Towards a constructive formalization of Perfect Graph Theorems

Abhishek Kr Singh[1] and Raja Natarajan[2]

[1] Tata Institute of Fundamental Research, Mumbai
abhishek.singh@tifr.res.in
[2] Tata Institute of Fundamental Research, Mumbai
raja@tifr.res.in

Abstract. Interaction between clique number $\omega(G)$ and chromatic number $\chi(G)$ of a graph is a well studied topic in graph theory. Perfect Graph Theorems are probably the most important results in this direction. Graph G is called *perfect* if $\chi(H) = \omega(H)$ for every induced subgraph H of G. The Strong Perfect Graph Theorem (SPGT) states that a graph is perfect if and only if it does not contain an odd hole (or an odd anti-hole) as its induced subgraph. The Weak Perfect Graph Theorem (WPGT) states that a graph is perfect if and only if its complement is perfect. In this paper, we present a formal framework for verifying these results. We model finite simple graphs in the constructive type theory of Coq Proof Assistant without adding any axiom to it. Finally, we use this framework to present a constructive proof of the Lovász Replication Lemma, which is the central idea in the proof of Weak Perfect Graph Theorem.

1 Introduction

The chromatic number $\chi(G)$ of a graph G is the minimum number of colours needed to colour the vertices so that every two adjacent vertices get distinct colours. Finding out the chromatic number of a graph is NP-Hard [5]. However, one obvious lower bound is clique number $\omega(G)$, the size of the biggest clique in G. Consider the graphs shown in Figure 1.

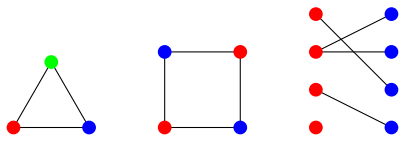

Fig. 1: Some graphs where $\chi(G) = \omega(G)$

In each of the above cases $\chi(G) = \omega(G)$, i.e. the number of colours needed is the minimum we can hope. Can we always hope $\chi(G) = \omega(G)$ for every graph

G? The answer is no. Consider the cycle of odd length 5 and its complement shown in Figure 2. In this case one can see that $\chi(G) = 3$ and $\omega(G) = 2$ (i.e. $\chi(G) > \omega(G)$). A cycle of odd length greater than or equal to 5 is called an *odd hole*. Complement of an odd hole is called an *odd anti-hole*. Indeed, the gap between $\chi(G)$ and $\omega(G)$ can be made arbitrarily large. Consider the other graph shown in Figure 2 which consist of two disjoint 5-cycles with all possible edges between the two cycles.

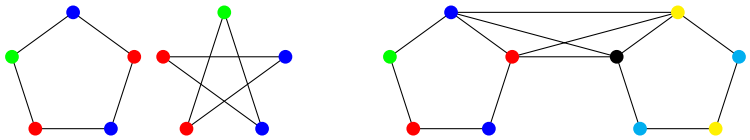

Fig. 2: Some graphs where, $\chi(G) > \omega(G)$. For k disjoint 5-cycles $\chi(G) = 3k$ and $\omega(G) = 2k$.

This graph is a special case of the general construction where we have k disjoint 5-cycles with all possible edges between any two copies. In this case one can show [7] that $\chi(G) = 3k$ but $\omega(G) = 2k$. In fact, there is an even stronger result [9] which constructs triangle-free Micielski graph M_k that satisfies $\chi(M_k) = k$.

In 1961, Claude Berge noticed the presence of odd holes (or odd anti-holes) as induced subgraph in all the graphs presented to him that does not have a nice colouring, i.e. $\chi(G) > \omega(G)$. However, he also observed some graphs containing odd holes, where $\chi(G) = \omega(G)$. Consider the graphs shown in Figure 3.

