Graph and BST Representations

Ryan Davis

Ryan.Davis3@Marist.edu

November 16, 2024

This document explores the representation and analysis of undirected graphs and binary search trees (BSTs). The graphs are represented as adjacency matrices, adjacency lists, and linked objects. For linked objects, depth-first and breadth-first traversals are performed. Additionally, a BST is implemented to store items, with insertion paths and in-order traversal outputs recorded. Asymptotic analyses for both data structures are provided.

Contents

1 Introduction **3**

2 Graph Representations and Explanations **3**
- 2.1 Adjacency Matrix Implementation . . . 3
- 2.2 Adjacency List Implementation . . . 4
- 2.3 Linked Objects Representation . . . 5

3 Graph Traversals and Explanations **6**
- 3.1 Depth-First Traversal . . . 7
- 3.2 Breadth-First Traversal . . . 7

4 Binary Search Tree Implementation and Explanations **8**
- 4.1 BST Node and Insertion . . . 8
- 4.2 In-Order Traversal . . . 9
- 4.3 Lookup Operation . . . 10

5 Asymptotic Analysis **11**
- 5.1 Graph Representations . . . 11
  - 5.1.1 Space Complexity . . . 11
  - 5.1.2 Edge Lookup Time Complexity . . . 12
  - 5.1.3 Neighbor Iteration Time Complexity . . . 12
- 5.2 Graph Traversals . . . 12
- 5.3 Binary Search Tree Operations . . . 13

1 Introduction

This document details implementations and analyses of graph representations and traversal algorithms, as well as binary search tree operations. The graphs are represented in three forms: adjacency matrices, adjacency lists, and linked objects. We perform depth-first and breadth-first traversals on the linked objects representation. A binary search tree (BST) is also implemented to store items, with detailed analysis of insertion paths and traversal outputs. Asymptotic running times of all operations are analyzed, providing insights into their efficiency and scalability.

2 Graph Representations and Explanations

In this section, we provide code listings and detailed explanations for the implementations of the adjacency matrix, adjacency list, and linked objects representations of a graph.

2.1 Adjacency Matrix Implementation

```java
1 public class Graph {
2     private int[][] adjacencyMatrix; // Line 2
3     private Map<String, Integer> vertexIndexMap; // Line 3
4     private int vertexCount; // Line 4
5 
6     public Graph() { // Line 6
7         this.adjacencyMatrix = new int[0][0]; // Line 7
8         this.vertexIndexMap = new HashMap<>(); // Line 8
9         this.vertexCount = 0; // Line 9
10     }
11 
12     public void addVertex(String vertexId) { // Line 11
13         if (!vertexIndexMap.containsKey(vertexId)) { // Line 12
14             vertexIndexMap.put(vertexId, vertexCount); // Line 13
15             vertexCount++; // Line 14
16             adjacencyMatrix = resizeMatrix(adjacencyMatrix, vertexCount); // Line 15
17         }
18     }
19 
20     public void addEdge(String u, String v) { // Line 18
21         if (vertexIndexMap.containsKey(u) && vertexIndexMap.containsKey(v)) { // Line 19
22             int uIndex = vertexIndexMap.get(u); // Line 20
23             int vIndex = vertexIndexMap.get(v); // Line 21
24             adjacencyMatrix[uIndex][vIndex] = 1; // Line 22
25             adjacencyMatrix[vIndex][uIndex] = 1; // Line 23
26         }
27     }
28 
29     private int[][] resizeMatrix(int[][] oldMatrix, int newSize) { // Line 26
30         int[][] newMatrix = new int[newSize][newSize]; // Line 27
31         for (int i = 0; i < oldMatrix.length; i++) { // Line 28
32             System.arraycopy(oldMatrix[i], 0, newMatrix[i], 0, oldMatrix[i].length); // Line 29
33         }
34         return newMatrix; // Line 31
35     }
36 }
```

Listing 1: Adjacency Matrix Creation and Edge Addition

Explanation:

- **Line 2**: Declares the adjacency matrix as a two-dimensional integer array.
- **Line 3**: Declares a map to associate vertex IDs (`String`) with their indices (`Integer`) in the adjacency matrix.

