Proof Pearl: Dijkstra's Shortest Path Algorithm Verified with ACL2

J Strother Moore[1] and Qiang Zhang[2]

[1] Department of Computer Sciences, University of Texas at Austin, Austin, Texas 78712, USA moore@cs.utexas.edu,
WWW home page: http://cs.utexas.edu/users/moore

[2] Department of Computer Sciences, University of Texas at Austin, Austin, Texas 78712, USA qzhang@cs.utexas.edu,
WWW home page: http://cs.utexas.edu/users/qzhang

Abstract. We briefly describe a mechanically checked proof of Dijkstra's shortest path algorithm for finite directed graphs with nonnegative edge lengths. The algorithm and proof are formalized in ACL2.

1 Introduction and Related Work

Dijkstra's shortest path algorithm [2, 3] finds the shortest paths between vertices of a finite directed graph with nonnegative edge lengths. This paper formalizes that claim in ACL2 [7] and briefly describes a mechanically checked proof of it.

ACL2 is a Boyer-Moore style theorem prover by Kaufmann and Moore that supports a first-order logic based on recursively defined functions and inductively constructed objects. The syntax is that of Lisp, which we use (and paraphrase) in this paper – contrary to the TPHOLS tradition – since "proof pearls" are meant to show how certain theorems are proved in certain systems. The ACL2 syntax does not include quantifiers, but the logic provides a means of introducing "Skolem functions" providing full first-order power. This facility is crucial to the proof described here.

We represent graphs in ACL2 with a list data structure called an association list. We define the function `dijkstra-shortest-path` to implement the algorithm. It takes two vertices, a and b, and a graph as input and it returns a value, say ρ. We prove that ρ, is either `nil` or a path in the graph from a to b, and that no path in the graph from a to b is shorter than ρ. In our formalization, the non-path `nil` has "infinite" length and all finite paths are shorter. Hence, our theorem ensures that if ρ is `nil`, there is no path from `a` to `b`.

Despite the age and classic nature of the algorithm, there is relatively little work on the correctness of Dijkstra's algorithm in the mechanical theorem proving literature. As far as we are aware, the first mechanically checked proof of the correctness of the algorithm was done in Mizar by Jing-Chao Chen [1] in a paper submitted March 17, 2003. For the record, the first ACL2 proof was completed in September, 2003. Our proof requires significant user guidance, but our script is about one third the size (in character count and line count). In addition, the

Mizar article draws upon notation and results in 22 other Mizar articles. The ACL2 proof uses no external definitions or theorems – everything is done from ACL2's basic "bootstrap theory."

Joe Hurd formalized and proved the reachability property of Dijkstra's algorithm in HOL and this result is cited in [5]. A similar algorithm, Floyd's all-pairs shortest path algorithm, was formalized and proved correct in Coq by Eric Fleury in July, 1990 [4] (unpublished manuscript). In February, 1998, Christine Paulin and Jean-Christophe Filliâtre proved Floyd's algorithm in Coq using a script that is available at [9].

In ACL2, Moore did the first proof of Dijkstra's algorithm in September, 2003. He then challenged Zhang, then a relatively new ACL2 user, to do it as an exercise, without seeing Moore's proof. Zhang completed his first proof in December, 2003, with some guidance from Moore. Then Zhang cleaned up his proof, removing many user-supplied proof hints in the process. The proof described here is Zhang's second proof.

The rest of this paper is organized as follows. In the next section we describe the formalization of the algorithm in ACL2. In the subsequent sections we give our specification, the main invariant, a sketch of the proof, and a typical user-supplied hint. In Section 7 we give some statistics about it. The complete script of our work is available online at http://www.cs.utexas.edu/users/-qzhang/shortest-path/index.html.

2 Formalization

In ACL2, ordered pairs are called conses and are constructed by the function `cons`. The left component of a `cons` is accessed with the function `car` and the right component is accessed with `cdr`. Conses are used to represent lists. The `car` of such a cons is the first element in the list and the `cdr` is the list containing the remaining elements. A list is said to be a *true-list* if its `cdr`-chain is terminated with `nil` rather than some other atom.

