Math.Comput.Sci. (2015) 9:23–39
DOI 10.1007/s11786-014-0183-z

A Graph Library for Isabelle

Lars Noschinski

Received: 5 September 2013 / Accepted: 28 January 2014 / Published online: 28 June 2014
© Springer Basel 2014

Abstract In contrast to other areas of mathematics such as calculus, number theory or probability theory, there is currently no standard library for graph theory for the Isabelle/HOL proof assistant. We present a formalization of directed graphs and essential related concepts. The library supports general infinite directed graphs (digraphs) with labeled and parallel arcs, but care has been taken not to complicate reasoning on more restricted classes of digraphs. We use this library to formalize a characterization of Euler Digraphs and to verify a method of checking Kuratowski subgraphs used in the LEDA library.

Keywords Graph theory · Isabelle · HOL · Euler · Kuratowski

Mathematics Subject Classification (2010) Primary 05C20; Secondary 05C45

1 Introduction

Modern proof assistants usually include a library covering many areas of mathematics. The formalizations in these libraries serve as a common basis for proof developments in this area, enabling the user to reuse results proved by others, and also keeping the developments aligned. For graph theory, despite its many applications, the situation is a different: no standard formalization has evolved yet.

In the HOL family of proof assistants, there have been a number of formalizations of specific graph-theoretic algorithms and results, but none has evolved into a general library. The formalization of Dijkstra' shortest path algorithm in Isabelle/HOL [17] covers only the essentials needed for this algorithm. Wong's [23] work on railway networks in HOL covers walks, trails, paths, degrees, union and operations to insert or delete vertices and arcs. However, we are not aware of any work building upon it. Chou [5] formalizes undirected graphs in HOL, including concepts like walks, paths, trails, reachability, connectedness, bridges, and rooted trees, as well as operations to combine graphs while preserving tree-ness. A similar set of notions is formalized in Coq by Duprat [7], using an inductive graph definition. The NASA PVS libraries cover both undirected and directed graphs (digraphs), but do not consider parallel arcs [4]. Mizar contains a comparatively large amount of graph theory, which is however

L. Noschinski (✉)
Lehrstuhl XXI, Institut für Informatik, Technische Universität München,
Boltzmannstr. 3, 85748 Garching, Germany
e-mail: noschinl@in.tum.de

split on six different formalizations of graphs. This covers formalizations of the algorithms by Dijkstra, Prim and Ford-Fulkerson, as well as a characterization of undirected Euler graphs [14]. Planar graphs have been formalized in the Flyspeck project [15] and in the proof of the Four-Color-Theorem [9], using specialized graph representations.

In this article, we present a formalization of directed graphs in Isabelle/HOL. This is the first attempt at a general graph library for this proof assistant. In part, our formalization resembles the one chosen by Wong [23] for his work on railway networks. We differ from his work in providing a more abstract setting, which enables re-use in different applications. In particular, it allows the user to choose a graph implementation depending on the needs of the application.

The contribution of our work is a comprehensive set of basic graph theory vocabulary. The library is general enough to reason about all common classes of directed graphs, including those with parallel and labeled arcs. This work has been motivated by the verification effort on the LEDA graph library [13]. The LEDA library contains a number certifying graph algorithms and our library has been successfully used to verify the certificates generated by some of these algorithms [1,20].

A major goal of our formalization is to enable convenient reasoning about digraphs, including good proof automation. For this reason, we emphasize the choice of an appropriate representation of digraphs and walks. For efficient proving it is important that the proof heuristics can solve frequently occurring proof obligations automatically. Therefore, we stress where the aim of better automation leads to a particular formulation. As a particular challenge, graph theory distinguishes between simple digraphs (without parallel arcs) and multi-digraphs (with parallel arcs). Textbooks often prefer the former, as they are easier to handle. Many applications, on the other hand, use parallel arcs. Our goal is that the option to work with multi-digraphs should not increase the proof effort needed on digraphs.

Based on a representation of directed multi-graphs, we have formalized the concepts of finite, loop-free and simple digraphs, walks, paths, cycles, (symmetric) reachability, isomorphisms, degree of vertices, (induced) subgraphs, (strong) connectedness and connected components, and trees and spanning trees. We also provide operations for the union of graphs, adding and removing arcs and many lemmas to combine these concepts. As case studies, we have formalized a characterization of directed Euler trails and the graph-theoretic part of a verification of the non-planarity certificates from the LEDA library. Moreover, this library has been used to formalize cycle checking algorithms in [8] and in the verification of checkers for certifying algorithms in the LEDA graph library. The library is available in the Archive of Formal Proofs [18].[1] It consists of around 7000 lines of Isabelle theories.

The paper is organized as follows: Sect. 2 gives a brief overview of Isabelle. Section 3 describes the representation of digraphs and discusses possible alternatives. Section 4 contains a selection of formalized concepts and Sects. 5 and 6 present two case studies about Euler digraphs and Kuratowski subgraphs. Section 7 concludes with a discussion of our results.

2 Background

2.1 Isabelle

Isabelle/HOL is a proof assistant based on polymorphic higher-order logic [16]. We use an *italic* font for variables and sans-serif for constants. HOL types include type variables ($\alpha, \beta, \ldots$), function types ($\alpha \to \beta$) and sets (α *set*). A term t of type α is written as $t :: \alpha$ and function application as $f\ t$. Record types can be declared with the `record` command. We write record literals as tuples and the selector functions have the same name as the field. The logical connectives follow the standard notation from mathematics. Lemmas are denoted $[\![P_1; \ \ldots\ ; P_n]\!] \implies Q$ where $P_1, \ldots, P_n$ are premises and Q is the conclusion.

The two proof styles commonly used in Isabelle are rewriting and natural deduction. Rewriting is mostly performed with the simplifier, which uses conditional equations and discharges the conditions again by simplification.

[1] At the time of this writing, the release version of the Archive of Formal Proofs includes an older version of this library. The version described here is available at http://www21.in.tum.de/~noschinl/Graph_Theory-2014.

Natural deduction rules are applied by hand or by one of various classical reasoning tools. Isabelle's most popular proof methods combine both styles. The proof tools can be configured to solve common goals automatically by declaring appropriate rules as simplification rules (for rewriting) or introduction or elimination rules (for natural deduction).

2.2 Locales

We use locales [2] to formalize graph theory independent from a concrete implementation. Locales allow us to structure formal proof developments by providing a local collection of constants and assumptions, as in the example below:

locale $\mathsf{L} =$ fixes $c :: \tau$ assumes $P\ c$
locale $\mathsf{K} =$ fixes $d :: \tau$ assumes $Q\ d$

The first command declares a new locale L with a local constant c of type τ and introduces a local assumption $P\ c$. All type variables occurring in τ are also fixed locally. The number of constants and assumptions is arbitrary. The fixed constants are called the parameters of the locale. Once defined, a locale can be extended with theorems and definitions, relying implicitly on the locale parameters and assumptions. We can also declare a locale inheriting from another locale:

locale $\mathsf{L'} = \mathsf{L} +$ fixes $c' :: \tau'$ assumes $P'\ c'$

The parameters of $\mathsf{L'}$ are c and c', the assumptions $P\ c$ and $P'\ c'$. $\mathsf{L'}$ also inherits all the theorems and definitions from L.

It is possible to access the theorems and definitions in a locale from the outside. In this case, the locale assumptions become explicit assumptions of the theorems and locale parameters (and type variables) are universally quantified. For constants defined in the locale, the locale parameters are now passed explicitly. To improve clarity, we will denote such parameters as a subscript.

To apply a locale to concrete parameters, it can be interpreted. An interpretation of L instantiates the parameter c with a concrete value t. Moreover, we can give arbitrary equations $s = s'$ to refine the interpretation. After proving that $P\ t$ and the additional equations hold, all the lemmas of L become available as in the locale, with c instantiated to t and s rewritten to s'. An embedding is an interpretation of a locale in another locale. The command

sublocale $\mathsf{K} \subseteq \mathsf{L}\ t$ where $s = s'$

embeds K into L, instantiating d with t and replacing s by s'. The terms t, s, s' may contain the parameters of K.

3 Representation

3.1 Digraphs

In this section, we present our representation of digraphs and discuss alternatives. For graph theoretic terms and definitions, we mostly follow Bang-Jensen and Gutin [3]. We will note where our definitions deviate. A digraph consists of vertices V and arcs A. An *arc* connects two vertices, going from the *tail* (or source) to the *head* (or target). Common representations of arcs are $A \subseteq V \times V$ for simple digraphs or $A \subseteq V \times L \times V$ for multi-digraphs, where L is some set of labels.

