Graph-Project Report

Abstract Data Structures

Course Title: CSC33500

Instructor: Douglas Troeger

Date: 05/20/23

Authors

Myesha Mahazabeen EMPLID: 24005884

Najia Jahan EMPLID: 23932033

Project 1.1 & 1.2: We have built an abstract data type for undirected graphs. Graph G(V, E) is represented as a list of vertices and edges.

Project 2.1: We have built another abstract data structure for representing the graph as an adjacency list. Abstract algorithms designed for graph 1.1&1.2 also work correctly for adjacency list representation.

Project 2.2: We have built another abstract data structure for representing the graph as an adjacency matrix. Abstract algorithms designed for graph 1.1&1.2 also work correctly for adjacency matrix representation.

Set.scm

make-empty-set:

Precondition: None.

Postcondition: Returns an empty set.

set set-insert:

Precondition: x is an element and s is a set.

Postcondition: Returns a new set with x inserted into s, preserving the order of elements in s. If x is already present in s, the function returns s itself.

set-member?:

Precondition: x is an element and s is a set.

Postcondition: Returns #t if x is a member of s, and #f otherwise.

set-union:

Precondition: set1 and set2 are sets.

Postcondition: Returns a new set that is the union of set1 and set2, containing all the elements from both sets. The order of elements in the resulting set may vary.

Base Case: When one or both sets are empty, the function returns the non-empty set as the result.

IH: Assume that the set-union function correctly computes the union of sets set1 and set2 with sizes n and m, respectively, where n + m = k.

IS: To prove that the function still produces the correct result when given set1 and set2 with sizes n and m, respectively, where n + m = k + 1, we consider two cases:

1. (set-member? (car set1) set2) is true: In this case, (car set1) is already present in set2, so it does not need to be included in the union. By IH, the set-union returns the union of cdr of set1 and set2.

2. (set-member? (car set1) set2) is false: In this case, (car set1') is not present in set2, so it needs to be included in the union. By IH, the set-union function correctly computes the union of cdr of set1 and (set-insert (car set1) set2).

```
13 (define (set-union set1 set2)
14   (cond ((null? set1) set2)
15         ((set-member? (car set1) set2)
16          (set-union (cdr set1) set2))
17         (else (set-union (cdr set1) (set-insert (car set1) set2)))))
18
```

set-intersection:

Precondition: set1 and set2 are sets.

Postcondition: Returns a new set that is the intersection of set1 and set2, containing only the elements that are present in both sets. The order of elements in the resulting set may vary.

Base Case: When set1 is empty, the function correctly returns an empty set. When set2 is empty, the function iterates through set1 and returns an empty set.

IH: Assume that for any two sets, set1 and set2, where both sets have fewer elements than the original sets set1 and set2, the set-intersection function correctly computes the intersection of set1 and set2.

IS: Assuming x is an element in set1 and set3 is the intersection of set1 and set2:

1. If x is in set2, the function correctly returns x inserted into set3.

2. If x is not in set2, the function correctly returns set3.

By the inductive hypothesis, the function works correctly for smaller inputs.

```
19 (define (set-intersection set1 set2)
20   (cond ((or (null? set1) (null? set2))'())
21         ((set-member? (car set1) set2)
22          (set-insert (car set1)
23                      (set-intersection (cdr set1) set2)))
24         (else (set-intersection (cdr set1) set2))))
```

set-delete:

Precondition: x is an element and s is a set.

Postcondition: Returns a new set with all occurrences of x removed from s. If x is not present in s, the function returns s itself.

Base Case: When the set s is empty, the function correctly returns an empty set.

IH: Assume that for any set s that has fewer elements than the original set s, the set-delete function correctly deletes the element x from.

IS: Assuming the function works correctly for smaller inputs, we prove its correctness for larger inputs. Let x be an arbitrary element and s be a set. By the definition of set-delete, the function correctly handles three cases:

1. If s is empty, it returns an empty set.

2. If the first element of s is equal to x, it returns the remaining portion of s without x.

3. If the first element of s is not equal to x, it recursively calls set-delete with x and the remaining portion of s, creating s. It constructs a new set by consing the first element of s with s, ensuring x is not included.

By assuming the correctness of the function for smaller inputs (inductive hypothesis), we conclude that the function correctly deletes x from the set s.

Queue.scm

push-q

Precondition: q is a list.

Postcondition: returns a new list obtained by adding x to the front of q, preserving the order of elements in q.

pop-front

Precondition: q is a non-empty list.

Postcondition: returns the result of (cdr q), which is q without its first element.

peek-front

Precondition: non-empty queue q.

Postcondition: returns the first element of q without modifying q.

make-empty

Pre-conditions:function is defined and available.

