Graph Algorithms

Jan van Eijck
CWI & ILLC, Amsterdam

ESSLLI, Opole, August 9, 2012

Abstract

This lecture discusses a number of well-known graph algorithms, develops testable specifications for them, and uses the specifications to write assertive (self-testing) versions of the algorithms.

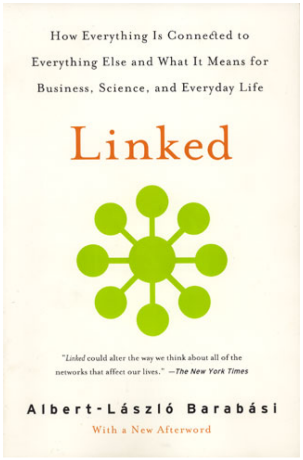
```
module GraphsAlgs

where

import List
import While
import Assert
import Reasoning (update,updates)
```

The modules `While` and `Assert` give the code for while loops and for assertion and invariant wrappers that was developed and discussed in the lecture on Algorithm Specification.

Background on the key importance of graph theory for the analysis of what goes on in social and other networks can be found in [1]. An enlightening introduction to graph algorithms is [4].

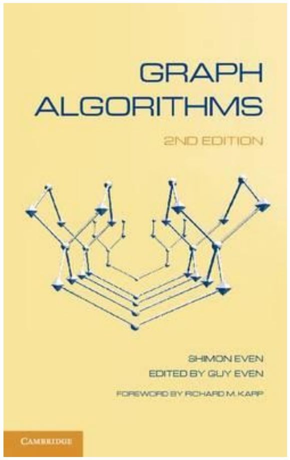

How Everything Is Connected to Everything Else and What It Means for Business, Science, and Everyday Life

Linked

"*Linked* could alter the way we think about all of the networks that affect our lives." —*The New York Times*

Albert-László Barabási

With a New Afterword


GRAPH ALGORITHMS

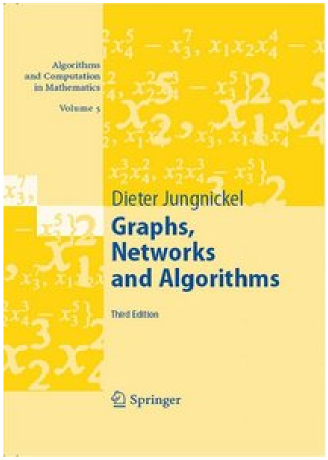
2ND EDITION

SHIMON EVEN

EDITED BY GUY EVEN

FOREWORD BY RICHARD M. KARP

CAMBRIDGE


Algorithms and Computation in Mathematics

Volume 5

Dieter Jungnickel

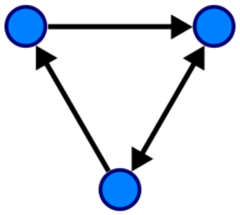
Graphs, Networks and Algorithms

Third Edition

Springer

A Graph Reachability Algorithm

A directed graph G without parallel edges is a pair (V, E) with $E \subseteq V^2$. If $(v_1, v_2) \in E$, we write this as $v_1 \rightarrow v_2$, and we say that there is an edge from v_1 to v_2.



A vertex y is reachable from a vertex x in $G = (V, E)$ if there is a path of $\rightarrow$ edges from x to y, or, equivalently, if $(x, y) \in E^*$, where E^* is the reflexive transitive closure of E.

In an algorithm for this, we can assume G is given by its edge set E.

Algorithm for Graph Reachability

Here is an algorithm that computes the set of reachable vertices from a given vertex.

Graph reachability algorithm

```
• Let edge set $E$ and vertex $x$ be given;
• $C := \{x\}$, $M := \{x\}$;
• while $C \neq \emptyset$ do:
    – select $y \in C$;
    – $N := \{z \mid (y, z) \in E, z \notin M\}$;
    – $C := (C - \{y\}) \cup N$;
    – $M := M \cup N$;
• return $M$.
```

Explanation

The algorithm works by maintaining a set C of current nodes. Initially, only x is current. Each node can be either marked (in M) or unmarked. The marked nodes are the nodes that have been current (have been in C) at some stage in the past, or are current now. Each step in the algorithm removes an item y from C, and adds all its unmarked successors to C, and to M. The algorithm halts when C becomes empty, at which stage M gives the nodes that are reachable from x.

Note that the while loop in the algorithm has two parameters, C and M; the variable N is local to the loop.

Haskell Version

```
reachable :: Eq a => [(a,a)] -> a -> [a]
reachable g x = reachable' g [x] [x]

reachable' :: Eq a => [(a,a)] -> [a] -> [a] -> [a]
reachable' g = while2
    (\ current _ -> not (null current))
    (\ current marked -> let
      (y,rest) = (head current, tail current)
      newnodes = [ z  | (u,z) <- g, u == y,
                                notElem z marked   ]
      current' = rest ++ newnodes
      marked'  = marked ++ newnodes
  in
      (current', marked'))
```

Strong Connectedness of a Graph

A directed graph is (strongly) connected if for every pair of vertices x, y, there is a path from x to y. Here is an implementation in terms of reachability.

```
isConnected :: Eq a => [(a,a)] -> Bool
isConnected g = let
    xs = map fst g `union` map snd g
  in
    forall xs (\x -> forall xs (\ y ->
       elem y (reachable g x)))
```

Operations on Relations

The algorithm from the previous section should fit the following specification:

$$\text{reachable } E\ x = \{y \mid xE^*y\}$$

where E^* is the reflexive transitive closure of E.

