Building Graph Theory Using Coq: Verification of Wigderson's Graph Coloring Algorithm

Siraphob (Ben) Phipathananunth, Jamison Homatas

Abstract

Formally verifying the correctness of algorithms has become extremely important due to the impact even a small error can have. While many algorithms can be considered proven on paper (i.e. informally) they do not consider implementation factors such as data structures and memory. Many algorithms operate on graphs which is a relatively simple concept but more complicated when implemented as a data structure. There is not much in literature on formal verification of graph algorithms and in particular coloring algorithms. Graph coloring is the problem of assigning a color to each vertex in a graph such that no two adjacent vertices have the same color. While the problem of coloring a 3-colorable graph may seem simple, it is actually NP-hard, so there has been an effort to find polynomial-time algorithms that color a 3-colorable graph with as few colors as possible. One of the first major advancements in this problem was made by Avi Wigderson in which he provided a polynomial-time algorithm to color a graph with n vertices using $O(\sqrt{n})$ colors [7]. Since then, there have been many more advancements in improving this, and now the best known algorithm uses $o(n^{\frac{1}{5}})$ colors [6]. We will formalize the correctness of Wigderson's algorithm to prove that it indeed correctly colors a 3-colorable graph with $O(\sqrt{n})$ colors. In particular, we will attain a bound of exactly $3\sqrt{n}$. We will also show that the algorithm is robust i.e. if the graph is not 3-colorable, the graph will either be correctly colored with $O(\sqrt{n})$ colors or will identify that the graph is not 3-colorable. We will prove the desired statements in a faithful way, encoding the graphs as a Coq data structure. Wigderson's algorithm has been a useful foundation for this problem, so verifying its correctness may be a valuable resource for extending this to other coloring algorithms or graph algorithms in general. Our code can be found at github.com/siraben/coq-wigderson.

1 Introduction

Graph coloring has many applications in which we want color graphs with as few colors as possible for tasks like scheduling or pattern matching. A k-colorable graph is a graph that can be colored in k colors. For planar graphs, there is the well known 4-color theorem that has been proven in Coq, stating that all planar graphs are 4-colorable. This can be applied to geographical maps, which can be represented as planar graphs. Wigderson's algorithm aims to find a coloring approximation for 3-colorable graphs. Since it is difficult to find an exact coloring, the algorithm will produce a valid coloring with at most $3\sqrt{n}$ colors. If the graph is not 3-colorable, then either a valid approximation is returned or a certificate that the input was not 3-colorable.

Throughout our paper we will refer to paper proofs as *informal* proofs and proofs in a proof assistant as *formal* proofs. Our development will closely follow the paper by Wigderson [7] in which he presents a proof of correctness with the given color bound and its polynomial running time, with appropriate lemmas. We will prove the correctness of the algorithm with respect to a *functional* model of it in Coq's underlying language, Gallina. This is in contrast to the paper which gives an imperative algorithm in pseudocode. The reason for this is to improve the ease of the proof development by avoiding reasoning with Hoare logic. We will refer to texts such as Software Foundations [1], especially volume 3 which details techniques for proving properties about functional algorithms and data structures. There is also a formalization of Kempe's graph coloring algorithm in the text, but it does not verify the amount of colors needed and can leave the graph partially colored. Nevertheless it would help guide our initial development.

The operation of Wigderson's algorithm is rather straightforward, but at the more rigorous level, this become more difficult. Even basic operations may require many definitions and lemmas to be proven correct. We must reformulate definitions which are compatible with data structures. We then must build off of our definitions to establish lemmas which concern the properties of graphs and colorability. We then translate the algorithm into a functional version for Coq and use these lemmas to prove the correctness and robustness of the algorithm.