- **Line 4**: Initializes a counter for the number of vertices in the graph.
- **Line 6**: Defines the constructor for the `Graph` class.
- **Line 7**: Initializes the adjacency matrix to an empty 0x0 matrix.
- **Line 8**: Initializes the `vertexIndexMap` as a new `HashMap`.
- **Line 9**: Sets the initial `vertexCount` to zero.
- **Line 11**: Declares the `addVertex` method to add a new vertex to the graph.
- **Line 12**: Checks if the vertex ID is not already in the `vertexIndexMap`.
- **Line 13**: Adds the vertex ID to the `vertexIndexMap` with the current `vertexCount` as its index.
- **Line 14**: Increments the `vertexCount` to account for the new vertex.
- **Line 15**: Calls `resizeMatrix` to adjust the size of the adjacency matrix to accommodate the new vertex.
- **Line 18**: Declares the `addEdge` method to add an edge between two vertices.
- **Line 19**: Checks if both vertices exist in the `vertexIndexMap`.
- **Line 20**: Retrieves the index of vertex `u` from the `vertexIndexMap`.
- **Line 21**: Retrieves the index of vertex `v` from the `vertexIndexMap`.
- **Line 22**: Sets the adjacency matrix entry `[uIndex][vIndex]` to 1 to represent an edge from `u` to `v`.
- **Line 23**: Sets the adjacency matrix entry `[vIndex][uIndex]` to 1 to represent the undirected edge.
- **Line 26**: Declares the `resizeMatrix` method to resize the adjacency matrix.
- **Line 27**: Creates a new two-dimensional integer array `newMatrix` with the new size.
- **Line 28**: Begins a loop to copy the old matrix data into the new matrix.
- **Line 29**: Copies each row from the old matrix to the new matrix using `System.arraycopy`.
- **Line 31**: Returns the `newMatrix` after resizing.

Overview:

This code implements a graph using an adjacency matrix. It maintains a mapping from vertex IDs to indices in the matrix, allowing for efficient storage and retrieval of edge information. The graph supports adding vertices and edges, dynamically resizing the adjacency matrix as new vertices are added. The adjacency matrix represents edges with a 1 (edge exists) or 0 (no edge), providing a quick way to check for connections between any two vertices.

2.2 Adjacency List Implementation

```java
public class Graph {
    private Map<String, TreeSet<String>> adjacencyList; // Line 2

    public Graph() { // Line 4
        this.adjacencyList = new HashMap<>(); // Line 5
    }

    public void addVertex(String vertexId) { // Line 7
        adjacencyList.putIfAbsent(vertexId, new TreeSet<>()); // Line 8
    }

    public void addEdge(String u, String v) { // Line 11
        adjacencyList.get(u).add(v); // Line 12
```

```
14          adjacencyList.get(v).add(u); // Line 13
15      }
16
17      public void printAdjacencyList() { // Line 16
18          for (String vertex : adjacencyList.keySet()) { // Line 17
19              System.out.println(vertex + ": " + adjacencyList.get(vertex)); // Line 18
20          }
21      }
22  }
```

Listing 2: Adjacency List Creation and Edge Addition

Explanation:

- **Line 2**: Declares the adjacency list as a map where each vertex ID maps to a `TreeSet` of adjacent vertex IDs.
- **Line 4**: Defines the constructor for the `Graph` class.
- **Line 5**: Initializes the `adjacencyList` as a new `HashMap`.
- **Line 7**: Declares the `addVertex` method to add a new vertex to the graph.
- **Line 8**: Adds the vertex to the `adjacencyList` if it is not already present, initializing its adjacency set.
- **Line 11**: Declares the `addEdge` method to add an undirected edge between two vertices.
- **Line 12**: Adds vertex `v` to the adjacency set of vertex `u`.
- **Line 13**: Adds vertex `u` to the adjacency set of vertex `v`.
- **Line 16**: Declares the `printAdjacencyList` method to display the adjacency list.
- **Line 17**: Iterates over each vertex in the `adjacencyList`.
- **Line 18**: Prints the vertex and its set of adjacent vertices.

Overview:

This code implements a graph using an adjacency list, where each vertex maintains a sorted set of its adjacent vertices. The use of a `TreeSet` ensures that the adjacent vertices are stored in order, which can be helpful for consistent traversal outputs. The graph supports adding vertices and edges, and provides a method to print the adjacency list representation.