To implement this specification, we need some operations on relations.

Binary relations as lists of ordered pairs:

```
type Rel a = [(a,a)]
```

Inclusion and Equality

The $\subseteq$ relation on sets (represented by lists):

```
containedIn :: Eq a => [a] -> [a] -> Bool
containedIn xs ys = forall xs (\ x -> elem x ys)
```

Set equality on sets is defined as $A = B \iff A \subseteq B \wedge B \subseteq A$. Implementation for sets represented as lists:

```
equalS :: Eq a => [a] -> [a] -> Bool
equalS xs ys = containedIn xs ys && containedIn ys xs
```

Relational Composition

The relational composition of two relations R and S on a set A:

$$R \circ S = \{(x,z) \mid \exists y \in A\ (xRy \wedge ySz)\}$$

For the implementation, it is useful to declare a new infix operator for relational composition.

```
infixr 5 @@

(@@) :: Eq a => Rel a -> Rel a -> Rel a
r @@ s =
  nub [ (x,z) | (x,y) <- r, (w,z) <- s, y == w ]
```

Least Fixpoint

Computing reflexive transitive closure of a binary relation: the reflexive transive closure of R is the relation that is the least fixpoint of the operation $\lambda S \mapsto S \cup (R \circ S)$, applied to the identity relation I.

Here is the least fixpoint operation on functions:

```
lfp :: Eq a => (a -> a) -> a -> a
lfp f x | x == f x  = x
        | otherwise = lfp f (f x)
```

Application to Compute Reflexive Transitive Closure

To apply the least fixpoint of $\lambda S \mapsto S \cup (R \circ S)$, to I, you start with I.

In the first step you apply $\lambda S \mapsto S \cup (R \circ S)$ to I. This gives $I \cup (R \circ I)$, that is: $I \cup R$.

In the second step you apply $\lambda S \mapsto S \cup (R \circ S)$ to $I \cup R$. This gives

$$I \cup R \cup (R \circ (I \cup R)),$$

that is: $I \cup R \cup R^2$.

And so on, until the process reaches a fixpoint, that is, until there is some n for which

$$I \cup R \cup R^2 \cup \cdots \cup R^n = I \cup R \cup R^2 \cup \cdots \cup R^n \cup R^{n+1}.$$

At this point you have computed the smallest relation S that is reflexive and transitive and contains R, i.e., you have reached the reflexive transitive closure of R.

Implementation of Reflexive Transitive Closure

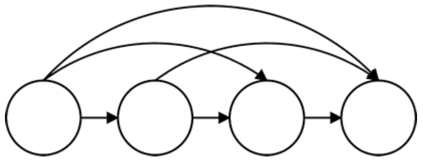
Here is the implementation (if you understand the code you can forget about the elaborate explanation above):

```
domain :: Eq a => Rel a -> [a]
domain r = nub (foldr (\ (x,y) -> ([x,y]++)) [] r)
```

```
rtc :: Eq a => Rel a -> Rel a
rtc r = let
          xs = domain r
          i  = [(x,x) | x <- xs ]
        in lfp (\ s -> (s `union` (r@@s))) i
```

Transitive Closure

Computing transitive closure: same as computing reflexive transitive closure, but starting out from the relation R. Notation for the transitive closure of R is R^+.



```
tc :: Eq a => Rel a -> Rel a
tc r = lfp (\ s -> (s `union` (r @@ s))) r
```

xR

If R is a binary relation on A and $x \in A$, then xR is the set $\{y \mid xRy\}$. Using this notation we can write the specification for `reachable` as:

$$\text{reachable } E\ x = xE^*.$$

Implementation of xR:

```
image :: Eq a => a -> Rel a -> [a]
image x r = [ y | (z,y) <- r, x == z ]
```

Assertive Version of Graph Reachability Algorithm

Use this for an assertive version of the graph reachability algorithm:

```
reachableA :: Eq a =>  [(a,a)] -> a -> [a]
reachableA =
  assert2 (\ g x ys -> equalS ys (image x (rtc g)))
           reachable
```

Note that this uses an inefficient algorithm for computing E^* to specify and test an efficient algorithm for E^*.

Minimum Spanning Trees of Graphs

A weighted undirected graph is a symmetric graph with weights assigned to the edges, in such manner that edges (x, y) and (y, x) have the same weight. Think of the weight as an indication of distance.

Here is a datatype for weighted graphs:

```
type Vertex = Int
type Edge = (Vertex,Vertex,Float)
type Graph = ([Vertex],[Edge])
```

If (x, y, w) is an edge, then the edge is from vertex x to vertex y, and its weight is w.

