COIS 3020H Assignment 1: Graphs README Documentation

<u>Description:</u>
This project explores the concepts of fundamental graph-based data structures implemented as an adjacency list using the C#programming language. The utilities of this project will model relationships between various entities through simple methods and complex algorithms.

<u>Contributors:</u>
Adam Kassana
Alexander Wolf
Carmen Tullio

Content Table

Adjacency List

The adjacenylist.cs is a file that defines a class for representing a weighted graph using adjacency list data structure. It provides numerous basic functions such as adding and removing vertices, edges, and traversal functions.

Class data fields:

- private readonly Dictionary<int, List<int>> adjacencyList

  A dictionary variable that stores the vertices as keys and their adjacent vertices as number integer values. This data structure represents the graph's adjacency list. The dictionary is the preferred data structure as it enables for dynamic graphs and ease of implementation.

- private List<int> recordVer

  A list that records the vertices of the graph separately. Has property accessor.

- private List<int[]> recordEdge

  A list that records the edges of the graph separately. Each element in the list in an array of two integers represents an edge used to maintain information about the graph's edges. Has property accessor.

- private List<int> recordWei

A list that records the weights of the graph separately. Each element in the list corresponds to the weight of the edge at the same index in the 'recordEdge' list. It serves to store edge weights should you choose to work with a weighted graph instead of unweighted. Has property accessor.

Constructor:

AdjacencyList();
Instantiate an adjacency list to represent a graph. Instantiates memberfield 'Dictionary adjancyList' to new instance.
e.g.

```
AdjacencyList list = new AdjacencyList();
```

Methods:

AddVertex

Method Header: public void AddVertex(int vertex)

Parameter: int vertex - integer variable representing the vertex

Time Complexity: O(1)

Adding a vertex to the adjacency list typically involves inserting a new key-value pair in the dictionary. This operation is constant time because it doesn't depend on the number of vertices or edges in the graph.

Description: This method is used to add a new vertex to the graph represented by the adjacency list. It ensures that the vertex is not duplicated, and if it doesn't already exist, it adds the vertex to the adjacency list.

AddEdge

Method Header: public void AddEdge(int source, int destination, bool bidirect = false, int weight = 1)

Parameters:

int source - The source vertex of the edge.

int destination - The destination vertex of the edge.

bool bidirect (optional, default: false) - A flag indicating whether the edge should be bidirectional.

int weight (optional, default: 1) - The weight of the edge.

Time Complexity: O(1)

Adding an edge to the adjacency list is also a constant time operation. It involves appending the destination vertex to the list of adjacent vertices for the source vertex in the dictionary.

Description:

This method is used to add an edge between two vertices in the graph represented by the adjacency list. It allows you to specify whether the edge is bidirectional and its weight. The method also includes error handling to ensure the weight is valid.

RemoveVertex

Method Header: public void RemoveVertex(int vertex)

Parameters: int vertex - The vertex you want to remove from the graph

Time Complexity: O(n^2)

The time complexity of this method depends on the number of neighbors of the vertex being removed where n is the number of vertices in the graph.

Description:

This method is used to remove a vertex from the graph represented by the adjacency list. It also removes any associated edges that connect to the removed vertex. The method ensures that the specified vertex exists in the graph before attempting to remove it.

RemoveEdge

Method Header: public void RemoveEdge(int source, int destination, bool bidirect)

Parameters:

- int source - The source vertex of the edge to be removed.
- int destination - The destination vertex of the edge to be removed.
- bool bidirect - A flag indicating whether the edge to be removed is bidirectional.

Time Complexity: O(E)

The time complexity of this operation is directly proportional to the number of edges in the graph

Description:

This method is used to remove an edge between two vertices in the graph represented by the adjacency list. It specifies whether the edge to be removed is bidirectional. The method ensures that the specified vertices and the edge exist in the graph before attempting to remove them.

Get Neighbors

Method Header: public List<int> GetNeighbors(int vertex)

Parameters: int vertex - The vertex for which to retrieve the neighbors

Time Complexity: O(1)

The time complexity of this method is O(1) which is a constant-time operation because the adjacency list is a data structure that allows direct access to a vertex's neighbors.

Description: This method is used to obtain a list of neighbors for a specified vertex in the graph represented by the adjacency

DepthFirstSearch

Method Header: public void DepthFirstSearch(int start)

Parameters: int start - the starting vertex to commence the search from.

Time Complexity: O(V+E)

Where V is the number of vertices in the graph, and E is the number of edges. Runtime is proportional to visiting all vertices and potentially checking all edges in the graph.

Description:

Invokes the depth first search traversal method from traversal.cs

BreadthFirstSearch

Method Header: public string BreadthFirstSearch(int start)

Parameters: int start - the starting vertex to commence the search from.

Time Complexity: O(V+E)

Where V is the number of vertices in the graph, and E is the number of edges. Initializing a visited array and a queue to track visited vertices will potentially result in visiting all vertices O(V) and checking the neighbors of each vertex O(E).

Description:

Invokes the breadth first search traversal method from traversal.cs

CycleDetect

Method Header: public void CycleDetect(int vertex)

Parameters: int vertex - The vertex for which cycle detection is initiated.

Description: The CycleDetect method is used to detect the presence of a cycle in the graph starting from a specified vertex (vertex). In the context of graph theory, a cycle is a path in the graph that starts and ends at the same vertex, and the detection of cycles is an important operation in graph analysis.

This method achieves cycle detection by utilizing the VerConnect method with the same vertex as both the source and destination. The VerConnect method checks whether there is a path that connects the specified vertex to itself, which indicates the presence of a cycle. If a path exists, it means the graph contains a cycle starting from the given vertex.