Post-conditions: new queue q is created and initialized as an empty queue.

stackADT.scm

We used the same stackADT file in our project that you provided.

graph.scm

==========for graph (V, E) ADT==========

make-graph

Precondition: Takes a set of vertices (V) and a set of edges (E) as arguments. Assumes V and E are valid sets.

Postcondition: Returns a graph (list) consisting of the set of vertices (V) and the set of edges (E).

get-V

Precondition: Takes a graph with a set of vertices and a set of edges.

Postcondition: Returns the set of vertices.

get-E

Precondition: Takes a graph with a set of vertices and a set of edges.

Postcondition: Returns the set of edges.

add-V

Adds a vertex (v) to the set (set).

Precondition: Takes a vertex (v) and a set (set) as arguments. Assumes the set is a valid set of vertices.

Postcondition: Returns the updated set with the vertex (v) added.

add-E

Adds an edge (e) to the set (set).

Precondition: Takes an edge (e) and a set (set) as arguments. Assumes the set is a valid set of edges.

Postcondition: Returns the updated set with the edge (e) added.

==========for adjacency list graph ADT==========

make-node

Precondition: takes a value as an argument

Postcondition: returns a node (list) with the value cons-ed with an empty list of edges

get-val

Precondition: takes a node as an argument

Postcondition: returns the value of the node

get-edges-l

Precondition: takes a list of nodes [ex for adj-list: (a (c b))]

Postcondition: returns a list of the car of the cdr of the list [ex: (c b)]

make-edge

Precondition: takes two nodes as arguments

Postcondition: returns an edge (list) connecting the first node to the second node.

get-val

Precondition: takes a node as an argument

Postcondition: returns the value of the node

get-edges-l

Precondition: takes a list of nodes [ex for adj-list: (a (c b))]

Postcondition: returns a list of the car of the cdr of the list [ex: (c b)]

insert-node

Precondition: takes a graph and a value as arguments

Postcondition: returns a new graph with the new node added to the beginning

first-node

Precondition: takes a graph

Postcondition: returns the first node with its corresponding edges from the list

rest-graph

Precondition: Takes a graph

Postcondition: Returns the sublist without the first node and its edges.

create-graph-l

Precondition: Takes a list of nodes as input. Assumes nodes is a valid list.

Postcondition: Returns a graph (list) where each node is represented by a list cons-ing the node with an empty list of edges.

Base Case: When the nodes list is empty, the function returns an empty list, which represents an empty graph.

IH: Assume that for a list of size n, the create-graph-l function correctly constructs a graph.

IS: Calling create-graph-l on the remaining nodes (cdr nodes) constructs an empty graph. Therefore, the only remaining step is to cons the pair (list (car nodes) '()) to the constructed empty graph, which ensures that the resulting graph includes the node (car nodes) with an empty adjacency list.

```
109 (define (create-graph-l nodes)
110   (cond ((null? nodes) '())
111         (else (cons (list (car nodes) '()) (create-graph-l (cdr nodes))))))
112
```

add-to-list

Precondition: The adj-list parameter represents an existing adjacency list, and x is the element to be added.

Postcondition: Returns an updated adjacency list with the element x added to it. If x is already present. In the original adjacency list, the function returns the original list without any modifications.

GI: length(new-adj-list) = length(adj-list) or length(new-adj-list) = length(adj-list) + 1

```
113 (define (add-to-list adj-list x)
114   (cond ((member? x adj-list) adj-list)
115         (else (cons x adj-list))))
116
```

add-edge-l

Precondition: Given a graph as adjacency list, and two nodes u and v.

Postcondition: Returns a new adjacency-list graph that includes the original graph with an added edge between nodes u and v.

make-graph-l

Precondition: g is a valid graph representation as a list of vertices and edges. And vertices in g are unique.

Postcondition: Returns a valid graph representation containing all the vertices and edges specified in g and has the same structure as g, but with the added edges.

Base Case: When graph g has an empty list for the edges, that means there are no edges to add to the graph, so it returns the graph as it is.
IH: Assume that for a given input g with a non-empty list of edges, the make-graph-inner function correctly creates a graph from the input.

The idea for the function is, It creates an empty graph g1 using the create-empty-graph-l function with the nodes from g. It then defines an inner recursive make-graph-inner function that takes a graph and a list of edges.
If the list of edges is empty, it returns the graph as it is. If there are still edges in the list, it calls add-edge-l with the current edge (car edges) and the graph to add the edge to the graph.
It then recursively calls make-graph-inner with the modified graph and the remaining edges (cdr edges).
IS: Finally, the make-graph-l function calls make-graph-inner with the empty graph g1 and the list of edges from g.
From the IH, which assumes that make-graph-inner correctly creates a graph from the input, we can conclude that the function correctly creates a graph from the given input for a non-empty list of edges.

```
72 (define (make-graph-l g)
73   (define g1 (create-graph-l (car g)))
74    (define (make-graph-inner graph edges)
75      (cond ((null? edges) graph)
76            (else (make-graph-inner (add-edge-l (car (car edges)) (car (cdr (car edges))) graph) (cdr edges)))))
77   (make-graph-inner g1 (car (cdr g))))
78
```