Proper Symmetric Graphs

The following function makes a list of edges into a proper symmetric graph, while also removing self loops and edges with non-positive weights.

```
mkproper :: [Edge] -> [Edge]
mkproper xs = let
  ys = List.filter
    (\ (x,y,w) -> x /= y && w > 0) xs
  zs = nubBy (\ (x,y,_) (x',y',_) ->
    (x,y) == (x',y') || (x,y) == (y',x')) ys
  in foldr
      (\ (x,y,w) us -> ((x,y,w):(y,x,w):us))
      [] zs
```

Example

This gives, e.g.:

```
*GA> mkproper [(1,2,3),(2,3,4),(3,2,-5)]
[(1,2,3),(2,1,3),(2,3,4),(3,2,4)]
```

Getting the vertices from an edge list

```
nodes :: (Eq a,Ord a) => [(a,a,b)] -> [a]
nodes = sort.nub.nds where
  nds = foldr (\ (x,y,_) -> ([x,y]++)) []
```

Making a graph from a list of edges:

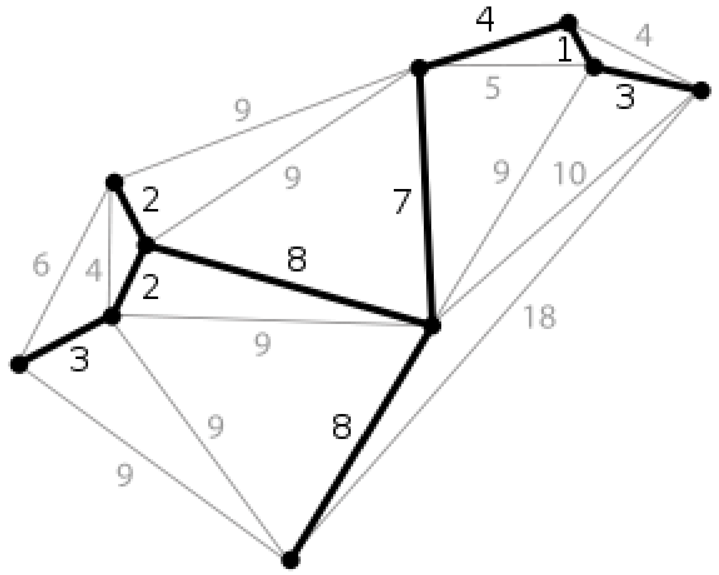
```
mkGraph ::  [Edge] -> Graph
mkGraph es = let
   new = mkproper es
   vs  = nodes new
 in (vs,new)
```

Spanning Trees, Minimum Spanning Trees

Let G be a weighted, undirected (i.e., symmetric) and connected graph. Assume there are no self-loops. (Or, if there are self-loops, make sure their weight is set to 0.)

A spanning tree for weighted graph G is a tree with a node set that coincides with the vertex set of the graph, and an edge set that is a subset of the set of edges of the graph.

A minimum spanning tree for weighted graph G is a spanning tree of G whose edges sum to minimum weight. Caution: minimum spanning trees are not unique.



Applications: finding the least amount of wire necessary to connect a group of workstations (or homes, or cities, or . . .).

Prim’s Algorithm

Prim’s minimum spanning tree algorithm finds a minimum spanning tree for an arbitrary weighted symmetric and connected graph. See [7], [8, 4.7].

Prim’s minimum spanning tree algorithm

1. Select an arbitrary graph node r to start the tree from.
2. While there are still nodes not in the tree
   - (a) Select an edge of minimum weight between a tree and non-tree node.
   - (b) Add the selected edge and vertex to the tree.

Prim, More Formally

A more formal version of this uses variables V_{in} for the current vertex set of the tree, V_{out} for the current vertex set outside the tree, and E_{tree} for the current edge set of the tree.

Prim's minimal spanning tree algorithm (ii)

1. Let $G = (V, E)$ be given.
2. Select an arbitrary graph vertex $r \in V$ to start the tree from. $V_{\text{in}} := \{r\}$, $V_{\text{out}} := V - \{r\}$, $E_{\text{tree}} := \emptyset$.
3. While $V_{\text{out}} \neq \emptyset$ (there are still nodes not in the tree):
   (a) Select a member (x, y, w) from E with $x \in V_{\text{in}}$, $y \in V_{\text{out}}$, such that there is no $(x', y', w') \in E$ with $x' \in V_{\text{in}}$, $y' \in V_{\text{out}}$ and $w' < w$ (the edge (x, y, w) is an edge of minimum weight between a tree and non-tree node).
   (b) $V_{\text{in}} := V_{\text{in}} \cup \{y\}$, $V_{\text{out}} := V_{\text{out}} - \{y\}$, $E_{\text{tree}} := E_{\text{tree}} \cup \{(x, y, w)\}$.
4. Return E_{tree}.

Remarks about the Algorithm

Note that the assumption that the graph is connected ensures that the "select a member (x, y, w) from E ..." action succeeds.

It is not at first sight obvious that Prim's algorithm always results in a minimum spanning tree, but this fact can be checked by means of Hoare assertions, which can be tested.