An *association list* (or *alist*) is a true-list in which the elements are pairs in which the `car` is said to be *associated with* or *bound to* the `cdr`. It is easy to write the function that looks up the value of a key in an alist, by recurring down the `cdr`-chain to the first pair that binds the key in question. It is also easy to write the function that copies an alist inserting a new binding for given key.

We use alists extensively in this work. A directed graph is an alist associating vertices with edge lists. An *edge list* is an association list associating vertices to nonnegative rationals called *edge lengths*. The function `graphp` checks that an object is of the shape just described. If the edge list associated with some vertex `u` in a graph `g` binds `v` to `w`, then it means there is an edge in `g` from `u` to `v` with edge length `w`. The function `all-nodes` collects a duplication-free true-list ("set") of all vertices mentioned in a graph. `(Neighbors u g)`, traditionally written $Neighbors(u, g)$, returns the set of all vertices reachable from vertex `u` via one edge in `g`.

A path `p` in a graph `g` is a non-empty list of vertices with the property that successive elements of `p` are linked by an edge in the graph `g`. `Path-len` returns the sum of the edge lengths of the edges in a path.

In ACL2 it is common to use the atom `nil` for a variety of extended meanings. It is used both as the terminal marker in true-lists and as the "false" truth-value. We also use it as "infinity" in our system of lengths. That is, we define a strict ordering `lt` ("less than") and its weaker counterpart `lte` ("less than or equal") so that `nil` dominates all rational lengths. We also use `nil` to denote a non-existent path; that is, if asked to find a path between two vertices where no such path exists, we will return `nil`. We define `path-len` to return `nil` (infinity) on the non-path `nil`. In an abuse of the strictness implied by the word "shorter," we define `(shorterp p1 p2 g)` to be `(lte (path-len p1 g) (path-len p2 g))`.[3]

The core of Dijkstra's shortest path algorithm is an iterative procedure, here called `dsp`, that computes a *path table*. In our work, path tables are association lists that pair vertices to paths. All the paths start at the same source vertex. Suppose the source vertex is `a`. Then if `u` is paired with path `p` in the path table, then `p` is a path from `a` to `u`. Other important invariants on the path table are discussed later. We define `(path u pt)` to return the path associated with `u` in `pt` (or `nil` if no path is associated with `u`) and we define `(d u pt g)` to return its length, `(path-len (path u pt) g)`.

The `dsp` function is defined recursively as shown below.

```
(defun dsp (ts pt g)
  (cond ((endp ts) pt)
        (t (let ((u (choose-next ts pt g)))
             (dsp (del u ts)
                  (reassign u (neighbors u g) pt g)
                  g)))))
```

We can interpret this operationally as follows. To compute `(dsp ts pt g)`, ask whether `ts` is empty. If so, return `pt`. Otherwise, let `u` be the value of the `choose-next` expression and call `dsp` recursively on `(del u ts)`, the `reassign` expression, and `g`.

From the traditional description of the iterative core of the algorithm the reader should be able to infer the definitions of the functions used above.

Repeat until `ts` is empty:

- Choose `u` in `ts` such that `(d u pt g)` is minimal.
- For each edge from `u` to some neighbor `v` with edge length `w`, if `(d v pt g)` $>$ `(d u pt g)` $+$ `w`, then modify `pt` so that the path associated with `v` is the current path to `u` in `pt`, extended onward to `v`, `(append (path u pt) (list v))`.
- Delete `u` from `ts`.

[3] Some authors write "(weakly) shorter."

We then define Dijkstra's algorithm as

```
(defun dijkstra-shortest-path (a b g)
  (let ((pt (dsp (all-nodes g) (list (cons a (list a))) g)))
    (path b pt)))
```

which may be described as:

Let `pt` be the final path table computed by `dsp` starting from an initial `ts` containing all the nodes of the graph and an initial path table pairing the source vertex `a`, with the singleton path that starts and ends at `a`.
Return the path associated with `b` in the final path table.

Given that ACL2 is a functional programming language, this algorithm may be executed on concrete input. Using the techniques discussed in [6] it is probably possible to speed this implementation of Dijkstra's algorithm up to near C speeds, but that is not our aim here.