2.3 Linked Objects Representation

```
public class Vertex {
    String id; // Line 2
    List<Vertex> neighbors; // Line 3

    public Vertex(String id) { // Line 5
        this.id = id; // Line 6
        this.neighbors = new ArrayList<>(); // Line 7
    }

    public void addNeighbor(Vertex neighbor) { // Line 9
        this.neighbors.add(neighbor); // Line 10
    }
}

public class Graph {
    private Map<String, Vertex> vertices; // Line 14

    public Graph() { // Line 16
        this.vertices = new HashMap<>(); // Line 17
```

```
20     }
21
22     public void addVertex(String vertexId) { // Line 19
23         vertices.putIfAbsent(vertexId, new Vertex(vertexId)); // Line 20
24     }
25
26     public void addEdge(String u, String v) { // Line 23
27         Vertex uVertex = vertices.get(u); // Line 24
28         Vertex vVertex = vertices.get(v); // Line 25
29         uVertex.addNeighbor(vVertex); // Line 26
30         vVertex.addNeighbor(uVertex); // Line 27
31     }
32 }
```

Listing 3: Vertex Class and Graph Implementation

Explanation:

- **Line 2**: Declares the `id` field to store the unique identifier of the vertex.
- **Line 3**: Declares the `neighbors` list to store adjacent vertices.
- **Line 5**: Defines the constructor for the `Vertex` class.
- **Line 6**: Assigns the provided `id` to the vertex.
- **Line 7**: Initializes the `neighbors` list as a new `ArrayList`.
- **Line 9**: Declares the `addNeighbor` method to add a neighbor to the vertex.
- **Line 10**: Adds the neighbor vertex to the `neighbors` list.
- **Line 14**: Declares the `vertices` map to store all vertices in the graph.
- **Line 16**: Defines the constructor for the `Graph` class.
- **Line 17**: Initializes the `vertices` map as a new `HashMap`.
- **Line 19**: Declares the `addVertex` method to add a new vertex to the graph.
- **Line 20**: Adds the vertex to the `vertices` map if it is not already present.
- **Line 23**: Declares the `addEdge` method to add an edge between two vertices.
- **Line 24**: Retrieves vertex `u` from the `vertices` map.
- **Line 25**: Retrieves vertex `v` from the `vertices` map.
- **Line 26**: Adds vertex `v` as a neighbor of vertex `u`.
- **Line 27**: Adds vertex `u` as a neighbor of vertex `v` to represent an undirected edge.

Overview:

This code represents a graph using linked objects, where each vertex is an object containing its identifier and a list of neighbor vertices. The graph maintains a map of vertex IDs to vertex objects for quick access. This representation is conducive to implementing traversal algorithms like DFS and BFS, as it allows direct navigation between connected vertices through object references.

3 Graph Traversals and Explanations

Depth-first search (DFS) and breadth-first search (BFS) are fundamental graph traversal algorithms. Below are the implementations and explanations for these traversals on the linked objects representation.

3.1 Depth-First Traversal

```java
1 public void depthFirstTraversal(Vertex vertex, Set<String> visited) { // Line 1
2     if (!visited.add(vertex.id)) return; // Line 2
3     System.out.print(vertex.id + " "); // Line 3
4     vertex.neighbors.sort(Comparator.comparingInt(v -> Integer.parseInt(v.id))); // Line 4
5     for (Vertex neighbor : vertex.neighbors) { // Line 5
6      if (!visited.contains(neighbor.id)) {//line 6
7                 depthFirstTraversal(neighbor, visited);//line 7
8             }
9     }
10 }
```

Listing 4: Depth-First Traversal Implementation

Explanation:

- **Line 1**: Declares the `depthFirstTraversal` method, which takes a `Vertex` object and a `Set` of visited vertex IDs.
- **Line 2**: Attempts to add the current vertex ID to the `visited` set; if it was already visited, returns to avoid cycles.
- **Line 3**: Prints the current vertex ID to the console.
- **Line 4**: Sorts the neighbors of the current vertex numerically by their IDs to ensure a consistent traversal order.
- **Line 5**: Begins a loop to iterate over each neighbor in the sorted `neighbors` list.
- **L**ine 6: Checks if the neighbor vertex has not been visited yet by seeing if its id is not in the visited set.
- **L**ine 7: If the neighbor is unvisited, recursively calls the depthFirstTraversal method on the neighbor, passing along the visited set.