We represent a digraph as a 4-tuple (V, A, t, h), where A is a set of abstract values and $t, h : A \to V$ map an arc to its tail resp. head. This formulation is also found in some textbooks [22].

Definition 1 *(Type of directed graphs)*

record $(\beta, \alpha)\ dg =$ $\mathsf{verts} :: \beta\ set,$ $\mathsf{arcs} :: \alpha\ set,$ $\mathsf{tail} :: \alpha \Rightarrow \beta,$ $\mathsf{head} :: \alpha \Rightarrow \beta$

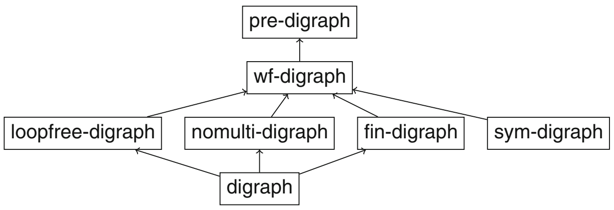

Fig. 1 Hierarchy of digraph locales. The *arrows* describe an inheritance relationship

We write V_G, A_G, $u_{\mathsf{t},G}$, and $u_{\mathsf{h},G}$ for $\mathsf{verts}\ G$, $\mathsf{arcs}\ G$, $\mathsf{tail}\ G\ u$, and $\mathsf{head}\ G\ u$. Two graphs are *compatible* if the projection functions are extensionally equal.[2]

$$\mathsf{compatible}\ G\ H = (\mathsf{tail}\ G = \mathsf{tail}\ H \wedge \mathsf{head}\ G = \mathsf{head}\ H)$$

We always use α and β for the types of arcs and vertices, respectively.

By choosing appropriate values for α, tail, and head, we can select the representation appropriate to a problem domain. This includes the representations with $A \subseteq V \times V$ or $A \subseteq V \times L \times V$ above. Another example are digraphs with $\beta = \alpha = \mathbb{N}$, which sometimes occur in the verification of imperative programs (where the number is an index into some data structure or heap). The latter representation has been used in the verification of C programs dealing with graphs [19].

We use locales to structure the different classes of digraphs (Fig. 1). The most basic class are well-formed digraphs. A digraph is well-formed if the endpoints of all arcs are vertices of the graph. Unless noted otherwise, we state lemmas in the wf-digraph locale and define functions and predicates in the pre-digraph locale. As this locale does not have any assumptions, the definitions of these functions can be used even if well-formedness has not yet been proved.

Definition 2 *(Well-formed Graphs)*

$\texttt{locale}\ \mathsf{pre\text{-}digraph} = \texttt{fixes}\ \mathsf{G} :: (\beta, \alpha)\ dg$
$\texttt{locale}\ \mathsf{wf\text{-}digraph} = \mathsf{pre\text{-}digraph} + \texttt{assumes}\ \forall a \in \mathsf{A}_G.\ a_{\mathsf{t},G} \in \mathsf{V}_G\ \texttt{and}\ \forall a \in \mathsf{A}_G.\ a_{\mathsf{h},G} \in \mathsf{V}_G$

In contrast to [3], we do not exclude the null graph (i.e. an empty set of vertices is allowed). We have found this slightly more convenient in our case studies and it makes digraphs closed under deletion of vertices or intersection. Harary and Read [10] discuss the merits of allowing this graph.

Often digraphs are required to be finite (i.e., the sets of arcs and vertices are finite), loop-free (i.e. head and tail of each arc are distinct) or free of parallel arcs. We also sometimes model undirected graphs as symmetric graphs (i.e. the relation described by the arcs is symmetric). For each of these properties we introduce a separate locale. The locale digraph combines the properties of a digraph class commonly used in textbooks. Locales for other combinations can be easily defined. As a general rule, we put lemmas in the most general locale possible, so that the more specialized locales inherit these.

Note that the generality of the arc type might be problematic if a proof requires adding an arc and the values of α are exhausted. As an pathological example, consider $\alpha = unit$, the type with a single element (), and the graph with $V = \{a, b\} :: \beta$ and $A = \{()\} :: unit$ where $a \neq b$ and the arc () has tail a and head b. This graph cannot be extended to a cycle. To avoid such cases, one needs to place additional restrictions on α, tail and head.

3.2 Specializing the Representation

In this section, we show how we can regain the simple representations we discarded in the previous section by specialization of our definition. In particular, we explain how we can use locales to eliminate the pre-digraph

[2] If we restrict ourselves to fixed vertex type β, we can use Isabelle's type class mechanism to define the projection functions for each type α once and for all, instead of defining them on a per-graph basis.

projection functions, which are not needed for many representations. We demonstrate our approach with the usual representation of graphs without parallel arcs.

Definition 3 *(Pair Digraphs)*

$\texttt{record}\ \beta\ \textit{pair-dg} = \textsf{pverts} :: \beta\ \textit{set},\ \textsf{parcs} :: (\beta \times \beta)\ \textit{set}$

$\texttt{locale}\ \textsf{pair-wf-digraph} = \texttt{fixes}\ G :: \beta\ \textit{pair-dg}$
$\quad \texttt{assumes}\ \forall e \in \textsf{parcs}\ G.\ \textsf{fst}\ e \in \textsf{pverts}\ G\ \texttt{and}\ \forall e \in \textsf{parcs}\ G.\ \textsf{snd}\ e \in \textsf{pverts}\ G$

Pair digraphs can be embedded in digraphs of type $(\beta, \beta \times \beta)\ dg$:

$$\overline{G} = (\textsf{pverts}\ G, \textsf{parcs}\ G, \textsf{fst}, \textsf{snd})$$

We embed the newly defined locale into wf-digraph:

$\texttt{sublocale}\ \textsf{pair-wf-digraph} \subseteq \textsf{wf-digraph}\ \overline{G}\ \texttt{where}$
$\quad \textsf{tail}\ \overline{G} = \textsf{fst}\ \texttt{and}\ \textsf{head}\ \overline{G} = \textsf{snd}\ \texttt{and}\ \textsf{V}_{\overline{G}} = \textsf{pverts}\ G\ \texttt{and}\ \textsf{A}_{\overline{G}} = \textsf{parcs}\ G$

The additional equations are used by the locale mechanism to rewrite the theorems embedded from wf-digraph and remove every occurrence of the selector functions of the *dg* in these theorems. This is important for automation: without it, the embedded lemmas are often cumbersome to use. For example, consider an introduction rule $[\![a \in \textsf{A}_G]\!] \implies a_{\textsf{t},G} \in \textsf{V}_G$, stating that the tail of an arc is a vertex of the digraph. It would have to be rewritten manually before we can apply it to a goal of the form $[\![\ldots]\!] \implies \textsf{fst}\ a \in \textsf{V}_G$.

Due to this rewriting, the embedded definitions of some functions do not depend on G anymore. An example is the predicate cas, which tests whether a sequence of arcs is consistent, i.e., their endpoints fit together:

Definition 4 *(Consistent Arc Sequence)*

$$\textsf{cas}_G\ u\ []\ v = (u = v)$$
$$\textsf{cas}_G\ u\ (a\#as)\ v = (a_{\textsf{t},G} = u \wedge \textsf{cas}_G\ (a_{\textsf{h},G})\ as\ v)$$

In pair-wf-digraph, the second equation is rewritten to:

$$\textsf{cas}_{\overline{G}}\ u\ (a\#as)\ v = (\textsf{fst}\ a = u \wedge \textsf{cas}_{\overline{G}}\ (\textsf{snd}\ a)\ as\ v)$$

So, if G and H are of type $\beta\ \textit{pair-dg}$, then $\textsf{cas}_{\overline{G}}$ and $\textsf{cas}_{\overline{H}}$ are the same function, denoted by two different terms. We replace such functions by a new constant without the digraph parameter. This makes proofs involving these functions easier for automated proof tools.

If we wanted to introduce a more general notion of graphs later on, we could use a similar approach to identify concepts on digraphs with the more general concepts. A candidate for such a step would be mixed graphs [3] as a common basis for directed and undirected graphs. This would allow us to share common concepts between these different types of graphs.

4 Operations and Properties

In this section, we present a number of basic concepts we have formalized.

4.1 Walks and Related Concepts

In textbooks, a walk is a finite alternating sequence $u_1 a_1 u_2 a_2 \ldots u_{k-1} a_{k-1} u_k$ of vertices u_i and arcs a_i of the digraph such that a_i has tail u_i and head u_{i+1}. We omit the vertices and represent a walk as list of arcs. The expression $\mathsf{awalk}_G\ u\ p\ v$ denotes that p is a walk form u to v. The predicate cas (Definition 4) separates the consistency from the membership properties. This is useful for proofs involving compatible graphs, as consistency of a walk is preserved over such graphs.

