Graph traversal and shortest paths

Introduction

This document describes width-first traversal of a directed graph of which each node has a finite number of outbound branches. A graph will be represented by a function *G* that given a node *N* returns a list of all nodes having an inbound branch from node *N*. The list returned by *G* may contain duplicates (according to *equal?*) The graph may be finite or infinite. It may be a tree but it may contain cycles too.

Procedure: (*traverse-graph S G*)

For arbitrary node type:

S : *node*	Starting node.
G : *node* → (*node* **...**)	Graph.

Let *V* be the set of all nodes *N* for which there is at least one path from *S* to *N*, *S* included. Procedure *traverse-graph* calls *G* exactly once with every element of *V*. If *V* is finite, then procedure *traverse-graph* halts after having processed all elements of *V*. If *V* is infinite, then procedure *traverse-graph* never returns. The procedure guarantees that every arbitrary node *N* of *V* will be found and that *G* will be called exactly once with this node, even in case *V* is infinite. This requires width-first traversal of the graph. Later procedure traverse-graph will be modified such as to find all shortest paths between two nodes of a graph.

Why depth-first traversal does not work

Consider the following depth-first graph traversal procedure:

```
#lang scheme                                          File depth-first-traversal.rkt

;;; Why depth-first traversal does not work for infinite graphs.
;;; Does not avoid visiting nodes more than once.

(define (traverse-depth-first S G)
 (define (traverse-depth-first N) (for-each traverse-depth-first (G N)))
 (traverse-depth-first S))
```

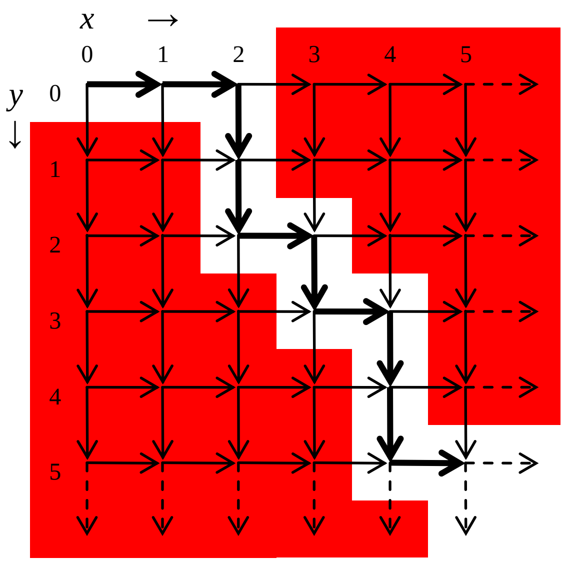
It is much simpler than a width-first traversal. However, depth-first traversal may fail on infinite graphs. Of course it is not possible to traverse an infinite graph all to the end. A method of traversal can be called complete if it satisfies the following property. Let *V* be the possibly infinite set of all nodes to which there is a path starting from a given node *N*. Given an arbitrary node $M \in V$, the method must guarantee that it will find node *M*. This is like counting natural numbers. Enumerating the numbers 0, 1, 2, etc. guarantees that no number will be skipped. In the following an infinite graph is shown on which depth-first traversal fails. The graph consists of all points (x,y) where x and y are natural numbers and every node (x,y) has a branch to node (x+1,y) and (x,y+1).

```
(define (make-point x y) (list x y))
(define x-coordinate car)
(define y-coordinate cadr)
```

```
(let/cc cc
 (let ((n 11))
  (define (G point)
   (if (zero? n) (cc) ; exit after having found 10 points
    (let ((x (x-coordinate point)) (y (y-coordinate point)))
     (set! n (sub1 n))
     (printf "Point found: ~s. " (list x y))
     (when (> x 0) (printf "All points with x<~s are lost. " x))
     (when (> y 0) (printf "All points with y<~s are lost. " y))
     (newline)
     (let ((down (make-point x (add1 y))) (right (make-point (add1 x) y)))
      (if (zero? (random 2)) ; return the two nodes in arbitrary order.
       (list down right)
       (list right down))))))
  (random-seed 0)
  (traverse-depth-first (make-point 0 0) G)))

#|
Displayed:
Point found: (0 0).
Point found: (1 0). All points with x<1 are lost.
Point found: (2 0). All points with x<2 are lost.
Point found: (2 1). All points with x<2 are lost. All points with y<1 are lost.
Point found: (2 2). All points with x<2 are lost. All points with y<2 are lost.
Point found: (3 2). All points with x<3 are lost. All points with y<2 are lost.
Point found: (3 3). All points with x<3 are lost. All points with y<3 are lost.
Point found: (4 3). All points with x<4 are lost. All points with y<3 are lost.
Point found: (4 4). All points with x<4 are lost. All points with y<4 are lost.
Point found: (4 5). All points with x<4 are lost. All points with y<5 are lost.
Point found: (5 5). All points with x<5 are lost. All points with y<5 are lost.
|#
```

The figure below shows what happens.

