Graph-Project Report

Algorithms Docs

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.

Algorithms.scm:

DFS

Precondition: The graph parameter should be a valid representation of a graph. The start-node parameter should be a valid node in the graph. The identifier parameter should be one of the following values: 'adj-list, or 'adj-matrix, or default 'normal-graph, indicating the format of the graph representation.

Postcondition: The function returns a list of nodes visited during the depth-first search traversal starting from the start-node. The returned list is in the order of visiting or the order in which the nodes are encountered during the traversal.

Base Case: When the stack is empty/null, the dfs-helper function correctly returns the visited list in reverse order, indicating that all nodes reachable from the start-node have been visited.

IH: Assume that the dfs-helper function works correctly for a graph with n nodes.

IS: We want to prove that the modified dfs-helper function also works correctly for a graph with n + 1 nodes.

Design-Idea: The dfs function initializes an empty visited list and a stack containing the start-node. It calls the helper function dfs-helper with the initial visited list and stack.

The dfs-helper function performs DFS recursively:

- If the stack is empty, it returns the visited list in reverse order. Otherwise, it retrieves the top node from the stack and the remaining stack.

- If the current node is already in the visited list, it calls dfs-helper recursively with the visited list and the remaining stack.
- If the current node is not visited, it retrieves the adjacent nodes of the current node based on the provided identifier ('adj-list', 'adj-matrix', or default normal-graph) and appends them to the stack. It also inserts the current node into the visited list.

Finally, it calls dfs-helper recursively with the updated visited list and the updated stack. By the IH we assume that the recursive call in dfs-helper works correctly for a graph with n nodes. On IS, dfs-helper will correctly visit all reachable nodes and return the visited list for a graph with n + 1 nodes.

Termination: The termination occurs when the stack is empty. Then the function immediately returns the visited list in reverse order, indicating that all reachable nodes have been visited.

```
31 (define (dfs graph start-node identifier)
32   (define visited '())
33   (define stack (push-l start-node (make-empty-stack-l)))
34
35   (define (dfs-helper visited stack)
36     (cond ((null? stack)
37            (reverse visited))
38           (else
39            (let ((node (top-l stack))
40                  (rest-stack (pop-l stack)))
41              (if (member node visited)
42                  (dfs-helper visited rest-stack)
43                  (let ((adjacent-nodes (cond ((equal? identifier 'adj-list)
44                                               (find-adj-l graph node))
45                                              ((equal? identifier 'adj-matrix)
46                                               (find-adj-matrix graph node))
47                                              (else
48                                               (get-neighbors graph node)))))
49                    (dfs-helper (set-insert node visited) (append adjacent-nodes rest-stack))))))))
50
51   (dfs-helper visited stack))
```

BFS

Precondition: The graph parameter should be a valid representation of a graph. The start-node parameter should be a valid node in the graph. The identifier parameter should be one of the following values: 'adj-list, or 'adj-matrix, or default 'normal-graph, indicating the format of the graph representation.

Postcondition: The function returns a list of nodes visited during the breadth-first search traversal starting from the start-node. The returned list is in the order of visiting or the order in which the nodes are encountered during the traversal.

Base Case: When the queue is empty (null? queue), the bfs-helper function correctly returns the visited list in reverse order, indicating that all nodes reachable from the start-node have been visited.

IH: Assume that the bfs-helper function works correctly for a graph with n nodes.

IS: We want to prove that the modified bfs-helper function also works correctly for a graph with n + 1 nodes.

Design-Idea: The bfs function initializes an empty visited list and a queue containing the start-node. It calls the helper function bfs-helper with the initial visited list and queue.

The bfs-helper function performs BFS recursively:

- If the queue is empty, it returns the visited list in reverse order. Otherwise, it retrieves the front node from the queue and the remaining queue.
- If the current node is already in the visited list, it calls bfs-helper recursively with the visited list and the remaining queue.
- If the current node is not visited, it retrieves the adjacent nodes of the current node based on the provided identifier ('adj-list', 'adj-matrix', or default normal-graph) and appends them to the queue. It also inserts the current node into the visited list.

