Algorithms: Assignment #5

Tanvi Magdum
magdum.t@northeastern.edu

February 13, 2023

Answers

1 Problem 1 Solution :

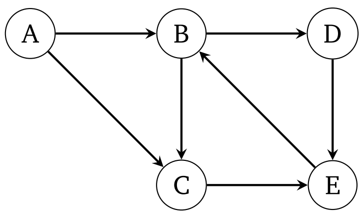
The reverse of a directed graph $G = (V, E)$ is another directed graph $G^R = (V, E^R)$ on the same vertex set, but with all edges reversed; that is, $E^R = \{(v, u) : (u, v) \epsilon E\}$.

Linear time algorithm to compute the adjacency list for G^R is as follows-

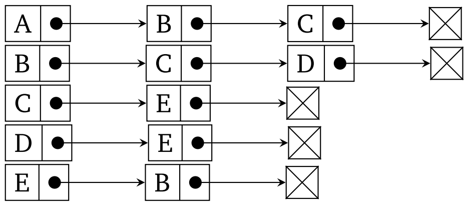
Algorithm-

1. Create an empty adjacency list for G^R and an empty list for each vertex in G.
2. Traverse list G and add each vertex in the list corresponding to vertex in G^R.
3. Reverse the direction of each edge in G^R by swapping the source and destination vertex of each edge.
4. Return adjacency list G^R as result of algorithm.

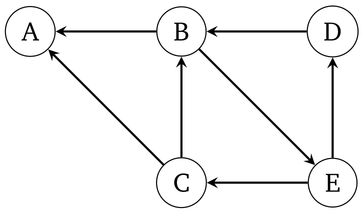
For example, let us consider the following directed graph-



Thus, adjacency list G would be like below. The first node of every list represent the vertices in the graph and other nodes represent the edges or connections. Suppose in first list A is the vertex and its connected edges B and C nodes are added in first list after A. The order does not matter after the first node, i.e. A->B->C and A->C->B both represent the same graph.

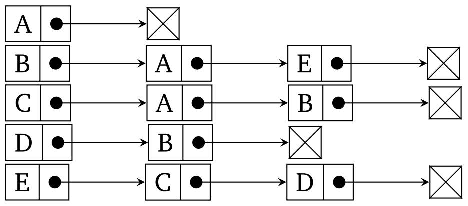


If we reverse this directed graph G to create graph G^R, it will look like the graph below. We need to transform the first graph into the below graph.



Now using the algorithm stated above, we need to create a new adjacency list for G^R. For each vertex in this list, there will be a list containing all its edges. If we look at graph G, A is directed towards B, so for first sub-list, the first node A is connected to B. In G^R, B is directed towards A, so the position of A should be in the second sub-list where first node is A, so that it becomes B->A. Similarly, in G A is directed towards C and hence it belongs to first sub-list. In G^R, the position of A will be in third sub-list where first node is C and C->A.

We can see that as we are traversing list G, we are defining position of every vertex of list G in the new adjacency list G^R depending upon the elements of the sub-lists in G. So, the direction is getting reversed automatically. We will get the below adjacency list G^R as result-



On a high-level view, we are simply traversing list G and adding elements in G^R which is a linear time operation. The time complexity of this operation is O(n), where n is the number of edges in graph G.

2 Problem 2 Solution :

Design and Algorithm for Median-Heap to maintain a collection of numbers S that supports Build(S), Insert(x), Extract(), and Peek() operations-

Median-heap can be implemented using two heap, a min-heap and a max-heap. The max-heap stores the elements that are less than or equal to median. The root of max-heap is the greatest element among all elements belonging to this heap. Similarly, the min-heap stores the elements that are greater than median. The root of this heap is the smallest element among all other elements in min-heap. The key idea behind Median-heap is to keep the heaps balanced, that means , the difference in between the sizes of both heaps should be greater than 1. To achieve this, we have to maintain the following invariants-

1. The size of min-heap is greater than 1 or equal to size of max-heap.
2. The root of min-heap is greater than equal to the root of max-heap.

We can implement Build(S), Insert(x), Extract(), and Peek() operations as below-

Build(S) :

1. Find the median of the input array using Find_Median(S).
2. Create two heaps, one max-heap and another min-heap.
3. Insert the elements from input array S into heap such that, elements that are less than or equal to median are inserted into max-heap, and elements that are greater than median are inserted into the min-heap.

