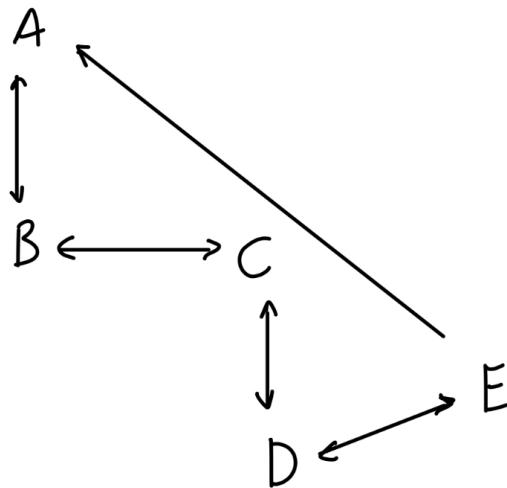


# Week 10 — Graphs

Anh Huynh

## 1.

Create a theoretical graph using a pen and paper OR electronically.



Edges are undirected except for  $E \rightarrow A$ , which is directed.

## 2.

Implement the graph created in step 1 and apply breadth and depth-first search algorithms using C++.

```
1 #include <iostream>
2 #include <vector>
3 #include <string>
4 #include <queue>
5 using namespace std;
6
7 // breadth-first search
8 void bfs(const vector<vector<int>>& adj, int start, const vector<string>& names){
9     vector<bool> visited(adj.size(), false); // for keeping track of which vertices are visited
10    queue<int> q; // queue for bfs
11
12    visited[start] = true; // mark start vertex as visited
13    q.push(start); // push start vertex in queue
14
15    cout << "BFS order: ";
16    while(!q.empty()){ // run until the queue is empty
17        int u = q.front(); // get next vertex from queue
```

```

18     q.pop();
19     cout << names[u] << " "; // print vertex name
20
21     for (int v : adj[u]){ // check all neighbors of current vertex
22         if (!visited[v]){ // if neighbor is not visited yet
23             visited[v] = true; // mark as visited
24             q.push(v); // add it to queue
25         }
26     }
27 }
28 cout << endl; // move to the next line after bfs order
29 }
30
31 // recursive depth-first search
32 void dfs(const vector<vector<int>>& adj, int u, vector<bool>& visited, const vector<string>&
33 names){
34     visited[u] = true; // mark current vertex as visited
35     cout << names[u] << " "; // print current vertex
36
37     for (int v : adj[u]){ // loop through all connected vertices
38         if (!visited[v]){ // if neighbor not visited
39             dfs(adj, v, visited, names); // visit neighbor through recursive function call
40         }
41     }
42 }
43
44 int main(){
45     vector<string> pt = {"A", "B", "C", "D", "E"}; // vertex labels
46     vector<vector<int>> adj(5); // adjacency list for the 5 vertices
47
48     // A <-> B
49     adj[0].push_back(1); // A -> B
50     adj[1].push_back(0); // B -> A
51
52     // B <-> C
53     adj[1].push_back(2); // B -> C
54     adj[2].push_back(1); // C -> B
55
56     // C <-> D
57     adj[2].push_back(3); // C -> D
58     adj[3].push_back(2); // D -> C
59
60     // D <-> E
61     adj[3].push_back(4); // D -> E
62     adj[4].push_back(3); // E -> D
63
64     // E -> A
65     adj[4].push_back(0); // E -> A
66
67     // call bfs from A (index 0)
68     bfs(adj, 0, pt);

```

```
68
69 // make visited array for dfs
70 vector<bool> visited(adj.size(), false);
71 cout << "DFS: ";
72 dfs(adj, 0, visited, pt); // run dfs starting from A
73 cout << endl;
74
75 return 0; // end
76 }
```

### 3.

**Compare both search algorithms in the context of Big O notations.**

Let  $V$  = number of vertices and  $E$  = number of edges.

Both BFS and DFS have the time complexity of  $O(V + E)$  since each vertex and edge is processed at most once, and a space complexity of  $O(V)$ . BFS uses a queue to visit nodes level by level while DFS uses recursion or a stack to explore as deep as possible before backtracking.

BFS or DFS — visits every vertex once ( $O(V)$ ) and looks at every edge once ( $O(E)$ ) so that gives  $O(V+E)$  time regardless of whether the graph is small, big, directed or undirected.

Video Link: <https://youtu.be/Eb4T4ItDKhM>