A green and black logo

Description automatically generated

Assignment-2

COMP-359

Design and Analysis of Algorithms

Birkaran Singh

Elijah Duchak

Wenjun Chen

Contents

[1. Introduction 3](#_Toc179388569)

[2. BFS Algorithm and Graph Implementation 3](#_Toc179388570)

[2.1 Graph Data Structure 3](#_Toc179388571)

[2.2 BFS Algorithm Implementation 4](#_Toc179388572)

[3. Memory Usage 7](#_Toc179388573)

[3.1 Memory Usage of the Queue 7](#_Toc179388574)

[3.2 Memory Usage of the Visited Set 8](#_Toc179388575)

[3.3 Overall Space Complexity 8](#_Toc179388576)

[3.4 Additional Considerations 8](#_Toc179388577)

# 1. Introduction

Breadth-First Search (BFS) is a fundamental graph traversal algorithm that explores nodes in a graph level by level, starting from a given source node. It systematically visits all neighbors of a node before moving on to the next level of the graph, ensuring that nodes closer to the source are visited before those that are further away. BFS is commonly used in various applications such as finding the shortest path in unweighted graphs, determining connectivity, and solving maze-like problems.

In this implementation, BFS is applied using an **array-based queue** to manage nodes during the traversal process. The task is to develop a custom graph structure, add and connect nodes, and implement BFS. An array is used to implement the queue, as required by the assignment instructions. The memory requirements of this array, which grows as nodes are explored, will be analyzed as part of the documentation.

This report focuses on the space complexity of the algorithm, particularly the memory usage of the array used to implement the queue, and provides an overview of the custom graph data structure implemented using linked lists.

# 2. BFS Algorithm and Graph Implementation

## 2.1 Graph Data Structure

In this implementation, the graph is composed of nodes, where each node contains a value and a list of neighboring nodes. The graph is represented using an **adjacency list**, with each node holding references to its neighbors. The nodes are connected using a **linked list**, as specified in the assignment instructions, which offers efficient dynamic addition of neighbors. This is especially useful for graphs where the number of neighbors per node can vary.

Each node in the graph is represented by an instance of the Node<T> class. This class contains the following components:

* **T value**: The data stored in the node, representing the unique identifier or content of the node (for example, a letter, a number, or a string).
* **List<Node<T>> neighbors**: A linked list (LinkedList) that holds references to all neighboring nodes. This allows the node to maintain a dynamically sized list of connections to other nodes. The linked list is chosen because it allows efficient addition and removal of neighbors, and its memory overhead is manageable compared to an array for dynamic lists.
* **Node<T> connect(Node<T> node) method**: This method establishes a connection (or edge) between two nodes. It ensures that the connection is **bidirectional**—if Node A is connected to Node B, then Node B must also be connected back to Node A. This is achieved by adding each node to the other’s list of neighbors.

A screenshot of a computer code

Description automatically generatedHere’s a representation of the Node<T> class:

In this setup, the **linked list** is crucial for maintaining flexibility in managing node connections, particularly as the graph grows or changes dynamically. Each node can connect to multiple neighbors, allowing for complex graph structures.

## 2.2 BFS Algorithm Implementation

The Breadth-First Search (BFS) algorithm is used to explore all nodes of a graph level by level. BFS starts from a designated node (called the **starting node**) and explores its neighbors first, before moving on to their neighbors, and so on. The primary data structure used to facilitate this traversal is a **queue**, which operates on a **FIFO (First In, First Out)** principle. In this assignment, the queue is implemented using a **raw array** instead of a Java library, adhering to the assignment requirements.

**Queue Implementation**

The queue in this implementation is represented as a fixed-size array of Node<T>, where T is the type of data stored in the nodes (e.g., strings or integers). The array is initialized with a size equal to the total number of nodes, n, since in the worst case, all nodes might be queued during the BFS traversal.

