CSE200A : Competitive Programming I
(Summer 2023)
Reading 3

FooBar: Competitive Programming Club of IIITD

June 2023

1 Traversals in Graphs

1.1 DFS (Depth First Search)

Depth-first search (DFS) is a straightforward graph traversal technique. The algo- rithm begins at a starting node and proceeds to all other nodes that are reachable from the starting node using the edges of the graph. Depth-first search always follows a single path in the graph as long as it finds new nodes. After this, it returns to previous nodes and begins to explore other parts of the graph. The algorithm keeps track of visited nodes, so that it processes each node only once.

```
// DFS Algorithm
void dfs(int curr, vector<vector<int>>& adj_list, vector<bool>&
   visited){
    visited[curr] = true;
    for(auto nxt : adj_list[curr]){
        if(!visited[nxt]){
            dfs(nxt, adj_list, visited);
        }
    }
    return;
}
```

Resource

1.2 BFS (Breadth First Search)

Breadth-first search (BFS) visits the nodes of a graph in increasing order of their distance from the starting node. Thus, we can calculate the distance from the starting node to all other nodes using breadth-first search. However, breadth-first search is more difficult to implement than depth-first search. Breadth-first search goes through the nodes one level

after another. First the search explores the nodes whose distance from the starting node is 1, then the nodes whose distance is 2, and so on. This process continues until all nodes have been visited.

```cpp
// BFS Algorithm
void bfs(int source, vector<vector<int>>& adj_list){
    int n = adj_list.size();
    queue<int> q;
    vector<bool> visited(n, false);

    visited[source] = true;
    q.push(source);
    while(!q.empty()){
        int curr = q.front();
        q.pop();
        for(auto nxt : adj_list[curr]){
            if(!visited[nxt]){
                visited[nxt] = true;
                q.push(nxt);
            }
        }
    }
    return;
}
```

Resource

2 Shortest Path Algorithms in Graphs

Graphs are fundamental data structures used to model various real-world scenarios, such as social networks, transportation systems, and computer networks. One common problem in graph theory is finding the shortest path between two vertices in a graph.

2.1 Dijkstra's Algorithm

Dijkstra's algorithm is used for finding the shortest path in a graph with non-negative edge weights.

2.1.1 Description

Dijkstra's algorithm starts from a source vertex and iteratively selects the vertex with the minimum distance, updating the distances of its adjacent vertices.

2.1.2 Approach

It starts from a source vertex and iteratively selects the vertex with the minimum distance, updating the distances of its adjacent vertices.

2.1.3 Complexity

Dijkstra's algorithm has a time complexity of $O((V + E)logV)$, where V is the number of vertices and E is the number of edges.

2.1.4 Code Implementation

```cpp
// Dijkstra's Algorithm
typedef pair<int, int> pii;

vector<int> dijkstra(vector<vector<pii>>& graph, int source) {
    vector<int> distances(graph.size(), numeric_limits<int>::max
       ());
    distances[source] = 0;

    priority_queue<pii, vector<pii>, greater<pii>> pq;
    pq.push({0, source});

    while (!pq.empty()) {
        int current_distance = pq.top().first;
        int current_vertex = pq.top().second;
        pq.pop();

        if (current_distance > distances[current_vertex]) {
            continue;
        }

        for (pii neighbor : graph[current_vertex]) {
            int neighbor_vertex = neighbor.first;
            int weight = neighbor.second;

            int distance = current_distance + weight;

            if (distance < distances[neighbor_vertex]) {
                distances[neighbor_vertex] = distance;
                pq.push({distance, neighbor_vertex});
            }
        }
    }

    return distances;
```

```
}
```

2.2 Bellman-Ford Algorithm

The Bellman-Ford algorithm handles graphs with negative edge weights and can also detect negative cycles.

2.2.1 Description

The Bellman-Ford algorithm performs a relaxation process by iteratively relaxing the edges, updating the shortest distance until no further updates are possible.

2.2.2 Approach

It performs a relaxation process by iteratively relaxing the edges, updating the shortest distance until no further updates are possible.

2.2.3 Complexity

The Bellman-Ford algorithm has a time complexity of O(V * E), where V is the number of vertices and E is the number of edges.

2.2.4 Code Implementation

```cpp
// Bellman-Ford Algorithm
struct Edge {
    int source;
    int destination;
    int weight;
};

vector<int> bellmanFord(vector<Edge>& edges, int num_vertices,
   int source) {
    vector<int> distances(num_vertices,INF);
    distances[source] = 0;

    for (int i = 0; i < num_vertices - 1; i++) {
        for (const Edge& edge : edges) {
            int u = edge.source;
            int v = edge.destination;
            int weight = edge.weight;

```

```
            if (distances[u] != INF && distances[u] + weight <
                distances[v]) {
                distances[v] = distances[u] + weight;
            }
        }
    }

    for (const Edge& edge : edges) {
        int u = edge.source;
        int v = edge.destination;
        int weight = edge.weight;

        if (distances[u] != INF && distances[u] + weight <
            distances[v]) {
            throw runtime_error("Graph contains a negative-weight
                cycle");
        }
    }

    return distances;
}
```

2.3 Floyd-Warshall Algorithm

The Floyd-Warshall algorithm solves the all-pairs shortest path problem, finding the shortest paths between all pairs of vertices in a graph.