Implementation

Element in a non-empty list with minimal f-value:

```
minim :: (Ord b) => (a -> b) -> [a] -> a
minim f = head .
          (sortBy (\ x y -> compare (f x) (f y)))
```

Prim’s Algorithm, initialisation:

```
prim :: [Edge] -> [Edge]
prim es = let
    (v:vs) = nodes es
    vout    = vs
    vin     = [v]
    tree    = []
 in prim' es vout vin tree
```

Prim’s Algorithm, while loop

```
prim’ :: [Edge] -> [Vertex] -> [Vertex] -> [Edge]
      -> [Edge]
prim’ es = while3
    (\ vout _ _ -> not (null vout))
    (\ vout vin tree -> let
        links = [(x,y,w) | (x,y,w) <- es,
                           elem x vin, elem y vout ]
        e@(x,y,w) = minim (\ (_, _, w) -> w) links
      in
        (vout\\[y],y:vin,e:tree))
```

Reasoning about Prim

How can we see that this is correct? Here is one way. The property to prove is:

If `es` is the edge list of a symmetric connected weighted graph G, then `prim es` gives the edges of a minimum spanning tree on G.

Let's look at subgraphs of G. If $G = (V, E)$, and $A \subseteq V$, then let G^A be the subgraph with node set A and edge set

$$\{(x, y, w) \mid (x, y, w) \in E, x \in A, y \in A\}.$$

We prove the following. At every stage of the algorithm,

```
prim' es vout vin tree
```

`tree` is a minimum spanning tree on the subgraph of G given by $G^{\texttt{vin}}$.

Proof by Induction

Induction on the size of `vin`.

Base case. `prim'` is initialized with `vin = [v]` and `tree = []`. This is correct, for $(\{v\}, \emptyset)$ is a minimum spanning tree for $G^{\{v\}}$.

Induction step. Assume at stage `prim' es vout vin tree` the edges in `tree` form a minimum spanning tree for $G^{\texttt{vin}}$. Let $e = (x, y, w) \in$ `es` be a link with $x \in$ `vin`, $y \in$ `vout`, with minimum weight. Then the next stage of the algorithm is

```
prim' es (vout\\[y]) (y:vin) (e:tree).
```

We have to show that the edges in `e:tree` are a minimum spanning tree for $G^{\texttt{(y:vin)}}$. Suppose they are not. Then, since `tree` is a minimum spanning tree for $G^{\texttt{vin}}$, there is some vertex x' inside `vin` and an edge $(x', y, w') \in E$ such that $w' < w$. Contradiction with the way in which the link (x, y, w) was selected.

Extracting Assertions from the Proof

Now that we have seen the proof, we can also turn it into an assertion or a loop invariant. Here is the specification for a minimal spanning subtree of a graph.

```
mst :: [Edge] -> [Edge] -> Bool
mst _ [] = True
mst g [e] = elem e g
mst g (e@(x,y,w):es) = let
    vs = nodes es
  in
     elem e g
     && mst g es
     && elem x vs
     && notElem y vs
     && forall g  (\ (x',y',w') ->
          (elem x' vs && notElem y' vs) ==> w <= w')
```

Self-Testing Version of Prim

Use this for a self-testing version of `prim`:

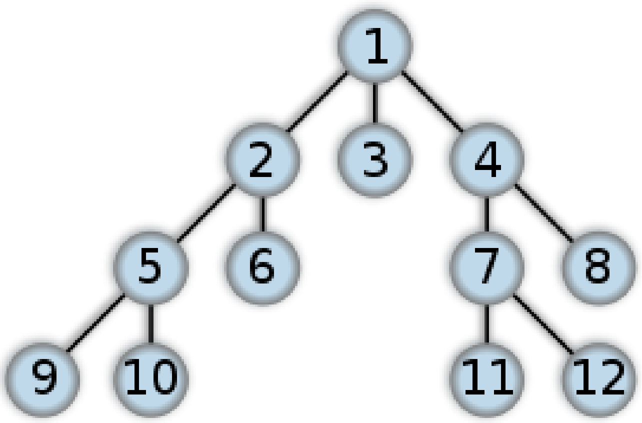
```
primA :: [Edge] -> [Edge]
primA = assert1 mst prim
```

We can also use it to formulate a loop invariant

```
prim'' :: [Edge] -> [Vertex] -> [Vertex] -> [Edge]
       -> [Edge]
prim'' es = while3
   (\ vout _ _ -> not (null vout))
   (invar3 (\ _ _ tree -> mst es tree)
   (\ vout vin tree -> let
        links = [(x,y,w) | (x,y,w) <- es,
                                    elem x vin, elem y vout ]
        e@(x,y,w) = minim (\ (_, _, w) -> w) links
      in
        (vout\\[y],y:vin,e:tree)))
```

Shortest Path Algorithms

For finding the shortest path between two vertices in an unweighted graph we can use breadth first search.



In this case the distance $d(x, y)$ between two vertices x and y in the graph is given as the length (number of edges) in the shortest path from x to y.

In case there is no path from x to y, $d(x, y)$ is undefined.

