Assignment Four

Josh Seligman

joshua.seligman1@marist.edu

November 18, 2022

1 Binary Search Tree

1.1 The Data Structure

A binary search tree is a data structure that, for each node in the tree, all child nodes on its left are less than the value and all child nodes on the right are greater than or equal to the value in the given node. Therefore, when performing an in-order traversal by printing out all left-hand nodes, then the value of the given node, and lastly all of the right-hand nodes, all values will be printed out in order. As shown in Figures 1.1 and 1.2, the values are inserted into the tree in the order in which they are received. This can impact the time it takes to traverse the tree to find a given element, which will be examined in Section 1.2.

1.2 Asymptotic Analysis

Algorithm 1 provides the pseudocode for performing a lookup on a binary search tree. As shown on lines 6 and 8, the area of the tree gets cut in half for each level of the recursion tree. This causes the expected runtime for a binary search tree lookup to be the same as binary search at $O(log_2 n)$. However, as displayed in Figures 1.1 and 1.2, the order in which the data arrive makes a huge difference in the number of checks needed to find an element. For instance, when looking for the number 8 in the tree in Figure 1.1, it will start with the 5 and go to the right because $8 > 5$, then compare 8 with the 7 and also go to the right because 8

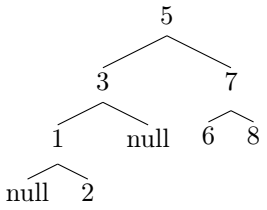

Figure 1.1: Sample binary search tree for the numbers 5, 3, 1, 2, 7, 6, 8.

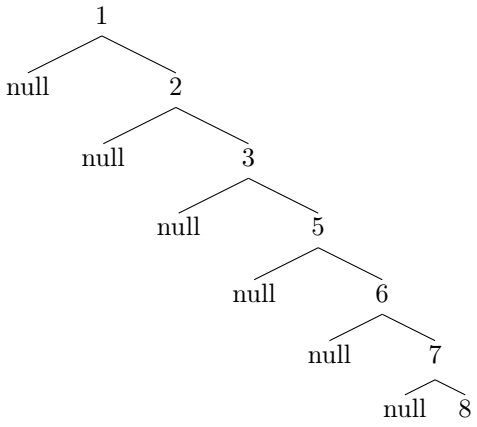

Figure 1.2: Sample binary search tree for the numbers 1, 2, 3, 5, 6, 7, 8.

Algorithm 1 Binary Search Tree Lookup. Assume *cur* starts off as the root of the tree.

```
1:  procedure BSTLookup(target, cur)
2:      out ← false    // Assume target is not found
3:      if target == cur.val then
4:          out ← true    // Found the target value
5:      else if target < cur.val then
6:          out ← BSTLookup(target, cur.left)    // Target is on the left
7:      else   // target ≥ cur.val
8:          out ← BSTLookup(target, cur.right)    // Target is on the right
9:      end if
10:     return out
11: end procedure
```

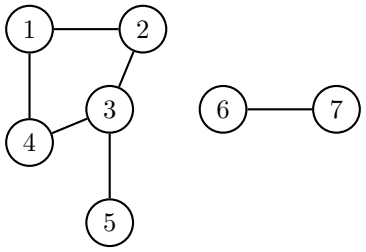

Figure 2.1: Sample undirected, unweighted graph consisting of 7 vertices and 6 edges.

> 7, and then end when it finds the 8. Since there are 7 elements in the tree, it should take around $log_2 7$ comparisons to find an element, and 3 comparisons is very close to the expected outcome. However, when trying to find 8 in the tree in Figure 1.2, one will have to compare 8 with every single element in the tree, which degrades the binary search tree lookup to be $O(n)$ as it is just doing a linear search. Therefore, when working with a binary search tree, it is important to make sure the data are shuffled in a random order to ensure the tree is as close to being balanced as possible. Since shuffling is an $O(n)$ operation, taking the time to shuffle a sorted array before putting the data in a binary search tree will save a lot of time in the long run especially when doing a lot of lookups within the tree.

2 Graphs

2.1 The Data Structure

A graph is a data structure that is composed of vertices that are connected via edges. Graphs can be useful for modeling networks of objects, such as LinkedIn connections or the many locations within a city map. The edges that connect the vertices together may have direction and weights, but this paper will focus on undirected and unweighted graphs. As illustrated in Figure 2.1, some of the vertices are connected to each other with various levels of cardinality. Additionally, some vertices may be disconnected from some areas of the graph. This is the case with vertices 6 and 7, which are on their own island separate from the rest of the graph.