Definition 5 *(Walk, Vertices of a Walk)*

$$\mathsf{awalk}_G\ u\ p\ v = (u \in \mathsf{V}_G \wedge \mathsf{set}\ p \subseteq \mathsf{A}_G \wedge \mathsf{cas}_G\ u\ p\ v)$$

This is equivalent to the following recursive definition we use for the simplifier:

$$\begin{aligned} \mathsf{awalk}_G\ u\ []\ v &= (u = v) \wedge u \in \mathsf{V}_G \\ \mathsf{awalk}_G\ u\ (a\#as)\ v &= a \in \mathsf{arcs}_G \wedge (u = a_{\mathsf{t},G}) \wedge \mathsf{awalk}_G\ a_{\mathsf{h},G}\ as\ v \end{aligned}$$

The function $\mathsf{aw\text{-}verts}_G$ computes the vertices of a walk. To deal with the empty list, we need to give a start vertex explicitly.

$$\begin{aligned} \mathsf{aw\text{-}verts}_G\ u\ [] &= u \\ \mathsf{aw\text{-}verts}_G\ u\ (a\#as) &= a_{\mathsf{t},G}\ \#\ \mathsf{aw\text{-}verts}_G\ a_{\mathsf{h},G}\ as \end{aligned}$$

We abbreviate $\mathsf{last}\ (\mathsf{aw\text{-}verts}_G\ u\ p)$ by $\mathsf{awl}_G\ u\ p$.

All walks consisting of a single vertex are represented by the empty list. This caused no problems in our experiments, as the awalk predicate mentions the start and end vertices of a walk.

Arc lists have a nice concatenation property: the concatenation of two lists is a walk if and only if both lists are already walks on their own and there exists a common endpoint. For the rewrite rule, we denote this common endpoint by awl, instead of an existential quantifier, as the simplifier's ability to deal with quantifiers is limited.

Lemma 6 *For the concatenation of two walks, we have the following equality:*

$$\mathit{awalk}_G\ u\ (p\ @\ q)\ v = (\mathit{awalk}_G\ u\ p\ (\mathit{awl}_G\ u\ p) \wedge \mathit{awalk}_G\ (\mathit{awl}_G\ u\ p)\ q\ v)$$

To remove the awl *terms, we use the conditional equation*

$$[\![\mathit{awalk}_G\ u\ p\ v]\!] \Longrightarrow \mathit{awl}_G\ u\ p = v. \tag{1}$$

Regarding proof automation: For the rule (1), note that rewriting will loop if v is already of the form $\mathsf{awl}_G\ u\ p$. We avoided this by implementing a general simplification procedure (simproc) which prevents rewriting in this case. A simproc is a plugin for Isabelle's simplifier, which can compute rewrite rules for a given pattern on demand. For that, we define a new constant $\mathsf{NOMATCH}\ ::\ \tau \to \tau \to \mathit{bool}$ with $\mathsf{NOMATCH}\ t\ p = \mathsf{True}$ and configure the simplifier to not rewrite the arguments of this constant. If the simproc encounters a $\mathsf{NOMATCH}$ term, it tests whether the term t matches the pattern p. This is done on a purely syntactic level, so the procedure, in contrast to normal rewrite rules, is able to distinguish between terms which are logically equal. If v does not match p, the simproc returns the equation $\mathsf{NOMATCH}\ t\ p = \mathsf{True}$, otherwise it fails, preventing the simplifier from rewriting the term.

This allows us to state (1) in a form safe for rewriting:

$$[\![\mathsf{awalk}_G\ u\ p\ v;\ \ \mathsf{NOMATCH}\ (\mathsf{awl}_G\ u\ p)\ v]\!]\implies \mathsf{awl}_G\ u\ p = v.$$

The order of the assumptions is important: When applying this rule, the simplifier first tries to prove $\mathsf{awalk}_G\ u\ p\ v$. This causes v to be instantiated. If now $\mathsf{awl}_G\ u\ p$ matches v, the simplifier fails to prove the assumptions of this rule and the rule is not applied. The use of a purely syntactic constant to guide the simplifier has been motivated by ACL2's rewriter [11].

There are two reasons to choose arc lists instead of alternating sequences: for alternating sequences, concatenation either removes the last vertex of the first sequence or the first vertex of the second sequence. For example, $(u, a, v, b, y) = (u, a, v) \frown (x, b, y)$ being a walk does not imply that (u, a, v) and (x, b, y) are walks, so Lemma 6 does not hold without additional knowledge about the components. A similar problem occurs with all representations that include (at least) one explicit start or end vertex. The other reason is that finite arc sequences map to Isabelle lists in a natural way, which allows us to use Isabelle's well-developed theory of lists, including the various induction schemes.

Like alternating sequences, the representation of walks as a list of vertices lacks the nice concatenation property of arc lists. For this reason, we usually found it more convenient to reason with arc lists than with vertex lists. Nevertheless, our formalization also includes some basic facts about walks as vertex lists and their relation to arc lists.

In his work, Wong [23] also represents walks as list of arcs, but excludes the single-vertex walk, deviating from most textbooks. This sometimes simplifies the formalization, since the vertices of a walk can be denoted without an explicit start vertex, but has the disadvantage that lemmas about the decomposition of walks become more complex.

Apart from walks, we also have formalized trails, paths and cycles as walks with additional properties:

Definition 7 *(Trails, Paths, and Cycles)*

$$\begin{aligned}
\mathsf{trail}_G\ u\ p\ v &= (\mathsf{awalk}_G\ u\ p\ v \wedge \mathsf{distinct}\ p)\\
\mathsf{apath}_G\ u\ p\ v &= (\mathsf{awalk}_G\ u\ p\ v \wedge \mathsf{distinct}\ (\mathsf{aw\text{-}verts}_G\ u\ p))\\
\mathsf{cycle}_G\ p &= (\exists u.\ \mathsf{awalk}_G\ u\ p\ u \wedge p \neq [] \wedge \mathsf{distinct}\ (\mathsf{tl}\ (\mathsf{aw\text{-}verts}_G\ u\ p)))
\end{aligned}$$

The function tl removes the first element of a list and $\mathsf{distinct}$ states that the elements of a list are distinct.

We follow Diestel [6] and Volkmann [22] here and allow cycles consisting of a single arc, i.e. loops. Bang-Jensen and Gutin [3] require at least two arcs.

It is often necessary to relate the arcs and the vertices of a walk. The following lemma serves as the basis for our lemmas to split a walk based on a vertex property, for example to shorten a walk to a path by removing parts where the vertices are not distinct.

Lemma 8 *If $\mathsf{awalk}_G\ u\ p\ v$ and $\mathsf{aw\text{-}verts}_G\ u\ p = xs\ @\ [y]\ @\ ys$ hold, then*

$$\begin{aligned}
&\exists q\ r.\ \mathsf{awalk}_G\ u\ q\ y \wedge \mathsf{awalk}_G\ y\ r\ v \wedge p = q\ @\ r\ \wedge\\
&\mathsf{aw\text{-}verts}_G\ u\ q = xs\ @\ [y] \wedge \mathsf{aw\text{-}verts}_G\ y\ r = [y]\ @\ ys
\end{aligned}$$

If we have a walk going from one set of vertices to another, we can extract the arc which marks the transition with the following lemma:

Lemma 9 *Let P, Q be predicates on vertices. We have:*

$$\begin{aligned}
&[\![\mathsf{awalk}_G\ u\ p\ v;\ \ p \neq [];\ \forall w \in \mathsf{set}\,(\mathsf{aw\text{-}verts}_G\ u\ p).\ P\ w \vee Q\ w;\ \ P\ u;\ \ Q\ v]\!]\\
&\quad\implies \exists a \in \mathsf{set}\ p.\ P\ a_{t,G} \wedge Q\ a_{h,G}
\end{aligned}$$

If we are only interested in whether a vertex v is reachable from another vertex u, an inductively defined relation is more convenient to use in proofs than the proposition $\exists p.\ \mathsf{awalk}_G\ u\ p\ v$. The basic relation here is the adjacency relation $\mathsf{adj}_G :: (\beta \times \beta)\ \mathit{set}$, also written as $u \rightarrow v$. One can use the usual operations on relations to define various reachability properties. For reflexive reachability relations, we need to restrict the domain to V_G. We use an inductively defined predicate rtrancl-on equivalent to $\mathsf{adj}_G^* \cap \mathsf{V}_G \times \mathsf{V}_G$ (on well-formed digraphs). This is also written as $u \rightarrow^* v$. The library contains the usual transitivity rules as well as induction rules progressing left-to-right and right-to-left:

$$[\![u \rightarrow^* v;\ u \in \mathsf{V}_G \longrightarrow P\ u;\ (\forall x\ y.\ u \rightarrow^* x \wedge x \rightarrow y \wedge P\ x) \longrightarrow P\ y]\!] \Longrightarrow P\ v$$
$$[\![u \rightarrow^* v;\ v \in \mathsf{V}_G \longrightarrow P\ v;\ (\forall x\ y.\ x \rightarrow y \wedge y \rightarrow^* v \wedge P\ y) \longrightarrow P\ x]\!] \Longrightarrow P\ u$$

4.2 Operations and Properties of Graphs

Arcs can be inserted and removed with the add-arc and del-arc operations. This always yields a well-formed digraph. For the reasons given in Sect. 3.1, we do not provide a function to build an arc from two vertices for the abstract graph type. To do this, we need additional assumptions on the projection functions.

Definition 10 *(Adding and Removing Arcs)*

$$\mathsf{add\text{-}arc}_G\ a = (\mathsf{V}_G \cup \{a_{\mathsf{t},G}, a_{\mathsf{h},G}\}, \mathsf{A}_G \cup \{a\}, \mathsf{tail}\ G, \mathsf{head}\ G)$$
$$\mathsf{del\text{-}arc}_G\ a = (\mathsf{V}_G, \mathsf{A}_G - \{a\}, \mathsf{tail}\ G, \mathsf{head}\ G)$$

Both functions are defined within pre-digraph, as they relate to a single graph. Functions taking more than one graph are defined outside of the pre-digraph locale. This means we need to mention the locale predicates explicitly.

The union of two digraphs is the digraph which has the arcs and vertices of both digraphs.

Definition 11 *(Union)*

$$\mathsf{union}\ G\ H = (\mathsf{V}_G \cup \mathsf{V}_H, \mathsf{A}_G \cup \mathsf{A}_H, \mathsf{tail}\ G, \mathsf{head}\ G)$$
$$\mathsf{Union}_G\ \mathcal{G} = \left(\bigcup_{G \in \mathcal{G}} \mathsf{V}_G, \bigcup_{G \in \mathcal{G}} \mathsf{A}_G, \mathsf{tail}\ G, \mathsf{head}\ G\right)$$

Note that for union we arbitrarily choose the projection functions of G, so the definition yields useful results only if G and H are compatible. The union of a set of digraphs $\mathcal{G}$ is defined w.r.t. a graph G, so the result is always compatible to G, even if $\mathcal{G}$ is empty.

In [3], connected digraphs are defined in two steps, first referring to the underlying connected undirected graph, then referring to the associated strongly connected symmetric digraph. We leave out the intermediary and use the associated symmetric digraph to define symmetric reachability.

Definition 12 *(Underlying Undirected Graph)* For $G :: (\beta, \alpha)\ dg$, the underlying undirected graph of type $\beta\ \mathit{pair\text{-}dg}$ is:

$$\mathsf{mk\text{-}symmetric}\ G = (\mathsf{V}_G, \textstyle\bigcup_{a \in \mathsf{A}_G} \{(a_{\mathsf{t},G}, a_{\mathsf{h},G}), (a_{\mathsf{h},G}, a_{\mathsf{t},G})\})$$

For the graph properties we expressed as locales (cf. Fig. 1), mk-symmetric G satisfies at least the same properties as G.

Now, we define connected digraphs.

Definition 13 *(Connectivity and Strong Connectivity)*

$$\mathsf{strongly\text{-}connected}\ G = (\mathsf{V}_G \neq \varnothing \wedge \forall u, v \in \mathsf{V}_G.\ u \to^* v)$$
$$\mathsf{connected} = \mathsf{strongly\text{-}connected}\ (\mathsf{mk\text{-}symmetric}\ G)$$

Note that we do not consider the null graph to be (strongly) connected. This makes the decomposition of digraphs into connected components unique.

Definition 14 *(Subgraphs and Strongly Connected Components)* A subgraph G of H is a well-formed graph G whose vertices and arcs are contained in H. If G contains all arcs of H only connecting vertices of G, then G is an induced subgraph.

$$\mathsf{subgraph}\ G\ H = (\mathsf{V}_G \subseteq \mathsf{V}_H \wedge \mathsf{A}_G \subseteq \mathsf{A}_H \wedge\ \mathsf{wf\text{-}digraph}\ G \wedge \mathsf{wf\text{-}digraph}\ H \wedge \mathsf{compatible}\ G\ H)$$
$$\mathsf{induced\text{-}subgraph}\ G\ H = (\mathsf{subgraph}\ G\ H \wedge \mathsf{A}_G = \{a \in \mathsf{A}_H.\ \{a_{\mathsf{t},H}, a_{\mathsf{h},H}\} \subseteq \mathsf{V}_H\}$$

A strongly connected component is a maximal strongly connected subgraph:

$$\mathsf{sccs}\ G = \{H.\ \mathsf{induced\text{-}subgraph}\ H\ G \wedge \mathsf{strongly\text{-}connected}\ H\ \wedge$$
$$\neg(\exists H'.\ \mathsf{induced\text{-}subgraph}\ H'\ G \wedge \mathsf{strongly\text{-}connected}\ H' \wedge \mathsf{V}_H \subsetneq \mathsf{V}_{H'})\}$$

A classic result we proved for these properties is the unique decomposition of a symmetric digraph into strongly connected components:

Lemma 15 *For a symmetric digraph G, the decomposition into SCCs is unique:*

$$[\![S \subseteq \mathit{sccs}\ G;\ \mathit{Union}_G\ S = G]\!] \implies S = \mathit{sccs}\ G$$

4.3 Digraph Isomorphisms

For our digraph representation, an isomorphism consists of four functions: a mapping of the vertices, a mapping of the arcs and the new projection functions. For a given graph, the projection function can be derived from the vertex and arc mappings. However, as we discuss below, providing explicit projection functions often eases reasoning.

Definition 16 *(Digraph Isomorphism)*

$\texttt{record}\ (\beta, \alpha, \beta', \alpha')\ \mathsf{iso} = \mathsf{iso\text{-}verts} :: \beta \Rightarrow \beta',\ \ \mathsf{iso\text{-}arcs} :: \alpha \Rightarrow \alpha',$
$\mathsf{iso\text{-}tail} :: \alpha' \Rightarrow \beta',\ \ \mathsf{iso\text{-}head} :: \alpha' \Rightarrow \beta'$

The function $\mathsf{app\text{-}iso} :: (\beta, \alpha, \beta', \alpha')\ \mathit{iso} \Rightarrow (\beta, \alpha)\ \mathit{dg} \Rightarrow (\beta', \alpha')\ \mathit{dg}$ applies the homomorphism to a digraph. Here f ‘ A denotes the image of the set A under f.

$$\mathsf{app\text{-}iso}\ h\ G = (\mathsf{iso\text{-}verts}\ h\ ‘\ \mathsf{V}_G, \mathsf{iso\text{-}arcs}\ h\ ‘\ \mathsf{A}_G, \mathsf{iso\text{-}tail}\ h, \mathsf{iso\text{-}head}\ h)$$

We write $h \cdot_{\mathsf{G}} G$, $h \cdot_{\mathsf{A}} a$, or $h \cdot_{\mathsf{V}} v$ for the application of h to a graph G, arc a, or vertex v. The predicate iso expresses under which conditions h is an isomorphism between G and its image: It must be injective and preserve the graph structure.

$$\mathsf{iso}_G\ h = \big(\mathsf{inj\text{-}on}\ (\mathsf{iso\text{-}verts}\ h)\ \mathsf{V}_G \wedge \mathsf{inj\text{-}on}\ (\mathsf{iso\text{-}arcs}\ h)\ \mathsf{A}_G$$
$$\wedge \big(\forall a \in \mathsf{A}_G.\ h \cdot_{\mathsf{V}} a_{\mathsf{t},G} = (h \cdot_{\mathsf{A}} a)_{\mathsf{t},h \cdot_{\mathsf{G}} G} \wedge h \cdot_{\mathsf{V}} a_{\mathsf{h},G} = (h \cdot_{\mathsf{A}} a)_{\mathsf{h},h \cdot_{\mathsf{G}} G}\big)\big)$$

The isomorphism predicate only demands the preservation of structure on A_G, not on the universe of α. This is in accordance with usual mathematic notation, but has the side-effect that projection functions of the image are defined uniquely only on a subset of α'. However, it is often possible to give sensible projections for the whole universe (e.g. for pair digraphs) and using those eases reasoning.

We provide rewrite rules to reduce the basic selector functions and most other (non-boolean-valued) functions on $h \cdot_{\mathsf{G}} G$ to functions on G. For many predicates we provide introduction rules. In particular this includes walks, trails, reachability and connectedness.