Breadth first search algorithm

```
1. Let $G$ be given by its edge set $E$. Let $s$ be some vertex of $G$.
2. $Q := \{s\}, T := \{s\}, d(s) := 0$.
3. while $Q \neq \emptyset$ do:
    (a) take some $u \in Q$ and let $Q := Q - \{u\}$;
    (b) for every edge $u \to v \in E$ with $v \notin T$ do:
        i. $T := T \cup \{v\}$,
        ii. $d(v) := d(u) + 1$,
        iii. $Q := Q \cup \{v\}$.
4. Return the function $d$.
```

Initialisation of the breadth first search

The initial distance function gives $ds = 0$ for the source node, and everywhere else $\uparrow$:

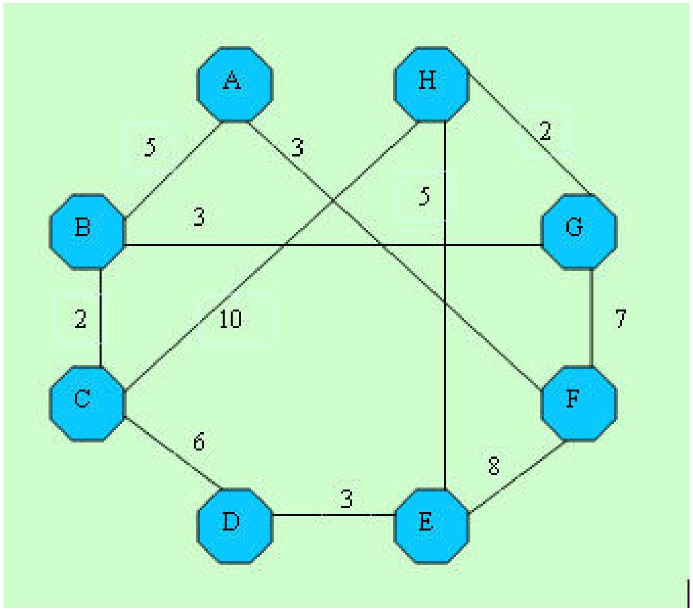
```
bfs :: Eq a => [(a,a)] -> a -> a -> Int
bfs g s = let
    d = update (\_ -> undefined) (s,0)
  in bfs' g [s] [s] d
```

While loop of the breadth first search

```
bfs' :: Eq a => [(a,a)] -> [a] -> [a]
             -> (a -> Int) -> a -> Int
bfs' g = while3 (\ q _ _ -> not (null q))
                (\ (u:q) t d -> let
                    new = [ y | (x,y) <- g,
                                  x == u, notElem y t ]
                    q' = q ++ new
                    t' = t `union` new
                    pairs = [(v, d u + 1) | v <- new ]
                    d' = updates d pairs
                  in (q',t',d'))
```

Dijkstra’s Algorithm

In case the edges have positive weights, a modification of this algorithm works for finding shortest paths. This is called Dijkstra’s algorithm [3].



Example eighted graph

Same as the picture, but with the nodes A, B, $\ldots$ renamed as 0, 1, $\ldots$.

```
exampleG = mkproper
             [(0,1,5),(0,5,3),(1,2,2),(1,6,3),(2,3,6),
              (2,7,10),(3,4,3),(4,5,8),(5,6,7),(6,7,2)]
```

Dijkstra's algorithm

```
1. Let $G$ be given by its edge set $E$. Let $s$ be some vertex of $G$.
2. $T := \{s\}$, $P := \emptyset$, $d(s) := 0$.
3. while $T \neq \emptyset$ do:
   (a) choose a $v \in T$ for which $d(v)$ is minimal;
   (b) $T := T - \{v\}$;
   (c) $P := P \cup \{v\}$;
   (d) for every edge $(v, u, w) \in E$ do:
       if $u \in T$ then $d(u) := \min(d(u), d(v) + w)$
       else if $u \notin P$ then do:
         i. $d(u) := d(v) + w$;
         ii. $T := T \cup \{u\}$
4. Return the function $d$.
```

Note throughout the execution of the algorithm, the function d is defined for all members of T and P.

In other words:

$$\{v \in V \mid d(v) \neq \uparrow\} = T \cup P$$

is an invariant of the algorithm.

Implementation

Implementation of Dijkstra’s algorithm: initialisation.

```
dijkstra :: [Edge] -> Vertex -> Vertex -> Float
dijkstra es s = let
    t = [s]
    p = []
    d = (\ x -> if x == s then 0 else undefined)
  in
    dijkstra' es t p d
```

Implementation, Ctd

Implementation of Dijkstra's algorithm: while loop.

```
dijkstra' :: [Edge] -> [Vertex] -> [Vertex]
             -> (Vertex -> Float) -> Vertex -> Float
dijkstra' es = while3
   (\ t _ _ -> not (null t))
   (\ t p d -> let
        v = minim (\ x -> d x) t
        pairs = [(u,w)| (x,u,w) <- es, x == v]
        old   = [(u, min (d u) (d v + w)) |
                          (u,w) <- pairs, elem u t ]
        new   = [(u, d v + w) | (u,w) <- pairs,
                            notElem u t, notElem u p ]
        d'    = updates d (old ++ new)
        t'    = (t \\ [v]) `union` (map fst new)
     in (t',v:p,d'))
```

A Check

In Haskell, we cannot test for definedness without the risk of generating errors, because undefined is implemented as error.

Instead we can assert that a distance function is defined on a given list of inputs by checking that it computes a value ≥ 0, as follows:

```
definedOn :: (Num b,Ord b) => [a] -> (a -> b) -> Bool
definedOn xs f = and (map (triv.f) xs)
  where triv = (\ x -> x >= 0)
```

An invariant of `dijkstra'` is that its d function is defined on its parameters T and P.

