


National University of Computer and Emerging Sciences, Lahore Campus

	Course:	Data Structure	Course Code:	
	Program:	BSCS	Semester:	4th
	Name:		Section:	4A, 4B
	Registration #:		Assessment	Assignment
	Due Date:	24th May, 2020		

Instruction/

Notes:

1. Late submissions will not be entertained.

2.

Q1: *Shortest path in a directed graph by Dijkstra's algorithm*

Given a directed graph and a source vertex in the graph, the task is to find the shortest distance and path from source to target vertex in the given graph where edges are weighted (non-negative) and directed from parent vertex to source vertices.

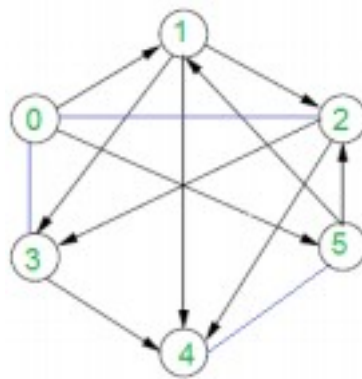
Approach:

1. Mark all vertices unvisited. Create a set of all unvisited vertices.
2. Assign zero distance value to source vertex and infinity distance value to all other vertices.
3. Set the source vertex as current vertex
4. For current vertex, consider all of its unvisited children and calculate their tentative distances through the current. (distance of current + weight of the corresponding edge) Compare the newly calculated distance to the current assigned value (can be infinity for some vertices) and assign the smaller one.
5. After considering all the unvisited children of the current vertex, mark the current as visited and remove it from the unvisited set.
6. Similarly, continue for all the vertex until all the nodes are visited.

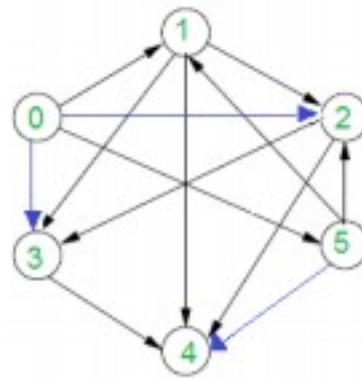
Q2: Find if there is a path between two vertices in an undirected graph. Given an undirected graph with N vertices and E edges and two vertices (U , V) from the graph, the task is to detect if a path exists between these two vertices. Print "Yes" if a path exists and "No" otherwise.

Q3: Shortest Path in Directed Acyclic Graph: Given a Weighted Directed Acyclic Graph and a source vertex in the graph, find the shortest paths from given source to all other vertices.

Q4: Assign directions to edges so that the directed graph remains acyclic. Given a graph with both directed and undirected edges. It is given that the directed edges don't form cycle. How to assign directions to undirected edges so that the graph (with all directed edges) remains acyclic even after the assignment? For example, in the below graph, blue edges don't have directions [Hint: Use topological Sorting]



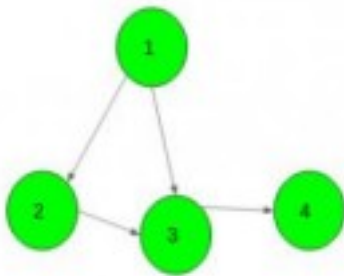
Given Graph



Graph after adding directions to undirected edges such that the graph remains acyclic.

Q5: Check if a directed graph is connected or not. Given a directed graph. The task is to check if the given graph is connected or not.

Examples:



Output : YES

Approach:

1. Take two bool arrays vis1 and vis2 of size N (number of nodes of a graph) and keep false in all indexes.
2. Start at a random vertex v of the graph G, and run a DFS(G, v).
3. Make all visited vertices v as vis1[v] = true.
4. Now reverse the direction of all the edges.
5. Start DFS at the vertex which was chosen at step 2.
6. Make all visited vertices v as vis2[v] = true.
7. If any vertex v has vis1[v] = false and vis2[v] = false then the graph is not connected.