3 Specification

Our specification of the algorithm is

```
(defthm main-theorem
  (implies (and (nodep a g)
                (nodep b g)
                (graphp g))
           (let ((rho (dijkstra-shortest-path a b g)))
             (and (or (null rho)
                      (pathp-from-to rho a b g))
                  (shortest-pathp a b rho g)))))
```

That is, suppose `a` and `b` are nodes in graph `g`. Let ρ be the output of Dijkstra's algorithm on `a`, `b`, and `g`. Then ρ is either `nil` or a path in `g` from `a` to `b`, and ρ is a (weakly) shortest path from `a` to `b` in `g`. Note that if ρ is `nil` the claim that it is nevertheless the shortest path from `a` to `b` is equivalent to the claim that there is no such path, since any true path from `a` to `b` is shorter than the infinite `path-len` of `nil`.

To formalize the idea that ρ is a (weakly) shortest path, we need universal quantification, which in ACL2 is rendered via the facility for defining "Skolem functions," `defun-sk` [8].

```
(defun-sk shortest-pathp (a b p g)
  (forall path (implies (pathp-from-to path a b g)
                        (shorterp p path g))))
```

The effect of this is a new axiom that ensures that `(shortest-pathp a b p g)` is true precisely if for every path, `path`, from `a` to `b` in `g`, `p` is (weakly) shorter than `path`. Thus, if we are given the assumption that `p` is a `shortest-pathp` then no path from `a` to `b` is strictly shorter. And if we wish to prove that `p` is a `shortest-pathp` we can do so by proving that it is shorter than some particular witness path from `a` to `b`. This witness path is given by the term `(shortest-pathp-witness a b p g)` and the Skolem function `shortest-pathp-witness` can be thought of as "trying" to construct a counterexample to the claim that `p` is the shortest path from `a` to `b`.

This witness function is used extensively in a series of hand-written hints used to carry out the most delicate arguments in the correctness proof. But while the various case splits and constructions used to conduct these arguments are the messiest part of the proof, the real crux of the proof is identifying and formalizing the invariant mentioned above.

4 The Invariant

The mechanical proof is mainly concerned with establishing an invariant on the temporary set, `ts` and the path table, `pt`, of `dsp`. The invariant also takes the starting vertex, `a`, and the graph, `g`.

Several concepts are used repeatedly in defining the invariant. One is the notion of the "final set," usually represented here by the variable `fs` and equal to the complement of the temporary set (with respect to the set of all nodes). Another is the idea of a path `p` being *confined* to `fs`, which means that every node in `p` except the last is a member of `fs`. We define the concept recursively.

```
(defun confinedp (p fs)
  (if (endp p) t
    (if (endp (cdr p)) t
      (and (memp (car p) fs)
           (confinedp (cdr p) fs)))))
```

A third important concept is that of `p` being a *shortest confined path*, meaning it is shorter than any path from `a` to `b` that is confined to `fs`. We need universal quantification (`defun-sk`) to formalize this.

```
(defun-sk shortest-confined-pathp (a b p fs g)
  (forall path (implies (and (pathp-from-to path a b g)
                             (confinedp path fs))
                        (shorterp p path g))))
```

We define the invariant as follows:

```
(defun invp (ts pt g a)
    (let ((fs (comp-set ts (all-nodes g))))
      (and (ts-propertyp a ts fs pt g)
           (fs-propertyp a fs fs pt g)
           (pt-propertyp a pt g))))
```

The invariant has three conjuncts, one each about the temporary set, the final set, and the path table, although this partitioning is somewhat artificial since all involve `fs` and `pt` to some extent.

We define `ts-propertyp` recursively to check that for every node in the temporary set, the path to that node in the path table is a shortest confined path to that node and the path is itself confined.

```
(defun ts-propertyp (a ts fs pt g)
  (if (endp ts) t
    (and (shortest-confined-pathp a (car ts)
                                    (path (car ts) pt)
                                    fs g)
         (confinedp (path (car ts) pt) fs)
         (ts-propertyp a (cdr ts) fs pt g))))
```

We define `fs-propertyp` recursively in a very similar fashion, except it checks that for every node in the final set, the path assigned to that node in the path table is a shortest path to that node and is confined.

```
(defun fs-propertyp (a fs fs0 pt g)
  (if (endp fs) t
    (and (shortest-pathp a (car fs) (path (car fs) pt) g)
         (confinedp (path (car fs) pt) fs0)
         (fs-propertyp a (cdr fs) fs0 pt g))))
```