Assertive Version of Dijkstra’s Algorithm

```
dijkstraA :: [Edge] -> Vertex -> Vertex -> Float
dijkstraA es s = let
    t = [s]
    p = []
    d = (\ x -> if x == s then 0 else undefined)
  in
    dijkstraA' es t p d
```

Assertive Version of Dijkstra’s Algorithm, Ctd

```
dijkstraA’ :: [Edge] -> [Vertex] -> [Vertex]
             -> (Vertex -> Float) -> Vertex -> Float
dijkstraA’ es = while3
   (\ t _ _ -> not (null t))
   (invar3 (\ t p d -> definedOn (t ‘union‘ p) d)
   (\ t p d -> let
         v = minim (\ x -> d x) t
         pairs = [(u,w)| (x,u,w) <- es, x == v]
         old   = [(u, min (d u) (d v + w)) |
                         (u,w) <- pairs, elem u t ]
         new   = [(u, d v + w) | (u,w) <- pairs,
                             notElem u t, notElem u p ]
         d’    = updates d (old ++ new)
         t’    = (t \\ [v]) ‘union‘ (map fst new)
      in (t’,v:p,d’)))
```

The Floyd-Warshall Algorithm

Another algorithm for shortest path is the Floyd-Warshall algorithm, which computes a distance table for the whole graph. Assume the graph is given as a node set N plus a distance function f that gives the distance between x and y if there is a direct link in the graph from x to y, and 0 otherwise.

Conversions

```
edges2fct :: [(Int, Int, Float)]
          -> Int -> Int -> Float
edges2fct es i j = let
   links = filter (\ (x,y,w) -> x == i && y == j) es
   ws = map (\ (_,_,w) -> w) links
  in
   if null ws then 0 else head ws
```

And conversely:

```
fct2edges :: Int -> (Int->Int->Float)
          -> [(Int, Int, Float)]
fct2edges n f =
  [(x,y,f x y) | x <- [0..n-1], y <- [0..n-1],
                 f x y > 0 ]
```

The Algorithm

Let $\{0, \ldots, N-1\}$ be the nodes in the graph.

The algorithm computes shortest distances in subgraphs $\{0, \ldots, k-1\}$ for $k \in \{0, \ldots, N-1\}$.

The nodes in $\{0, \ldots, k-1\}$ are the nodes that are available for stopovers.

The output of the algorithm is a function $d : N^3 \to \mathbb{R}$, where $d(k, i, j)$ gives the shortest distance in G between i and j when all stopover nodes are $< k$.

Initially, no nodes are available for stopovers, so we have $d(0, i, j) = f(i, j)$.

The algorithm in pseudo-code

Floyd-Warshall algorithm for shortest path

1. Let $N = \{0, \ldots, N-1\}$ be the nodes of a graph G. Let f be the distance function for G.
2. For each $(i, j) \in N^2$, let $d(0, i, j) :=$ if $f(i, j) \neq 0$ then $f(i, j)$ else ∞.
3. For each $k \in N$, all $(i, j) \in N^2$,
   let $d(k+1, i, j) := \min(d(k, i, j), d(k, i, k) + d(k, k, j))$.

This construction, by the way, is the cornerstone of Kleene's famous theorem stating that languages generated by automata are regular [6]. In fact, Floyd's and Warshaw's descriptions of the algorithm were provided later (in [5] and [9], respectively).

Haskell Version

Here is the Haskell version. First a definition of ∞:

```
infty :: Float
infty = 1/0
```

Initialisation of the distance table:

```
fw :: Int -> (Int -> Int -> Float)
   -> Int -> Int -> Float
fw n f = let
    init = \ i j ->
             if f i j > 0 then f i j else infty
 in
    fw' n init
```

Update

For the computation of the distance table we will use a variation on `update`:

```
update2 :: (Eq a,Eq b) => (a -> b -> c) -> (a,b,c)
        -> a -> b -> c
update2 f (u,v,w) x y =
    if u == x && v == y then w else f x y
```

Computation of the distance table

Computation by means of successive updates of the initial distance table:

```
fw' ::  Int -> (Int -> Int -> Float)
              -> Int -> Int -> Float
fw' n = let
    nodes = [0..n-1]
    pairs = [(i,j) | i <- nodes, j <- nodes]
  in
    for pairs (\ (i,j) ->
       for nodes (\ k d -> let
          k' = k-1
        in
          update2 d
              (i,j,min (d i j) (d i k' + d k' j))))
```

Assertive Version

We can use Dijkstra’s algorithm to write an assertive version of the Floyd-Warshall algorithm:

```
isShortest :: Int -> (Int -> Int -> Float)
           -> Int -> Int -> Float -> Bool
isShortest n f i j w = let
    g = fct2edges n f
  in
    w == dijkstra g i j
```

Assertive version of Floyd-Warshall:

```
fwA = assert4 isShortest fw
```

Some Checks

Try this out on an example graph function:

```
exampleF :: Int -> Int -> Float
exampleF = edges2fct exampleG
```

```
*GA> fwA 8 exampleF 0 4
11.0
```

Floyd-Warshall with Path Reconstruction

If one also wants to find the actual shortest path, a simple modification of the Floyd-Warshall algorithm suffices. An additional function next $: N^2 \to N$ is constructed, with next(i, j) equal to -1, to indicate that no intermediate point has been found on a shortest path yet.

