Functional Correctness of C Implementations of Dijkstra's, Kruskal's, and Prim's Algorithms

README for Artifact Evaluation, last updated May 4, 2021

I. Downloading

Our paper presents a Coq development that facilitates the verification of graph-manipulating programs. Our codebase is available to users in two forms:

Option A: Docker Image
We provide a Docker image that contains a fully functional, compiled, Coq-checked installation of our system. The machine also contains an installation of Emacs with ProofGeneral to allow users to browse our files and "step through" our proofs. To run our Docker image, proceed as follows:

1. Install Docker from https://www.docker.com/ and start up the Docker daemon on your machine
2. Run `docker pull anshumanmohan/certidpk_built`
3. Run `docker run -it anshumanmohan/certidpk_built bash`

If you are unfamiliar with Coq and Emacs in a command line setting, please refer to the Appendix, where we provide a helpful guide.

Option B: a GUI-based tour on your own machine
While the Docker image has the benefit of being precompiled, its sparse GUI is a barrier to serious development. We thus make our code publicly available (https://github.com/anshumanmohan/CertiGraph-VST) and invite users to build it on their own machines. The included Dockerfile is a good "recipe" for how to build this codebase and set up an environment for development, including which versions of relevant software to use.

II. Tour of Relevant Code

Our work is in `CertiGraph-VST/CertiGraph/`. CertiGraph is a large library with many examples, but our work is laid out over a subset of the files. The hyperlinks below lead to the public codebase for ease of reference.

The extensions to CertiGraph described in Section 2 of the paper are as follows:

- Mathematical foundations for adjacency matrices
- Spatial representations of adjacency matrices (three versions, as explained in the paper: 1 2 3)
- Similarly, mathematical and spatial developments for edge-list graphs
- Extensions to allow undirected graphs, and bridging between undirectedness and union-find

Now let us explore Dijkstra's algorithm in some detail. As explained in the paper, we verify three versions of the program, but we focus on the first version (dijkstra1.c) for now. The C code we wish to verify is here. We use CompCert's clightgen tool to generate a Coq-readable AST of that C code; that is here. The specification that we will verify is stated here. The specification relies heavily on:
(1) purely-mathematical definitions such as inv_popped (that are stated here),
(2) separation logic explanations of graph layout (that are stated here), and
(3) on a mathematical, Dijkstra-specific graph (DijkGG, fleshed out here)

DijkGG is a refinement of a more general mathematical graph, as described in the paper. It is worth studying this pair of files (one and two) to see how the latter refines the former.

Finally, the proof of Dijkstra's correctness is here; it relies heavily on a "pure" proof of correctness that is independent of spatial layout. The pure proof is here. The other versions of Dijkstra's algorithm (tagged 2 and 3 respectively) differ only in the spatial layout of the graph, and can be explored similar to above.

Our verifications of Prim's and Kruskal's algorithms are laid out similarly in their respective directories. Several of these algorithms rely on the binary heap verification that we describe in the paper; that is in its own directory.

III. Reusability of our Work

We have implemented standard treatments of algorithms in straightforward C. For a taste, we have developed a short snippet of code for each implementation of Dijkstra's algorithm that allows a user to create a arbitrary graphs with little toggles for "connectivity" and "inflation" and run our verified implementation of Dijkstra's algorithm on it. The snippet also prints out the paths that have been found.

Attaching an email from AEC chair Clément to this effect.

> In the case of a proof, the other criteria of the reusability badge are the most important: proper documentation, etc. The example sounds very interesting, but unless it's easy enough to be done by the reviewer with minimal guidance, it's a bit orthogonal to reusability. A better example might be a small snippet of code that combines uses Dijkstra's algorithm on a concrete graph with a few nodes, and instructions on how to compute the shortest path and to specialize the proof to that particular example so that it confirms that this particular run of the algorithm is correct (in that sense the proof would be "reusable" in that it can be integrated into other developments as a tool).
>
> (Feel free to include this email in your artifact description if you go this route, so that there's no doubt in the reviewers' mind)
>
> Cheers,
> Clément.

Appendix: Coq + Emacs + ProofGeneral Guide

For those unfamiliar with Coq, Emacs, and ProofGeneral, we provide a guided to opening, exploring, and understanding the verification of Dijkstra's algorithm inside our Docker build. Here we explain Emacs commands as `a+b, c+d`. By this we mean four keystrokes: "hold `a` and type `b`,

and then hold `c` and type `d`". The plus and the comma are meant for readability and are not to be typed.

1. After entering our Docker machine, type `emacs` to start Emacs.

2. To open a file, type `Ctrl+x, Ctrl+f`. This will enter you into "find file" mode, and you will see a prompt on the bottom left asking you for a file name. At the prompt, key in `~/CertiGraph-VST/CertiGraph/dijkstra/verif_dijkstra1.v`.

3. In the Docker machine, we have installed ProofGeneral, which is a plugin into Emacs that arms the simple text editor with additional proof-specific features. Since you just opened a Coq file (i.e. with a `.v` extension), ProofGeneral will automatically kick into action in "coq mode".

4. Now you can use ProofGeneral's commands to navigate the proof. In particular:
   `Ctrl+c, Ctrl+n` makes the editor "step through" the next line of the proof in a REPL style.
   `Ctrl+c, Ctrl+u` reverses this, retracting by one line.
   `Ctrl+c, Ctrl+b` steps through the entire file (warning, lengthy step).
   `Ctrl+c, Ctrl+RET` steps until whichever line the cursor is on.

5. When a particular line of code gets underlined and there are no complaints from ProofGeneral, that means that the commands/tactics on that line of code were accepted happily by Coq.

6. We will often see `Lemma <NAME>: <STATEMENT>. Proof. <TACTICS>. Qed.` The assertion here is that the TACTICS following Proof will prove the lemma's STATEMENT. This assertion is checked by the command `Qed`. So if we are able to "step through" until `Qed` without complaint from Coq, we know that the lemma was proved.

7. The key proof in this example is `Lemma body_dijkstra` starting on line 357. Its statement is a little obscure, but it is saying that the function find (`f_dijkstra` from our C code) conforms to the specification we defined for it (`dijkstra_spec` from line 48 of the file `dijkstra_spec1.v`).

8. `dijkstra_spec` combines definitions and relations defined in other parts of our development. In general, to dig a little deeper and see any definition more fully, users can move the cursor to the definition in question and type `Ctrl-c, Ctrl-a, Ctrl-p, RET`. This prints out the definition. Alternately, users can type `Ctrl-c, Ctrl-a, Ctrl-p` and then type out the name of the definition they are interested in, followed by `RET`. A little investigation of `dijkstra_spec` shows that this corresponds to the various component pieces that are described in the paper.

9. To exit Emacs, type `Ctrl-x, Ctrl-c`. You may be prompted to save changes to the file (we recommend not editing our files) and may be warned about exiting while active processes are running (this is okay, you can type "yes"). This will bring you back to the Docker machine's command line prompt. To exit the Docker machine and go back to your own machine, type `exit`.