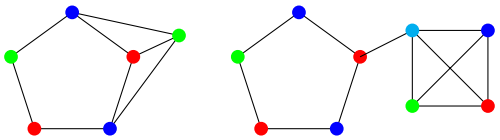

Fig. 3: Graphs with $\chi(G) = \omega(G)$, and having odd hole as induced subgraph.

A good way to avoid such artificial construction is to make the notion of nice colouring hereditary. We say that a graph H is an induced subgraph of G, if H is a subgraph of G and $E(H) = \{uv \in E(G) \mid u, v \in V(H)\}$. A graph G is then called a *perfect graph* if $\chi(H) = \omega(H)$ for all of its induced subgraphs H.

Berge observed that the presence of odd holes (or odd anti-holes) as induced subgraphs is the only possible obstruction for a graph to be perfect. These observations led Berge to the conjecture that a graph is perfect if and only if

it does not contain an odd hole (or an odd anti-hole) as its induced subgraph. This was soon known as the Strong Perfect Graph Conjecture (SPGC). Berge thought this conjecture to be a hard goal to prove and gave a weaker statement referred to as the Weak Perfect Graph Conjecture (WPGC): a graph is perfect if and only if its complement is perfect. Both the conjectures are theorems now. In 1972, Lovász proved a result [8] (known as Lovász Replication Lemma) which finally helped him to prove the WPGC. It took however three more decades to come up with a proof for SPGC. The proof of Strong Perfect Graph Conjecture was announced in 2002 by Chudnovsky et al. and finally published [3] in a 178-page paper in 2006.

In this paper, we present a formal framework for verifying these results. In Section 2 we provide an overview of the Lovász Replication Lemma which is the key result used in the proof of WPGT. In Section 3 we present a constructive formalization of finite simple graphs. In this constructive setting, we present a formal proof of the Lovász Replication Lemma (Section 4). We summarise the work in Section 5 with an overview of possible future works. The Coq formalization for this paper is available at [1].

2 Overview of Lovász Replication Lemma

Let G be a graph and $v \in V(G)$. We say that G' is obtained from G by repeating vertex v if G' is obtained from G by adding a new vertex v' such that v' is connected to v and to all the neighbours of v in G. For example, consider the graphs shown in Figure 4 obtained by repeating different vertices of a cycle of length 5.

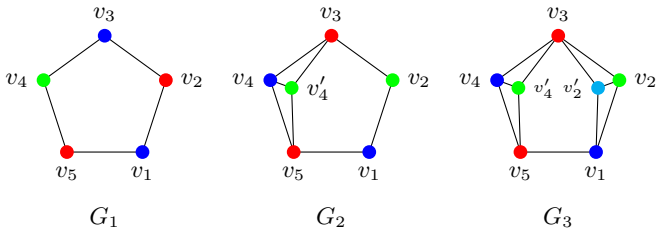

Fig. 4: Graph G_2 is obtained from G_1 by repeating vertex v_4 whereas the graph G_3 is obtained from G_2 by repeating vertex v_2. Note that $\chi(G_2) = \omega(G_2) = 3$ but $\chi(G_3) > \omega(G_3)$.

Note that the graph G_2 has a nice coloring (i.e. $\chi(G_2) = \omega(G_2)$), however the graph G_3 which is obtained by repeating vertex v_2 in graph G_2, does not have a nice coloring (i.e. $\chi(G_3) > \omega(G_3)$). Thus, property $\chi(G) = \omega(G)$ is not preserved by replication. Although, the property $\chi(G) = \omega(G)$ is not preserved, Lovász in 1971 came up with a surprising result which says that

perfectness is preserved by replication. It states that *if G' is obtained from a perfect graph G by replicating a vertex, then G' is perfect.* Note that this result does not apply to any graph shown in Figure 4, since none of them is a perfect graph. All of these graphs has an induced subgraph (odd hole of length 5) which does not admit a nice coloring.

The process of replication can be continued to obtain a graph where each vertex is replaced with a complete graph of arbitrary size (Figure 5). This gives us a generalised version of the Lovász Replication lemma.