Finally, we define `pt-propertyp` to check that for every entry in the path table is either `nil` or a path from `a` to the node with which it is associated in the table.

```
(defun pt-propertyp (a pt g)
  (if (endp pt) t
    (and (or (null (cdar pt))
             (pathp-from-to (cdar pt) a (caar pt) g))
         (pt-propertyp a (cdr pt) g))))
```

5 Mechanical Proof

The proof breaks down into two main lemmas. The first is that the invariant holds initially.

```
(defthm invp-0
  (implies (nodep a g)
           (invp (all-nodes g) (list (cons a (list a))) g a)))
```

The second is that the invariant holds as `dsp` recurs.

```
(defthm invp-choose-next
  (implies (and (invp ts pt g a)
                (my-subsetp ts (all-nodes g))
```

```
                (graphp g)
                (consp ts)
                (setp ts)
                (nodep a g)
                (equal (path a pt) (list a)))
           (let ((u (choose-next ts pt g)))
             (invp (del u ts)
                   (reassign u (neighbors u g) pt g)
                   g a)))
  :hints ...)
```

From these two, it is straightforward to prove

```
(defthm invp-last
  (implies (and (nodep a g)
                (graphp g))
           (invp nil
                 (dsp (all-nodes g)
                      (list (cons a (list a)))
                      g)
                 g a)))
```

and `main-theorem` follows without much more work.

6 Hints

The hardest part of the proof is, of course, the proof of `invp-choose-next`. We present only one of the major cases. `Dsp` uses `choose-next` to choose a vertex, `u`, in `ts` whose associated path in `pt` is of minimal length. Why is this path the shortest path to that vertex? Here is the lemma that states that it is.

```
(defthm choose-next-shortest
  (implies (and (graphp g)
                (consp ts)
                (my-subsetp ts (all-nodes g))
                (invp ts pt g a))
           (shortest-pathp a
                           (choose-next ts pt g)
                           (path (choose-next ts pt g) pt)
                           g))
  :hints ...)
```

ACL2 cannot prove this without help. Help is given by the user in the form of hints. We first describe the proof and then show the actual hints.

Let the `choose-next` term above be u and let its associated path in `pt` be δ. Let *fs* be the "final set," `(comp-set ts (all-nodes g))`. We know, from the `invp` hypothesis, that δ is the shortest path to u that is confined to *fs*. We wish to show it is the shortest path (confined or not). Suppose it is not. Then there is a shorter path, say σ, to u that is not confined to *fs*, i.e., σ contains a vertex v in `ts`. Let σ' be the initial portion of σ up to and including v. Then σ' is shorter than σ, terminates on a node in `ts`, and is confined to *fs*. But the path in `pt` associated with v is, by `invp`, shorter than σ'. And δ is shorter than that path by the selection criteria in `choose-next`. Hence, δ is shorter than σ.

The actual term for σ above is `(shortest-pathp-witness a` u δ `g)`. And the actual term for σ' is `(find-partial-path` σ *fs*`)`. `Find-partial-path` is a user-defined recursive function that finds the subpath of a path that terminates in the first node outside of *fs*.

Hints in ACL2 are generally coded by listing a series of instantiations of previously proved lemmas. These instances are conjoined to the hypotheses of the goal theorem and then used freely by ACL2. To code the above hint we tell ACL2 not to expand the definitions of `shorterp`, `path` and `pathp` and we provide two instances. The ellipsis in the display above for `choose-next-shortest` is filled in by:

```
(("Goal" :in-theory (disable shorterp path pathp)
  :use ((:instance pathp-partial-path (p σ) (s fs))
        (:instance shorterp-by-partial-and-choose-next
         (u u) (path σ') (v (car (last σ')))))))
```

The expression following the symbol `:use` specifies that the theorem prover is to add two lemma instances to the hypotheses of the goal. The first lemma, `pathp-partial-path`, instantiated above says that `find-partial-path` constructs a confined path to its last node. The given substitution replaces the variable symbol `p` in the lemma by σ and the variable `s` by *fs*. The second lemma says that if the path to `u` in `pt` is shorter than the path to `v` in `pt`, and `ts-propertyp` holds, and `path` is a confined path to `v`, then the path to `u` is shorter than `path`.