2 Wigderson's Coloring Approximation Algorithm

For now, we will assume that any input graph is 3-colorable before we discuss robustness. The idea of Wigderson's algorithm is to find vertices with degree of least k. Finding these high-degree vertices allows us to color more vertices at once; we are able to 2-color the neighborhood for each of these vertices. Then we remove the colored vertices and continue this until no such high-degree vertices remain. Then color the remaining vertices with new colors. Then the pseudo-code algorithm he cites [7] is as follows where $\Delta(G)$ is the maximum degree of any vertex in G:

Algorithm 1 Widgerson's 3-coloring Approximation Algorithm

```
Input: A 3-colorable graph $G(V, E)$
$n \leftarrow |V|$
$i \leftarrow 1$
while $\Delta(G) \geq k$ do
    $H \leftarrow$ the subgraph of $G$ induced by the neighborhood $N_G(v)$
    2-color $H$ with colors $i, i+1$
    color $v$ with color $i+2$.
    $i \leftarrow i+2$
    $G \leftarrow$ the subgraph of $G$ resulting from it by deleting $N_G(v) \cup \{v\}$
end while
color $G$ with colors $i, i+1, i+2, \ldots, \Delta(G)$ and halt
```

For a 3-colorable graph, there exists some coloring in which each vertex has a color. The neighborhood of any vertex must be one of the two other colors, so the neighborhood is 2-colorable. We can find a 2-coloring easily in linear time by recursively forcing colors. We do this for vertices with higher degrees to eliminate as many colors as possible. Finally, we naively color the remaining vertices.

In the while loop, i is incremented by 2 and 3 colors are used. This means there will be overlap between the final color used on the previous iteration and the first color used on the current iteration. This is possible since the final color assigned on each iteration is to v. Since the neighborhood of v, $N_G(v)$ was already colored, reusing this color for other vertices will not cause any contradictions. To make verification easier, we instead set the color of high-degree to color 1 every iteration and use two unique colors for the neighborhoods.

2.1 Finding a Bound of Colors Used

Let n be the number of vertices in the graph. In a dense graph, it is possible that all vertices have at least degree k, but each iteration also removes at least $k+1$ vertices from the graph. We can remove at most n vertices, so $(k+1)x \leq n$ where x is the number of iterations, and thus $x \leq \frac{n}{k+1}$. Then once the loop terminates, then $\Delta(G) < k$, so we can use a polynomial time algorithm to color these vertices using at most $1 + \Delta(G) < 1 + k$ colors. Therefore, we use at most k colors to color these vertices. This gives an upper bound of $k + \frac{2n}{k}$ colors used since there are 2 new colors used each iteration. We want to balance these two terms by selecting an appropriate k as follows

$$\begin{aligned} k &= \frac{2n}{k} \\ k^2 &= 2n \\ k &= \sqrt{2n} \end{aligned}$$

This leads to a bound of $\sqrt{2n} + \frac{2n}{\sqrt{2n}} = 2\sqrt{2n} = \sqrt{8}\sqrt{n} \approx 2.828\sqrt{n} = O(\sqrt{n})$. For sake of simplicity, we will use $k = \sqrt{n}$ as Wigderson did. This will give us a bound of $\sqrt{n} + \frac{2n}{\sqrt{n}} = 3\sqrt{n} = O(\sqrt{n})$.

2.2 Translation to Functional Algorithm

We want to further explain the steps of the algorithm to convert this into a functional program to be used in Coq. We use the updated color assignment process we described and use the value $k = \sqrt{n}$. The algorithm can be described in two phases: the first where we color the high-degree vertices and their neighborhoods, and the second is coloring the remaining vertices. We first present the pseudocode for both Phase I and Phase II of the algorithm, each of which are divided into further subroutines.

In Phase I, the first function two-colors the connected component of a vertex. It arbitrarily selects a color and forces the adjacent vertices until we must arbitrarily select another color for the next connected component. The next function applies this to the whole neighborhood of a vertex. Finally, the Phase I function selects high-degree vertices and colors them and their neighborhoods until there are no more high-degree vertices remaining. This leaves us with a graph with no high degree vertex in which we will then use for Phase II.

In phase II, the goal is to color the remaining graph with $d+1$ colors where d is the maximum degree of the graph. The first function removes (non-adjacent) vertices with degree d and assigns them the same color. The second function simply applies this for all degrees from d down to 0 which will fully color the graph with $d+1$ colors.

