CMPT 435L ALGORITHMS
Assignment 3

November 15th 2024

Jonathan "Grant" Bever

Link to github repo

PART 1: Undirected Graphs and how we can represent them

Undirected Graphs represent a way we can present data but unlike stacks, queues, and hashmaps from assignments prior it is not a data structure but we can apply the Matrix and Adjacency list data structures to undirected graphs.

Format / Representation

Vertices:
[1, 2, 3, 4, 5, 6, 7]

Edges:
[1, 2], [1, 5], [1, 6], [2, 3], [2, 5], [2, 6], [3, 4], [4, 5], [5, 6], [5, 7], [6, 7]

Matrix Representation:

$$\begin{bmatrix} 0 & 1 & 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 \end{bmatrix}$$

Adjacency List Representation:

- 1: 2, 5, 6
- 2: 1, 3, 5, 6
- 3: 2, 4
- 4: 3, 5
- 5: 1, 2, 4, 6, 7
- 6: 1, 2, 5, 7
- 7: 5, 6

Code: Graph Conversion Functions

Function: toMatrix

```
export function toMatrix(graph: { vertices: number[]; edges: [number, number][] }) {
    const vertices = graph.vertices;
    const vertexIndexMap = new Map<number, number>();
    vertices.forEach((vertex, index) => vertexIndexMap.set(vertex, index));
    const size = vertices.length;
    const matrix = Array.from({ length: size }, () => Array(size).fill(0));

    graph.edges.forEach(([v1, v2]) => {
      const i = vertexIndexMap.get(v1);
      const j = vertexIndexMap.get(v2);
      matrix[i][j] = 1;
      matrix[j][i] = 1;
      }
    });

    return matrix;
}
```

Function: toAdjList

```
export function toAdjList(graph: Graph): AdjList {
    const adjList: AdjList = {};
    graph.vertices.forEach(vertex => (adjList[vertex] = []));
    graph.edges.forEach(([v1, v2]) => {
      adjList[v1].push(v2);
      adjList[v2].push(v1);
    });

    return adjList;
}
```

Now that we have these data structures coded we can perform some algorithms on them.

Traversals

There are multiple ways to traverse a graph but the two ways we are going to are Breadth-first and Depth-first.

As the name implies Breadth First traversals start at the root and work their way through each layer while Depth-first traversals start at the bottom and work their way up from there. This is how I coded it.

Depth First Traversal:

```
export function dfs(graph: AdjList, start: number, visited: Set<number> = new Set()){
    console.log(start);
    visited.add(start);
// Visit all the unvisited neighbors
    for (const neighbor of graph[start]){
        if (!visited.has(neighbor)) {
            dfs(graph, neighbor, visited);
        }
    }
}
```

Breadth First Traversal:

```
export function bfs(graph: AdjList, start: number){
    const visited = new Set<number>();
    const queue: number[] = [start];
    visited.add(start);
    while (queue.length > 0){
        const vertex = queue.shift()!;
        console.log(vertex);
// Visit all unvisited neighbors
        for (const neighbor of graph[vertex]){
            if (!visited.has(neighbor)){
                visited.add(neighbor);
                queue.push(neighbor);
            }
        }
    }
}
```

The asymptotic running time of these traversals is $\mathbf{O}(\boldsymbol{V} + \boldsymbol{E})$ Where V is equal to the number of vertices and E is equal to the number of edges. This is because BFS and DFS traverse each edge and vertex, ensuring each vertex is processed once and each edge is explored once.

Binary Search Trees

Binary Search Trees are a type of graph and a data structure. BSTs can only have two child leaf nodes and are the child of a leaf node unless they are the root, the left child is less then the parent while the right is more so if my root is 6, my children are 4 and 8 then 4 would be on the left and 8 on the right. If I added 5 it would be the right child of 4 since its less than 6 but more than 4.

```
class LeafNode {
    value: string;
    left: LeafNode | null = null;
    right: LeafNode | null = null;
    constructor(value: string) {
        this.value = value;
    }
}
export class BinarySearchTree {
    root: LeafNode | null = null;
    // Insert a new node in the BST
    insert(value: string): void {
        if (this.root === null) {
            this.root = new LeafNode(value);
            console.log(`Root: ${value}`);  //Print root item
        } else {
            this._insert(this.root, value, "");
        }
    }
    private _insert(node: LeafNode, value: string, path: string):
        void {
        if (value < node.value) {
            if (node.left === null) {
                node.left = new LeafNode(value);
                console.log(`${value}: ${path}L`);
            } else {
                this._insert(node.left, value, `${path}L, `);
            }
        } else {
            if (node.right === null) {
                node.right = new LeafNode(value);
                console.log(`${value}: ${path}R`);
            } else {
                this._insert(node.right, value, `${path}R, `);
            }
        } } }
```

Now that we have our BST we can perform an In-Order Traversal

```
//In-order traversal of the BST (this is where we print the values)
    inOrderTraversal(node: LeafNode | null = this.root): void {
        if (node !== null) {
            this.inOrderTraversal(node.left);
            console.log(node.value); //Print the value of the node
            this.inOrderTraversal(node.right);
        }
    }
```

When we get the output of this function you may notice that it looks a lot like a sort and that's because in many ways it is. Hence the name "In-Order Traversal" the function traverses the tree in order and prints the value along the way.

The asymptotic running time for this algorithm is $\mathbf{O}(\boldsymbol{n})$ where n = the number of nodes, making this algorithm linear. This makes a lot of sense because we visit every single node once, going from one to the next, performing constant time operations. This is also one of the rare algorithms where there isn't a worst scenario that changes our running time since no matter what we visit each node once.

We can also perform searches very effectively, using this simple lookup algorithm...

```
 // Find an item in the BST
find(value: string): { path: string; comparisons: number } {
return this._find(this.root, value, "", 0);
}
private _find(node: LeafNode | null, value: string, path: string, comparisons: number): { path: string; comparisons: number } {
    comparisons++;
    if (value === node.value) {
        return { path, comparisons };
    } else if (value < node.value) {
        return this._find(node.left, value, path + "L, ", comparisons);
    } else {
        return this._find(node.right, value, path + "R, ", comparisons);
    }
}
```

The inherent traits of Binary Search Trees make this simple algorithm very effective.

searching for 42 out of 666 items I had an average comparison count of 11.95. To refresh our memory from assignment two, searching for 42 random items out of 666 with binary search I had 8.49 average comparisons and while 11.95 is higher than that it still is not bad at all, it certainly is better than linear search's 328 comparisons.

The asymptotic running time of a search on a Binary Search Trees are $\mathbf{O(\log n)}$. Assuming you are working with a fairly balanced tree at each step you reduce the search space by half, similar to binary search.