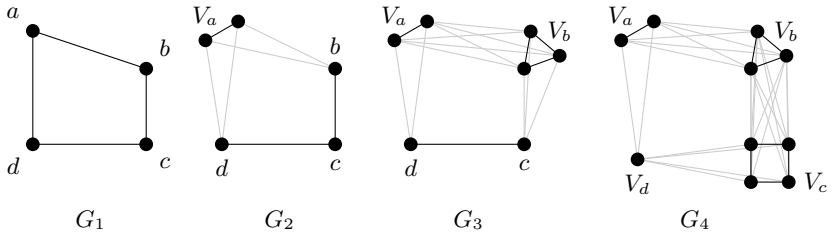

Fig. 5: The graphs resulting from repeated replication of vertices a, b and c of G_1 to form cliques V_a ,V_b and V_c of sizes 2, 3 and 4 respectively.

Let G be a perfect graph and $f : V(G) \to N$. Let G' be the graph obtained by replacing each vertex v_i of the graph G with a complete graph of order $f(v_i)$. Then G' is a perfect graph. For example, consider the graphs shown in Figure 5. Vertex a in G_1 is replaced by a complete graph V_a of order 2 to obtain G_2. Similarly, vertex b in G_2 is replaced by a clique V_b of size 3 to obtain G_3. Since G_1 is perfect, all the other graphs (G_2, G_3 and G_4) obtained by repeated replications are also perfect.

3 Modeling Finite Simple Graphs

There are very few formalization of graphs in Coq. The most extensive among these is due to the formalization of four color theorem [6] which considers only planar graphs. We use a definition for finite graphs which is closest to the one used by Doczkal et. al. [4]. However due to reasons explained in this section we represent the vertices of our graphs as sets over `ordType` instead of `eqType`. We define finite simple graphs as a dependent record with six fields.

```
Record UG (A:ordType) : Type:= Build_UG {
nodes :> list A;   nodes_IsOrd : IsOrd nodes;    edg: A -> A -> bool;
edg_irefl: irefl edg;   no_edg: edg only_at nodes;    edg_sym: sym edg  }.
Variable G: UG A.
```

The last line in the above code declares a finite graph `G` whose vertices come from an infinite domain `A`. The first field of `G` can be accessed using the term

`(nodes G)`. It is a list that represents the set of vertices of `G`. The second field of `G` ensures that the list of vertices can be considered as a set (details in Sec 3.1). The third field, which is accessed using the term `(edg G)`, is a binary relation representing the edges of the graph `G`. The terms `(edg_irefl G)` and `(edg_sym G)` are proof terms whose type ensures that the edge relation of `G` is irreflexive and symmetric. These restrictions on edge makes the graph `G` simple and undirected. For simplicity in reasoning, the edge relation is considered false everywhere outside the vertices of `G`. This fact is represented by the proof term `(no_edg G)`.

3.1 Vertices as constructive sets

In our work we only consider finite graphs. Vertices of a finite graph can be represented using a finite set. The Mathematical Components library [6] describes an efficient way of working with finite sets. Finite sets are implemented using finite functions (`ffun`) built over a finite type (`finType`). Since all the elements of a set now come from a finite domain (i.e. `finType`) almost every property on the set can be represented using computable (boolean) functions. These boolean functions can be used to do case analysis on different properties of a finite set in a constructive way.

The proof of WPGT involves expansion of graph in which the vertices of the initial graph are replaced with cliques of different sizes. Therefore, in our formalization we can't assume that the vertices of our initial graph are sets over some `finType`. Instead, we represent the set of vertices of a graph `G` as a list whose elements come from an infinite domain (defined as `ordType`).