7 Some Details and Statistics

The entire proof script contains 39 `defun`s and 125 `defthm`s. The `defthm`s can be broken into to two broad categories: elementary lemmas about the basic ideas and "custom" lemmas for this particular proof. We classified as "custom" any lemma mentioning `choose-next`, `reassign`, `ts-propertyp`, `fs-propertyp`, `pt-propertyp`, `invp`, `dsp`, or `dijkstra-shortest-path`.

There are 68 elementary lemmas about finite set theory, the notions of shorter and shortest path, elementary path properties (including that of being confined) and manipulation (including the notion of finding a confined subpath), and structural properties of association lists, paths, tables, and graphs. Here are a few.

```
(defthm comp-set-id
  (equal (comp-set s s) nil))
(defthm neighbor-implies-nodep
  (implies (memp v (neighbors u g))
```

```
          (memp v (all-nodes g))))
(defthm shortest-pathp-corollary
  (implies (and (shortest-pathp a b p g)
                (pathp-from-to path a b g))
           (shorterp p path g)))
(defthm confinedp-append
  (implies (and (confinedp p s)
                (memp (car (last p)) s))
           (confinedp (append p (list v)) s)))
(defthm path-len-append
  (implies (pathp p g)
           (equal (path-len (append p (list v)) g)
                  (plus (path-len p g)
                        (edge-len (car (last p)) v g)))))
```

All are used by ACL2 as conditional rewrite rules. For example, the last theorem is used to rewrite `(path-len (append ...))` to the `plus` expression, provided `(pathp p g)` can be established. (`Plus` is just addition extended to handle `nil` as"infinity.")

There are 57 custom lemmas, including four shown in this paper: `invp-0`, `invp-choose-next`, `invp-last`, and `choose-next-shortest`. Some are easy to prove lemmas that "explain" the fact that functions like `ts-propertyp` are recursively defined quantifiers:

```
(defthm ts-propertyp-prop-lemma1
  (implies (and (ts-propertyp a ts fs pt g)
                (memp v ts))
           (and (shortest-confined-pathp a v (path v pt) fs g)
                (confinedp (path v pt) fs))))
```

In all, we had to give 51 hints. About 30 of these were hints only to disable (i.e., avoid using) certain definitions or theorems. Twenty-three times we had to instruct the theorem prover to `:use` instances of certain theorems, as illustrated above, and a total of 31 instances were mentioned in the script. The vast majority of the hints were used in the custom theorems: 37 of the 51 hints, 19 of the 23 `:use` hints for 28 of the 31 instances.

The proof takes about 67 seconds on a 2.4 GHz Intel Xeon^{TM} running ACL2 Version 2.9 compiled under GNU Common Lisp.

References

1. Jing-Chao Chen. Dijkstra's shortest path algorithm. *Journal of Formalized Mathematics*, vol. 15, 2003.
2. E. W. Dijkstra. A note on two problems in connection with graphs. *Numer. Math. 1*, pages 269–271, 1959.
3. Shimon Even. *Graph Algorithms*, chapter 1. Computer Science Press, Inc., 1979.

4. Eric Fleury. Implantation des algorithmes de floyd et de dijkstra dans le Calcul des Constructions. Rapport de Stage, July 1990.
5. M. Gordon, J. Hurd, and K. Slind. Executing the formal semantics of the accellera property specification language by mechanized theorem proving. In D. Geist, editor, *Proceedings of CHARME 2003*, volume 2860 of *Lecture Notes in Computer Science*, pages 200–215. Springer Verlag, 2003.
6. D. Greve and M. Wilding. Using mbe to speed a verified graph pathfinder. In *ACL2 Workshop 2003*, Boulder, Colorado, July 2003. http://www.cs.utexas.edu/users/moore/acl2/workshop-2003/.
7. M. Kaufmann, P. Manolios, and J S. Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Press, Boston, MA., 2000.
8. M. Kaufmann and J S. Moore. Structured theory development for a mechanized logic. *Journal of Automated Reasoning*, 26(2):161–203, 2001.
9. C. Paulin and J. C. Filliâtre. http://pauillac.inria.fr/cdrom/www/coq/contribs/-floyd.html.