Overview:

The depth-first traversal method recursively explores as far as possible along each branch before backtracking. By marking visited vertices, it avoids infinite loops in cyclic graphs. Sorting neighbors ensures that traversal order is consistent across runs.

3.2 Breadth-First Traversal

```java
public void breadthFirstTraversal(Vertex startVertex) { // Line 1
    Set<String> visited = new HashSet<>(); // Line 2
    Queue<Vertex> queue = new LinkedList<>(); // Line 3
    queue.add(startVertex); // Line 4
    visited.add(startVertex.id); // Line 5

    while (!queue.isEmpty()) { // Line 7
        Vertex current = queue.poll(); // Line 8
        System.out.print(current.id + " "); // Line 9
        current.neighbors.sort(Comparator.comparingInt(v -> Integer.parseInt(v.id))); // Line 10
        for (Vertex neighbor : current.neighbors) { // Line 11
            if (visited.add(neighbor.id)) { // Line 12
                queue.add(neighbor); // Line 13
            }
        }
    }
}
```

Listing 5: Breadth-First Traversal Implementation

Explanation:

- **Line 1**: Declares the `breadthFirstTraversal` method starting from a given `startVertex`.
- **Line 2**: Initializes a `Set` to keep track of visited vertex IDs.
- **Line 3**: Initializes a `Queue` to manage the order of traversal.
- **Line 4**: Adds the `startVertex` to the queue.
- **Line 5**: Marks the `startVertex` as visited by adding its ID to the `visited` set.
- **Line 7**: Begins a loop that continues until the queue is empty.
- **Line 8**: Retrieves and removes the head of the queue, assigning it to `current`.
- **Line 9**: Prints the ID of the `current` vertex.
- **Line 10**: Sorts the neighbors of the `current` vertex numerically.
- **Line 11**: Begins a loop to iterate over each neighbor of the `current` vertex.
- **Line 12**: Attempts to add the neighbor's ID to the `visited` set; if successful (not already visited), proceeds.
- **Line 13**: Adds the neighbor to the queue for future traversal.

Overview:

The breadth-first traversal method explores all neighbors at the current depth before moving to the next level. It uses a queue to track the order of vertices to visit.

4 Binary Search Tree Implementation and Explanations

The binary search tree (BST) is a data structure that facilitates efficient insertion, deletion, and lookup operations. Below are the implementations and explanations for the BST used in this assignment.

4.1 BST Node and Insertion

```java
public class BST {
    static class Node {
        String value; // Line 3
        Node left, right; // Line 4

        Node(String value) { // Line 6
            this.value = value; // Line 7
            this.left = this.right = null; // Line 8
        }
    }

    private Node root; // Line 11

    public void insert(String value) { // Line 13
        StringBuilder path = new StringBuilder(); // Line 14
        root = insertRecursive(root, value, path); // Line 15
        System.out.println("Inserted " + value + " with path: " + path); // Line 16
    }

    private Node insertRecursive(Node current, String value, StringBuilder path) { // Line 18
        if (current == null) { // Line 19
            return new Node(value); // Line 20
        }
        if (value.compareTo(current.value) < 0) { // Line 22
            path.append("L, "); // Line 23
            current.left = insertRecursive(current.left, value, path); // Line 24
        } else if (value.compareTo(current.value) > 0) { // Line 26
            path.append("R, "); // Line 27
```

```
29            current.right = insertRecursive(current.right, value, path); // Line 28
30        }
31        return current; // Line 30
32    }
33}
```

Listing 6: BST Node and Insertion Methods

Explanation:

