the Theory of Computing, pages 421–429, 1988, (preliminary 1985)*

Appendix A: Implementation ketch (Pseudo-code) of:
Zero-knowledge proof of Graph 3-Colorability, ProtocolZKPG3C

Public objects:

Some of these public objects are (sample) common inputs or (commitments) committed values of the sender (prover) which are used as inputs for the receiver (verifier) for verification.

Object Edge {
    Vertex u;
    Vertex v;
}

Let a sample Graph G be:
V = {1, 2, 3, 4};
E = {e = (u, v)}, where u, v Є V,
n = |V| and m = |E|;
G = (E, V);
(Common input) sample Graph G = {(1, 2), (2, 3), (2, 4), (3, 4)}

(Prover commitments)
Commitment *coloring Cp* **= {1:0, 2:0, 3:0, 4:0}** *//**ephemeral** encoded coloring and labeling at each phase*
Commitment *public-keys **K1, K2, K3 … Kn** //**ephemeral** public keys at each phase*

Prover program:

(Sample secrete) coloring *S* = {1:0, 2:1, 3:2, 4:0, 5:1, 6:0, … n} *//prover's secrete coloring*

Commit phase

//permutation program: use any secure pseudo-random-generator,
Permute (parameter) {
    // example the well-know BBS (Blum-Blum-Shub) pseudorandom generator will work well
}

//Permutation of coloring:
//Permute the colors {0, 1, 2} and compose with the given coloring to get a new permutated coloring P.
Prover_permutate (coloring s) {
    Pr = Permute ({0, 1, 2}); // randomly permute the colors
    P[i] = permute-coloring: = Pr(S (i));
}

//commit and write the commitments to the public object Cp (like, writing to R/W communicate tape)
//we may consider it as RSA encryption module
Prover_Commit (permutated coloring P) **{**
For i = 1 to n
    // relabel (rename) the vertices uniformly at random to get different labeling
    v[i] = Random_relabeling (V); //{1, 2, … n}

    //considering public-key encryption (RSA), n distinct key-pairs are needed
    Cp[i] = Commit$_{K[i]}$ (P[i], v[i]); //v[i] Є V random ordering of {1, 2, 3… n}
}

Reveal Phase

// (In a similar way) we may consider it as RSA decryption module
// Prover reveals the commitment (encoded values) color c and its label (name) v
Prover_reveal (selected edge se) **{**
//gives de-commitment of the values for the vertices labeled "**u**" and "**v**"
(P[se.u], V[se.u]) = **Reveal** $_{K[se.u]}$ **(C[se.u]);**
(P[se.v], V[se.v]) = **Reveal** $_{K[se.v]}$ **(C[se.v]);**
}

Verifier program:

Verifier_select_edge (G, Cp) {
// relabel the vertices uniformly at random to get different labeling of the permuted //coloring
Edge se = select_edge(G); //{(1,2) ,… m} –select edge uniformly at random
}

Verifier_verify (revealed edge e) {
//1. Verify coloring
If (Cp[e.u] = Cp[e.v])
Reject and stop; //bad coloring

//2. Verify commitment (binding)
//encrypt the revealed values back and verify (compare with) prover's commitments at commit //phase. Here the corresponding public keys K[e.u] and K[e.v] are used
Cv[e.u] = Commit$_{K[e.u]}$ **(P[e.u], V[e.u]);** //Cv – commitment verifier, encrypted revealed values
Cv[e.v] = Commit$_{K[e.v]}$ **(P[e.v], V[e.v]);**

If ((Cv[e.u] = Cp[e.u]) and (Cv[e.v] = Cp[e.v]))
Accept and continue challenge until convinced;
Else
Reject and stop; //ambiguous, commitment is not binding
}

The whole protocol program:

ProtocolPG3C (sample graph G, coloring)
{
//Prover: in each phase
//1. Permutate a typical sample coloring of the graph
Permuted coloring **p** = **prover_permutate** (G, coloring);

//2. Commit to the values of permuted coloring independently
Committed coloring **Cp** = **prover_commit** (G, **p**);

//Verifier: in each phase
//1. Select one edge uniformly at random
Selected edge **se** = verifier_select_edge (G, **Cp**);

//Prover
//3. Reveal the colors (along with their label) of the selected edge
Revealed edge e = prover_reveal (**se**);

//verifier
//2. Verify the reveal colors and decide to accept or reject
Verifier_verify (revealed edge e);
}

Suppose we want to run the protocol (4*m = 16 times) with predetermined selection for the verifier. Considering the above given sample inputs, we may run it as follows:

Run () {
For i = 1 to 4*m (16)

    ProtocolPG3C (G, coloring);

}