

Algorithms

Karl Solomon

December 31, 2024

Contents

I	Data Structures	2
1	Queue/FIFO	2
2	Stack/LIFO	2
3	Heap/Priority Queue	2
4	Tree/Graph	2
4.1	Binary Tree	2
4.2	Graphs	4
5	Lists	8
5.1	Singly Linked List	8
5.2	Doubly Linked List	8
II	Types of Algorithms	8
6	Greedy	8
7	DP	8
8	DFS	8
9	BFS	8
10	Sorting	8
11	Network Flow	8
11.1	Dijkstra's	8

Part I

Data Structures

1 Queue/FIFO

2 Stack/LIFO

3 Heap/Priority Queue

4 Tree/Graph

4.1 Binary Tree

- Check if two BST are same

```
1 bool isSame(Node* root1, Node* root2) {
2     if (!root1 && !root2) return true;
3     if (!root1 || !root2) return false;
4     return root1->data == root2->data && isSame(root1->left, root2
5         ->left) && isSame(root1->right, root2->right);
6 }
```

- Insert Element

```
1 void insertNode(Node* root, int data) {
2     if (root == nullptr) {
3         root = new Node(data);
4     } else {
5         if (data < root->data) {
6             insertNode(root->left, data);
7         } else {
8             insertNode(root->right, data);
9         }
10    }
11 }
```

- Delete Element

```
1 Node* deleteBSTNode(Node* root, int data) {
2     if (root == nullptr) return root;
3     if (root->data == data) {
4         // Only 1 node (root)
5         if (root->left == nullptr && root->right == nullptr) {
6             delete root;
7             root = nullptr;
8         }
9         // Only 1 child (right)
10        else if (root->left == nullptr) {
11            Node* temp = root->right;
12            delete root;
13            root = temp;
14        }
15        // Only 1 child (left)
16        else if (root->right == nullptr) {
17            Node* temp = root->left;
18            delete root;
19            root = temp;
20        }
21        // Both children exist. Take either min from right sub-
22        // tree (implemented here) or max from left sub-tree and
23        // set
```

```

22         // to deleted node's position
23     else {
24         Node* temp = root->right;
25         while (temp->left != nullptr) {
26             temp = temp->left;
27         }
28         root->data = temp->data;
29         root->right = deleteBSTNode(root->right, temp->data);
30     }
31 }
32 return root;
33 }

```

- Lowest Common Ancestor

```

1 Node* lowestCommonAncestor(Node* root, int data1, int data2) {
2     // Base case
3     if (root == nullptr) {
4         return nullptr;
5     }
6
7     // If both data1 and data2 are smaller than root, then LCA
8     // lies in left subtree
9     if (data1 < root->data && data2 < root->data) {
10        return lowestCommonAncestor(root->left, data1, data2);
11    }
12
13    // If both data1 and data2 are greater than root, then LCA
14    // lies in right subtree
15    if (data1 > root->data && data2 > root->data) {
16        return lowestCommonAncestor(root->right, data1, data2);
17    }
18
19    // If one data is on the left and the other is on the right,
20    // then root is the LCA
21    return root;
22 }

```

- Determine if tree is BST

```

1 bool isBST(Node* root, int min, int max) {
2     if (root == nullptr) return true;
3     if (root->data < min || root->data > max) return false;
4     return isBST(root->left, min, root->data - 1) && isBST(root->
5         right, root->data + 1, max);
6 }
7
8 bool isBST(Node* root) {
9     bool res = true;
10    if (root == nullptr) {
11        return res;
12    }
13    if (root->left == nullptr && root->right == nullptr) {
14        return res;
15    }
16    return isBST(root, INT_MIN, INT_MAX);
17 }

