CAV2021 (author)

Help / Log out

My Submissions	CAV2021	Premium	Conference	News	EasyChair

Rebuttal Information for Submission

Response

Rebuttal Email and Response	
Response:	We thank the reviewers for their appreciation of our work. Detailed comments for reviewer 1: We wrestled with the amount of space to give to our priority queue. You are correct that binary heaps are a standard verification benchmark, and we should have said so. We will add a “related work” subsection to S5 and cite both of the implementations you mention (Why3 and Dafny). We will also do a literature search to find others (and would incorporate any further associated references you suggest). Our binary implementation is more advanced than either the Why3 or Dafny solutions since we provide both decrease-key as well as Floyd’s bottom-up heap construction. The latter requires a generalization to the key loop invariant of “sink,” which we give, and the former is surprisingly hard to find even in algorithms textbooks. We also found a rather subtle potential overflow related to unsigned integer constants in C (i.e., we handle the nitty-gritty of bare-metal code). Lastly, the binary queue makes a major difference in performance as compared to a naïve priority queue. All of that said, we agree with your overall point, and will aim to include additional information about our Kruskal implementation going forward. We also agree with you that we should have discussed more-automated approaches to graph algorithms in more depth. We will certainly examine and cite the Why3 gallery going forward, and will also examine Chen et al. We were likewise not familiar with "Simpler proofs with decentralized invariants," but will take a close look. We were familiar with “Verifying the Correctness and Amortized Complexity of a Union-Find Implementation in Separation Logic with Time Credits,” and note that this paper was cited in the CertiGraph paper (reference [59] in the submitted paper), which verifies the union-find we incorporate in the present result. We agree that one of the key challenges in verifying Kruskal’s algorithm is a proper treatment of union-find. As we expand our S4.4 discussion of Kruskal, it would be natural to expand our S4.5 discussion of related work with appropriate union-find references. We will clarify what we mean by “functional correctness”. We do allow edge weights of 0 and will clarify the text. No special treatment was needed in the proofs. The line "Red text in an annotation indicates changes compared to the annotation immediately prior" refers to colors used in figure 1. We will clarify.

	We only have pen-and-paper proofs about time complexities. At the present time the VST framework does not provide support for proving time or space bounds. We do not believe that we use any particularly interesting facets of CompCert C in the present work (CompCert's overflow semantics should closely mirror the ANSI C standard). Accordingly, any ANSI C compiler should be fine, although one would then lose the end-to-end correctness guarantee. We agree that it would be interesting to benchmark CompCert code against other compilers. We agree that stack-allocated structures are often simpler than heap-allocated structures, and so -- perhaps -- our hammer is a bit big for such contexts. That said, "porting" code that uses heap-allocated structures to a stack-allocated context -- or vice versa -- is relatively simple in our framework (as we discussed in S3.1 and S6, porting required less than a 1% change to the formal proofs). Thank you for your other comments on typos, etc. We will use them to improve the text.
Time:	Mar 31, 20:47 GMT
Authors can update:	no
Email:	Dear [*FIRST-NAME*], Thank you for your submission to CAV2021. The CAV2021 rebuttal period will be between March 29th and March 31st (AoE). During this time, you will have access to the current state of your reviews and have the opportunity to submit a response of up to 750 words. Please keep in mind the following during this process: * The response must focus on any factual errors in the reviews and any questions posed by the reviewers. It must not provide new research results or reformulate the presentation. Try to be as concise and to the point as possible. * The rebuttal period is an opportunity to react to the reviews, but not a requirement to do so. Thus, if you feel the reviews are accurate and the reviewers have not asked any questions, then you do not have to respond. * The reviews are as submitted by the PC members, without any coordination between them. Thus, there may be inconsistencies. Furthermore, these are not the final versions of the reviews. The reviews can later be updated to take into account the discussions at the program committee meeting, and we may find it necessary to solicit other outside reviews after the rebuttal period. * The program committee will read your responses carefully and take this information into account during the discussions. On the other hand, the program committee will not directly respond to your responses, either before the program committee meeting or in the final versions of the reviews.

	* Your response will be seen by all PC members who have access to the discussion of your paper, so please try to be polite and constructive. The reviews on your paper are attached to this letter. To submit your response you should log on the EasyChair Web page for CAV2021 and select your submission on the menu. Best wishes, Alexandra and Rustan [*REVIEWS*]
Time:	Mar 29, 09:54 GMT

Reviews

Review 1

Overall evaluation: **2**: (accept)

This paper presents the formal verification of three proeminent graph processing algorithms: Dijkstra's shortest path and Kruskal's and Prim's minimum spanning tree computation. The authors also describe their effort to verify binary heaps, a common ingredient of graph algorithms. The authors report on several extensions to existing platforms (e.g., edge-labeled graphs, undirected graphs, and graph implementation data structures), as well as subtle details in the implementation of the aforementioned algorithms, namely an overflow during shortest path computation and that Prim's algorithm can directly compute minimum spanning forests. Some proof statistics are presented.

