Summary on Adaptors

Christoph Madlener

December 9, 2020

Contents

theory *Adaptors-Summary*
imports
DDFS-Berge-Adaptor
DDFS-Library-Adaptor
begin

This theory summarizes insights from implementing adaptors between different graph representations and aspects to consider when doing so. Ideally, an adaptor allows to easily transfer theorems from one representation to another by first establishing the relationship of fundamental concepts and then using existing lemmas from one representation to obtain them for the other. In this work specifically an adaptor from graph representation A to graph representation B works in the following way:

1. take a graph (as represented) in A
2. build an equivalent graph in B
3. prove relationship of fundamental concepts – this typically means proving equalities (of e.g. the notion of reachability)
4. obtain advanced lemmas for A by using existing ones from B

In other words an A-to-B adaptor allows transferring lemmas from B to A. This slightly confusing convention arises as the graphs are adapted from A to B the lemmas on the other hand go in the "converse" direction in this process. This also means that an adaptor is not bidirectional, i.e. for transferring lemmas from and to both involved representations two adaptors are needed.

1 Graph-Theory and DDFS

The first adaptor in *Adaptors.DDFS-Library-Adaptor* implements both directions for *Graph-Theory.Digraph*s and *AGF.DDFS*.

1.1 Graph-Theory to DDFS

The first part implements the adaptor from Noschinski's Graph-Theory digraphs to *AGF.DDFS* graphs. Constructing the DDFS graph is relatively straightforward as *Graph-Theory.Digraph* already provides *arcs-ends*. This definition maps the set of (abstract) arcs to a set of pairs which is a graph in the DDFS representation. As DDFS is not capable of representing multigraphs, multi-arcs are "lost".

One obstacle arises from the fact that DDFS uses an implicit vertex set whereas Graph-Theory uses an explicit one. This entails that the set of vertices of the DDFS graph generally is only a subset of the set of vertices of the original graph.

thm *wf-digraph.dVs-subset-verts*

This manifests itself in that we sometimes have to explicitly assume a node to be in the implicit vertex set of our constructed graph, e.g. in

thm *wf-digraph.reachable-imp'*

The greater challenge, however, is posed by the aforementioned capability of Graph-Theory to represent multigraphs. Walks in a multigraph are accurately described by a list of arcs. An arc in Graph-Theory is simply an element of some type $'b$ and a graph has to specify $'b \Rightarrow 'a$ functions for mapping an arc to its tail and head vertex respectively. However, reconstructing an arc from its ends is not trivial. Arcs in Graph-Theory will be referred to as abstract arcs in the following.

This leads to two possible ways of how to deal with the proof of equality of arc-walks:

1. Start with an arc-walk in Graph-Theory, i.e. a list of abstract arcs. An arc-walk in DDFS can then be obtained by mapping each arc to its ends with *arc-to-ends*.

2. Start with an arc-walk in DDFS, i.e. a list of pairs. As mentioned above, reconstructing the abstract arcs directly is not possible, so we resorted to using the choice operator to obtain them, see *pre-digraph.arc-from-ends*.

Both of these approaches exhibit some problems which essentially force us to introduce (strong) assumptions to our lemmas. In both cases **nitpick** was a really helpful tool by finding counter-examples to uncover these additionally necessary assumptions.

1.1.1 The *arc-to-ends* approach

When starting out with an arc-walk p in a graph G in Graph-Theory, it is relatively straight-forward to prove that *map* (*arc-to-ends* G) p is an arc-walk in the DDFS graph, see

thm *wf-digraph.awalk-imp-awalk-map-arc-to-ends*

For empty walks we have to ensure that the start (and in this case also end) vertex are in the implicit vertex set of the DDFS graph.

When trying to prove the converse direction of this lemma, i.e. if *map* (*arc-to-ends* G) p is an arc-walk in the DDFS graph then p is an arc-walk in the original graph, **nitpick** quickly finds the following counter-example:

$$G \equiv (\{a_1\}, \{b_1 \equiv (a_1, a_1)\})$$
$$p = [b_2 \equiv (a_1, a_1)]$$

In this case the DDFS graph is simply $\{(a_1, a_1)\}$, so clearly after mapping b_2 to its ends the resulting list $[(a_1, a_1)]$ is a walk from a_1 to a_1. However, b_2 is not even part of the original graph G, thus p is not an arc-walk in G. To alleviate this issue we have to add the assumption *set* $p \subseteq$ *arcs* G.

thm *wf-digraph.awalk-map-arc-to-ends-imp-awalk*
thm *wf-digraph.awalk-iff-awalk*

1.1.2 The *pre-digraph.arc-from-ends* approach

When choosing the "converse" approach of starting with an arc-walk p in the DDFS graph, again it is relatively straight-forward to prove that *map pre-digraph.arc-from-ends p* is an arc-walk in the original graph. The reason being that in this case the choice operator is well-defined in the sense that for each pair in p there is a corresponding abstract arc in Graph-Theory graph G.

