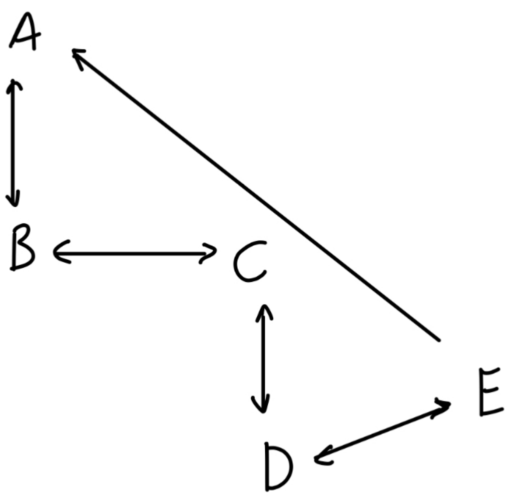
Week 10 — Graphs

Anh Huynh

1.

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



Edges are undirected except for E —> A, which is directed.

2.

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

```
#include <iostream>
#include <vector>
#include <string>
#include <queue>
using namespace std;

// breadth-first search
void bfs(const vector<vector<int>>& adj, int start, const vector<string>& names){
    vector<bool> visited(adj.size(), false); // for keeping track of which vertices are visited
    queue<int> q; // queue for bfs

    visited[start] = true; // mark start vertex as visited
    q.push(start); // push start vertex in queue

    cout << "BFS order: ";
    while(!q.empty()){ // run until the queue is empty
        int u = q.front(); // get next vertex from queue
```

```
        q.pop();
        cout << names[u] << " "; // print vertex name

        for (int v : adj[u]){ // check all neighbors of current vertex
            if (!visited[v]){   // if neighbor is not visited yet
                visited[v] = true; // mark as visited
                q.push(v); // add it to queue
            }
        }
    }
    cout << endl; // move to the next line after bfs order
}

// recursive depth-first search
void dfs(const vector<vector<int>>& adj, int u, vector<bool>& visited, const vector<string>& names){
    visited[u] = true; // mark current vertex as visited
    cout << names[u] << " "; // print current vertex

    for (int v : adj[u]){ // loop through all connected vertices
        if (!visited[v]){   // if neighbor not visited
            dfs(adj, v, visited, names); // visit neighbor through recursive function call
        }
    }
}

int main(){
    vector<string> pt = {"A", "B", "C", "D", "E"}; // vertex labels
    vector<vector<int>> adj(5); // adjacency list for the 5 vertices

    // A <-> B
    adj[0].push_back(1); // A -> B
    adj[1].push_back(0); // B -> A

    // B <-> C
    adj[1].push_back(2); // B -> C
    adj[2].push_back(1); // C -> B

    // C <-> D
    adj[2].push_back(3); // C -> D
    adj[3].push_back(2); // D -> C

    // D <-> E
    adj[3].push_back(4); // D -> E
    adj[4].push_back(3); // E -> D

    // E -> A
    adj[4].push_back(0); // E -> A

    // call bfs from A (index 0)
    bfs(adj, 0, pt);
```

```

    // make visited array for dfs
    vector<bool> visited(adj.size(), false);
    cout << "DFS: ";
    dfs(adj, 0, visited, pt); // run dfs starting from A
    cout << endl;

    return 0; // end
}
```

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: