Findings from Working on Pair-Graph

Christoph Madlener

January 23, 2021

Contents

theory *Findings-Pair-Graph*
imports
AGF.Pair-Graph
AGF.Vwalk
AGF.Awalk
AGF.Component
AGF.SCC-Graph
begin

1 Introduction

This document summarizes the findings and experiences while working on and with a graph representation, where a graph is simply a set of pairs ((*'a* × *'a*) *set*). This representation is referred to as pair graph from now on. While initially the focus lay on comparing different graph representations, it shifted gradually to how to move theorems between graph representations. This culminated in moving the major parts of theory on graph components from the existing AFP entry *Graph-Theory.Graph-Theory* by Noschinski to the pair graph representation.

Note that some of these findings have been documented earlier. Refer to the documents [1] Ports/Findings_Ports_Walks.pdf and Adaptors/output/document.pdf in this repository. The most important points will be restated here while offering additional insights gained since the writing of those documents.

2 Different Graph Representations

In order to reason about graphs, one has to decide first how to represent graphs in Isabelle/HOL. One very common representation for (directed) graphs in graph theory is a pair (V, E) where V is the set of vertices and $E \subseteq V^2$ is the set of edges (or arcs). Obviously there are many more ways for defining graphs. Generally we have more than one way of formalizing a representation in Isabelle/HOL. Enumerating and evaluating all of these possibilities is not feasible. Instead we give a non-exhaustive categorization which cover the graph representations considered in this project. In particular we differentiate representations with explicit vs. implicit vertex sets and concrete vs. abstract edges.

2.1 Explicit Vertex Set

Strictly following common representation of graphs from above, one might be inclined to start by explicitly defining a vertex set. When following this approach, we need to enforce the wellformedness constraint that edges only

[1] The pair graph representation is called DDFS in those documents.

connect "existing" vertices (formulated as $E \subseteq V^2$ above, of course this may look different depending on the edge representation). In Noschinki's *Graph-Theory.Graph-Theory* this minor hassle is dealt with by wrapping parts where wellformedness is relevant in the locale *wf-digraph*. We will see shortly that an explicit vertex set has the full expressive power for graphs, while an implicit vertex set cannot capture all graphs.

2.2 Implicit Vertex Set

If we don't want to deal with wellformedness of graphs we can take the approach of an implicit vertex set; i.e. the vertex set is defined as the union of all start and end points of all edges, as is done in e.g. pair graph with *dVs*. While this frees us of having to worry about (and formulate) wellformedness we cannot represent all graphs using this representation: It is not possible to have disconnected vertices (i.e. there is no edge from or to a vertex). One might argue that in many cases such vertices are not of interest. However, this limitation quickly becomes apparent when working with strongly connected components (SCCs) and the SCC-graph. A vertex with no incoming (or no outgoing) edge forms an SCC with only one vertex and no edge, as does the SCC graph for a strongly connected graph. However, a graph with a single vertex in an implicit vertex set setting can only be represented when there is a loop on this vertex, which is generally not true. The same happens in the SCC-graph if there is an SCC which is not connected to any other component.

2.3 Concrete Edge Type

It remains to decide on how to represent edges. Again, we can strictly follow the representation mentioned before and represent an edge simply as a pair $('a \times 'a)$, which is done in pair graph. All edges of a graph are then a $('a \times 'a)$ *set*. In *AGF.TA-Graphs* Wimmer chooses a different approach of representing the edges as a binary predicate $'a \Rightarrow 'a \Rightarrow bool$. Both of these are concrete in the sense that we are able to e.g. add an arc at any point.

In their current form these representations are not capable of representing multigraphs. Extending both of them to add this capability is possible (e.g. by changing to $('a \times 'a)$ *multiset* resp. $'a \Rightarrow 'a \Rightarrow nat$). Adding additional attributes (like weights) to the edges can be facilitated by defining functions that map edges (or two vertices) to these attributes.

2.4 Abstract Edge Type

In Noschinki's *Graph-Theory.Graph-Theory* edges are simply of some type $'b$. A graph then has to map each edge to its start (*head*) and end (*tail*).

Here, in general we do not know what an edge is, hence, we might also struggle to construct an edge if we want to add it to the graph. On the other hand this representation facilitates multigraphs out of the box. In addition this representation can also be more concrete in some sense when adding attributes to edges: We can simply define $'b$ to be a tuple (or even a record) holding the attribute values. We still need to define (trivial) functions to extract them. So while in development we might need to jump through some extra hoops, the added flexibility for a user of the graph library might be worth it.

3 Automation

An early goal of the project was to evaluate whether one representation is advantageous with regards to the proof automation. Out of the considered representations, no such benefit could be observed at the moment. One reason for this is that in all cases another layer of abstraction is added with further definitions (like walks, reachability, etc.) which is virtually interchangeable between all representations. We also observe that in order to argue about reachability *Graph-Theory.Graph-Theory* basically falls back to a $('a \times 'a)$ *set* representation (see *arcs-ends*, *Digraph.reachable*).

Obtaining better automation appears to be more of an engineering task, rather than being inherent to a graph representation.

4 Adaptors

After concluding that there is no "best" representation, the question arose of how to move lemmas between representations conveniently. The approach followed here is best described as an adaptor. In the ideal case an adaptor allows to obtain lemmas in one representation, by first translating all terms involved to another representation where these lemmas have already been proven.

4.1 Construction

Let's first consider how an adaptor is constructed and what it achieves in the end. Say we have a lemma in graph representation B that we want to transfer to representation A. An adaptor for this purpose needs to define how any graph G_A in representation A can be embedded in a graph G_B in representation B. In addition an equivalence between all involved concepts in the lemma (e.g. reachability in G_A is the same as in G_B) have to be established. It may happen that these equivalences require additional

assumptions. These assumptions will then carry over to the lemma we are transferring.