thm *wf-digraph.awalk-imp-awalk-map-arc-from-ends*

However, the other direction again requires additional care, in this case due to the involved choice operator. The goal is to prove that given *map*

pre-digraph.arc-from-ends p is an arc-walk in G, p is an arc-walk in the DDFS graph. Once more **nitpick** quickly gives a counter-example, which can be boiled down to the following:

$$G \equiv (\{a_1\}, \{(b_1 \equiv (a_1, a_1)\})$$
$$p = [(a_1, a_2)]$$

Here, the pair (a_1, a_2) does not correspond to an arc in G, the choice operator in *pre-digraph.arc-from-ends* will still map this pair to the only arc b_1. However, $[b_1]$ is a valid arc-walk in G, while p is clearly not an arc-walk in the DDFS graph $\{(a_1, a_1)\}$.

The only way to fix this is to add an assumption *set* $p \subseteq E$ to make the choice operator "well-behaved".

1.1.3 Vertex walks

Graph-Theory also offers a formalization of vertex-walks, i.e. a walk is a list of vertices, which for multigraphs is not necessarily an accurate representation. For non-multigraphs this representation is absolutely fine, as an arc is uniquely identified by its ends. For this adaptor it is also easier to work with vertices, as we have already established a subset relation between the vertices of the DDFS graph and the vertices of the original graph.

The only small nuisances are (once more) the implicit vertex set of DDFS and the fact that the definition of vertex walks differs on the treatment of empty lists.

thm *wf-digraph.vwalk-iff*

1.1.4 Conclusion

We introduced two ways of dealing with the discrepancy between abstract arcs in Graph-Theory and pairs in DDFS with regard to arc-walks. In both cases the strictly greater expressiveness of Graph-Theory became apparent and poses challenges. Both approaches required the introduction of (possibly unwieldy) "translation" functions between the types of arcs and strong additional assumptions to prove the relationship of arc-walks in both formalizations. The question remains how sensible it is to implement the adaptor in this direction (at least for arc-walks), as we cannot prove all the properties of arc-walks (for multigraphs) in DDFS anyways.

For vertex walks a different picture is drawn, as they work conceptually the same way in both formalizations. It highlights the more nuanced differences, though. The most prominent being the implicit vs. explicit vertex set. While an explicit vertex set allows disconnected vertices, in general some kind of well-formedness condition on the arcs has to be imposed. An implicit

vertex set on the other hand does not impose any additional conditions, but disconnected vertices are not easily representable. If this trade-off is better the one way or the other has to be considered on a case by case basis. Similar reasoning can be applied to the question if an empty walk is considered a walk or not.

1.2 DDFS to Graph-Theory

The second part of *Adaptors.DDFS-Library-Adaptor* implements the adaptor from DDFS graphs to Graph-Theory graphs. This direction works smoothly, as the more expressive Graph-Theory easily accommodates DDFS graphs. Constructing the graph and all the proofs for the fundamental concepts are straightforward. This adaptor can in some sense be seen as a special case of *Graph-Theory.Pair-Digraph* with a practically implicit vertex set.

This direction appears to be more useful and straightforward, as it allows to easily reuse lemmas from a more powerful formalization. Hence, in a context where this additional expressiveness is not required, one can easily switch to a more manageable and conceptually simpler representation.

2 Berge to DDFS

The second adaptor is between the undirected *AGF.Berge* representation (i.e. a graph is a set of sets) and DDFS. The main goal of this endeavour was to evaluate how feasible it is to derive lemmas about undirected graphs from lemmas about symmetric graphs. As before, building the (now symmetric) DDFS graph from the undirected representation is straightforward. Proving fundamental relations between vertices, edges and walks also does not pose a challenge. Obtaining more advanced lemmas about these concepts for the undirected case afterwards works out nicely as well.

However, big parts of the Berge formalization deal with the edges in a walk (cf. *edges-of-path*), which is essentially a list of two-element sets. When defining this for the DDFS case this becomes a list of pairs. While formulating (mostly) equivalent lemmas in this form is possible, the relation between *edges-of-path* and *edges-of-dpath* already involves mapping each pair back to a two-element set.

thm *graph-abs.edges-of-path-eq*

Proving advanced lemmas in the set formulation from the ones in the pair formulation in some cases becomes somewhat of a hassle, e.g.;

thm *graph-abs.edges-of-path-for-inner′*

or straight-up impossible:

thm *graph-abs.distinct-edges-of-vpath'*

2.1 Conclusion

Building an adaptor from an undirected to a (symmetric) directed representation for graphs is certainly possible and useful, depending on the concepts one wants to argue about. As long as the arguments mainly follow from whether two vertices are connected or not, the transition between an undirected graph and a symmetric directed graph works smoothly. On the other hand, when the argumentation is mainly done on the basis of concrete edges, working with the adaptor can become awkward rather quickly.

end