Insert(x) :

1. Compare x with the median value.

2. If x is less than or equal to median, then insert it into the max-heap.
3. If x is greater than median, then insert it into the min-heap.
4. Check if the invariants are maintained, otherwise re-balance heaps, if necessary.

Peek() :

1. Return median of given data, that is, the root node of the min-heap or max-heap, depending upon which heap has more elements.

Extract() :

1. Compare the sizes of both heaps.
2. If sizes are equal, then extract the top element from the max-heap, which is the median. Remove this from the max-heap and return it.
3. If size of one heap is greater than other, then extract the top element from larger heap. Remove this element and return it.
4. Check if the invariants are maintained, otherwise re-balance heaps, if necessary.

Pseudocode-

```
max_heap: max-heap to store elements less than or equal to median
min_heap: min-heap to store elements greater than median
median: median value

Build(S):
    median = Find_Median(S)
    for x in S:
        if x <= median:
            max_heap.insert(x)
        else:
            min_heap.insert(x)
    rebalance_heaps()

Insert(x):
    if x <= median:
        max_heap.insert(x)
    else:
        min_heap.insert(x)
    rebalance_heaps()

Peek():
    return median

Extract():
    if max_heap.size == min_heap.size:
        return max_heap.extract_top()
    else if max_heap.size > min_heap.size:
        return max_heap.extract_top()
    else:
        return min_heap.extract_top()
    rebalance_heaps()

rebalance_heaps():
    if max_heap.size - min_heap.size >= 2:
        min_heap.insert(max_heap.extract_top())
    else if min_heap.size - max_heap.size >= 2:
        max_heap.insert(min_heap.extract_top())
    update_median()

update_median():
    if max_heap.size == min_heap.size:
        median = (max_heap.top + min_heap.top)/2
    else if max_heap.size > min_heap.size:
        median = max_heap.top
    else:
        median = min_heap.top
```

The time complexity of Build(S) is O(n), where n is the number of elements in input array S. We are assuming access to Find_Median(S) method which finds the median in linear time. Constructing two heaps from input array also takes O(n) time. Thus total complexity for Build(S) is O(n). The time complexity of Insert(x) and Extract() will be O(log n), as the heaps are implemented using binary trees and insertion and deletions take O(log n) time in a binary tree. The Peek() operation takes O(1) time as it simply returns the median value.

3 Problem 3 Solution :

3.1 Problem 3.1 Solution :

Breadth-First-Search(BFS) and Depth-First-Search(DFS) are two algorithms for traversing a graph, tree or any data structure represented with vertices and edges. The difference between them is the order in which they visit the vertices.

BFS visits all the vertices at a particular level before going to next level. It starts at the root node, then visits all neighbours those are at same distance from root, that is, level 1. Then it moves to neighbours of 1st level elements those are at same distance, level 2 and so on until all nodes are visited. BFS is used to find the shortest path between between two nodes in an unweighted graph, or to find connected components of a graph. The time complexity of BFS is O(v+e), where v is the number of vertices and e the number of edges in a graph.

DFS visits vertices as far from the root as possible before backtracking. It starts at root, visits a vertex, then moves to neighbour that has not been visited yet, then visits all its neighbours in similar fashion until it reaches the depth (a node which does not have any child node). Then it starts backtracking, visits a different neighbour and follows the same process until all the nodes are visited. DFS is usually used to find connected components, or to check if a cycle exists in the graph. The time complexity of DFS is O(v+e), where v is the number of vertices and e the number of edges in a graph.

In a nutshell, BFS visits nodes level by level, and DFS visits nodes as far as possible before backtracking. In the given tree structure, using BFS the nodes can be visited in either of the following orders-

1. {a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q}
2. {a, c, b, g, f, e, d, k, j, i, h, p, o, n, m, l, q}

Using DFS, the order would look like-

1. {a, b, d, e, c, f, h, l, m, q, i, g, j, n, o, k, p}
2. {a, c, g, k, p, j, o, n, f, i, h, m, q, l, b, e, d}

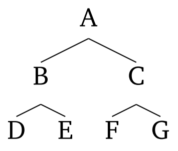
3.2 Problem 3.2 Solution :

One way to perform this is using Depth-First-Search (DFS) algorithm. We start from the root of the tree and then visit all the vertices in depth-first manner. The algorithm is as follows-

1. Start from the root of the tree and perform DFS, marking the vertices that we visit.
2. While visiting a vertex u, if it has not been visited before, visit its left child first, then right child.
3. If the vertex u has already been visited, then we have reached the back edge and we can backtrack to its parent.