Algorithm 2 Phase I Algorithm

```
Input: A graph $G(V, E)$ with $|V| = n$
function TWO-COLOR-VERTEX($v, c_1, c_2$)
    Color $v$ with color $c_1$
    if $v$ has any uncolored neighbors then
        TWO-COLOR-VERTEX($x, c_2, c_1$) for all uncolored neighbors $x$ of $v$
    end if
    return new coloring of $G$
end function
function TWO-COLOR-NEIGHBORHOOD(coloring $f$ of $N$)
    if there exists an uncolored vertex in $f$ then
        $v \leftarrow$ the first uncolored vertex from $f$
        $f \leftarrow$ TWO-COLOR-VERTEX($v, c_1, c_2$)
        TWO-COLOR-NEIGHBORHOOD($f$)
    end if
    return $f$
end function
function PHASE-1(graph $G(V, E)$)
    $f \leftarrow$ empty coloring
    if there exists a vertex with degree at least $\sqrt{n}$ then
        $v \leftarrow$ first vertex with degree at least $\sqrt{n}$
        $f \leftarrow f$ with $v$ assigned color 1
        $f \leftarrow$ TWO-COLOR-NEIGHBORHOOD($f$)
        $G \leftarrow G - (v \cup N_G(v))$
        $f, G \leftarrow$ PHASE-1($G$)
    end if
    return $f, G$
end function
return PHASE-1($G$)
```

Algorithm 3 Phase II Algorithm

```
Input: A graph $G(V, E)$ with maximum degree $d$
function COLOR-D($G, d, c, f$)
    if there exists a vertex with degree $d$ then
        $v \leftarrow$ first vertex with degree $d$ in $G$
        $f \leftarrow f$ with color $c$ assigned to $v$
        remove $v$ from $G$
        return COLOR-D($G, d, c, f$)
    end if
    return $G$
end function
function COLOR-ALL-D($G, d, f$, colors $c_0, c_2, \ldots, c_d$)
    if $d \geq 0$ then
        $G \leftarrow$ COLOR-D($G, d, c_d, f$)
        return COLOR-ALL-D($G, d-1, f, c_0, \ldots, c_{d-1}$)
    end if
    return $G, f$
end function
return COLOR-ALL-D($G, d, f, c_0, \ldots, c_d$)
```

2.3 Understanding Correctness and Robustness

We will now present the informal proofs of correctness to help us translate these ideas formally into Coq.

Lemma 1. *The subgraph formed by the neighborhood of a vertex in a n-colorable graph is $n-1$-colorable.*

Proof. Let G be a n-colorable graph and let v be an arbitrary vertex in G. Then there exists a coloring of G using at most n different colors. Vertex v must be assigned some color c. Then all vertices adjacent to v i.e. the neighborhood of v will have colors different than c. Since the graph is n-colorable, one of these being c, the neighborhood can only use at most $n-1$ colors. □

In the Phase I algorithm, we attempt to 2-color each neighborhood of high-degree vertices. For a 2-colorable graph, the 2-coloring function will work since we are simply forcing the choices logically. If this 2-coloring fails, then the neighborhood is not two colorable, and by Lemma 1, this means the graph is not 3-colorable. In this case, we simply return this as a certificate that the input graph was not 3-colorable. The color of the high-degree vertex will be assigned color 1. For the next high-degree vertex, each of its neighbors cannot be a high-degree vertex already used since this would mean the vertex would have been colored. Thus, we can reuse the color 1. Each step uses $\sqrt{n}$ new vertices, so this means there are at most $\frac{n}{\sqrt{n}} = \sqrt{n}$ iterations. This means there are $2\sqrt{n}+1$ colors in this process. Since the loop terminates when there are no more vertices of at least degree $\sqrt{n}$, we know that after this process the uncolored vertices will have degree less than $\sqrt{n}$ i.e. maximum degree is at most $\sqrt{n}-1$. The final process simply requires assigning different colors for each degree. Since we can assign the same color to each vertex in a 1-colorable graph, Phase II will work by induction. If we assume the process will succeed for $d-1$ and produce a d-coloring, then we remove each vertex with degree d. We cannot remove two neighboring vertices since the degree of the neighbors will decrease by 1 once remove. Therefore, we can use this color added to the d-coloring to form a $d+1$ coloring as desired. This gives us a total of $3\sqrt{n}+1$ colors (we can reuse a color in the final step for exactly $3\sqrt{n}$).