Q6: Eulerian Path in undirected graph: An Euler path, in a graph or multigraph, is a walk through

the graph which uses every edge exactly once. An Euler circuit is an Euler path which starts and stops at the same vertex. Our goal is to find a quick way to check whether a graph (or multigraph) has an Euler path or circuit. A graph has an Euler circuit if and only if the degree of every vertex is even. A graph has an Euler path if and only if there are at most two vertices with odd degree. Given an adjacency matrix representation of an undirected graph. Find if there is any Eulerian Path in the graph. If there is no path print “No Solution”. If there is any path print the path.

Hashing

Q1: Given an array with n distinct elements, convert the given array to a form where all elements are in range from 0 to n-1. The order of elements is same, i.e., 0 is placed in place of smallest element, 1 is placed for second smallest element, ... n-1 is placed for largest element.

Input: arr[] = {10, 40, 20}

Output: arr[] = {0, 2, 1}

Input: arr[] = {5, 10, 40, 30, 20}

Output: arr[] = {0, 1, 4, 3, 2}

Method 1 (Simple)

A Simple Solution is to first find minimum element replace it with 0, consider remaining array and find minimum in the remaining array and replace it with 1 and so on. Time complexity of this solution is $O(n^2)$

Method 2 (Efficient)

The idea is to use hashing and sorting. Below are steps.

- 1) Create a temp array and copy contents of given array to temp[]. This takes $O(n)$ time. 2) Sort temp[] in ascending order. This takes $O(n \log n)$ time.
- 3) Create an empty hash table. This takes $O(1)$ time.
- 4) Traverse temp[] from left to right and store mapping of numbers and their values (in converted array) in hash table. This takes $O(n)$ time on average.
- 5) Traverse given array and change elements to their positions using hash table. This takes $O(n)$ time on average.

Q2: [Coalesced hashing](#) is a collision avoidance technique when there is a fixed sized data. It is a combination of both Separate chaining and Open addressing. It uses the concept of Open Addressing(linear probing) to find first empty place for colliding element from the bottom of the hash table and the concept of Separate Chaining to link the colliding elements to each other through pointers. The hash function used is **$h=(key) \% (\text{total number of keys})$** . Inside the hash table, each node has three fields:

- $h(key)$: The value of hash function for a key.

- Data: The key itself.
- Next: The link to the next colliding elements.

The basic operations of Coalesced hashing are:

1. **INSERT(key):** The insert Operation inserts the key according to the hash value of that key if that hash value in the table is empty otherwise the key is inserted in first empty place from the bottom of the hash table and the address of this empty place is mapped in NEXT field of the previous pointing node of the chain.(Explained in example below).
2. **DELETE(Key):** The key if present is deleted.Also if the node to be deleted contains the address of another node in hash table then this address is mapped in the NEXT field of the node pointing to the node which is to be deleted
3. **SEARCH(key):** Returns **True** if key is present, otherwise return **False**.

The best case complexity of all these operations is $O(1)$ and the worst case complexity is $O(n)$ where n is the total number of keys.It is better than separate chaining because it inserts the colliding element in the memory of hash table only instead of creating a new linked list as in separate chaining.

Q3: Rearrange characters in a string such that no two adjacent are same using hashing. Given a string `str` with repeated characters, the task is to rearrange the characters in a string such that no two adjacent characters are same. If it is possible then print Yes else print No.

Approach: The idea is to store the frequency of each character in an unordered map and compare maximum frequency of character with the difference of string length and maximum frequency number. If the maximum frequency is less than the difference then it can be arranged otherwise not.

These are some key points in hashing:

1. The purpose of hashing is to achieve search, insert and delete complexity to $O(1)$. • Hash function is designed to distribute keys uniformly over the hash table.
2. Load factor α in hash table can be defined as number of slots in hash table to number of keys to be inserted.
3. For open addressing, load factor α is always less than one.
4. The complexity of insertion, deletion and searching using open addressing is $1/(1-\alpha)$. • The complexity of insertion, deletion and searching using chaining method is $(1+\alpha)$.