This algorithm guarantees that every edge has been visited exactly twice, because when an algorithm visits a vertex and its two children, it crosses each edge twice, once while going down to the child and another while coming back to parent vertex.

As an example, let us consider the following tree structure.



We start from root which is A. Now since it is DFS, we need to travel to the farthest or deepest element possible from A. According to algorithm, we traverse the left child first. So first edge will be A—>B. Since B also has children, we traverse to its left child D. So, B—>D is created. Now D does not have any children. So we need to backtrack to its parent. Now we traverse to B, D's parent. So an edge D—>B is created. We see that B—D edge is traversed twice. First time while going down to D and second time while backtracking to B.

Since, B's left part is done, we traverse to its right child E, B—>E. E also does not have any children, backtrack to B with E—>B. Now we need to backtrack to B's parent, since we have visited all B's children, so an edge B—>A is created. Similarly for A's right part, the edges will be created as - C—>F, F—>C, C—>G, G—>C and lastly C—>A. We came back to the root from where we started, thus we have visited all the vertices in this tree in order {A, B, D, E, C, F, G}, and each edge was traversed exactly twice using DFS.

The running time of this algorithm is O(n), where n is the number of vertices in the tree. This is because, each vertex is visited once, and each edge is crossed twice, so the total number of operations is 2n, which is linear in the size of tree.

3.3 Problem 3.3 Solution :

The diameter of a tree is the length of the longest path between two nodes in a tree. The path can pass through any nodes, can contain multiple number of edges, but cannot repeat any node or any edge.

In the given tree, we see that there are total 6 levels assuming the root level as level 1. On second level, all children are at same distance from their parent at first level. Similarly, all children on third level are at same distance from their second level parents, and so on. For traversal, if we need to find distance between two nodes, then it is mandatory that we traverse by visiting their parents. In the given example, while finding distance between d and i, the path would be like- {d, b, a, c, f, i}. So here we have edges- d—>b, b—>a, a—>c, c—>f, f—>i, total 5 edges required. So the distance between d and i is 5. Note that we have not visited a node or traversed an edge more than once.

The height of this tree is 5 since it requires 5 edges to traverse from root a to the deepest node q. So, diameter has to be greater than 5. Now, if we need to calculate distance between the deepest element in the left side of root a which can be d or e and the deepest element in the right side of root a, which is only q, we have to start from suppose d, go upto to a and then traverse to q. We know that traversing from a to q requires 5 edges, but additionally now we will require 2 more edges, since d has to pass through a and d requires 2 edges to go to a. So, now the total distance between d and q becomes 7 and thus the diameter has to be 7 atleast. We need to check if diameter exceeds 7. Since, e lies at the same level as d, it will also take the same distance. Node b lies at an upper level of d, so it requires one edge less than that of d to traverse to q. If we go in the right side tree of root a at c, then we will require one edge less than the height to traverse to q. As, we go closer to q, the distance will decrease. So we skip the left side tree of node c. The deepest nodes in the right side tree of c are n, o and p. Now p to q also takes total 7 edges, 3 to traverse up to c and 4 down to q. p's parents would require less edges to travel than p. In short, we can see that the distance cannot exceed 7. So, the maximum distance, thus diameter of tree is 7.

3.4 Problem 3.4 Solution :

We can find the diameter of a tree by using DFS.

Algorithm-

1. Choose any node in the tree as a starting point.
2. Perform a DFS from starting node. Keep a track of maximum distance from starting node and any other node in the tree.
3. Once the node with maximum distance is found, store it as x.
4. Perform another DFS from node x. Keep track of maximum distance from x to any other node in the tree.
5. The maximum distance found in previous step is the diameter of the tree.

This algorithm takes into account all possible paths in tree and by performing 2 separate DFSs, it ensures that longest path is found. The running time of this algorithm is O(n), because each node in tree is visited twice, while first DFS and second DFS and each node takes O(1) to visit. So, total running time is proportional to 2n that is, O(n).