3 Formalizing Definitions in Coq

It is not too difficult to explain what a graph is in words, but defining this from scratch in Coq is fairly involved. This requires breaking down each component of the graph to a low level for Coq to understand. We will include definitions all data structures and concepts we use such as graphs and colorings.

3.1 Overview of Coq

Our work takes place in a logical framework of constructive type theory, such as that of Coq. We have the hierarchy of *type universes* and a universe of *propositions*, called `Prop`, written as $\mathbb{P}$. At the type level, we have the unit type $\mathbf{1}$, the empty type $\mathbf{0}$, function types $A \to B$, products $A \times B$, sums $A + B$, dependent products $\forall (x : X), F(x)$ and dependent sums $\exists (x : X), F(x)$. Note that function application associates to the left in Coq and every function takes one argument, so we can write $f\,a\,b$ instead of $(f(a))(b)$. For propositions, these are written as they are in logic, $(\top, \bot, \to, \wedge, \vee, \forall, \exists)$. For brevity we will not go into details about the Coq type system here, which is well-covered in literature.

An important point to make is that Coq's logic is *constructive*. In particular we do not assume excluded middle (LEM), which is the statement $\forall (p : \mathbb{P}), p \vee \neg p$. This is obvious in classical mathematics but since we are working in type theory, disjunctions have to be computable (that is, we can always find out in finite time if the disjunction is the left or the right element. In general this does not hold in Coq (because p could be undecidable), but fortunately for us everything is pretty much decidable (comparing integers, checking if a key is in a map or set, etc.).

Coq consists of two languages, *Gallina* and *Ltac*. Gallina is the specification language of Coq and can be thought of as the expressions in Coq. Gallina is purely functional and has support for dependent types and dependent pattern matching. *Ltac* is the tactic language of Coq and is what is used to carry out formal proofs. An introduction can be found in [5] and [2]. It suffices to say that, from a usability standpoint, *Ltac* commands operate on the current *proof state*, which is the context consisting of hypothesis and a goal. The commands may introduce new hypotheses, clear existing ones, allow application of one hypothesis to another, discriminate a value in context, and so on.

3.2 Graphs and Data Structures

We represent a directed graph with an adjacency set representation. A directed graph is a map from vertices to sets of vertices. The type of vertices is the positive integers. We use the Coq library for efficient functional maps and sets which use positive integers for the keys. Throughout the development and our paper, `S.t` refers to the type of sets of positive integers, `E.t` refers to positive integers, and `M.t A` refers to a map from positive integers to values of type `A`.

```
Definition node := E.t.
Definition nodeset := S.t.
Definition nodemap: Type -> Type := M.t.
Definition graph := nodemap nodeset.
```

We then define various functions on graphs, such as computing the adjacency set of a graph.

```
Definition adj (g: graph) (i: node) : nodeset :=
  match M.find i g with Some a => a | None => S.empty end.
```

An undirected graph is defined as a graph where for every vertex i and every vertex j that is in i's adjacency list, i is also in j's adjacency list. The **subset_nodes** function takes a predicate P that takes a node and its adjacency set and returns the subset of nodes that satisfy the predicate. To remove a vertex from a graph, we remove its entry in the map and remove the vertex from all the adjacency sets. We define a predicate **high_deg** that is true when a vertex's adjacency set has size less than K.

```
Definition undirected (g: graph) : Prop :=
   forall i j, S.In j (adj g i) -> S.In i (adj g j).

Definition nodes (g: graph) : nodeset := Mdomain g.

Definition subset_nodes
                    (P: node -> nodeset -> bool)
                    (g: graph) : nodeset :=
   M.fold (fun n adj s => if P n adj then S.add n s else s) g S.empty.

Definition remove_node (n: node) (g: graph) : graph :=
  M.map (S.remove n) (M.remove n g).

Definition high_deg (K: nat) (n: node) (adj: nodeset) : bool := K <? S.cardinal adj.
```

3.3 Defining Graph Properties and Colorability

We define a coloring as a map from the vertices to a set of colors. This coloring is considered a k-coloring for a set of size at at most k. We say a graph is k-colorable if there exists a k-coloring. This means that the coloring is induced from the 3-coloring, so we need to define this notion formally. With our definition, if we assume a graph is 3-colorable, then we can use the 3-coloring given to us by Coq. Our goal is to only use these colorings for establishing definitions and lemmas and not as the coloring generated by the algorithm.