2.3.1 Description

The Floyd-Warshall algorithm builds a matrix of shortest distances by considering all intermediate vertices and systematically updating the distances.

2.3.2 Approach

It builds a matrix of shortest distances by considering all intermediate vertices and systematically updating the distances.

2.3.3 Complexity

The Floyd-Warshall algorithm has a time complexity of $O(V^3)$, where V is the number of vertices.

2.3.4 Code Implementation

```cpp
// Floyd-Warshall Algorithm
vector<vector<int>> floydWarshall(const vector<vector<int>>&
    graph) {
     int num_vertices = graph.size();
     //first initialise dist[][] with given edge weights


     for (int k = 0; k < num_vertices; k++) {
         for (int i = 0; i < num_vertices; i++) {
             for (int j = 0; j < num_vertices; j++) {
                 dist[i][j] = min(dist[i][j], dist[i][k] + dist[k
                     ][j]);
             }
         }
     }

     return dist;
}
```

2.4 Limitations of Shortest Path Algorithms

2.4.1 Dijkstra's Algorithm

Dijkstra's algorithm has the following limitations:

1. It cannot handle graphs with negative edge weights. It assumes that all edge weights are non-negative.
2. It may not produce correct results if the graph contains cycles with negative total weights.
3. Dijkstra's algorithm only works for connected graphs, where there is a path between every pair of vertices. It cannot handle disconnected graphs where some vertices are unreachable from others.

2.4.2 Bellman-Ford Algorithm

The Bellman-Ford algorithm has the following limitations:

1. It can handle graphs with negative edge weights, but it is less efficient than Dijkstra's algorithm for graphs without negative weights.
2. The algorithm has a time complexity of $O(V \cdot E)$, where V is the number of vertices and E is the number of edges. This makes it slower than Dijkstra's algorithm for dense graphs.
3. If the graph contains a negative-weight cycle reachable from the source vertex, the algorithm will enter an infinite loop. Additionally, it cannot handle disconnected graphs,

as it requires a path from the source to all other vertices.

2.4.3 Floyd-Warshall Algorithm

The Floyd-Warshall algorithm has the following limitations:

1. The algorithm has a high time complexity of $O(V^3)$, where V is the number of vertices. It becomes inefficient for large graphs.
2. It requires storing a matrix of size $V \times V$, which may consume a significant amount of memory for large graphs.
3. The algorithm assumes that there are no negative-weight cycles in the graph. If there is a negative-weight cycle, the algorithm may produce incorrect results or go into an infinite loop. Additionally, the Floyd-Warshall algorithm can handle both connected and disconnected graphs, as it calculates the shortest paths between all pairs of vertices.

3 Topological sorting

Topological sorting is a common algorithm used in graph theory to order the nodes of a directed acyclic graph (DAG) in such a way that for every directed edge (u, v), node u comes before node v in the sorted order. This sorting order is useful for tasks that have dependencies, where each task must be completed before its dependent tasks can be executed.

3.1 Algorithm

The algorithm for topological sorting can be implemented using either Depth-First Search (DFS) or Breadth-First Search (BFS) techniques. Here, we will cover both approaches.

3.1.1 Depth-First Search (DFS) Approach

```
1  // Graph representation using adjacency list
2  class Graph {
3      int V;
4      list<int> *adj;
5
6  public:
7      Graph(int V);
8      void addEdge(int u, int v);
9      void topologicalSortUtil(int v, bool visited[], stack<int>&
           result);
10     void topologicalSort();
11 };
12
13 Graph::Graph(int V) {
14     this->V = V;
15     adj = new list<int>[V];
```

```
}

void Graph::addEdge(int u, int v) {
    adj[u].push_back(v);
}

void Graph::topologicalSortUtil(int v, bool visited[], stack<int
   >& result) {
    visited[v] = true;

    list<int>::iterator i;
    for (i = adj[v].begin(); i != adj[v].end(); ++i)
        if (!visited[*i])
            topologicalSortUtil(*i, visited, result);

    result.push(v);
}

void Graph::topologicalSort() {
    stack<int> result;
    bool* visited = new bool[V];
    for (int i = 0; i < V; i++)
        visited[i] = false;

    for (int i = 0; i < V; i++)
        if (!visited[i])
            topologicalSortUtil(i, visited, result);

    while (!result.empty()) {
        cout << result.top() << " ";
        result.pop();
    }
}
```

