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](#_Toc180089432)

[1.1 Topic 3](#_Toc180089433)

[1.2 Introduction 3](#_Toc180089434)

[1.3 Plan & Work logging 4](#_Toc180089435)

[2. BFS Algorithm and Graph Implementation 4](#_Toc180089436)

[2.1 Graph Data Structure (n = 15) 4](#_Toc180089437)

[2.2 Breadth-First Search (BFS) Algorithm Implementation 7](#_Toc180089438)

[Queue Implementation Using Raw Array 7](#_Toc180089439)

[Edge Case Considerations: 8](#_Toc180089440)

[3. Memory Usage 10](#_Toc180089441)

[3.1 Memory Usage of the Queue 10](#_Toc180089442)

[3.2 Memory Usage of the Visited Set 11](#_Toc180089443)

[3.3 Overall Space Complexity 11](#_Toc180089444)

[Total Space Complexity: 12](#_Toc180089445)

[3.4 Additional Considerations 12](#_Toc180089446)

# 1. Introduction

## Topic

Implement breadth-first search:

* Implement own graph data structure
* Add nodes
* Connect nodes
* Implement connections with linked-lists (use a library API for the lists)
* Use an array to implement the queue you will need while performing BFS
* How much memory will the queue need, in general, for input graph with n nodes?

## Introduction

Breadth-First Search (BFS) is a fundamental algorithm for traversing graphs, where nodes are explored in a breadthward motion—meaning all neighbors of a node are visited before moving on to the next level. Starting from a designated source node, BFS explores the closest nodes first, expanding to the rest of the graph systematically. It is often used for tasks such as finding the shortest path in unweighted graphs, determining if a graph is connected, or solving maze-related problems.

Custom Implementation

In this project, BFS is implemented using an array-based queue to manage the nodes during traversal, adhering to the assignment requirements. The graph structure is developed from previous users, featuring nodes connected via linked lists, which offer flexible and efficient handling of dynamic neighbor connections. The BFS queue is implemented as a raw array, ensuring strict control over memory allocation.

Space Complexity Analysis

The focus of the report includes an analysis of the space complexity, particularly in regard to the queue’s memory usage. The array, designed to store the nodes during traversal, grows with the number of nodes being explored. This is especially relevant in scenarios where all nodes could be queued simultaneously, thus requiring memory for the entire node set.

GUI Visualization & Target Node Input

In a further enhancement, the implementation includes GUI-based visualization. This addition allows users to visualize the BFS traversal in real-time, seeing which nodes are visited and in what order. Moreover, the GUI enables users to manually input the target node, offering a more interactive experience with the algorithm.

This combination of BFS, custom graph structure, manual input, and visualization provides both theoretical and practical insights into graph traversal techniques. The memory management of the BFS array and its impact on overall performance are documented to ensure clarity regarding space complexity considerations.

## Plan & Work logging

|  |  |  |
| --- | --- | --- |
| Name | Student ID | Subtopic |
| Birkaran Singh | 300196917 | Report |
| Elijah Duchak | 300162556 | Algorithm |
| Wenjun Chen | 300213502 | BFS graph & algorithm |

|  |  |
| --- | --- |
| October 2nd | Created the WhatsApp group and GitHub for assignment discussion |
| October 3rd | Discussed the assignment and the subtopic part for each |
| October 5th | Finish the BFS graph data structure and explanation |
| October 6th -7th | Initial algorithm |
| October 8th -15th | Improvement algorithm |
| October 11th to 16th | Writing and finalizing the report |

**GitHub detail:** <https://github.com/haxetros/Comp-359-Assignment-2.git>

# 2. BFS Algorithm and Graph Implementation

## 2.1 Graph Data Structure (n = 15)

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**, where each node holds references to its neighboring nodes. To efficiently manage the connections, a **linked list** is used to represent the list of neighbors for each node, as specified in the assignment.

The **linked list** allows for dynamic addition and removal of neighbors, making it an appropriate choice for graphs where the number of neighbors per node can vary.