Reflection, eqType and ordType The `finType` in ssreflect is defined on a base type called `eqType`. The `eqType` is defined as a dependent record which packs together a type (`T:Type`) and a boolean comparison operator (`eqb: T → T → bool`) that can be used to check the equality of any two elements of type `T`. Therefore, it tries to capture the notion of a domain with decidable equality. For example, consider the following canonical instance which connects natural numbers to the theory of `eqType`.

```
Canonical nat_eqType: eqType:=
{|Decidable.E:= nat; Decidable.eqb:= Nat.eqb;  Decidable.eqP:= nat_eqP|}.
```

Here, `Nat.eqb` is a boolean function that checks the equality of two natural number and the term `nat_eqP` is name of a lemma which ensures that the function `eqb` evaluates equality in correct way.

Lemma 1. `nat_eqP (m n:nat): reflect (m = n)(Nat.eqb m n)`

Note, the use of `reflect` predicate to specify a boolean function. It is a common practice in the ssreflect library. Once we connect a proposition `P` with a boolean `B` using the `reflect` predicate we can easily navigate between them. This makes case analysis on `P` possible even though the Excluded Middle

principle is not provable for an arbitrary proposition `Q` in the constructive type theory of Coq. Consider the following lemma (from `GenReflect.v`), which makes case analysis possible on a predicate `P`.

Lemma 2. `reflect_EM (P: Prop)(b:bool): reflect P b -> P ∨ ¬ P.`

To keep the library constructive we follow this style of proof development. All the basic predicates on sets are connected with their corresponding boolean functions using reflection lemmas. For example, consider the predicates mentioned in Table 1.

Propositions (P:Prop)	Boolean functions (b:bool)	Reflection lemmas
In a l	memb a l	membP
Equal l s	equal l s	equalP
∃ x, (In x l ∧ f x)	existsb f l	existsbP
∀ x, (In x l -> f x)	forallb f l	forallbP

Table 1: Some decidable predicates on sets from the file `SetReflect.v`

These lemmas can be used to do case analysis on any statement about sets containing elements of `eqType`. However, in this framework we can't be constructive while reasoning about properties of power sets. Hence, we base our set theory on `ordType` which is defined as a subtype of `eqType`.

The `ordType` inherits all the fields of `eqType` and has an extra boolean operator which we call the less than boolean operator (i.e. `ltb`). This new operator represents the notion of ordering among elements of `ordType`.

Sets as Ordered Lists Let `T` be an `ordType`. Sets on domain `T` is then defined as a dependent record with two fields. The first field is a list of elements of type `T` and the second field ensures that the list is ordered using the `ltb` relation of `T`.

```
Record set_on  (T:ordType): Type := { S_of :> list T;  IsOrd_S : IsOrd S_of }.
```

All the basic operations on sets (e.g. union, intersection and set difference) are implemented using functions on ordered lists which outputs an ordered list. An important consequence of representing sets using ordered list is the following lemma (from `OrdList.v`) which states that the element wise equal sets can be substituted for each other in any context.

Lemma 3. `set_equal (A: ordType)(l s: set_on A): Equal l s -> l = s.`

Another advantage of representing sets using ordered list is that we can now enumerate all the subsets of a set `S` in a list using the function `pw(S)`. Moreover, we have following lemma which states that the list generated by `pw(S)` is a set. The details of function `pw(S)` can be found in the file `Powerset.v`.

Lemma 4. `pw_is_ord (S: list A): IsOrd (pw S).`

Since all the subsets of `S` are present in the list `pw(S)` we can express any predicate on power set using a boolean function on list. This gives us a constructive framework for reasoning about properties of sets as well as their power sets. For example, consider the following definition of a boolean function `forall_xyb` and its corresponding reflection lemma `forall_xyP` from the file `SetReflect.v`.

```
Definition forall_xyb (g:A->A->bool)(l:list A):=
            (forallb (fun x=> (forallb (fun y => g x y) l )) l).
Lemma forall_xyP (g:A->A->bool) (l:list A):
  reflect (forall x y, In x l-> In y l-> g x y)  (forall_xyb g l).
```

3.2 Decidable Edge relation

The edges of graph `G` are represented using a decidable binary relation on the vertices of `G`. Hence, one can check the presence of an edge between vertices `u` and `v` by evaluating the expression `(edg G u v)`. The decidability of edge relation is useful for defining many other important properties of graphs as decidable predicates.