We must define the concept of a neighborhood, but before we can do that, we need to a define a subgraph. This is simply a subset of vertices whose adjacency lists are subset of the adjacency lists of the entire graph. In Coq we have

```
Definition is_subgraph (g' g : graph) :=
  S.Subset (nodes g') (nodes g) /\ forall v, S.Subset (adj g' v) (adj g v).
```

Then an induced subgraph is the subgraph generated from a subset of vertices in which each edge in the original graph between these vertices is contained in the subgraph.

```
Definition subgraph_of (g : graph) (s : S.t) : graph :=
  M.fold (fun v adj g' => if S.mem v s then M.add v (S.inter s adj) g' else g') g empty_graph.
```

Combining these definitions, the neighborhood of a vertex v is the induced subgraph formed by the set of neighbors of v with v removed.

```
Definition neighborhood (g : graph) v := remove_node v (subgraph_of g (neighbors g v)).
```

Now we define what a coloring is and what it mean for a coloring to be OK and complete on a graph. A coloring is a map from vertices to colors. A coloring f is OK on a graph g with respect to a palette p if for every adjacent pair i, j, if i is colored then $f(i) \in p$ and if j is colored then $f(j) \in p$ and $f(i) \neq f(j)$. A coloring is complete if every vertex in g has a color and the coloring is OK. This is an important distinction to make because the colorings that our algorithm builds up are partial and must be combined to eventually make a complete coloring.

```
Definition colors := S.t.
Definition coloring := M.t node.
Definition coloring_ok (palette: S.t) (g: graph) (f: coloring) :=
 forall i j, S.In j (adj g i) ->
     (forall ci, M.find i f = Some ci -> S.In ci palette) /\
     (forall ci cj, M.find i f = Some ci -> M.find j f = Some cj -> ci<>cj).
Definition coloring_complete (palette: colors) (g: graph) (f: coloring) :=
 (forall i, M.In i g -> M.In i f) /\ coloring_ok palette g f.
```

Now, combining colorings and neighborhoods, we can talk about what it means to restrict a coloring on a neighborhood of a vertex.

```
Definition restrict_on_nbd (f : coloring) (g : graph) (v : node) :=
  restrict f (nodes (neighborhood g v)).
```

We also have to define the degree of a vertex, and the maximum degree of a graph. This is needed so we can state things such as "a graph with maximum degree d can be colored with at most d+1 colors."

4 Building Lemmas

Now that we have established the fundamental graph and colorability definitions. We will use these to form lemmas corresponding to the properties of the algorithm.

4.1 General Lemmas

Naturally, since we are building graph theory from scratch, there are many lemmas that we have to prove, even ones that are fairly obvious. We omit their proofs here but summarize their statements in words:

- the subgraph relation is reflexive and transitive
- induced subgraphs are indeed subgraphs
- the graph after removing a node (and associated edges) is a subgraph
- the max degree is less than or equal to the size of any adjacency set
- the subgraph relation implies non-strict inequality of max degree
- disjoint and OK colorings can be unioned to form larger colorings
- the restriction of an OK coloring is OK
- the constant coloring on a graph with max degree 0 is OK and complete
- map restriction commutes with mapping over the values of a map

- various inversion an consistency lemmas about restrictions (if a key/value pair is in the restricted map, then the key is in the set we restricted by)
- given a 2-colorable graph a function that colors a vertex v with c_1 and all of its neighbors c_2 (where $c_1 \neq c_2$ is a valid coloring on the graph and a complete coloring on the neighborhood with v
- finding a vertex with a given degree d is decidable

