Structure and Interpretation of Computer Programs 2

Problem Set No. 4

Submitted by: Shlomi Fish
ID: 0-3386577-5

Homework Exercises

Problem 1.1

```
;;; Graph Description
;;; Graph == (a hash of node_ids => nodes ; value)
;;;     node == (node_id ;
;;;           hash of (dest node_id => edges) ;
;;;           hash of (src node_id => back edges) ;
;;;           value)
;;;
;;;     edge == (start_node_id ; end_node_id ; value)

(define-structure edge start end value)

(define-structure node id edges back-edges value)

(define (make-empty-node id)
    (make-node id (make-eq-hash-table 20) (make-eq-hash-table 20) #f)
)

(define-structure graph nodes value directed)

(define (make-empty-graph directed)
    (make-graph (make-eq-hash-table 20) #f directed)
)

(define (graph-add-node graph node-id)
    (let
        (
            (node (make-empty-node node-id))
            (the-graph-nodes (graph-nodes graph))
        )
        (hash-table/put! the-graph-nodes node-id node)
        node
    )
)

(define (graph-add-edge graph src dest)
    (let*
        (
            (edge (make-edge src dest #f))
            (the-graph-nodes (graph-nodes graph))
```

```
            (src-node (hash-table/get the-graph-nodes src #f))
            (dest-node (hash-table/get the-graph-nodes dest #f))
            (src-node-edges (node-edges src-node))
            (dest-node-edges (node-edges dest-node))
            (src-node-back-edges (node-back-edges src-node))
            (dest-node-back-edges (node-back-edges dest-node))
        )
        (hash-table/put! src-node-edges dest edge)
        (hash-table/put! dest-node-back-edges src edge)
        (if (not (graph-directed graph))
            (begin
                (hash-table/put! src-node-back-edges dest edge)
                (hash-table/put! dest-node-edges src edge)
                edge
            )
            ;;; Else - return a failure indication.
            #f
        )
    )
)

(define (graph-get-node g node-id)
    (hash-table/get (graph-nodes g) node-id #f)
)

(define (graph-get-list-of-nodes g)
    (hash-table/key-list (graph-nodes g))
)

(define (graph-get-first-node-id g)
    (car (graph-get-list-of-nodes g))
)

(define (graph-get-num-nodes g)
    (hash-table/count (graph-nodes g))
)

;;; Note - each edge may be repeated twice
(define (graph-get-list-of-edges g)
    (apply append
        (map
            (lambda (node) (hash-table/datum-list (node-edges node)))
            (graph-get-list-of-nodes g)
        )
    )
)

(define (graph-get-edge graph source dest)
    (hash-table/get (node-edges (graph-get-node graph source)) dest #f)
)

(define (graph-nodes-for-each g callback)
    (hash-table/for-each (graph-nodes g) callback)
```

```
)

(define (node-edges-for-each node callback)
    (hash-table/for-each (node-edges node) callback)
)
```

Problem 1.2

```
(load "graphs.scm")


(define g (make-empty-graph #f))
(graph-add-node g 'a)
(graph-add-node g 'b)
(graph-add-node g 'c)
(graph-add-node g 'd)
(graph-add-edge g 'a 'b)
(graph-add-edge g 'b 'c)
(graph-add-edge g 'a 'c)
(graph-add-edge g 'c 'd)

(define (print-graph g)
    (display "graph [value = ")
    (display (graph-value g))
    (display "]")
    (newline)
    (display "{")
    (newline)
    (graph-nodes-for-each g
        (lambda (node-id node)
            (display "    node ")
            (display node-id)
            (display " [value = ")
            (display "]")
            (newline)
            (display "    {")
            (newline)
            (node-edges-for-each node
                (lambda (dest-node edge)
                    (display "        -> ")
                    (display dest-node)
                    (display " [value = ")
                    (display (edge-value edge))
                    (display "]")
                    (newline)
                )
            )
            (display "    }")
            (newline)
        )
    )
    (display "}")
```

```
    (newline)
)

;(print-graph g)
```