- **Line 3**: Declares the `value` field to store the data in the node.
- **Line 4**: Declares the `left` and `right` child nodes.
- **Line 6**: Defines the constructor for the `Node` class.
- **Line 7**: Assigns the provided `value` to the node.
- **Line 8**: Initializes the `left` and `right` children to `null`.
- **Line 11**: Declares the `root` of the BST.
- **Line 13**: Declares the `insert` method to add a value to the BST.
- **Line 14**: Initializes a `StringBuilder` to record the path taken during insertion.
- **Line 15**: Calls `insertRecursive` to perform the actual insertion.
- **Line 16**: Prints a message indicating the value inserted and the path taken.
- **Line 18**: Declares the `insertRecursive` helper method.
- **Line 19**: Checks if the current node is `null`; if so, creates a new node with the value.
- **Line 20**: Returns the new node to be attached to the parent.
- **Line 22**: Compares the value to insert with the current node's value.
- **Line 23**: Appends "L, " to the path, indicating a move to the left child.
- **Line 24**: Recursively calls `insertRecursive` on the left child.
- **Line 26**: Else if the value is greater, proceeds to the right child.
- **Line 27**: Appends "R, " to the path.
- **Line 28**: Recursively calls `insertRecursive` on the right child.
- **Line 30**: Returns the current node to maintain the tree structure.

Overview:

This code defines a BST with methods for inserting values. Each node contains a value and references to its left and right children. The insertion method places new values in the correct position to maintain the BST property, where left descendants are less than the node and right descendants are greater. The path taken during insertion is recorded and printed.

4.2 In-Order Traversal

```
1 public void inOrderTraversal() { // Line 1
2     System.out.println("In-Order Traversal of BST:"); // Line 2
3     inOrderRecursive(root); // Line 3
4     System.out.println(); // Line 4
5 }
6
7 private void inOrderRecursive(Node node) { // Line 6
8     if (node == null) return; // Line 7
```

```java
9       inOrderRecursive(node.left); // Line 8
10      System.out.print(node.value + " "); // Line 9
11      inOrderRecursive(node.right); // Line 10
12  }
```

Listing 7: In-Order Traversal Method

Explanation:

- **Line 1**: Declares the `inOrderTraversal` method to initiate the traversal.
- **Line 2**: Prints a header message for the traversal.
- **Line 3**: Calls the `inOrderRecursive` helper method starting from the root.
- **Line 4**: Prints a newline after traversal is complete.
- **Line 6**: Declares the `inOrderRecursive` helper method.
- **Line 7**: Base case: if the node is `null`, returns.
- **Line 8**: Recursively calls `inOrderRecursive` on the left child.
- **Line 9**: Prints the value of the current node.
- **Line 10**: Recursively calls `inOrderRecursive` on the right child.

Overview:

The in-order traversal method visits nodes in ascending order for a BST. It first visits the left leaf, then the current node, and finally the right leaf. This traversal will give you the elements in sorted order.

4.3 Lookup Operation

```java
1   public int find(String value) { // Line 1
2       StringBuilder path = new StringBuilder(); // Line 2
3       int comparisons = findRecursive(root, value, path, 0); // Line 3
4       if (comparisons != -1) { // Line 4
5           System.out.println("Found " + value + " with path: " + path + " in " + comparisons + " comparisons."); // Line 5
6       } else {
7           System.out.println(value + " not found in BST."); // Line 7
8       }
9       return comparisons; // Line 8
10  }
11
12  private int findRecursive(Node current, String value, StringBuilder path, int comparisons) { // Line 10
13      if (current == null) return -1; // Line 11
14      comparisons++; // Line 12
15      if (value.equals(current.value)) return comparisons; // Line 13
16      if (value.compareTo(current.value) < 0) { // Line 15
17          path.append("L, "); // Line 16
18          return findRecursive(current.left, value, path, comparisons); // Line 17
19      } else {
20          path.append("R, "); // Line 19
21          return findRecursive(current.right, value, path, comparisons); // Line 20
22      }
23  }
```

Listing 8: BST Lookup Method

Explanation:

- **Line 1**: Declares the `find` method to search for a value in the BST.
- **Line 2**: Initializes a `StringBuilder` to record the search path.

- **Line 3**: Calls `findRecursive` to perform the actual search.
- **Line 4**: Checks if the value was found (`comparisons` not equal to -1).
- **Line 5**: If found, prints the value, path, and number of comparisons.
- **Line 7**: If not found, prints a message indicating the value was not found.
- **Line 8**: Returns the number of comparisons made during the search.
- **Line 10**: Declares the `findRecursive` helper method.
- **Line 11**: Base case: if the current node is `null`, returns -1 indicating not found.
- **Line 12**: Increments the comparison count.
- **Line 13**: Checks if the current node's value matches the search value.
- **Line 15**: If the search value is less than the current node's value, proceeds left.
- **Line 16**: Appends "L, " to the path.
- **Line 17**: Recursively calls `findRecursive` on the left child.
- **Line 19**: Else, proceeds right and appends "R, " to the path.
- **Line 20**: Recursively calls `findRecursive` on the right child.