4.2 Phase I Lemmas

We first want to establish that the neighborhood of a vertex in a 3-colorable graph is 2-colorable. This seems simple, but due to the definition of a mapping, this is quite challenging. We first show that a 3-colorable neighborhood and vertex v will induce a 3-coloring that only uses 2 colors. We need to show that this coloring can be transformed into a 2-coloring. Even though the coloring doesn't change, the information at the type level changes, we go from `coloring_complete p g f` to `coloring_complete (S.remove ci p) (neighborhood g v) (restrict_on_nbd f g v)`, meaning we can use the fact that f is uses one less color and it is a complete coloring on the neighborhood.

```
Lemma nbd_Sn_colorable_n : forall (g : graph) (f : coloring) (p : colors) (n : nat),
    coloring_complete p g f ->
    n_coloring f p (S n) ->
    forall v ci, M.find v f = Some ci ->
            n_coloring (restrict_on_nbd f g v) (S.remove ci p) n
         /\ coloring_complete (S.remove ci p) (neighborhood g v) (restrict_on_nbd f g v).
```

We also show the contrapositive of this statement (this is immediate since $A \to B$ implies $\neg B \to \neg A$ constructively). This shows that if the neighborhood is not colorable, then the graph is not 3-colorable. We can use this later to show that if our 2-coloring algorithm fails, then the graph is not 3-colorable. This allows us to avoid using the 2-coloring given to us by Coq in showing that the 2-coloring function is valid. This will also allow us to prove robustness of the algorithm.

This lemma allows us to 2-color the neighborhood of any vertex in a 3-colorable graph. We also need some more lemmas for Phase I of the algorithm. We need to show that removing a high degree vertex reduces the size of the graph by 1, and that removing its neighborhood reduces the size of the graph by at least $\sqrt{K}$.

We also want to show that high-degree vertex selected will not be adjacent to each other. This would imply that each neighborhood we select is entirely disjoint. This will also imply that so we can remove $\sqrt{K}$ new vertices at each step. We show that this means the process will terminate and leave the remaining graph with maximum degree $\sqrt{K} - 1$.

We also want to build properties about combining colorings of neighborhoods together. We can combine them individually to reform the entire graph. In particular, we want to use this fact to show that the partial colorings of neighborhoods will form a valid coloring when combined together. We can show this works for any two disjoint partial colorings and apply induction for the whole process.

4.3 Phase II Lemmas

5 Algorithm Implementation and Theorem Proofs

We have described the phases of the algorithm is pseudocode, but we now must translate this into Coq. We will do this in a manner corresponding to our lemmas. We will apply these lemmas to our functions and prove correctness and robustness.

5.1 Phase I

We postulate the existence of a 2-coloring function that takes a graph g, a vertex v, two colors c_1 and c_2 and 2-colors the neighborhood of v, or fails. Note that the function would not take any proof information, we would have to separately prove that if the coloring failed then it would imply the neighborhood of v is not 2-colorable.

```
Definition two_color_nbd (g : graph) (v : node) (c1 c2 : positive) : option coloring.
Admitted.
```

The following theorem states the completeness of the 2-coloring algorithm with distinct colors c_1, c_2 on a neighborhood of a vertex v if there existed some coloring m such that the vertex was assigned some distinct color c_3 while m completely colored the rest of the graph.

```
Lemma two_color_nbd_complete : forall (g : graph) (v : node) c1 c2 c3,
    c1 <> c2 ->
    c1 <> c3 ->
    c2 <> c3 ->
    no_selfloop g ->
    undirected g ->
    M.In v g ->
    (exists m, M.find v m = Some c3 /\ coloring_complete (SP.of_list [c1;c2;c3]) g m) ->
    coloring_complete (SP.of_list [c1;c2])
                      (subgraph_of g (nodes (neighborhood g v))) (two_color_nbd g v c1 c2).
```