Cliques, Stable Set and Graph Coloring Consider the following definition of a complete graph `K` present in the graph `G`. Note that the proposition `Cliq G K` is decidable because it is connected to a term of type bool (i.e. `cliq G K`) by the reflection lemma `cliqP`.

```
Definition cliq(G:UG)(K:list A):=forall_xyb (fun x y=> (x==y) || edg G x y) K.
Definition Cliq(G:UG)(K:list A):=(forall x y,In x K->In y K-> x=y \/ edg G x y).
Lemma cliqP (G: UG)(K: list A): reflect (Cliq G K) (cliq G K).
```

In a similar way we also define independence set (or stable set) and graph coloring using decidable predicates. The details can be found in the file `UG.v` and `MoreUG.v`. Most of these these definitions together with their reflection lemmas are listed in Table 2.

Propositions (P:Prop)	Boolean functions (b:bool)	Reflection lemmas
Subgraph G1 G2	subgraph G1 G2	subgraphP
Ind_Subgraph G1 G2	ind_subgraph G1 G2	ind_subgraphP
Stable G I	stable G I	stableP
Max_I_in G I	max_I_in G I	max_I_inP
Cliq G K	cliq G K	cliqP
Max_K_in G K	max_K_in G K	max_K_inP
Coloring_of G f	coloring_of G f	coloring_ofP

Table 2: Decidable predicates on finite graphs (from `UG.v` and `MoreUG.v`).

We call a graph G to be a nice graph if $\chi(G) = \omega(G)$. A graph G is then called a perfect graph if every induced subgraph of it is a nice graph.

```
Definition Nice (G: UG): Prop:= forall n, cliq_num G n -> chrom_num G n.
Definition Perfect (G: UG): Prop:= forall H, Ind_subgraph H G -> Nice H.
```

In this setting we have the following lemma establishing the obvious relationship between $\chi(G)$ and $\omega(G)$. Here the expression `(clrs_of f G)` represents the set containing all colors used by `f` to color the vertices of `G`.

Lemma 5. `more_clrs_than_cliq_size (G: UG)(K: list A)(f: A -> nat): Cliq_in G K-> Coloring_of G f -> |K| <= |clrs_of f G|.`

Lemma 6. `more_clrs_than_cliq_num (G: UG)(n:nat)(f: A->nat): cliq_num G n-> Coloring_of G f -> n <= |clrs_of f G|.`

3.3 Graph Isomorphism

It is typically assumed in any proof involving graphs that isomorphic graphs have exactly the same properties. However, in a formal setting we need a proper representation for graph isomorphism to claim the exact behaviour of isomorphic graphs.

```
Definition iso_using (f: A->A)(G G': @UG A) := (forall x, f (f x) = x) /\
  (nodes G') = (img f G)  /\   (forall x y, edg G x y = edg G' (f x) (f y)).
Definition iso (G G': @UG A) := exists (f: A->A), iso_using f G G'.
```

Consider the following lemmas which shows the symmetric nature of graph isomorphism.

Lemma 7. `iso_sym (G G': UG): iso G G' -> iso G' G.`

Note the self invertible nature of `f` which makes it injective on both `G` and `G'`. The second condition (i.e. `(nodes G') = (img f G)`) expresses the fact that f maps all the vertices of `G` to the vertices of `G'`.