```

- Traversal

- Level-Order

```

1  /*
2  * - We use a 'std::queue<Node*>' to keep track of nodes to
    visit.
3  * - We start by pushing the root node into the queue.
4  * - While the queue is not empty, we:
5  *   - Dequeue the front node, process it by adding its data
    to the result queue.
6  *   - Enqueue its left and right children if they exist.
7  * - This process continues until all nodes have been visited,
    resulting in a level-order traversal of the BST.
8  */
9
10 std::queue<int> traverseBSTLevelOrder(Node* root) {
11     std::queue<int> result;
12     if (root == nullptr) {
13         return result;
14     }
15
16     std::queue<Node*> nodeQueue;
17     nodeQueue.push(root);
18
19     while (!nodeQueue.empty()) {
20         Node* current = nodeQueue.front();
21         nodeQueue.pop();
22         result.push(current->data);
23
24         if (current->left != nullptr) {
25             nodeQueue.push(current->left);
26         }
27         if (current->right != nullptr) {
28             nodeQueue.push(current->right);
29         }
30     }
31
32     return result;
33 }

```

- In-order: Left subtree, then node, then right subtree

```

1  std::vector<int> bstTraverseInOrder(Node* root) {
2      std::vector<int> res;
3      if (root == nullptr) return res;
4      res = bstTraverseInOrder(root->left);
5      res.push_back(root->data);
6      std::vector<int> right = bstTraverseInOrder(root->right);
7      res.insert(res.end(), right.begin(), right.end());
8  }

```

- PreOrder: Node, then left subtree, then right subtree

- PostOrder: Left subtree, then right subtree, then Node

4.2 Graphs

Storing Graphs:

- Adjacency Matrix: 2D array $N \times N$. $[n, m] = \text{nonZeroNum}$ means there is an edge from node n to node m . All nodes should not include edges to self. Edges can be weighted (to suggest things like cost/distance). In directional-graphs you can have $[n, m] = \text{nonZero}$, but $[m, n] = 0$. Cons: $O(N^2)$ memory. This is especially bad for sparse graphs.

- Adjacency List: Store edges as a list of neighbor nodes. Each node contains a list of neighbors. It is possible to represent weights here with a list of `std::pair<Node*, int>`.

Graph Algorithms:

- Bellman-Ford

```

1  #include <iostream>
2  #include <limits>
3  #include <vector>
4  /*
5   * - We define a 'Graph' class with a list of 'Edge' structures,
6     *   each containing a source, destination, and weight.
7   * - The 'bellmanFord' function initializes distances and
8     *   relaxes all edges |V| - 1 times, where |V| is the number of
9   *   vertices.
10  * - It checks for negative-weight cycles by attempting one more
11  *   relaxation.
12  * - If a negative-weight cycle is detected, it prints a message
13  *   .
14  * - Otherwise, it prints the shortest distances from the start
15  *   vertex to all other vertices.
16  */
17 struct Edge {
18     int source, destination, weight;
19 };
20
21 class Graph {
22 public:
23     int vertices;
24     std::vector<Edge> edges;
25
26     Graph(int v) : vertices(v) {}
27
28     void addEdge(int source, int destination, int weight) { edges.
29         push_back({source, destination, weight}); }
30     // The Bellman-Ford algorithm, is used to find the shortest
31     // paths from a single source vertex to all other vertices
32     // in a graph. The algorithm can handle graphs with negative
33     // weight edges:
34     void bellmanFord(int start) {
35         std::vector<int> distance(vertices, std::numeric_limits<
36             int>::max());
37         distance[start] = 0;
38
39         // Relax edges |V| - 1 times
40         for (int i = 0; i < vertices - 1; ++i) {
41             for (const auto& edge : edges) {
42                 if (distance[edge.source] != std::numeric_limits<
43                     int>::max() &&
44                     distance[edge.source] + edge.weight < distance
45                     [edge.destination]) {
46                     distance[edge.destination] = distance[edge.
47                         source] + edge.weight;
48                 }
49             }
50         }
51
52         // Check for negative-weight cycles
53         for (const auto& edge : edges) {
54             if (distance[edge.source] != std::numeric_limits<int
55                 >::max() &&