Traversing a graph is commonly done by using either a depth-first search or breadth-first search. As per the name, depth-first search works to traverse through the graph with the goal of going as deep into the graph as quickly as possible. Algorithm 2 provides the pseudocode for depth-first search. Lines 6-10 emphasize the nature of the algorithm as a depth-first search is recursively performed on the first neighbor, which will not return until it has exhaustively searched through every node that can be reached through the neighbor before moving on to the next neighbor. Alternatively, one can use a breadth-first search to traverse a graph, by going through all of the direct connections to the starting vertex, then going through those vertices' connections, and so on until all levels are searched. As shown in Algorithm 3, a queue is used to keep track of the order of the vertices that have to be searched as those closer to the initial vertex are going to be enqueued first and, therefore, will be checked first. The order in which vertices are processed varies between the 2 algorithms. For instance, in Figure 2.1, a depth-first search starting at vertex 1 will go to vertex 2 and then vertex 3. From vertex 3, vertex 4 will be searched, but there are no connections to vertex 4 that have already been searched, which causes the recursion to end and allow for vertex 5 to be searched. Thus, the order is 1, 2, 3, 4, 5. On the other hand, with a breadth-first search, after seeing vertex 1, vertices 2 and 4 will be added to the queue first. When visiting vertex 2, vertex 3 will be added to the queue, and vertex 5 will be added to the queue when visiting vertex 3. Because of the use of the queue, the order of the vertices being checked becomes 1, 2, 4, 3, 5.

Algorithm 2 Graph Depth-First Search. *start* can begin at any point in the graph.

```
1: procedure GraphDFS(start)
2:     if !start.isProcessed then    // Make sure the vertex has not already been visited
3:         print start.id
4:         start.isProcessed ← true
5:     end if
6:     for neighborVertex in start.neighbors do    // Check all neighbors
7:         if !neighbor.isProcessed then    // Go down the rabbit hole if the neighbor has not been visited
8:             GraphDFS(neighbor)
9:         end if
10:    end for
11: end procedure
```

Algorithm 3 Graph Breadth-First Search. *start* can begin at any point in the graph.

```
1: procedure GraphBFS(start)
2:     checkQueue ← new Queue
3:     checkQueue.enqueue(start)    // Add the start to the need to check queue
4:     start.isProcessed ← true
5:     while !checkQueue.isEmpty() do
6:         checkVertex ← checkQueue.dequeue()
7:         print checkVertex.id
8:         for neighbor in checkVertex.neighbors do    // Check all neighbors
9:             if !neighbor.isProcessed then    // Add it to the queue if not visited
10:                checkQueue.enqueue(neighbor)
11:                neighbor.isProcessed ← true
12:            end if
13:        end for
14:    end while
15: end procedure
```

2.2 Asymptotic Analysis

In the case of graph traversals through depth-first and breadth-first searches, the worst possible case is when the graph is complete and all vertices are connected to every other vertex as this is the situation with the most number of edges and connections between nodes. Redundant connections change the graph to be a multigraph, which is not the focus of the paper and will be ignored.

Depth-first search is described in Algorithm 2, and its C++ implementation can be found in Listing 2. In Algorithm 2, lines 2-5 define an if-statement that contains a check to see if a variable is true or false, a print statement, and an assignment. All of these operations are constant time operations and run in $O(1)$ time. Next, lines 6-10 define a for-loop that iterates over all of the vertex's neighbors. In the worst case described above, this loop will run for each of the neighbors, which is every other vertex, or $v - 1$ times. Inside the loop, there is an if-statement that runs in constant just like the one on line 2 as well as a recursive call to the function. Since the function is only called if the neighbor has not been visited yet, it will run once for each vertex. Therefore, the loop will run a total of v times, which causes the algorithm to be $O(v^2)$ because each function call requires a loop over all of the other vertices.

Breadth-first search is described in Algorithm 3, and its C++ implementation can be found in Listing 3. Lines 2-4 of Algorithm 3 contain 2 assignments and a call to an *enqueue* function that all run in constant time. Next, the while-loop defined on line 5 has a condition that will vary based on the number of elements added to the queue over time, which, for now, will be unknown. Lines 6 and 7 consist of dequeue assignment and a print statement, which run in $O(1)$ time. Lines 8-13 define a for-loop that iterates over all neighbors of the vertex being checked. Similar to depth-first search, this loop will run $v-1$ times at the worst case because each vertex is connected to every other vertex, so all vertices have to be checked. However, the neighboring vertices are only added to the end of the queue if they have not been processed yet, which means that each vertex will be added to the queue no more than 1 time. Therefore, the outer loop will run for each vertex in the graph. Also, the body of the for-loop contains all constant time operations that are not dependent on the number of vertices in the graph. Therefore, since the outer loop is running v times and the inner loop runs $v - 1$ times, the runtime of breadth-first search is $O(v^2)$.