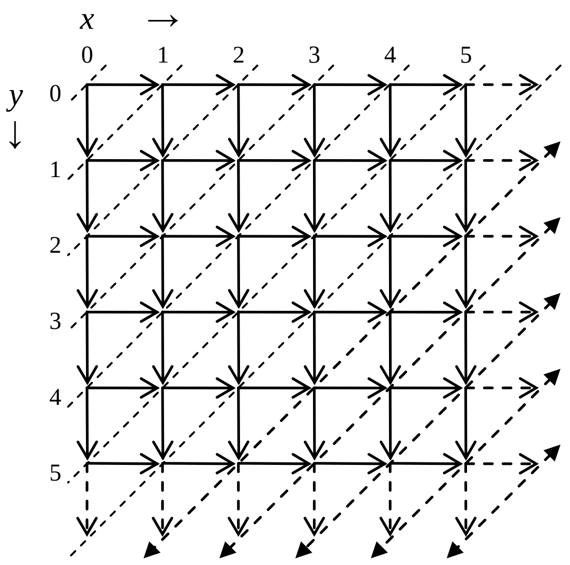


Random depth-first traversal

After reaching point (5,5) none of the points in the red areas will ever be found.

Width-first traversal

In the following the same graph is traversed in width-first order.



In width-first order the points are found along diagonals. For all points on the same diagonal the minimal number of branches required to reach these points starting from point (0,0) is the same. For each next diagonal the required number of branches is one more than for the previous diagonal.

Width-first order traversal requires that a frontier *F* of nodes is maintained. *F* should list all nodes found so far and whose outbound branches have not yet been followed. In the example above, the initial frontier consist of the first diagonal containing point (0,0) only. The second frontier is the second diagonal with the points (0,1) and (1,0). The *n*th diagonal contains all points (*x*,*y*) with $x+y=n$. Initially *F* contains node *S* only, id est, the set of nodes that can be reached from *S* without any branch. By calling *G* for each element of *F*, a new frontier is found. Nodes that have already been processed are removed from the lists returned by *G*, the lists are appended to each other and duplicates are removed in order to find the next frontier. After *n* iterations, *F* contains all nodes that can be reached along *n* branches but not along less than *n* branches. Let *V* be the set of all nodes *N* such that there is a path from the starting node to *N*. If *V* is finite the frontier will become empty after a finite number of iterations, in which case the procedure halts. If *V* is infinite the frontier never becomes empty and procedure *traverse-graph* never returns. Because *G* may contain cycles, it is necessary to memorize all nodes for which *G* has already been called.

File width-first-traversal.rkt

```
#lang scheme

; Procedure: (traverse-graph S G)
; For arbitrary node type N:
; S : N              starting node.
; G : N → (N ...)    graph.

(define (traverse-graph S G)
 (define H (make-hash)) ; Memorizes all nodes that have already been processed.
 (define (put! N) (hash-set! H N #f)) ; Marks node N as being found.
 (define (new? N) (not (hash-has-key? H N))) ; Do we have a new node?
 (define (filtered-G F) (filter new? (G F))) ; Returns new nodes only.
 (define (next-F F) ; F is the list of nodes whose outbound branches must yet be followed.
  (unless (null? F)
   (for-each put! F)
   (next-F (remove-duplicates (apply append (map filtered-G F))))))
 (next-F (list S)))

(define (make-point x y) (list x y))
(define x-coordinate car)
(define y-coordinate cadr)
```

```
(let/cc cc
 (let ((n 15))
  (define (G point)
   (if (zero? n) (cc) ; exit after having found 15 points
    (let ((x (x-coordinate point)) (y (y-coordinate point)))
     (set! n (sub1 n))
     (printf "Point found: ~s (diagonal ~s)~n" (list x y) (+ x y))
     (let ((down (make-point x (add1 y))) (right (make-point (add1 x) y)))
      (if (zero? (random 2)) ; return the two nodes in arbitrary order.
       (list down right)
       (list right down))))))
  (random-seed 0)
  (traverse-graph (make-point 0 0) G)))

#| Displayed:
Point found: (0 0) (diagonal 0)
Point found: (1 0) (diagonal 1)
Point found: (0 1) (diagonal 1)
Point found: (2 0) (diagonal 2)
Point found: (1 1) (diagonal 2)
Point found: (0 2) (diagonal 2)
Point found: (2 1) (diagonal 3)
Point found: (3 0) (diagonal 3)
Point found: (1 2) (diagonal 3)
Point found: (0 3) (diagonal 3)
Point found: (3 1) (diagonal 4)
Point found: (2 2) (diagonal 4)
Point found: (4 0) (diagonal 4)
Point found: (1 3) (diagonal 4)
Point found: (0 4) (diagonal 4) |#
```