Time Complexity: O(V+E)

The method first checks if two vertices are directly connected, and if not, it resorts to BFS to determine if there is an indirect connection.

ShortPath

Method Header: public string ShortPath(int source, int destination)

Parameters:

- int source - The index of the source vertex from which the shortest path is determined.
- int destination - The index of the destination vertex to which the shortest path is calculated.

Description: To find the shortest path between two vertices in a graph represented by an adjacency list by finding the most efficient route between two locations in a network or navigating a graph with weighted edges. It utilizes 'VerConnect' to determine if a path exists. It then calls 'Traversal.ShortPath' to determine the shortest path

Time Complexity: O(|E| + |V|log|V|)

VerConnect

Method Header: public bool VerConnect(int source, int destination, bool print = true)

Parameters:

- int source - The index of the source vertex.
- int destination - The index of the destination vertex.
- bool print (optional, default: true) - A flag indicating whether to print information about the connectivity.

Description: The VerConnect method is used to determine the connectivity between two vertices in a graph represented by an adjacency list. It checks if there is a direct connection between the source and destination vertices. If a direct connection exists, it can optionally print information about the connectivity. If there is no direct connection, it employs the Traversal.VerIndirConnect method to explore indirect connectivity through a breadth-first search (BFS).

Time Complexity: O(V + E)

Where V is the number of vertices, and E is the number of edges in the graph.

Kruskal

Method Header: public int Kruskal()

Parameters: None

Description: Invokes the Kruskal method in ‘Traversal.cs’

Time Complexity: O(E log E)

Where E is the number of edges in the graph. This complexity is due to sorting the edges by weight.

Traversal

This class provides essential algorithms and methods for traversing and analyzing graphs represented as adjacency lists.

Class data fields:

- private static bool[] visited = { false };

  To track whether a vertex in the graph has been visited during traversal algorithms like depth-first search (DFS) and breadth-first search (BFS). It acts as a marker for vertices, allowing the traversal algorithms to keep track of the visited status of each vertex.

Methods:

BreadthFirstSearch

Method Header: public static string BreadthFirstSearch(AdjacencyList adj, int start)

Parameters:

- AdjacencyList adj - An instance of the adjacency list representing the graph.
- int start - The index of the starting vertex for the breadth-first search.

Description: The BreadthFirstSearch method traverses a graph represented by an adjacency list in a breadth-first order, starting from a specified vertex. It uses a queue to keep track of visited vertices and their neighbors, ensuring that the traversal covers all connected components in the graph.

Time Complexity: O(V + E)

Where V is the number of vertices, and E is the number of edges in the graph.

DepthFirstSearch

Method Header: public static void DepthFirstSearch(AdjacencyList adj, int start)

Parameters:

- AdjacencyList adj - An instance of the adjacency list representing the graph.
- int start - The index of the starting vertex for the depth-first search.

Description: The DepthFirstSearch method traverses a graph represented by an adjacency list in a depth-first order, starting from a specified vertex. It explores each branch as deeply as possible before backtracking. The traversal is performed recursively.

Time Complexity: O(V + E)

Where V is the number of vertices, and E is the number of edges in the graph.

DFSRecursive

Method Header: private static void DFSRecursive(AdjacencyList adj, int vertex)

Parameters:

- AdjacencyList adj - An instance of the adjacency list representing the graph.
- int vertex - The index of the vertex currently being visited during the depth-first search.

Description: The DFSRecursive method is a utility function used within the DepthFirstSearch method. It recursively traverses a graph represented by an adjacency list in a depth-first order, starting from a specified vertex. This function marks the current vertex as visited and explores its neighbors, recursively calling itself for unvisited neighbors.

Time Complexity: O(V + E)

where V is the number of vertices, and E is the number of edges in the graph.

VerIndirConnect

Method Header: public static bool VerIndirConnect(AdjacencyList adj, int start, int destination)

Parameters:

- AdjacencyList adj - An instance of the adjacency list representing the graph.
- int start - The index of the starting vertex.
- int destination - The index of the destination vertex.
- bool print (optional, default: true) - A flag indicating whether to print information about connectivity.

Description: The VerIndirConnect method checks for indirect connectivity between two vertices in a graph represented by an adjacency list. It first verifies if there's no direct connection between the two vertices. If not, it uses a breadth-first search (BFS) to determine if an indirect connection exists. The method can optionally print information about connectivity.

Time Complexity: O(V + E)

Where V is the number of vertices, and E is the number of edges in the graph.

ShortPath

Method Header: public static string ShortPath(AdjacencyList adj, int start, int finish)

Parameters:

- AdjacencyList adj - An instance of the adjacency list representing the graph.
- int start - The index of the starting vertex.
- int finish - The index of the destination vertex.

Description: The ShortPath method finds the shortest path between two vertices in a graph represented by an adjacency list. It uses a modified form of breadth-first search (BFS) to explore the graph, starting from the start vertex and ending at the finish vertex. The method tracks distances and paths during traversal.

Time Complexity: O(V + E)

Where V is the number of vertices, and E is the number of edges in the graph.

Kruskal

Method Header: public static int Kruskal(AdjacencyList adj, bool print = true)

Parameters:

- AdjacencyList adj - An instance of the adjacency list representing the graph.
- bool print (optional, default: true) - A flag indicating whether to print information about the minimum spanning tree (MST).

Description: The Kruskal method implements Kruskal's algorithm to find the minimum spanning tree (MST) in a graph represented by an adjacency list. It can optionally print information about the edges included in the MST.

Time Complexity: O(E log E), where V is the number of vertices and E is the number of edges in the graph.