Floyd-Warshall algorithm with path reconstruction

1. Let $N = \{0, \ldots, N-1\}$ be the nodes of a graph G. Let f be the distance function for G.
2. For all $(i, j) \in N^2$, let $d(0, i, j) :=$ if $f(i, j) \neq 0$ then $f(i, j)$ else ∞.
3. For all $(i, j) \in N^2$, let next$(i, j) := -1$.
4. For all $k \in N$, all $(i, j) \in N^2$,
   let $d(k+1, i, j) := \min(d(k, i, j), d(k, i, k) + d(k, k, j))$;
   if $d(k, i, k) + d(k, k, j)) < d(k, i, j)$ then next$(i, j) := k$.

Getting the Path

The path can now be reconstructed from the two functions d and next, as follows:

GetPath(i,j)

```
1. if $d(i,j) = \infty$ then return “no path”.
2. $k :=$ next$(i,j)$.
3. if $k = -1$ then return $[]$
   else return GetPath$(i,k)$ $+\!\!+[k]$ $+\!\!+$GetPath$(k,j)$.
```

Implementation

Implementation of the modified Floyd-Warshall algorithm: the modified algorithm returns a function that computes a pair (w, k) consisting of a distance w and an intermediate node k. Initialisation of the distance-next table:

```
mfw :: Int -> (Int -> Int -> Float) ->
       Int -> Int -> (Float, Int)
mfw n f = let
  init = \ i j ->
              (if f i j > 0 then f i j else infty, -1)
 in
    mfw' n init
```

Computation of the distance-next table

```
mfw' ::  Int -> (Int -> Int -> (Float, Int))
               -> Int -> Int -> (Float, Int)
mfw' n = let
    nodes = [0..n-1]
    pairs = [(i,j) | i <- nodes, j <- nodes]
  in
    for pairs (\ (i,j) ->
       for nodes (\ k d -> let
           k'  = k-1
           f   = \ x y -> fst (d x y)
         in
           if f i k' + f k' j < f i j then
             update2 d (i,j,(f i k' + f k'  j, k'))
           else d))
```

Reconstructing the path

```
getPath :: (Int -> Int -> (Float, Int))
        -> Int -> Int -> [Int]
getPath d i j = let
    dist = fst (d i j)
    k    = snd (d i j)
  in
    if dist == infty then error "no path"
    else if k == -1 then []
    else getPath d i k ++ [k] ++ getPath d k j
```

This gives

```
*GA> getPath (mfw 8 exampleF) 0 1
[]
*GA> getPath (mfw 8 exampleF) 0 2
[1]
*GA> getPath (mfw 8 exampleF) 0 3
[1,2]
*GA> getPath (mfw 8 exampleF) 0 4
[5]
*GA> getPath (mfw 8 exampleF) 0 5
[]
*GA> getPath (mfw 8 exampleF) 0 6
[1]
*GA> getPath (mfw 8 exampleF) 0 7
[1,6]
```

Warshall’s Algorithm for Transitive Closure

Warshall’s original version (see [9]) computed transitive closure for directed graphs without weights. First in pseudo-code:

Warshall’s algorithm for transitive closure

```
1. Let $N = \{0, \ldots, N-1\}$ be the nodes of a graph $G$.
   Let $r : N^2 \to \{T, F\}$ give its edge relation.

2. For all $(i, j) \in N^2$, let $t(0, i, j) := r(i, j)$, $t(k, i, j) :=\uparrow$ if $k > 0$.

3. For all $k \in N$, all $(i, j) \in N^2$,
   let $t(k+1, i, j) := t(k, i, j) \vee (t(k, i, k) \wedge t(k, k, j))$.
```

Implementation

Initialisation of the t function is not necessary, for it is given by the graph function itself.

```
warshall :: Int -> (Int->Int->Bool)
         -> Int -> Int -> Bool
warshall n = let
    nodes = [0..n-1]
    pairs = [(i,j) | i <- nodes, j <- nodes]
  in
    for pairs (\ (i,j) ->
       for nodes (\ k t -> let
         k' = k-1
       in
         update2 t
            (i,j, t i j || (t i k' && t k' j))))
```

Assertive Version

We can use our earlier definition of transitive closure to turn this into assertive code:

```
isTC :: Int -> (Int->Int->Bool) -> (Int->Int->Bool)
     -> Bool
isTC n r t = let
    r' = [(i,j)| i <- [0..n-1], j <- [0..n-1], r i j ]
    t' = [(i,j)| i <- [0..n-1], j <- [0..n-1], t i j ]
  in
    equalS t' (tc r')
```

```
warshallA :: Int -> (Int->Int->Bool) -> Int -> Int
          -> Bool
warshallA = assert2 isTC warshall
```

Shortest Path in the Presence of Negative Weights

It is not always possible to use one algorithm to check another in this simple way.

The Bellman-Ford algorithm for shortest path [2] serves the same purpose as Dijkstra's algorithm (computing shortest paths from a given vertex in a weighted directed graph), but for a wider class of graphs: unlike in the case of Dijkstra's algorithm, weights are allowed to be negative.

No Negative Cycles

To ensure that the notion of shortest path still makes sense in this context, it is assumed that the graph contains no negative cycles. A negative cycle in a weighted directed graph is a path

$$v_1 \xrightarrow{w_1} v_2 \xrightarrow{w_2} v_3 \longrightarrow \cdots \longrightarrow v_1$$

with the property that the weights on the path from v_1 to v_1 sum to a negative number. In the presence of negative cycles, the notion of shortest path loses its sense, for one could always make a path shorter by taking one more turn through a negative loop.