Finally, it calls bfs-helper recursively with the updated visited list and the updated queue. By the IH, we assume that the recursive call in bfs-helper works correctly for a graph with n nodes. On IS, bfs-helper will correctly visit all reachable nodes and return the visited list for a graph with n + 1 nodes.

Termination: The termination occurs when the queue is empty/null, function immediately returns the visited list in reverse order, indicating that all reachable nodes have been visited.

```
61 (define (bfs graph start-node identifier)
62   (define visited '())
63   (define queue (make-empty-q))
64 
65   (define (bfs-helper visited queue)
66     (cond ((null? queue)
67            (reverse visited))
68           (else
69            (let ((node (peek-front queue))
70                  (rest-queue (pop-front queue)))
71              (if (member node visited)
72                  (bfs-helper visited rest-queue)
73                  (let ((adjacent-nodes (cond ((equal? identifier 'adj-list)
74                                               (find-adj-l graph node))
75                                              ((equal? identifier 'adj-matrix)
76                                               (find-adj-matrix graph node))
77                                              (else
78                                               (get-neighbors graph node)))))
79                    (bfs-helper (set-insert node visited) (append rest-queue adjacent-nodes))))))))
80 
81   (bfs-helper visited (push-q start-node queue)))
```

Acyclic?:

Pre-conditions: It is assumed that the graph is a valid graph representation (a graph with a list of V and E, or a graph represented as adjacency list, or a graph represented as adjacency matrix.

Post-conditions: The acyclic? function returns #t if the graph is acyclic and #f if it has cycles.

Base Case: Considering a graph with n = 0 nodes, the graph is empty. In this case, the code correctly returns #t because an empty graph is acyclic.

IH: Assume that the acyclic? function correctly determines whether a graph with n nodes is acyclic.

IS: We want to prove that the acyclic? function also works correctly for a graph with n + 1 nodes.

Design Idea: The acyclic? function initiates a depth-first search (DFS) starting from the first node ((car nodes)). It uses the dfs-recur helper function to perform the recursive traversal.

The dfs-recur function checks if the current node has been visited before. If it has, it checks if the node is present in the ancestors list. If it is, it indicates a cycle in the graph, and the code correctly returns #f (false), indicating that the graph has cycles.

If the node has not been visited before, the function retrieves the neighbors of the current node based on the provided identifier ('adj-list', 'adj-matrix', or default 'get-neighbors') and recursively calls dfs-recur for each neighbor. It updates the visited and ancestors lists accordingly.

The dfs-recur function accumulates the results of the recursive calls using the accumulate-left function, which checks if any of the recursive calls returned #f (indicating a cycle).

If at least one recursive call returns #f, the result of accumulate-left is #f. Otherwise, the result is #t.

The acyclic? function then negates the result obtained from the initial node ((not visited)) to check if there are any cycles in the graph.

By IH, the acyclic? function correctly determines whether a graph with n nodes is acyclic. During the inductive step, we add one more node to the graph (total of n + 1 nodes) and perform the same DFS traversal. Since the property holds for n nodes, the code correctly handles the additional node as well.

Termination: The termination case is reached when all nodes in the graph have been visited and processed by the DFS traversal algorithm.

```
(define (acyclic? graph identifier)
  (define (dfs-recur visited ancestors node)
    (if (member node visited)
        (member node ancestors)
        (let ((neighbors (cond ((equal? identifier 'adj-list)
                                (find-adj-l graph node))
                               ((equal? identifier 'adj-matrix)
                                (find-adj-matrix graph node))
                               (else
                                (get-neighbors graph node)))))
          (accumulate-left (lambda (acc neighbor)
                             (dfs-recur (set-insert node visited) (set-insert node ancestors) neighbor))
                           #f
                           neighbors))))

  (let ((nodes (if (equal? identifier 'adj-list)
                   (get-nodes-l graph)
                   (get-V graph))))
    (cond ((null? nodes) #t)
          (else
           (let ((visited (dfs-recur '() '() (car nodes))))
             (not visited))))))
```