Aseel Ahmad - 324885417

מערכות תכנות 2 מטלה 1

hasPathDFS:

This function implements a Depth-First Search (DFS) algorithm to check if there is a path from a given source node to a destination node in a graph.

The function hasPathDFS takes four parameters: the graph in which we're searching for a path, the src (source) node, the dest (destination) node, and a set of visited nodes to keep track of visited nodes during traversal.

It starts with a base case: if the source and destination nodes are the same, it returns true, indicating that a path exists.

The current node (src) is marked as visited by inserting it into the visited set.

It then iterates through all the adjacent nodes of the current node using a for loop.

For each adjacent node i, it checks if there is an edge between the current node src and node i, and if i has not been visited yet.

If both conditions are met, it recursively calls hasPathDFS with node i as the new source node.

If the recursive call returns true, it means a path is found from node i to the destination dest, so the function returns true.

If no path is found after iterating through all adjacent nodes, the function returns false.

hasPath:

It initializes a set to keep track of visited nodes and then calls hasPathDFS to find if there's a path between the given source and destination nodes in the graph.

isConnected:

This function checks if every pair of vertices in the graph is connected, meaning there's a path between them. It iterates through all pairs of vertices and uses hasPath to check connectivity. If there's any pair of vertices for which there's no path between them, it returns false, indicating that the graph is not connected. Otherwise, it returns true.

isContainsCycle:

This function iterates over all nodes in the graph and initiates a DFS from each unvisited node. If a cycle is found during any of these DFS traversals, it returns true, indicating that the graph contains a cycle. If no cycle is found after searching through all nodes, it returns false.

dfs:

This function performs a depth-first search starting from a given node in the graph. It recursively explores all neighbors of the current node. If it encounters an unvisited neighbor, it marks it as visited and continues the search. If it encounters a visited neighbor that is not the parent of the current node (indicating a back edge), it identifies a cycle and stores the cycle path. It then returns true to indicate that a cycle was found.

shortestPath:

This function implements Dijkstra's algorithm to find the shortest path between two vertices

in a graph.
This function takes a graph, a source vertex, and a destination vertex as input and returns the shortest path from the source to the destination vertex. It first initializes some data structures to keep track of distances, parents, and visited vertices. Then it uses a priority queue to select the vertex with the smallest distance from the source vertex at each step.
Dijkstra's Algorithm: The core of the function is a while loop that continues until the priority queue is empty. In each iteration, it extracts the vertex with the smallest distance from the priority queue. If this vertex is the destination vertex, it constructs the shortest path by tracing back through the parent vertices and returns it. Otherwise, it updates the distances and parents of adjacent vertices if a shorter path is found. This process continues until either the destination vertex is reached or all reachable vertices are visited.

isBipartite:
This function takes a graph as input and returns a string indicating whether the graph is bipartite or not, along with the two sets of vertices if it is bipartite. It uses a breadth-first search (BFS) approach to traverse the graph and assign colors to vertices such that no two adjacent vertices have the same color.
BFS Algorithm: The function starts by initializing an empty queue and an array to store colors for each vertex. It then iterates through all vertices of the graph. For each unvisited vertex, it assigns a color and starts BFS from that vertex. During BFS, it assigns the opposite color to each neighbor of the current vertex. If it encounters a neighbor with the same color as the current vertex, it returns "0", indicating that the graph is not bipartite. If BFS completes without finding such a case, it divides the vertices into two sets based on their colors and returns a string indicating that the graph is bipartite along with the two sets.

negativeCycle:
This function takes a graph as input and returns true if the graph contains a negative cycle, otherwise false. It first initializes an array to store distances to all vertices, setting the distance to the source vertex as 0 and distances to all other vertices as infinite.
Bellman-Ford Algorithm: The function then iteratively relaxes all edges in the graph for |V| - 1 times, where |V| is the number of vertices. During each iteration, it updates the distance to each vertex if a shorter path is found through the current edge. This process guarantees finding the shortest path from the source vertex to all other vertices if the graph doesn't contain a negative cycle.
After |V| - 1 iterations, the function checks for negative cycles by iterating over all edges again. If there's any edge that can further decrease the distance to a vertex, it indicates the presence of a negative cycle.

Graph::Graph()
This is the constructor of the Graph class. The constructor initializes numOfEdges and numOfVertices to 0, and the adjacencyMatrix is an empty matrix.

void Graph::loadGraph(const std::vector<std::vector<int>> &matrix)
This function is responsible for loading a graph from a given adjacency matrix.It takes a vector of vectors of integers (matrix) representing the adjacency matrix of the graph.It first checks if the matrix is square by ensuring that each row has the same number of elements as the number of rows.Then, it updates the number of vertices (numOfVertices) to the size of the matrix (number of rows), assigns the given matrix to the adjacencyMatrix, and counts the number of edges by iterating over the adjacency matrix and counting non-zero entries.Finally, it updates the number of edges (numOfEdges) with the calculated count.

size_t Graph::getNumVertices()
This function returns the number of vertices in the graph.

int Graph::getNumEdges()
This function returns the number of edges in the graph.

std::vector<std::vector<int>> Graph::getGraph()
This function returns the adjacency matrix of the graph.It returns the adjacencyMatrix.

void Graph::printGraph()
This function prints information about the graph, including the number of vertices and edges.

bool Graph::hasEdge(int from, int to) const
This function checks if there is an edge between two vertices , It takes two vertex indices (from and to) as arguments . It first checks if the vertex indices are valid , If the indices are valid, it checks if there is a non-zero entry in the corresponding position of the adjacency matrix, indicating the presence of an edge between the vertices.it returns true if there is an edge between the vertices,otherwise false.

```
aseela@aseela:~/Downloads/CPP_EX1_24-main$ make demo
clang++ -std=c++14 -Werror -Wsign-conversion --compile Demo.cpp -o Demo.o
clang++ -std=c++14 -Werror -Wsign-conversion --compile Graph.cpp -o Graph.o
clang++ -std=c++14 -Werror -Wsign-conversion --compile Algorithms.cpp -o Algorithms.o
clang++ -std=c++14 -Werror -Wsign-conversion Demo.o Graph.o Algorithms.o -o demo
aseela@aseela:~/Downloads/CPP_EX1_24-main$ ./demo
Graph with 3 vertices and 4 edges
1
0 -> 1 -> 2
0
The graph is bipartite: A={0, 2}, B={1}
Graph with 5 vertices and 8 edges
0
-1
1
0
Graph with 5 vertices and 10 edges
1
0 -> 2 -> 3 -> 4
1
0
The graph is bipartite: A={0, 2, 4}, B={1, 3}
Input graph is not a square matrix.
aseela@aseela:~/Downloads/CPP_EX1_24-main$ make test
clang++ -std=c++14 -Werror -Wsign-conversion --compile TestCounter.cpp -o TestCounter.o
clang++ -std=c++14 -Werror -Wsign-conversion --compile Test.cpp -o Test.o
clang++ -std=c++14 -Werror -Wsign-conversion TestCounter.o Test.o Graph.o Algorithms.o -o
aseela@aseela:~/Downloads/CPP_EX1_24-main$ ./test
[doctest] doctest version is "2.4.11"
[doctest] run with "--help" for options
aseela@aseela:~/Downloads/CPP_EX1_24-main$ make clean
rm -f *.o demo test
aseela@aseela:~/Downloads/CPP_EX1_24-main$
```