The main problem with both depth-first and breadth-first traversals is that they are unaware of any disconnected nodes to the part of the graph they are searching. For instance, as mentioned in the example walkthroughs of these algorithms in Section 2.1 for Figure 2.1, only vertices 1-5 were visited and 6 and 7 were left completely out of the loop because they are separate from the rest of the graph. The only solution to this problem is to pick up the searching algorithms with either vertex 6 or 7. The C++ implementation of these algorithms in Listings 2 and 3 adds an additional $O(v)$ operation on lines 7-16 that iterates through each vertex and starts up the searching algorithm if the vertex has not been visited yet, which helps ensure complete coverage of the graph.

3 Appendix

3.1 Binary Search Tree

```
1  bool BinarySearchTree::search(std::string target, int* comparisons, bool shouldPrintPath) {
2      // Run the search starting with the root of the tree
3      return searchHelper(target, root, comparisons, shouldPrintPath);
4  }
5
6  bool BinarySearchTree::searchHelper(std::string target, BinaryTreeNode<std::string>* cur,
       int* comparisons, bool shouldPrintPath) {
7      // Assume the element is not found (cur is nullptr)
8      bool out = false;
9
10     if (cur != nullptr) {
11         // Compare the strings
12         int strComp = target.compare(cur->data);
```

```
13
14          // A comparison was made, so increment the counter
15          if (comparisons != nullptr) {
16              (*comparisons)++;
17          }
18
19          if (strComp == 0) {
20              // The element was found, so return true
21              out = true;
22          } else if (strComp < 0) {
23              // Check the left side because the target is less than the current value
24              if (shouldPrintPath) {
25                  std::cout << "L";
26              }
27              out = searchHelper(target, cur->left, comparisons, shouldPrintPath);
28          } else {
29              // Check the right side because the target is greater than the current value
30              if (shouldPrintPath) {
31                  std::cout << "R";
32              }
33              out = searchHelper(target, cur->right, comparisons, shouldPrintPath);
34          }
35      }
36
37      return out;
38  }
```

Listing 1: Binary Search Tree Lookup (C++)

3.2 Graphs

```
1  void Graph::depthFirstSearch() {
2      // Make sure everything is cleared
3      clearProcessedStates();
4
5      Node<Vertex*>* start = vertices->getHead();
6
7      while (start != nullptr) {
8          // If the vertex has not been processed, it is disconnected from the rest of the
               graph
9          if (!start->data->isProcessed) {
10             // Start the DFS from the current node to explore that region of the graph
11             std::cout << "DFS: ";
12             depthFirstSearch(start->data);
13             std::cout << std::endl;
14         }
15         start = start->next;
16     }
17 }
18
19 void Graph::depthFirstSearch(Vertex* start) {
20     if (!start->isProcessed) {
21         // Print out the vertex id only if it hasn't been processed yet
22         std::cout << start->getId() << " ";
23         start->isProcessed = true;
24     }
25     // Iterate through all neighbors
26     Node<Vertex*>* cur = start->getNeighbors()->getHead();
27     while (cur != nullptr) {
28         if (!cur->data->isProcessed) {
29             // Run a DFS starting from the neighbor if it hasn't been processed already
30             depthFirstSearch(cur->data);
31         }
32         cur = cur->next;
33     }
```

```
}
```

Listing 2: Graph Depth-First Search (C++)

```
void Graph::breadthFirstSearch() {
    // Reset the processed states
    clearProcessedStates();

    Node<Vertex*>* start = vertices->getHead();

    while (start != nullptr) {
        // If the vertex has not been processed, it is disconnected from the rest of the graph
        if (!start->data->isProcessed) {
            // Start the BFS from the current node to explore that region of the graph
            std::cout << "BFS: ";
            breadthFirstSearch(start->data);
            std::cout << std::endl;
        }
        start = start->next;
    }
}

void Graph::breadthFirstSearch(Vertex* start) {
    // Create the queue for the vertices to check
    Queue<Vertex*> verticesToCheck;

    // Add the start to the queue
    Node<Vertex*>* startNode = new Node<Vertex*>(start);
    verticesToCheck.enqueue(startNode);
    startNode->data->isProcessed = true;

    // Continue until no more vertices are left to check
    while (!verticesToCheck.isEmpty()) {
        // Get the next vertex and print it out
        Node<Vertex*>* check = verticesToCheck.dequeue();
        std::cout << check->data->getId() << " ";

        Node<Vertex*>* cur = check->data->getNeighbors()->getHead();
        while (cur != nullptr) {
            if (!cur->data->isProcessed) {
                // Create a new node and add it to the queue
                Node<Vertex*>* newNode = new Node<Vertex*>(cur->data);
                verticesToCheck.enqueue(newNode);
                newNode->data->isProcessed = true;
            }
            cur = cur->next;
        }

        delete check;
    }
}
```

Listing 3: Graph Breadth First Search (C++)