**Node<T> Class Representation**

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

**Components:**

* **T value**:  
  This field holds the data stored in the node. It could represent a unique identifier such as a letter, number, or string.
* **List<Node<T>> neighbors**:  
  A **linked list** that contains references to neighboring nodes. This linked list allows dynamic management of node connections (i.e., adding or removing neighbors as needed).
* **Node<T> connect(Node<T> node)** method:  
  This method establishes an **edge** (connection) between the current node and another node. It ensures that the connection is **bidirectional**—meaning if Node A is connected to Node B, Node B is also connected back to Node A. This is achieved by adding each node to the other’s list of neighbors.

A screen shot of a computer program

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

**Explanation:**

* **Value**: Each node has a value representing its data.
* **LinkedList**: The neighbors of each node are stored in a LinkedList, which allows efficient dynamic modification.
* **connect()**: The connection between nodes is bidirectional, so when Node A connects to Node B, both will reference each other in their respective neighbor lists.

**Queue Implementation**

For the BFS traversal, the **queue** is updated to handle nodes using a raw array, as required by the assignment. Since we are not using Java's built-in Queue class, we manually handle the enqueue and dequeue operations using the **head** and **tail** pointers on the array.

A screen shot of a computer program

Description automatically generated**Queue Code Using Raw Array:**

**Explanation:**

* The **queue** is an array of Node<T> objects.
* The **enqueue** method adds nodes to the queue at the tail, while the **dequeue** method removes them from the head.
* The queue operates in a **FIFO** manner, adhering to BFS requirements.

## 2.2 Breadth-First Search (BFS) Algorithm Implementation

The **Breadth-First Search (BFS)** algorithm explores a graph level by level, starting from a designated node (called the starting node) and systematically traversing through its neighboring nodes before progressing to the next level of nodes. BFS is particularly useful for finding the shortest path between nodes in an unweighted graph.

In this implementation, the BFS algorithm uses a **raw array** to implement the queue, adhering to the assignment's requirements to avoid built-in Java library data structures like ArrayDeque. The key idea behind BFS is to use a **queue**, which operates on a **First In, First Out (FIFO)** principle, ensuring that nodes are explored in the order they are encountered.

### **Queue Implementation Using Raw Array**

The queue is implemented as a fixed-size array of Node<T>, where T is the type of data stored in the nodes (for example, strings or integers). The array size is defined by n, the number of nodes in the graph, since in the worst-case scenario, all nodes might need to be enqueued during the traversal.

#### **Key Components of the Queue Implementation**:

* **Node<T>[] queue = new Node[n];**  
  This array represents the queue that holds nodes waiting to be processed. The size n corresponds to the total number of nodes in the graph.
* **Head Pointer (head)**:  
  The head pointer tracks the front of the queue, where nodes are dequeued (removed for processing). It starts at index 0 and increments as nodes are processed.
* **Tail Pointer (tail)**:  
  The tail pointer tracks the end of the queue, where new nodes are enqueued (added). It also starts at index 0 and increments as new nodes are discovered and added to the queue.

#### **Steps of BFS Traversal**:

1. **Enqueue the Starting Node**:  
   The BFS traversal begins by **enqueuing the starting node** (i.e., adding it to the queue at the tail position). The node is marked as **visited** to prevent reprocessing.
2. **Dequeue a Node**:  
   Next, a node is **dequeued** from the head position of the queue for processing. The neighbors of the current node are then examined.
3. **Enqueue Unvisited Neighbors**:  
   For each unvisited neighbor of the current node, the algorithm **enqueues the neighbor** at the tail position of the queue and marks it as visited. This ensures that all nodes are processed in the correct level order.
4. **Continue Until Target or Queue is Empty**:  
   The process repeats: nodes are dequeued, their neighbors are examined, and unvisited neighbors are enqueued. This continues until the target node is found or the queue becomes empty, indicating that all reachable nodes have been explored.

#### **Handling Queue Overflow**:

In the case where the queue becomes full (i.e., the tail pointer reaches n), a **queue overflow** exception is raised. This serves as a safeguard to ensure that the algorithm does not attempt to enqueue more nodes than the array can handle.