Problem 1.3

One possible meaning is a graph that has a corresponding node for every node in the original graph and a corresponding edge for every such edge, and each edge or node has the same value as the orignal graph.

```
(load "prob1.2.scm")

(define (generic-copy g new-value-graph new-value-node new-value-edge)
    (define ret (make-empty-graph (graph-directed g)))
    (graph-nodes-for-each g
        (lambda (node-id node)
            (set-node-value! (graph-add-node ret node-id) (new-value-node node))
        )
    )
    (graph-nodes-for-each g
        (lambda (src-node-id src-node)
            (define ret-src-node (graph-get-node g src-node-id))
            (node-edges-for-each src-node
                (lambda (dest-node-id edge)
                    ;(if (not (hash-table/get (hash-table/get (node-edges (graph-nodes ret)) ret-src-no
                    (if (not (graph-get-edge ret src-node-id dest-node-id))
                        (set-edge-value! (graph-add-edge ret src-node-id dest-node-id) (new-value-edge e
                    )
                )
            )
        )
    )
    ret
)

(define (homologic-copy g)
    (define (unity x) x)
    (generic-copy g unity unity unity)
)

(define (value-copy g)
    (generic-copy g
        graph-value
        node-value
        edge-value
    )
)
```

Problem 1.4

A tree is a non-directed graph that contains no cycles.

```
(load "graphs.scm")


(define (traverse-tree g callback)
    (define first-node-id (graph-get-first-node-id g))
    (define (traverse node-id orig-node-id)
        (define node (graph-get-node g node-id))
        (callback node)
        (node-edges-for-each node
            (lambda (next-node-id edge)
                (if (not (eq? next-node-id orig-node-id))
                    (traverse next-node-id node-id)
                )
            )
        )
    )
    (traverse first-node-id #f)
)

(define g (make-empty-graph #f))
(graph-add-node g 'a)
(graph-add-node g 'b)
(graph-add-node g 'c)
(graph-add-node g 'd)
(graph-add-node g 'e)
(graph-add-edge g 'a 'b)
(graph-add-edge g 'b 'c)
(graph-add-edge g 'c 'd)
(graph-add-edge g 'b 'e)

(traverse-tree g (lambda (node) (display (node-id node)) (newline)))
```

Problem 1.5

```
(load "graphs.scm")

(define (get-cost edge)
    (let
        ((value (edge-value edge)))

        (if (pair? value)
            (car value)
            value
        )
    )
)
```

```
(define (is-marked? edge)
    (and (pair? (edge-value edge)) (cdr (edge-value edge)))
)

(define (mark edge)
    (set-edge-value! edge (cons (edge-value edge) #t))
)

(define (minimum-spanning-tree graph)
    (define encountered-nodes (make-eq-hash-table 20))
    (define first-node (car (graph-get-list-of-nodes graph)))
    (hash-table/put! encountered-nodes first-node 0)
    (define (iterate)
        (if (< (hash-table/count encountered-nodes) (graph-get-num-nodes graph))
            (let ()
                (define source #f)
                (define dest #f)
                (define weight 0)
                (hash-table/for-each encountered-nodes
                    (lambda (src-id no-use)
                        (node-edges-for-each (graph-get-node graph src-id)
                            (lambda (dest-id edge)
                                (if (and (not (hash-table/get encountered-nodes dest-id #f))
                                         (or (not source) (> weight (get-cost edge)))
                                    )
                                    (begin
                                        (set! source src-id)
                                        (set! dest dest-id)
                                        (set! weight (get-cost edge))
                                    )
                                )
                            )
                        )
                    )
                )
                (mark (graph-get-edge graph source dest))
                (hash-table/put! encountered-nodes dest 0)
                (iterate)
            )
        )
    )
    (iterate)
    'SUCCESS
)

(define g (make-empty-graph #f))
(graph-add-node g 'a)
(graph-add-node g 'b)
(graph-add-node g 'c)
(graph-add-node g 'd)
(set-edge-value! (graph-add-edge g 'a 'b) 5)
(set-edge-value! (graph-add-edge g 'b 'c) 10)
(set-edge-value! (graph-add-edge g 'a 'c) 50)
```

```
(set-edge-value! (graph-add-edge g 'c 'd) 6)
```

Problem 1.7

Working with graphs under Scheme proved to be rather tedious, partly because I did not abstract all the functionality at first (and rather used the lower-level API) and partly because Scheme is overly verbose and has limited support for many elementary data-structures.

A language for graphs is a set of objects and procedures above Scheme that allows for manipulating graphs. I believe such a language could be useful for some purposes in Scheme. On the other hand, the Scheme specification lacks many primitives that are either available or easy to implement in other languages. Adding a graph manipulation would be a step in the right direction, but there should also be primitives for many other important things.