Overview:

This code implements a lookup operation in the BST. It traverses the tree moving left or right depending on the comparison with the current node's value. It records the path taken and counts the number of comparisons made. This method finds a value if it exists in the tree and provides.

5 Asymptotic Analysis

This section provides theoretical time and space complexities and analyses for the graph representations, traversal algorithms, and BST operations, including discussions on edge lookups, neighbor iteration.

5.1 Graph Representations

5.1.1 Space Complexity

Adjacency Matrix:

- **Space Complexity:** $\mathcal{O}(V^2)$, where V is the number of vertices.
- **Explanation:** An adjacency matrix requires storage for every possible pair of vertices, resulting in $V \times V$ space usage regardless of the number of edges.

Adjacency List:

- **Space Complexity:** $\mathcal{O}(V + E)$, where E is the number of edges.
- **Explanation:** The adjacency list stores only existing edges, with each edge represented once (or twice for undirected graphs), leading to linear space in the number of vertices and edges.

Linked Objects Representation:

- **Space Complexity:** $\mathcal{O}(V + E)$.
- **Explanation:** Similar to the adjacency list, each vertex object contains references to its neighbors, resulting in space proportional to the sum of vertices and edges.

5.1.2 Edge Lookup Time Complexity

Adjacency Matrix:

- **Time Complexity:** $\mathcal{O}(1)$.
- **Explanation:** Checking for the existence of an edge between two vertices involves a direct index access in the matrix.

Adjacency List:

- **Time Complexity:** $\mathcal{O}(k)$, where k is the degree of the vertex.
- **Explanation:** Edge lookup requires scanning the adjacency list of a vertex, which is proportional to its degree.

Linked Objects Representation:

- **Time Complexity:** $\mathcal{O}(k)$.
- **Explanation:** Similar to the adjacency list, checking for a neighbor involves iterating over the list of adjacent vertices.

5.1.3 Neighbor Iteration Time Complexity

Adjacency Matrix:

- **Time Complexity:** $\mathcal{O}(V)$.
- **Explanation:** Iterating over neighbors requires scanning an entire row (or column) in the matrix.

Adjacency List:

- **Time Complexity:** $\mathcal{O}(k)$.
- **Explanation:** Neighbors are stored directly in a list, allowing iteration proportional to the number of adjacent vertices.

Linked Objects Representation:

- **Time Complexity:** $\mathcal{O}(k)$.
- **Explanation:** Each vertex contains a list of its neighbors, enabling efficient iteration.

5.2 Graph Traversals

Depth-First Search (DFS):

- **Time Complexity:** $\mathcal{O}(V + E)$.
- **Space Complexity:** $\mathcal{O}(V)$ due to the recursion stack and visited set.
- **Explanation:** DFS explores each vertex and edge once. The traversal is efficient for both dense and sparse graphs.

Breadth-First Search (BFS):

- **Time Complexity:** $\mathcal{O}(V + E)$.
- **Space Complexity:** $\mathcal{O}(V)$ due to the queue and visited set.
- **Explanation:** BFS also visits each vertex and edge once, processing vertices in a level-order fashion.

5.3 Binary Search Tree Operations

Insertion and Lookup:

- **Average Case Time Complexity:** $\mathcal{O}(\log n)$, where n is the number of nodes.
- **Worst Case Time Complexity:** $\mathcal{O}(n)$.
- **Space Complexity:** $\mathcal{O}(n)$ for storing n nodes.
- **Explanation:** n a balanced BST, operations occur efficiently due to the minimal height of the tree. However, if the tree becomes skewed (unbalanced), its height can increase significantly, up to the number of nodes n, resulting in linear time operations.

References

[1] Oracle, *Java Platform, Standard Edition Documentation*, 2023.

[2] Cormen, T.H., Leiserson, C.E., Rivest, R.L., and Stein, C., *Introduction to Algorithms*, Third Edition, MIT Press, 2009.