For example the adaptor in the second part of *AGF.Pair-Graph-Library-Adaptor* defines how to construct a ('*a*, '*a* × '*a*) *pre-digraph* from a pair graph. This adaptor (and others built on top of it) are then used to transfer lemmas from *Graph-Theory.Graph-Theory* to pair graph.

4.2 Evaluation

In general it seems worthwhile to try and use an adaptor if one wants to move material from one representation to another. If the concepts involved in the adaptor from both representations work nicely together, large (interactive) proofs don't have to be "copied". Also building an adaptor is usually achievable with reasonable effort. Hence, should it turn out not to work sufficiently well, one can still resort to proving lemmas directly in the desired representation, without investing too much time.

To further illustrate the above points let's have a look at cases where challenges arise, followed by an example where adaptors offered great convenience. They occurred when moving lemmas on graph components from *Graph-Theory.Graph-Theory* to pair graph. In this feat the *AGF.Pair-Graph-Library-Component-Adaptor* has been instrumental.

Graph-Theory.Digraph-Component defines *pre-digraph.max-subgraph*, a predicate which given a graph predicate, a graph and a (sub-)graph, if that second graph is a maximal subgraph fulfilling the predicate. This is later used to give an alternative definition of SCCs as maximal induced subgraphs. To easily move lemmas including maximal subgraphs, we ideally want to prove that for a graph predicate P, a maximal subgraph H of G in pair graph, the embedded graph of H is a maximal subgraph of the embedding of G. The problem now is that in pair graph, a graph predicate is of type ('*a* × '*a*) *set* ⇒ *bool*, while in *Graph-Theory.Graph-Theory* it has type ('*a*, '*a* × '*a*) *pre-digraph* ⇒ *bool*. So in order to state the goal above, we also have to "translate" or embed the predicate itself. Since the predicate in pair graph simply takes the edge set, a natural way to carry out this embedding is to simply simulate this behavior, i.e. use $P \circ$ *arcs*, where *arcs* maps a ('*a*, '*a* × '*a*) *pre-digraph* to its edge set. It turns out, however, that this is not equivalent: While in pair graph we can effectively exclude a vertex by specifying it appears in no edge, this does not work in the embedding with an explicit vertex set: While the edges including this vertex are also excluded in the embedded graph, there is nothing stopping that vertex from being included in the vertex set. In fact, when following this approach, a maximal subgraph in the embedding will always have all the vertices of the original graph. However, we can also prove that any vertex which appears in the embedded maximal subgraph but not in the original one is disconnected.

Refer to the following lemmas for further details.

thm *max-subgraph-imp-max-subgraph-all-verts*
thm *max-subgraph-digraph-of-additional-verts-disconnected*

There may be a more involved embedding of graph predicates which mends this problem. For now we chose to resort to proving lemmas of pair graph involving maximal subgraphs without an adaptor.

We already considered the discrepancy between implicit and explicit vertex sets in the context of SCCs and SCC graphs. These discrepancies are also highlighted when working with an adaptor: There are definitions involved here (like restricting a graph to a vertex set, see ($\restriction$)) which don't work too well in the adaptor in general. However, in the context of SCCs we have additional assumptions, which allow to employ the adaptor anyways. This is best illustrated by checking out the following lemma and the lemmas leading up to it:

thm *sccs-dVs-image-sccs-conv*

Now to the promised example where adaptors worked out very nicely. A result in graph theory states that a directed acyclic graph (DAG) has a topological sorting. Another result is that the SCC graph forms a DAG. Thus, a topological sorting for the SCC graph exists. Wimmer proved the first result in *AGF.TA-Graphs*. In the adaptor *AGF.TA-Graph-Library-Adaptor* he then goes on to prove that embedding a DAG of Graph-Theory in TA-Graph also is a DAG. This already allows to transfer the result on topological sortings from TA-Graph to Graph-Theory. In addition he proves that the SCC graph in Graph-Theory forms a DAG.

In *AGF.SCC-Graph* we prove that embedding the SCC graph of a pair graph into a Graph-Theory graph results in a subgraph of the SCC graph of the original embedded graph. It immediately follows that a subgraph of a DAG is also a DAG. This allows us to use our existing adaptor to transfer both results to pair graph.

One remark I would like to make is that the adaptors for pair graph were used for the most part to transfer material completely new to this representation (e.g. there was no definition of an SCC in pair graph before). This makes building the adaptor (and also using it) easier, as we can define the involved concepts such that they are equivalent in the embedding. However, there are also multiple ways to define graph theoretic concepts (e.g. is an empty sequence a walk or not). Hence, if there is already some existing material in some area and we want to transfer additional material, the definitions might not match exactly. In this case the aforementioned additional assumptions for equivalences come into play and might make it less convenient to work with an adaptor.

5 Conclusion

At this point it does not seem like there is one superior graph representation. Hence, it seems reasonable to choose the representation which is best suited to the specific needs of the application area. This is especially true since we can move large parts of already developed material between representations effectively using adaptors.
Still it would be nice to somehow unify the efforts, and not spread out material on graph theory on a number of (sometimes very similar) representations going forward. However, this begs multiple questions:

- Which representation(s) are chosen going forward?
- How are these efforts coordinated?
- (Closely linked to the first question) How to gain acceptance by the users for the result?

Regarding the last question it might be instrumental to examine more closely how so many different representations came to be in the first place. Or put more concretely, why do users feel the need or desire to start from scratch when working on/with graph theory, when there are already multiple representations which at least include the basic graph-theoretic concepts to build upon.

end