An inverse of an isomorphism can be determined automatically. Note that the inverse is not unique, as we do not restrict the functions in an isomorphism to the domain of the graph.

Definition 17 *(Inverse Isomorphism)*

$$\mathsf{inv\text{-}iso}_G\ h = \Big((\mathsf{iso\text{-}verts}\ h)^{-1}_{\mathsf{V}_G}, (\mathsf{iso\text{-}arcs}\ h)^{-1}_{\mathsf{A}_G}, \mathsf{head}\ G, \mathsf{tail}\ G\Big)$$

For f injective on S, we denote by f_S^{-1} the inverse function of f on S (defined by the choice operator).

The inverse isomorphism is useful to define a set of automatically usable simplification rules. As an example, consider the function $\mathsf{in\text{-}deg}$ returning the number of incoming arcs for a vertex. For an isomorphism h, the following two (conditional) equations reduce $\mathsf{in\text{-}deg}_{h \cdot_{\mathsf{G}} G}$ to $\mathsf{in\text{-}deg}_G$:

$$u \in \mathsf{V}_G \implies \mathsf{in\text{-}deg}_{h \cdot_{\mathsf{G}} G}\ (\mathsf{iso\text{-}verts}\ h\ u) = \mathsf{in\text{-}deg}_G\ u \tag{2}$$

$$[\![u \in \mathsf{V}_{h \cdot_{\mathsf{G}} G};\ u = \mathsf{iso\text{-}verts}\ h\ v]\!] \implies \mathsf{in\text{-}deg}_{h \cdot_{\mathsf{G}} G}\ u = \mathsf{in\text{-}deg}_G\ v \tag{3}$$

As usual in rewriting, Isabelle's simplifier can apply an equation if the left hand side matches a subterm of the goal and it is able to discharge the assumptions. Fresh variables on the right hand side are not allowed.

So rule (2) can only be applied if the vertex is given in a very special form. The more general rule (3) is not suitable for the simplifier as the variable v does not occur on the left hand side.

The function $\mathsf{inv\text{-}iso}$ allows us to get rid of this variable:

$$[\![u \in \mathsf{V}_{h\ G}]\!] \implies \mathsf{in\text{-}deg}_{h \cdot_{\mathsf{G}} G}\ u = \mathsf{in\text{-}deg}_G\ (\mathsf{iso\text{-}verts}\ (\mathsf{inv\text{-}iso}\ h)\ u)$$

In particular, in combination with the other simplification rules we define for isomorphisms, this rule can be used whenever the simplifier can prove $u = \mathsf{iso\text{-}verts}\ h\ v$ for some v. In this case, it rewrites to the same right hand side as (2).

The rewrite rules for $\mathsf{in\text{-}arcs}$, $\mathsf{out\text{-}arcs}$, and $\mathsf{out\text{-}deg}$ use this strategy as well.

5 Euler Graphs

In this section, we present a characterization of directed Euler graphs. The undirected variant of this theorem is one of the basic results any introductory textbook covers. We demonstrate that our library allows for a nice proof. An Euler trail is a trail that contains each arc of a graph exactly once and touches every vertex. A graph that has an Euler trail is called an Euler graph.

Definition 18 *(Euler Trail)*

$$\mathsf{euler\text{-}trail}_G\ u\ p\ v = (\mathsf{trail}_G\ u\ p\ v \wedge \mathsf{set}\ p = \mathsf{A}_G \wedge \mathsf{set}\ (\mathsf{aw\text{-}verts}_G\ u\ p) = \mathsf{V}_G)$$

For connected graphs, the condition on the vertices can be dropped:

Lemma 19

$$[\![\mathit{connected}\ G]\!] \implies \mathit{euler\text{-}trail}_G\ u\ p\ v = (\mathit{trail}_G\ u\ p\ v \wedge \mathit{set}\ p = \mathsf{A}_G)$$

We give a formalization of the well-known characterization of Euler digraphs. A finite digraph has an Euler trail if and only if it is connected and either

- for all vertexes the in-degree equals the out-degree or
- there are two vertexes u and v such that the difference between in- and out-degree of u resp. v is -1 resp. 1 and for all other vertexes, the in-degree equals the out-degree.

In the first case, we get a closed Euler trail, in the second case an open Euler trail from u to v. We introduce the predicate arc-balanced, which allows us to shorten the degree condition to $\exists u\ v.\ \mathsf{arc\text{-}balanced}_G\ u\ \mathsf{A}_G\ v$. The functions in-arcs, out-arcs, in-deg and out-deg denote the set of incoming/outgoing arcs of a vertex, respectively its cardinality.

Definition 20 *(Arc Balance)*

$$\mathsf{arc\text{-}balance}_G\ w\ A = \mathsf{int}\ |\mathsf{in\text{-}arcs}_G\ w \cap A| - \mathsf{int}\ |\mathsf{out\text{-}arcs}_G\ w \cap A|$$

$$\begin{aligned}\mathsf{arc\text{-}balanced}_G\ u\ A\ v = \big(&\mathsf{if}\ u = v\ \mathsf{then}\ \forall w \in \mathsf{V}_G.\ \mathsf{arc\text{-}balance}_G\ w\ A = 0\\ &\mathsf{else}\ (\forall w \in \mathsf{V}_G - \{u, v\}.\ \mathsf{arc\text{-}balance}_G\ w\ A = 0) \wedge\\ &\quad \mathsf{arc\text{-}balance}_G\ u\ A = -1 \wedge \mathsf{arc\text{-}balance}_G\ v\ A = 1\big)\end{aligned}$$

We formalized the full characterization, but will only discuss the existence of Euler trails here. The proof of the other direction is straightforward. We consider closed Euler trails first.

Theorem 21 *(Closed Euler Trail) For a finite digraph G:*

$$[\![\mathit{connected}\ G;\ \forall u \in \mathit{V}_G.\ \mathit{in\text{-}deg}_G\ u = \mathit{out\text{-}deg}_G\ u]\!] \implies \exists u\ p.\ \mathit{euler\text{-}trail}_G\ u\ p\ u$$

Proof Note that the degree condition of this lemma is equivalent to

$$\forall u \in \mathsf{V}_G.\ \mathsf{arc\text{-}balance}_G\ u\ \mathsf{A}_G = 0 \tag{4}$$

The proof given by Bang-Jensen and Gutin[3] is constructive: it starts with an empty trail and proves that a trail can always be extended until it contains all arcs.[3] The existence of an empty trail is trivial (the empty digraph is not connected). The extensibility argument is inductive; we model this as an induction on the number of arcs not in the trail. For the induction step, note that trails are always balanced, i.e.

$$[\![\mathsf{trail}_G\ u\ p\ v]\!] \implies \mathsf{arc\text{-}balanced}_G\ u\ (\mathsf{set}\ p)\ v \tag{5}$$

We consider two cases. If the end vertices of the trail are distinct, (4) and (5) guarantee the existence of an incident arc. If the trail is closed, we need to find an arc that "touches" the trail (i.e., has a common vertex with the trail, but is not part of it). Then we rotate the trail, such that the common vertex is at both ends and attach the arc.

Such an arc exists as the digraph is connected: if no arc touches the trail, then there is a vertex which is not in the trail. As the digraph is connected, there is a walk from the start of the trail to this vertex. Then Lemma (9) can be used to derive a contradiction.

So far, we have proved that an Euler trail p exists. The equality of the end vertices follows from (4) and $\mathsf{arc\text{-}balanced}_G\ u\ (\mathsf{set}\ p)\ v = \mathsf{arc\text{-}balanced}_G\ u\ \mathsf{A}_G\ v$ (as $\mathsf{set}\ p = \mathsf{A}_G$).

[3] Bang-Jensen and Gutin [3] restrict their proof to loop-free digraphs. This is actually not necessary as our formalization shows.

The textbook proof of the characterization of open Euler trails usually proceeds as follows: Let u, v be the vertices with $\mathsf{arc\text{-}balance}_G\ u\ \mathsf{A}_G = -1$ and $\mathsf{arc\text{-}balance}_G\ u\ \mathsf{A}_G = 1$. Add an arc from v to u to G. The resulting graph has a closed Euler trail. Removing the additional arc from this trail yields an open Euler trail for G.

With the usual set-theoretic formulation, it is always possible to add an arc to a (multi-)graph. However, in the type system of Isabelle/HOL the universe of the arc type might already be exhausted by the existing arcs. A pathological example for this case was given in Sect. 3.1, but this issue is not restricted to our abstract graph representation; more concrete representations, like $\alpha = \beta \times \gamma \times \beta$, have the same issues, if γ is not large enough.