Lemma 8. `iso_one_one (G G': UG)(f: A-> A): iso_using f G G'-> one_one_on G f.`

Lemma 9. `iso_subgraphs (G G' H :UG) (f: A-> A) : iso_using f G G'-> Ind_subgraph H G -> (∃ H', Ind_subgraph H' G' ∧ iso_using f H H').`

For the graphs `G` and `G'` Lemma 9 states that every induced subgraph `H` of `G` has an isomorphic counterpart `H'` in `G'`. In a similar way we can prove that the stable sets and cliques in `G` has isomorphic counterparts in `G'`. For example, consider the following lemmas from `IsoUG.v` summarising these results.

Lemma 10. `iso_cliq (G G': UG)(f:A-> A)(K:list A):iso_using f G G'-> Cliq G K -> Cliq G' (img f K).`

Lemma 11. `iso_stable (G G': UG)(f: A-> A)(I: list A):iso_using f G G'-> Stable G I-> Stable G' (img f I).`

Lemma 12. `iso_coloring(G G':UG)(f:A->A)(C: A->nat):iso_using f G G' -> Coloring_of G C -> Coloring_of G' (fun (x:A) => C (f x)).`

Lemma 13. `perfect_G' (G G':UG ): iso G G'-> Perfect G -> Perfect G'.`

3.4 Graph constructions

Adding (or removing) edges in an existing graph to obtain a new graph is a common procedure in proofs involving graphs. In such circumstances an explicit specification of all the fields of the new graph becomes a tedious job.

For example, consider the definition of following function `(nw_edg G a a')`.

```
Definition nw_edg(G:UG)(a a':A):= fun(x y:A)=> match (x==a), (y==a') with
                                  | _ , false => (edg G) x y
                                  | true, true => true
                                  |false, true => (edg G) x a
                            end.
```

The term `(nw_edg G a a')` can be used to describe the edge relation of graph `G'` shown in Figure 6, which is obtained from `G` by repeating the vertex `a` to `a'`.

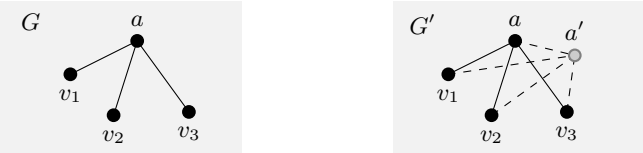

Fig. 6: Graph G' is obtained from G by repeating vertex a to a'.

This function has a simple definition and hence it is easy to prove various properties about it. For example, we can prove following results establishing connections between the edges of `G` and `G'`.

Lemma 14. `nw_edg_xa_xa' (G: UG)(a a' x:A): (edg G) x a -> (nw_edg G a a') x a'.`

Lemma 15. `nw_edg_xy_xy (G: UG)(a a' x y:A)(P': ¬ In a' G): (edg G) x y -> (nw_edg G a a') x y`

Lemma 16. `nw_edg_xy_xy4 (G: UG)(a a' x y:A)(P: In a G)(P': ¬In a' G): y ≠ a' -> (edg G) x y = (nw_edg G a a') x y.`

Although, the term `(nw_edg G a a')` contains all the essential properties of the construction it doesn't have the irreflexive and symmetric properties necessary for an edge relation. Hence, we can't use this term for edge relation while declaring `G'` as an instance of `UG`. To ensure these properties one can add more branches to the match statement and provide an extra term `P` of type `a ≠ a'` as argument to the function. However, this will result in a more complex function and proving even the essential properties of the new function becomes hard.

Instead of writing complex edge relations every time we define functions namely `mk_irefl`, `mk_sym` and `E_res_to` which make minimum changes and

convert any binary relation on vertices into an edge relation. For example consider the following specification lemmas for the functions mk_irefl and mk_sym.

Lemma 17. mk_ireflP (E: A -> A-> bool): irefl (mk_irefl E).

Lemma 18. mk_symP (E: A-> A-> bool): sym (mk_sym E).

Lemma 19. irefl_inv_for_mk_sym (E: A-> A-> bool): irefl E -> irefl (mk_sym E).