3.1.2 Breadth-First Search (BFS) Approach

```
// Graph representation using adjacency list
class Graph {
    int V;
    list<int> *adj;

public:
    Graph(int V);
    void addEdge(int u, int v);
    vector<int> topologicalSort();
};

```

```cpp
Graph::Graph(int V) {
    this->V = V;
    adj = new list<int>[V];
}

void Graph::addEdge(int u, int v) {
    adj[u].push_back(v);
}

vector<int> Graph::topologicalSort() {
    vector<int> result;

    // Compute in-degree for each node
    vector<int> inDegree(V, 0);
    for (int u = 0; u < V; ++u) {
        for (auto v : adj[u])
            inDegree[v]++;
    }

    // Queue to store nodes with in-degree 0
    queue<int> q;
    for (int u = 0; u < V; ++u) {
        if (inDegree[u] == 0)
            q.push(u);
    }

    while (!q.empty()) {
        int u = q.front();
        q.pop();
        result.push_back(u);

        // Decrease in-degree of neighbors
        for (auto v : adj[u]) {
            inDegree[v]--;
            if (inDegree[v] == 0)
                q.push(v);
        }
    }

    return result;
}
```

3.2 Complexity Analysis

The time complexity of the topological sort algorithms is $O(V + E)$, where V is the number of vertices and E is the number of edges in the graph.

4 Trees

4.1 Introduction

A tree is a simple connected acyclic graph. This means, it has only 1 connected component, and has no cycle. The following terminologies occur frequently when talking about trees

- Leaf: A vertex in a tree having degree 1.
- Root: In some cases, a particular vertex of the tree is labelled as the root. It is often visualised as the "topmost" node of the tree.
- Ancestor/Descendent: Ancestor of a vertex u is the vertex v such that there exists a path from the root to v to u in the tree. In this case, u is the descendent of v.
- Parent/Child: If vertex v is the ancestor of u and there is an edge between u and v, v is known as the parent of u. Likewise, u is the child of v. In a rooted tree, every vertex other than the root has a parent. Every vertex, other than the leaves, have children.
- Forest: A simple acyclic graph. Note that these need not be connected. Forests are often thought of as a collection of 1 or more trees.

4.2 Difference from other graphs

Note that trees are a subset of graphs. The following conditions are satisfied by all trees:

1. A tree having n vertices always has exactly $n - 1$ edges.
2. A tree always has 1 connected component.
3. A tree has no cycles.

An interesting observation is, if any 2 of the above conditions are satisfied, the third is automatically satisfied as well. Can you see why?

4.3 Euler tour

A Eulerian path is a path in a graph that passes through all of its edges exactly once. A Eulerian cycle is a Eulerian path that is a cycle. Using an Euler tour we can save the order in which points are visited in a Tree, then operations like 'checking if a is ancestor of b' becomes an O(1) operation.

Resource

4.4 Binary lifting

Binary lifting is a pre-processing algorithm, used to reduce certain tasks from linear to logarithmic time complexity. This algorithm has a strong resemblance with modular exponentiation.

Resource

4.5 Practice problems

Search a node in Binary Tree

Second Largest element in n-ary tree

How to determine if a binary tree is height-balanced?

Find maximum (or minimum) in Binary Tree

Replace every node with depth in N-ary Generic Tree

Lowest Common Ancestor of a Binary Tree

5 DP on trees

5.1 Introduction

Dynamic Programming is a technique to solve problems by breaking them down into overlapping sub-problems which follow the optimal substructure.

In DP problems, we define a certain function/subproblem on certain values/states of the problem. On trees, these are defined on the nodes of trees, and are calculated recursively based on the children of the tree. DP on trees is performed on rooted trees. In case a root is not defined, we specify an appropriate node as the root before solving the problem.

5.2 Example

Problem: Given a tree T of N nodes, where each node i has C_i coins attached to it. You have to choose a subset of nodes such that no two adjacent nodes (i.e., nodes connected directly by an edge) are chosen, and the sum of coins attached to the chosen subset is maximum.

Solution: For convenience, let us assume the nodes are numbered 1 to n, and root 1 is the root node.

Subproblem Definition $dp[u][i]$ represents the maximum value of chosen coins for the subtree having root u, where $i = 1$ denotes that node u has been selected, and $i = 2$ denotes that node u has been left out.

Recurrence

$$dp[u][1] = C_u + \sum_{v=\text{ child of } u} dp[v][2]$$

$$dp[u][2] = \sum_{v=\text{ child of } u} dp[v][1]$$

Solution subproblem

The solution is given by $\max(dp[1][1], dp[1][2])$.

For a detailed explanation of the above solution and more solved examples, you may refer to the resource below.

Resource

5.3 Practice problems

Tree House(Easy-med)

Bakry and Partitioning(Easy-Medium)

Valid Paths(Medium)

Nauuo and Circle(Medium)

King Killing(Hard)