** General appreciation:
This is a very interesting piece of work. Tackling the verification of graph algorithms is always a very demanding task, moreover if such verification effort is applied to real world code. Using C as the implementation language poses, yet, another challenge as one must constantly be careful about pointers manipulation. The extensions made to the CertiGraph library are relevant, as edge-labeled and undirected graphs are pervasive in graph-manipulating algorithms. The overflow bound on Dijkstra's shortest path algorithm is interesting, but I find Prim's application to forest the most shining result of this paper: it shows that, through formal verification, it is possible to uncover relevant properties, even for classical algorithms that have been extensively analyzed for decades. I look forward for the accompanying artifact, which I believe will nicely complement the paper. Finally, another strong point of this work is its extensive list of references and the care the authors took to spread related work across different sections of the text. This really helps during the reading process, and adequately compares this work with existing literature.

My only main concern about this paper is the importance given to binary heaps verification. Binary heaps have been extensively used as case studies for other tools (I shall detail further on this later). While I acknowledge the central role of this data structure in graph processing algorithms, I believe it takes too much space in the text and, sometimes, shadows the effort of graph algorithms verification, which should be the main contribution of this paper. Already on the abstract, it feels odd and a little artificial the inclusion of binary heaps. From the title there is no clue whatsoever that binary heaps are going to take a central role in this paper. In my honest opinion, the entire section on verified binary heaps (Section 5) could be much reduced and more content could be added into the Kruskal's algorithm presentation (Section 4.4). For instance, it would be much interesting to see more details about the union find specification and its use for proving Kruskal's implementation. Other (minor) concerns are related to technical questions (which I believe the authors can address at rebuttal-time) and some from related work (which I believe the authors can address in the final version). I shall detail on those in what follows.

In the end, I believe this work deserves to be presented and I support its acceptance as a regular paper at this year's CAV.

** Detailed comments:

*** Abstract:
- I am not very keen on seeing references at an abstract. The authors should consider removing it (reference [59]).

*** 1. Introduction
- "[...] and prove in Coq the full functional correctness [...]". I am always skeptical when someone makes claim of full functional correctness. How do measure "full"? Does it mean you have verified every possible property about this program? What about proving complexity bounds of execution? I agree complexity claims tend to be interpreted as non-functional requirements, but graph algorithms naively implemented (i.e., with wrong complexity behavior) are (almost) useless. Have the authors considered analyzing time-complexity of their implementations?

- On the use of CompCert semantics. I think using the C semantics implemented in a certified compiler is the right choice, and it should be the standard in the formal methods community. However, and for the remaining of the paper, CompCert just seems to be a nice ingredient of the work and not much else is said. What would have been the implications of resorting to other C standards? Was CompCert semantics crucial at any point in the proof (e.g., arithmetic overflows)? Even if I acknowledge that VST aims for CompCert C code, would the authors still be able to compile their C certified code, in different compilers, without compromising their verification results? Also, it would have been interesting to see some benchmarks on the compiled code issued by CompCert, and compare it to world-wide used implementations of Dijkstra, Prim, and Kruskal algorithms. I believe this would have been a fine addition to the Engineering considerations (Sec. 6) section.

*** 2.1 Pure reasoning for adjacency matrix-represented graphs
- "Dijkstra's algorithm requires *positive* edge weights [...]". Are the weights strictly positive? Or can one relax (I believe so) this property to *non-negative* weights? Is there any special treatment for zero-weighted edges?

*** 2.2 New spatial representations for edge-labeled graphs
- On stack-allocated adjacency matrices. While I appreciate the need for Separation Logic (SL) to express disjointedness of heap-allocated structures, I struggle more on the application of SL when data is

allocated on the stack. Since stack-allocated 2D and 1D arrays can be
represented in "flat" memory without aliasing, it feels one can use a
different program logic, almost with no memory model at all, where
Verification Conditions are much easier to express and discharge. Have the
authors considered this alternative approach? For instance, using a type
and effect system with regions to control memory management? I believe it
could increase the degree of proof automation.

*** 3.1 Verified Dijkstra's algorithm in C
- "Red text in an annotation indicates changes compared to the annotation
immediately prior". I simply cannot understand this sentence. What prior
annotation are the authors referring to?

*** 4.4 Kruskal's Algorithm
- I agree with the authors that union find is a very challenging case study
for deductive verification. In that matter, the authors should consider
citing the work by Charguéraud and Pottier "Verifying the Correctness and
Amortized Complexity of a Union-Find Implementation in Separation Logic
with Time Credits" (Journal of Automated Reasoning, 2019). This presents
the verification of an Union Find implementation (in the form of a
directed graph) in SL, using a very elegant specification. This ultimately
leads to a much more concise and simpler proof. Indeed, Filliâtre et
al. were able to directly reuse such a specification and provide a
fully-automatic proof of union find (this is described in the recent paper
"Simpler proofs with decentralized invariants", Journal of Logical and
Algebraic Methods in Programming, 2021).