==========for adjacency matrix graphADT==========

change-row :

Precondition: row is a list of pairs. Each pair in row has a cell value and a boolean flag and the cell is a valid value.
Postcondition: returns a modified row with the boolean flag set to #t for pairs matching the specified cell and the unchanged row if no matches are found.

add-edge:

Precondition: mat is a non-empty matrix (list of lists) and edge is a list representing an edge.
Postcondition: returns a new matrix with the edge added, modifying the appropriate sublist based on vertex matches.

add-all-edges:

Precondition: input parameters should be a list of edges and a valid matrix.
Postcondition: returns a matrix with all the edges added from the input list, while preserving the original dimensions and content of the input matrix.

make-graph-m:

Pre-condition: input gr must be a valid graph representation.
Post-condition: returns a graph-matrix representation of gr, where each element represents an edge between vertices or absence of an edge with #f.

add-row lbl:

Precondition: input parameters lbl and g should be valid lists representing appropriate data structures for the functions to operate on.
Postcondition: creates an adjacency matrix representing a graph, adding rows and edges according to the input, resulting in a modified or new adjacency matrix.

Graph-computations.scm

==========for normal graph (V E)==========

get-neighbors

Precondition: The seq parameter is a valid sequence. And The graph parameter represents a valid graph structure.
Postcondition: The get-neighbors function returns the correct list of neighbors for the given node in the graph.
Base Case: When seq is empty, the function correctly returns the accumulator acc.

IH: Assume that for any graph smaller-graph and node smaller-node, the get-neighbors function correctly retrieves neighbors when called with smaller-graph and smaller-node.

IS: Assuming the get-neighbors function works correctly for smaller inputs, we want to show that it works correctly for larger inputs.

By the definition of get-neighbors, the function retrieves the edges from the graph and uses accumulate-left (assuming it works correctly) to accumulate neighbors. Assuming that the get-neighbors function correctly retrieves neighbors for smaller graphs (IH), we conclude that it correctly retrieves neighbors for the current graph.

```
16 (define (get-neighbors graph node)
17   (let ((edges (get-E graph)))
18     (accumulate-left (lambda (acc e)
19                        (let ((u (car e))
20                              (v (cadr e)))
21                          (cond ((equal? u node) (cons v acc))
22                                ((equal? v node) (cons u acc))
23                                (else acc))))
24                      '()
25                      edges)))
26
```

==========for adjacency list==========

Find-adj-l:

Precondition: The graph g is valid (represented as adjacency list), and the node value n exists in the graph g.

Postcondition: The function returns a list of adjacent nodes of the node n in the graph g.

Base Case: For the base case, when the graph g contains only one node, it checks if n is equal to the value of that one node. If equal, it returns the edges of that node (get-edges-l (first-node g)).

IH: Let's assume that the find-adj-l function works correctly for a graph g with n nodes.

IS: We want to prove that it also works correctly for a graph g with n + 1 nodes.

Design-Idea: The function performs the following steps:

- Checks if the given node n is equal to the value of the first node, it returns the edges of the first node. Otherwise, it makes a recursive call to find-adj-l with the remaining graph (rest-graph-l g) and the same node n.

By IH, we assume that the recursive call in the find-adj-l function works correctly for a graph g with n nodes. Therefore, we can conclude that the function find-adj-l will correctly find the adjacent nodes for the given node n for a graph g with n + 1 nodes.

```
142 (define (find-adj-l g n)
143   (cond ((equal? n (get-val (first-node g))) (get-edges-l (first-node g)))
144         (else (find-adj-l (rest-graph-l g) n))))
145
```

member?

Precondition: Given a list lst and an element x.

Postcondition: The function returns #t if x is a member of lst, and #f otherwise.

```
38 (define (member? x lst)
39   (cond ((null? lst) #f)
40         ((eq? x (car lst)) #t)
41         (else (member? x (cdr lst)))))
42
```

==========for adjacency matrix==========

filter:

Pre-condition: pred is a one-argument function and lst is a list.

Post-condition: returns a new list containing elements from lst that satisfy the predicate function pred.

extract-left-elements:

Precondition: lst is a list of pairs with car and cdr elements.

Postcondition: returns a list of the car elements from the pairs in lst where the cdr is #t, without modifying the original input.

find-adj-matrix:

Preconditions: the graph parameter must be a non-empty list representing an adjacency matrix, and the node parameter must be a valid node in the graph.

Postconditions: returns the adjacency nodes of the input node if found in the graph, or an empty list if the node is not present.