Belmann-Ford algorithm

Belmann-Ford algorithm for shortest path in the presence of negative weights

1. Let G be given as (V, E), with $E \subseteq V^2$. Let s be the source.
2. For each $v \in V$ do:
   (a) if $v = s$ then $d(v) := 0$ else $d(v) := \infty$;
   (b) if $(v, s) \in E$ then $p(v) := s$ else $p(v) := \perp$.
3. For i from 1 to $|V| - 1$ do:
   for each $u \xrightarrow{w} v$ in E do: if $d(u) + w < d(v)$ then:
   (a) $d(v) := d(u) + w$;
   (b) $p(v) := u$.
4. For each $u \xrightarrow{w} v$ in E:
   if $d(u) + w < d(v)$ then error "Graph contains a negative-weight cycle".

Implementation

The initialisation of the two functions:

```
bfInit :: Vertex -> [Edge]
       -> (Vertex -> Float,Vertex -> Vertex)
bfInit s es = let
    d  = update (\ _ -> infty) (s,0)
    p  = \ _ -> undefined
  in
    (d,p)
```

Implementation of the main “for” loop

```
bfLoop ::  [Edge]
       -> (Vertex -> Float,Vertex -> Vertex)
       -> (Vertex -> Float,Vertex -> Vertex)
bfLoop es = let
    ns = nodes es
    is = [1..length(ns) - 1]
  in
    for is (\ _ (d,p) -> let
          us    =
             filter (\ (u,v,w) -> d(u) + w < d(v)) es
          pairs = map (\ (u,v,w) -> (v,d(u) + w)) us
          vs    = map (\ (u,v,_) -> (v,u)) us
       in
          (updates d pairs, updates p vs))
```

The whole program

```
bf :: Vertex -> [Edge] ->
               (Vertex -> Float,Vertex -> Vertex)
bf s es = bfLoop es (bfInit s es)
```

‘Correcting’ the Order

```
infixl 2 ##

(##) :: a -> (a -> b) -> b
x ## f = f x
```

```
belmanFord :: Vertex -> [Edge] ->
              (Vertex -> Float,Vertex -> Vertex)
belmanFord s es = bfInit s es ## bfLoop es
```

Tracing Back

We can use the predecessor function to reconstruct the shortest path, by tracing back from the destination to the source, using the predecessor function:

```
traceBack :: (Vertex -> Vertex)
          -> Vertex ->  Vertex -> [Vertex]
traceBack p s t =
   if s == t then [s]
   else t : traceBack p s (p t)
```

This gives, e.g.:

```
*GA> traceBack (snd (bf exampleG 0 )) 0 7
[7,6,1,0]
```

Implementation of the check for negative cycles

```
bfCheck :: [Edge] -> (Vertex -> Float) -> Bool
bfCheck es d =
  forall es (\ (u,v,w) -> d u + w >= d v)
```

We can use the check for negative cycles — once we have convinced ourselves that it is correct — as an assertion about the computed distance function:

```
bfLoopA ::  [Edge]
        -> (Vertex -> Float,Vertex -> Vertex)
        -> (Vertex -> Float,Vertex -> Vertex)
bfLoopA =
    assert2 (\ es _ (d,_) -> bfCheck es d) bfLoop
```

The whole program again, with the assertion added

```
bfA :: [Edge] -> Vertex
    -> (Vertex -> Float,Vertex -> Vertex)
bfA es s = bfLoopA es (bfInit s es)
```

Summary

If we have several graph algorithms that perform similar computations (connectedness, transitive closure, shortest path), then we can use one algorithm as a specification for the other, and vice versa.

Assertive code provides documentation and tests at the same time. It describes what algorithms are supposed to compute, but it has to do so in terms of other algorithms.

If an assertive function raises an error both the implementation of the function and the implementation of the asserted specification can be at fault.

References

[1] Albert-László Barabási. Linked. Penguin, 2002.

[2] Richard Bellman. On a routing problem. Quarterly of Applied Mathematics, 16:87–90, 1958.

[3] E.W. Dijkstra. A note on two problems in connection with graphs. Numerische Mathematik, 1:269–271, 1959.

[4] Shimon Even. Graph Algorithms, 2nd Edition. Cambridge University Press, 2012.

[5] Robert W. Floyd. Algorithm 97: Shortest path. Communications of the ACM, 5(6):345, June 1962.

[6] S.C. Kleene. Representation of events in nerve nets and finite automata. In C. E. Shannon and J. McCarthy, editors, Automata Studies, pages 3–42. Princeton University Press, 1956.

[7] R.C. Prim. Shortest connection networks and some generalizations. Bell System Technical Journal, 36:1389–1401, 1957.

[8] Steven S. Skiena. The Algorithm Design Manual. Springer Verlag, New York, 1998. Second Printing.

[9] Stephen Warshall. A theorem on Boolean matrices. Journal of the ACM, 9(1):11–12, 1962.

[10] Jin Y. Yen. An algorithm for finding shortest routes from all source nodes to a given destination in general networks. Quarterly of Applied Mathematics, 27:526–530, 1970.