We can avoid this problem by requiring γ to be infinite (as we are only interested in the Euler trails of finite digraphs). From this result, we can recover the result for arbitrary arc types α by isomorphism.

Lemma 22 *Let $G :: (\beta, \beta \times \mathbb{N} \times \beta)$ dg be a finite digraph with* $\mathsf{tail}\ G = \mathsf{fst}$, $\mathsf{head}\ G = \mathsf{snd} \circ \mathsf{snd}$ *and natural numbers as arc labels. Then we have*

$$[\![\mathit{connected}\ G;\ \{u, v\} \subseteq V_G;\ \mathit{deg\text{-}cond}\ G\ u\ v]\!] \implies \exists p.\ \mathit{euler\text{-}trail}_G\ u\ p\ v$$

where $\mathit{deg\text{-}cond}\ G\ u\ v$ abbreviates the following condition:

$$\begin{aligned}&(\forall w \in V_G - \{u, v\}.\ \mathit{in\text{-}deg}_G\ w = \mathit{out\text{-}deg}_G\ w) \wedge \\ &\quad \mathit{in\text{-}deg}_G\ u + 1 = \mathit{out\text{-}deg}_G\ u \wedge \mathit{out\text{-}deg}_G\ v + 1 = \mathit{in\text{-}deg}_G\ u\end{aligned}$$

Proof This lemma cannot be stated inside the locale (since we have specialized the arc type), but inside the proof we interpret the locale for this concrete graph G.

We construct a new graph $H = \mathsf{add\text{-}arc}_G\ (v, \ell, u)$ where $\ell \in \mathbb{N}$ is a label not occurring in G. Such a label exists as $\mathbb{N}$ is infinite. H is again a finite digraph. For the interpretation of $\mathsf{fin\text{-}digraph}$ with H we use rewriting equations similar to those in Sect. 3.2.

H satisfies the degree condition of Lemma 21. After a case analysis with the cases $u = w$, $u \neq w$, $v = w$ and $v \neq w$, we can automatically prove this for all $w \in \mathsf{V}_H$ using the obvious lemmas relating $\mathsf{add\text{-}arc}$ and degrees and obtain a closed Euler trail for H.

This Euler trail contains the arc (v, ℓ, u). We rotate the trail so that this arc is the first arc of the trail. Removing this arc then yields an open Euler trail for G.

Now we can prove the version for arbitrary arc types. We use the results about isomorphisms from Sect. 4.3 to transfer Lemma 22.

Theorem 23 (Open Euler Trail) *Let $G :: (\beta, \alpha)$ dg be a finite digraph.*

$$[\![\mathit{connected}\ G;\ \{u, v\} \subseteq V_G;\ \mathit{deg\text{-}cond}\ G\ u\ v]\!] \implies \exists p.\ \mathit{euler\text{-}trail}_G\ u\ p\ v$$

Proof We start by obtaining a function $f\ :\ \alpha \to \mathbb{N}$ that is injective on arcs_G and use it to construct an isomorphism h:

$$\mathsf{h} = (\lambda v.\ v, \lambda a.\ (\mathsf{tail}\ G\ a, f\,a, \mathsf{head}\ G\ a), \mathsf{fst}, \mathsf{snd} \circ \mathsf{snd})$$

The property $\mathsf{iso}_G\ \mathsf{h}$ follows by unfolding the definitions of iso and h and the injectivity of f. We then obtain an Euler trail for the graph $H = \mathsf{h} \cdot_{\mathsf{G}} G$ by using Lemma 22. Thanks to the lemmas we proved about isomorphisms the assumptions of the lemma can be discharged automatically.

We transfer the trail to $G = (\mathsf{inv\text{-}iso}\ \mathsf{h}) \cdot_{\mathsf{G}} H$ by applying the inverse isomorphism. The proof is again a simple application of the isomorphism lemmas. □

6 Certifying Non-planarity

In this section, we prove the core idea of a non-planarity checker sound and complete.

A well-known theorem by Kuratowski states that an (undirected) graph is planar if and only if it has a subgraph which is a subdivision of a $\mathsf{K}_{3,3}$ or K_5[12], the complete bipartite graph on 3 and 3 vertices and the complete graph on 5 vertices. We call such a subgraph a Kuratowski subgraph and $\mathsf{K}_{3,3}$ and K_5 Kuratowski graphs.

The LEDA library contains a decision procedure for graph planarity which returns a Kuratowski subgraph to certify non-planarity. A checker algorithm then confirms this certificate using the following procedure: it contracts the arcs of the certificate, leaving only the vertices of degree 3 or more. If the result is a $\mathsf{K}_{3,3}$ or K_5, the witness is accepted. We define a function $\mathsf{certify}$ implementing this procedure and show that it proves the existence of a Kuratowski subgraph (and hence non-planarity). Moreover, we give an characterization of the class of accepted certificates. In other work [19], we use this result to verify a C-implementation of non-planarity checker. This is done by proving that the C-code computes the $\mathsf{certify}$ function.

We model undirected graphs as finite and symmetric digraphs, using the representation from Sect. 3.2. When referring to an arc and its reverse arc, we will also use the term edge. We will implicitly use the embedding function wherever necessary. The same is done in Isabelle using coercive subtyping [21]. The following definitions of complete (bipartite) graphs, subdivision, and contracted graphs give the basis for this development:

Definition 24 *(Complete (Bipartite) Graphs)* $G :: \beta\ \mathit{pair\text{-}dg}$ is a complete graph on m vertices, if it consists of m vertices and all edges connecting these, but no loops.

$$\mathsf{K}_m\ G = (\mathsf{finite}\ \mathsf{V}_G \wedge |\mathsf{V}_G| = m \wedge \mathsf{A}_G = \{(u, v) \in \mathsf{V}_G \times \mathsf{V}_G \mid u \neq v\})$$

G is complete bipartite on m and n vertices, if it consists of two disjoint sets of m and n vertices and exactly the edges between these two sets.

$$\begin{aligned}\mathsf{K}_{m,n}\ G = (&\mathsf{finite}\ \mathsf{V}_G \wedge (\exists U\ V.\ \mathsf{V}_G = U \cup V \wedge U \cap V \\ &= \varnothing \wedge |U| = m \wedge |V| = n \wedge \mathsf{A}_G = U \times V \cup V \times U))\end{aligned}$$

Definition 25 *(Subdivision)* A subdivision splits an edge (u, v) by inserting a new node w. We define the graph $\mathsf{subdivide}\ G\ (u, v)\ w$ as

$$(\mathsf{V}_G \cup \{w\}, \mathsf{A}_G - \{(u, v), (v, u)\} \cup \{(u, w), (w, u), (v, w), (w, v)\})$$

Then, subdivision can be defined inductively:

$$\mathsf{subdivision}\ G\ G$$
$$[\![a \in \mathsf{A}_G;\ w \notin \mathsf{V}_G;\ \mathsf{subdivision}\ G\ H]\!] \Longrightarrow \mathsf{subdivision}\ G\ (\mathsf{subdivide}\ H\ a\ w)$$

For $\mathsf{subdivide}$, we prove some basic properties. In particular, this operation preserves the wellformedness of a graph. We also introduce functions to subdivide and contract an edge in a path.

Based on these definitions, we can now give a definition of planarity:

Definition 26 *(Planarity)* A digraph is planar if and only if it has no Kuratowski subgraph.

$$\mathsf{planar}\ G = \neg(\exists H.\ \mathsf{subgraph}\ H\ G \wedge (\exists I.\ \mathsf{subdivision}\ I\ H \wedge (\mathsf{K}_{3,3}\ I \vee \mathsf{K}_5\ I)))$$

To check whether a graph G is a subdivision of another graph H, one can undo the subdivision step-by-step by contracting those vertices of G which are not in H and have degree 2 (vertices created by subdivision always have

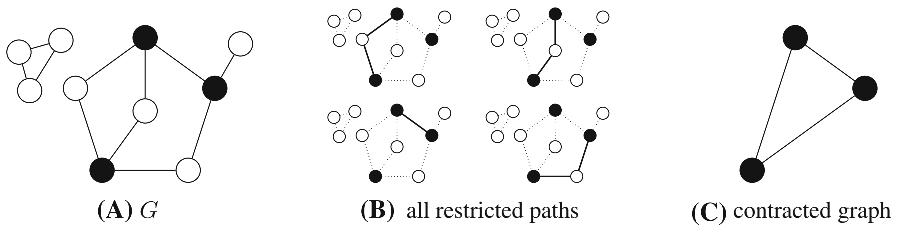

Fig. 2 A graph G and its restricted paths and contracted graph w.r.t. $\mathsf{V}_{3,G}$ (in *black*). Neither the isolated *circle* nor the node of degree 1 are on any restricted path (or in $\mathsf{V}_{3,G}$), so they do not contribute to contracted graphs **a** G **b** all restricted paths **c** contracted graph

this degree). So, to check whether G is a subdivision of a $\mathsf{K}_{3,3}$ or K_5, one contracts all vertices of degree 2 (as all vertices of a $\mathsf{K}_{3,3}$ or K_5 have a degree of 3 or more).