```

```

44         distance[edge.source] + edge.weight < distance[
           edge.destination]) {
45             std::cout << "Graph contains a negative-weight
           cycle" << std::endl;
46             return;
47         }
48     }
49
50     // Print the distances
51     for (int i = 0; i < vertices; ++i) {
52         std::cout << "Distance from vertex " << start << " to
           vertex " << i << " is " << distance[i] << std::
           endl;
53     }
54 }
55 };
56
57 int main() {
58     Graph g(5);
59     g.addEdge(0, 1, -1);
60     g.addEdge(0, 2, 4);
61     g.addEdge(1, 2, 3);
62     g.addEdge(1, 3, 2);
63     g.addEdge(1, 4, 2);
64     g.addEdge(3, 2, 5);
65     g.addEdge(3, 1, 1);
66     g.addEdge(4, 3, -3);
67
68     g.bellmanFord(0);
69
70     return 0;
71 }

```

• Dijkstra's

```

1  #include <iostream>
2  #include <limits>
3  #include <queue>
4  #include <vector>
5
6  using namespace std;
7
8  typedef pair<int, int> Edge; // (weight, vertex)
9  /*
10   * In this program:
11   *
12   * - We define a 'Graph' class with an adjacency list to store
           edges as pairs of (weight, vertex).
13   * - The 'dijkstra' function initializes distances and uses a
           priority queue to explore the shortest paths.
14   * - It updates the shortest path estimates and pushes them into
           the priority queue.
15   * - Finally, it prints the shortest distances from the start
           vertex to all other vertices.
16   *
17   */
18
19 class Graph {
20 public:
21     int vertices;
22     vector<vector<Edge>> adjList;
23
24     Graph(int v) : vertices(v), adjList(v) {}

```

```

25
26 void addEdge(int source, int destination, int weight) {
27     adjList[source].push_back(make_pair(weight, destination));
28     adjList[destination].push_back(make_pair(weight, source));
29     // For undirected graph
30 }
31 // Dijkstra's algorithm, is used to find the shortest paths
32 // from a single source vertex to all other vertices
33 // in a graph with non-negative edge weights:
34 void dijkstra(int start) {
35     vector<int> distance(vertices, numeric_limits<int>::max())
36     ;
37     priority_queue<Edge, vector<Edge>, greater<Edge>> pq;
38
39     distance[start] = 0;
40     pq.push(make_pair(0, start));
41
42     while (!pq.empty()) {
43         int currentVertex = pq.top().second;
44         pq.pop();
45
46         for (const auto& edge : adjList[currentVertex]) {
47             int weight = edge.first;
48             int neighbor = edge.second;
49
50             if (distance[currentVertex] + weight < distance[
51                 neighbor]) {
52                 distance[neighbor] = distance[currentVertex] +
53                     weight;
54                 pq.push(make_pair(distance[neighbor], neighbor
55                     ));
56             }
57         }
58     }
59
60     // Print the distances
61     for (int i = 0; i < vertices; ++i) {
62         cout << "Distance from vertex " << start << " to
63             vertex " << i << " is " << distance[i] << endl;
64     }
65 }
66 };
67
68 int main() {
69     Graph g(5);
70     g.addEdge(0, 1, 10);
71     g.addEdge(0, 4, 5);
72     g.addEdge(1, 2, 1);
73     g.addEdge(1, 4, 2);
74     g.addEdge(2, 3, 4);
75     g.addEdge(3, 4, 9);
76     g.addEdge(3, 0, 7);
77
78     g.dijkstra(0);
79
80     return 0;
81 }

```

- Ford-Fulkerson

- Kruskals's

- Nearest neighbor

- Prim's

- DFS

- BFS

5 Lists

5.1 Singly Linked List

5.2 Doubly Linked List

Part II

Types of Algorithms

6 Greedy

7 DP

8 DFS

9 BFS

10 Sorting

11 Network Flow

11.1 Dijkstra's