#### **Termination**:

The BFS terminates when:

* The **target node** is found, or
* The **queue becomes empty**, meaning there are no more nodes to explore.

### **Edge Case Considerations**:

* **Empty Graph**: If the graph is empty (i.e., n = 0), the algorithm terminates immediately since there are no nodes to explore.
* **Disconnected Graph**: If the graph is disconnected, BFS will only explore the connected component containing the starting node.

A screenshot of a computer program

Description automatically generatedThe implementation of BFS looks like this:

**BFS Operation Example**

Consider a small graph where we initiate the Breadth-First Search (BFS) at node A, with the goal of locating the target node O. The BFS traversal can be described in the following steps:

1. **Enqueue the Starting Node**: The BFS begins by enqueuing the starting node, A, marking it as visited to ensure it is not processed again. The queue now contains only A.
2. **Explore Neighbors**: Next, we dequeue node A from the front of the queue. At this point, we examine A’s neighboring nodes, which include B, C, D, and E. Each unvisited neighbor is enqueued and marked as visited, so the queue now contains B, C, D, and E.
3. **Level-by-Level Traversal**: With all neighbors of A processed, BFS continues to the next level by dequeuing the next node in the queue. The algorithm dequeues node B and checks its neighbors. If B has unvisited neighbors, they are enqueued similarly. This process repeats for nodes C, D, and E.
4. **Searching for the Target Node**: As BFS proceeds level by level, each dequeued node’s neighbors are explored in turn. If, at any point, the target node O is found among the neighbors, the search terminates successfully.
5. **Termination of the Algorithm**: The BFS continues until either the target node O is discovered or the queue becomes empty, indicating that all reachable nodes have been explored without finding O.

# 3. Memory Usage

## 3.1 Memory Usage of the Queue

A screenshot of a computer

Description automatically generatedThe 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.

**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) for the queue. Thus, the total space complexity due to the queue is 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) 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 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.
* The visited set requires 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. Since each node is added to the visited set only once, the total memory usage for the set also amounts to O(n).

## 3.3 Overall Space Complexity

The space complexity of the BFS algorithm is largely influenced by two key components:

1. **Queue Memory**:  
   The queue holds nodes that are being processed or are awaiting processing. In the worst case, the queue may need to hold all nodes in a single level of the graph. The maximum memory required by the queue is proportional to the number of nodes, which means the queue will require O(n) space, where **n** is the total number of nodes in the graph.
2. **Visited Set Memory**:  
   The visited set keeps track of all the nodes that have already been processed to prevent revisiting the same nodes and causing infinite loops in cyclic graphs. As each node is marked as visited once, the visited set also requires O(n) space, since we may need to store information about all **n** nodes.

## Total Space Complexity:

Considering both the queue and the visited set, the total space complexity of the BFS algorithm can be expressed as the sum of the two:

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 means that, in the worst case, BFS will require O(n) memory for a graph with **n** nodes.

## 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.

**References**  
  
Google. (n.d.). *BFS\_graph.Drawio*. Google Drive. ” Wenjun Chen” <https://drive.google.com/file/d/19d9LIW6XkPM8Lk1ZaSoW5GjuwOEIileF/view?usp=sharing>

Google. (n.d.). *BFS\_graph*. Google Docs. ” Wenjun Chen” <https://docs.google.com/document/d/1uZ7L2CQAXR08pcV8aus-xiloFTa3PQ_Q2X40wkUlf44/edit?usp=sharing>

Dupire, W. by: F. (2024, January 25). *Breadth-first search algorithm in Java*. Baeldung. <https://www.baeldung.com/java-breadth-first-search>

*Source of graphpanel.java*. GraphPanel.java. (n.d.). <https://cs.smu.ca/~porter/csc/465/code/horstmann/ch08/graphed/GraphPanel.java2html>

Fadi-Eid. (n.d.). *GraphPanel-Java/app at main · fadi-eid/graphpanel-java*. GitHub. <https://github.com/Fadi-Eid/GraphPanel-Java/tree/main/App>