To avoid having to construct the intermediate graphs, the checker procedure $\mathsf{certify}$ we want to verify approximates the contraction of all those vertices at once by computing the contracted graph $\mathsf{contr\text{-}graph}\ G\ V$ as defined below. Here, V is the set of vertices to be preserved by the contraction. We usually assume $\mathsf{V}_{3,G} \subseteq V$, where $\mathsf{V}_{3,G} = \{v \in \mathsf{V}_G.\ 3 \leq \mathsf{in\text{-}deg}_G\ v\}$ denotes the set of vertices of G of degree 3 or more. An restricted path w.r.t. V is a non-empty path were only the end vertices are in V. These denote exactly those parts of the graph which need to be contracted to single edges.

Definition 27 *(Checker Procedure)*

$$\begin{aligned}
\mathsf{iapath}_G\ V\ u\ p\ v = {}& \mathsf{apath}_G\ u\ p\ v \wedge p \neq [] \wedge \{u, v\} \subseteq V \wedge \\
& \mathsf{set}\ (\mathsf{inner\text{-}verts}_G\ p) \cap V = \varnothing \\
\mathsf{contr\text{-}graph}\ G\ V = {}& \big(V, \{(u, v) \mid \exists p.\ \mathsf{iapath}_G\ V\ u\ p\ v\}\big) \\
\mathsf{certify}\ G\ C = {}& (\mathsf{let}\ H = \mathsf{contr\text{-}graph}\ G\ \mathsf{V}_{3,G}\ \mathsf{in}\ \mathsf{subgraph}\ C\ G \wedge (\mathsf{K}_{3,3}\ H \vee \mathsf{K}_5\ H))
\end{aligned}$$

Figure 2 illustrates these definitions. It is easy to see that a graph is not always a subdivision of its contracted graph: isolated vertices or cycles or parallel restricted paths are removed by the contraction, even though they cannot be constructed by subdivision.

This does not matter for the correctness of the checker procedure as we have not yet specified exactly which class of subgraphs we want to be accepted as certificates: The procedure is complete, if, at the very least, all Kuratowski subgraphs are be accepted (so that every non-planar graph is certifiable). For soundness, on the other hand, it suffices if the accepted subgraphs are non-planar.

Instead of just proving correctness, we also give a characterization of the accepted certificates. To this end, we identify the class of slim graphs, which are subdivisions of their contracted graphs.[4]

Definition 28 *(Slim Graph)* A slim graph is a graph that is minimal in the sense that removing a vertex or an arc would lead to a smaller contracted graph. This means that a slim graph only consists of the vertices and arcs on restricted paths. Parallel restricted paths do not contribute to the contracted graph and are also excluded.

$$\begin{aligned}
\mathsf{is\text{-}slim}_G\ V = \big(& V \subseteq \mathsf{V}_G \wedge (\forall v \in \mathsf{V}_G.\ v \in V \vee (\mathsf{in\text{-}deg}_G\ v < 3 \\
& \wedge (\exists x\ p\ y.\ \mathsf{iapath}_G\ V\ x\ p\ y \wedge v \in \mathsf{set}\ (\mathsf{aw\text{-}verts}_G\ x\ p)))) \\
& \wedge (\forall a \in \mathsf{A}_G.\ \exists x\ p\ y.\ \mathsf{iapath}_G\ V\ x\ p\ y \wedge a \in \mathsf{set}\ p) \\
& \wedge (\forall x\ y\ p\ q.\ (\mathsf{iapath}_G\ V\ x\ p\ y \wedge \mathsf{iapath}_G\ V\ x\ q\ y) \longrightarrow p = q)\big)
\end{aligned}$$

[4] If one is only interested in the correctness, there is no need to introduce this concept. Instead, completeness and soundness can be proved by induction over the subdivision relation and the size of the graph, respectively.

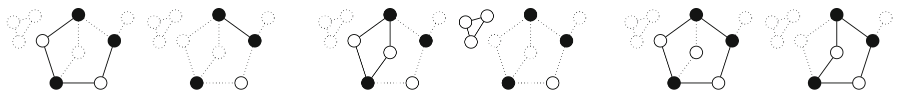

Fig. 3 Some slim and non-slim subgraphs of G from Fig. 2a. **a** Shows some slim subgraphs w.r.t $\mathsf{V}_{3,G}$ (in *black*). The two graphs depicted in **b** are not slim: the first one has parallel restricted paths, the second one contains vertices not on a restricted path. One of the two graphs in **c** is arbitrarily chosen as the slim graph of G

Moreover we define a function $\mathsf{slim}\ G$ which computes a graph G', the slim graph of G, as follows: For each (unordered) pair $u, v \in \mathsf{V}_{3,G}$ connected by an restricted path in G, arbitrarily choose an restricted path connecting u and v. G' then is the graph consisting of the arcs and vertices on all the chosen restricted paths.

These definitions are illustrated in Fig. 3. The idea of $\mathsf{slim}\ G\ V$ is to remove just the parts of G which make G non-slim. Proving that $\mathsf{slim}\ G\ V$ is a slim subgraph of G is straightforward, except for the proof that there are no parallel restricted paths, which we omit here for the sake of brevity.

A slim graph is indeed always a subdivision of its contracted graph:

Lemma 29

$$[\![\textit{is-slim}_G\ V]\!] \implies \textit{subdivision}\ (\textit{contr-graph}\ G\ V)\ G$$

Proof We prove this by induction on $|\mathsf{V}_G - V|$. The proof of the base case follows directly from $\mathsf{contr\text{-}graph}\ G\ \mathsf{V}_G = G$ for slim graphs G.

For the induction step, from G being slim we obtain a $w \in \mathsf{V}_G - V$, which is an inner vertex of some restricted path p. Hence p contains the arc sequence $(u, w), (w, v)$ where u, v are the only two neighbors of w in G. As G is slim, $(u, v) \notin \mathsf{A}_G$: otherwise we could replace $(u, w), (w, v)$ by (u, v) in p and get a parallel restricted path. From this, we can construct a graph H such that $G = \mathsf{subdivide}\ H\ (u, v)\ w$. Note that the $\mathsf{subdivide}$ operation does not change the contracted graph.

As w does not occur in this graph, $|\mathsf{V}_H - V| < |\mathsf{V}_G - V|$. As G is slim and a subdivision of H, H is also slim. By the induction hypothesis, we know that $\mathsf{subdivision}\ (\mathsf{contr\text{-}graph}\ H\ V)\ H)$ and hence $\mathsf{subdivision}\ (\mathsf{contr\text{-}graph}\ G\ V)\ G$. □

In particular, this already implies that the checker procedure is sound for slim certificates. For a non-slim certificate C, we make use of the fact that C and $\mathsf{slim}\ C$ have the same contracted graph w.r.t. $\mathsf{V}_{3,C}$.

Lemma 30

$$\textit{contr-graph}\ (\textit{slim}\ G)\ \mathsf{V}_{3,G} = \textit{contr-graph}\ G\ \mathsf{V}_{3,G}$$

Proof Follows as both G and $\mathsf{slim}\ G$ have the same nodes connected by restricted paths. □

As the slim graph of C is a subgraph of C, we can use that to prove the soundness of the checker procedure.