5.2 Phase II

We fully define the second phase of the coloring. First we write a function to iteratively extract a vertex of a given degree and remove it from a graph. Separately we also prove that if the degree d that was given is the max degree then `extract_vertices_deg` exhausts all the vertices of max degree and the graph returned has maximum degree of one less.

```
Function extract_vertices_deg (g : graph) (d : nat) {measure M.cardinal g}
    : list (node * graph) * graph :=
  match extract_deg_vert_dec g d with
  | inl v =>
      let (l, g') := extract_vertices_deg (remove_node (`v) g) d in
      ((`v, g') :: l, g')
  | inr _ => (nil, g)
  end.
```

```
Function phase2 (g : graph) {measure max_deg g} : coloring * graph :=
  match max_deg g with
  | 0 => (constant_color (nodes g) 1, (@M.empty _))
  | S n => let (ns, g') := extract_vertices_deg g (S n) in
           let ns' := SP.of_list (map fst ns) in
           let (f', g'') := phase2 g' in
           (Munion (constant_color ns' (Pos.of_nat (S n))) f', g'')
  end.
```

6 Use of automated theorem proving

We make extensive use of *CoqHammer*, an automated reasoning tool for dependent type theory. It comes with two main commands, `sauto` [3] and `hammer` [4]. `sauto` perform as a general intuitionistic proof search, while `hammer` translates the current goal and a selection of hypothesis (identified with k-nearest neighbors) into first-order logic then calls external theorem provers such as Z3, Vampire and CVC4. When the external theorem prover succeeds then CoqHammer attempts to minimize the dependencies then reconstructs the proof inside Coq by passing these lemmas to `sauto`. Note that the external theorem provers can be untrusted but this does not violate Coq's safety properties because the proof has to be reconstructed inside Coq. Sometimes the external theorem provers succeed but proof reconstruction fails. This is usually resolved by manually proving the goal with the returned dependencies.

We give some demonstrations of the tool below. The statement we wish to prove is the transitivity of the subgraph relation:

```
Lemma subgraph_trans : forall g g' g'',
  is_subgraph g g' ->
  is_subgraph g' g'' ->
  is_subgraph g g''.
Proof. best.
```

The user proceeds by writing the tactic `best`, which then finishes the proof with the following message.

```
Replace the `best` tactic with:
sfirstorder

No more subgoals.
```

Although this lemma isn't hard to prove by manually, being able to quickly check if something is immediately provable or if it can be proved with some additional lemmas identified with `hammer` allows us to spend less time in tedious details of a proof and focus on larger arguments. In addition, the proof scripts become more maintainable, since the proof search still works when hypothesis are adjusted or even the order of conjunctions and so on are changed.

7 Conclusion

Due to time constraints we were not able to fully prove Wigderson's algorithm in Coq, however we have built up a theory of graphs from scratch whereby it's simply a matter of effort to state and prove missing pieces, such as the implementation of a 2-coloring algorithm and its proof of correctness. Most of our admissions are well-known or already verified such as two 2-coloring algorithm. We have also defined both phases of the algorithm and proved the majority of lemmas relating to each phase. This is a useful foundation for graph theory and verifying graph algorithms using the standard library Coq. While many graph theoretical results have been proven in Coq, this serves as a case study to understand formally building properties which can be applied to abstract data structures and algorithms.

References

[1] Appel, A et al. (2021) Verified Functional Algorithms. In Software Foundations. Volume 3. Version 1.5.1. https://softwarefoundations.cis.upenn.edu/vfa-current/index.html

[2] Bertot, Y. (2006). Coq in a Hurry. doi:10.48550/ARXIV.CS/0603118

[3] Czajka, Ł. (2020). Practical Proof Search for Coq by Type Inhabitation. Peltier & V. Sofronie-Stokkermans, Automated Reasoning (28–57). Cham: Springer International Publishing.

[4] Czajka, Ł., & Kaliszyk, C. (2018). Hammer for Coq: Automation for Dependent Type Theory. Journal of Automated Reasoning, 61. doi:10.1007/s10817-018-9458-4

[5] Delahaye, D. (2000). A Tactic Language for the System Coq. 85–95. doi:10.1007/3-540-44404-1_7

[6] Kawarabayashi, K.-I., & Thorup, M. (2017). Coloring 3-Colorable Graphs with Less than $n^{1/5}$ Colors. J. ACM, 64(1). doi:10.1145/3001582

[7] Wigderson, A. (1982). A New Approximate Graph Coloring Algorithm. Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing, 325–329. San Francisco, California, USA. doi:10.1145/800070.802207