In the following example the graph is infinite again. It contains exactly one node for each natural number and is almost a tree. Starting from 0, there is a path to every natural number. Each natural number *n* has one inbound branch from node floor(*n*/10) and 10 outbound branches to the nodes 10*n*+*i* for $0 \le i < 10$. The graph is traversed starting from 0. The example is protected by means of an escape continuation such as to halt when number *stop* is found. Procedure traverse-graph guarantees that it finds every arbitrary natural number given for the *stop* argument.

```
(define (test stop)
 (test-equal stop
  (let/cc cc
   (define H (make-hash))
   (define (G n)
    (cond
     ; Exit when stop is found.
     ((= n stop) (cc stop))
     ; Check that n has not yet been processed.
     ((hash-has-key? H n) (error 'test "duplicate node" n))
     ; Declare n processed and return natural numbers 10n+i for 0 ≤ i <10.
     (else (hash-set! H n #f)
      (let ((inc (let ((k (* 10 n))) (lambda (i) (+ k i)))))
       (build-list 10 inc)))))
     (traverse-graph 0 G))))

(require (only-in redex test-equal test-results))
(for ((stop (in-range 0 100))) (test stop))
(test-results)
```

Shortest paths in a graph

We consider directed graphs whose branches have length 1. Hence, the length of a path is the number of branches it contains. A path $N_0 \rightarrow N_1 \rightarrow N_2$ **...** will be represented by the list (N_0 N_1 N_2 **...**) or its reverse. The length of a path is one less than the length of the representing list. In procedure *traverse-graph*, a frontier *F* was used that after *n* iterations contains all nodes that can be reached along *n* branches but not along less than *n* branches. Therefore, the procedure can be adapted such as to find all minimum length paths from a given node *S* to a given node *T*. In this case, it can be necessary to call procedure *G* multiple times with the same node. The frontier *F* now must be a list of reversed paths (*Q* **...** *S*), each path being a list of nodes showing a path from *S* to *Q* in reversed order. Initially *F* is ((*S*)). If `(map car` *F*`)` contains node *T*, then all shortest paths from *S* to *T* have been found and all lists starting with *T* must be selected from *F*, reversed and returned. If *T* has not yet been found, apply *filtered-G* to *Q* for each path (*Q* **...** *S*) in *F*. Each node *N* returned by *filtered-G* gives rise to a new reversed path (*N Q* **...** *S*). The new paths become the new frontier for recursion.

```
;Procedure (shortest-paths S T G) → R                    File shortest-paths.rkt

;For any type N:
;S  : N                 starting node
;T  : N                 destination node
;G  : N → (N ...)       procedure representing a graph
;R  : ((S N ... T) ...) list of paths from S to T if S≠T
;R  : ((S))             list of paths if S=T

(define (shortest-paths S T G)
 (define H (make-hash))
 (define (put! P) (hash-set! H (car P) #f))
 (define (new? N) (not (hash-has-key? H N)))
 (define (filtered-G N) (filter new? (G N)))
 (define (extend P)
  (let ((Ns (remove-duplicates (filtered-G (car P)))))
   (map (lambda (N) (cons N P)) Ns)))
 (define (next-F F)
  (if (null? F) '() ; No paths found.
   (let ((Ps (filter (lambda (P) (equal? (car P) T)) F)))
    (if (pair? Ps) (map reverse Ps) ; The paths have been found.
     (begin (for-each put! F) ; Point T has not yet been found.
      (next-F (apply append (filter pair? (map extend F)))))))))
 (next-F (list (list S))))
```

In the following tests, shortest paths are looked for in a grid (*x*,*y*) where *x* and *y* are exact integer numbers and each node (*x*,*y*) has outbound branches to (*x*,*y*−1), (*x*,*y*+1), (*x*−1,*y*) and (*x*+1,*y*), id est, up, down, to the right and to the left. Let Δ*x* be the absolute difference between the *x*-coordinates of the starting and destination point and likewise Δ*y* the absolute difference of the *y*-coordinates. Then the number of shortest paths is $\binom{\Delta x+\Delta y}{\Delta x} = \binom{\Delta x+\Delta y}{\Delta y} = \frac{(\Delta x+\Delta y)!}{\Delta x!\,\Delta y!}$. The test procedure checks that the correct number of shortest paths is found.

```
(define (test-shortest-paths from to)
 (define (G P)
  (let ((x (x-coordinate P)) (y (y-coordinate P)))
   (list
    (make-point x (add1 y))
    (make-point x (sub1 y))
    (make-point (add1 x) y)
    (make-point (sub1 x) y))))
 (test-equal (length (shortest-paths from to G))
  (let
   ( (x (abs (- (x-coordinate from) (x-coordinate to))))
    (y (abs (- (y-coordinate from) (y-coordinate to)))))
   (binomial (+ x y) x))))

(define-syntax define/c ; For definition of caching functions.
 (syntax-rules ()
  ((_ (name arg ...) . body)
   (define name
    (let ((cache (make-hash)))
     (lambda (arg ...)
      (apply values
       (hash-ref! cache (list arg ...)
        (call-with-values (lambda () . body) list)))))))))

(define/c (binomial n k) (/ (factorial n) (factorial k) (factorial (- n k))))
(define/c (factorial k) (if (zero? k) 1 (* k (factorial (sub1 k)))))

(let*
 ((from (make-point 0 0))
  (range (in-range -5 6))
  (points (for*/list ((x range) (y range)) (make-point x y))))
 (for ((to points)) (test-shortest-paths from to)))

(test-results)
```