Lemma 20. sym_inv_for_mk_irefl (E: A->A-> bool): sym E -> sym (mk_irefl E).

Note that these functions do not change the properties ensured by each other. The file UG.v contains other invariance results about these functions proving that these functions work well when used together. For example, consider the following declaration of G' as an instance of UG.

```
Definition ex_edg(G:UG)(a a':A):=
                      mk_sym(mk_irefl((nw_edg G a a') at_ (add a' G))).
Variable G: UG.
Definition G':= refine({| nodes:= add a' G; edg:= (ex_edg G a a');
                      |}); unfold ex_edg. all: auto. Defined.
```

Note that the term (ex_edg G a a') obtained from (nw_edg G a a') by using these functions have all the properties of an edge relation. Now, we can simply use the tactic all: auto to discharge all the proof obligations generated while declaring G' as an instance of UG. Therefore these functions can significantly ease the construction of new graphs.

All the important properties of the final edge relation (i.e. ex_edg G a a') can now be derived from the properties of nw_edg G a a' by using the specification lemmas for the functions mk_irefl, mk_sym and E_res_to. For example consider following lemmas (from Lovasz.v) which describes the final edge relation (i.e. ex_edg) of graph G'.

Lemma 21. Exy_E'xy (x y:A)(P: In a G)(P': ¬ In a' G): edg G x y -> edg G' x y.

Lemma 22. In_Exy_eq_E'xy (x y:A)(P: In a G)(P': ¬ In a' G): In x G-> In y G-> edg G x y=edg G' x y.

Lemma 23. Exy_eq_E'xy (x y:A)(P: In a G)(P': ¬ In a' G): x ≠ a'-> y ≠ a'-> edg G x y = edg G' x y.

Lemma 24. Exa_eq_E'xa' (x:A)(P: In a G)(P': ¬ In a' G): x ≠ a-> x ≠ a'-> edg G x a = edg G' x a'.

Lemma 25. Eay_eq_E'a'y (y:A)(P: In a G)(P': ¬ In a' G): y ≠ a -> y ≠ a' -> edg G a y = edg G' a' y.

4 Constructive proof of Lovász Replication Lemma

Let G and G' be the graphs discussed in section 3.4, where G' is obtained from G by repeating the vertex a to a'. Then we have the following lemma.

Lemma 26. `ReplicationLemma Perfect G -> Perfect G'.`

Proof: We prove this result using induction on the size of graph G.

– Induction hypothesis(IH): `∀ X, |X|<|G|-> Perfect X -> Perfect X'`

Let H' be an arbitrary induced subgraph of G', then we need to prove that χ(H') = ω(H'). We prove this equality in both of the following cases.

– **Case-1** (H' ≠ G'): In this case H' is strictly included in G'. We further do case analysis on the proposition (a ∈ H').
  - **Case-1A** (a ∉ H'): In this case if a' ∉ H' then H' is an induced subgraph of G and hence χ(H') = ω(H'). Now consider the case when a' ∈ H'. Let H be the induced subgraph of H' restricted to the vertex-set (H'\a') ∪ {a}. Note that H' is isomorphic to H and H is an induced subgraph of G. Hence H' is a perfect graph and we have χ(H') = ω(H').
  - **Case-1B** (a ∈ H'): Again in this case if a' ∉ H' then H' is an induced subgraph of G and hence χ(H') = ω(H'). Now we are in the case where a ∈ H' , a' ∈ H' and H' is strictly included in G'. Therefore, the set H'\a' is strictly included in G. Let H be the induced subgraph of G with vertex set H'\a'. Note that H' can be obtained by repeating a to a' in H. But, we know that |H| < |G|, hence H' is a perfect graph by induction hypothesis (IH) and we have χ(H') = ω(H').
– **Case-2** (H' = G'): In this case we need to prove χ(G') = ω(G'). We further split this case into two sub cases.
  - **Case-2A**: In this case we assume that there exists a clique K of size ω(G) such that a ∈ K. Hence K gets extended to a clique of size ω(G)+1 in G' and ω(G')= ω(G)+1. Now we can assign a new color to the vertex a' which is different from all the colors assigned to G. Hence χ(G')= χ(G)+1=ω(G)+1=ω(G').
  - **Case-2B**: In this case we assume that a does not belong to any clique K of size ω(G). Since G is a perfect graph let f be a coloring of graph G which uses exactly ω(G) colors. Let G* = { v∈ G: f(v) ≠ f(a) ∨ v=a }. For the subgraph G* we can then show that ω(G*) < ω(G). Hence there must exist a coloring f* which uses ω(G*) colors for coloring G*. Since ω(G*) < χ(G) we can safely assume that f* does not use the color f (a) for coloring the vertices of G*. Now consider a coloring f' which assigns a vertex x color f*(x) if x belongs to G* otherwise f' (x) = f (a). Note that the number of colors used by f is at most ω(G). Hence χ(G') = ω(G').