*** References:
- Reference 39 is ill-typed. It misses the year and conference
information. I believe this is a reference to the ITP 2019 paper.

** Related work:
I find the authors presentation quiet unfair towards more automated
verification tools. Such class of verification frameworks, in particular
SMT-based ones, have made impressive progresses in the last decades and
are now mature verification tools. There are already a very interesting
number of verified graph algorithms in frameworks such as Dafny, VeriFast, or
Why3. In particular, Why3 features an online gallery solely dedicated to the
verification of graph algorithms
(http://toccata.lri.fr/gallery/graph.en.html), which I recommend the authors
to revise and cite in their paper.

The authors should also consider citing the more recent work by Chen, Cohen,
Lévy, Merz, and Théry "Formal Proofs of Tarjan's Strongly Connected
Components Algorithm in Why3, Coq and Isabelle" (ITP, 2019). This paper
nicely presents how different verification tools can be used in
collaboration, in order to achieve better proof results.

On the verification of binary heaps. The VACID-0 benchmark features a set of
challenges for formal verification. One of such challenges is, indeed, the
verification of binary heaps. Solutions for such challenge have been
implemented, for instance, in Dafny (K. R. M. Leino and M. Moskal. VACID-0:
Verification of ample correctness of invariants of data-structures,
edition 0, Tools and Experiments Workshop at VSTTE, 2010) and in Why3 (Asma

Tafat and Claude Marché, Binary Heaps Formally Verified in Why3, https://hal.inria.fr/inria-00636083/document, 2011).

** Typos:
- Throughout the paper: "e.g." and "i.e." should be preceded and succeed by commas: ",e.g.,"

- Page 1: "source vertex source". Remove the last "source".

- Page 3: "[...] then detail our extensions we make [...]". I believe the authors intention was to write "*the* extensions we make".

- Page 3 (and throughout the rest of the paper): Symbols "V/\epsilon" start a new sentence. One should definitely avoid to start new sentences and new lines with symbols.

- Page 5: "An valid upath [...]" --> "A valid upath".

- Page 5: "[...] all valid edges have src <= dst". The words src and dst should be typed using tt family font.

- Page 10: "Indeed, our initial verifications of C code [...]". Indeed what? Please, avoid starting a new paragraph with a speech connector.

- Page 11: "[...] as an formal exercise [...]" --> "as *a* formal exercise"

- Page 13: "[...] a verified proof of Kruskal's [...]". What is a verified proof?

- Page 19: "[...] whose semantics has been precisely defined semantics [...]". Remove the last *semantics*.

- General remark: in code listings, if you are never going to refer to line numbers, please simply do not include them.

- General remark: please consider using the LaTeX `cite` package in order to sort multiple citations. For instance, [31, 54, 55, 2] (page 12) is just awful to look at.

Review 2

Overall evaluation:	**2**: (accept) This paper describes the foundational verification (using Coq's VST Toolchain framework) of C implementations of several textbook graph algorithms for implementing heaps and using them to compute shortest paths, spanning trees. While the algorithms are well known even to undergraduates, the paper shows that implementing them well is quite tricky as one has to be careful with all sorts of issues involving arithmetic overflows/underflows, and that, surprisingly, some steps shown in the textbooks are, in fact, not needed to establish correctness. The verification is a heroic effort, requiring reasoning about about non-trivial properties of pointers, arrays, how C records are layed out and arithmetic overflow etc.

Thus, while the CAV audience may mostly subscribe to Lipton, De Millo and Perlis' position that formal proofs are not fun to read about, and while there are no giant algorithmic insights on proof automation, I think the paper would nevertheless be an excellent addition for several reasons.

First, the paper is an excellent exposition of all the nitty gritty low level details that are routinely abstracted away, but which are essential to verify the "functional correctness" of low-level code.

Second, the authors had to develop and implement various principles and libraries for reasoning about graphs (in coq) that may be of independent interest.

Third, there are some nice insights about the algorithms themselves that arose as part of the formal proof.

Finally, and perhaps most importantly, the paper is an excellent source of a "challenge" problem for the "automated verification" community. The authors have done the hard work
of identifying some really challenging invariants and properties -- beyond, e.g. the relatively simple, list-segment kinds of abstractions we are familiar with -- that could inspire much follow on work on developing suitable abstract domains, or invariant synthesis
algorithms.

In short, I think this paper provides excellent inspiration for those looking for new, challenging and important problems where verification can benefit from the aid of computation,
making it an excellent fit for CAV.