Key components of the queue implementation:

* **Node<T>[] queue = new Node[n];**: This is the array that stores the nodes waiting to be processed. The size n corresponds to the total number of nodes in the graph.
* **head pointer**: This pointer tracks the front of the queue, where nodes are dequeued (removed for processing). It starts at index 0 and moves forward as nodes are processed.
* **tail pointer**: This pointer tracks the end of the queue, where new nodes are enqueued (added). It moves forward as new nodes are encountered during the traversal.

The BFS traversal follows these steps:

1. **Enqueue the starting node** by adding it to the queue at the tail position, and marking it as visited.
2. **Dequeue a node** from the head position of the queue, and check its neighbors.
3. For each unvisited neighbor, **enqueue** it at the tail position of the queue and mark it as visited.
4. Repeat the process until either the target node is found or the queue becomes empty (indicating that all nodes have been explored).

The implementation of BFS looks like this:

A screenshot of a computer program

Description automatically generated

**BFS Operation Example**

Let’s consider a small graph where we start BFS at node A, and the target node is O:

* First, A is enqueued.
* A’s neighbors (B, C, D, E) are visited and enqueued.
* BFS proceeds level by level, checking each node’s neighbors until the target node O is found.

# 3. Memory Usage

## 3.1 Memory Usage of the Queue

The queue is a critical component of the Breadth-First Search (BFS) algorithm. In this implementation, the queue is represented as a raw array of size **n**, where **n** is the total number of nodes in the graph. This queue stores the nodes that are waiting to be processed as the BFS algorithm traverses the graph level by level.

A screenshot of a computer program

Description automatically generated

**Memory Required for the Queue:**  
For a graph with **n** nodes, the queue is initially empty but can grow to hold a maximum of **n** nodes during traversal. The key observation about BFS is that it explores each level of the graph fully before moving to the next level. Therefore, in the worst-case scenario, the queue may need to hold all nodes from a single level.

* **Worst-case scenario:** The queue will require space for all nodes in the graph when BFS needs to explore a particularly wide level (i.e., when a node has many children).
* **Queue memory:** Each entry in the array represents a reference to a Node<T> object, leading to a memory requirement of O(n)O(n)O(n) for the queue. Thus, the total space complexity due to the queue is O(n)O(n)O(n).

## 3.2 Memory Usage of the Visited Set

The visited set is employed to track all nodes that have already been processed during the BFS traversal. This mechanism ensures that nodes are not visited multiple times, preventing infinite loops in graphs containing cycles.

**Visited Set Memory:**  
The visited set requires O(n)O(n)O(n) memory, where **n** is the total number of nodes in the graph. Each node must be marked as visited once to avoid reprocessing.

* The visited set is implemented using a HashSet, which provides constant-time lookups and inserts. This efficiency makes it suitable for checking whether a node has been visited.
* Since each node is added to the visited set only once, the total memory usage for the set also amounts to O(n)O(n)O(n).

## 3.3 Overall Space Complexity

The space complexity of the BFS algorithm is primarily determined by the following factors:

* The queue, which necessitates O(n)O(n)O(n) space in the worst case.
* The visited set, which also demands O(n)O(n)O(n) space.

Consequently, the total space complexity of the BFS algorithm can be expressed as:

Total Space Complexity=O(n)+O(n)=O(n)\text{Total Space Complexity} = O(n) + O(n) = O(n)Total Space Complexity=O(n)+O(n)=O(n)

This indicates that for a graph with **n** nodes, the BFS algorithm will utilize O(n)O(n)O(n) memory.

## 3.4 Additional Considerations

* **Node Storage:** Each node in the graph maintains a list of its neighbors. In sparse graphs, where nodes have few connections, the memory used for neighbor storage is minimal. Conversely, in dense graphs, where each node connects to many others, the memory required to store neighbors can be substantial. In such cases, the adjacency list may require O(n+m)O(n + m)O(n+m) space, where **m** represents the number of edges in the graph.
* **Queue Array Bounds:** The queue is implemented as a fixed-size array with a maximum capacity of **n**. This implementation simplifies the memory analysis, as the array cannot dynamically expand beyond **n** nodes. However, when dealing with very large graphs, it is crucial to ensure that the array size is sufficient to accommodate all nodes.

**Bibliography**