Note that all the cases in the above proof correspond to predicates on sets and finite graphs. Since we have decidable representations for all of these predicates, we could do case analysis on them without assuming any axiom. □

5 Conclusions and Future Work

Formal verification of a mathematical theory can often lead to a deeper understanding of the verified results and hence increases our confidence in the theory. However, the task of formalization soon becomes overwhelming because the length of formal proofs blows up significantly. In such circumstances having a library of facts on commonly occurring mathematical structures can be really helpful. The main contribution of this paper is a constructive formalisation of finite simple graphs in the Coq proof assistant [2]. This formalization can be used as a framework to verify other important results on finite graphs. To keep the formalisation constructive we follow a proof style similar to the small scale reflections technique of the ssreflect. We use small boolean functions in a systematic way to represent various predicates over sets and graphs. These functions together with their specification lemmas can help in avoiding the use of Excluded-Middle in the proof development. We also describe functions to ease the process of new graph construction. These functions can help in discharging most of the proof obligation generated while creating a new instance of finite graph. Finally, we use this framework to present a fully constructive proof of the Lovász Replication Lemma [8], which is the central idea in the proof of Weak Perfect Graph Theorem. One can immediately extend this work by formally verifying Weak Perfect Graph Theorem in the same framework. Another direction of work could be to add in the present framework all the basic classes of graphs and decompositions involved in the proof of Strong Perfect Graph Theorem. This can finally result in a constructive formalisation of strong Perfect Graph Theorem in the Coq proof assistant.

References

1. Coq formalization. `https://github.com/Abhishek-TIFR/List-Set`.
2. The Coq Standard Library. `https://coq.inria.fr/library/`.
3. Maria Chudnovsky, Neil Robertson, Paul Seymour, and Robin Thomas. The strong perfect graph theorem. *ANNALS OF MATHEMATICS*, 164:51–229, 2006.
4. Christian Doczkal, Guillaume Combette, and Damien Pous. A formal proof of the minor-exclusion property for treewidth-two graphs. In Jeremy Avigad and Assia Mahboubi, editors, *Interactive Theorem Proving*, pages 178–195, 2018.
5. Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1990.
6. Georges Gonthier and Assia Mahboubi. An introduction to small scale reflection in Coq. *Journal of Formalized Reasoning*, 3(2):95–152, 2010.
7. Andras Gyarfas, Andras Sebo, and Nicolas Trotignon. The chromatic gap and its extremes. *Journal of Combinatorial Theory, Series B*, 102(5):1155 – 1178, 2012.
8. L. Lovász. Normal hypergraphs and the perfect graph conjecture. *Discrete Mathematics*, 2(3):253 – 267, 1972.
9. Jan Mycielski. Sur le coloriage des graphs. *Colloquium Mathematicae*, 3(2):161–162, 1955.