Theorem 31 (Soundness) *Let G, C be graphs. If* $\mathsf{certify}\ G\ C$ *holds, then G is non-planar.*

Proof From the assumption, C is a subgraph of G and $\mathsf{contr\text{-}graph}\ C\ \mathsf{V}_{3,C}$ is a Kuratowski graph. As the subgraph transition is transitive, also $\mathsf{slim}\ C$ is a subgraph of G. By Lemmas 30 and 29, $\mathsf{slim}\ C$ is a subdivision of $\mathsf{contr\text{-}graph}\ C\ \mathsf{V}_{3,C}$ (and hence a Kuratowski subgraph of G). □

The crucial point for the completeness of the checker procedure is that a Kuratowski graph has no loops and all vertices in have degree 3 or more. This ensures that the contraction of a subdivision of a Kuratowski G graph preserves exactly the vertices of G

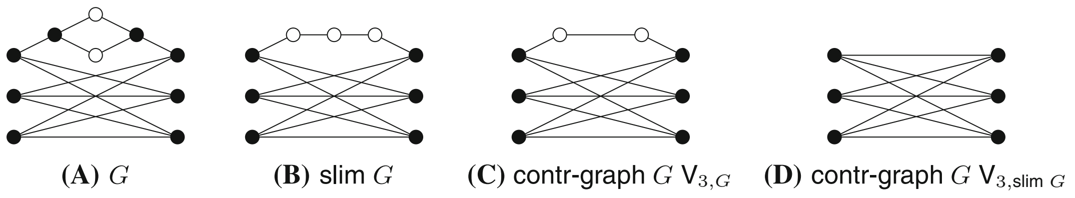
(A) G
(B) slim G
(C) contr-graph G $V_{3,G}$
(D) contr-graph G $V_{3,\text{slim } G}$

Fig. 4 The *black nodes* are vertices of degree 3. The slim graph of G is obviously a subdivision of a $K_{3,3}$. Still, the contracted graph of G w.r.t. $V_{3,G}$ is not a $K_{3,3}$ (but the contracted graph w.r.t. $V_{3,\text{slim } G}$ is) **a** G **b** slim G **c** contr-graph G $V_{3,G}$ **d** contr-graph G $V_{3,\text{slim } G}$

Lemma 32 *Let G, H be graphs such that G is loop-free and H a subdivision of G. If all vertices in G are of degree 3 or more, then* contr-graph H $V_{3,H} = G$.

Proof Straightforward induction over the subdivision. □

Lemma 33 (Completeness) *Let G, C be graphs. If C is a Kuratowski subgraph of G, then the predicate* certify G C *holds.*

$$\exists H.\ (K_{3,3}\ H \vee K_5\ ) \wedge \textit{subdivision}\ H\ C]\!]\implies \textit{certify}\ G\ C$$

Proof Follows directly from Lemma 32. □

We now have a closer look at the class of accepted certificates. Looking at the proof of Theorem 31, we see that a certificate is accepted only if its slim graph is a subdivision of a Kuratowski graph. As illustrated in Fig. 4, this is not a sufficient condition: in addition, the slim graph must have the same vertices of degree 3 or more as the original graph. Now we can give a characterization of the certificates accepted by the checker procedure.

Theorem 34 (Characterization of the Accepted Certificates) *For a graph G, the checker procedure accepts exactly those graphs C as a certificate, which are subgraphs of G and have a slim graph which is a subdivision of a Kuratowski graph. Moreover, the slim graph must have the same vertices of degree* 3 *or more as C itself.*

$$\textit{certify}\ G\ C \longleftrightarrow \textit{subgraph}\ C\ G \wedge V_{3,C} = V_{3,\textit{slim}\ C} \quad \wedge (\exists H.\ (K_{3,3}\ H \vee K_5\ H) \wedge \textit{subdivision}\ H\ (\textit{slim}\ C))$$

Proof Assume that certify G C holds. Then C is a subgraph of G by the definition of certify and the contracted graph D of C w.r.t. $V_{3,C}$ is a Kuratowski graph.

We first note that $V_{3,\text{slim } C} = V_{3,C}$ holds: By Lemma 30, the contracted graph of slim C w.r.t. $V_{3,C}$ is the same as D. The vertices of D are $V_{3,C}$ and, as D is a Kuratowski graph, all of them have degree 3 or more in D. As the degree of a vertex in the contracted graph is less or equal to the degree of the same vertex in the original graph, this implies $V_{3,C} \subseteq V_{3,\text{slim } C}$ and thus equality.

Hence, D is the contracted graph of slim C w.r.t. $V_{3,\text{slim } C}$. Then, by Lemma 29, slim C is a subdivision of D and thus the right hand side of the characterization follows.

For the other direction, it suffices to show that the contracted graph of C w.r.t. $V_{3,C}$ is a Kuratowski graph. As the contraction of a subdivision of a Kuratowski graph is again a Kuratowski graph (cf. Lemma 32), the contracted graph of slim C w.r.t. $V_{3,\text{slim } C}$ is a Kuratowski graph. As $V_{3,C} = V_{3,\text{slim } C}$, this is the same as the contracted graph of C w.r.t. $V_{3,C}$ and we are finished. □

7 Conclusion

In this paper, we have presented the first implementation of a graph library in Isabelle/HOL covering a wide range of fundamental properties. Based on that, we have formalized a characterization of Euler digraphs and a core part of a LEDA checker-algorithm for non-planarity of graphs. The library is already being used in other projects [1,8,20] and we expect it to become the standard for formalizations about directed graphs in Isabelle.

To achieve this goal, it is important that the chosen graph representation can be used to express as much of directed graph theory as possible. Therefore we have chosen an abstract representation that supports multi-digraphs and can be instantiated to many common representations, stemming from both mathematics (arcs as pairs of vertices) and implementation (arcs are pointers into some datastructure).

To make proofs convenient, we have provided a careful setup of automation mechanisms. The heuristics are able to discharge a large number of frequently occurring goals. Examples for the setup are the special simplification procedure from Sect. 4.1 and the use of locales to recover less general graph representations, but also the introduction of the inverse isomorphism.

References

1. Alkassar, E., Böhme, S., Mehlhorn, K., Rizkallah, C.: A framework for the verification of certifying computations. JAR (2013). doi:10.1007/s10817-013-9289-2
2. Ballarin, C.: Locales: A module system for mathematical theories. JAR (2013). doi:10.1007/s10817-013-9284-7
3. Bang-Jensen, J., Gutin, G.Z.: Digraphs: Theory, Algorithms and Applications. 2nd edn. Springer, New York (2009)
4. Butler, R.W., Sjogren, J.A.: A PVS graph theory library. Tech. Rep., NASA Langley (1998)
5. Chou, C.: A formal theory of undirected graphs in higher-order logic. In: Proceedings of TPHOLs '94. pp. 144–157. Springer, New York (1994)
6. Diestel, R.: Graph Theory, GTM, vol. 173. 4 edn. Springer, New York (2010)
7. Duprat, J.: A Coq toolkit for graph theory. Rapport de recherche 2001-15. LIP ENS, Lyon (2001)
8. Esparza, J., Lammich, P., Neumann, R., Nipkow, T., Schimpf, A., Smaus, J.G.: A fully verified executable LTL model checker. In: Proceedings of CAV 2013, pp. 463–478 (2013)
9. Gonthier, G.: computer-checked proof of the Four Colour Theorem (2005)
10. Harary, F., Read, R.: Is the null-graph a pointless concept? In: Graphs and Combinatorics, pp. 37–44. Springer, New York (1974)
11. Hunt, Warren A., J., Kaufmann, M., Krug, R.B., Moore, J.S., Smith, E.W.: Meta reasoning in ACL2. In: Proceedings of TPHOLs '05, pp. 163–178. Springer, New York (2005)
12. Kuratowski, C.: Sur le problème des courbes gauches en topologie. Fundam. Math. **15**(1), 271–283 (1930)
13. Mehlhorn, K., Näher, S.: LEDA: A Platform for Combinatorial and Geometric Computing. Cambridge University Press, Cambridge (2000)
14. Nakamura, Y., Rudnicki, P.: Euler circuits and paths. Formaliz. Math. **6**(3), 417–425 (1997)
15. Nipkow, T., Bauer, G., Schultz, P.: Flyspeck I: Tame graphs. In: Proc. IJCAR '06. pp. 21–35. Springer, New York (2006)
16. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL: A Proof Assistant for Higher-Order Logic. Springer, New York (2002)
17. Nordhoff, B., Lammich, P.: Dijkstra's shortest path algorithm. Arch. Formal Proofs (2012). http://afp.sf.net/entries/Dijkstra_Shortest_Path
18. Noschinski, L.: Graph theory. Arch. Formal Proofs (2013). http://afp.sf.net/devel-entries/Graph_Theory.shtml, Formal proof development
19. Noschinski, L., Rizkallah, C., Mehlhorn, K.: Verification of certifying computations through AutoCorres and Simpl. In: Proceedings of NFM '14. doi:10.1007/978-3-319-06200-6_4
20. Rizkallah, C.: An axiomatic characterization of the single-source shortest path problem. Arch. Formal Proofs (2013). http://afp.sf.net/entries/ShortestPath.shtml, Formal proof development
21. Traytel, D., Berghofer, S., Nipkow, T.: Extending Hindley-Milner type inference with coercive structural subtyping. In: APLAS '11. pp. 89–104. Springer, New York (2011)
22. Volkmann, L.: Fundamente der Graphentheorie. Springer, New York (1996)
23. Wong, W.: A simple graph theory and its application in railway signaling. In: Proceedings TPHOLs '91. pp. 395–409. IEEE (1991)